Éléments d'algorithmique

D. Beauquier, J. Berstel, Ph. Chrétienne

6 février 2005

Première édition par Masson, 1992.

Version 6 février 2005

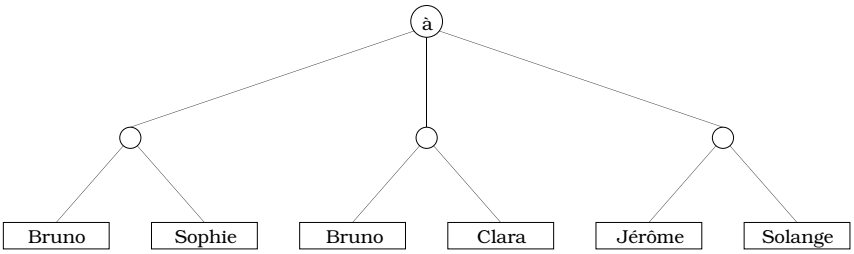

à
Bruno
Sophie
Bruno
Clara
Jérôme
Solange

Table des matières

Liste des figures

Automates

Motifs

Géométrie algorithmique

Planification de trajectoires

Avant-propos

«Encore un livre d'algorithmique!» Les ouvrages consacrés à l'algorithmique et aux structures de données paraissent, depuis quelque temps, à un rythme soutenu et régulier. Aussi devons-nous expliquer pourquoi un livre supplémentaire sur ce sujet nous a paru utile.

Le présent livre se distingue d'autres traités d'algorithmique par deux aspects : d'une part, un accent particulier est mis sur les nouvelles structures d'arbres apparues ces dernières années (arbres bicolores, arbres persistants) ; d'autre part, nous développons plus en détail trois applications de l'algorithmique : l'optimisation combinatoire, la recherche de motifs dans un texte, et la géométrie algorithmique. Outre leur intérêt propre et leur importance intrinsèque, ces trois applications illustrent de façon exemplaire l'usage que l'on peut faire de structures de données sophistiquées, et les gains en temps et en place qui résultent de leur emploi judicieux.

Pour chacun de ces trois thèmes, nous mettons en place les bases nécessaires, tant algorithmiques que théoriques. Puis, nous présentons, à l'aide de problèmes typiques, un échantillon des algorithmes les plus efficaces, employant des structures de données intéressantes. Parmi les algorithmes et les structures de données qui méritent d'être mentionnés plus spécialement, citons :

- le coût amorti des opérations de rééquilibrage dans les arbres 2–4 (chapitre 6) ;
- les algorithmes de manipulation des arbres bicolores (*ibid.*) ;
- les arbres persistants (*ibid.*) ;
- les algorithmes de Goldberg–Tarjan et d'Ahuja–Orlin pour les flots (chapitre 8) ;
- l'algorithme de Hopcroft de minimisation d'un automate fini (chapitre 9) ;
- les algorithmes de Simon, et de Boyer et Moore pour la recherche de motifs dans un texte (chapitre 10) ;
- la gestion dynamique de l'enveloppe convexe d'un ensemble fini de points (chapitre 11) ;
- l'algorithme de balayage de calcul d'une subdivision planaire à l'aide d'arbres persistants (*ibid.*) ;
- l'algorithme de Fortune de calcul du diagramme de Voronoï (*ibid.*) ;
- deux algorithmes de planification de trajectoires (chapitre 12).

A qui s'adresse ce livre ?

Ce livre est issu de cours donnés par les auteurs en licence et maîtrise d'informatique et de mathématiques dans les universités Paris VI et Paris VII, en magistère, et pour partie dans divers DEA et DESS.

Dans sa forme présente, il s'adresse principalement aux étudiants en licence et en maîtrise d'informatique et de mathématiques, et aux étudiants des écoles d'ingénieurs. Nous ne supposons chez le lecteur qu'une connaissance rudimentaire de l'informatique. Les premiers chapitres décrivent les bases algorithmiques sur lesquelles s'appuieront les chapitres suivants. Le niveau d'exposition est, à de rares exceptions près, celui des étudiants en licence d'informatique.

Organisation du livre

Ce livre s'organise en deux parties. La première partie, constituée des chapitres 1 à 5, contient un exposé des bases algorithmiques et mathématiques, avec notamment un chapitre sur les techniques d'évaluation de coûts d'algorithmes, un bref chapitre sur les structures de données usuelles, un chapitre plus long sur la théorie des graphes et un chapitre assez succinct sur les algorithmes de tri par comparaison. Les trois premiers chapitres, et en partie les deux chapitres suivants, peuvent être considérés comme des chapitres de référence, auxquels on se reportera si nécessaire.

La deuxième partie, consacrée à des thèmes plus avancés, débute par un long chapitre sur les arbres de recherche. Les deux chapitres suivants sont consacrés à l'optimisation combinatoire. Puis viennent deux chapitres sur la recherche de motifs dans un texte, et enfin deux chapitres sur la géométrie algorithmique et la planification de trajectoires.

Présentation des algorithmes

Les algorithmes sont présentés dans un langage proche de Pascal ; assez rarement, et principalement dans le chapitre 3 (Structures de données), des références explicites à Pascal sont faites, avec des programmes complets, prêts à cuire. Nous nous efforçons de présenter les algorithmes avec suffisamment de détails pour que l'écriture de programmes soit facile ; en particulier, nous décrivons soigneusement les structures de données à employer.

Contenu du livre

Le premier chapitre contient la définition des diverses mesures du coût d'un algorithme : coût dans le cas le plus défavorable, coût moyen, coût amorti ; il se termine par une borne inférieure sur la complexité du tri par comparaison.

La première section du deuxième chapitre contient la définition des notations dites de Landau, ainsi que des exemples de manipulation. Dans la deuxième section, nous abordons la résolution de divers types d'équations de récurrence auxquelles conduit l'analyse d'un algorithme. Nous décrivons notamment la résolution des récurrences linéaires et des récurrences de partition.

Dans le chapitre 3, nous passons en revue les structures de données élémentaires, à savoir les piles, files, listes, ainsi que les arbres binaires et les arbres binaires de recherche. Nous décrivons ensuite les files de priorité et leur implémentation au moyen de tas. Nous terminons par un algorithme de gestion de partitions.

Le chapitre 4 est consacré aux bases de la théorie des graphes. Après les définitions, nous exposons l'algorithme de Roy-Warshall et d'autres problèmes d'accessibilité. Les parcours classiques d'un graphe (parcours en profondeur, en largeur) sont décrits. Puis, nous présentons en détail l'algorithme de Tarjan de calcul des composantes fortement connexes d'un graphe, qui est linéaire en fonction du nombre d'arcs du graphe.

Le chapitre 5 présente trois algorithmes de tri interne éprouvés et efficaces : le tri rapide, le tri fusion et le tri par tas. Nous en donnons l'analyse des coûts en moyenne et dans le pire des cas. Le chapitre se termine par un algorithme de tri externe, le tri polyphasé qui est fondé sur les suites de Fibonacci.

Le chapitre 6 contient la description de plusieurs familles d'arbres binaires de recherche équilibrés. Nous décrivons les arbres AVL, puis les arbres a–b, avec notamment des opérations plus élaborées comme la concaténation et la scission, et évaluons le coût amorti d'une suite d'opérations. Ensuite, nous présentons les arbres bicolores, et enfin une réalisation de structures persistantes sur ces arbres.

Le chapitre 7 traite deux problèmes fondamentaux d'optimisation combinatoire : la recherche d'un arbre couvrant de coût minimum et la recherche des chemins de coût minimum issus d'un sommet dans un graphe orienté valué. Pour le premier problème, nous présentons une implémentation efficace des algorithmes de Kruskal et de Prim. Quant à la recherche des chemins de coût minimum, nous exposons l'itération fondamentale de Ford, l'algorithme de Dijkstra pour des coûts positifs et sa variante $A*$, l'algorithme de Bellman pour un graphe sans circuits et enfin l'algorithme Paps.

Le chapitre 8 traite des problèmes de flots optimaux dans les graphes. Nous prouvons d'abord le théorème d'Hoffman sur l'existence d'un flot compatible, puis étudions les décompositions d'un flot. Après avoir établi le théorème de Ford et Fulkerson, nous analysons les algorithmes les plus performants pour le calcul du flot maximum, à savoir l'algorithme primal des distances estimées au puits (algorithme d'Ahuja et Orlin) et deux variantes efficaces de l'algorithme dual du préflot, l'algorithme de Karzanov et l'algorithme des excès échelonnés. Nous considérons ensuite le problème du flot de coût minimum et présentons un algorithme dû à Goldberg et Tarjan. Nous terminons par l'algorithme d'Edmonds et Karp pour la recherche d'un plan de transport de coût minimum.

Dans le chapitre 9, préliminaire au chapitre suivant, nous présentons les bases de la théorie des automates finis. Après avoir démontré l'équivalence entre les automates finis et les automates finis déterministes, nous établissons le théorème de Kleene qui montre que les langages reconnaissables et les langages rationnels sont une seule et même famille de langages. Nous prouvons l'existence et l'unicité

d'un automate déterministe minimal reconnaissant un langage donné. Pour la construction de cet automate, nous présentons l'algorithme de Hopcroft de minimisation.

Dans le chapitre 10, nous considérons d'abord le problème de la recherche d'une ou de toutes les occurrences d'un mot dans un texte. Nous présentons l'algorithme naïf, l'algorithme de Morris et Pratt, et sa variante due à Knuth, Morris et Pratt, l'implémentation par automate fini et l'algorithme de Simon, et pour finir l'algorithme de Boyer et Moore, dans sa version de Horspool et dans la version complète. Ensuite, nous considérons la recherche d'une occurrence de plusieurs motifs, et décrivons l'algorithme de Aho et Corasick. Dans la dernière section, nous étudions la recherche d'occurrences de mots décrits par une expression rationnelle.

La première section du chapitre 11 contient un rappel de quelques notations classiques en géométrie euclidienne, ainsi que la mise en place d'outils qui seront utiles dans les sections suivantes. La deuxième section est consacrée aux algorithmes de calcul de l'enveloppe convexe d'un ensemble fini de points du plan. Nous terminons par un algorithme plus sophistiqué de gestion dynamique d'enveloppes convexes. Quelques problèmes de localisation d'un point dans le plan divisé en régions sont étudiés dans la troisième section ; on y utilise des structures de données assez complexes, en particulier les ensembles ordonnés persistants. La dernière section est consacrée aux diagrammes de Voronoï de points et de segments.

Dans le chapitre 12, nous étudions un problème de planification de trajectoires, aussi appelé le problème du « déménageur de piano ». Nous donnons des algorithmes performants dans deux cas particuliers typiques : l'algorithme de translation d'un segment dans un environnement polygonal dû à Schwartz et Sharir, et celui du déplacement d'un disque dû à Ó'Dúnlaing et Yap. Ce dernier utilise les diagrammes de Voronoï de segments.

Exercices et notes

Les chapitres sont en général suivis de notes et d'exercices. Les notes bibliographiques sont volontairement succinctes. Leur but n'est pas de retracer la paternité des résultats ou des algorithmes présentés, mais de conseiller des lectures complémentaires. On y trouvera notamment des renvois nombreux au *Handbook of Theoretical Computer Science* qui contient, lui, d'abondantes listes de références bibliographiques.

Remerciements

Nous avons bénéficié, durant la préparation de ce livre, de commentaires, discussions et remarques de nombreux collègues. Nous remercions en particulier Paul Blanchard, Luc Boasson, Maxime Crochemore, Clara Daquin, Christiane Frougny, Marie-Paule Gascuel, Irène Guessarian, Michelle Morcrette, Dominique Perrin, Michel Pocchiola, Andreas Podelski, Imre Simon, Michèle Soria, Volker Strehl, Anne Verroust, Mariette Yvinec.

Chapitre 1

Préliminaires

Dans ce chapitre préliminaire, nous donnons d'abord une classification des problèmes du point de vue de leur complexité algorithmique, nous discutons ensuite les mesures de complexité, puis nous donnons une borne inférieure sur la complexité du tri par comparaison.

1.1 Les algorithmes et leur coût

1.1.1 Algorithmes

Un *algorithme* est un ensemble d'opérations de calcul élémentaires, organisé selon des règles précises dans le but de résoudre un problème donné. Pour chaque donnée du problème, l'algorithme retourne une réponse après un nombre fini d'opérations. Les *opérations élémentaires* sont par exemple les opérations arithmétiques usuelles, les transferts de données, les comparaisons entre données, etc. Selon le niveau d'abstraction où l'on se place, les opérations arithmétiques et les objets sur lesquels elles portent peuvent être plus ou moins compliquées. Il peut s'agir simplement d'additionner des entiers naturels, ou de multiplier des polynômes, ou encore de calculer les valeurs propres d'une matrice. Pour un système de calcul formel, il s'agit là d'opérations élémentaires, parce qu'elles ont été programmées et sont disponibles dans des bibliothèques, plus ou moins transparentes à l'utilisateur; dans un langage comme Pascal, ces opérations ne sont pas disponibles.

Il apparaît utile de ne considérer comme véritablement élémentaires que les opérations dont le *temps de calcul* est constant, c'est-à-dire ne dépend pas de la *taille* des opérandes. Par exemple, l'addition d'entiers de taille bornée *a priori* (les `integer` en Pascal) est une opération élémentaire; l'addition d'entiers de taille quelconque ne l'est pas. De même, le test d'appartenance d'un élément à un ensemble n'est pas une opération élémentaire en ce sens, parce que son

temps d'exécution dépend de la taille de l'ensemble, et ceci même si dans certains langages de programmation, il existe des instructions de base qui permettent de réaliser cette opération.

L'*organisation* des calculs, dans un algorithme, est souvent appelée sa *structure de contrôle*. Cette structure détermine l'ordre dans lequel il convient de tester des conditions, de répéter des opérations, de recommencer tout ou partie des calculs sur un sous-ensemble de données, etc. Là aussi, des conventions existent sur la façon de mesurer le coût des opérations; la richesse des structures de contrôle dépend fortement des langages de programmation. Le langage Pascal est peut-être l'un des plus pauvres dans ce domaine parmi les langages modernes. En général, le coût des structures de contrôle peut être négligé, parce qu'il est pris en compte, asymptotiquement, par les opérations élémentaires. En fait, le surcoût provenant par exemple d'une programmation récursive n'est vraiment sensible que lorsqu'elle entraîne des recopies massives de données.

1.1.2 Problèmes intraitables

Contrairement à une idée largement répandue, tout problème ne peut être résolu par un algorithme. Ce n'est pas une question de taille des données, mais une impossibilité fondamentale. On doit la preuve de ce fait aux logiciens des années 30 et 40. Pour en donner une démonstration rigoureuse, il convient évidemment de formuler très précisément la notion d'algorithme. Il apparaît que les possibilités des ordinateurs et des langages de programmation sont parfaitement couvertes par la définition mathématique de ce qui est résoluble algorithmiquement.

On est donc en présence d'une première dichotomie, entre problèmes *insolubles* algorithmiquement et les autres. Parmi les problèmes qui sont algorithmiquement solubles, on peut encore distinguer une hiérarchie en fonction de la *complexité* des algorithmes, complexité mesurée en temps de calcul, c'est-à-dire en nombre d'opérations élémentaires, et exprimée en fonction de la taille du problème. La *taille* elle-même est un paramètre qui mesure le nombre de caractères nécessaires pour décrire une donnée du problème. Pour une matrice carrée à coefficients bornés par exemple, son *ordre* est un bon paramètre de la taille; pour un polynôme, le *degré* peut être une indication de la taille, sauf si l'on sait par exemple que le polynôme n'a que très peu de coefficients non nuls, auquel cas le nombre de coefficients non nuls est un meilleur indicateur. Le temps de calcul d'un algorithme croît en général en fonction de la taille des données, et la vitesse de la croissance est une mesure de la complexité du problème. Une croissance *exponentielle* ou plus rend un problème intraitable pour des données de grande taille. Même une croissance polynomiale, disons comme une puissance cinquième de la taille des données, rend la résolution pratiquement impossible pour des données d'une certaine taille.

Evidemment, il existe de très nombreux problèmes pour lesquels tout algorithme a une croissance au moins exponentielle. Dans cette catégorie, on trouve les algo-

rithmes dont le résultat lui-même est constitué d'un nombre exponentiel ou plus de données. Par exemple, la génération de toutes les permutations de n éléments prend un temps plus qu'exponentiel, simplement parce qu'il y a $n!$ permutations. Il existe toute une classe de problèmes, qui sont des problèmes d'optimisation combinatoire, où l'on cherche une solution optimale parmi un nombre exponentiel de solutions réalisables. Une bonne stratégie, si elle existe, permet de trouver une solution optimale sans énumérer l'ensemble des solutions réalisables; on peut alors résoudre le problème en temps disons polynomial. Bien entendu, on ne connaît pas toujours une telle stratégie; mais on ne sait pas à l'heure actuelle si, pour tout problème de ce type, il existe un algorithme polynomial. La conjecture la plus problable est qu'un très grand nombre de problèmes de ce type ne puissent pas être résolus par un algorithme polynomial. On aurait ainsi une deuxième division des problèmes, entre ceux qui sont solubles, mais intraitables, et les autres, pour lesquels il existe des algorithmes efficaces. L'un des objectifs de l'algorithmique est l'étude de ces problèmes, pour lesquels on cherche les algorithmes les plus efficaces, tant du point de vue du temps de calcul que des besoins en place.

1.1.3 Sur la présentation d'algorithmes

Pour présenter les algorithmes de manière formelle, nous utilisons dans ce livre un langage proche de Pascal. Toutefois, en vue de ne pas alourdir inutilement la lecture, nous n'en suivons pas strictement la syntaxe; de plus, nous introduisons quelques ellipses, et quelques variantes sur les structures de contrôle. La plupart parlent d'elles-mêmes.

Ainsi, pour économiser l'écriture de parenthèses (début,fin), nous utilisons les mots finsi, finpour, et fintantque comme délimiteurs de portée, lorsqu'un tel délimiteur est utile. Nous écrivons donc :

si *test* alors *suite d'instructions* sinon *suite d'instructions* finsi
tantque *test* faire *suite d'instructions* fintantque
pour $\cdots$ faire *suite d'instructions* finpour

D'autres variantes concernent la structure de contrôle. Ainsi, retourner, et plus rarement exit provoque (comme en C) la sortie de la procédure ou fonction courante, et le retour vers la procédure appelante.

Enfin, nous utilisons largement la version séquentielle des opérateurs booléens et et ou (comme en Modula), baptisés etalors et oualors. Dans l'évaluation d'une expression de la forme a etalors b, on évalue d'abord a, et on n'évalue b que si a est vrai. Le même mécanisme vaut *mutatis mutandis* pour oualors. On économise ainsi beaucoup de programmation confuse.

1.2 Mesures du coût

Considérons un problème donné, et un algorithme pour le résoudre. Sur une donnée x de taille n, l'algorithme requiert un certain temps, mesuré en nombre d'opérations élémentaires, soit $c(x)$. Le coût en temps varie évidemment avec la taille de la donnée, mais peut aussi varier sur les différentes données de même taille n. Par exemple, considérons l'algorithme de tri qui, partant d'une suite $(a_1, \ldots, a_n)$ de nombres réels distincts à trier en ordre croissant, cherche la première descente, c'est-à-dire le plus petit entier i tel que $a_i > a_{i+1}$, échange ces deux éléments, et recommence sur la suite obtenue. Si l'on compte le nombre d'*inversions* ainsi réalisées, il varie de 0 pour une suite triée à $n(n-1)/2$ pour une suite décroissante. Notre but est d'évaluer le coût d'un algorithme, selon certains critères, et en fonction de la taille n des données.

1.2.1 Coût dans le cas le plus défavorable

Le coût $C(n)$ d'un algorithme dans le cas *le plus défavorable*ou dans le cas *le pire* («worst-case» en anglais) est par définition le maximum des coûts, sur toutes les données de taille n :

$$C(n) = \max_{|x|=n} c(x)$$

(on note $|x|$ la taille de x.) Dans l'algorithme ci-dessus, ce coût est $n(n-1)/2$. Cette mesure du coût est réaliste parce qu'elle prend en compte toutes les possibilités.

1.2.2 Coût moyen

Dans des situations où l'on pense que le cas le plus défavorable ne se présente que rarement, on est plutôt intéressé par le coût moyen de l'algorithme. Une formulation correcte de ce coût moyen suppose que l'on connaisse une *distribution de probabilités* sur les données de taille n. Si $p(x)$ est la probabilité de la donnée x, le *coût moyen* $\gamma(n)$ d'un algorithme sur les données de taille n est par définition

$$\gamma(n) = \sum_{|x|=n} p(x)c(x)$$

Le plus souvent, on suppose que la distribution est uniforme, c'est-à-dire que $p(x) = 1/T(n)$, où $T(n)$ est le nombre de données de taille n. Alors, l'expression du coût moyen prend la forme

$$\gamma(n) = \frac{1}{T(n)} \sum_{|x|=n} c(x) \tag{2.1}$$

Continuons notre exemple du tri par transposition. Le nombre de données de taille n, c'est-à-dire de suites de n nombres réels distincts, est infini. Or, seul l'ordre relatif des éléments d'un suite intervient dans l'algorithme. Une donnée de taille n peut donc être assimilée à une permutation de l'ensemble $\{1, \ldots, n\}$. On a alors $T(n) = n!$. Le coût $c(x)$ d'une permutation x est le nombre de transpositions d'éléments adjacents nécessaires pour transformer x en la permutation identique; ce nombre est le *nombre d'inversions* de x, qui est par définition le nombre $b_1 + \cdots + b_n$, où b_j est le nombre d'entiers $1 \leq i < j$ tels que $x_i > x_j$. On peut montrer que le nombre moyen d'inversions est $n(n-1)/4$, de sorte que le coût moyen de l'algorithme, pour la distribution uniforme, est $n(n-1)/4$. Comme le montre cet exemple, l'évaluation du coût moyen est en général bien plus compliquée que l'évaluation du coût dans le cas le plus défavorable.

Considérons un autre exemple, à savoir la génération de toutes les parties d'un ensemble E à n éléments, disons de $\{1, \ldots, n\}$. Chaque partie X est représentée par son vecteur caractéristique x, avec

$$x[i] = \begin{cases} 1 & \text{si } i \in X \\ 0 & \text{sinon} \end{cases}$$

Concrètement, on utilise un type `suite` pour représenter ces vecteurs. Le calcul du premier sous-ensemble, la partie vide de E, revient à initialiser la suite x à $(0, \ldots, 0)$. Les parties suivantes s'obtiennent en procédant de droite à gauche dans le vecteur courant x. On remplace tous les 1 par 0 jusqu'à rencontrer un 0. Celui-ci est remplacé par 1 (c'est très exactement l'algorithme d'incrémentation, en binaire). Voici une réalisation :

```
PROCEDURE Suivante (VAR x:suite; n:integer; VAR d:boolean);
   VAR
      i: integer;
   BEGIN
      i := n;
      WHILE (i > 0) AND (x[i] = 1) DO BEGIN
         x[i] := 0; i := i - 1
      END;
      d := i = 0; IF NOT d THEN x[i] := 1
   END;
```

Dans la boucle `WHILE`, l'opérateur `AND` est séquentiel. La variable booléenne `d` repère si la suite donnée en argument représente la «dernière» partie de l'ensemble E, à savoir E lui-même.

Analysons le coût d'un appel de la procédure `Suivante`, sur le tableau x, mesuré par le nombre de comparaisons $x[i] = 1$. Le coût est n lorsque les $x[i]$ valent 1 pour $1 \leq i \leq n$, c'est-à-dire lorsque l'on a atteint la dernière partie. Le coût est aussi n lorsque $x[1] = 0$ et les autres éléments de x valent 1. Quel est le coût moyen de la procédure, sur tous les appels? Pour l'évaluer, nous reprenons l'équation (2.1)

et l'écrivons sous la forme

$$\gamma(n) = \frac{1}{T(n)} \sum_{i=1}^{n} i c_i$$

où c_i est le nombre de données x pour lequelles le coût est i, c'est-à-dire telles que $c(x) = i$. Pour $1 \leq i < n$, le nombre de comparaisons est i, si x se termine par un élément 0 suivi par $i-1$ éléments égaux à 1. Le nombre de suites de cette forme est 2^{n-i}. Enfin, on a $c_n = 2$, de sorte que

$$\gamma(n) = \frac{1}{2^n} \left(2n + \sum_{i=1}^{n-1} i 2^{n-i} \right)$$

Le nombre entre parenthèses est égal à $2^{n+1} - 2$, et on a

$$\gamma(n) = \frac{2^{n+1} - 2}{2^n}$$

donc le coût moyen du calcul de la partie suivante est approximativement 2.

Comme ces exemples l'indiquent, l'estimation du coût moyen d'un algorithme est en général plus délicate que l'estimation du coût dans le cas le plus défavorable, et nécessite une bonne dose de connaissances combinatoires sur les objets traités.

1.2.3 Coût amorti

Lorsqu'une *suite* d'opérations est effectuée sur une structure, le coût total est égal à la somme des coûts des opérations individuelles. Connaissant des estimations du coût (dans le cas le plus défavorable) des opérations individuelles, on obtient une estimation du coût total en sommant ces majorations du coût des opérations individuelles. Or très souvent, cette majoration du coût total qui, à chaque étape, procède par une évaluation pessimiste, est trop grossière. On observe en effet que le cas le plus défavorable «ne se répète pas». Plus précisément, dans de nombreux algorithmes, le cas le plus défavorable provient d'un déséquilibre exceptionnel qui ne peut pas se produire plusieurs fois de suite, car le traitement de ce cas défavorable «rétablit» l'équilibre. Il y a alors un phénomène de compensation entre opérations consécutives, d'amortissement des coûts, que nous explicitons dans cette section. Commençons par un exemple.

Exemple. Considérons une *pile* (voir chapitre 3), et définissons une *opération* comme étant une suite (éventuellement vide) de dépilements suivie d'un empilement (ce genre d'opérations se rencontre par exemple en analyse syntaxique). Le *coût* d'une opération o est $c(o) = 1 + d$, où d est le nombre de dépilements précédant l'empilement.

Partant de la pile vide P_0, on effectue une suite $o_1, \ldots, o_n$ d'opérations :

$$P_0 \xrightarrow{o_1} P_1 \longrightarrow \ldots \xrightarrow{o_n} P_n$$

où P_k est la pile après la k-ième opération, et l'on veut estimer le coût total

$$c_n = c(o_1) + \cdots + c(o_n)$$

Une méthode simple consiste à majorer le coût de chaque opération o_k par k. En effet, dans le cas le plus défavorable, l'opération o_k vide complètement la pile P_{k-1} avant de procéder à l'empilement, et P_{k-1} contient au plus $k-1$ éléments. On a donc $c(o_k) \leq k$, d'où $c(n) = O(n^2)$.

Mais, comme déjà dit plus haut, le cas le plus défavorable (ici : la pile P_{k-1} contient $k-1$ éléments) ne se produit pas deux fois de suite. En d'autres termes, notre analyse n'a pas pris en compte une contrainte globale qui, dans le cas d'une pile, exprime que l'on ne peut pas dépiler des éléments avant de les avoir empilés. Plus précisément, notons d_k le nombre de dépilements de la k-ième opération, de sorte que

$$c(o_k) = 1 + d_k$$

Alors $c_n = n + d$ avec $d = d_1 + \cdots + d_n$. Or n est le nombre total d'empilements et d est le nombre total de dépilements. Comme on ne peut dépiler plus que l'on a empilé, on a $d \leq n$ et donc $c_n \leq 2n$. Cet argument s'explicite par ce que l'on nomme la *technique du potentiel* : à chaque pile P_k, on associe un nombre h_k (son «potentiel») qui est toujours positif ou nul; le coût de l'opération o_k s'exprime à l'aide d'une variation du potentiel. Dans notre cas, prenons pour h_k la hauteur de la pile P_k. On a $h_0 = 0$,

$$h_k = h_{k-1} - d_k + 1 \qquad (k \geq 1)$$

et bien sûr $h_k \geq 0$ pour $k = 1, \ldots, n$. Comme $d_k = h_{k-1} - h_k + 1$, le coût de l'opération o_k est

$$c(o_k) = 2 + h_{k-1} - h_k$$

Il en résulte bien entendu que $c_n = c(o_1) + \cdots + c(o_n) = 2n + h_0 - h_n = 2n - h_n \leq 2n$. On appelle *coût amorti* de l'opération o (relativement au potentiel h) la valeur

$$a(o) = c(o) + h'' - h'$$

où h' est le potentiel de la pile avant l'opération, et h'' est le potentiel de la pile après l'opération o. Dans notre cas, le coût amorti est égal à 2. On obtient

$$a(o_1) + \cdots + a(o_n) = c(o_1) + \cdots + c(o_n) + h_n - h_0$$

et comme $h_n \geq 0$, $h_0 = 0$, on a

$$c(o_1) + \cdots + c(o_n) \leq a(o_1) + \cdots + a(o_n)$$

donc le coût amorti de la suite des opérations est une majoration du coût total.

On peut voir ce procédé de comptage d'une autre manière, plus ludique. Supposons que l'ordinateur qui doit exécuter la suite d'opérations travaille avec des

jetons, comme une vulgaire machine à sous («coin-operated computer», disent les Américains). Chaque fois que l'on insère un jeton, l'ordinateur est disposé à travailler pendant une durée fixe, puis il s'arrête et attend le jeton suivant. Si une opération se termine avant épuisement du temps acheté, ce temps de calcul est disponible pour l'opération suivante. Le coût total d'une suite d'opérations est alors proportionnel au nombre de jetons qu'il faut introduire, et l'avoir est égal à la valeur du potentiel. Le coût amorti correspond au nombre de jetons à introduire par opération (ici 2) pour pouvoir effectuer les opérations.

Plus formellement, on considère une famille de «structures» sujettes à des opérations qui les transforment. De plus, on suppose défini un *potentiel*, c'est-à-dire une fonction h qui à chaque structure associe un nombre réel positif ou nul. En particulier, on demande que le potentiel de la structure vide soit nul.

Par définition, le *coût amorti* d'une opération o qui, appliquée à la structure S, donne la structure $S' = o(S)$, relativement au potentiel h, est

$$a(o) = c(o) + h(S') - h(S)$$

Le coût amorti d'une suite d'opérations $o_1, \ldots, o_n$ est la somme des coûts amortis des opérations individuelles.

Proposition 2.1. *Soit h un potentiel, et soit $o_1, \ldots, o_n$ une suite d'opérations appliquée à une structure de potentiel nul. La somme des coûts de la suite d'opérations est majorée par son coût amorti relativement à h :*

$$c(o_1) + \cdots + c(o_n) \leq a(o_1) + \cdots + a(o_n)$$

Preuve. Soit S_0 la structure de départ initialement vide, et soit S_k la structure obtenue après la k-ième opération :

$$S_0 \xrightarrow{o_1} S_1 \xrightarrow{o_2} \ldots \xrightarrow{o_n} S_n$$

On a $a(o_k) = c(o_k) + h(S_k) - h(S_{k-1})$. Comme les valeurs du potentiel s'éliminent deux à deux, on obtient

$$a(o_1) + \cdots + a(o_n) = c(o_1) + \cdots c(o_n) + h(S_n) - h(S_0)$$

Or $h(S_0) = 0$ et $h(S_n) \geq 0$, ce qui établit l'inégalité. ∎

Dans la pratique, on essaie de trouver, en utilisant son intuition, un potentiel qui fournit une majoration aisée du coût amorti des opérations, ce qui donne ensuite une majoration du coût total.

Exemple. Revenons sur l'exemple de la génération des parties d'un ensemble à n éléments. Notons o_k le k-ième appel de la procédure `Suivante`. Le coût total c_N du calcul des N premiers sous-ensembles est $c_N = c(o_1) + \cdots c(o_N)$. Définissons un potentiel h sur le tableau x par : $h(x) =$ le nombre d'éléments égaux à 1 dans

x. Le coût de l'opération o_k qui transforme le $k-1$-ième tableau $x^{(k-1)}$ en le k-ième tableau $x^{(k)}$ est alors

$$c(o_k) = 2 + h(x^{(k-1)}) - h(x^{(k)})$$

et le coût amorti est

$$a(o_k) = c(o_k) + h(x^{(k)}) - h(x^{(k-1)}) = 2$$

La somme des coûts amortis est $2N$, et en vertu de la proposition précédente, ceci majore le coût total. Ce résultat est d'une autre nature que l'évaluation en moyenne donnée plus haut, dans la mesure où il ne fait appel à aucune hypothèse probabiliste.

1.3 Une borne inférieure

Après avoir trouvé et analysé un algorithme qui résout un problème donné, il est naturel de se demander si cet algorithme est le meilleur possible. Il existe plusieurs façons d'améliorer, parfois substantiellement, un algorithme, ou un programme, en organisant mieux les données, en économisant des calculs d'indices, en regroupant certaines opérations. Ces optimisations peuvent prendre en compte les caractéristiques d'un langage de programmation spécifique, voire les particularités d'un compilateur. La question de l'optimalité d'un algorithme, vue indépendamment d'une implémentation particulière, se pose différemment : on se demande s'il existe un algorithme qui nécessite moins d'opérations élémentaires. Pour cela, on estime le nombre d'opérations requises par *tout* algorithme qui résout le problème donné. Là encore, la formulation du problème et de la réponse se fait en fonction de la taille des données. On cherche le nombre d'opérations nécessaires pour résoudre le problème dans le cas le plus défavorable pour les données de taille n.

Pour apporter une réponse à cette question, il faut d'abord définir précisément la classe d'algorithmes et les opérations élémentaires. Nous décrivons ci-dessous le modèle des *arbres de décision*qui permet de prouver que tout algorithme de tri qui opère par comparaisons nécessite au moins $\lceil n \ln n \rceil - n$ comparaisons pour trier des suites de n éléments. Cette borne inférieure vaut pour le cas le plus défavorable : pour tout algorithme, et pour tout n, il existe une suite dont le tri exige au moins $\lceil n \ln n \rceil - n$ comparaisons.

Dans un algorithme de tri par comparaisons, on utilise uniquement des comparaisons pour obtenir des informations sur la suite $(a_1, \ldots, a_n)$ de données. En d'autres termes, étant donnés a_i et a_j, on effectue l'un des test $a_i < a_j$, $a_i \leq a_j$, $a_i = a_j$ pour déterminer leur position relative. Aucune autre information n'est accessible sur ces éléments. Pour prouver le résultat annoncé, il suffit de se restreindre aux suites dont les éléments sont distincts; nous pouvons donc nous contenter du test $a_i \leq a_j$.

Le comportement d'un algorithme de tri, sur des suites de longueur n, peut être représenté par un arbre binaire (voir chapitre 4), appelé arbre de décision. Dans un arbre de décision, chaque nœud est étiqueté par une comparaison, notée $a_i : a_j$, pour des entiers $1 \leq i, j \leq n$, et chaque feuille est étiquetée par une permutation $(\sigma(1), \ldots, \sigma(n))$ (voir figure 3.1).

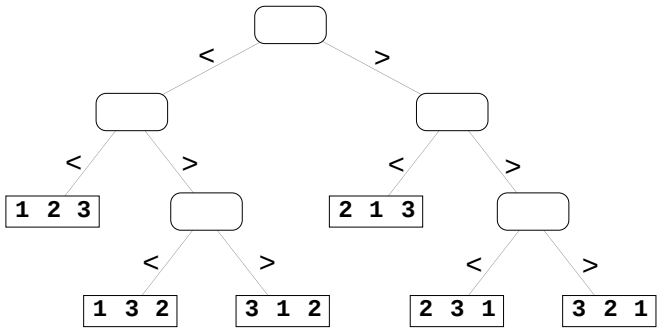


Figure 3.1: *L'arbre de décision d'un algorithme de tri.*

L'exécution de l'algorithme de tri sur une suite $(a_1, \ldots, a_n)$ correspond à un chemin dans l'arbre, menant de la racine à une feuille. A chaque nœud, une comparaison $a_i < a_j$ est faite, et le résultat de la comparaison indique si les comparaisons suivantes se font dans le sous-arbre gauche ou le sous-arbre droit. Lorsque une feuille est atteinte, l'algorithme de tri a établi que l'ordre est $a_{\sigma(1)} < a_{\sigma(2)} < \cdots < a_{\sigma(n)}$. Chacune des $n!$ permutations doit apparaître comme étiquette d'une feuille, pour que l'algorithme puisse proprement séparer tous les cas de figure.

La longueur du chemin de la racine à une feuille donne le nombre de comparaisons effectuées, et dans le cas le plus défavorable, le nombre de comparaisons nécessaires est donc égal à la hauteur de l'arbre. Une minoration de la hauteur d'un arbre de décision fournit donc une borne inférieure sur le nombre de comparaisons.

Lemme 3.1. *Soit A un arbre binaire. Si A a $n!$ feuilles, alors la hauteur de A est au moins $\lceil n \ln n \rceil - n$.*

Preuve. Soit h la hauteur de A. Alors $n! \leq 2^h$, donc $\ln n! \leq h \ln 2 \leq h$. Or $\ln n! \geq n \ln n - n$ (voir chapitre 2), d'où le résultat. ∎

Corollaire 3.2. *Tout algorithme de tri par comparaisons nécessite, dans le cas le plus défavorable, au moins $\lceil n \ln n \rceil - n$ comparaisons pour trier des suites de longueur n.* ∎

Il est important de noter que ce résultat ne porte que sur une famille bien précise d'algorithmes de tri, à savoir les tris par comparaisons. Il existe d'autres techniques de tris qui peuvent s'appliquer lorsque l'on a des informations sur la nature

des clés. Le tri par champs par exemple (« bucket sort » en anglais) est un tri en temps linéaire qui s'emploie lorsque les clés sont des chaînes de caractères.

La borne inférieure sur les algorithmes de tri est « générique » en ce sens que, pour de nombreux problèmes apparemment complètement différents, on obtient également une borne inférieure en $n \ln n$, en se ramenant au problème du tri. Il en est ainsi par exemple pour le calcul de l'enveloppe convexe d'un ensemble de points dans le plan (chapitre 11, section 2). La démarche est la suivante : on considère un algorithme A pour le problème donné, et on montre que cet algorithme peut aussi être utilisé, parfois après quelques transformations, comme algorithme de tri par comparaisons. En vue du corollaire ci-dessus, le nombre d'opérations de l'algorithme A est minoré. Comme l'argument vaut pour n'importe quel algorithme A, on obtient la borne inférieure recherchée.

Chapitre 2

Evaluations

La première section de ce chapitre contient la définition des notations dites de Landau, à savoir O, Ω, et θ, ainsi que des exemples de manipulation. Dans la deuxième section, nous abordons la résolution de divers types de récurrences. En effet, l'analyse d'un algorithme conduit le plus souvent à des équations de récurrence. Dans certains cas, on peut les résoudre complètement et donner une forme close pour les fonctions qu'elles définissent; plus nombreux sont les cas où l'on doit se contenter d'une évaluation asymptotique; on ne donne alors qu'un ordre de grandeur.

Introduction

L'efficacité en temps d'un algorithme se mesure en fonction d'un paramètre, généralement la « taille » du problème. Ce n'est évidemment pas le temps physique, exprimé en millisecondes ou en heures qui importe. Ce temps dépend trop de toutes sortes de contingences matérielles, comme l'équipement dont on dispose, le langage de programmation et la version du compilateur employés, etc. On mesure le temps requis par un algorithme en comptant le nombre d'opérations élémentaires effectuées, où une opération est élémentaire lorsqu'elle prend un temps constant. Si l'on veut être très précis, on doit aller plus loin, et classer les opérations élémentaires selon leur nature, comme des incréments de compteurs, des opérations arithmétiques, des opérations logiques, etc. Il est apparu qu'une telle précision n'apportait pas d'information substantielle, et il est aujourd'hui communément admis que même le décompte exact du nombre d'opérations n'est utile que dans une phase ultime de la réalisation d'un algorithme par un programme : dans les autres situations, on se contente d'un ordre de grandeur.

2.1 Notations de Landau

On évalue l'efficacité d'un algorithme en donnant l'ordre de grandeur du nombre d'opérations qu'il effectue lorsque la taille du problème qu'il résout augmente. On parle ainsi d'algorithme linéaire, quadratique, logarithmique, etc. Les *notations de Landau* sont un moyen commode d'exprimer cet ordre de grandeur. Trois situations sont décrites par ces notations. La plus fréquente, la notation O introduite ci-dessous, donne une *majoration* de l'ordre de grandeur; la notation Ω en donne une *minoration*, et la notation θ deux bornes sur l'ordre de grandeur. La force des notations de Landau réside dans leur concision. En contre-partie, leur emploi demande de la vigilance et un certain entraînement.

2.1.1 Notation O

On considère une fonction $g : \mathbb{R} \to \mathbb{R}$. Etant donné un point $x_0 \in \mathbb{R} \cup \{-\infty, +\infty\}$, on désigne par $O(g)$ l'ensemble des fonctions f pour lesquelles il existe un voisinage$V$ de x_0 et une constante $k > 0$ tels que

$$|f(x)| \leq k\,|g(x)| \qquad (x \in V)$$

Dans le cas des fonctions définies sur $\mathbb{R}$, un *voisinage* d'un point x_0 est une partie de $\mathbb{R}$ contenant un intervalle ouvert contenant le point x_0. On peut donc, dans la définition ci-dessus, remplacer le terme «voisinage» par «intervalle ouvert». Si la fonction g ne s'annule pas, il revient au même de dire que le rapport

$$\left|\frac{f(x)}{g(x)}\right|$$

est *borné* pour $x \in V$.

Exemple. Au voisinage de 0, on a

$$x^2 \in O(x), \quad \ln(1+x) \in O(x)$$

Par commodité, on écrit x^2 à la place de «la fonction qui à x associe x^2»; nous utiliserons par la suite cette abus d'écriture. Pour l'évaluation de la complexité des algorithmes, nous nous intéressons à la comparaison de fonctions au voisinage de $+\infty$. Ce cas est couvert, dans la définition précédente, si l'on prend $x_0 = +\infty$; un voisinage est alors un ensemble contenant un intervalle (ouvert) de la forme $]a, +\infty[$. Par conséquent, on a $f \in O(g)$ au voisinage de $+\infty$ s'il existe deux nombres $k, a > 0$ tels que

$$|f(x)| \leq k\,|g(x)| \text{ pour tout } x > a$$

Exemple. Au voisinage de l'infini, on a

$$x \in O(x^2), \quad \frac{\ln x}{x} \in O(1), \quad x + 1 \in O(x)$$

Exemple. Pour tout polynôme $P(x) = a_0x^k + a_1x^{k-1} + \cdots + a_k$, on a $P(x) \in O(x^k)$ au voisinage de l'infini. En effet, pour $x \geq 1$,

$$|P(x)| \leq |a_0|\, x^k + |a_1|\, x^{k-1} + \cdots + |a_k| \leq (|a_0| + \cdots + |a_k|)\, x^k$$

Souvent, les fonctions à comparer sont elles-mêmes à valeurs positives. Dans ce cas, on peut omettre les valeurs absolues.

Lors de l'étude de fonctions mesurant les performances d'algorithmes, l'argument est en général entier (*taille* du problème) et on se place au voisinage de l'infini, ce qui signifie que l'on considère les performances de l'algorithme pour des tailles importantes du problème; dans ce cas, les mêmes considérations restent valables lorsque le domaine des fonctions est restreint à l'ensemble $\mathbb{N}$ des entiers naturels.

Une des difficultés dans la familiarisation avec ces concepts provient de la *convention de notation* (justement «de Landau») qui veut que l'on écrive

$$f = O(g), \qquad \text{ou encore} \qquad f(x) = O(g(x)) \qquad \text{au lieu de} \qquad f \in O(g)$$

De manière analogue, on écrit

$$O(f) = O(g) \qquad \text{lorsque} \qquad O(f) \subset O(g)$$

Notons tout de suite que la relation $f = O(g)$ n'implique pas nécessairement que $g = O(f)$.

Exemple. Les formules

$$x^2 + 5x = O(x^2) = O(x^3)$$

dénotent en fait les relations

$$x^2 + 5x \in O(x^2) \subset O(x^3)$$

La notation de Landau permet l'écriture usuelle de certaines opérations arithmétiques; en définissant

$$f + O(g) = \{f + h \mid h \in O(g)\}$$
$$fO(g) = \{fh \mid h \in O(g)\}$$

on a par exemple $h = f + O(g)$ si et seulement si $h - f \in O(g)$. Les formules suivantes sont faciles à vérifier et fort utiles (ici c est une constante; sans la notation de Landau, il faudrait utiliser un signe d'appartenance au lieu de l'égalité dans la première formule, et un signe d'inclusion dans les autres).

$$\begin{aligned} f &= O(f) \\ O(-f) &= O(f) \\ cO(f) &= O(f) \\ O(f) + O(f) &= O(f) \\ O(f)O(g) &= O(fg) \\ fO(g) &= O(fg) \end{aligned}$$

Si f et g sont à valeurs positives, alors

$$O(f) + O(g) = O(f + g)$$

Vérifions par exemple la dernière formule. Si $h \in O(f)+O(g)$, alors il existe deux constantes $k, k' > 0$ et $a > 0$ telles que $|h(x)| \leq kf(x) + k'g(x)$ pour $x \geq a$. Soit K le plus grand des nombres k et k'. Alors on a $|h(x)| \leq K(f(x) + g(x))$ parce que f et g sont à valeurs positives. La formule est fausse si l'on prend $g = -f$ par exemple.

Exemple. On a $2^{n+1} = O(2^n)$ mais en revanche $3^n \notin O(2^n)$ et de même $(n + 1)! \notin O(n!)$ (puisque $(n + 1)!/n!$ tend vers $+\infty$ avec n). Observons aussi que $f(n) = O(n)$ implique $f(n)^2 = O(n^2)$, mais n'implique pas que $2^{f(n)} = O(2^n)$.

2.1.2 Notations Ω et θ

Deux notations semblables à la notation O sont couramment employées pour la description de la complexité des algorithmes.On désigne par $\Omega(g)$ l'ensemble des fonctions f pour lesquelles il existe deux nombres $k, a > 0$ tels que

$$|f(x)| \geq k|g(x)| \text{ pour tout } x > a$$

En d'autres termes, on a

$$f \in \Omega(g) \iff g \in O(f)$$

Exemple. Le nombre de comparaisons de tout algorithme de tri de suites de longueur n est $\Omega(n \ln n)$, d'après le chapitre précédent.

Enfin, on désigne par $\theta(g)$ l'ensemble des fonctions f pour lesquelles il existe des nombres k_1, k_2, $a > 0$ tels que

$$k_1 |g(x)| \leq |f(x)| \leq k_2 |g(x)|$$

pour tout $x > a$. En d'autres termes, on a

$$\theta(g) = O(g) \cap \Omega(g)$$

Ceci signifie donc que f et g croissent «de façon comparable». Plus précisément, si g ne s'annule pas, alors pour tout $x > a$

$$k_1 \leq \frac{|f(x)|}{|g(x)|} \leq k_2$$

Exemple. Pour tout $a, b > 1$, on a

$$\log_a n = \theta(\log_b n)$$

puisque $\log_a n = \log_a b \log_b n$.

Cet exemple ne doit pas faire croire que si $f = \theta(g)$, alors le quotient $|f(x)/g(x)|$ tend vers une limite non nulle, c'est-à-dire $|f| \sim c|g|$ pour une constante $c > 0$ (rappelons que $f \sim g$ signifie que le quotient $f(x)/g(x)$ tend vers la limite 1). En revanche, si $|f| \sim c|g|$, alors $f = \theta(g)$.

Exemple. Soit $f(x) = x(2 + \sin x)$. On a $x \leq f(x) \leq 3x$ pour tout $x > 0$, donc $f(x) = \theta(x)$. En revanche, le quotient $f(x)/x$ ne tend pas vers une limite lorsque x tend vers $+\infty$.

2.1.3 Exemples

Suivant l'usage dans l'analyse d'algorithmes, on entendra désormais que toutes les estimations de fonctions se font au voisinage de l'infini.

Proposition 1.1. *Pour tout $k \geq 1$, on a*

$$\sum_{i=1}^{n} i^k = \theta(n^{k+1}) \tag{1.1}$$
$$\log n! = \theta(n \log n)$$
$$\sum_{i=1}^{n} \frac{1}{i} = \theta(\log n)$$

La preuve de ces formules est l'occasion d'introduire une technique éprouvée d'évaluation de sommes utilisant des intégrales. Les inégalités requises sont énoncées dans le lemme élémentaire suivant.

Lemme 1.2. *Soient $a \leq b$ deux entiers, et soit $f : [a, b] \to \mathbb{R}$ une fonction croissante et continue. Alors*

$$\sum_{i=a}^{b-1} f(i) \leq \int_a^b f(x)\, dx \leq \sum_{i=a+1}^{b} f(i) \qquad \blacksquare$$

Bien entendu, des inégalités analogues s'obtiennent pour des fonctions décroissantes.

Preuve de la proposition. Commençons par (1.1). Il est clair que $\sum_{i=1}^{n} i^k = O(n^{k+1})$: il suffit pour cela de majorer chaque terme de la somme par n^k. Il est clair aussi que $\sum_{i=1}^{n} i^k = \Omega(n^k)$: il suffit pour cela de négliger tous les termes sauf le dernier. La difficulté est donc de remplacer la minoration grossière $\Omega(n^k)$ par la minoration plus précise $\Omega(n^{k+1})$. Pour ce faire, nous utilisons le lemme

précédent. La fonction $x \mapsto x^k$ est croissante et continue sur l'intervalle $[0, n]$. On a donc

$$\sum_{i=1}^{n} i^k \geq \int_0^n x^k \, dx = \frac{n^{k+1}}{k+1}$$

ce qui prouve que la somme est dans $\Omega(n^{k+1})$.

Prouvons la deuxième formule. On a clairement $\log n! = O(n \log n)$. D'autre part, la fonction $x \mapsto \ln x$ est croissante et continue sur l'intervalle $[1, n]$. Par le lemme, on a donc

$$\ln n! = \sum_{i=2}^{n} \ln i \geq \int_1^n \ln x \, dx = [x \ln x - x]_1^n$$

donc

$$\ln n! \geq n \ln n - n + 1 \geq \frac{n \ln n}{2}$$

dès que $\ln n \geq 2$. Ceci montre que $\ln n! \in \Omega(n \ln n)$. Comme $\ln n = \theta(\log n)$, la même formule vaut pour log.

Considérons enfin la troisième formule. La fonction $x \mapsto 1/x$ est décroissante et continue sur l'intervalle $[1, n+1]$. On a donc par le lemme

$$\sum_{i=1}^{n} \frac{1}{i} \geq \int_1^{n+1} \frac{1}{x} \, dx = \ln(n+1)$$

De même,

$$\sum_{i=2}^{n} \frac{1}{i} \leq \int_1^n \frac{1}{x} \, dx = \ln n$$

d'où

$$\ln(n+1) \leq \sum_{i=1}^{n} \frac{1}{i} \leq 1 + \ln n. \qquad \blacksquare$$

Remarquons que, pour les trois formules de la proposition, on dispose en fait d'expressions plus précises dont l'établissement dépasse le cadre de ce livre. On a

$$\sum_{i=1}^{n} i^k = \frac{B_{n+1}(n+1) - B_{n+1}}{k+1}$$

où $B_n(x)$ est le n-ième *polynôme de Bernoulli* et $B_n = B_n(0)$ est le n-ième *nombre de Bernoulli*. La *formule de Stirling*

$$n! \sim n^n e^{-n} \sqrt{2\pi n} \left(1 + \frac{1}{12n} + \frac{23}{288n^2} + \cdots\right)$$

permet de retrouver l'évaluation asymptotique de $\log n!$. Enfin, les *nombres harmoniques* $H_n = 1 + \frac{1}{2} + \cdots + \frac{1}{n}$ satisfont

$$H_n = \ln n + \gamma + \frac{1}{2n} + \cdots$$

où γ est la constante d'Euler.

Exemple. Considérons un exemple plus substantiel. Il a pour but de développer une estimation de la croissance asymptotique d'une fonction définie implicitement, en employant une méthode d'affinages successifs appelée en anglais « bootstrapping ». Soit la fonction f donnée par l'équation

$$f(t)e^{f(t)} = t$$

On veut l'étudier lorsque t tend vers $+\infty$. On constate d'abord que, pour $t > 0$, on a $f(t) > 0$. On peut donc passer au logarithme, et on obtient

$$\ln f(t) + f(t) = \ln t$$

et l'expression

$$f(t) = \ln t - \ln f(t) \tag{1.2}$$

L'équation de définition de $f(t)$ montre que pour $t \geq e$, on a $f(t) \geq 1$. Par conséquent, $\ln f(t) \geq 0$, et l'équation (1.2) montre que $f(t) \leq \ln t$ et donc

$$f(t) = O(\ln t)$$

Ceci « amorce la pompe ». Il résulte ensuite de cette relation que

$$\ln f(t) = \ln O(\ln t) = O(\ln \ln t)$$

et par conséquent on peut affiner l'équation (1.2) en

$$f(t) = \ln t + O(\ln \ln t)$$

On reporte ceci à nouveau dans l'équation (1.2), et on obtient

$$\begin{aligned} f(t) &= \ln t - \ln(\ln t + O(\ln \ln t)) \\ &= \ln t - \ln(\ln t\,(1 + O(\ln \ln t)/\ln t)) \\ &= \ln t - \ln \ln t - \ln\,(1 + O(\ln \ln t)/\ln t) \\ &= \ln t - \ln \ln t + O(\ln \ln t/\ln t) \end{aligned}$$

Cette dernière expression donne une description déjà fort précise du comportement asymptotique de la fonction f.

Exemple. On veut donner une expression asymptotique, lorsque n tend vers $+\infty$, de

$$\sqrt[n]{n} \quad \text{ et de } \quad n\left(\sqrt[n]{n} - 1\right)$$

Pour cela, on se rappelle qu'au voisinage de 0, la fonction e^x admet le développement limité

$$e^x = 1 + x + O(x^2)$$

Comme $\sqrt[n]{n} = e^{\ln n/n}$, et que $\ln n/n$ est voisin de 0 lorsque n est voisin de ∞, on a

$$\sqrt[n]{n} = e^{\ln n/n} = 1 + \ln n/n + O\left(\ln n/n\right)^2$$

Cela montre également que

$$n\left(\sqrt[n]{n} - 1\right) = \ln n + O\left(\ln^2 n/n\right).$$

2.2 Récurrences

L'analyse des performances d'un algorithme donne en général des équations implicites, où le temps de calcul, pour une taille des données, est exprimé en fonction du temps de calcul pour des données plus petites. Résoudre ces équations n'est pas toujours possible, et l'on se contente de donner des équivalents qui décrivent, de manière satisfaisante, le comportement des algorithmes. Dans cette section, on passe en revue certaines techniques permettant d'obtenir des expressions explicites exactes ou approchées pour des fonctions données par des relations de récurrence.

2.2.1 Récurrences linéaires à coefficients constants

Nous considérons ici les relations de récurrence linéaires à coefficients constants. Des relations plus générales sont décrites dans la section suivante.

Relations de récurrence homogènes

Une *suite récurrente linéaire* est une suite $(u_n)_{n\geq 0}$ de nombres (réels) qui vérifie une relation du type suivant :

$$u_{n+h} = a_{h-1}u_{n+h-1} + \cdots + a_0 u_n, \qquad n = 0, 1, 2, \ldots \quad a_0 \neq 0 \tag{2.1}$$

où h est un entier strictement positif et $a_{h-1}, \ldots, a_0$ sont des nombres fixés. Une telle suite est entièrement déterminée par ses h premières valeurs $u_0, \ldots, u_{h-1}$. On dit que la suite est d'ordre h. Le *polynôme associé* ou *caractéristique*de l'équation (2.1) est par définition

$$G(X) = X^h - a_{h-1}X^{h-1} - \cdots - a_1 X - a_0$$

On note ses racines $\omega_1, \ldots, \omega_r$, la multiplicité de ω_i étant notée n_i pour $i = 1, \ldots, r$.

Considérons la série génératrice (formelle) de la suite (u_n) :

$$u(X) = \sum_{n=0}^{\infty} u_n X^n \tag{2.2}$$

et soit $B(X) = X^h G(1/X)$ le polynôme réciproque de $G(X)$, c'est-à-dire

$$B(X) = 1 - a_{h-1}X - \cdots - a_0 X^h$$

Posons $A(X) = B(X)u(X)$. En vertu de la formule (2.1), il vient

$$A(X) = \sum_{j=0}^{h-1} \left(u_j - \sum_{i=1}^{j} a_{h-1-i} u_{j-i} \right) X^j$$

Ainsi $A(X)$ est un polynôme de degré au plus $h-1$, et la formule

$$u(X) = \frac{A(X)}{B(X)} \tag{2.3}$$

montre que la série formelle est une série rationnelle. En décomposant la fraction rationnelle (2.3) en éléments simples, on obtient une expression du type

$$u(X) = \sum_{i=1}^{r} \sum_{j=1}^{n_i} \frac{\beta_{i,j}}{(1-\omega_i X)^j}$$

(rappelons que n_i est la multiplicité de la racine ω_i). Comme

$$\frac{1}{(1-\omega X)^j} = \sum_{n=0}^{\infty} \binom{n+j-1}{j-1} \omega^n X^n$$

le terme général de la suite (u_n) est donné par une expression de la forme

$$u_n = \sum_{i=1}^{r} P_i(n)\omega_i^n \tag{2.4}$$

où

$$P_i(X) = \sum_{j=1}^{n_i} \beta_{i,j} \binom{X+j-1}{j-1} \tag{2.5}$$

est un polynôme de degré au plus $n_i - 1$ pour $i = 1, \ldots, r$. La formule (2.4) est très utile; c'est elle qui donne la forme close de u_n; c'est aussi elle qui permet d'obtenir une estimation asymptotique lorsque la racine de plus grand module est unique et réelle positive par exemple. Une expression (2.4) est appelée un *polynôme exponentiel*.

Réciproquement, tout polynôme $P_i(X)$ de degré n_i s'écrit sous la forme (2.5) parce que les polynômes

$$\binom{X+j}{j} = \frac{1}{j!}(X+1)\cdots(X+j) \qquad j \geq 0$$

forment une base de l'espace des polynômes. Si le terme général de la suite (u_n) est donné par (2.4), on obtient donc une expression (2.3) puis une relation (2.1) pour la suite. Nous avons donc montré qu'une suite récurrente a trois représentations équivalentes : la relation de récurrence, l'expression comme fraction rationnelle de sa série génératrice, et l'expression close de ses termes sous forme de polynôme exponentiel.

Exemple. Considérons la suite

$$\begin{aligned} &t_n = 3t_{n-1} + 4t_{n-2} \qquad n \geq 2 \\ &t_0 = 0, \quad t_1 = 1 \end{aligned}$$

Elle est d'ordre 2, son polynôme caractéristique est $X^2 - 3X - 4$, et ses racines sont -1 et 4. On obtient

$$u(X) = \frac{X}{1 - 3X - 4X^2} = \frac{1}{5}\left(\frac{1}{1-4X} - \frac{1}{1+X}\right)$$

d'où l'expression

$$t_n = \frac{4^n - (-1)^n}{5} \qquad n \geq 0$$

Exemple. La suite

$$\begin{array}{l} t_{n+3} = 5t_{n+2} - 8t_{n+1} + 4t_n \qquad n \geq 0 \\ t_0 = 0, \quad t_1 = 1, \quad t_2 = 2 \end{array}$$

est d'ordre 3, et son polynôme caractéristique $X^3 - 5X^2 + 8X - 4$ a la racine simple 1 et la racine double 2. On obtient

$$u(X) = \frac{X - 3X^2}{1 - 5X + 8X^2 - 4X^3} = -\frac{2}{1-X} + \frac{5/2}{1-2X} - \frac{1/2}{(1-2X)^2}$$

En développant, ceci donne

$$t_n = 2^{n+1} - n2^{n-1} - 2 \qquad n \geq 0$$

Notons que la même relation de récurrence, avec les conditions initiales $t_0 = 1$, $t_1 = 3$, et $t_2 = 7$, conduit à l'expression $t_n = 2^{n+1} - 1$. Ceci est dû au fait que la fraction

$$u(X) = \frac{1 - 2X}{1 - 5X + 8X^2 - 4X^3}$$

se simplifie, et que la suite (t_n) est, dans ce cas, aussi solution d'une relation d'ordre 2.

Exemple. L'exemple le plus populaire et sans doute aussi le plus ancien (il date de 1202) est la suite (F_n) de Fibonacci définie par

$$\begin{array}{l} F_{n+2} = F_{n+1} + F_n \qquad n \geq 0 \\ F_0 = 0, \quad F_1 = 1 \end{array}$$

Le polynôme caractéristique $X^2 - X - 1$ a les deux racines

$$\phi = \frac{1+\sqrt{5}}{2} \quad \text{et} \quad \bar{\phi} = \frac{1-\sqrt{5}}{2}$$

et sa série génératrice

$$u(X) = \frac{X}{1 - X - X^2} = \frac{1}{\sqrt{5}}\left(\frac{1}{1-\phi X} - \frac{1}{1-\bar{\phi}X}\right)$$

donne l'expression

$$F_n = \frac{1}{\sqrt{5}}(\phi^n - \bar{\phi}^n) \qquad n \geq 0$$

Asymptotiquement, on a

$$F_n \sim \frac{1}{\sqrt{5}}\phi^n$$

Comme exemple d'application, considérons l'*algorithme d'Euclide*de calcul du pgcd de deux entiers positifs, non tous les deux nuls, par divisions successives :

```
fonction PGCD(x, y);
   si y = 0 alors PGCD:= x
   sinon PGCD:=PGCD(y, x mod y)
   finsi.
```

Notons $n(x, y)$ le nombre de divisions avec reste effectuées par l'algorithme. Alors

$$n(x,y) \;=\; \text{si } y = 0 \text{ alors } 0 \text{ sinon } 1 + n(y, x \bmod y)$$

Nous allons prouver que pour $x > y \geq 0$,

$$n(x,y) = k \;\Rightarrow\; x \geq F_{k+2} \tag{2.6}$$

Pour $k = 0$, on a $x \geq 1 = F_2$, et pour $k = 1$, on a $x \geq 2 = F_3$. Supposons donc $k \geq 2$, et considérons les divisions euclidiennes

$$x = qy + z,\ 0 \leq z < y$$
$$y = q'z + u, 0 \leq u < z$$

On a $n(y, z) = k - 1$, donc $y \geq F_{k+1}$, et de même $z \geq F_k$, par conséquent $x \geq F_{k+1} + F_k = F_{k+2}$. Ceci prouve (2.6). Comme $F_n \geq \frac{1}{\sqrt{5}}(\phi^n - 1)$, on a $\sqrt{5}x + 1 \geq \phi^{k+2}$, donc pour $x > y \geq 0$,

$$n(x,y) \leq \log_\phi(5x + 1) - 2$$

Ce résultat est le théorème de Lamé. En particulier, $n(x, y) = O(\log x)$.

Relations de récurrence à second membre

Nous considérons maintenant des suites $(u_n)_{n\geq 0}$ de nombres qui vérifient une relation du type suivant :

$$u_{n+h} = a_{h-1}u_{n+h-1} + \cdots + a_0 u_n + v_n, \qquad n = 0, 1, 2, \ldots \quad a_0 \neq 0 \tag{2.7}$$

où h est un entier positif et $a_{h-1}, \ldots, a_0$ sont des nombres fixés, et où v_n est une fonction de n. On peut réécrire (2.7) en

$$u_{n+h} - a_{h-1}u_{n+h-1} - \cdots - a_0 u_n = v_n, \qquad n \geq 0, \quad a_0 \neq 0$$

ce qui explique le terme de relation de récurrence avec second membre. Comme dans le cas traité plus haut, une telle suite est entièrement déterminée par ses h premières valeurs $u_0, \ldots, u_{h-1}$. On dit que la suite est d'ordre h. Considérons les séries génératrices (formelles) des suites (u_n) et (v_n) :

$$u(X) = \sum_{n=0}^{\infty} u_n X^n \qquad v(X) = \sum_{n=0}^{\infty} v_n X^n$$

Les calculs faits pour les récurrences homogènes conduisent cette fois-ci à l'expression

$$u(X) = \frac{A(X) + X^h v(X)}{B(X)} \tag{2.8}$$

où comme plus haut $B(X) = 1 - a_{h-1}X - \cdots - a_0 X^h$ et

$$A(X) = \sum_{j=0}^{h-1} \left(u_j - \sum_{i=1}^{j} a_{h-1-i} u_{j-i} \right) X^j$$

Calculer les polynômes $A(X)$ et $B(X)$ revient à résoudre l'équation sans second membre, qui est ensuite «corrigée» par la série $v(X)$.

Nous nous intéressons au cas particulier où la suite (v_n) vérifie elle-même une relation de récurrence. Dans ce cas, la série génératrice v est elle-même une fraction rationnelle :

$$v(X) = \frac{P(X)}{Q(X)}$$

pour des polynômes P et Q. L'équation (2.8) prend alors la forme

$$u(X) = \frac{A(X)Q(X) + X^h P(X)}{B(X)Q(X)} \tag{2.9}$$

ce qui montre que la série u est encore rationnelle.

Exemple. Considérons la suite (t_n) définie par

$$\begin{aligned} t_n &= 2t_{n-1} + 1 \qquad n \geq 1 \\ t_0 &= 0 \end{aligned}$$

La suite associée à l'équation homogène $t_n = 2t_{n-1}$, avec $t_0 = 0$ est nulle. Donc $A(X) = 0$, $B(X) = 1 - 2X$. Par ailleurs, $v(X) = 1/(1-X)$, et

$$u(X) = \frac{X}{(1-2X)(1-X)}$$

En particulier, la suite (t_n) vérifie la relation de récurrence $t_{n+2} = 3t_{n+1} - 2t_n$, et bien entendu $t_n = 2^n - 1$.

Une autre façon de résoudre (2.7) lorsque la suite v_n est une suite récurrente linéaire, est de substituer cette relation de récurrence. Si

$$v_{n+k} = b_1 v_{n+k-1} + \cdots + b_k v_n \qquad n \geq 0$$

alors pour $n \geq 0$

$$u_{n+h+k} = \sum_{i=1}^{h} a_i u_{n+h+k-i} + \sum_{j=1}^{k} b_j \left(u_{n+h+j} - \sum_{i=1}^{h} a_i u_{n+h-i+k-j} \right)$$

ce qui, après réarrangement, donne une relation de récurrence explicite pour (u_n).

Exemple. Considérons la suite

$$\begin{aligned} &t_{n+1} - 2t_n = n + 2^n \qquad n \geq 1 \\ &t_0 = 0 \end{aligned}$$

La suite $v_n = n + 2^n$ vérifie la relation $v_{n+3} = 4v_{n+2} - 5v_{n+1} + 2v_n$, d'où

$$t_{n+4} - 2t_{n+3} = 4(t_{n+3} - 2t_{n+2}) - 5(t_{n+2} - 2t_{n+1}) + 2(t_{n+1} - 2t_n)$$

et finalement

$$t_{n+4} = 6t_{n+3} - 7t_{n+2} - 2t_n$$

Exemple. La fonction récursive de calcul des nombres de Fibonacci est souvent utilisée pour tester l'efficacité de compilateurs ou d'interprètes. Elle s'écrit :

```
fonction FIBONACCI(n);
  si n = 0 ou n = 1 alors
    FIBONACCI := n
  sinon
    FIBONACCI := FIBONACCI(n − 1)+FIBONACCI(n − 2)
  finsi.
```

Notons r_n le nombre d'appels de FIBONACCI pour le calcul du nombre F_n. On a

$$\begin{aligned} &r_n = 1 + r_{n-1} + r_{n-2} \qquad n \geq 2 \\ &r_0 = r_1 = 1 \end{aligned}$$

On trouve sans peine que

$$r_n = 2F_{n+1} - 1 = \theta(\phi^n)$$

donc que le temps d'exécution est proportionnel à la valeur calculée. Ceci n'est pas étonnant puisque la procédure forme le résultat en additionnant des 0 et des 1.

Exemple. Parfois, on peut se ramener à des récurrences linéaires par un changement de variables, ou un changement de valeurs (remplacer une fonction par son logarithme par exemple). Considérons la fonction t définie sur les entiers qui sont des puissances de 2 par

$$\begin{aligned} t(1) &= 6 \\ t(n) &= n(t(n/2))^2 \qquad n > 1 \text{ puissance de } 2 \end{aligned}$$

On peut vérifier directement que $t(n) = 2^{3n-2}3^n/n$; pour trouver cette formule, on fait d'abord un changement de variable en posant $s(k) = t(2^k)$ pour $k \geq 0$, d'où la relation

$$\begin{aligned} s(0) &= 6 \\ s(k) &= 2^k(s(k-1))^2 \end{aligned}$$

Ensuite, on pose $u_k = \log_2 s(k)$, et on obtient la relation de récurrence linéaire

$$\begin{aligned} u_0 &= \log 6 \\ u_k &= k + 2u_{k-1} \end{aligned}$$

Cette dernière se résout par les méthodes indiquées ci-dessus, et donne

$$u_k = 2^k u_0 + 2^{k+1} - k - 2$$

d'où en reportant :

$$s(k) = 2^{u_k} = 6^{2^k} 2^{2^{k+1}} 2^{-k-2}$$

et le résultat, et posant $n = 2^k$.

2.2.2 Récurrences diverses

Voici quelques techniques permettant de résoudre certaines relations de récurrences qui ne sont pas de la forme précédente.

Méthode des facteurs sommants

La méthode des facteurs sommants permet de traiter des suites (u_n) définies par une relation de récurrence linéaire d'ordre 1, de la forme

$$a_n u_n = b_n u_{n-1} + c_n \qquad n \geq 1$$

où a_n, b_n, c_n sont des fonctions de n. On pose alors

$$f_n = \frac{a_1 \cdots a_{n-1}}{b_1 \cdots b_n} \qquad n \geq 1$$

avec $f_1 = 1/b_1$. On a $f_n a_n = f_{n+1} b_{n+1}$. En posant

$$y_n = f_n a_n u_n$$

la relation se réécrit

$$y_n = y_{n-1} + f_n c_n$$

Cette relation se résout immédiatement en

$$y_n = y_0 + \sum_{i=1}^{n} f_i c_i$$

et on obtient, pour u_n, l'expression

$$u_n = \frac{u_0 + \sum_{i=1}^{n} f_i c_i}{a_n f_n}$$

Exemple. Considérons la suite (u_n) définie par

$$\begin{aligned} u_0 &= a \\ u_n &= bn^k + nu_{n-1} \qquad n > 1 \end{aligned}$$

où a et b sont des réels, et $k \in \mathbb{N}$. Les formules précédentes s'appliquent avec $c_n = bn^k$, $a_n = 1$, $b_n = n$, d'où $f_n = 1/n!$ et

$$u_n = n! \left(a + b \sum_{i=1}^{n} \frac{i^k}{i!} \right)$$

Maintenant, la série de terme général $i^k/i!$ converge, et le nombre

$$B_k = \frac{1}{e} \sum_{i=1}^{\infty} \frac{i^k}{i!}$$

est le k-ième *nombre de Bell*. On a donc

$$u_n \sim n!(a + beB_k)$$

Changement de valeurs

Dans certaines situations, on peut remplacer la fonction à évaluer par son logarithme, et obtenir une expression plus simple pour la relation de récurrence.

Exemple. Le nombre de fonctions booléennes à n variables booléennes est donné par

$$\begin{aligned} t(1) &= 4 \\ t(n) &= (t(n-1))^2 \qquad n > 1 \end{aligned}$$

Posons $u_n = \log t(n)$. On a alors

$$\begin{aligned} u_1 &= 2 \\ u_n &= 2u_{n-1} \qquad n > 1 \end{aligned}$$

d'où $u_n = 2^n$ et $t(n) = 2^{2^n}$.

Voici un exemple où le même procédé est appliqué, mais où les calculs sont beaucoup plus compliqués.

Exemple. Le nombre d'arbres binaires de hauteur strictement inférieure à n est donné par

$$\begin{aligned} x_0 &= 1 \\ x_{n+1} &= x_n^2 + 1 \qquad n \geq 0 \end{aligned}$$

L'arbre vide est de hauteur -1. Nous allons montrer qu'il existe une constante

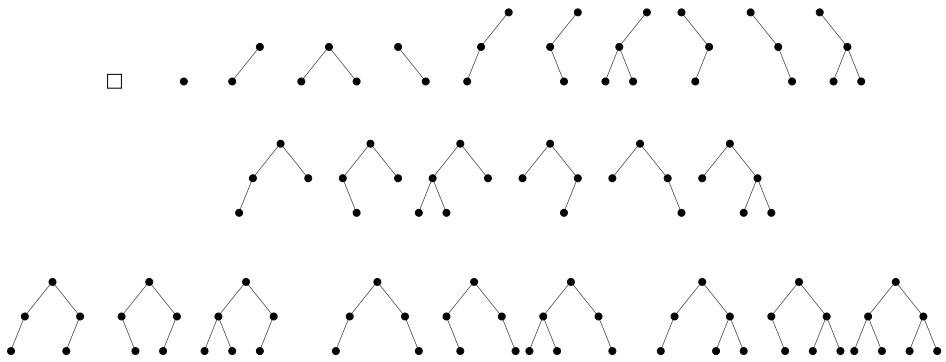


Figure 2.1: *Les arbres binaires de hauteur* $-1 \leq h \leq 2$.

k telle que $x_n = \left\lfloor k^{2^n} \right\rfloor$. Posons pour cela $y_n = \ln x_n$. On a

$$y_{n+1} = 2y_n + \alpha_n$$

avec

$$\alpha_n = \ln\left(1 + \frac{1}{x_n^2}\right)$$

On en déduit l'expression

$$y_n = 2^n \left(\frac{\alpha_0}{2} + \cdots + \frac{\alpha_{n-1}}{2^n}\right)$$

Posons

$$Y_n = 2^n \sum_{i=0}^{\infty} \frac{\alpha_i}{2^{i+1}}, \qquad r_n = Y_n - y_n$$

Comme la suite (x_n) est croissante, et donc la suite (α_n) est décroissante, on a l'estimation suivante pour r_n :

$$r_n = Y_n - y_n = 2^n \sum_{i=n}^{\infty} \frac{\alpha_i}{2^{i+1}} \leq \alpha_n \sum_{i=0}^{\infty} \frac{1}{2^{i+1}} \leq \alpha_n$$

Si l'on définit le nombre k par

$$k = \exp\left(\sum_{i=0}^{\infty} \frac{\alpha_i}{2^{i+1}}\right)$$

on a $Y_n = 2^n \ln k$, donc $x_n = e^{y_n} = e^{Y_n - r_n} = k^{2^n} e^{-r_n}$ et il ne reste plus qu'à utiliser la majoration pour r_n :

$$x_n \leq k^{2^n} = x_n e^{r_n} \leq x_n e^{\alpha_n} = x_n \left(1 + \frac{1}{x_n^2}\right) = x_n + \frac{1}{x_n} < x_n + 1$$

pour $n > 1$, d'où

$$x_n = \left\lfloor k^{2^n} \right\rfloor$$

Evidemment, l'expression de k n'est pas facile à évaluer...

2.2.3 Récurrences de partitions

Les relations de récurrence que nous considérons maintenant sont de la forme :

$$\begin{aligned} &t(n_0) = d \\ &t(n) = at(n/b) + f(n) \qquad n > n_0 \end{aligned}$$

On les rencontre naturellement lorsque l'on cherche un algorithme récursif pour résoudre un problème de taille n par la méthode des sous-problèmes (« diviser pour régner ») : on remplace le problème par a sous-problèmes, chacun de taille n/b. Si $t(n)$ est le coût de l'algorithme pour la taille n, il se compose donc de $at(n/b)$ plus le temps $f(n)$ pour recomposer les solutions des problèmes partiels en une solution du problème total. Dans de nombreux cas, les équations ci-dessus sont en fait des inéquations. Le coût $\tau(n)$ vérifie

$$\begin{aligned} &\tau(n_0) = d \\ &\tau(n) \leq a\tau(n/b) + f(n) \qquad n > n_0 \end{aligned}$$

Il en résulte que $\tau(n) \leq t(n)$ pour tout n, et donc que la fonction t constitue une majoration du coût de l'algorithme. Avant de continuer, un exemple.

Exemple. Soit à calculer le produit de deux entiers u, v vérifiant $0 \leq u, v < 2^{2n}$ pour un entier n. On peut donc écrire u et v en binaire sur $2n$ bits. Posons

$$u = 2^n U_1 + U_0, \quad v = 2^n V_1 + V_0$$

avec $0 \leq U_0, U_1, V_0, V_1 < 2^n$. Ainsi U_1 est formé des n bits de plus fort poids de u, etc. On a

$$uv = (2^{2n} + 2^n)U_1V_1 + 2^n(U_1 - U_0)(V_0 - V_1) + (2^n + 1)U_0V_0$$

Cette formule montre que l'on peut décomposer le problème de la multiplication de deux nombres à $2n$ bits en 3 multiplications de nombres à n bits, à savoir U_1V_1, $(U_1 - U_0)(V_0 - V_1)$, U_0V_0, et quelques additions et décalages. On a donc ramené un problème de taille $2n$ en 3 sous-problèmes de taille n ($a = 3, b = 2$ dans les formules ci-dessus). Soit $t(n)$ le temps requis pour multiplier deux nombres à n bits par cette méthode. Alors

$$t(2n) = 3t(n) + cn, \quad t(1) = 1$$

pour une constante c, puisque les décalages et l'addition prennent un temps linéaire en n. Il résulte de la proposition ci-dessous que $t(n) = \theta(n^{\log_2 3})$.

Un grand nombre de situations rencontrées dans l'analyse d'algorithmes sont couvertes par l'énoncé que voici :

Proposition 2.1. *Soit $t : \mathbb{N} \to \mathbb{R}_+$ une fonction croissante au sens large à partir d'un certain rang, telle qu'il existe des entiers $n_0 \geq 1$, $b \geq 2$ et des réels $k \geq 0$, $a > 0$, $c > 0$ et $d > 0$ pour lesquels*

$$\begin{array}{ll} t(n_0) = d & \\ t(n) \ = at(n/b) + cn^k & \quad n > n_0,\ n/n_0 \ \textit{puissance de}\ b \end{array} \tag{2.10}$$

Alors on a

$$t(n) = \begin{cases} \theta(n^k) & \textit{si } a < b^k \\ \theta(n^k \log_b n) & \textit{si } a = b^k \\ \theta(n^{\log_b a}) & \textit{si } a > b^k \end{cases} \tag{2.11}$$

Preuve. Soit d'abord $n = n_0 b^p$ pour un entier p. Alors

$$t(n) = da^p \sum_{i=0}^{p-1} ca^i \left(n/b^i\right)^k = \delta(n) + cn^k \gamma(n)$$

avec

$$\delta(n) = da^p = \theta(n^{\log_b a}) \ \text{ et } \ \gamma(n) = \sum_{i=0}^{p-1} \left(a/b^k\right)^i$$

parce que $a^p = n^{\log_b a}/a^{\log_b n_0}$. Or

$$\begin{array}{ll} \gamma(n) \sim \dfrac{1}{1 - a/b^k} & \text{si } \ a/b^k < 1 \\ \gamma(n) = p \sim \log_b n & \text{si } \ a/b^k = 1 \\ \gamma(n) \sim \alpha \dfrac{a^p}{b^{kp}} & \text{si } \ a/b^k > 1 \end{array}$$

avec $\alpha = 1/(a/b^k - 1)$. Les deux premières formules s'en déduisent. Si $a > b^k$, on a

$$cn^k\gamma(n) \sim \alpha c n_0^k a^p = \theta(n^{\log_b a})$$

D'où le résultat.

Soit maintenant n assez grand, et soit p tel que $n_0 b^p < n \leq n_0 b^{p+1}$. Posons $n_- = n_0 b^p$ et $n_+ = n_0 b^{p+1} = b n_-$. Comme $t(n_-) \leq t(n) \leq t(n_+)$, et que $f(bn) = \theta(f(n))$ pour chacune des trois fonctions f intervenant au second membre de (2.11), on a $t(n) = \theta(f(n))$. ∎

Voici quelques applications de ces formules dans le cas le plus fréquent, où $b = 2$ et $n_0 = 1$:

Exemple. Soit t une fonction croissante qui vérifie $t(1) = 1$. Si pour $n > 1$, puissance de 2, on a

$$\begin{array}{lll}
t(n) = t(n/2) + c & \text{alors} & t(n) = \theta(\log n) \qquad (a = 1, k = 0) \\
t(n) = 2t(n/2) + cn & \text{alors} & t(n) = \theta(n \log n) \qquad (a = 2, k = 1) \\
t(n) = 2t(n/2) + cn^2 & \text{alors} & t(n) = \theta(n^2) \qquad (a = 2, k = 2) \\
t(n) = 4t(n/2) + cn^2 & \text{alors} & t(n) = \theta(n^2 \log n) \qquad (a = 4, k = 2) \\
t(n) = 7t(n/2) + cn^2 & \text{alors} & t(n) = \theta(n^{\log 7}) \qquad (a = 7, k = 2)
\end{array}$$

Remarque. Parfois, l'analyse d'un algorithme ne permet pas d'obtenir une information aussi précise que celle de l'équation (2.10). Si, dans cette équation, on remplace le terme cn^k par $O(n^k)$, la même conclusion reste vraie, en remplaçant dans (2.11) les θ par des O.

Exemple. Soit t une fonction croissante au sens large et vérifiant

$$\begin{array}{ll}
t(1) & = 1 \\
t(n) & = 2t(n/2) + \log n \qquad n > 1 \text{ et puissance de } 2
\end{array}$$

Comme $\log n = O(n)$, on a $t(n) = O(n \log n)$. En fait, on peut montrer que $t(n) = \theta(n)$ (voir ci-dessous).

Proposition 2.2. *Soit $t : \mathbb{N} \to \mathbb{R}_+$ une fonction croissante au sens large à partir d'un certain rang, telle qu'il existe des entiers $n_0 \geq 1$, $b \geq 2$ et des réels $a > 0$, et $d > 0$ et une fonction $f : \mathbb{N} \to \mathbb{R}_+$pour lesquels*

$$\begin{array}{ll}
t(n_0) & = d \\
t(n) & = at(n/b) + f(n) \qquad n > n_0,\ n/n_0 \text{ une puissance de } b
\end{array}$$

Supposons de plus que

$$f(n) = cn^k(\log_b n)^q$$

pour des réels $c > 0$, $k \geq 0$ et q. Alors

$$t(n) = \begin{cases} \theta(n^k) & \text{si } a < b^k \text{ et } q = 0 \\ \theta(n^k (\log_b n)^{1+q}) & \text{si } a = b^k \text{ et } q > -1 \\ \theta(n^k \log\log_b n) & \text{si } a = b^k \text{ et } q = -1 \\ \theta(n^{\log_b a}) & \text{si } a = b^k \text{ et } q < -1 \\ \theta(n^{\log_b a}) & \text{si } a > b^k \end{cases}$$

Preuve. Procédons comme dans la démonstration du résultat précédent. Soit d'abord $n = n_0 b^p$ pour un entier p. Alors

$$\begin{aligned} t(n) &= da^p + \sum_{i=0}^{p-1} a^i f(n/b^i) \\ &= da^p + \sum_{i=0}^{p-1} a^i f(n_0 b^{p-i}) = da^p + \sum_{i=1}^{p} a^{p-i} f(n_0 b^i) \\ &= da^p + c \sum_{i=1}^{p} a^{p-i} (n_0 b^i)^k (\log_b n_0 b^i)^q \\ &= da^p + c n_0^k a^p \sum_{i=1}^{p} \left(b^k/a\right)^i (\log_b n_0 + i)^q \end{aligned}$$

et comme $a^p = \theta(n^{\log_b a})$, on a $t(n) = \theta(n^{\log_b a} \gamma(n))$, où

$$\gamma(n) = \sum_{i=1}^{p} h^i (r+i)^q$$

avec $h = b^k/a$ et $r = \log_b n_0$. Si $h = 1$, alors

$$\gamma(n) = \sum_{i=1}^{p} (r+i)^q = \begin{cases} \theta(p^{1+q}) = \theta(\log_b n^{1+q}) & \text{si } q > -1 \\ \theta(\log_b p) = \theta(\log_b \log_b n) & \text{si } q = -1 \\ \theta(1) & \text{si } q < -1 \end{cases}$$

Si $h < 1$, alors $\gamma(n) = \theta(1)$. Enfin, si $h > 1$ et $q = 0$, alors $\gamma(n) = \theta(n^k / n^{\log_b a})$. D'où le résultat. ∎

Exemple. Si la fonction t vérifie, pour tout n puissance de 2,

$t(n) = 2t(n/2) + \log n$	alors	$t(n) = \theta(n)$	$(h = 1/2, k = 0, q = 1)$
$t(n) = 3t(n/2) + n \log n$	alors	$t(n) = \theta(n^{\log 3})$	$(h = 2/3, k = q = 1)$
$t(n) = 2t(n/2) + n \log n$	alors	$t(n) = \theta(n \log^2 n)$	$(h = 1, k = q = 1)$
$t(n) = 5t(n/2) + (n \log n)^2$	alors	$t(n) = \theta(n^{\log 5})$	$(h = 4/5, k = q = 2)$

Exemple. Soit t une fonction vérifiant

$$\begin{aligned} t(1) &= a \\ t(n) &= t(\lfloor n/2 \rfloor) + t(\lceil n/2 \rceil) + bn \qquad n > 1 \end{aligned}$$

Clairement, t est croissante, et comme $t(n) = 2t(n/2) + bn$ lorsque n est une puissance de 2, on a $t(n) = \theta(n \log n)$. Certaines définitions récurrentes de ce type peuvent être résolues de manière complète (voir exercices).

Proposition 2.3. *Soient $\alpha_1, \cdots, \alpha_k$ des fonctions de $\mathbb{N}$ dans lui-même telles qu'il existe un nombre réel $0 < K < 1$ avec*

$$\alpha_1(n) + \cdots + \alpha_k(n) \leq Kn \qquad (n > n_0)$$

Si une fonction t vérifie

$$t(n) = \begin{cases} a_0 & n \leq n_0 \\ t(\alpha_1(n)) + \cdots + t(\alpha_k(n)) + bn & n > n_0 \end{cases}$$

alors $t(n) = O(n)$.

Preuve. Nous allons montrer que

$$t(n) \leq Dn + a_0 \quad \text{avec} \quad D = (b + (k-1)a_0)/(1-K)$$

Cette inégalité est évidemment vérifiée pour $n \leq n_0$. Si $n > n_0$, alors

$$\begin{aligned} t(n) &= t(\alpha_1(n)) + \cdots + t(\alpha_k(n)) + bn \\ &\leq ka_0 + D(\alpha_1(n) + \cdots + \alpha_k(n)) + bn \\ &\leq ka_0 + DKn + bn \end{aligned}$$

Or $ka_0 + DKn + bn \leq Dn + a_0$ si, et seulement si $(k-1)a_0 + bn \leq D(1-K)n$, et cette dernière inégalité est satisfaite par l'expression donnée pour D. ∎

Exemple. Considérons la fonction définie par

$$t(n) = \begin{cases} a & n \leq n_0 \\ t(\lfloor n/5 \rfloor) + t(\lfloor 7n/10 \rfloor) + bn & n > n_0 \end{cases}$$

On a ici $\alpha_1(n) = \lfloor n/5 \rfloor$ et $\alpha_2(n) = \lfloor 7n/10 \rfloor$, et

$$\alpha_1(n) + \alpha_2(n) \leq 9n/10$$

Donc $t(n) = O(n)$.

2.2.4 Récurrences complètes

Il s'agit de suites (u_n) définies par des relations de récurrences portant sur toute la suite, plus précisément où u_n est défini en fonction de tous les u_k, pour $k < n$.

Exemple. Les relations de récurrence

$$t_n = cn + \frac{2}{n}\sum_{k=0}^{n-1} t_k \quad \text{ou} \quad t_n = n + \max_{1 \leq k < n} \left(\sum_{i=1}^{k-1} t_{n-i} + \sum_{i=k}^{n-1} t_i \right)$$

sont des relations de récurrence complètes.

Considérons la suite (t_n) vérifiant la relation de récurrence

$$t_n = h_n + \frac{2}{n}\sum_{k=0}^{n-1} t_k$$

pour $n \geq 1$, et $t_0 = 0$. Dans une première étape, nous allons remplacer cette relation de récurrence par une relation plus simple. Pour cela, observons que

$$n(t_n - h_n) = 2\sum_{k=0}^{n-1} t_k$$

On soustrait de cette équation la même équation pour $n-1$, ce qui donne

$$n(t_n - h_n) - (n-1)(t_{n-1} - h_{n-1}) = 2t_{n-1}$$

ou encore

$$nt_n = (n+1)t_{n-1} + g_n$$

en posant $g_n = nh_n - (n-1)h_{n-1}$.

Dans une deuxième étape, on applique la méthode des facteurs sommants. On a

$$t_n = \frac{1}{nf_n}\left(t_0 + \sum_{k=1}^{n} f_k g_k\right)$$

avec

$$f_n = \frac{(n-1)!}{(n+1)!} = \frac{1}{n(n+1)}$$

d'où

$$t_n = (n+1)\left(\sum_{k=1}^{n} \frac{g_k}{k(k+1)}\right)$$

Considérons maintenant le cas particulier, qui intervient dans l'analyse du tri rapide (voir chapitre 5), où

$$\begin{aligned} h_0 &= h_1 = 0 \\ h_k &= k+1 \qquad (k \geq 2) \end{aligned}$$

On a alors

$$\begin{aligned} g_0 &= 0,\ g_1 = 6 \\ g_k &= 2k \qquad (k \geq 2) \end{aligned}$$

et par conséquent

$$\begin{aligned} \sum_{k=1}^{n} \frac{g_k}{k(k+1)} &= 1 + \sum_{k=3}^{n} \frac{2k}{k(k+1)} = 1 + 2\sum_{k=4}^{n} \frac{1}{k} \\ &= 1 + 2(H_{n+1} - 1 - \frac{1}{2} - \frac{1}{3}) = 2(H_{n+1} - \frac{4}{3}) \end{aligned}$$

où H_n est le n-ième nombre harmonique. Il en résulte que

$$t_n = 2(n+1)(H_{n+1} - \frac{4}{3})$$

Notes

L'analyse d'algorithmes fait fréquemment appel à des outils mathématiques bien plus sophistiqués que ceux présentés ici. En plus du traité de Knuth

D. E. Knuth, *The Art of Computer Programming*, Vol I–III, Addison-Wesley, 1968–1973, on peut consulter, pour un bonne introduction et des compléments,

R. Graham, D. E. Knuth, O. Patashnik, *Concrete Mathematics*, Addison-Wesley, 1989,

D. H. Greene, D. E. Knuth, *Mathematics for the Analysis of Algorithms*, Birkhäuser, 1982.

Nous avons adopté la terminologie de

C. Froidevaux, M. C. Gaudel, M. Soria, *Types de données et algorithmes*, McGraw-Hill, 1990.

Exercices

2.1. Donner une expression simple pour la fonction t définie sur les entiers de la forme $n = 2^{2^k}$ par

$$\begin{aligned} t(2) &= 1 \\ t(n) &= 2t(\sqrt{n}) + \log n \qquad n \geq 4 \end{aligned}$$

2.2. Montrer que, quels que soient les entiers positifs n et d, on a

$$\sum_{k=0}^{d} 2^k \log(n/2^k) = 2^{d+1} \log \frac{n}{2^{d-1}} - 2 - \log n$$

En déduire que

$$\sum_{i=1}^{n} \log(n/i) \leq 5n$$

2.3. Montrer que la fonction t définie par

$$\begin{aligned} t(1) &= 0 \\ t(n) &= t(\lfloor n/2 \rfloor) + t(\lceil n/2 \rceil) + n \qquad n > 2 \end{aligned}$$

est $t(n) = n \lfloor \log n \rfloor + 2n - 2^{1+\lfloor \log n \rfloor}$.

2.4. Montrer que la fonction t définie par

$$\begin{aligned} t(1) &= 0 \\ t(n) &= t(\lfloor n/2 \rfloor) + t(\lceil n/2 \rceil) + n - 1 \qquad n > 2 \end{aligned}$$

est $t(n) = n\lceil \log n \rceil + 1 - 2^{\lceil \log n \rceil}$.

2.5. Montrer que

$$\sum_{i=1}^{n} \lfloor \log i \rfloor = 2 + (n+1)\lfloor \log n \rfloor - 2^{1+\lfloor \log n \rfloor}$$

2.6. Donner l'expression exacte de la fonction t définie par

$$\begin{aligned} t(1) &= 0, \quad t(2) = 2 \\ t(n) &= t(\lfloor n/2 \rfloor - 1) + t(\lceil n/2 \rceil) + t(1 + \lceil n/2 \rceil) + n \qquad n > 2 \end{aligned}$$

2.7. Montrer que pour

$$\begin{aligned} t(1) &= a \\ t(n) &= t(\lfloor n/2 \rfloor) + t(\lceil n/2 \rceil) + t(1 + \lfloor n/2 \rfloor) + bn \qquad n > 1 \end{aligned}$$

on a $t(n) = \theta(n^{\log 3})$.

2.8. Donner une expression pour la fonction t définie sur les entiers naturels par

$$t(n) = \begin{cases} 0 & n = 0 \\ 1 + t(n-1) & n \text{ impair} \\ 1 + t(n/2) & n \text{ pair} \end{cases}$$

2.9. Montrer que la suite (t_n) définie par $t_1 = 1$ et

$$t_n = n + \max_{1 \le k < n} \left(\sum_{i=1}^{k-1} t_{n-i} + \sum_{i=k}^{n-1} t_i \right)$$

vérifie $t_n \le 4n$.

2.10. On considère la suite (t_n) définie par

$$\begin{aligned} t_0 &= 1/4 \\ t_n &= \frac{1}{4 - t_{n-1}} \qquad n > 0 \end{aligned}$$

Montrer que $t_n = u_n/u_{n+1}$, où (u_n) est une suite récurrente linéaire d'ordre 2; donner son expression close. Quelle est la limite de t_n lorsque n tend vers l'infini?

2.11. Montrer que la suite définie par

$$\begin{aligned} t(n+2) &= \frac{1 + t(n+1)}{t(n)} \qquad n \ge 2 \\ t(0) &= a, \quad t(1) = b \end{aligned}$$

est périodique.

Chapitre 3

Structures de données

Dans ce chapitre, nous passons en revue les structures de données élémentaires, à savoir les piles, les files, les listes, ainsi que les arbres binaires et les arbres binaires de recherche. Nous décrivons ensuite les files de priorité et leur implémentation au moyen de tas. La dernière section traite du problème de la gestion efficace d'une partition.

3.1 Types de données et structures de données

Dans un langage de programmation comme Pascal, les objets sont déclarés avec leur *type*. Les types constituent une description du format de représentation interne des données en machine, et ils sont par là-même un outil d'abstraction, puisqu'ils libèrent le programmeur du souci de la représentation physique des données. Ainsi, il n'est pas nécessaire de savoir comment est représentée une variable de type `char` ou `boolean`, voire `string`, du moment que l'on sait la manipuler de la manière convenue.

Les types prédéfinis sont peu nombreux. C'est pourquoi des *constructeurs de types* permettent de définir des types plus complexes, et donc des structures de données moins élémentaires. Là encore, l'implémentation précise d'un `array of char` par exemple importe peu au programmeur (est-il rangé par ligne, par colonne ?), aussi longtemps que sa manipulation satisfait à des règles précises. En revanche, lorsque l'on veut réaliser une pile — disons de caractères — , les problèmes se posent autrement, puisqu'il n'existe pas, en Pascal par exemple, de constructeur permettant de définir des `stack of char` : il est nécessaire de recourir à une structure de données.

Une *structure de données* est l'implémentation explicite d'un ensemble organisé d'objets, avec la réalisation des opérations d'accès, de construction et de modification afférentes.

Un *type de données abstrait* est la description d'un ensemble organisé d'objets et des opérations de manipulation sur cet ensemble. Ces opérations comprennent les moyens d'accéder aux éléments de l'ensemble, et aussi, lorsque l'objet est dynamique, les possibilités de le modifier. Plus formellement, on peut donner une définition mathématique d'un type abstrait : c'est une structure algébrique, formée d'un ou de plusieurs ensembles, munis d'opérations vérifiant un ensemble d'axiomes.

Avec ces définitions, une structure de données est la réalisation, l'implémentation explicite d'un type de données. Décrire un type de données, le *spécifier* comme on dit, c'est décrire les opérations possibles et licites, et leur effet. Décrire une structure de données, c'est expliciter comment les objets sont représentés et comment les opérations sont implémentées. Pour l'évaluation d'algorithmes, il importe de plus de connaître leur coût en temps et en place.

Du point de vue des opérations, un type abstrait est à un type de base d'un langage de programmation ce que l'en-tête de procédure est à un opérateur. La procédure réalise une opération qui n'est pas élémentaire au sens du langage de programmation; de la même manière, un type de données abstrait propose une organisation qui n'est pas offerte par le langage. Une réalisation du type abstrait, c'est-à-dire une implémentation par une structure de données, correspond alors à l'écriture du corps d'une procédure.

Le concept de type abstrait ne dépend pas du langage de programmation considéré, et on peut se livrer à l'exercice (un peu stérile) de définir abstraitement les booléens, les réels, etc. En revanche, l'éventail des types concrets varie d'un langage de programmation à un autre. Ainsi, les piles existent dans certains langages de programmations, les tableaux ou les chaînes de caractères dans d'autres.

On peut également considérer un type abstrait comme une boîte noire qui réalise certaines opérations selon des conventions explicites. Cette vision correspond à un style de programmation que l'on pourrait appeler la *programmation disciplinée*. Le programmeur convient d'accéder à la réalisation d'un type de données uniquement à travers un ensemble limité de procédures et de fonctions, et s'interdit notamment tout accès direct à la structure qui serait rendue possible par une connaissance précise de l'implémentation particulière. De nombreux langages de programmation facilitent ce comportement par la possibilité de créer des modules ou unités séparés.

Lorsqu'un type de données est implémenté, on peut se servir de ses opérations comme opérations élémentaires, et en particulier les utiliser dans l'implémentation de types de données plus complexes. Ceci conduit à une *hiérarchie* de types, où l'on trouve, tout en bas de l'échelle, des types élémentaires comme les booléens, entiers, réels, tableaux, à un deuxième niveau les piles, files, listes, arbres, puis les files de priorités, dictionnaires, graphes.

La *réalisation efficace* de types de données par des structures de données qui implémentent les opérations de manière optimale en temps et en place constitue

l'une des préoccupations de l'algorithmique. Le temps requis par une opération de manipulation d'un type s'exprime en fonction du temps requis par les opérations intervenant dans sa réalisation. Ces opérations elles-mêmes sont peut-être complexes, et implémentées dans une autre structure de données. En descendant la hiérarchie, on arrive à des opérations élémentaires (opérations arithmétiques ou tests sur des types élémentaires, affectation de scalaires) dont le temps d'exécution est considéré, par convention, comme constant, et ceci quel que soit le langage de programmation utilisé.

Pour les types de données abstraits les plus simples, plusieurs réalisations efficaces sont faciles à obtenir et à décrire. Il en est ainsi des piles, des files, des listes, et dans une moindre mesure des dictionnaires et des files de priorités, qui sont des ensembles munis de fonctions d'accès spécifiques. La réalisation d'arbres ayant de bons comportements (arbres 2–4, bicolores, persistants, voir chapitre 6) requiert en revanche un soin certain.

3.2 Les structures linéaires

Une *liste linéaire*sur un ensemble E est une suite finie $(e_1, \ldots, e_n)$ d'éléments de E. La liste est *vide* si $n = 0$. Les éléments de la suite sont soit accessibles directement, par leur indice, soit indirectement, quand la liste est représentée par une suite d'objets chaînés. Dans les implémentations, un indice est représenté par une *place*, et on accède à l'élément de E qui figure à cette place par une fonction particulière appelée *contenu* (ou parfois *clé*) (voir figure 2.1). Cette représentation

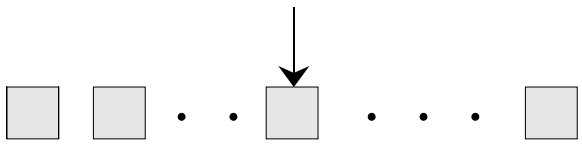


Figure 2.1: *Une liste et une place.*

sépare proprement la position d'un élément dans la liste de l'élément lui-même. De plus, elle permet de définir, sur une place, d'autres fonctions, comme la fonction successeur qui donne la place suivante.

On considère dans cette section plusieurs variantes des listes linéaires et quelques implémentations usuelles. Dans une liste, on distingue deux côtés, le début et la fin, et les opérations de manipulation peuvent se faire de chacun des deux côtés. Selon la nature des opérations autorisées, on définit des listes plus ou moins contraintes. Les *piles* sont des listes où l'insertion et la suppression ne se font que d'un seul et même côté. Les *files* permettent l'insertion d'un côté, et la suppression de l'autre. Les *files bilatères* permettent les insertions et suppressions des deux côtés; enfin, les *listes* ou *listes pointées* autorisent des insertions et suppressions

aussi à l'« intérieur » de la liste et pas seulement aux extrémités. De manière très parlante, une pile est appelée dans la terminologie anglo-saxonne une structure LIFO (« last-in first-out ») et une file une structure FIFO (« first-in first-out »).

3.2.1 Piles

Une *pile* est une liste linéaire où les insertions et suppressions se font toutes du même côté. Les noms spécifiques pour ces opérations sont *empiler* pour insérer et *dépiler* pour supprimer. Si la pile n'est pas vide, l'élément accessible est le *sommet* de la pile. La propriété fondamentale d'une pile est que la suppression annule l'effet d'une insertion : si on empile un élément, puis on dépile, on retrouve la pile dans l'état de départ. Les opérations sur une pile dont les éléments sont de type `élément` sont les suivantes :

PILEVIDE(p : pile) ;
: Crée une pile vide p.

SOMMET(p : pile) : élément ;
: Renvoie l'élément au sommet de la pile p ; bien sûr, p doit être non vide.

EMPILER(x : élément ; p : pile) ;
: Insère x au sommet de la pile p.

DÉPILER(p : pile) ;
: Supprime l'élément au sommet de la pile p ; la pile doit être non vide.

EST-PILEVIDE(p : pile) : booléen ;
: Teste si la pile p est vide.

Avant de discuter les implémentations, donnons un exemple d'utilisation de ce type.

Exemple. Le *tri par insertion* d'une suite $(x_1, \ldots, x_n)$ fonctionne comme suit : si $n = 0$, la suite est triée ; sinon, on trie $(x_1, \ldots, x_{n-1})$ en une suite croissante, puis on insère x_n dans cette suite en le comparant successivement aux éléments de la suite. Plus précisément, l'insertion de x dans une suite $L = (y_1, \ldots, y_m)$ se fait par comparaison :

- si L est vide, le résultat est (x) ; sinon
- si $x < y_1$, le résultat est $(x, y_1, \ldots, y_m)$; sinon
- le résultat est composé de y_1 et du résultat de l'insertion de x dans $(y_2, \ldots, y_m)$.

En représentant une suite triée par une pile, voici comment réaliser le tri par insertion :

```
procédure TRI-PAR-INSERTION(n, L);
  PILEVIDE(L);
  pour i de 1 à n faire INSÉRER(x_i, L);
  pour i de 1 à n faire
    x_i :=SOMMET(L); DÉPILER(L)
  finpour.
```

avec

```
procédure INSÉRER(x, L);
  si EST-PILEVIDE(L) oualors x <SOMMET(L) alors
    EMPILER(x, L)
  sinon
    y :=SOMMET(L); DÉPILER(L);
    INSÉRER(x, L); EMPILER(y, L)
  finsi;
  retourner(L).
```

(L'opérateur *oualors* est un *ou* séquentiel : la deuxième partie du test n'est évaluée que si la première rend la valeur faux. Voir aussi chapitre 1.) La procédure d'insertion demande un nombre d'opérations proportionnel à la longueur de L, et la réalisation du tri par insertion est donc en $O(n^2)$ opérations. L'*implémentation* la plus répandue d'une pile utilise un tableau p et un indice sp (indice de sommet de pile). L'indice indique l'emplacement du sommet de pile (une variante consiste à désigner le premier emplacement vide). Avec ces structures, l'implémentation des opérations est la suivante :

- PILEVIDE(p) se réalise par $sp := 0$;
- SOMMET(p) est : $p[sp]$;
- EMPILER(x, p) se réalise par $sp := sp + 1$; $p[sp] := x$;
- DÉPILER(p) se réalise par $sp := sp - 1$;
- EST-PILEVIDE(p) est : $sp = 0$?.

En toute rigueur, un test de débordement est nécessaire dans la procédure EMPILER, pour éviter de dépasser les bornes du tableau p.

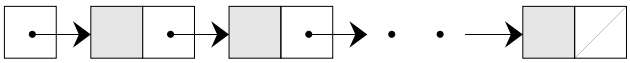


Figure 2.2: *Un pile représentée par une liste chaînée.*

L'implémentation par *liste simplement chaînée* est séduisante parce qu'elle permet une grande souplesse. Une pile est alors réalisée par un pointeur vers un couple formé de l'élément au sommet de pile, et d'un pointeur vers le couple suivant (voir figure 2.2). En Lisp par exemple, les opérations s'implémentent comme suit :

- PILEVIDE(p) se réalise par `(setq p nil)`;
- SOMMET(p) est `(car p)`;
- EMPILER(x, p) se réalise par `(setq p (cons x p))`;
- DÉPILER(p) se réalise par `(setq p (cdr p))`;
- EST-PILEVIDE(p) est `(equal p nil)`.

Chacune de ces opérations demande un temps constant.

3.2.2 Files

Une *file* est une liste linéaire où les insertions se font toutes d'un même côté et les suppressions toutes de l'autre côté. Les noms spécifiques pour ces opérations sont *enfiler* pour insérer et *défiler* pour supprimer. Les opérations sur les files sont syntaxiquement les mêmes que sur les piles; c'est par leur effet qu'elles diffèrent : l'élément supprimé est le premier arrivé dans la file. Ainsi, une file se comporte donc comme une file d'attente; on dit aussi *queue*, en anglais «first-in first-out»(FIFO). Les opérations sur une file dont les éléments sont de type `élément` sont les suivantes :

FILEVIDE(f : file);
: Crée une file vide f.

TÊTE(f : file) : élément;
: Renvoie l'élément en tête de la file f; bien sûr, f doit être non vide.

ENFILER(x : élément; f : file);
: Insère x à la fin de la file f.

DÉFILER(f : file);
: Supprime l'élément en tête de la file f; la file doit être non vide.

EST-FILEVIDE(f : file) : booléen;
: Teste si la file f est vide.

Deux *implémentations* des files sont courantes : l'une par un tableau avec deux variables d'indices, représentant respectivement le début et la fin de la file, l'autre par une liste circulaire. L'implémentation par tableau est simple, et utile quand on sait borner *a priori* le nombre d'insertions, comme cela se présente souvent dans des algorithmes sur les graphes. L'implémentation par *liste circulaire* est plus souple, mais un peu plus complexe (voir figure 2.3).

Dans cette implémentation, une file est repérée par un pointeur sur sa *dernière* place (si la file est vide, le pointeur vaut `nil`). La tête de la file est donc l'élément suivant, et peut être facilement supprimée; de même, l'insertion en fin de file se fait facilement.

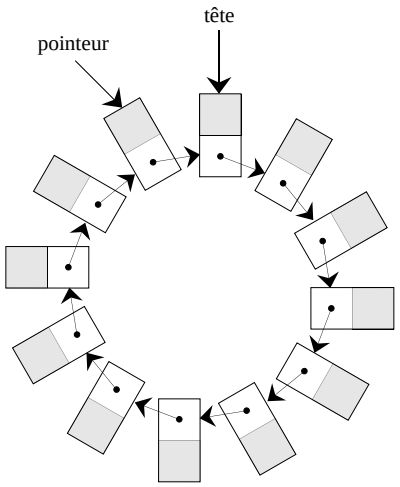


Figure 2.3: *La file* $(c, a, b, c, c, a, b, d, a, c, c, d)$.

Dans la réalisation ci-dessous, on utilise les types suivants :

```
TYPE
   file = ^bloc;
   bloc = RECORD
              cont: element;
              suiv: file
          END;
```

Le type `élément` est supposé connu. Les en-têtes de procédure sont :

```
PROCEDURE filevide  (VAR f : file);
FUNCTION  est_filevide (f : file): boolean;
PROCEDURE enfiler (x: element; VAR f : file);
PROCEDURE defiler (VAR f : file);
FUNCTION  tete (f : file): element;
```

La réalisation fait en plus appel à une fonction `faireplace` qui crée un bloc supplémentaire lorsque cela est nécessaire.

```
PROCEDURE filevide (VAR f: file);
   BEGIN
      f := NIL
   END;

FUNCTION est_filevide (f : file): boolean;
   BEGIN
      est_filevide := f  = NIL
   END;

FUNCTION faireplace (x: element): file;
   VAR
```

```
      q: file;
   BEGIN
      new(q); q^.cont := x; q^.suiv := NIL ;
      faireplace := q
   END;

PROCEDURE enfiler (x: element; VAR f : file);
   VAR
      q: file;
   BEGIN
      IF est_filevide(f) THEN BEGIN
         f  := faireplace(x); f ^.suiv := f
      END
      ELSE BEGIN
         q := faireplace(x);
         q ^.suiv := f ^.suiv; f ^.suiv := q; f  := q
      END
   END;

PROCEDURE defiler (VAR f : file);
   VAR
      q: file;
   BEGIN
      q := f ^.suiv;
      IF f  = f ^.suiv THEN
         f  := NIL
      ELSE
         f ^.suiv := f ^.suiv^.suiv;
      dispose(q)
   END;

FUNCTION tete (f : file): element;
   BEGIN
      tete := f ^.suiv^.cont
   END;
```

3.2.3 Listes

Une *liste* est une liste linéaire où les insertions et suppressions se font non seulement aux extrémités, mais aussi à l'intérieur de la liste. Les listes constituent un type de données très souple et efficace. Ainsi, on peut parcourir des listes (on ne parcourt ni les files ni les piles) sans les détruire, on peut rechercher un élément donné, on peut concaténer des listes, les scinder, etc. Dans la manipulation des listes, la distinction entre les places et leur contenu est importante. Une place est propre à une liste, c'est-à-dire ne peut pas être partagée entre plusieurs listes; en revanche, un élément peut figurer dans plusieurs listes, ou plusieurs fois dans une même liste.

Nous commençons par décrire les opérations de base sur les listes, et exprimons ensuite d'autres opérations à l'aide des opérations de base. Les 5 opérations d'accès sont les suivantes :

EST-LISTEVIDE(L : liste) : booléen ;
Teste si la liste L est vide.

CONTENU(p : place ; L : liste) : élément ;
Donne l'élément contenu dans la place p ; bien sûr, p doit être une place de L.

PREMIER(L : liste) : place ;
Donne la première place dans L ; indéfini si L est la liste vide.

SUIVANT(p : place ; L : liste) : place ;
Donne la place suivante ; indéfini si p est la dernière place de la liste.

EST-DERNIER(p : place ; L : liste) : booléen ;
Teste si la place p est la dernière de la liste L.

Les 4 opérations de construction et de modification sont :

LISTEVIDE(L : liste) ;
Crée une liste vide L, de longueur 0.

INSÉRERAPRÈS(x : élément ; p : place ; L : liste) ;
Crée une place contenant x et l'insère à la place suivant p dans L.

INSÉRERENTÊTE(x : élément ; L : liste) ;
Crée une place contenant x et l'insère comme premier élément de L.

SUPPRIMER(p : place ; L : liste) ;
Supprime l'élément qui se trouve à la place p dans la liste L : si avant la suppression la liste est $(e_1, \ldots, e_p, \ldots, e_n)$, alors après suppression, la liste est $(e_1, \ldots, e_{p-1}, e'_p, \ldots, e'_{n-1})$, avec $e'_j = e_{j+1}$ pour $j = p, \ldots, n-1$.

Au moyen de ces 9 opérations primitives, on peut en réaliser d'autres ; ainsi, TÊTE(L) =CONTENU(PREMIER(L), L) donne le premier élément d'une liste L. En voici d'autres :

CHERCHER(x : élément ; L : liste) : booléen ;
Teste si x figure dans la liste L.

TROUVER(x : élément ; p : place ; L : liste) : booléen ;
Teste si x figure dans la liste L. Dans l'affirmative, p contient la première place dont le contenu est x.

Cette fonction s'écrit comme suit :

```
fonction TROUVER(x : élément; p : place; L : liste) : booléen;
  si EST-LISTEVIDE(L) alors retourner (faux)
  sinon p :=PREMIER(L);
    tantque x ≠CONTENU(p) faire
      si EST-DERNIER(p) alors retourner (faux) sinon p :=SUIVANT(p)
    fintantque
  finsi;
  retourner (vrai).
```

On programme la fonction CHERCHER de la même manière.

Plusieurs *implémentations* des listes sont possibles; le choix de l'implémentation dépend des opérations effectivement utilisées. Si seules les opérations de base sont utilisées, on peut employer une implémentation simple; pour plus de souplesse, on utilisera une implémentation par liste doublement chaînée. C'est elle qui permet notamment de concaténer deux listes, ou de scinder une liste en temps constant.

L'implémentation par un *tableau* est simple : une place correspond à un indice, et la liste est conservée dans les premiers emplacements du tableau. Une variable supplémentaire tient à jour la longueur de la liste. Certaines des fonctions sont particulièrement simples à réaliser, comme *suivant* ou *contenu*. L'insertion et la suppression prennent un temps linéaire en fonction de la taille de la liste, puisqu'il faut déplacer toute la partie de la liste à droite de la position considérée. Dans les arbres par exemple, la liste des fils d'un sommet peut être rangée dans un tableau, si l'on connaît une majoration de leur nombre, comme c'est le cas pour les arbres *a*–*b* (voir chapitre 6).

L'implémentation par une *liste chaînée* est plus souple. Une place est un *pointeur* vers un couple formé de l'élément et d'un pointeur vers le couple suivant. Une liste est un pointeur vers un premier couple. Cette structure permet la réalisation de chacune des 9 opérations de base en un temps constant. En revanche, d'autres opérations, comme la concaténation de deux listes, prennent un temps non constant.

L'implémentation par une *liste doublement chaînée circulaire* est la plus souple, et permet aussi la réalisation efficace d'opérations supplémentaires sur les listes. Détaillons cette structure. Une place est un *pointeur* vers un triplet formé de l'élément et de deux pointeurs, l'un vers la place précédente, l'autre vers la place suivante. La liste est circulaire, de sorte que la première place est la place qui suit la dernière. Une liste est un pointeur vers le premier élément de la liste. Il a la valeur `nil` quand la liste est vide. En Pascal, les déclarations de type sont :

```
TYPE
   place = ^bloc;
   bloc = RECORD
             cont: element;
```

```
            suiv, prec: place
          END;
   liste = place;
```

où `élément` est un type défini par ailleurs. Les en-têtes de procédures et fonctions se déclarent comme suit :

```
(* fonctions d'acces *)
FUNCTION est_listevide (L: liste): boolean;
FUNCTION contenu (p: place; L: liste): element;
FUNCTION premier (L: liste): place;
FUNCTION suivant (p: place; L: liste): place;
FUNCTION est_dernier (p: place; L: liste): boolean;
(* constructeurs *)
PROCEDURE listevide (VAR L: liste);
PROCEDURE insererapres (x: element; p: place; L: liste);
PROCEDURE insererentete (x: element; VAR L: liste);
PROCEDURE supprimer (VAR p: place; VAR L: liste);
```

Voici l'implémentation des cinq fonctions d'accès :

```
IMPLEMENTATION (* des fonctions d'acces *)
FUNCTION est_listevide (L: liste): boolean;
        BEGIN
      est_listevide := L = NIL
        END;
FUNCTION contenu (p: place; L: liste): element;
        BEGIN
      contenu := p^.cont
        END;
FUNCTION premier (L: liste): place;
        BEGIN
      premier := L
        END;
FUNCTION suivant (p: place; L: liste): place;
        BEGIN
      suivant := p^.suiv
        END;
FUNCTION est_dernier (p: place; L: liste): boolean;
        BEGIN
      est_dernier := p^.suiv = L
        END;
```

Pour l'implémentation des fonctions de construction, nous utilisons quelques procédures auxiliaires.

```
PROCEDURE chainer (p, q: place);
        BEGIN
```

```
      q^.prec := p; p^.suiv := q
   END;
```

Cette procédure lie deux places (figure 2.4). Elle est utilisée pour créer, insérer et supprimer une place.

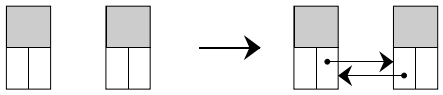


Figure 2.4: *Effet de* `chainer(p,q)`.

La procédure `dechainer` (figure 2.5) délie une place.

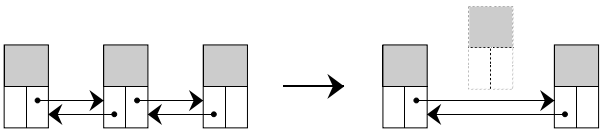


Figure 2.5: *Effet de* `dechainer(q)`.

```
FUNCTION faireplace (x: element): place;
   VAR
        q: place;
        BEGIN
      new(q); faireplace := q;
      q^.cont := x; chainer(q, q)
        END;
PROCEDURE enchainer (p, q: place);
        BEGIN
      chainer(q, p^.suiv); chainer(p, q)
        END;
PROCEDURE dechainer (q: place);
        BEGIN
      chainer(q^.prec, q^.suiv); dispose(q)
        END;
```

Les constructeurs s'écrivent :

```
IMPLEMENTATION (* des constructeurs *)
PROCEDURE listevide (VAR L: liste);
   BEGIN
      L := NIL
        END;
```

```
PROCEDURE insererapres (x: element; p: place; L: liste);
   VAR
      q: place;
   BEGIN
      q := faireplace(x); enchainer(p, q)
   END;
PROCEDURE insererentete (x: element; VAR L: liste);
   VAR
        q: place;
      BEGIN
   IF est_listevide(L) THEN
      L := faireplace(x)
   ELSE BEGIN
      q := faireplace(x);
      enchainer(L^.prec, q);
      L := q
      END
   END;
PROCEDURE supprimer (VAR p: place; VAR L: liste);
   VAR
        q: place;
        BEGIN
      q := p; p := p^.suiv;
      IF p = q THEN
         L := NIL
      ELSE BEGIN
         IF q = L THEN  L := p; dechainer(q)
        END;
      dispose(q)
   END;
```

Voici quelques fonctions ou opérations sur les listes qui sont faciles à réaliser avec les listes doublement chaînées :

DERNIER(L : liste) : place;
: Donne la dernière place dans L; indéfini si L est la liste vide.

SUIVANT-CYCLIQUE(L : liste) : place;
: Donne la place suivante dans L, et la première place de L si p est la dernière; indéfini si L est la liste vide.

PRÉCÉDENT(p : place; L : liste) : place;
: Donne la place précédente; indéfini si p est la première place de L.

INSÉRER-DIRIGÉ(x : élément; p : place; L : liste; d : direction);
: Insère x après la place p dans L si d =après, et avant p si d =avant.

INSÉRER-EN-QUEUE(x : élément; L : liste);
: Insère x en fin de liste.

CONCATÉNER(L_1, L_2 : liste);
: Accroche la liste L_2 à la fin de la liste L_1 : Si $L_1 = (e_1, \ldots, e_n)$ et $L_2 = (e'_1, \ldots, e'_m)$, alors la procédure retourne $L_1 = (e_1, \ldots, e_n, e'_1, \ldots, e'_m)$.

SCINDER(L : liste; p : place; L_1, L_2 : liste);
: Coupe la liste L en deux listes L_1 et L_2 telles que p soit la dernière place de L_1. La liste L est supposée non vide.

Voici une implémentation, en temps constant, des deux dernières opérations :

```
PROCEDURE concatener (VAR L1: liste; L2: liste);
   VAR
        q1, q2: place;
        BEGIN
      IF L1 = NIL THEN  L1 := L2
      ELSE IF L2 = NIL THEN
      ELSE BEGIN
         q1 := dernier(L1); q2 := dernier(L2);
         chainer(q1, L2); chainer(q2, L1)
          END
   END;
PROCEDURE scinder (L: liste; p: place; VAR L1, L2: liste);
   VAR
        q: place;
        BEGIN
      L1 := L;
      IF est_dernier(p, L) THEN L2 := NIL
      ELSE BEGIN
         L2 := p^.suiv; q := dernier(L);
         chainer(p, L1);        chainer(q, L2)
        END
        END;
```

Le *parcours* d'une liste doublement chaînée, avec l'exécution d'une instruction notée I pour chaque place, se fait en temps linéaire, par :

si non EST-LISTEVIDE(L) alors
  p :=PREMIER(L);
  répéter
    I;
    p :=SUIVANT-CYCLIQUE(p)
  jusqu'à p =PREMIER(L)
finsi.

Remarquons que, dans l'implémentation par une liste simplement chaînée, chaque place pointe en fait sur une liste, à savoir sur le reste de la liste commençant à cette

place. Naturellement, la place suivant la dernière est alors la liste vide, en général représentée par `nil`. On peut donc parcourir une telle liste par l'instruction

tantque $p \neq$ `nil` faire I ; p :=SUIVANT(p) fintantque.

Proposition 2.1. *Les opérations de liste* PREMIER, DERNIER, SUIVANT, CONCATÉNER, SCINDER *se réalisent en temps* $O(1)$. *Les opérations* CHERCHER, TROUVER, SUPPRIMER *et le parcours d'une liste se réalisent en temps* $O(n)$, *où* n *est la longueur de la liste.*

3.3 Arbres

On considère d'abord l'implémentation des arbres binaires, puis des arbres plus généraux. Au chapitre suivant, nous étudions plus en détail ces familles d'arbres, et nous nous bornons ici aux structures de données.

3.3.1 Arbres binaires

La représentation la plus naturelle des arbres binaires s'appuie sur leur définition récursive : un arbre binaire est soit l'arbre vide, soit formé d'une *racine* et de deux arbres binaires disjoints, appelés sous-arbres *gauche* et *droit* (voir section 4.3.4). Lorsque l'arbre est étiqueté, ce qui est le cas notamment pour les arbres binaires de recherche, un sommet comporte un champ supplémentaire, qui est son *contenu* ou sa *clé*. Nous commençons par décrire les opérations de base sur les arbres binaires étiquetés, et exprimons ensuite d'autres opérations à l'aide de ces opérations de base. Les 5 opérations d'accès sont les suivantes :

EST-ARBREVIDE(a : arbre) : booléen ;
: Teste si l'arbre a est vide.

CONTENU(s : sommet) : élément ;
: Donne l'élément contenu dans le sommet s.

RACINE(a : arbre) : sommet ;
: Donne le sommet qui est la racine de a ; indéfini si a est l'arbre vide.

SOUS-ARBRE-GAUCHE(a : arbre) : arbre ;
: Donne le sous-arbre gauche ; indéfini si a est l'arbre vide.

SOUS-ARBRE-DROIT(a : arbre) : arbre ;
: Donne le sous-arbre droit ; indéfini si a est l'arbre vide.

Les opérations de construction et de modification sont :

ARBREVIDE(a : arbre) ;
: Crée un arbre vide a.

FAIRE-ARBRE(x : élément ; g, d : arbre) : arbre ;

Crée un sommet qui contient x, et retourne l'arbre ayant ce sommet pour racine, et g et d comme sous-arbres gauche et droit.

FIXER-CLÉ(x : élément ; a : arbre) ;
Le contenu de la racine de a devient x. Indéfini si a est l'arbre vide.

FIXER-GAUCHE(g : arbre ; a : arbre) ;
Remplace le sous-arbre gauche de a par g. Indéfini si a est l'arbre vide.

FIXER-DROIT(d : arbre ; a : arbre) ;
Remplace le sous-arbre droit de a par d. Indéfini si a est l'arbre vide.

Les trois dernières opérations peuvent être réalisées à l'aide de FAIRE-ARBRE : par exemple, FIXER-CLÉ(x, a) est équivalent à FAIRE-ARBRE($x, G(a), D(a)$). Pour faciliter l'écriture et la lecture, nous abrégeons SOUS-ARBRE-GAUCHE en G et SOUS-ARBRE-DROIT en D. Si nous les introduisons explicitement, c'est par souci d'efficacité dans les implémentations : il n'y a pas lieu de créer un nouveau sommet si l'on veut simplement changer son contenu. A l'aide de ces opérations de base, on peut définir de nombreuses autres opérations. Il est par exemple commode de disposer de l'abréviation :

CLÉ(a : arbre) : élément ;
Donne le contenu de la racine de a, indéfini si a est vide.

ainsi que d'opérations sur les feuilles, comme :

EST-FEUILLE(a : arbre) : booléen ;
Teste si la racine de l'arbre a est une feuille ; indéfini si a est vide.

FAIRE-FEUILLE(x : élément) : arbre ;
Crée un arbre réduit à un seul sommet qui contient x.

Bien entendu, FAIRE-FEUILLE(x) s'obtient par ARBREVIDE(g), ARBREVIDE(d), suivi de FAIRE-ARBRE(x, g, d), et EST-FEUILLE(a) est égal à la conjonction de
EST-ARBREVIDE($G(a)$) et EST-ARBREVIDE($D(a)$)

Dans certains cas, il est commode de disposer de FILS-GAUCHE et de FILS-DROIT, procédures qui rendent la racine du sous-arbre gauche, respectivement droit.

3.3.2 Dictionnaires et arbres binaires de recherche

Un *dictionnaire* est un type de données opérant sur les éléments d'un ensemble totalement ordonné, appelés les *clés* et doté des opérations suivantes :

DICOVIDE(d : dictionnaire) ;
Crée un dictionnaire d vide.

EST-DICOVIDE(d : dictionnaire) : booléen;
Teste si le dictionnaire d est vide.

CHERCHER(x : clé; d : dictionnaire) : booléen;
Teste si la clé x figure dans le dictionnaire d.

INSÉRER(x : clé; d : dictionnaire);
Insère x dans le dictionnaire d.

SUPPRIMER(x : clé; d : dictionnaire);
Supprime x dans le dictionnaire d.

Les arbres binaires de recherche se prêtent particulièrement bien à l'implémentation des dictionnaires. On les verra en détail au chapitre 6. Il suffit ici de savoir qu'un arbre binaire de recherche est un arbre binaire dont chaque sommet est muni d'une clé prise dans un ensemble totalement ordonné. De plus, pour chaque sommet de l'arbre, les clés contenues dans son sous-arbre gauche sont strictement inférieures à la clé du sommet considéré et cette clé est elle-même strictement inférieure aux clés contenues dans le sous-arbre droit. En d'autres termes, un parcours symétrique de l'arbre, comme décrit au chapitre 4, donne une liste des clés en ordre croissant. Donnons d'abord les en-têtes des opérations, qui sont les mêmes que dans un arbre :

CHERCHER(x : clé; a : arbre) : booléen;
Teste si x figure dans l'arbre a.

INSÉRER(x : clé; a : arbre);
Insère x dans l'arbre a selon l'algorithme exposé ci-dessous; on suppose que x ne figure pas dans a, et on crée donc un sommet dont la clé est x.

SUPPRIMER(x : clé; a : arbre);
Supprime la clé x et un sommet de l'arbre a selon l'algorithme exposé ci-dessous; on suppose que x figure dans a.

Voici la réalisation de la recherche dans un arbre :

```
fonction CHERCHER(x : clé; a : arbre) : booléen;
  si EST-ARBREVIDE(a) alors retourner (faux)
  sinon si x=CLÉ(a) alors retourner (vrai)
  sinon si x <CLÉ(a) alors
    retourner CHERCHER(x, G(a))
  sinon
    retourner CHERCHER(x, D(a)).
```

Pour l'insertion, on procède de manière similaire :

```
procédure INSÉRER(x : clé; a : arbre);
  si EST-ARBREVIDE(a) alors a :=FAIRE-FEUILLE(x)
  sinon si x <CLÉ(a) alors
    INSÉRER(x, G(a))
  sinon
    INSÉRER(x, D(a)).
```

(La réalisation, en Pascal par exemple, de cette procédure pose quelques problèmes techniques car on ne peut transmettre en référence le résultat de l'évaluation d'une fonction...) La suppression est plus difficile à réaliser parce qu'il faut conserver l'ordre sur l'arbre résultat. Soit x la clé à supprimer, et soit s le sommet dont x est le contenu. Si s est une feuille, il suffit de la supprimer. Si s n'a qu'un seul fils, on le remplace par son fils unique. Si s a deux fils, on cherche son descendant t dont le contenu le précède dans l'ordre infixe : c'est le sommet le plus à droite dans son sous-arbre gauche. On supprime ce sommet (qui n'a pas de fils droit), après avoir remplacé le contenu de s par le contenu de t. (On pourrait aussi bien, symétriquement, remplacer le contenu de s par celui du sommet qui lui succède dans l'ordre infixe.)

Voici une réalisation de cet algorithme :

```
procédure SUPPRIMER(x : clé; a : arbre );
  si x <CLÉ(a) alors SUPPRIMER(x, G(a))
  sinon si x >CLÉ(a) alors SUPPRIMER(x, D(a))
  sinon si EST-ARBREVIDE(G(a)) alors a := D(a)
  sinon si EST-ARBREVIDE(D(a)) alors a := G(a)
  sinon FIXER-CLÉ(SUPPRIMER-MAX(G(a)), a).
```

L'appel à la fonction SUPPRIMER-MAX se fait donc dans le cas où l'arbre a deux sous-arbres non vides, et où le contenu de la racine de a est à supprimer. La fonction SUPPRIMER-MAX réalise à la fois la suppression du sommet de plus grande clé parmi les descendants, et retourne le contenu de ce sommet.

```
fonction SUPPRIMER-MAX(a : arbre ) : clé;
  si EST-ARBREVIDE(D(a)) alors
    SUPPRIMER-MAX :=CLE(a); ARBREVIDE(a)
  sinon SUPPRIMER-MAX :=SUPPRIMER-MAX(D(a)).
```

Plusieurs *implémentations* des arbres sont possibles; la plus souple utilise des pointeurs. A chaque sommet on associe deux pointeurs, l'un vers le sous-arbre

gauche, l'autre vers le sous-arbre droit. Un arbre est donné par un pointeur vers sa racine. Un sommet comporte en plus un champ qui contient la clé associée. On est donc conduit à définir les types suivants :

```
TYPE
   arbre = ^sommet;
   sommet = RECORD
               val: element;
               g, d: arbre
            END;
```

3.3.3 Arborescences

Une *arborescence ordonnée* s'implémente en associant, à chaque sommet, la liste linéaire de ses fils. Lorsque le nombre de fils est borné *a priori*, comme c'est le cas par exemple pour les arbres a–b (voir chapitre 6), on utilisera un tableau pour représenter cette liste, sinon une liste chaînée.

Ainsi, à chaque sommet est associé un couple (premier fils, frère droit). Cette façon d'implémenter les arborescences ordonnées revient en fait à se ramener aux arbres binaires. Plus formellement, on associe à chaque sommet s d'une arborescence ordonnée un sommet $\beta(s)$ d'un arbre binaire dont le fils gauche est l'image, par β, du premier fils de s, et le fils droit l'image du frère suivant de s si s a un frère, l'arbre vide sinon. La figure 3.1 explique la construction.

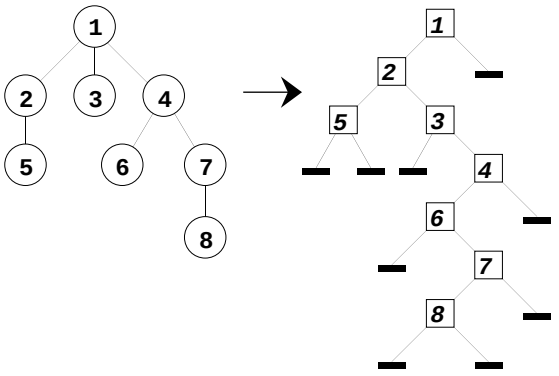


Figure 3.1: *Une arborescence représentée comme arbre binaire.*

Les autres types d'arbres ou d'arborescences s'implémentent comme des arborescences ordonnées.

3.4 Files de priorité

Les files de priorité constituent un type de données intéressant, et qui illustre bien comment une hiérarchie de structures peut permettre d'implémenter un type de données.

Une *file de priorité* est un type de données opérant sur des clés, donc sur les éléments d'un ensemble totalement ordonné, et muni des opérations suivantes :

FILEPVIDE(f : fileP);
: Crée une file de priorité f vide.

EST-FILEPVIDE(f : fileP) : booléen;
: Teste si la file de priorité f est vide.

MINIMUM(f : fileP) : clé;
: Donne la plus petite clé contenue dans f.

SUPPRIMER-MIN(f : fileP);
: Supprime la plus petite clé de f ; indéfini si f est vide.

INSÉRER(x : clé; f : fileP);
: Insère x dans la file de priorité f.

Il est commode de disposer de la combinaison de MINIMUM et de SUPPRIMER-MIN sous la forme :

EXTRAIRE-MIN(f : fileP) : clé;
: Donne la plus petite clé de f, et la supprime de f ; indéfini si f est vide.

Fréquemment, et notamment dans les algorithmes de tri, on convient qu'une même clé peut figurer plusieurs fois dans une file.

Si les clés sont toutes distinctes, on peut réaliser une file de priorité à l'aide d'un arbre binaire de recherche. L'efficacité des opérations dépend alors de l'efficacité de l'insertion et de la suppression de la plus petite clé dans un arbre binaire de recherche. On verra au chapitre 6 comment ces opérations peuvent être réalisées en temps logarithmique, moyennant un rééquilibrage de l'arbre binaire. Nous présentons ici une autre implémentation, au moyen d'un tas.

Un *tas* est un arbre tournoi parfait, défini ci-dessous. On verra comment implémenter un tas très simplement à l'aide d'un tableau.

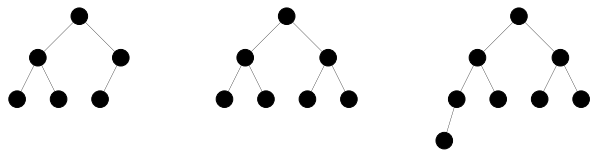


Figure 4.1: *Les arbres parfaits à* $6, 7$*, et* 8 *sommets.*

Un *arbre parfait* est un arbre binaire dont toutes les feuilles sont situées sur deux niveaux au plus, l'avant-dernier niveau est complet, et les feuilles du dernier

niveau sont regroupées le plus à gauche possible. La figure 4.1 donne des exemples d'arbres parfaits. Notons qu'il n'y a qu'un seul arbre parfait à n sommets, pour chaque entier n.

Un *arbre tournoi* est un arbre binaire dont les sommets sont munis d'une clé, et tel que pour tout sommet autre que la racine, la clé du sommet est plus grande que celle de son père. En d'autres termes, les clés sur un chemin croissent de la racine vers une feuille.

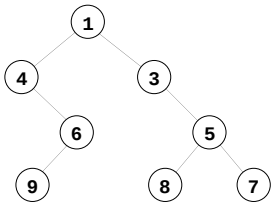


Figure 4.2: *Un arbre tournoi.*

La figure 4.2 montre un arbre tournoi, et la figure 4.3 un tas (arbre tournoi parfait).

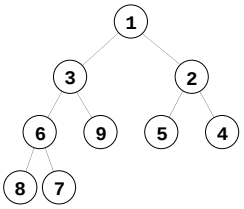


Figure 4.3: *Un tas (tournoi parfait).*

Sur les arbres parfaits, on utilise certaines des opérations sur les arbres binaires : EST-ARBREVIDE, ARBREVIDE, RACINE, PÈRE, FILS-GAUCHE, FILS-DROIT. Par ailleurs, on définit les opérations spécifiques suivantes :

EST-FEUILLE, qui teste si le sommet en argument est une feuille ;

DERNIÈRE-FEUILLE, qui donne la feuille la plus à droite du dernier niveau de l'arbre ;

CRÉER-FEUILLE, qui crée une feuille au dernier niveau de l'arbre, ou entame un niveau supplémentaire si le dernier niveau est plein.

SUPPRIMER-FEUILLE, qui supprime cette feuille.

Ces opérations conservent donc la perfection d'un arbre. Pour les arbres tournoi, la fonction CONTENU donne le contenu (la clé) d'un sommet, et on fait appel à une procédure supplémentaire, ÉCHANGER-CONTENU(p, q) qui échange les clés des sommets p et q.

Etudions comment on implémente les opérations des files de priorités au moyen des opérations des tas. La détermination du minimum est évidente : c'est le contenu de la racine du tas. Elle se fait donc en temps constant.

Pour l'*insertion* d'une clé x, on crée d'abord une feuille contenant cet élément (et qui est donc ajoutée au dernier niveau du tas) ; ensuite on compare son contenu à celui de son père ; s'il est plus petit, on l'échange, et on continue en progressant vers la racine du tas. Voici comment cette procédure se réalise :

```
procédure INSÉRERTAS(x : clé; a : tas);
  q :=CRÉER-FEUILLE(x, a);
  tantque q ≠RACINE(a) etalors CONTENU(PÈRE(q)) >CONTENU(q) faire
    ÉCHANGER-CONTENU(q,PÈRE(q)); q :=PERE(q)
  fintantque.
```

L'*extraction* de la plus petite clé se fait sur un canevas similaire :

```
fonction EXTRAIREMIN(a : tas);
  EXTRAIREMIN :=CONTENU(RACINE(a));
  ÉCHANGER-CONTENU(RACINE(a),DERNIÈRE-FEUILLE(a));
  SUPPRIMER-FEUILLE(a);
  si EST-ARBREVIDE(a) est faux alors
    p :=RACINE(a);
    tantque non EST-FEUILLE(p) faire
      f :=FILS-MIN(p, a);
      si CONTENU(f) <CONTENU(p)
      alors ÉCHANGER-CONTENU(f, p) sinon exit;
      p := f
    fintantque
  finsi.
```

On remplace donc le contenu de la racine par le contenu de la dernière feuille. Cette feuille est supprimée. Puis, on descend de la racine vers les feuilles pour mettre la clé à une place convenable si elle ne l'est pas, c'est à dire si elle est plus grande que l'une des clés des fils. Si le sommet en question a deux fils, on échange la clé du père avec la plus petite des clés de ses fils. On obtient ainsi une correction locale ; on poursuit l'opération tant que nécessaire. Revenons à l'exemple de la figure 4.3.

La suppression de la clé 1 amène la clé 7 à la racine. La descente qui s'ensuit est illustrée sur la figure 4.4. La fonction FILS-MIN(p, a) retourne le fils de p dont le contenu est le plus petit. Elle s'écrit :

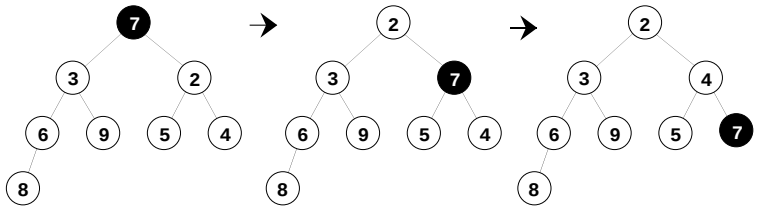


Figure 4.4: *Mise en place de la clé* 7 *par comparaison aux fils.*

```
fonction FILS-MIN(p : sommet; a : tas) : sommet;
  g :=FILS-GAUCHE(p);
  si g =DERNIÈRE-FEUILLE(a) alors FILS-MIN:= g
  sinon d :=FILS-DROIT(p);
    si CONTENU(g) <CONTENU(d) alors FILS-MIN:= g sinon FILS-MIN:= d
  finsi.
```

Il est clair que la hauteur d'un tas est logarithmique en fonction du nombre de ses sommets, donc que l'insertion et l'extraction d'une clé prennent un temps logarithmique, sous réserve que les opérations élémentaires s'implémentent en temps constant.

Comme nous l'avons annoncé plus haut, il existe une implémentation très efficace des tas à l'aide d'un couple formé d'un tableau a et d'un entier n donnant le nombre de clés présents dans le tas. Les sommets du tas étant numérotés niveau par niveau de la gauche vers la droite, chaque sommet est représenté par l'indice d'une clé d'un tableau a. Le contenu d'un sommet p est rangé dans $a[p]$, et est donc accessible en temps constant. Passons en revue les opérations; en face de leur nom, nous indiquons leur réalisation :

EST-ARBREVIDE	$n = 0$?
ARBREVIDE	$n := 0$
RACINE	1
PÈRE(p)	$\lfloor p/2 \rfloor$
FILS-GAUCHE(p)	$2p$
FILS-DROIT(p)	$2p + 1$

Les autres opérations se réalisent aussi simplement :

EST-FEUILLE(p)	$p > \lfloor n/2 \rfloor$?
DERNIÈRE-FEUILLE	n
CRÉER-FEUILLE(x, a)	$n := n + 1; a[n] := x$
SUPPRIMER-FEUILLE	$n := n - 1$
CONTENU(p)	$a[p]$

Seule la procédure ÉCHANGER-CONTENU(p, q) qui échange les clés des sommets p et q, demande trois instructions. L'efficacité pratique de la réalisation des files de priorités est évidemment améliorée si l'on travaille directement sur le tableau, et si l'on remplace les appels de procédures par les instructions correspondantes. Voici une implémentation concrète. On définit les types :

```
TYPE
   sommet = 1..taillemax;
   tas = RECORD
            n: integer;
            a: ARRAY[sommet] OF cle
         END;
```

où `taillemax` et `cle` sont prédéfinis. La traduction des procédures ci-dessus se fait littéralement, comme suit :

```
PROCEDURE inserertas (x: cle; VAR t: tas);
   VAR
      q: sommet;
   BEGIN
      WITH t DO BEGIN
         n := n + 1; a[n] := x; q := n;
         WHILE (q <> 1) AND (a[q DIV 2] > a[q]) DO BEGIN
            echanger(a[q], a[q DIV 2]); q := q DIV 2
         END
      END;
   END;

FUNCTION extrairemin (VAR t: tas): cle;
   VAR
      p, f: sommet;
   FUNCTION fils_min (q: sommet): sommet;
      VAR
         g: sommet;
      BEGIN
         WITH t DO BEGIN
            g := 2 * q;
            fils_min := g;
            IF (g < n) AND (a[g] > a[g + 1]) THEN
               fils_min := g + 1
         END
      END; (* de "fils_min" *)
   BEGIN
      WITH t DO BEGIN
         extrairemin := a[1];
         echanger(a[1], a[n]);
         n := n - 1;
```

```
        IF n > 0 THEN BEGIN
           p := 1;
           WHILE 2 * p <= n DO BEGIN
              f := fils_min(p);
              IF a[f] < a[p] THEN
                 echanger(a[f], a[p])
              ELSE exit(extrairemin);
              p := f
           END;
        END
     END
  END; (* de "extrairemin" *)
```

Bien entendu, `echanger` est une procédure d'échange de deux clés.

3.5 Gestion des partitions

3.5.1 Le problème « union-find »

Dans cette section, nous considérons le problème de la gestion efficace d'une partition d'un ensemble $U = \{1, \ldots, N\}$. Les opérations considérées sont

TROUVER(x)
: Donne le nom de la classe à laquelle appartient l'élément x de U.

UNIR(A, B, C)
: Donne une nouvelle classe, de nom C, qui est la réunion des classes de noms A et B.

Ce problème est appelé le problème de l'union et recherche (« union-find problem » en anglais) ; il apparaît dans de nombreuses situations, et notamment dans la recherche d'un arbre couvrant minimum considéré plus loin.

Dans la suite, nous supposerons que la partition initiale est la partition la plus fine, dont les classes sont composées d'un seul élément. Les noms des classes sont des entiers dans $\{1, \ldots, N\}$, et au début, le nom de la classe $\{i\}$ est i.

Une solution simple du problème est de représenter la partition par un tableau *classe* de taille N qui, pour un élément $x \in U$ contient le nom *classe*$[x]$ de la classe contenant x. L'opération TROUVER se réduit au calcul de *classe*$[x]$. L'opération UNIR(A, B, C) est simple mais plus longue : on parcourt le tableau *classe* et on remplace tous les A et B par C.

La solution que nous considérons maintenant en détail utilise une structure plus élaborée. Chaque classe est représentée par une arborescence, et la partition est représentée par une forêt. Le nom d'une classe est la racine de son arborescence (voir figure 5.1).

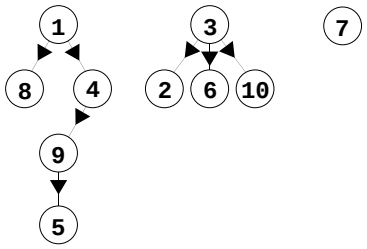


Figure 5.1: *Partition en trois classes, de noms* 1, 3, *et* 7.

La forêt est rangée dans un tableau qui, pour chaque élément x autre qu'une racine, donne le père de x. Le père d'une racine est par exemple le nombre 0. Avec cette représentation, on a :

TROUVER(x)
: Donne le nom de la racine de l'arborescence contenant x.

UNIR(x, y, z)
: Prend les arborescences de racines x et y, et en fait une arborescence de racine z; z est soit égal à x, soit égal à y.

3.5.2 Union pondérée et compression des chemins

Clairement, UNIR(x, y, z) se réalise en temps constant. Le temps d'exécution d'un TROUVER(x) est proportionnel à la profondeur de x. Si au total n unions sont exécutées, ce temps est au plus $O(n)$. On peut réduire considérablement ce temps si l'on parvient à éviter la formation d'arborescences trop «filiformes». Pour cela, il convient de réaliser l'union de manière plus soignée. On utilise à cet effet la règle dite de l'*union pondérée*. Lors de l'exécution de UNIR(x, y, z), la racine de la plus petite des deux arborescences est accrochée comme fils à la racine de la plus grande. Appelons *taille* de x le nombre de sommets de l'arborescence de racine x. Alors l'union pondérée se réalise en temps constant par la procédure que voici :

```
procédure UNION-PONDÉRÉE(x, y, z);
   si taille[x] <taille[y] alors
      z := y; père[x] := y
   sinon
      z := x; père[y] := x
   finsi;
   taille[z] :=taille[x]+taille[y].
```

Nous allons voir que cette règle est très efficace. Notons $t_i(x)$ et $h_i(x)$ la taille et la hauteur du sommet x après la i-ième opération d'union pondérée. On a $t_0(x) = 1$,

$h_0(x) = 0$. Si la i-ième union pondérée est UNIR(x, y, z), et si $t_{i-1}(x) < t_{i-1}(y)$, alors $z = y$, et on a

$$\begin{array}{ll} t_i(x) = t_{i-1}(x), & t_i(y) = t_{i-1}(x) + t_{i-1}(y) \\ h_i(x) = h_{i-1}(x), & h_i(y) = \max(1 + h_{i-1}(x), h_{i-1}(y)) \end{array}$$

Lemme 5.1. *Si l'on utilise la règle de l'union pondérée, alors*

$$t_i(x) \geq 2^{h_i(x)}$$

pour tout $x \in U$. En particulier, après n unions, on a $h_n(x) \leq \log(n+1)$.

Preuve. Si $h_i(x) = 0$, alors $t_i(x) = 1$. Supposons donc $h_i(x) \geq 1$. Alors x a un fils y avec $h_i(y) = h_i(x) - 1$. Considérons l'opération d'union pondérée qui a fait de y un fils de x. Si cette opération est la j-ième, avec $j \leq i$, alors $t_{j-1}(x) \geq t_{j-1}(y)$, et donc $t_j(y) = t_{j-1}(y)$ et $t_j(x) \geq 2 \cdot t_j(y)$. Ensuite, comme y n'est plus racine d'une arborescence, sa taille ne varie plus. En revanche, la taille de x peut augmenter. De même, la hauteur de y ne varie plus, donc $h_i(y) = h_{j-1}(y)$. Par récurrence, on a $t_{j-1}(y) \geq 2^{h_{j-1}(y)}$, donc

$$t_i(x) \geq t_j(x) \geq 2 \cdot t_j(y) \geq 2 \cdot 2^{h_i(y)} = 2^{h_i(x)}.$$

Comme n unions ne créent que des arborescences d'au plus $n+1$ sommets, on a $t_n(x) \leq n+1$ pour tout x, d'où la seconde inégalité. ∎

Proposition 5.2. *Avec la représentation par forêt et la règle de l'union pondérée, une suite de $n-1$ opérations* UNION-PONDÉRÉE *et de m opérations* TROUVER *se réalise en temps $O(n + m \log n)$.*

Preuve. Chaque union pondérée prend un temps $O(1)$. De plus, chaque arborescence obtenue durant les $n-1$ unions a une hauteur au plus $\log n$, donc une opération TROUVER prend un temps $O(\log n)$. ∎

Une autre façon de diminuer le coût des TROUVER est la *compression des chemins*. Si l'on exécute un TROUVER(x), on parcourt un chemin $x_0, x_1, \ldots, x_\ell$ de $x = x_0$ vers la racine $r = x_\ell$ de l'arborescence contenant x. On peut comprimer ce chemin en faisant un deuxième parcours du chemin pendant lequel on fait, de chaque sommet x_i, $0 \leq i < \ell$, un fils de la racine r (voir figure 5.2). Même si cette compression ne réduit pas le coût de ce TROUVER (en fait, elle en double le coût), les coûts des TROUVER subséquents qui s'appliquent aux sommets dans les sous-arbres des x_i sont fortement réduits. Voici la réalisation du TROUVER avec compression :

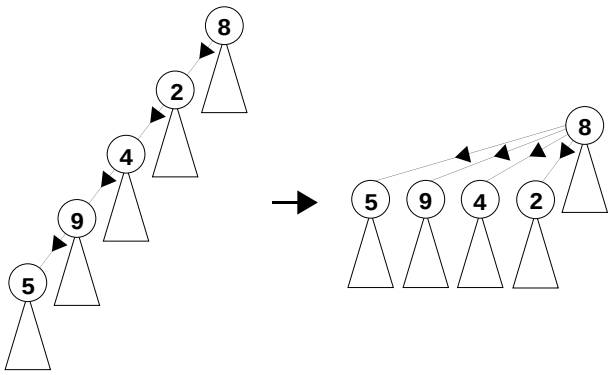


Figure 5.2: *Compression d'un chemin.*

```
procédure TROUVER-AVEC-COMPRESSION(x);
  r := x;
  tantque r n'est pas racine faire r :=père[r] fintantque;
  tantque x ≠ r faire
    y :=père[x]; père[x] := r; x := y
  fintantque;
  retourner(r).
```

La compression des chemins seule conduit déjà à un algorithme efficace. La combinaison de la compression des chemins et de l'union pondérée fournit un algorithme très efficace, dont le temps d'exécution est presque linéaire. Pour les valeurs des paramètres que l'on rencontre en pratique, l'algorithme est linéaire.

Pour pouvoir présenter l'analyse de l'algorithme, nous avons besoin d'une notation. Soit $A : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ la fonction définie par

$$\begin{array}{ll} A(i,0) = 1 & \text{pour } i \geq 1; \\ A(0,j) = 2j & \text{pour } j \geq 0; \\ A(i+1,j+1) = A(i,A(i+1,j)) & \text{pour } i,j \geq 0. \end{array}$$

Cette fonction est une variante de la fonction d'Ackermann. Elle croît très rapidement, et n'est pas primitive récursive. Voici une table des premières valeurs de A :

$i\backslash j$	0	1	2	3	4	5
0	0	2	4	6	8	10
1	1	2	4	8	16	32
2	1	2	4	16	65536	2^{65536}
3	1	2	4	65536	$2^{\cdot^{\cdot^{\cdot^{2}}}}\Big\}65536$ fois	

Soit α la fonction définie pour $m \geq n$ par

$$\alpha(m,n) = \min\{z \geq 1 \mid A(z, 4\lceil m/n \rceil) > \log n\}$$

Cette fonction est une sorte d'inverse de la fonction A, et croît très lentement. Pour $m \geq n$, on a $\alpha(m,n) \leq \alpha(n,n)$. On a $\alpha(n,n) = 1$ pour $1 \leq n < 2^{16}$, et $\alpha(n,n) = 2$ pour $2^{16} \leq n < 2^{65536}$. Pour des valeurs de m et n apparaissant en pratique, on a donc $\alpha(m,n) \leq 2$.

Nous allons prouver le résultat suivant, dont la démonstration n'est pas facile, et peut être omise en première lecture :

Théorème 5.3. *Avec les règles de l'union pondérée et de la compression des chemins, une suite de $n-1$ opérations* UNIR *et de m opérations* TROUVER *($m \geq n$) se réalise en temps $O(m \cdot \alpha(m,n))$.*

3.5.3 Preuve du théorème

Note 3.5.1

Pour démontrer le théorème 5.3, nous reformulons le problème. Soit T une forêt obtenue par $n-1$ opérations UNIR avec la règle de l'union pondérée. Nous autorisons que la forêt soit ensuite modifiée par ce que nous appelons des « TROUVER partiels ». Un TROUVER partiel dans une arborescence parcourt un chemin $x_0, x_1, \ldots, x_k$ d'un sommet x_0 à un sommet x_k qui n'est pas nécessairement la racine. Ensuite, chacun des sommets $x_0, x_1, \ldots, x_{k-1}$ est fait fils du sommet x_k. Le coût du TROUVER partiel est k.

Il est facile de voir qu'une suite arbitraire de $n-1$ opérations UNIR et de m TROUVER peut être simulée par une suite de $n-1$ UNIR *suivie* d'une suite de m TROUVER partiels, et que le coût de la simulation est le coût de la suite originelle. En effet, les UNIR de la simulation sont juste les UNIR de la suite de départ, et chaque TROUVER de la suite de départ est remplacé par un TROUVER partiel avec les mêmes extrémités. Il suffit donc de démontrer la majoration pour une suite de $n-1$ opérations UNIR suivies de m TROUVER partiels.

Soit T la forêt obtenue après les $n-1$ opérations UNIR, et soit $h(x)$ la hauteur d'un sommet x dans T. Pour chaque arc (x,y) de T (les arcs sont orientés en direction de la racine), on a $h(x) < h(y)$.

Le coût total des TROUVER partiels est proportionnel au nombre d'arcs parcourus lors de tous les TROUVER partiels. Soit F l'ensemble de ces arcs. Nous allons montrer que $|F| = O(m\alpha(m,n))$ en répartissant les arcs en groupes, et en évaluant le nombre d'arcs dans chaque groupe. La répartition des arcs de F se fait en fonction de la hauteur de leurs extrémités. C'est pourquoi on définit, pour $i, j \geq 0$, les ensembles de sommets

$$G_{i,j} = \{x \mid A(i,j) \leq h(x) < A(i,j+1)\}.$$

Ainsi, on a par exemple

$$\begin{array}{ll} G_{0,0} = \{x \mid 0 \leq h(x) \leq 1\} & \\ G_{k,0} = \{x \mid h(x) = 1\} & \text{pour } k \geq 1 \\ G_{k,1} = \{x \mid 2 \leq h(x) \leq 3\} & \text{pour } k \geq 0 \end{array}$$

Lemme 5.4. *Pour $k \geq 0$ et $j \geq 1$, on a $|G_{k,j}| \leq 2n/2^{A(k,j)}$.*

Preuve. Soit $\ell \geq 1$ un entier avec $A(k,j) \leq \ell < A(k,j+1)$, et comptons le nombre de sommets x tels que $h(x) = \ell$. Deux sommets distincts de hauteur ℓ sont racines de sous-arbres disjoints, et la taille du sous-arbre est, d'après le lemme 5.1, au moins 2^ℓ. Il y a donc au plus $n/2^\ell$ sommets de hauteur ℓ. Par conséquent,

$$|G_{k,j}| \leq \sum_{\ell=A(k,j)}^{A(k,j+1)-1} \frac{n}{2^\ell} \leq \frac{2n}{2^{A(k,j)}}$$ ∎

L'ensemble F des arcs des TROUVER partiels est réparti en classes N_k pour $k = 0, \ldots, z+1$, où z est un paramètre fixé ultérieurement et une classe N' par :

$$N_k = \{(x,y) \in F \mid k = \min\{i \mid \exists j : x, y \in G_{i,j}\}\}$$

pour $0 \leq k \leq z$, et

$$N' = \{(x,y) \in F \mid h(x) = 0,\ h(y) \geq 2\}$$

enfin

$$N_{z+1} = F - N' - \bigcup_{0 \leq k \leq z} N_k$$

Pour un $i \geq 1$ fixé, les nombres $A(i,j)$, $(j \geq 0)$ partitionnent les entiers positifs en intervalles $[A(i,j), A(i,j+1)[$, et $(x,y) \in N_k$ si k est le plus petit entier i tel que les hauteurs de x et y appartiennent au *même* intervalle relativement aux nombres $A(i,j)$, $j \geq 0$. Dans N_{z+1} on trouve tous les arcs (x,y) pour lesquels ce nombre est plus grand que k, à l'exception des arcs (x,y) pour lesquels x est une feuille. Ceux-ci se répartissent dans N_0 si $h(y) = 1$, et dans N', si $h(y) \geq 2$.

Posons, pour $0 \leq k \leq z+1$,

$$L_k = \{(x,y) \in N_k \mid \text{sur le chemin du TROUVER partiel contenant } (x,y), \text{ l'arc } (x,y) \text{ est le dernier arc dans } N_k\}$$

Notons en effet que si (x,y) figure sur le chemin d'un TROUVER partiel, les sommets x et y deviennent ensuite frères, donc ne peuvent plus se retrouver sur un tel chemin.

Lemme 5.5. *On a les inégalités suivantes :*

(1) $|L_k| \leq m$ *pour* $0 \geq k \geq z+1$; $|N'| \leq m$;
(2) $|N_0 - L_0| \leq n$;
(3) $|N_k - L_k| \leq n$ *pour* $1 \leq k \leq z$;
(4) $|N_{z+1} - L_{z+1}| \leq b(z,n)$, *avec* $b(z,n) = \min\{i \mid A(i,z) \geq \log n\}$.

Preuve. (1) Chaque TROUVER partiel a au plus un arc dans L_k, donc $|L_k| \leq m$. De même, chaque TROUVER partiel a au plus un arc (x, y) dans N', parce que x est une feuille, donc $|N'| \leq m$.

(2) Soit $(x, y) \in N_0 - L_0$. Il existe j tel que

$$2j = A(0, j) \leq h(x) < h(y) < A(0, j + 1) = 2(j + 1)$$

donc tel que $h(x) = 2j$ et $h(y) = 2j + 1$. Par ailleurs, il existe un arc $(s, t) \in N_0$ qui vient après l'arc (x, y) sur le chemin du même TROUVER partiel. On a donc

$$h(y) \leq h(s) < h(t)$$

A nouveau, il existe j' tel que $h(s) = 2j'$, $h(t) = 2j' + 1$, donc en fait $j < j'$ et $h(y) < h(s)$. Après compression du chemin, le père de x est un ancêtre de t, soit u, et en particulier $h(u) \geq h(t) > A(0, j + 1)$. Par conséquent, aucun TROUVER partiel ultérieur ne peut contenir un arc dont x est l'extrémité initiale, et qui appartienne à N_0. Il en résulte que $|N_0 - L_0| \leq n$.

Une partie de la preuve est commune pour les cas (3) et (4). Soit x un sommet, et supposons que $x \in G_{k,j}$ pour un k avec $1 \leq k \leq z + 1$, c'est-à-dire

$$A(k, j) \leq h(x) < A(k, j + 1).$$

Soit q le nombre d'arcs dans $N_k - L_k$ débutant par x. Notons-les

$$(x, y_1), \ldots, (x, y_q)$$

avec $h(y_1) \leq \cdots \leq h(y_q)$. Pour chaque i, $1 \leq i \leq q$, il existe un arc $(s_i, t_i) \in N_k$ qui figure ultérieurement dans le chemin du TROUVER partiel de (x, y_i). Par conséquent,

$$h(y_i) \leq h(s_i) < h(t_i)$$

De plus, après compression du chemin, le père de x est un ancètre de t_i (éventuellement t_i lui-même). Ceci prouve que $h(t_i) \leq h(y_{i+1})$. Maintenant, on a $(x, y_i) \notin N_{k-1}$ et $(s_i, t_i) \notin N_{k-1}$. Par définition, il existe $j' < j''$ avec

$$h(x) < A(k - 1, j') \leq h(y_i) \leq h(s_i) < A(k - 1, j'') \leq h(t_i)$$

et en particulier

$$h(y_i) < A(k - 1, j'') \leq h(y_{i+1}).$$

Il existe donc un entier ℓ tel que

$$h(y_1) < A(k - 1, \ell) \leq A(k - 1, \ell + q - 2) \leq h(y_q) \tag{5.1}$$

Les preuves de (3) et (4) se séparent maintenant.

Fin de la preuve de (3). Nous montrons que pour $x \in G_{k,j}$, $1 \leq k \leq z$, $j \geq 0$, il y a au plus $A(k, j)$ arcs dans $N_k - L_k$. Ceci est évident si $q = 1$. C'est également

clair si $j = 0$ ou $j = 1$, car pour ces valeurs de j, il n'y a pas d'arc dans N_k. On peut donc supposer $q \leq 2$, $j \leq 2$. Comme $x \in G_{k,j}$ et $y_q \in N_k$, on a $y_q \in G_{k,j}$ et par conséquent

$$h(y_q) < A(k, j+1) = A(k-1, A(k, j))$$

Ceci prouve, en vue de (5.1), que

$$A(k-1, \ell+q-2) < A(k-1, A(k, j))$$

donc

$$\ell + q - 2 < A(k, j)$$

Or $\ell \geq 1$ parce que $h(y_1) \leq 2$ et $A(k-1, \ell) > h(y_1)$, donc $A(k-1, \ell) > 2$. Par conséquent

$$q \leq A(k, j)$$

De cette inégalité, on obtient

$$|N_k - L_k| \leq \sum_{j \geq 2} |G_{k,j}| \cdot A(k, j)$$

et en majorant $|G_{k,j}|$ par la valeur donnée dans le lemme 5.4

$$|N_k - L_k| \leq \sum_{j \geq 2} 2n \cdot A(k, j)/2^{A(k,j)}$$

et comme $A(k, j) \geq 2^j$,

$$|N_k - L_k| \leq \sum_{j \geq 2} 2n \cdot \frac{2^j}{2^{2^j}} \leq 5n/8$$

Fin de la preuve de (4). Par le lemme 5.1, on a $h(y_q) \leq \log n$, et de l'équation (5.1) on obtient

$$A(z, \ell + q - 2) \leq \log n$$

La définition même de $b(z, n)$ implique que

$$\ell + q - 2 \leq b(z, n)$$

et comme ci-dessus, $q \leq b(z, n)$. Il y a donc au plus $b(z, n)$ arcs dans $N_{z+1} - L_{z+1}$ pour chaque sommet x qui n'est pas une feuille, d'où l'inégalité cherchée. ∎

Preuve du théorème. On a

$$\begin{aligned} |F| &= \sum_{k=0}^{z+1} |L_k| + |N'| + \sum_{k=0}^{z+1} |N_k - L_k| \\ &\leq m(z+3) + n + \frac{5}{8} nz + b(z, n) \end{aligned} \tag{5.2}$$

Fixons le paramètre z à

$$z = \min\{i \mid A(i, 4\left\lceil \frac{m}{n} \right\rceil) > \log n\} = \alpha(m, n).$$

Alors

$$b(z, n) = b(\alpha(m, n), n) = \min\{j \mid A(\alpha(m, n), j) > \log n\} \leq 4\left\lceil \frac{m}{n} \right\rceil \leq 8\frac{m}{n}$$

donc par (5.2)

$$|F| \leq (m + \frac{5}{8}n)\alpha(m, n) + 11m + n = O(m, \alpha(m, n))$$

parce que $m \leq n$. ∎

Notes

Un *type abstrait* est un ensemble organisé d'objets, muni d'un ensemble d'opérations permettant de les manipuler. Le langage des *types abstraits algébriques* permet de donner une description formelle des types abstraits, comme éléments d'une algèbre hétérogène particulière. Chaque opération sur un type est considérée comme une application (éventuellement partielle) qui, à une ou plusieurs instances du type, associe une nouvelle instance du type. Par exemple, empiler est une fonction (*pile, objet*) $\mapsto$ *pile*, et dépiler est une fonction *pile* $\mapsto$ *pile* qui n'est définie que si la pile de départ n'est pas vide. Ces fonctions doivent en outre vérifier certaines propriétés qui s'expriment souvent par des équations. Ainsi, pour toute pile p, et tout objet x, on doit avoir

$$\textit{dépiler}(\textit{empiler}(p, x)) = p.$$

Un type abstrait algébrique est défini alors comme l'ensemble de toutes les algèbres satisfaisant ces conditions. Cette approche algébrique de la description d'un type abstrait est présentée systématiquement dans

C. Froidevaux, M. C. Gaudel, M. Soria, *Types de données et algorithmes*, McGraw-Hill, 1990.

Les structures de données décrites dans ce chapitre sont des plus classiques, et sont traitées dans tous les livres d'algorithmique. Une présentation voisine est donnée dans le livre de Froidevaux, Gaudel, Soria, et dans

T. H. Cormen, C. E. Leiserson, R. L. Rivest, *Introduction to Algorithms*, MIT Press, McGraw-Hill, 1990.

L'analyse de l'algorithme de gestion des partitions est due à Tarjan. Nous suivons l'exposition de Mehlhorn. Tarjan a montré que l'algorithme n'est pas linéaire en général. L'existence d'un algorithme linéaire reste ouvert.

Les pagodes sont dues à Françon, Viennot et Vuillemin.

Exercices

3.1. Montrer que l'on peut implémenter une pile avec deux files. Quelle est la complexité en temps des opérations ?

3.2. Montrer que l'on peut implémenter une file avec deux piles. Quelle est la complexité en temps des opérations ?

3.3. Implémenter une liste avec un tableau.

3.4. Ecrire une procédure PURGER qui supprime toutes les répétitions dans une liste. Par exemple, le résultat de cette procédure sur la liste (a, b, a, a, b, c, a) est (a, b, c).

3.5. Ecrire une procédure RENVERSER qui retourne une liste. Par exemple, le résultat de cette procédure sur la liste (a, b, c, a, b, d) est (d, b, a, c, b, a).

3.6. Ecrire une procédure FUSION qui fusionne deux listes. Les listes sont triées au départ, le résultat est également trié, et un élément qui figurerait dans les deux listes de départ ne figure qu'une fois dans la liste résultat. Par exemple, la fusion des listes $(3, 5, 8, 11)$ et $(2, 3, 5, 14)$ est la liste $(2, 3, 5, 8, 11, 14)$.

3.7. Les arborescences ordonnées dont les sommets ont un nombre borné de fils peuvent s'implémenter en associant à chaque sommet un tableau de pointeurs vers ses fils. Réaliser les opérations de manipulation d'arborescences ordonnées dans cette représentation.

3.8. Un *arbre fileté* («threaded tree») est une structure de données pour représenter les arbres binaires, déclarée par

```
TYPE
   arbre = ^sommet;
   sommet = RECORD
               val: element;
               g, d: arbre;
               gvide, dvide: boolean
            END;
```

Pour un sommet s, `gvide` est vrai si et seulement si le sous-arbre gauche de s est vide, et dans ce cas, `g` pointe sur le sommet qui précède s en ordre symétrique. Dans le cas contraire, `g` pointe comme usuellement vers la racine du sous-arbre gauche de s. Bien entendu, la même convention vaut pour le sous-arbre droit.

a) Décrire des algorithmes pour parcourir un arbre binaire fileté en ordre préfixe, suffixe, symétrique.

b) Décrire l'insertion et la suppression dans un tel arbre, et décrire les implications des rotations (voir chapitre 6) sur cette représentation.

3.9. Une *pagode* est une représentation des tournois. Chaque sommet s d'une pagode contient deux pointeurs $g(s)$ et $d(s)$ définis comme suit.

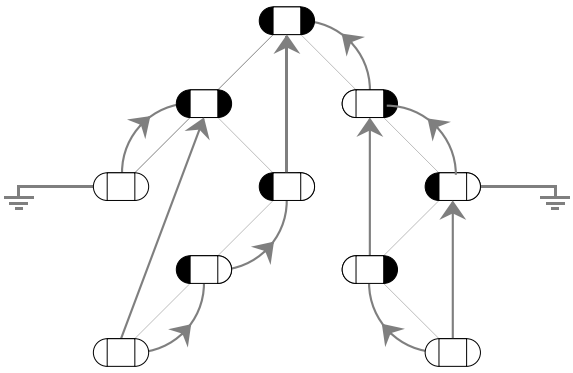


Figure 5.3: *Un arbre fileté.*

(1) si s est la racine ou est un fils droit, $g(s)$ pointe vers le premier sommet en ordre symétrique du sous-arbre de racine s ;
(2) si s est un fils gauche, $g(s)$ pointe vers le père de s.

Le pointeur $d(s)$ est défini de manière symétrique.

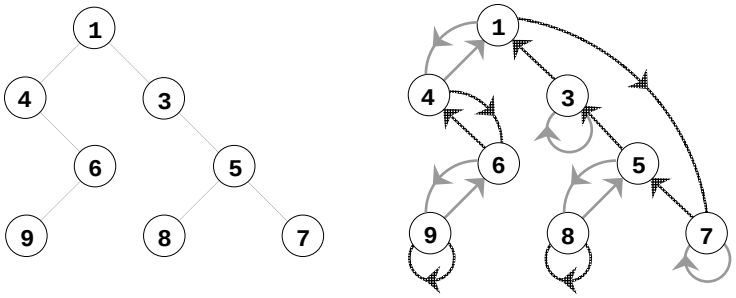


Figure 5.4: *Un tournoi et sa pagode.*

a) Ecrire une procédure d'insertion d'une clé dans une pagode, en l'insérant le plus à droite possible. En déduire qu'un parcours symétrique donne les clés dans leur ordre d'insertion.

b) Ecrire une procédure d'extraction de la plus petite clé présente.

c) Décrire un algorithme de fusion de deux pagodes.

3.10. On peut utiliser, pour la représentation d'une partition, en plus du tableau *classe*, une liste circulaire pour chaque classe. Réaliser les opérations UNIR et TROUVER et montrer qu'avec la règle de l'union pondérée, on peut réaliser une suite de $n-1$ UNIR et de m TROUVER en temps $O(m + n \log n)$.

3.11. Analyser l'algorithme de gestion de partitions qui utilise l'UNIR naïf et un TROUVER avec compression des chemins.

Chapitre 4

Graphes

Ce chapitre commence par la définition des graphes et de certains objets fondamentaux comme les chemins, les chaînes, les circuits et les cycles. Pour les graphes sans circuit, sont ensuite introduites les notions de rang et de liste topologique. La section 2 présente l'algorithme de Roy-Warshall dans le cadre général de la recherche des valeurs optimales des chemins entre tous les couples de sommets d'un graphe dont les arcs sont valués par les éléments d'un semi-anneau. On examine ensuite certains cas particuliers comme le calcul de la relation d'accessibilité, l'algorithme de Floyd et le calcul du langage des chemins d'un graphe étiqueté. La section 3 définit les arbres, les arborescences et leurs principales variantes attribuées. La section 4 introduit la notion de parcours d'un graphe non orienté et présente des algorithmes spécifiques pour les parcours en profondeur et en largeur. Les parcours sont ensuite étendus aux graphes orientés et l'algorithme de Tarjan, qui détermine les composantes fortement connexes d'un graphe, est présenté en tant qu'application des propriétés des parcours en profondeur. On termine par l'étude des parcours spécifiques aux arborescences.

Introduction

Les graphes constituent certainement l'outil théorique le plus utilisé pour la modélisation et la recherche des propriétés d'ensembles structurés. Ils interviennent chaque fois que l'on veut représenter et étudier un ensemble de liaisons (orientées ou non) entre les éléments d'un ensemble fini d'objets. Citons comme cas particuliers de liaisons des applications aussi diverses qu'une connexion entre deux processeurs d'un réseau informatique, une contrainte de précédence entre deux tâches d'un problème d'ordonnancement, la possibilité pour un système de passer d'un état à un autre ou encore une incompatibilité d'horaires. S'agissant d'un outil fondamental, la recherche des algorithmes les plus efficaces pour réaliser les opérations de base sur les graphes constitue un objectif essentiel de l'algorithmique.

4.1 Définitions et propriétés élémentaires

4.1.1 Définitions

Un *graphe orienté* $G = (S, A)$ est composé d'un ensemble *fini* S d'éléments appelés *sommets* et d'une partie A de $S \times S$ dont les éléments sont appelés *arcs*. Les arcs d'un graphe orienté constituent donc une relation sur l'ensemble de ses sommets. Un arc (x, y) représente une liaison *orientée* entre l'origine x et l'extrémité y. Si (x, y) est un arc, y est un *successeur* de x, x est un *prédécesseur* de y et si $x = y$, l'arc (x, x) est appelé *boucle*.

Lorsque l'orientation des liaisons n'est pas une information pertinente, la notion de *graphe non orienté* permet de s'en affranchir. Un *graphe non orienté* $G = (S, A)$ est composé d'un ensemble *fini* S d'éléments appelés *sommets* et d'une famille de *paires* de S dont les éléments sont appelés *arêtes*. Etant donné un graphe orienté, sa *version non orientée* est obtenue en supprimant les boucles et en substituant à chaque arc restant (x, y) la paire $\{x, y\}$.

Deux sommets distincts d'un graphe orienté (respectivement non orienté) G sont *adjacents* s'ils sont les extrémités d'un même arc (respectivement d'une même arête). Soient $G = (S, A)$ un graphe orienté (respectivement non orienté) et T un sous-ensemble de S. L'ensemble noté $\omega(T)$ des arcs (respectivement arêtes) de A dont une extrémité est dans T et l'autre dans $S - T$ est appelé le *cocycle* associé à T. L'ensemble noté $\mathcal{B}(T)$ des sommets de $S - T$ adjacents à au moins un sommet de T est appelé *bordure* de T. Si le sommet u appartient à $\mathcal{B}(T)$, on dit aussi que u est adjacent à T. Le graphe $G = (S, A)$ est dit *biparti* s'il existe un sous-ensemble de sommets T tel que $A = \omega(T)$.

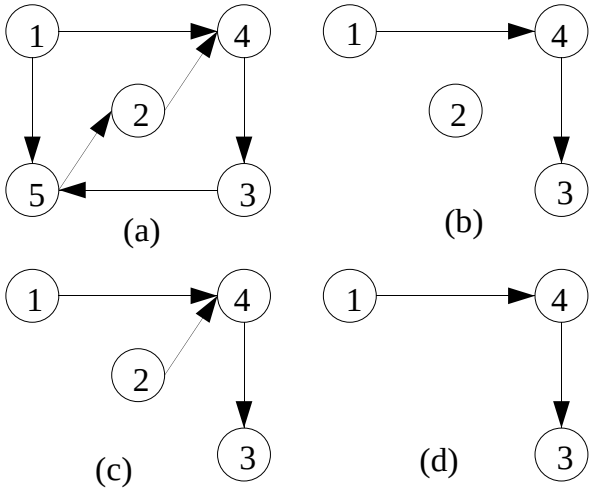


Figure 1.1: *Sous-graphes.*

Un *sous-graphe* du graphe $G = (S, A)$ est un couple $G' = (S', A')$ pour lequel $S' \subset S$ et $A' \subset A$. Le sous-graphe G' est un graphe *partiel* de G si $S' = S$. Si A'

est l'ensemble des arcs de A dont les deux extrémités sont dans S', le sous-graphe G' est dit *induit* par S'. Si S' est l'ensemble des extrémités des arcs de A', le sous-graphe G' est dit *induit* par A'.

Les sommets d'un graphe sont représentés par des points distincts du plan. Un arc (x, y) d'un graphe orienté est représenté par une flèche d'origine x et d'extrémité y, une arête d'un graphe non orienté est représentée par une ligne joignant ses deux extrémités. Sur la figure 1.1 sont représentés en a) un graphe orienté G, en b) un sous-graphe de G, en c) le graphe partiel de G induit par les sommets $\{1, 2, 3, 4\}$ et en d) le sous-graphe induit par les arcs $\{(1, 4), (4, 3)\}$. Le graphe G est biparti car tout arc est incident à $T = \{1, 2, 3\}$.

4.1.2 Implémentations d'un graphe

Parmi les implémentations possibles d'un graphe orienté $G = (S, A)$, on peut en retenir essentiellement trois qui interviennent dans la plupart des problèmes théoriques et pratiques : la matrice d'adjacence, la matrice d'incidence et la liste des successeurs.

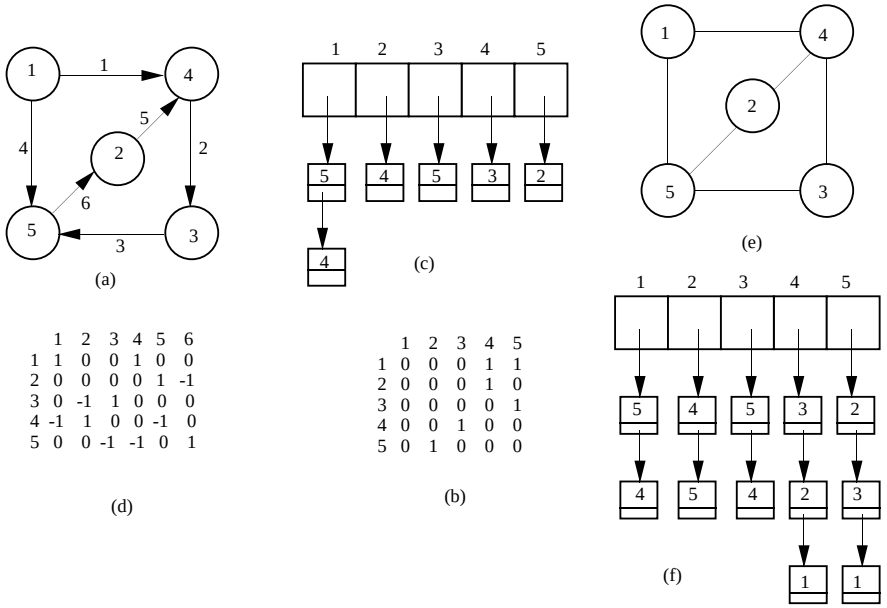


Figure 1.2: *Implémentations d'un graphe.*

La *matrice d'adjacence* $M = M(G)$ est une matrice carrée, indicée par S, définie par :

$$m_{ij} = \begin{cases} 1 & \text{si } (i,j) \in A \\ 0 & \text{sinon} \end{cases}$$

où le 0 et le 1 seront entiers ou booléens selon la convenance.

Si G est un graphe sans boucles, sa *matrice d'incidence* «sommets-arcs» $\Delta(G)$ est définie par :

$$\delta_{xa} = \begin{cases} 1 & \text{si } x \text{ est l'origine de l'arc } a \\ -1 & \text{si } x \text{ est l'extrémité de l'arc } a \\ 0 & \text{sinon} \end{cases}$$

Cette matrice possède une propriété très importante pour certains domaines de l'optimisation combinatoire comme la théorie des flots (voir chapitre 8) : le déterminant de toute sous-matrice carrée extraite vaut 0, 1 ou -1. Une matrice satisfaisant cette propriété est dite *totalement unimodulaire*.

Les matrices d'ajacence et d'incidence du graphe orienté de la figure 1.2 sont les suivantes :

$$M = \begin{pmatrix} 0&0&0&1&1 \\ 0&0&0&1&0 \\ 0&0&0&0&1 \\ 0&0&1&0&0 \\ 0&1&0&0&0 \end{pmatrix} \quad \Delta = \begin{pmatrix} 1&0&0&1&0&0 \\ 0&0&0&0&1&-1 \\ 0&-1&1&0&0&0 \\ -1&1&0&0&-1&0 \\ 0&0&-1&-1&0&1 \end{pmatrix}.$$

La *liste des successeurs* (en anglais : adjacency list) est un tableau $(q_1, \ldots, q_n)$ où q_i est un pointeur sur une liste des successeurs du sommet i.

Dans le cas d'un graphe non orienté, l'implémentation la plus utilisée est la *liste des voisins* qui est un tableau $(q_1, \ldots, q_n)$ où q_i est un pointeur sur une liste des sommets adjacents au sommet i. C'est la version non orientée de la liste des successeurs.

Notons que les tailles de ces diverses représentations sont sensiblement différentes. Si n est le nombre de sommets et si m est le nombre d'arcs (ou d'arêtes pour un graphe non orienté), la taille de la matrice d'adjacence est en $\theta(n^2)$, celle de la matrice d'incidence en $\theta(nm)$ alors que la taille de la liste des successeurs (ou des voisins) est en $\theta(n+m)$.

La figure 1.2 montre la liste des successeurs (c) du graphe orienté (a) et la liste des voisins (d) du graphe non orienté (b).

4.1.3 Chemins, chaînes, circuits, cycles

Soit $G = (S, A)$ un graphe orienté.

Un *chemin* d'origine x et d'extrémité y est une suite finie non vide de sommets $c = (s_0, \ldots, s_p)$ telle que : $s_0 = x$, $s_p = y$ et pour $k = 0, .., p-1$, $(s_k, s_{k+1}) \in A$. La *longueur* du chemin c est p, c'est le nombre d'arcs (non nécessairement distincts) empruntés par ce chemin.

La notion de chemin nous permet d'introduire les *ascendants* et les *descendants* d'un sommet. Soit x un sommet de S, un sommet y est un ascendant (respectivement descendant) de x s'il existe un chemin de y à x (respectivement de x à y)

dans G. Le sommet y est un ascendant (respectivement descendant) *propre* du sommet x s'il existe un chemin de longueur non nulle de y à x (respectivement de x à y). Notons qu'un sommet peut être ascendant propre et descendant propre de lui-même.

Un chemin $c = (s_0, \ldots, s_p)$ est *simple* si les arcs $(s_{i-1}, s_i), i = 1, .., p$ sont deux à deux distincts. Un chemin $c = (s_0, \ldots, s_p)$ est *élémentaire* si ses sommets sont distincts deux à deux. Un chemin $(s_0, \ldots, s_p)$ est un *circuit* si $p \geq 1$ et $s_0 = s_p$. Un circuit $c = (s_0, \ldots, s_p)$ est *élémentaire* si le chemin $(s_0, \ldots, s_{p-1})$ est élémentaire. Soulignons qu'un circuit élémentaire n'est pas un chemin élémentaire et que, de la même façon, un chemin élémentaire n'est pas un circuit élémentaire.

Considérons le graphe orienté de la figure 1.2. La suite $(5, 2, 4, 3, 5)$ est un circuit élémentaire mais n'est pas un chemin élémentaire. Par contre la suite $(1, 4, 3)$ est un chemin élémentaire.

Considérons maintenant un graphe non orienté $G = (S, A)$ et définissons les notions correspondantes de chaîne et cycle. Une *chaîne* d'origine x et d'extrémité y est une suite finie de sommets $(s_0, \ldots, s_p)$ telle que $s_0 = x$, $s_p = y$, deux sommets consécutifs quelconques de la liste sont les extrémités d'une arête, et ces arêtes sont *distinctes* deux à deux. Notons que si $(s_0, \ldots, s_p)$ est une chaîne, il en est de même pour $(s_p, \ldots, s_0)$.

Une chaîne $c = (s_0, \ldots, s_p)$ est *élémentaire* si les sommets de la liste c sont deux à deux distincts. Une chaîne $(s_0, \ldots, s_p)$ est un *cycle* si $s_0 = s_p$. Un cycle $(s_0, \ldots, s_p)$ est *élémentaire* si $p \geq 1$ et si la liste $(s_0, \ldots, s_{p-1})$ est une chaîne élémentaire.

Pour le graphe non orienté représenté sur la figure 1.2, la liste $(5, 1, 4, 1)$ n'est pas une chaîne et $(1, 5, 3, 4)$ est une chaîne élémentaire.

Un graphe non orienté est *connexe* si pour tout couple de sommets, il existe une chaîne ayant ces deux sommets comme extrémités. Par extension, un graphe orienté est connexe si sa version non orientée (c'est-à-dire le graphe non orienté obtenu en supprimant les orientations et les boucles) est connexe. La relation sur l'ensemble des sommets d'un graphe non orienté définie par $x \sim y$ s'il existe une chaîne de x à y est une relation d'équivalence dont les classes sont appelées *composantes connexes* du graphe. Les graphes induits par les composantes connexes sont bien sûr des graphes connexes.

4.1.4 Lemme de König

Les chemins d'un graphe constituent en général un ensemble infini (il faut et il suffit que le graphe possède des circuits). Par contre les chemins élémentaires sont en nombre fini et constituent pour de nombreux problèmes d'optimisation un ensemble *dominant* lorsque l'existence d'un chemin optimal élémentaire est prouvée. Le lemme de König montre que l'existence d'un chemin d'origine x et d'extrémité y entraîne celle d'un chemin élémentaire entre ces mêmes sommets.

Soit $G = (S, A)$ un graphe orienté. Un chemin c est dit *extrait* d'un chemin c' si les deux chemins c et c' ont la même origine et la même extrémité et si la suite des arcs de c est une sous-suite de celle des arcs de c'. La relation «est extrait de» définie sur l'ensemble des chemins de G est une relation d'ordre et le lemme de König ci-dessous montre que ses éléments minimaux sont les chemins élémentaires de G.

Lemme 1.1 (de König)**.** *De tout chemin on peut extraire un chemin élémentaire.*

Preuve. Nous raisonnons par induction sur la longueur p du chemin c. Si $p = 0$, c est élémentaire. Soit donc $c = (x_0, \ldots, x_p)$ un chemin à p sommets avec $p > 0$. Si c n'est pas élémentaire, il existe un couple (r, s) tel que $0 \leq r < s \leq p$ et $x_r = x_s$. Le chemin $c' = (x_0, .., x_r, x_{s+1}, .., x_p)$ est extrait de c et strictement plus petit que c. On peut donc extraire de c', et donc de c, un chemin élémentaire. ∎

Remarque. Le lemme de König s'étend aux circuits d'un graphe orienté et aussi aux chaînes et aux cycles d'un graphe non orienté.

4.1.5 Graphes sans circuit

Les graphes sans circuit interviennent dans de nombreux domaines, en particulier chaque fois que l'on étudie une relation d'ordre partiel. Pour ces graphes, on connaît des algorithmes spécifiques dont l'efficacité est souvent due à l'utilisation d'une liste des sommets qui permet, lors de la visite d'un sommet de la liste, d'être sûr que tous ses ascendants propres ont déjà été visités. Une *liste topologique* des sommets d'un graphe $G = (S, A)$ est une permutation $(s_1, \ldots, s_n)$ des sommets de S telle que pour tout arc (s_i, s_j) on a $i < j$.

Proposition 1.2. *Un graphe orienté $G = (S, A)$ est sans circuit si et seulement s'il existe une liste topologique des sommets de G.*

Preuve. La condition suffisante résulte directement de la définition d'une liste topologique. Démontrons que la condition est nécessaire par induction sur $n = \text{Card}(S)$. La propriété est bien sûr vraie si $n = 1$. Supposons qu'elle le soit pour un graphe à $n - 1$ sommets ($n \geq 2$). Le graphe G ne possédant pas de circuits, il existe au moins un sommet s sans descendants propres car dans le cas contraire, on pourrait construire un chemin infini, donc contenant nécessairement un circuit. Le sous-graphe G' de G induit par $S - \{s\}$ a $n - 1$ sommets et ne possède pas de circuits. Il existe donc une liste topologique L' des sommets de G' ; la liste $L'.(s)$ formée de L' concaténée avec la liste (s) est alors une liste topologique des sommets de G. ∎

La propriété précédente conduit naturellement à un algorithme pour tester si un graphe orienté G est sans circuit. Le principe de cet algorithme est de rechercher

une sortie s de G. S'il n'en n'existe pas alors G possède un circuit sinon la réponse pour G est la même que pour G_s. En utilisant une liste des successeurs pour coder G, on obtient une complexité $O(n^2)$. Nous donnerons dans la section 4.4.4 un algorithme plus efficace, de complexité $O(n+m)$, fondé sur un parcours en profondeur du graphe G.

Il peut exister plusieurs listes topologiques d'un même graphe sans circuit. Le graphe représenté sur la figure 1.3 ne possède qu'une seule liste topologique $(3, 2, 1, 6, 7, 5, 4)$ car il existe un seul chemin élémentaire passant par tous les sommets.

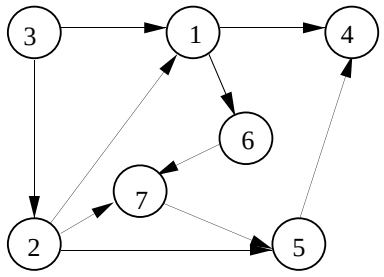


Figure 1.3: *Un graphe sans circuit.*

Plus généralement, soit $\leq$ une relation d'ordre sur un ensemble fini E. Une *extension linéaire* de $\leq$ est un ordre total $\preceq$ sur E qui contient $\leq$, c'est-à-dire tel que :

$$x \leq y \Longrightarrow x \preceq y.$$

Etant donné un ordre total $\preceq$ sur S, on peut constituer la suite $(s_1, \ldots, s_n)$ des éléments de S telle que :

$$s_1 \preceq s_2 \preceq \ldots \preceq s_n.$$

L'opération qui détermine une extension linéaire s'appelle le *tri topologique* et le résultat du tri est une liste topologique. Déterminer une extension linéaire équivaut bien entendu à *numéroter* les sommets du graphe, c'est-à-dire à trouver une bijection $\nu : S \mapsto \{1, \ldots, n\}$ vérifiant :

$$(x, y) \in A \Longrightarrow \nu(x) < \nu(y).$$

Soit $G = (S, A)$ un graphe sans circuit. Le *rang* du sommet s, noté $\rho(s)$, est la longueur maximale d'un chemin d'extrémité s. Notons que $\rho(s)$ est bien défini puisque les chemins de G sont élémentaires et donc en nombre fini. La figure 1.4 montre un graphe sans circuit et à sa droite le même graphe rangé.

Proposition 1.3. *Pour tout sommet x, les ascendants propres de x ont un rang strictement inférieur au rang de x, les descendants propres de x ont un rang strictement supérieur au rang de x, et si x est de rang $k > 0$, il possède un ascendant propre de rang $k-1$.*

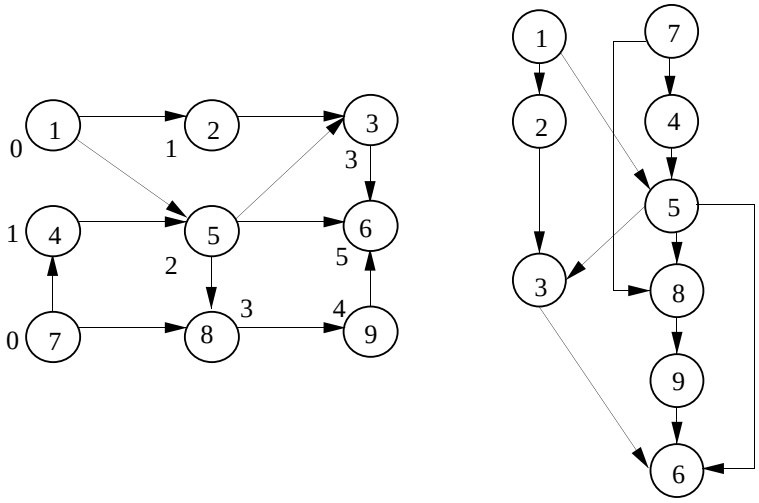


Figure 1.4: *Classement par rang.*

Preuve. Si y est un ascendant propre de x, alors $\rho(x) > \rho(y)$. De même, si y est un descendant propre de x, alors $\rho(x) < \rho(y)$. Enfin si tous les ascendants propres de x étaient de rang inférieur ou égal à $k-2$, on aurait : $\rho(x) \leq k-1$, d'où la contradiction. ∎

Remarque. La liste des sommets d'un graphe sans circuit, ordonnée par rang croissant au sens large, est une liste topologique.

4.2 Accessibilité

Etant donné un graphe orienté $G = (S, A)$, le sommet y est dit *accessible* à partir du sommet x s'il existe un chemin de x à y. L'objet de cette section est le calcul de la relation d'accessibilité. A cet effet, nous décrivons l'algorithme de Roy-Warshall qui fournit la matrice booléenne de la relation d'accessibilité. Cet algorithme qui utilise comme structure de données la matrice booléenne associée au graphe initial a une complexité $O(n^3)$, où n est le nombre de sommets de G.

Nous posons $S = \{1, 2, \cdots, n\}$ et pour $k \in S$, nous notons $E(k)$ l'ensemble $\{1, 2, \cdots, k\}$; par convention l'ensemble $E(0)$ est vide. Si $c = (v_1, \ldots, v_k)$ est un chemin de G, l'intérieur $I(c)$ de ce chemin est l'*ensemble* des sommets du sous-chemin $(v_2, \ldots, v_{k-1})$; si $k \leq 2$, $I(c)$ est l'ensemble vide. Enfin nous notons $G_k = (S, A_k)$ le graphe défini par :

$$(i, j) \in A_k \Longleftrightarrow \exists c : i \to j, \quad I(c) \subset E(k).$$

Remarque. Le graphe G_0 est le graphe initial G complété d'une boucle en chaque sommet; le graphe G_n est le graphe de la relation d'accessibilité.

4.2.1 Algorithme de Roy-Warshall

L'algorithme de Roy-Warshall calcule la suite des graphes G_1, $G_2, \ldots, G_n$ en utilisant le théorème suivant :

Théorème 2.1. *Pour tout $i, j, k \in S$, l'arc (i, j) appartient à A_k si et seulement si $(i, j) \in A_{k-1}$ ou $((i, k) \in A_{k-1}$ et $(k, j) \in A_{k-1})$.*

Preuve. Si $(i, j) \in A_k$, il existe d'après le lemme de König un chemin élémentaire c de i à j dont l'intérieur $I(c)$ est inclus dans $E(k)$. Si $I(c)$ ne contient pas k, alors $I(c)$ est inclus dans $E(k-1)$ et donc $(i, j) \in A_{k-1}$. Si $I(c)$ contient k, nous notons c' (respectivement c'') le sous-chemin de c de i à k (respectivement le sous-chemin de c de k à j). Le chemin c étant élémentaire, $I(c')$ et $I(c'')$ sont inclus dans $E(k-1)$; il en résulte que $(i, k) \in A_{k-1}$ et $(k, j) \in A_{k-1}$.

Réciproquement on a : $(i, j) \in A_{k-1} \Longrightarrow (i, j) \in A_k$, car $E(k-1) \subset E(k)$. Supposons maintenant que $(i, k) \in A_{k-1}$ et $(k, j) \in A_{k-1}$; il existe alors un chemin c' (respectivement c'') de i à k (respectivement de k à j) tel que $I(c')$ (respectivement $I(c'')$) soit inclus dans $E(k-1)$. L'intérieur du chemin c obtenu par la concaténation de c' et c'' est donc inclus dans $E(k)$ et par conséquent $(i, j) \in A_k$. ∎

Une implémentation possible de l'algorithme de Roy-Warshall utilise comme structure de données une seule matrice booléenne, notée a et initialisée avec la matrice d'adjacence de G ; cette implémentation est donnée par la procédure ROY-WARSHALL(G) suivante :

```
procédure ROY-WARSHALL(G);
  {g est la matrice d'adjacence du graphe G}
  {a est la matrice de travail}
  a := g;
  pour i de 1 à n faire a_ii := 1;
  pour k de 1 à n faire
    pour i de 1 à n faire
      pour j de 1 à n faire
        a_ij := a_ij ou (a_ik et a_kj);
  retourner(a).
```

L'utilisation d'une seule matrice de travail rend nécessaire la proposition suivante pour valider la procédure ROY-WARSHALL(G).

Proposition 2.2. *Pour tout $i, j, k \in S$, les deux équivalences suivantes sont vraies :*

(1) $(i, k) \in A_{k-1} \Longleftrightarrow (i, k) \in A_k$;

(2) $(k,j) \in A_{k-1} \Longleftrightarrow (k,j) \in A_k$.

Preuve. (laissée à titre d'exercice). ∎

Théorème 2.3. *La procédure* ROY-WARSHALL(G) *détermine la matrice de la relation d'accessibilité d'un graphe orienté $G = (S, A)$ en $O(n^3)$ opérations élémentaires.*

Preuve. Notons $a^{(k)}$ la valeur de la matrice a calculée par la procédure ROY-WARSHALL(G) lors de l'itération k et supposons que jusqu'au calcul de l'élément (i,j) de a à l'itération k, les valeurs $a_{rs}^{(p)}$ soient correctes. Lors du calcul de $a_{ij}^{(k)}$, l'élément (i,j) de la matrice a contient initialement la valeur $a_{ij}^{(k-1)}$, par contre l'élément (i,k) de la matrice a contient la valeur $a_{ik}^{(k-1)}$ si $k > j$ ou la valeur $a_{ik}^{(k)}$ si $k < j$; de même l'élément (k,j) de la matrice a contient la valeur $a_{kj}^{(k-1)}$ si $k > i$ ou la valeur $a_{ik}^{(k)}$ si $k < i$. Cependant la proposition précédente nous assure que la valeur calculée $a_{ij}^{(k)}$ est correcte. ∎

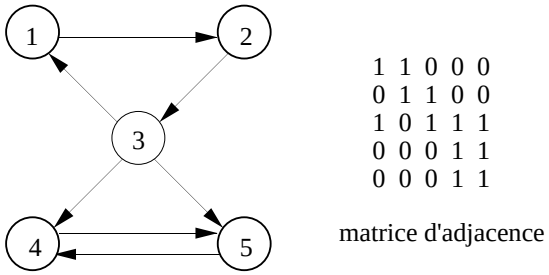


Figure 2.1: *Un graphe orienté et sa matrice d'adjacence.*

Exemple. Considérons le graphe orienté de la figure 2.1. Les matrices d'adjacence des graphes G_1, G_2 et G_3 sont les suivantes :

$$A_1 = \begin{pmatrix} 1&1&0&0&0 \\ 0&1&1&0&0 \\ 1&1&1&1&1 \\ 0&0&0&1&1 \\ 0&0&0&1&1 \end{pmatrix} \quad A_2 = \begin{pmatrix} 1&1&1&0&0 \\ 0&1&1&0&0 \\ 1&1&1&1&1 \\ 0&0&0&1&1 \\ 0&0&0&1&1 \end{pmatrix} \quad A_3 = \begin{pmatrix} 1&1&1&1&1 \\ 1&1&1&1&1 \\ 1&1&1&1&1 \\ 0&0&0&1&1 \\ 0&0&0&1&1 \end{pmatrix}$$

Les graphes G_4 et G_5 sont égaux au graphe G_3, si bien que la matrice de la relation d'accessibilité est celle du graphe G_3.

4.2.2 Autres problèmes d'accessibilité

Nous considérons dans ce paragraphe la notion de graphe étiqueté, c'est-à-dire de graphe dont chaque arc (i,j) porte une étiquette e_{ij} choisie dans un ensemble $\mathcal{E}$.

L'idée centrale de l'algorithme de Roy-Warshall est de résoudre la suite des sous-problèmes obtenus en faisant « grossir » de l'ensemble vide jusqu'à S lui-même, l'ensemble des sommets qui peuvent appartenir à l'intérieur d'un chemin. Une formule de récurrence permet de passer simplement de la solution d'un sous-problème à celle du sous-problème suivant. Cette même idée conduit à la résolution d'une classe de problèmes analogues lorsque l'ensemble des étiquettes possède une structure de *semi-anneau*. Nous illustrons cette méthodologie en traitant deux problèmes classiques sur les graphes étiquetés :

a) Si $\mathcal{E} = \mathbb{R}$, déterminer pour tout couple de sommets la valeur maximale des chemins entre ces deux sommets ;
b) Si $\mathcal{E}$ est l'ensemble des parties non vides d'un alphabet A, déterminer pour tout couple de sommets le langage des étiquettes des chemins entre ces deux sommets.

Chemins de valeur maximale : algorithme de Floyd

Soit $G = (S, A)$ un graphe orienté dont chaque arc est valué par un nombre réel. Nous appelons *valeur* d'un chemin la somme des étiquettes des arcs de ce chemin. Nous supposons de plus que tout circuit du graphe a une valeur négative ou nulle (on dit encore qu'il n'existe pas de *circuits absorbants*). Le problème consiste alors à déterminer, pour tout couple de sommets (i, j) la valeur maximale des chemins de i à j, notée a_{ij}.

Etant donné un couple (i, j), le lemme de König et l'absence de circuits positifs montrent qu'il suffit de calculer la valeur maximale des chemins élémentaires de i à j et donc que a_{ij} est fini. La recherche sera donc conduite dans l'ensemble des chemins élémentaires de G.

Nous notons $a_{ij}^{(k)}$ la valeur maximale d'un chemin de i à j dont l'intérieur est inclus dans $E(k)$ et nous prenons comme valeurs initiales, pour $k = 0$:

$$a_{ij}^{(0)} = \begin{cases} e_{ij}, & \text{si } (i,j) \in A \text{ et } i \neq j\,; \\ 0, & \text{si } i = j\,; \\ -\infty, & \text{sinon.} \end{cases}$$

La proposition suivante permet alors le calcul des $a_{ij}^{(k)}$:

Proposition 2.4. *Pour tout $i, j, k \in S$, on a $a_{ij}^{(k)} = \max\{a_{ij}^{(k-1)}, a_{ik}^{(k-1)} + a_{kj}^{(k-1)}\}$.*

Preuve. La valeur maximale d'un chemin de i à j dont l'intérieur est inclus dans $E(k-1)$ est par définition $a_{ij}^{(k-1)}$.

La valeur maximale d'un chemin de i à j dont l'intérieur est inclus dans $E(k)$ et contient k est $a_{ik}^{(k-1)} + a_{kj}^{(k-1)}$ car tout chemin de cet ensemble est constitué de la concaténation d'un chemin de i à k dont l'intérieur est inclus dans $E(k-1)$ et d'un chemin de k à j dont l'intérieur est inclus dans $E(k-1)$.

Comme les deux ensembles de chemins précédents recouvrent tous les chemins de i à j dont l'intérieur est inclus dans $E(k)$, la valeur maximale cherchée est $\max\{a_{ij}^{(k-1)}, a_{ik}^{(k-1)} + a_{kj}^{(k-1)}\}$. ∎

L'algorithme issu de cette proposition a été découvert par Floyd, la procédure ci-dessous de complexité $O(n^3)$ implémente cet algorithme et n'utilise qu'une seule matrice de travail a.

```
procédure FLOYD(G, e);
  {g est la matrice d'adjacence du graphe G}
  {e est la matrice des étiquettes du graphe G}
  {a est la matrice résultat}
  pour i de 1 à n faire
    pour j de 1 à n faire
      a_ij :=si g_ij = 1 alors e_ij sinon −∞
    finpour;
    a_ii := 0
  finpour;
  pour k de 1 à n faire
    pour i de 1 à n faire
      pour j de 1 à n faire
        a_ij^(k) = max{a_ij^(k−1), a_ik^(k−1) + a_kj^(k−1)}
  retourner(a).
```

La figure 2.2 montre un graphe étiqueté par des réels. Les matrices, où $-\infty$ est

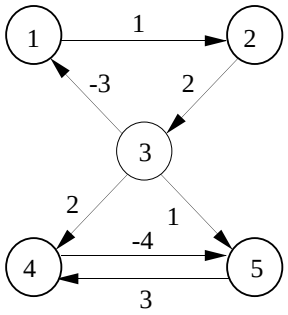


Figure 2.2: *Un graphe étiqueté.*

codé par un tiret, calculées par la procédure FLOYD(G) sont les suivantes :

$$a^{(0)} = \begin{pmatrix} 0 & 1 & - & - & - \\ - & 0 & 2 & - & - \\ -3 & - & 0 & 2 & 1 \\ - & - & - & 0 & -4 \\ - & - & - & 3 & 0 \end{pmatrix} a^{(1)} = \begin{pmatrix} 0 & 1 & - & - & - \\ - & 0 & 2 & - & - \\ -3 & -2 & 0 & 2 & 1 \\ - & - & - & 0 & -4 \\ - & - & - & 3 & 0 \end{pmatrix} a^{(2)} = \begin{pmatrix} 0 & 1 & 3 & - & - \\ - & 0 & 2 & - & - \\ -3 & -2 & 0 & 2 & 4 \\ - & - & - & 0 & -4 \\ - & - & - & 3 & 0 \end{pmatrix}$$

$$a^{(3)} = \begin{pmatrix} 0 & 1 & 3 & 5 & 4 \\ -1 & 0 & 2 & 4 & 3 \\ -3 & -2 & 0 & 2 & 1 \\ - & - & - & 0 & -4 \\ - & - & - & 3 & 0 \end{pmatrix} a^{(4)} = \begin{pmatrix} 0 & 1 & 3 & 5 & 4 \\ -1 & 0 & 2 & 4 & 3 \\ -3 & -2 & 0 & 2 & 1 \\ - & - & - & 0 & -4 \\ - & - & - & 3 & 0 \end{pmatrix} a^{(5)} = \begin{pmatrix} 0 & 1 & 3 & 7 & 4 \\ -1 & 0 & 2 & 6 & 3 \\ -3 & -2 & 0 & 4 & 1 \\ - & - & - & 0 & -4 \\ - & - & - & 3 & 0 \end{pmatrix}$$

Langage des chemins d'un graphe étiqueté

Nous considérons ici un alphabet A et prenons pour $\mathcal{E}$ l'ensemble des parties non vides de A. L'ensemble des étiquettes d'un chemin de i à j est le langage produit des étiquettes associées aux arcs successifs du chemin. Nous appelons alors langage des chemins de i à j, et nous notons L_{ij} l'ensemble des étiquettes des chemins de i à j. Le problème consiste alors à déterminer pour chaque couple (i, j) le langage L_{ij}.

Ce problème est classique dans le cadre de la théorie des automates, on peut en particulier se servir de la construction itérative des L_{ij} réalisée par l'algorithme dans la preuve du théorème de Kleene (voir chapitre 9).

Nous notons $L_{ij}^{(k)}$ le langage des chemins de i à j dont l'intérieur est inclus dans $E(k)$ et nous prenons pour $k = 0$ les valeurs initiales suivantes :

$$L_{ij}^{(0)} = \begin{cases} A_{ij} & \text{si } i \neq j \\ A_{ij} \cup \{1\} & \text{sinon} \end{cases}$$

où 1 représente le mot vide et A_{ij} est l'ensemble des étiquettes des arcs de i à j. La proposition suivante permet alors le calcul des langages $L_{ij}^{(k)}$:

Proposition 2.5. *Pour tout $i, j, k \in S$, on a*

$$L_{ij}^{(k)} = L_{ij}^{(k-1)} \cup L_{ik}^{(k-1)} L_{kk}^{(k-1)*} L_{kj}^{(k-1)}.$$

Preuve. Un mot du second membre est soit dans $L_{ij}^{(k-1)}$, soit dans l'ensemble $L_{ik}^{(k-1)}(L_{kk}^{(k-1)})^* L_{kj}^{(k-1)}$. Dans le premier cas le mot appartient à $L_{ij}^{(k)}$, dans le second cas, il s'écrit uvw où u appartient à $L_{ik}^{(k-1)}$, v à $(L_{kk}^{(k-1)})^*$ et w à $L_{kj}^{(k-1)}$. Par définition u est donc une étiquette d'un chemin de i à k dont l'intérieur est inclus dans $E(k-1)$, v est la concaténation d'un nombre fini d'étiquettes de circuits de k à k dont l'intérieur est inclus dans $E(k-1)$, enfin v est l'étiquette d'un chemin de k à j dont l'intérieur est inclus dans $E(k-1)$. Le mot formé par la concaténation de u, v et w est alors une étiquette du chemin obtenu par concaténation des chemins associés.

Réciproquement, considérons un mot $u \in L_{ik}^{(k)}$. Si l'intérieur de ce chemin ne contient pas k, u appartient par définition à $L_{ij}^{(k-1)}$; si l'intérieur de ce chemin contient k, ce chemin est formé de la concaténation d'un chemin de i à k (premier passage par k en tant que sommet intermédiaire), d'une suite finie (éventuellement

vide) de chemins de k à k (associés à tous les passages successifs par k en tant que sommet intermédiaire) et d'un dernier chemin de k à j. Par construction, l'intérieur de tous ces chemins est inclus dans $E(k-1)$. Le mot u est donc la concaténation d'un mot de $L_{ik}^{(k-1)}$, d'une suite de mots de $L_{kk}^{(k-1)}$ et d'un mot de $L_{kj}^{(k-1)}$. ∎

Nous pouvons donc déduire de la proposition précédente que, quel que soit le couple (i, j), le langage des chemins de i à j est obtenu à partir des langages réduits aux lettres de A par une suite finie d'opérations d'union, concaténation et étoile.

4.2.3 Semi-anneaux et accessibilité

Un ensemble K muni de deux opérations binaires associatives $\oplus$ et $\otimes$ et de deux éléments distingués 0 et 1 est un *semi-anneau* si

1. $(K, \oplus, 0)$ est un monoïde commutatif, c'est-à-dire $a \oplus b = b \oplus a$ pour $a, b \in K$ et $a \oplus 0 = 0 \oplus a = a$ pour tout a dans K ;
2. $(K, \otimes, 1)$ est un monoïde et $a \otimes 1 = 1 \otimes a = a$ pour tout a dans K ;
3. l'opération $\otimes$ est distributive par rapport à $\oplus$, c'est-à-dire : $a \otimes (b \oplus c) = (a \otimes b) \oplus (a \otimes c)$ et $(a \oplus b) \otimes c = (a \otimes c) \oplus (a \otimes b)$;
4. l'élément 0 est un zéro pour l'opération $\otimes$: $0 \otimes a = a \otimes 0 = 0$.

On note $(K, \oplus, \otimes, 0, 1)$ un semi-anneau. Par exemple $(\mathbb{N}, +, \times, 0, 1)$ est un semi-anneau. Un semi-anneau $(K, \oplus, \otimes, 0, 1)$ est *complet* si toute famille $\{a_i\}_{i \in I}$ d'éléments de K admet une somme notée $\bigoplus_{i \in I} a_i$ sujette aux conditions suivantes :

a) si I est fini, $I = \{i_1, \ldots, i_n\}$, alors $\bigoplus_{i \in I} a_i = a_{i_1} \oplus a_{i_2} \ldots \oplus a_{i_n}$;

b) la somme est associative : si $I = \bigcup_{j \in J} I_j$, avec $I_j \cap I_k = \emptyset$ pour $j \neq k$, alors :

$$\bigoplus_{i \in I} a_i = \bigoplus_{j \in J} (\bigoplus_{i \in I_j} a_i)$$

c) l'opération $\otimes$ est distributive par rapport à $\oplus$, c'est-à-dire :

$$(\bigoplus_{i \in I} a_i) \otimes (\bigoplus_{j \in J} b_j) = \bigoplus_{i \in I} \bigoplus_{j \in J} (a_i \otimes b_j).$$

Cette définition de la complétude est assez lourde, il suffit de retenir que les sommes infinies sont autorisées et que l'on peut les manipuler comme des sommes finies.

Exemples.

1) Le semi-anneau de Boole $(\mathcal{B}, \vee, \wedge, 0, 1)$, avec $\mathcal{B} = \{0, 1\}$ est complet. On a :

$$\bigvee_{i \in I} a_i = \begin{cases} 1 & \text{s'il existe } i \in I \text{ tel que } a_i = 1\,; \\ 0 & \text{sinon.} \end{cases}$$

2) Soit $\mathcal{N} = \mathbb{N} \cup \{+\infty\}$, avec $a + (+\infty) = (+\infty) + a = +\infty$. Alors $(\mathcal{N}, \min, +, \infty, 0)$ est un semi-anneau complet. La distributivité s'exprime par :

$$a + \min\{b, c\} = \min\{a + b, a + c\}.$$

3) Soit A un alphabet (voir chapitre 9) et A^* l'ensemble des mots sur A. Alors $(\mathcal{P}(A^*), \cup, \cdot, \emptyset, \epsilon)$ est un semi-anneau complet.

4) Soit $\mathcal{R} = \mathbb{R} \cup \{+\infty, -\infty\}$, avec $(+\infty) + (-\infty) = +\infty$. Alors $(\mathcal{R}, \min, +, +\infty, 0)$ est un semi-anneau complet.

5) Soit $\mathcal{N} = \mathbb{N} \cup \{+\infty\}$. Alors $(\mathcal{N}, \max, \min, 0, +\infty)$ est un semi-anneau complet. On a $\max\{a, +\infty\} = +\infty$, $\max\{0, a\} = a$ et

$$\min\{a, \max\{b, c\}\} = \max\{\min\{a, b\}, \min\{a, c\}\}.$$

Dans un semi-anneau complet, on définit l'*étoile* a^* d'un élément a par :

$$a^* = 1 \oplus a \oplus a^2 \oplus \ldots \oplus a^n \oplus \ldots$$

avec $a^0 = 1$, et $a^{n+1} = a \otimes a^n$. Dans le semi-anneau de Boole, on a $a^* = 1$ pour tout a. Dans celui de l'exemple 2), on a $a^* = 0$. Dans celui de l'exemple 4), on a

$$a^* = \min_{n \in \mathbb{N}} na = \begin{cases} 0 & \text{si } a \geq 0, \\ -\infty & \text{sinon.} \end{cases}$$

Les coûts des chemins

Soit maintenant $G = (S, A)$ un graphe et soit K un semi-anneau complet. Soit $c : A \mapsto K$ une application qui à un arc u de A associe son *coût* $c(u)$. On étend c aux chemins comme suit : si $\gamma = (x_0, \ldots, x_n)$ est un chemin, alors

$$c(\gamma) = c(x_0, x_1) \otimes c(x_1, x_2) \otimes \ldots \otimes c(x_{n-1}, x_n).$$

(Notons que pour $n = 0$, $c(\gamma) = 1$). Si Γ est un ensemble de chemins, on pose :

$$c(\Gamma) = \bigoplus_{\gamma \in \Gamma} c(\gamma)$$

(et en particulier $c(\emptyset) = 0$). Le problème général qui nous intéresse est le calcul des éléments

$$c_{ij} = \bigoplus_{\gamma \in \Gamma_{ij}} c(\gamma)$$

où Γ_{ij} est l'ensemble des chemins de i à j dans G. Revenons sur nos exemples. Soit $G = (S, A)$ un graphe.

1) Avec le semi-anneau de Boole, fixons le coût d'un arc égal à 1. On a alors :

$$c_{ij} = \begin{cases} 1 & \text{si } \Gamma_{ij} \neq \emptyset, \\ 0 & \text{sinon.} \end{cases}$$

Ainsi, $c_{ij} = 1$ si et seulement si j est accessible de i.
2) Avec le semi-anneau de Floyd $(\mathcal{N}, \min, +, \infty, 0)$, le coût d'un chemin est la *somme* des coûts des arcs successifs qui le composent, le coût c_{ij} est le minimum des coûts des chemins de i à j. Ce coût est $+\infty$ s'il n'y a pas de chemin de i à j.
3) Avec la terminologie du chapitre 9, c_{ij} est l'ensemble des mots qui sont les étiquettes d'un chemin de i à j.
4) Avec le semi-anneau $(\mathcal{R}, \min, +, +\infty, 0)$, à nouveau c_{ij} est le minimum des coûts des chemins de i à j. Notons que s'il existe un circuit de coût négatif passant par le sommet i, alors $c_{ii} = -\infty$.
5) Appelons *capacité* d'un arc u le nombre $c(u)$. La *capacité d'un chemin* est la capacité minimale des arcs qui le composent et pour deux sommets i et j, c_{ij} est la *capacité maximale* d'un chemin de i à j.

Nous allons voir que l'algorithme de Roy-Warshall (voir section 4.2.1) s'étend au calcul des valeurs c_{ij}.

Soit $G = (S, A)$, avec $S = \{1, \ldots, n\}$. Rappelons que $E(k) = \{1, \ldots, k\}$ et que $I(\gamma)$ est l'intérieur d'un chemin γ. Notons Γ_{ij}^k l'ensemble des chemins γ dont l'intérieur est contenu dans $E(k)$. Soit $c : A \mapsto K$ une fonction de coût dans un semi-anneau complet K. On pose :

$$c_{ij}^k = c(\Gamma_{ij}^k) = \bigoplus_{\gamma \in \Gamma_{ij}^k} c(\gamma).$$

Lemme 2.6. *On a, pour i et j, dans S*

$$c_{ij}^k = c_{ij}^{k-1} \oplus c_{ik}^{k-1} \otimes c_{kk}^{k-1^*} \otimes c_{kj}^{k-1}. \tag{2.1}$$

Preuve. Notons D le membre droit de l'équation 2.1. On a en vertu de la distributivité généralisée

$$c_{kk}^{k-1^*} = \bigoplus_{m \in \mathbb{N}} \bigoplus_{\{\gamma_1, \ldots, \gamma_m\} \in \Gamma_{kk}^{k-1}} c(\gamma_1) \otimes c(\gamma_2) \ldots \otimes c(\gamma_m)$$

et par conséquent

$$D = \bigoplus_{\gamma \in \Gamma_{ij}^{k-1}} c(\gamma) \oplus \bigoplus_{\alpha \in \Gamma_{ik}^{k-1}, \delta \in \Gamma_{kj}^{k-1}} \bigoplus_{m \in \mathbb{N}} \bigoplus_{\gamma_1, \ldots, \gamma_m \in \Gamma_{kk}^{k-1}} c(\alpha) \otimes c(\gamma_1) \otimes c(\gamma_2) \ldots \otimes c(\gamma_m) \otimes c(\delta).$$

Tout chemin $\gamma \in \Gamma_{ij}^k$ est soit chemin de Γ_{ij}^{k-1}, soit se décompose *de manière unique* en $\gamma = \alpha(\gamma_1, \ldots, \gamma_m)\delta$, où $\alpha \in \Gamma_{ik}^{k-1}$, $\delta \in \Gamma_{kj}^{k-1}$, $m \in \mathbb{N}$, $\gamma_1, \ldots, \gamma_m \in \Gamma_{kk}^{k-1}$. D'où le résultat. ∎

Dans les cas particuliers des semi-anneaux K tels que $a^* = 1$ pour tout $a \in K$, la formule 2.1 se simplifie en :

$$c_{ij}^k = c_{ij}^{k-1} \oplus c_{ik}^{k-1} \otimes c_{kj}^{k-1}.$$

Il en est ainsi pour l'accessibilité et pour les plus courts chemins sans circuit de coût négatif. Dans ce cas la proposition 2.2 reste vraie et l'algorithme de Roy-Warshall se transpose directement en remplaçant l'avant-dernière ligne par :

$$a_{ij}^k := a_{ij}^{k-1} \oplus (a_{ik}^{k-1} \otimes a_{kj}^{k-1}).$$

On obtient alors le théorème :

Théorème 2.7. *Le calcul des c_{ij}, pour $i, j \in S$, se fait par l'algorithme de Roy-Warshall en $O(n^3)$ opérations $\oplus$, $\otimes$ et $*$, où n est le nombre de sommets du graphe.*

Preuve. (immédiate d'après ce qui précède.) ∎

4.2.4 Forte connexité

Soit $G = (S, A)$ un graphe orienté. La relation d'accessibilité définie précédemment est un préordre sur S (réflexive et transitive). Si l'on note $\rightarrow$ cette relation, la relation d'équivalence sur S induite par ce préordre, que l'on notera $\leftrightarrow$ est définie par : $i \leftrightarrow j$ si $i \rightarrow j$ et $j \rightarrow i$. Les classes d'équivalence de la relation $\leftrightarrow$ s'appellent les *composantes fortement connexes* de G. Deux sommets distincts appartiennent à une même classe si et seulement s'ils appartiennent à un même circuit. La relation induite par $\rightarrow$ sur l'ensemble quotient des classes d'équivalence définit le *graphe quotient* de G par $\rightarrow$. On appelle alors *graphe réduit* de G le graphe obtenu à partir du graphe quotient en supprimant ses boucles. Ce graphe est *sans circuit* car l'existence d'un circuit passant par deux classes distinctes α et β entraînerait l'équivalence pour $\leftrightarrow$ d'un sommet quelconque de α et d'un sommet quelconque de β, d'où la contradiction. Le graphe de la figure 2.3 possède deux composantes fortement connexes $C' = \{1, 2, 3\}$ et $C'' = \{4, 5\}$. La détermination des composantes fortement connexes d'un graphe est un problème fréquent dans l'analyse des graphes d'état (chaînes de Markov, réseaux de Petri,...) modélisant l'évolution d'un système. On distingue alors souvent les classes «finales» qui correspondent aux sorties du graphe réduit (dès que l'état du système est un sommet d'une classe finale, les états postérieurs appartiendront à cette classe) et les autres dont les sommets sont dits «transitoires». Pour l'exemple de la figure 2.3, C'' est la seule classe finale et les états $\{1, 2, 3\}$ sont transitoires. On verra au paragraphe 4.4.5 un algorithme efficace de calcul des composantes fortement connexes.

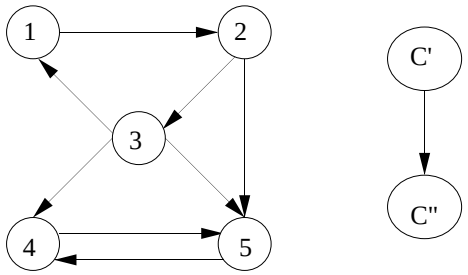


Figure 2.3: *Un graphe orienté et son graphe réduit.*

4.3 Arbres et arborescences

Les arbres représentent une structure *minimale* en nombre de liaisons pour connecter un ensemble de sommets. A ce titre, ils constituent une classe fondamentale de graphes intervenant dans de nombreux problèmes d'optimisation de réseaux. Les arborescences fournissent des structures de données à la base de nombreux algorithmes performants.

4.3.1 Arbres

Nous considérons dans ce paragraphe des graphes non orientés, et nous rappelons que dans un graphe non orienté, la longueur d'un cycle est supérieure ou égale à trois.

Soit G un graphe à n sommets ($n \geq 1$) ; nous allons prouver que les six propriétés ci-dessous sont équivalentes. Un graphe vérifiant l'une de ces propriétés est un arbre.

1) G est un graphe connexe sans cycle ;
2) G est un graphe connexe possédant $n-1$ arêtes ;
3) G est un graphe sans cycle possédant $n-1$ arêtes ;
4) G est un graphe tel que deux sommets quelconques sont liés par une seule chaîne ;
5) G est un graphe connexe qui perd sa connexité par suppression d'une arête quelconque ;
6) G est un graphe sans cycle tel que l'adjonction d'une arête quelconque crée un cycle et un seul.

Pour démontrer l'équivalence de ces définitions, deux lemmes préliminaires seront utiles.

Lemme 3.1. *Un graphe connexe à n sommets possède au moins $n-1$ arêtes.*

Preuve. (Induction sur n.) La propriété est vraie pour $n = 1$. Supposons $n \geq 2$ et soit $G = (S, A)$ un graphe connexe à n sommets. Considérons un sommet u de S. Notons G_u le sous-graphe de G induit par $S - \{u\}$ et soient C_1,C_2,...,C_p

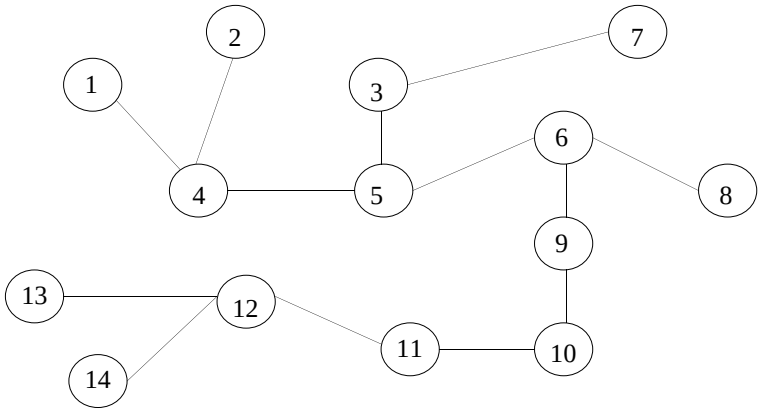


Figure 3.1: *Un arbre à quatorze sommets.*

les composantes connexes de G_u. Sur la figure 3.2 sont représentés à gauche un graphe orienté G et à droite le graphe G_u. Pour tout $k = 1, \ldots, p$, il existe au moins une arête liant u à un sommet de C_k, sinon G ne serait pas connexe. D'autre part les sous-graphes G^k de G induits par les C_k sont connexes. On a donc par induction $m_k \geq n_k - 1$, où n_k (respectivement m_k) désigne le nombre de sommets (respectivement d'arêtes) de G^k. Il en résulte que :

$$m \geq (\sum_{k=1}^{p} m_k) + p \geq \sum_{k=1}^{p} n_k = n - 1.$$ ∎

Lemme 3.2. *Un graphe à n sommets ayant au moins n arêtes possède un cycle.*

Preuve. (Induction sur n.) La propriété est vraie pour $n \leq 3$. Soit donc $n \geq 4$ et $G = (S, A)$ un graphe à n sommets possédant m arêtes ($m \geq n$). Supposons en raisonnant par l'absurde que G soit sans cycles et considérons un sommet u de S. Notons G_u le sous-graphe de G induit par $S - \{u\}$, C_1,C_2,...,C_p les composantes connexes de G_u et G^k le sous-graphe de G induit par C_k (voir figure 3.2). Chaque G^k est sans cycle, donc $m_k \leq n_k - 1$ et le nombre d'arêtes incidentes à u est $m - \sum_{k=1}^{p} m_k \geq m + p - \sum_{k=1}^{p} n_k \geq p + 1$. Il existe donc un k dans $\{1, \ldots, p\}$ tel que u soit adjacent à deux sommets distincts a et b de C_k. Le graphe G^k étant connexe, il existe une chaîne de a à b dans G^k et donc un cycle dans G passant par u, a et b. ∎

Proposition 3.3. *Soit G un graphe à n sommets, les six propriétés suivantes sont équivalentes :*

(1) *G est un graphe connexe sans cycle;*
(2) *G est un graphe connexe possédant $n - 1$ arêtes;*
(3) *G est un graphe sans cycle possédant $n - 1$ arêtes;*
(4) *G est un graphe tel que deux sommets quelconques sont liés par une seule chaîne;*

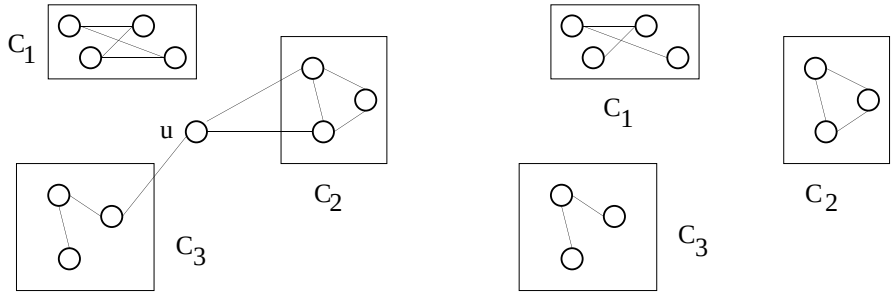


Figure 3.2: *Le graphe G_u.*

(5) *G est un graphe connexe qui perd la connexité par suppression d'une arête quelconque;*
(6) *G est un graphe sans cycle tel que l'adjonction d'une arête quelconque crée un cycle et un seul.*

Preuve. Nous montrons que $1 \Rightarrow 2 \Rightarrow 3 \Rightarrow 4 \Rightarrow 5 \Rightarrow 6 \Rightarrow 1$.

$1 \Rightarrow 2$. Soit G un graphe à n sommets connexe et sans cycle. D'après le lemme 1, on a $m \geq n-1$ et d'après le lemme 2 on a $m \leq n-1$, d'où il résulte que $m = n-1$.

$2 \Rightarrow 3$. Un graphe G connexe et possédant $n-1$ arêtes est sans cycle car dans le cas contraire, la suppression d'une arête quelconque du cycle entraînerait l'existence d'un graphe connexe à n sommets et $n-2$ arêtes.

$3 \Rightarrow 4$. Supposons qu'un graphe G à $n-1$ arêtes soit sans cycle et ne soit pas connexe. Le graphe induit par chaque composante connexe qui est connexe et sans cycle possède un sommet de plus que d'arêtes. Le nombre de composantes connexes de G étant au moins deux, le graphe G lui même possède au plus $n-2$ arêtes. De plus l'existence de deux chaînes distinctes entre deux sommets entraîne celle d'un cycle dans G.

$4 \Rightarrow 5$. Soit G un graphe tel que pour toute paire de sommets, il existe une chaîne et une seule ayant ces deux sommets pour extrémités. Le graphe G est bien sûr connexe. Si G ne perdait pas la connexité par suppression d'une arête $\{a, b\}$, une chaîne n'utilisant pas cette arête lierait a et b, ce qui contredit l'hypothèse d'unicité de la chaîne liant a et b.

$5 \Rightarrow 6$. Si G est connexe et perd la connexité par suppression d'une arête quelconque, il est sans cycle car dans le cas contraire, la suppression d'une arête du cycle ne lui ferait pas perdre la connexité. De plus l'adjonction d'une arête $\{a, b\}$ crée un cycle puisqu'il existe déjà une chaîne liant a et b dans G. Ce cycle passe par l'arête $\{a, b\}$ puisque G est sans cycle. De plus si l'adjonction de $\{a, b\}$ créait deux cycles distincts, il existerait dans G deux chaînes distinctes de a à b et donc un cycle.

$6 \Rightarrow 1$. Si l'adjonction d'une arête quelconque $\{a, b\}$ crée un cycle, G est connexe car dans le cas contraire, l'adjonction d'une arête entre deux sommets appartenant

à deux composantes connexes distinctes ne créerait pas de cycles. ∎

4.3.2 Arborescences

On obtient une arborescence à partir d'un arbre en distinguant l'un de ses sommets. Une *arborescence* est donc un couple formé d'un arbre H et d'un sommet distingué r appelé *racine*. Une arborescence (H, r) peut aussi être définie naturellement en tant que graphe orienté en substituant l'arc (u, v) à l'arête $\{u, v\}$ si la chaîne de r à v dans H passe par u, l'arc (v, u) sinon. Une arborescence de racine r est alors un graphe orienté tel qu'il existe un chemin *unique* de r à n'importe quel sommet.

L'une des propriétés les plus importantes d'une arborescence est sa structure récursive que nous mettons en évidence par le lemme suivant :

Lemme 3.4. *Soit $A = (H, r)$ une arborescence possédant au moins deux sommets, H_r le graphe induit par $S - \{r\}$ et $r_1, \ldots, r_k$ les sommets adjacents à la racine. Les sous-graphes induits par les composantes connexes de H_r constituent k arbres disjoints contenant chacun un et un seul des sommets r_i.*

Preuve. Chaque composante connexe de H_r contient un et un seul r_i car H ne possède pas de cycle. Les k sous-graphes induits par ces composantes qui sont connexes et sans cycle sont des arbres disjoints. ∎

La figure 3.3 illustre la structure récursive d'une arborescence.

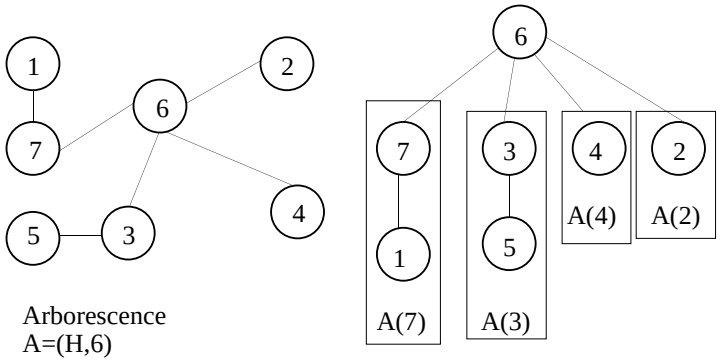


Figure 3.3: *Structure récursive d'une arborescence.*

Etant donnée l'importance des structures arborescentes, certaines notations spécifiques ont été créées. Les *ascendants* et les*descendants* d'un sommet ont leur définition usuelle pour l'arborescence considérée comme un graphe orienté. Tout sommet s (sauf la racine) a un prédécesseur unique noté $p(s)$ et appelé *père* de s. Inversement les successeurs d'un sommet s sont ses *fils*. Si le sommet s n'a pas de fils, il est appelé *feuille* (ou encore *sommet externe*, dans le cas contraire, c'est un

noeud (ou encore *sommet interne*). La *profondeur* d'un sommet est la longueur du chemin de la racine à ce sommet. La *hauteur* d'un sommet u est la longueur maximale d'un chemin d'origine u. Si u et v sont deux sommets, l'*ancêtre* de u et v est l'ascendant commun de u et v de profondeur maximale.

Par conformité avec les six définitions initiales (en particulier : $m = n - 1$), tous les arbres que nous avons définis jusqu'ici possèdent au moins un sommet. Il est cependant commode, surtout dans le cadre des applications informatiques liées aux structures de données et leur programmation (par exemple l'utilisation du pointeur « nil » du langage Pascal), de considérer qu'un arbre vide est aussi un arbre. Sauf mention contraire, nous adopterons désormais cette convention et noterons Λ une arborescence vide.

4.3.3 Arborescences ordonnées

Il est souvent utile de définir pour chaque sommet d'une arborescence un ordre total sur les fils de ce sommet. C'est par exemple le cas pour les arbres 2-3 (voir chapitre 6) ou encore les arbres de dérivation dans une grammaire. Si l'on adjoint à chaque sommet interne d'une arborescence un ordre total sur les fils de ce sommet, on définit une *arborescence ordonnée*. Les relations d'ordre locales permettent de munir l'ensemble des sommets de l'arborescence d'un ordre total.

Soit A une arborescence ordonnée, la liste L représentant l'ordre total sur les sommets de A est construite récursivement comme suit : si A est l'arborescence vide, la liste L est vide; sinon soient $L_1, \ldots, L_k$ les listes associées aux ordres totaux pour les sous-arborescences de racines $r_1, \ldots, r_k$ et $(r_1, \ldots, r_k)$ la liste associéee à l'ordre total sur les fils $r_1, \ldots, r_k$ de la racine, la liste L est égale à $(r) \cdot L_1 \cdot L_2 \cdots L_k$ où l'on note $L \cdot L'$ la concaténation des deux listes L et L'.

Remarque. La liste associée à l'ordre total sur les sommets d'une arborescence ordonnée est une liste préfixe au sens défini dans la section 4.4.5.

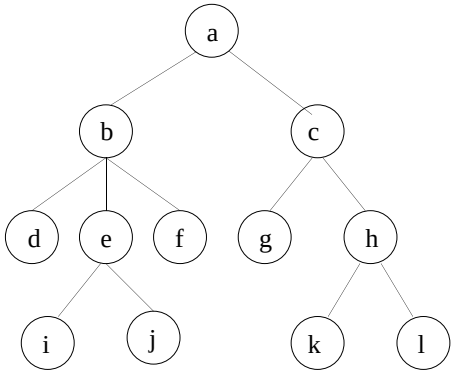


Figure 3.4: *Une arborescence.*

La figure 3.4 représente une arborescence ordonnée dans laquelle, comme il est d'usage, les fils d'un même sommet sont dessinés de la gauche vers la droite dans l'ordre croissant. L'ordre total est donc $(a, b, d, e, i, j, f, c, g, h, k, l)$.

4.3.4 Arbres positionnés et arbres binaires

Un *arbre positionné d'arité p* est une arborescence telle que les arcs liant les frères d'un même sommet à leur père sont étiquetés par des éléments distincts de l'ensemble $\{1, \ldots, p\}$. On parle alors du $k^{ième}$ fils si l'arc a l'étiquette k. Un arbre positionné d'arité p est *complet* si chacun de ses noeuds possède p fils.

Un *arbre binaire* est un arbre positionné d'arité 2. Au lieu de l'étiqueter sur $\{1, 2\}$, il est plus parlant d'utiliser les étiquettes « gauche » et « droit ». Chaque nœud possède donc soit un fils « droit », soit un fils « gauche », soit les deux.

Une définition récursive des arbres binaires, fondamentale dans les applications informatiques, s'énonce comme suit :

(1) l'arbre vide est un arbre binaire ;
(2) soient A_1 et A_2 deux arbres binaires et s un sommet n'appartenant ni à A_1 ni à A_2, alors l'arborescence de racine s, de sous-arbre gauche A_1 et de sous-arbre droit A_2 est un arbre binaire.

La distinction entre fils gauche et fils droit entraîne quelques notations supplémentaires. Soit A un arbre binaire complet et u l'un de ses sommets internes, $A_g(u)$ (respectivement $A_d(u)$) désigne le sous-arbre de A dont la racine est le fils gauche de u (respectivement le fils droit de u). Si u est la racine on note A_g (respectivement A_d) le sous-arbre gauche (respectivement le sous-arbre droit) de A. Le sous-arbre de racine u est noté $A(u)$. La figure 3.5 présente un exemple d'arbre binaire.

4.3.5 Arbre binaire complet

Un *arbre binaire complet* est un arbre binaire non vide et complet, c'est-à-dire tel que chaque noeud possède zéro ou deux fils. La figure 3.5 représente un arbre binaire complet. La définition récursive suivante des arbres binaires complets est également très utile :

(1) une arborescence réduite à sa racine est un arbre binaire complet ;
(2) soient A_1 et A_2 deux arbres binaires complets et s un sommet n'appartenant ni à A_1 ni à A_2, l'arborescence A de racine s, de sous-arbre gauche A_1 et de sous-arbre droit A_2 est un arbre binaire complet.

La figure 3.6 montre les premiers arbres binaires complets. La proposition simple qui suit illustre le raisonnement par induction fréquemment pratiqué sur les arbres binaires complets.

Proposition 3.5. *Un arbre binaire complet à p ($p \geq 1$) sommets externes possède $p - 1$ sommets internes.*

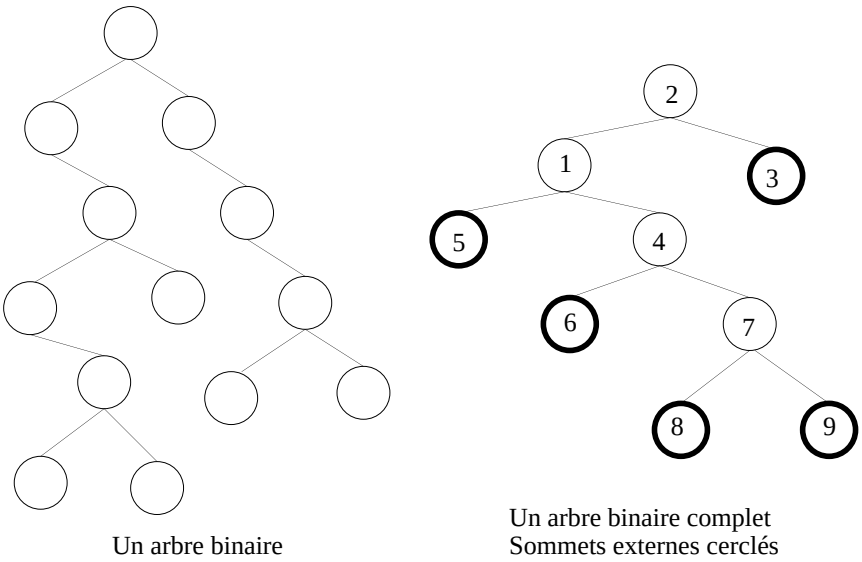


Figure 3.5: *Arbre binaire et arbre binaire complet.*

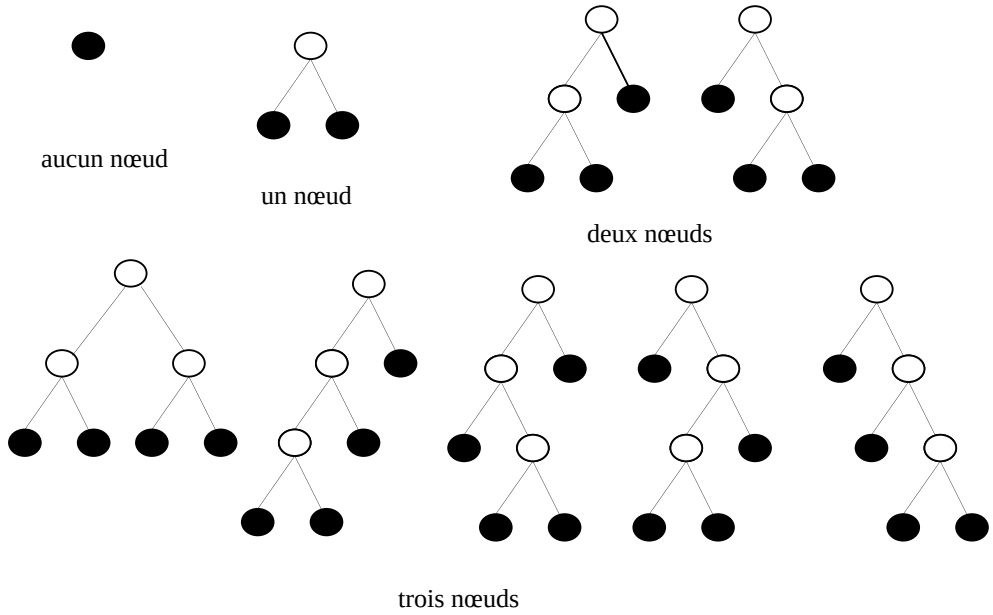


Figure 3.6: *Les premiers arbres binaires complets.*

Preuve. (Induction sur p.) Si $p = 1$, l'arbre binaire complet est réduit à un sommet qui est externe et la propriété est vraie. Soient $p \geq 2$ et A un arbre binaire complet à p sommets externes. L'arbre A est formé d'une racine r, d'un sous-arbre gauche A_1 qui est un arbre binaire complet à p_1 sommets externes ($p_1 \geq 1$) et d'un sous-arbre droit A_2 qui est un arbre binaire complet à p_2 sommets externes ($p_2 \geq 1$). On a $p_1 + p_2 = p$ et par récurrence le nombre de sommets internes de A_1 est donc $p_1 - 1$ et celui de A_2 est $p_2 - 1$. Le nombre de sommets internes de A est donc $1 + (p_1 - 1) + (p_2 - 1) = p - 1$. ∎

Les arbres binaires complets ne constituent en fait qu'une représentation plus structurée des arbres binaires. En effet, l'opération d'*effeuillage* permet d'associer à un arbre binaire complet un arbre binaire (sous-jacent). L'image résultant de l'effeuillage d'un arbre binaire complet A est le sous-graphe de A induit par ses sommets internes. L'effeuillage n'est pas une opération injective mais on peut montrer que tous les arbres binaires complets ayant la même image sont isomorphes. Ils ne diffèrent en fait que par les noms des feuilles supprimées. La figure 3.7 représente un arbre binaire complet et à sa droite l'arbre effeuillé.

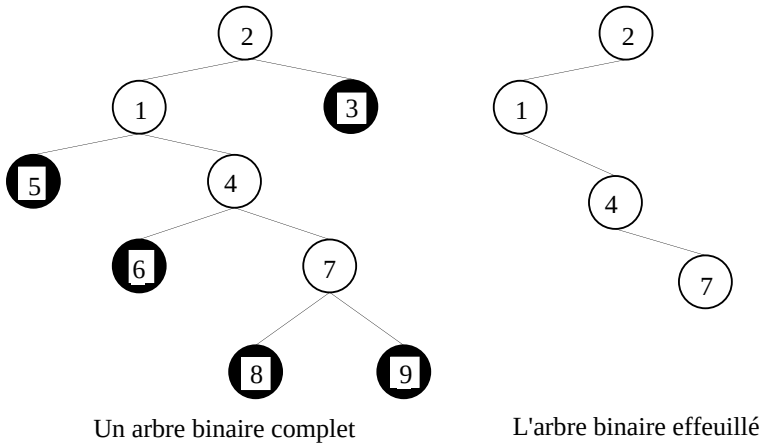


Figure 3.7: *Effeuillage d'un arbre binaire complet.*

L'opération réciproque appelée *complétion* consiste à compléter à deux le nombre de fils de chaque sommet de A. Cette définition ne précise cependant pas la règle de choix des noms des nouveaux sommets. La figure 3.8 montre l'opération de complétion.

4.4 Parcours d'un graphe

4.4.1 Parcours d'un graphe non orienté

Dans cette section nous considérons un graphe non orienté connexe $G = (S, A)$. Un *parcours* de G à partir de l'un de ses sommets s est une liste de sommets L telle que :

- le premier sommet de L est s,
- chaque sommet de S apparaît une fois et une seule dans L,
- tout sommet de la liste (sauf le premier) est adjacent à au moins un sommet placé avant lui dans la liste.

Nous présentons d'abord un algorithme générique de calcul d'un parcours. Nous étudions ensuite les deux types de parcours les plus utilisés : les parcours *en profondeur* et les parcours *en largeur*.

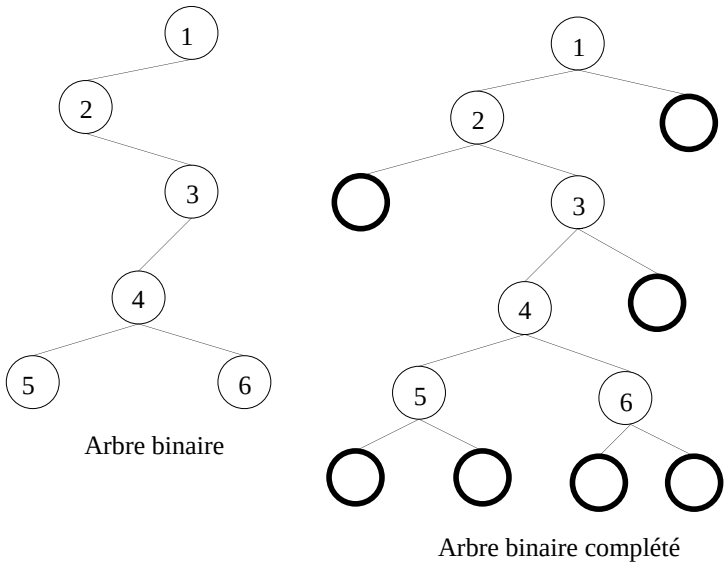


Figure 3.8: *Complétion d'un arbre binaire.*

Définitions et notations

Soit T une partie non vide de S, la *bordure* $\mathcal{B}(T)$ de T est l'ensemble des sommets de $S - T$ adjacents à T. Pour le graphe de la figure 4.1, la bordure de $\{3, 5, 6\}$ est $\{1, 2, 4\}$.

Soit L une liste de sommets de G. Le *support* de L, noté $\sigma(L)$, est l'ensemble des sommets présents dans la liste (par exemple $\sigma(1, 2, 1, 3, 5, 3, 3) = \{1, 2, 3, 5\}$). La liste des k premiers sommets de L est notée L_k. Un sommet u de L est *fermé* si tous ses voisins dans G appartiennent à $\sigma(L)$, dans le cas contraire il est *ouvert*. Comme les listes qui interviennent dans ce chapitre ne contiennent qu'une seule occurrence de chaque élément de leur support, nous utiliserons la notation (abusive mais plus simple) $\mathcal{B}(L)$ à la place de $\mathcal{B}(\sigma(L))$.

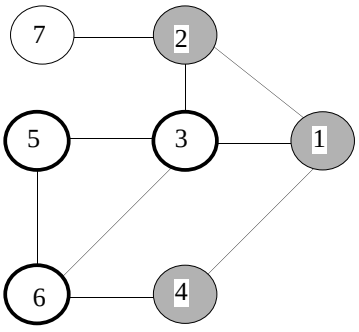


Figure 4.1: *Bordure d'un sous-ensemble de sommets.*

Algorithme générique et propriétés des parcours

Nous donnons dans cette section un algorithme générique de construction d'un parcours. Nous présentons ensuite les propriétés fondamentales des parcours et nous proposons un premier niveau d'implémentation de l'algorithme générique.

Considérons la procédure PARCOURS-GÉNÉRIQUE(G, s) ci-dessous :

```
procédure PARCOURS-GÉNÉRIQUE(G, s);
  L := (s);
  pour k de 1 à n − 1 faire
    choisir un sommet v dans B(L);
    L := L · (v)
  fintantque.
```

La proposition suivante établit l'égalité entre l'ensemble des listes construites par la procédure et l'ensemble des parcours de G.

Proposition 4.1. *Les parcours de G sont exactement les listes construites par* PARCOURS-GÉNÉRIQUE(G, s).

Preuve. Soit L une liste de PARCOURS-GÉNÉRIQUE(G, s). Montrons par induction sur le nombre d'itérations que la liste L_k obtenue après k itérations satisfait l'invariant suivant : « L_k est un parcours du sous-graphe de G induit par $\sigma(L_k)$ ». La propriété est vraie pour $k = 0$. Considérons l'itération k ($0 < k \leq n-1$) et soit u un sommet de l'ensemble non vide $S - \sigma(L_{k-1})$. Comme G est connexe, il existe une chaîne élémentaire du sommet s de $\sigma(L_{k-1})$ au sommet u de $S - \sigma(L_{k-1})$ dont l'une des arêtes $\{x, y\}$ vérifie $x \in S - \sigma(L_{k-1})$ et $y \in \sigma(L_{k-1})$. Le choix du sommet x est donc possible à l'itération k tant que $k < n$. Comme d'une part L_{k-1} est un parcours du sous-graphe de G induit par $\sigma(L_{k-1})$ (hypothèse d'induction) et que d'autre part le sommet x appartient à $\mathcal{B}(L_{k-1})$, la liste $L_k = L_{k-1} \cdot (x)$ est un parcours du sous-graphe de G induit par $\sigma(L_k) = \sigma(L_{k-1}) \cup \{x\}$. La liste L_{n-1} est donc un parcours de G.

Réciproquement soit $L = (x_1, \ldots, x_n)$ un parcours de G à partir de s. Par définition d'un parcours, x_{k+1} appartient à $\mathcal{B}(\{x_1, \ldots, x_k\}) = \mathcal{B}(L_{k-1})$ pour k de 1 à $n - 1$. L est donc aussi la liste obtenue par PARCOURS-GÉNÉRIQUE en choisissant le sommet x_{k+1} à l'itération k. ∎

Soit $L = (x_1, \ldots, x_n)$ un parcours de G. Si nous associons à chaque sommet x_k ($k \in \{2, \ldots, n\}$) un sommet x'_k appartenant à $\mathcal{B}(\{x_k\}) \cap \{x_1, \ldots, x_{k-1}\}$, le sous-graphe de G induit par les arêtes $\{x_k, x'_k\}$, appelées *arêtes de liaison*, est un arbre couvrant de G. Cette propriété est très importante car elle montre qu'un parcours emprunte un nombre minimal d'arêtes de liaison.

Proposition 4.2. *Soit $L = (x_1, \ldots, x_n)$ un parcours de G. Pour tout k, $(2 \leq k \leq n)$, soit x'_k un sommet de $\{x_1, \ldots, x_{k-1}\}$ adjacent à x_k. Le sous-graphe induit par les arêtes $\{x_k, x'_k\}$, $(2 \leq k \leq n)$, est un arbre couvrant de G.*

Preuve. Nous notons G_p le sous-graphe de G induit par les arêtes $\{x_k, x'_k\}$ ($k \in \{1, \ldots, p\}$) et nous raisonnons par récurrence sur p. La propriété est vraie pour $p = 1$ car on a toujours $x'_1 = s$, et G_1 est donc réduit à l'arête $\{s, x_2\}$. Supposons $p \geq 2$; le graphe G_p est connexe car obtenu par adjonction à l'arbre G_{p-1} (hypothèse d'induction) d'un nouveau sommet x_{p+1} et d'une nouvelle arête incidente à x_{p+1} et à un sommet de G_{p-1}. Le graphe G_p qui est connexe et possède $p-1$ arêtes et p sommets (car G_{p-1} possède $p-2$ arêtes et $p-1$ sommets) est donc un arbre. Il en résulte que G_n est un arbre couvrant de G. ∎

La figure 4.2 représente le parcours $(1, 3, 4, 6, 7, 2, 5)$ d'un graphe non orienté et montre l'arbre couvrant associé (arêtes épaisses).

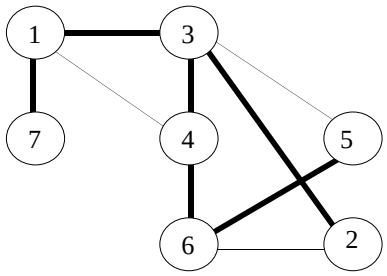


Figure 4.2: *Parcours d'un graphe et arbre couvrant.*

Avant d'étudier des parcours particuliers, nous proposons une implémentation partielle de l'algorithme générique décrivant le calcul des sommets ouverts. Nous appelons *degré résiduel* d'un sommet x, et nous le notons $r(x)$, le nombre de voisins de x qui n'appartiennent pas à la liste en cours. Un sommet de la liste en cours (nous dirons encore *sommet visité*) est donc ouvert si et seulement si son degré résiduel est non nul.

```
procédure PARCOURS(G, s);
  pour tout sommet x dans S faire
    r(x) := d(x); FERMER(x)
  finpour;
  {d(x) est le degré de x dans G}
  OUVRIR(s); EXAMINER-VOISINS(s);
  tantqu'il reste un sommet non visité faire
    choisir une arête de liaison {x, y} où x est ouvert et y non visité;
    L := L · (y);
    si r(y) > 0 alors
      OUVRIR(y);EXAMINER-VOISINS(y)
    finsi
  fintantque.
```

A chaque étape, l'algorithme PARCOURS ci-dessus choisit une arête de liaison $\{x, y\}$ où x est un sommet ouvert et y un sommet non visité, ouvre le sommet y si $r(y) > 0$ et diminue d'une unité le degré résiduel de chacun des voisins de y dans G. L'algorithme utilise un tableau booléen pour gérer l'ensemble des sommets visités, un tableau booléen pour gérer l'ensemble des sommets ouverts, un tableau linéaire pour gérer les degrés résiduels et une liste des voisins pour coder G.

La procédure EXAMINER-VOISINS(y) fait la mise à jour des degrés résiduels et des sommets ouverts à chaque visite d'un nouveau sommet. Sa complexité est $O(d(y))$.

```
procédure EXAMINER-VOISINS(y);
pour tout voisin z de y dans G faire
  r(z) := r(z) − 1;
  si z est visité et r(z) = 0 alors FERMER(z)
finpour.
```

Dans la mesure où l'algorithme de choix de l'arête de liaison n'est pas précisé, il n'est pas possible d'évaluer la complexité globale de l'algorithme PARCOURS. Cependant, comme chaque sommet est visité une fois et une seule, la procédure EXAMINER-VOISINS est également exécutée une fois et une seule pour chaque sommet. Il en résulte la proposition suivante :

Proposition 4.3. *Dans l'algorithme* PARCOURS, *la complexité des opérations autres que le choix de l'arête de liaison est* $O(n + m)$.

Nous allons désormais étudier deux types de parcours en particularisant la règle de choix des arêtes de liaison.

4.4.2 Parcours en profondeur

Un parcours L du graphe G à partir de s est dit *en profondeur* si, à chaque étape, l'arête de liaison $\{x, y\}$ choisie est telle que le sommet x soit le dernier sommet visité ouvert.

Une implémentation possible de l'algorithme de choix consiste alors à utiliser une pile de sommets visités posssédant les propriétés suivantes :

- tous les sommets ouverts sont dans la pile,
- si un sommet ouvert x a été visité avant un sommet ouvert y, alors y est placé plus haut que x dans la pile.

A partir d'une pile contenant initialement le seul sommet s, les opérations sur la pile, à chaque étape du parcours, sont les suivantes :

a) Si le sommet de pile t est ouvert, une arête $\{t, y\}$ est choisie comme arête de liaison et on empile y ;
b) Si le sommet de pile est fermé, on procède à des dépilements tant que le sommet de pile est fermé (et la pile non vide).

La procédure PROFONDEUR(G, s) ci-dessous réalise ces opérations de pile pour choisir les arêtes de liaison.

```
procédure PROFONDEUR(G, s) ;
  CRÉER(Π) ;
  pour tout sommet x dans S faire
    r(x) := d(x) ; FERMER(x)
  finpour ;
  L := (s) ; OUVRIR(s) ; EMPILER(s, Π) ; EXAMINER-VOISINS(s) ;
  tantque Π est non vide faire
    t :=SOMMET(Π) ;
    si t est ouvert alors
      choisir une arête {t, y} telle que y soit non visité ;
      L := L · (y) ; EMPILER(y, Π) ;
      si r(y) > 0 alors OUVRIR(y) ; EXAMINER-VOISINS(y) finsi
    sinon DÉPILER(Π)
    finsi
  fintantque.
```

Nous proposons au lecteur, à titre d'exercice, de démontrer que la pile Π gérée par la procédure PROFONDEUR(G, s) possède les deux propriétés requises. On peut plus précisément montrer, par récurrence sur le nombre de sommets visités, que la liste des sommets contenus de bas en haut dans la pile lors de la visite du sommet x est la liste des sommets du chemin de s à x dans l'arborescence de liaison en cours. Cette propriété est intéressante car elle induit une autre implémentation possible de l'algorithme de choix pour un parcours en profondeur. Il suffit en effet

de construire la fonction père de l'arborescence de liaison au fur et à mesure de la création des arêtes de liaison. Si le sommet visité en cours x est fermé, on utilise alors la fonction père pour remonter au dernier sommet visité ouvert.

Sur le graphe de la figure 4.3, la procédure PROFONDEUR calcule le parcours en profondeur $(1, 3, 4, 6, 5, 2, 7)$. Les contenus de la pile et de la liste sont donnés

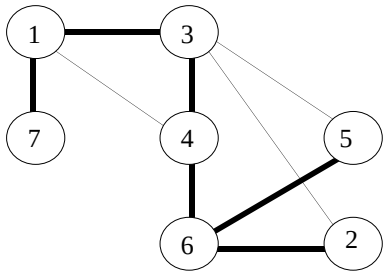


Figure 4.3: *Parcours en profondeur.*

dans le tableau ci-dessous. La première ligne correspond à la pile et la seconde à la liste :

(1)	(1,3)	(1,3,4)	(1,3,4,6)	(1,3,4,6,5)	(1,3,4,6)	(1,3,4,6,2)
(1)	(1,3)	(1,3,4)	(1,3,4,6)	(1,3,4,6,5)	(1,3,4,6,5)	(1,3,4,6,5,2)

(1,3,4,6)	(1,3,4)	(1,3)	(1)
(1,3,4,6,5,2)	(1,3,4,6,5,2)	(1,3,4,6,5,2)	(1,3,4,6,5,2)

(1,7)	(1)	()
(1,3,4,6,5,2,7)	(1,3,4,6,5,2,7)	(1,3,4,6,5,2,7)

Proposition 4.4. *La complexité en temps de la procédure* PROFONDEUR *est* $O(n + m)$.

Preuve. Evaluons d'abord la complexité des opérations sur la pile. Comme un sommet est visité une fois et une seule, il est empilé une fois et une seule. Il est donc dépilé au plus une fois et en fait exactement une fois en raison de la condition de terminaison. La complexité des opérations sur la pile est donc $O(n)$. Comme la complexité des opérations autres que le choix des arêtes de liaison dans l'algorithme PARCOURS est $O(n+m)$ (proposition 4.3), la complexité globale de la procédure PROFONDEUR est $O(n + m)$. ∎

Nous avons décrit jusqu'ici des algorithmes itératifs pour le calcul d'un parcours en profondeur à partir d'un sommet s. Nous présentons maintenant un algorithme récursif qui résout ce problème. Le principe de cet algorithme est de visiter s, puis de réaliser un appel récursif pour chaque voisin de s non encore visité. Un appel pour un sommet x est terminal si tous les voisins de x sont déja visités. La procédure PROFONDEUR-RÉCURSIF ci-dessous implémente cet algorithme.

```
procédure PROFONDEUR-RÉCURSIF(s);
  VISITER(s);
  pour tout voisin x de s faire
    si x est non visité alors
      VISITER(x); PROFONDEUR-RÉCURSIF(x)
    finsi
  finpour.
```

La procédure VISITER(x) met à jour le tableau des sommets visités et la liste des sommets visités. Cet algorithme, dont l'analyse est proposée en exercice, a une complexité $O(\max\{n, m\})$. L'une de ses variantes, utilisée pour le calcul des composantes fortement connexes d'un graphe orienté, est étudiée dans la section 4.4.5.

4.4.3 Parcours en largeur

Un parcours L du graphe G à partir de s est dit *en largeur* si, à chaque étape, l'arête de liaison $\{x, y\}$ choisie est telle que le sommet x soit le premier sommet visité ouvert.

Une implémentation possible de l'algorithme de choix consiste ici à utiliser une file de sommets visités posssédant les propriétés suivantes :

- tous les sommets ouverts sont dans la file,
- si un sommet ouvert x a été visité avant un sommet ouvert y, alors y est placé après x dans la file.

A partir d'une file contenant initialement le seul sommet s, les opérations sur la file, à chaque étape du parcours, sont les suivantes :

a) Si la tête de file t est un sommet ouvert, une arête $\{t, y\}$ est choisie comme arête de liaison et l'on enfile y;
b) Si la tête de file est un sommet fermé, on procède à des défilements tant que la tête de file est un sommet fermé (et la file non vide).

La procédure LARGEUR(G, s) ci-dessous réalise ces opérations de file pour choisir les arêtes de liaison.

```
procédure LARGEUR(G, s);
  CRÉER(Φ);
  pour tout sommet x dans S faire
    r(x) := d(x); FERMER(x)
  finpour;
  L := (s); OUVRIR(s); ENFILER(s, Φ); EXAMINER-VOISINS(s);
  tantque Φ est non vide faire
    t :=TÊTE(Φ);
    si t est ouvert alors
      choisir une arête {t, y} telle que y soit non visité;
      L := L · (y); ENFILER(y, Φ);
      si r(y) > 0 alors OUVRIR(y); EXAMINER-VOISINS(y) finsi
    sinon DÉFILER(Φ)
    finsi
  fintantque.
```

Sur le graphe de la figure 4.4, la procédure LARGEUR calcule le parcours en largeur $(1, 7, 4, 3, 6, 5, 2)$. Par un raisonnement analogue à celui fait pour la procédure PROFONDEUR, la complexité de la procédure LARGEUR est $O(n + m)$.

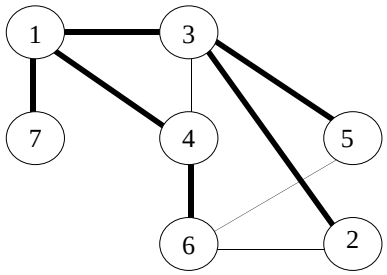


Figure 4.4: *Parcours en largeur.*

Parcours d'un graphe non orienté quelconque

Considérons le cas où le graphe G n'est pas connexe. Un parcours de G est défini comme une liste de sommets telle que :

- chaque sommet de S apparait une fois et une seule dans la liste,
- chaque sommet de la liste (sauf le premier) appartient à la bordure du sous-ensemble des sommets placés avant lui dans la liste, si toutefois cette bordure est non vide.

La condition de connexité n'est alors plus requise si les sommets de la liste en cours constituent une composante connexe de G.

Il résulte de cette définition qu'un parcours de G est une liste $L = L_1 \cdot L_2 \cdots L_p$ où les L_j sont des parcours des sous-graphes (connexes) induits par les p composantes connexes de G. Les arêtes de liaison constituent alors une famille de p arbres où chaque arbre couvre une composante connexe de G. Le parcours $(9, 7, 6, 10, 8, 1, 2, 3, 4, 5)$ est un parcours en profondeur du graphe de la figure 4.5. Les arêtes épaisses correspondent à des arbres couvrant les deux composantes connexes.

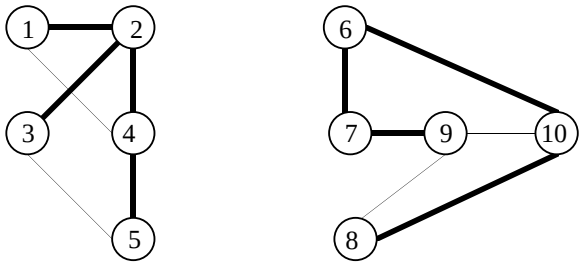


Figure 4.5: *Un graphe non orienté non connexe.*

4.4.4 Parcours d'un graphe orienté

Dans cette section nous considérons un graphe G orienté et *sans boucles*. Pour un graphe orienté, la *bordure* $\Gamma(T)$ d'une partie T de S est le sous-ensemble des sommets de $S - T$ qui sont les extrémités d'un arc dont l'origine est dans T. Si L est une liste de sommets de G, nous noterons encore abusivement $\Gamma(L)$ la bordure du support de L. Un sommet d'une liste L est *fermé* si tous ses successeurs dans G appartiennent à $\sigma(L)$, dans le cas contraire il est *ouvert*. Sur l'exemple de la figure 4.6, la bordure de $\{1, 2\}$ est $\{3, 4, 5\}$.

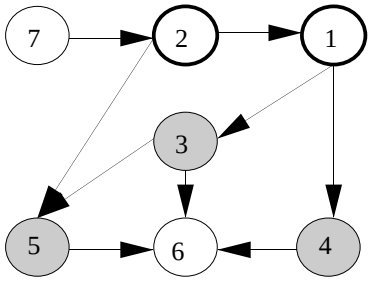


Figure 4.6: *Bordure pour un graphe orienté.*

Un parcours de G est une liste des sommets de G telle que :

- chaque sommet de S apparaît une fois et une seule dans la liste,
- chaque sommet de la liste (sauf le premier) appartient à la bordure du sous-ensemble des sommets placés avant lui dans la liste, si toutefois cette bordure est non vide.

Les propriétés des parcours des graphes non orientés se prolongent au cas orienté. La notion d'arête de liaison est remplacée par celle d'*arc de liaison*. Le graphe partiel des arcs de liaison possède en particulier la propriété suivante que nous énonçons sans démonstration :

Proposition 4.5. *Les arcs de liaison d'un parcours constituent une forêt couvrante du graphe G.* ∎

La liste L=$(4, 5, 9, 8, 7, 6, 10, 11, 1, 3, 2)$ est un parcours du graphe orienté de la figure 4.7. La forêt couvrante contient deux arborescences A_1 et A_2 (en trait plein).

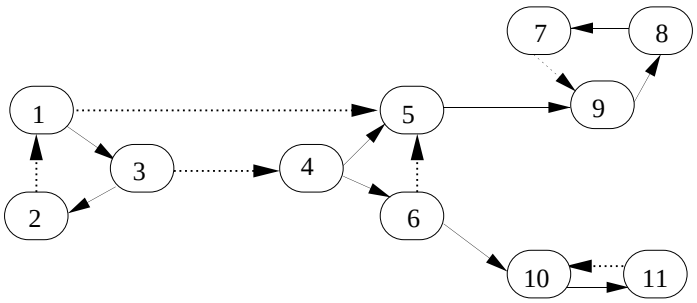


Figure 4.7: *Un graphe orienté.*

Parcours en profondeur d'un graphe orienté

Un parcours de G est dit *en profondeur* si l'origine de chaque arc de liaison est le dernier sommet ouvert déjà visité.

La procédure PARCOURS-PROFONDEUR-GRAPHE-ORIENTÉ ci-dessous calcule un parcours en profondeur en utilisant essentiellement la procédure PROFONDEUR-RÉCURSIF-ORIENTÉ, qui est l'adaptation directe de la procédure PROFONDEUR-RÉCURSIF présentée dans la section 4.4.2 pour le cas non orienté.

```
procédure PARCOURS-PROFONDEUR-GRAPHE-ORIENTÉ(G);
   tant qu'il existe un sommet de G non visité faire
      choisir un sommet non visité s;
      PROFONDEUR-RÉCURSIF-ORIENTÉ(s)
   fintantque.
```

La procédure PROFONDEUR-RÉCURSIF-ORIENTÉ s'écrit alors :

```
procédure PROFONDEUR-RÉCURSIF-ORIENTÉ(s)
  VISITER(s);
  pour tout successeur x de s faire
    si x est non visité alors
      VISITER(x); PROFONDEUR-RÉCURSIF-ORIENTÉ(x)
    finsi
  finpour.
```

Nous présentons maintenant quelques propriétés des parcours en profondeur qui seront particulièrement utiles au calcul des composantes fortement connexes.

Soit L un parcours en profondeur de G. Nous notons $\mathcal{F}$ la forêt couvrante induite par L et $r(x)$ le rang d'un sommet x dans la liste L. En dehors des arcs de $\mathcal{F}$, le parcours L permet de distinguer trois autres classes d'arcs dans G. Un arc (x, y) est *arrière* si y est un ascendant de x dans $\mathcal{F}$. Un arc (x, y) est *avant* si x est un ascendant de y dans $\mathcal{F}$. Un arc (x, y) est *transverse* si ses deux extrémités appartiennent à deux arborescences différentes, ou si x et y ont un ancêtre commun z dans $\mathcal{F}$ distinct de x et de y. Pour le parcours $(4, 5, 9, 8, 7, 6, 10, 11, 1, 3, 2)$ du graphe de la figure 4.7, les arcs $(7, 9)$, $(11, 10)$ et $(2, 1)$ sont arrière, les arcs $(6, 5)$, $(1, 5)$ et $(3, 4)$ sont transverses et il n'y a pas d'arc avant.

Le lemme suivant établit une propriété des arcs transverses.

Lemme 4.6. *Si (x, y) est un arc transverse pour un parcours en profondeur L, alors $r(y) < r(x)$.*

Preuve. Soit $L = L_1 \cdot L_2 \cdots L_q$ un parcours en profondeur et soient $\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_q$ les arborescences de la forêt couvrante. Par définition de L, il n'existe pas d'arc dont l'origine soit dans $\mathcal{A}_i$ et l'extrémité dans $\mathcal{A}_j$ avec $i < j$.

Soit (x, y) un arc transverse tel que $x \in \mathcal{A}_i$ et $y \in \mathcal{A}_j$ avec $i \neq j$. On a nécessairement $i > j$ et donc $r(x) > r(y)$.

Soit (x, y) un arc transverse dont les deux extrémités x et y appartiennent à la même arborescence et soit $z \notin \{x, y\}$ l'ancêtre commun de x et y dans cette arborescence. Si $r(x) < r(y)$, lorsque le sommet x est visité, le sommet y ne l'est pas encore. Il existe donc à cette étape du parcours un chemin de sommets non visités de x à y. Donc y sera accessible *dans l'arborescence* à partir de x. D'où la contradiction. ∎

Le classement des arcs de G relativement à un parcours en profondeur L fixé est à la base d'un algorithme performant pour tester si un graphe est sans circuit. En effet, le lemme suivant montre que G possède un circuit si et seulement s'il existe au moins un arc arrière.

Lemme 4.7. *Soient G un graphe orienté et L l'un de ses parcours en profondeur. Le graphe G est sans circuit si et seulement s'il n'existe pas d'arc arrière.*

Preuve. La condition nécessaire est évidente. Soit γ un circuit de G et supposons qu'il n'existe pas d'arc arrière. D'après le lemme 4.6, le circuit γ possède au moins un arc transverse et un arc avant. Considérons alors le sommet z, origine d'un arc avant de γ, dont le rang est minimum et notons y le prédécesseur de z dans γ. Soit x un sommet de γ distinct de z. Si l'arc de γ d'origine x est transverse, alors, d'après le lemme 4.6 son rang $r(x)$ est plus grand que le rang du premier sommet, origine d'un arc avant, rencontré à partir de x sur le circuit; on a donc $r(x) > r(z)$. Si l'arc de γ d'origine x est avant, on a $r(x) > r(z)$ par définition de z. Donc, lorsque le sommet z est visité, aucun autre sommet de γ n'est encore visité. Il en résulte l'existence dans $\mathcal{F}$ d'un chemin de z à y. L'arc (y, z) est donc arrière et transverse (par définition de z). Contradiction. ∎

Pour tester si un graphe orienté est sans circuit, il suffit donc de construire un parcours en profondeur de ce graphe et de tester si les arcs de G qui n'appartiennent pas à la forêt couvrante sont «avant» ou «transverse». La complexité de cet algorithme est en fait celle du parcours, soit $O(n + m)$.

Si le graphe G est fortement connexe, la forêt couvrante du parcours se réduit à une seule arborescence couvrante. Dans le cas contraire, nous montrons que le sous-graphe de $\mathcal{F}$ induit par chaque composante fortement connexe est une arborescente couvrante de cette composante fortement connexe.

Lemme 4.8. *Soient L un parcours en profondeur de G, $\mathcal{F}$ la forêt induite par L et C une composante fortement connexe de G. Le sous-graphe de $\mathcal{F}$ induit par C est une arborescence couvrant C.*

Preuve. Soit s le premier sommet de C visité dans L. Tous les autres sommets de C sont accessibles à partir de s dans le sous-graphe de G induit par C et n'ont pas encore été visités. Ils seront donc visités après s et accessibles dans $\mathcal{F}$ à partir de s. La proposition en résulte. ∎

La proposition précédente met en évidence le premier sommet du parcours L appartenant à une composante fortement connexe donnée C. Nous appellerons ce sommet *point d'entrée* de L dans C. Nous donnons maintenant une caractérisation de ces points d'entrée qui est à la base d'un algorithme efficace pour le calcul des composantes fortement connexes d'un graphe. Nous fixons maintenant un parcours en profondeur L.

Soit x un sommet, le *rang d'attache* de x par rapport à $\mathcal{F}$ est défini par

$$\rho(x) = \min\{r(z) \mid z \in AT(x) \cup \{x\}\},$$

où $AT(x)$ est l'ensemble des sommets extrémités d'un arc arrière ou transverse dont l'origine est un descendant de x dans $\mathcal{F}$ et dont l'extrémité appartient à une composante fortement connexe dont le point d'entrée est un ascendant propre de x dans $\mathcal{F}$. Le sommet de rang $\rho(x)$ est appelé *point d'attache* du sommet x dans $\mathcal{F}$ et est noté $a(x)$.

Soit $B \subset A$ un sous-ensemble des arcs de G. Il sera commode d'écrire respectivement $x \xrightarrow{B} y$, $x \xrightarrow{(*,B)} y$, et $x \xrightarrow{(+,B)} y$ s'il existe de x à y respectivement un arc de B, un chemin dans B et un chemin non nul dans B. Si l'on note R l'ensemble des arcs arrière, T l'ensemble des arcs transverses, $X = R \cup T$ et F l'ensemble des arcs de $\mathcal{F}$, alors un sommet z appartient à $AT(x)$ si et seulement s'il existe deux sommets y et t tels que

$$x \xrightarrow{(*,F)} y \xrightarrow{X} z \xrightarrow{(*,A)} t \xrightarrow{(+,F)} x. \tag{4.1}$$

Il résulte directement de cette définition que les sommets de $AT(x)$ sont dans la même composante fortement connexe que x.

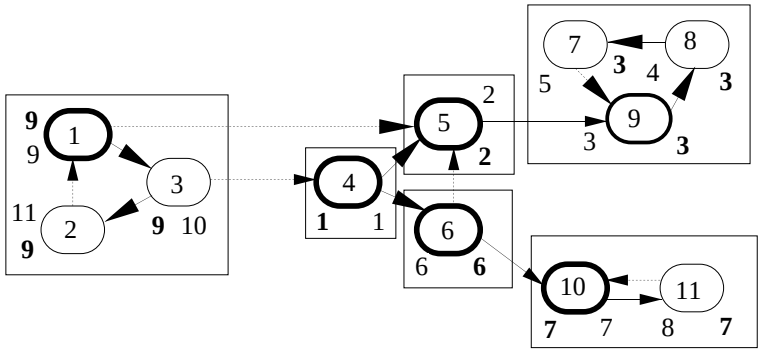


Figure 4.8: *Rangs d'attache et points d'attache.*

La figure 4.8 illustre les définitions du rang d'attache et du point d'attache, L=$(4, 5, 9, 8, 7, 6, 10, 11, 1, 3, 2)$ est un parcours en profondeur d'un graphe G possédant six composantes fortement connexes (encadrées). Les points d'entrée de ces composantes sont cerclés de noir. Le rang d'un sommet est à côté de ce sommet. Le rang d'attache d'un sommet est également inscrit en gras à côté de ce sommet.

Le lemme suivant fournit une caractérisation des points d'entrée.

Lemme 4.9. *Un sommet x est un point d'entrée si et seulement si $x = a(x)$.*

Preuve. Si x est un point d'entrée de la composante fortement connexe C, alors $AT(x) = \emptyset$. En effet, si $z \in AT(x)$ il existe y et t qui vérifient 4.1. On a donc $z \in C$ puisque $z \in AT(x)$ et $t \notin C$ puisqu'aucun ascendant propre de x dans $\mathcal{F}$ n'appartient à C. D'où la contradiction. Il en résulte que $\rho(x) = r(x)$ et que $a(x) = x$.

Réciproquement, si x appartient à la composante fortement connexe C et n'est pas son point d'entrée, nous notons e le point d'entrée de C. On a alors $e \xrightarrow{(+,F)} x$ et $x \xrightarrow{(+,A)} e$. Or comme il n'existe pas de chemin de x à e dans $\mathcal{F}$, il existe un sommet z qui n'est pas un descendant de x dans $\mathcal{F}$ tel que :

$$x \xrightarrow{(*,F)} y \xrightarrow{X} z \xrightarrow{(*,A)} e. \tag{4.2}$$

Le sommet z appartient à $AT(x)$ et nous montrons que $r(z) < r(x)$. Soit t l'ancêtre de z et x dans $\mathcal{F}$. Le sommet t est distinct de x puisque z n'est pas un descendant de x dans $\mathcal{F}$. Si $t = z$ alors z est un ascendant propre de x et $r(z) < r(x)$. Si t est distinct de z et $r(z) > r(x)$, lors de la visite de x seul le sommet x du chemin de x à z a été visité et donc z sera un descendant de x dans $\mathcal{F}$, ce qui contredit l'hypothèse. On a donc $r(z) < r(x)$ et donc $a(x) \neq x$. ∎

4.4.5 Calcul des composantes fortement connexes

Nous présentons dans cette section un algorithme efficace, dû à Tarjan, pour le calcul des composantes fortement connexes d'un graphe orienté quelconque G. Cet algorithme réalise simultanément un parcours en profondeur L du graphe G et le calcul des rangs d'attache de chaque sommet. L'implémentation de cet algorithme par la procédure CFC ci-dessous est une variante de la procédure PROFONDEUR-RÉCURSIF présentée dans la section 4.4.2. Cette variante utilise une pile Π qui empile les sommets au fur et à mesure de leur insertion dans le parcours en profondeur et dépile tous les sommets de chaque nouvelle composante fortement connexe détectée lorsque le parcours en profondeur revient sur un point d'entrée. La procédure CFC se distingue de la procédure PROFONDEUR-RÉCURSIF par les trois points suivants :

a) La notion de voisin est remplacée par celle de successeur ;

b) Lors de l'examen des successeurs du sommet x en cours dans le parcours en profondeur, deux cas sont possibles :

1. Si le successeur y examiné n'est pas déjà visité, alors le rang $r(y)$ est calculé, le rang d'attache $\theta(y)$ est initialisé à $r(y)$, le sommet y est empilé dans Π, un appel récursif est exécuté pour le sommet y et le rang d'attache $\theta(x)$ est actualisé à $\min\{\theta(x), \theta(y)\}$.
2. Si le successeur y examiné est déjà visité, le rang d'attache $\theta(x)$ est actualisé à $\min\{\theta(x), r(y)\}$.

c) Lorsque tous les successeurs d'un sommet x ont été traités, la valeur de $\theta(x)$ est le rang d'attache de x. Si $\theta(x) = r(x)$, le sommet x est un point d'entrée d'une composante fortement connexe C dont les sommets autres que x sont situés au-dessus de x dans Π. Cette pile est alors dépilée jusqu'au sommet x.

La procédure DESC(s) ci-dessous détermine dans le tableau c les composantes fortement connexes qui sont des descendants de la composante fortement connexe contenant s dans le graphe réduit de G. L'élément $c(x)$ est le numéro de la composante fortement connexe contenant x. On suppose également qu'un tableau de booléens est utilisé pour tester l'appartenance d'un sommet à la pile Π.

```
procédure DESC(s);
(1)  EMPILER(s, Π); V := V ∪ {s};
(2)  r := r + 1; r(s) := r; θ(s) := r;
(3)  pour tout successeur x de s dans G faire
(4)     si x ∉ V alors DESC(x); θ(s) := min{θ(s), θ(x)}
(5)     sinon
(6)        si x est dans Π alors θ(s) := min{θ(s), r(x)} finsi
(7)     finsi
(8)  finpour;
(9)  si θ(s) = r(s) alors
(10)    k := k + 1;
(11)    répéter z :=DÉPILER(Π); c(z) := k
(12)    jusqu'à z = s;
(13) finsi.
```

L'algorithme de calcul des composantes fortement connexes de G est implémenté par la procédure CFC(G) ci-dessous.

```
procédure CFC(G)
   V := ∅; {V est l'ensemble des sommets visités}
   r := 0; {r est le compteur pour les rangs}
   k := 0; {k numérote les composantes fortement connexes}
   PILEVIDE(Π);
   tantque V ≠ S faire
      choisir s dans S − V; DESC(s)
   fintantque.
```

Les deux propositions suivantes établissent la complexité et la validité de la procédure CFC.

Proposition 4.10. *Soit G un graphe possédant n sommets et m arcs. La complexité de la procédure* CFC *est $O(\max\{m, n\})$.*

Preuve. Pour évaluer la complexité de la procédure CFC, examinons les opérations *supplémentaires* réalisées par rapport à un simple parcours en profondeur du graphe G. Lors de chaque étape du parcours en profondeur, seules des opérations de mise à jour de complexité $O(1)$ sont introduites. En ce qui concerne la pile Π, chaque sommet est empilé et dépilé exactement une fois. Il en résulte que la complexité globale de la procédure CFC est $O(\max\{m, n\})$. ∎

Proposition 4.11. *Soit s un sommet du graphe G. La procédure* DESC(s) *détermine les composantes fortement connexes de $G^+(s)$.*

Preuve. Au cours d'un parcours en profondeur d'un graphe G, un sommet passe successivement par trois états. Tant qu'il n'a pas été visité, il est *libre*. Lorsqu'il est visité (c'est-à-dire lorsqu'il est l'extrémité d'un mouvement avant du parcours), il devient *examiné*. Lorsqu'il devient l'extrémité d'un mouvement arrière du parcours, il est *exclu*. L'ordre dans lequel les sommets sont examinés est celui du parcours en profondeur. La liste L^- associée à l'ordre d'exclusion des sommets vérifie les propriétés suivantes :

- le dernier sommet exclu d'une composante fortement connexe est son point d'entrée,
- les points d'entrée des composantes fortement connexes apparaissent dans l'ordre d'un parcours *postfixe* du graphe réduit de G.

La liste d'exclusion pour l'exemple de la figure 4.9 est $(7, 8, 9, 5, 11, 10, 6, 4)$.

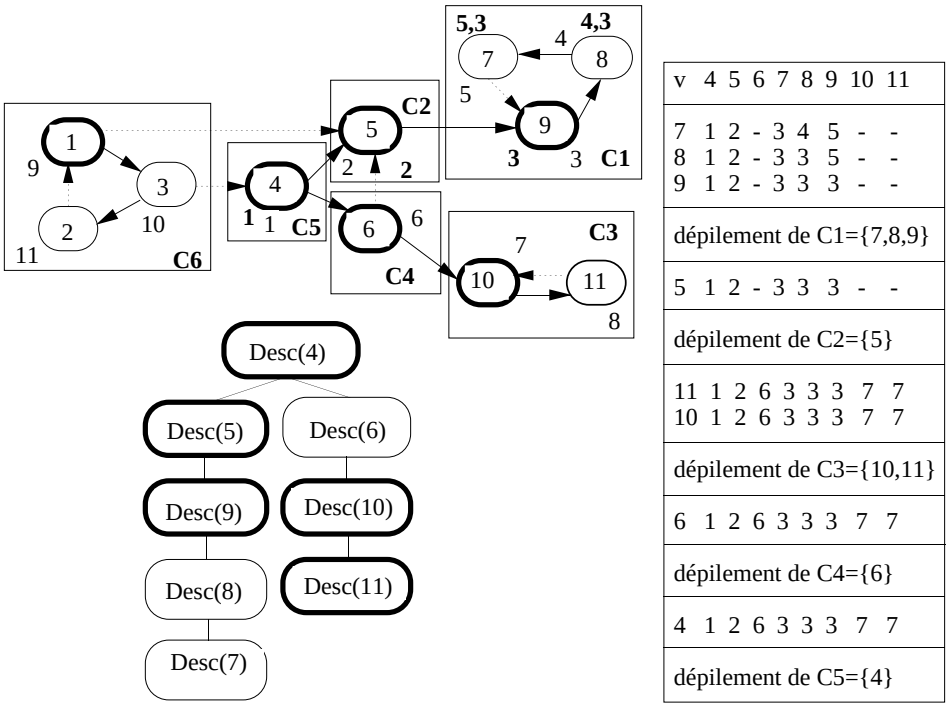


Figure 4.9: *Une exécution de* DESC.

Supposons dans un premier temps que les valeurs $\theta(x)$ calculées par la procédure DESC(s) pour tous les sommets de $G^+(s)$ soient effectivement les rangs d'attache de ces sommets. La liste d'exclusion L^- du parcours en profondeur de $G^+(s)$ correspond à l'ordre dans lequel les appels récursifs de la procédure DESC pour les sommets de $G^+(s)$ se terminent. Il résulte alors du lemme 4.9 que lorsque le nouveau sommet exclu x satisfait $\theta(x) = r(x)$, ce sommet est un point d'entrée. Donc les «macro-dépilements» de la pile sont réalisés par DESC lors des exclusions successives des points d'entrée. Or, lors de l'exclusion d'un point d'entrée x, la pile contient dans sa partie supérieure à partir du sommet x les sommets de la composante fortement connexe C de x. En effet, en raison de l'ordre postfixe des

macro-dépilements, tous les descendants propres de C dans le graphe réduit de G ont déjà été «macro-dépilés».

Nous montrons maintenant que, pour tout sommet x de $G^+(s)$, lors de la terminaison de l'appel DESC(x), la valeur calculée $\theta(x)$ est égale au rang d'attache du sommet x. Nous raisonnons par récurrence sur le nombre d'appels terminés de la procédure DESC.

Considérons la terminaison de l'appel DESC(v). Nous montrons d'abord que si le sommet w est un fils de v dans $\mathcal{F}$ et si $r(v) > \rho(w)$ alors on a nécessairement $\rho(v) \leq \rho(w)$. En effet, soit (y, x) un arc arrière ou transverse tel que $w \xrightarrow{(*,F)} y$ et $x \xrightarrow{(*,A)} e$ où x est le point d'attache de w et e est le point d'entrée de la composante fortement connexe contenant x. On a $\rho(w) = r(x)$ et donc $r(x) < r(v)$. Comme $x \in AT(v)$, on a $\rho(v) \leq \rho(w)$. Nous montrons maintenant que si l'arc (v, w) est un arc arrière ou transverse dont l'extrémité w appartient à une composante fortement connexe C non encore «macro-dépilée» et si $r(v) > r(w)$ alors on a nécessairement $\rho(v) < r(w)$. En effet, soit e le point d'entrée de C. L'appel DESC(e) n'est pas terminé lors de la terminaison de DESC(v). Le sommet e qui est un ascendant de w est aussi un ascendant de v. En effet, dans le cas contraire, $r(e) \leq r(w) < r(v)$ impliquerait que l'appel DESC(e) soit terminé avant DESC(v). Il en résulte que $w \in AT(v)$ et $\rho(v) < r(w)$.

En utilisant l'hypothèse de récurrence pour les successeurs de v qui sont des fils de v dans $\mathcal{F}$, la valeur $\theta(v)$ calculée par l'appel DESC(v) est égale à $\min\{r(v), \alpha, \beta\}$ où, si S désigne les fils de v dans $\mathcal{F}$ et T les autres successeurs de v dans G :

$$\alpha = \min\{\rho(w) \mid w \in S\} \quad \beta = \min\{r(w) \mid w \in T\}.$$

D'après ce qui précède, cette valeur correspond à celle du rang d'un sommet de $AT(v)$. Montrons que lors de la terminaison de DESC(v), on a $\theta(v) = \rho(v)$. Nous considérons à cet effet un arc (x, y) arrière ou transverse dont l'origine est un descendant de v dans $\mathcal{F}$ et l'extrémité y appartient à $AT(v)$ et nous montrons que $\theta(v) \leq r(y)$. Nous notons e le point d'entrée de la composante fortement connexe contenant y et envisageons deux cas :

- Supposons $x = v$. D'après l'hypothèse de récurrence, les composantes fortement connexes ont été correctement calculées jusqu'à la terminaison des appels de DESC antérieurs à DESC(v). Le sommet y est donc dans la pile puisque DESC(e) n'est pas terminé. Or lors de l'examen du successeur y de v dans l'appel DESC(v), la valeur $\theta(v)$ est actualisée (ligne (6)) à une valeur inférieure ou égale à $r(y)$.

- Supposons $x \neq v$. Le sommet x est alors un descendant propre de v dans $\mathcal{F}$. Si z est le fils de v dans $\mathcal{F}$ dont x est le descendant, après la terminaison de l'appel DESC(z), la valeur $\theta(v)$ est actualisée (ligne (4)) et rendue inférieure ou égale à $\rho(z)$ donc à $r(y)$.

Il en résulte que $\theta(v) = \rho(v)$. ∎

La figure 4.9 montre en a) l'arbre des appels récursifs où les sommets grisés correspondent aux points d'entrée (donc aux macro-dépilements), en b) les valeurs des rangs d'attache en cours après la terminaison de DESC(v).

4.4.6 Parcours d'une arborescence

La structure récursive des arborescences permet de définir des parcours spécifiques fondés sur d'autres règles que la propriété d'adjacence commune aux parcours étudiés jusqu'ici. Deux parcours sont très utilisés : le parcours *préfixe* pour lequel les ascendants propres d'un sommet sont placés avant ce sommet dans la liste et le parcours *postfixe* pour lequel les descendants propres d'un sommet sont placés avant ce sommet dans la liste.

Soit A une arborescence de racine r. La structure récursive de A permet d'établir simplement les algorithmes des parcours préfixe et postfixe.

```
fonction LPREF(A) :liste;
  si A = Λ alors LPREF :=()
  sinon
    L := (r);
    pour chaque fils u de r faire
      L :=L·LPREF(A(u));
    LPREF :=L
  finsi.
```

```
fonction LPOST(A) :liste;
  si A = Λ alors LPOST :=()
  sinon
    L := ();
    pour chaque fils u de r faire
      L :=L·LPOST(A(u));
    LPOST :=L · (r)
  finsi.
```

Pour l'arborescence de la figure 4.10, $(6, 7, 1, 3, 5, 4, 2)$ est le parcours préfixe et $(1, 7, 5, 3, 4, 2, 6)$ est le parcours postfixe.

Proposition 4.12. *La liste* LPREF(A) *contient chaque sommet de A une fois et une seule et tous les ascendants propres d'un sommet sont placés avant ce sommet dans la liste.*

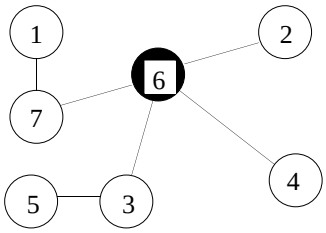


Figure 4.10: *Une arborescence.*

Preuve. (Induction sur le nombre n de sommets de A.) Si $n \leq 1$, la propriété est vraie. Soit une arborescence $A = (H, r)$ à n sommets ($n \geq 2$). Par définition, LPREF(A) est la concaténation des listes (r), LPREF($A(r_1)$), ..., LPREF($A(r_k)$). Par induction, pour tout $j \in \{1, \ldots, k\}$, LPREF($A(r_j)$) vérifie la proposition pour la sous-arborescence $A(r_j)$. La liste LPREF(A) contient donc chaque sommet de A une et une seule fois. D'autre part, soit u un sommet de A et v un ascendant propre de u. Si $v = r$ alors il est placé avant u. Sinon, par induction, v est placé avant u dans la sous-arborescence $A(r_j)$ qui contient u et la proposition est donc vérifiée. ∎

Parcours symétrique d'un arbre binaire

Les arbres binaires sont très utilisés comme structures de données permettant de manipuler efficacement les ensembles ordonnés. Le *parcours symétrique* d'un arbre binaire est une liste des sommets de l'arbre telle que tout sommet est placé dans la liste après les sommets de son sous-arbre gauche et avant les sommets de son sous-arbre droit. Une telle liste est unique puisque la définition impose la place de la racine, puis successivement la place des racines de tous les sous-arbres.

L'algorithme récursif suivant détermine le parcours symétrique noté SYM(A) d'un arbre binaire A de racine r :

```
fonction SYM(A :arbre binaire) :liste;
   si A = Λ alors SYM:= ()
   sinon
      SYM :=SYM(A_g) · (r)· SYM(A_d)
   finsi.
```

Le parcours symétrique de l'arborescence de la figure 4.11 est $(12, 2, 5, 4, 6, 3, 1, 7, 10, 11, 8, 9)$.

Proposition 4.13. *La liste* SYM(A) *est le parcours symétrique de l'arbre binaire A.*

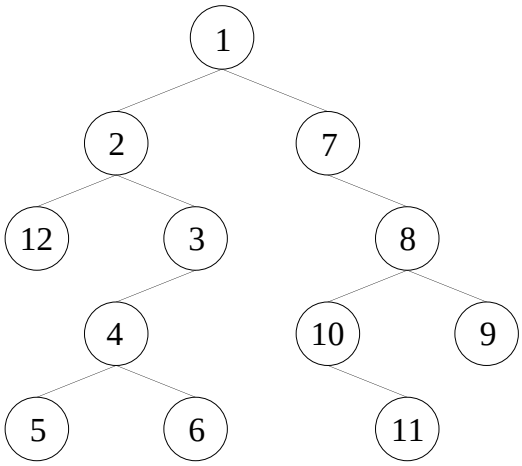


Figure 4.11: *Un arbre binaire.*

Preuve. (Induction sur le nombre n de sommets de A.) La propriété est vraie pour $n = 0$ puisque $\textsc{Sym}(\Lambda)$ est vide. Soit A un arbre binaire à $n \geq 1$ sommets. Par définition, $\textsc{Sym}(A)$ est la concaténation des listes $\textsc{Sym}(A_g)$, (r) et $\textsc{Sym}(A_d)$. Par induction la propriété est vraie pour tout sommet de A_g et de A_d et elle est vraie par construction pour la racine r de A, donc elle est vraie pour tout sommet de A. ∎

Notes

Les ouvrages sur les graphes sont nombreux. Nous avons utilisé les définitions du livre de référence :

C. Berge, *Graphes*, Gauthier-Villars (troisième édition), 1983.

D'autres livres utiles sont :

F. Harrary, *Graph Theory*, Addison-Wesley, Reading, Mass, 1969 ;
M. Gondran et M. Minoux, *Graphes et Algorithmes*, Eyrolles, 1979.

L'algorithme de calcul des composantes fortement connexes est de :

R. Tarjan, Depth First Search and Linear Graph Algorithms, *SIAM J. Computing* **1** (1972), 146–160.

Exercices

4.1. Démontrer que la matrice d'incidence sommets-arcs d'un graphe orienté sans boucles est totalement unimodulaire, c'est-à-dire que le déterminant de toute sous-matrice carrée extraite vaut 0, +1 ou −1.

4.2. Soit $G = (S, A)$ un graphe orienté. On considère la famille $\mathcal{G}$ des graphes partiels de G qui ont même fermeture transitive que G. Un graphe H de $\mathcal{G}$ est *minimal* si le graphe obtenu à partir de H en supprimant un arc quelconque n'est plus dans $\mathcal{G}$. Montrer qu'en général il existe dans $\mathcal{G}$ plusieurs graphes minimaux. Démontrer que si le graphe G est sans circuit, alors $\mathcal{G}$ ne contient qu'un seul graphe minimal (qui est appelé graphe de Hasse de G).

4.3. Ecrire un algorithme de complexité $O(\max\{m, n\})$ pour déterminer si un graphe à n sommets et m arcs est sans circuit.

4.4. Soient $G = (S, A)$ un graphe orienté et $c : A \mapsto \mathbb{N}$. Soit p un entier fixé. Pour tout couple de sommets i, j, on cherche les longueurs des p plus courts chemins (relativement à c) de i à j. On utilise pour cela le semi-anneau $(\mathbb{N}^p, \min, +, 0^p, 1^p)$ des vecteurs à p coordonnées dans $\mathbb{N}$ où les opérations $+$ et min sont définies pour deux vecteurs $a = (a_1, \ldots, a_p)$ et $b = (b_1, \ldots, b_p)$ par $\min\{a, b\} = (c_1, \ldots, c_p)$, où les c_i, $i \in \{1, \ldots, p\}$ sont les p plus petits éléments de $a_1, a_2, \ldots, a_p, b_1, b_2 \ldots, b_p$ et $a+b = c$, où les c_i, $i \in \{1, \ldots, p\}$ sont les p plus petits éléments parmi les p^2 sommes $a_i + b_j$, $i \in \{1, \ldots, p\}$, $j \in \{1, \ldots, p\}$. Montrer que le vecteur c_{ij}^n défini par la formule de récurrence 2.1 fournit les valeurs des p plus courts chemins de i à j dans G.

4.5. Soit K_n l'ensemble des matrices carrées d'ordre n à coefficients dans un semi-anneau complet K. Montrer comment munir K_n d'une structure de semi-anneau complet. On définit pour $M \in K_n$ la matrice M^* par :

$$M^* = I \oplus M \oplus M^2 \ldots \oplus M^h \oplus \ldots .$$

Soit $G = (S, A)$ un graphe à n sommets, soit $c : A \mapsto K$ une fonction coût et soit $M = m_{ij}$ la matrice définie par :

$$m_{ij} = \begin{cases} c((i,j)) & \text{si } (i,j) \in A \\ 0 & \text{sinon.} \end{cases}$$

Démontrer que :

$$m_{ij}^* = c_{ij} \quad \text{où} \quad c_{ij} = \bigoplus_{\gamma \in \Gamma_{ij}} c(\gamma).$$

4.6. Montrer que pour $\mathcal{R} = \mathbb{R} \cup \{+\infty, -\infty\}$, $(\mathcal{R}, \min, \max, +\infty, -\infty)$ est un semi-anneau complet. Soit $G = (S, A)$ un graphe non orienté, $c : A \mapsto \mathbb{R}$ une fonction coût et soit

$$c_{ij} = \min_{\gamma \in \Gamma_{ij}} c(\gamma)$$

où Γ_{ij} est l'ensemble des chaînes élémentaires de i à j. Soit

$$U = \{(i, j) \in A \mid c_{ij} = c((i, j))\}.$$

Montrer que le sous-graphe (S, U) est un arbre couvrant de coût minimal (au sens du chapitre 7).

4.7. Démontrer la proposition 2.2.

4.8. Soit $G = (S, A)$ un graphe orienté connexe à n sommets et m arcs. On appelle *cycle élémentaire* de G une suite d'arcs de G telle que deux arcs consécutifs quelconques de la suite (y compris le dernier et le premier) aient une extrémité commune et que ces extrémités communes successives soient toutes distinctes. On associe à un cycle élémentaire de G le vecteur de $\{0, 1, -1\}^m$ pour lequel la coordonnée associée à l'arc u vaut 0 si le cycle n'emprunte pas l'arc u, 1 si le cycle emprunte l'arc u dans le sens de u et -1 si le cycle emprunte l'arc u dans le sens opposé à u. Démontrer que le sous-espace vectoriel de $\mathbb{R}^m$ engendré par les cycles élémentaires de G est de dimension $m - n + 1$. En déduire que si G possède p composantes connexes, le nombre maximum de cycles « indépendants » de G est $m - n + p$.

4.9. Montrer que l'on peut utiliser un parcours en largeur d'un graphe non orienté à partir d'un sommet s pour déterminer les longueurs des plus courtes chaînes de s à tous les autres sommets du graphe.

4.10. Soit G un graphe non orienté connexe et soit s l'un de ses sommets. Montrer que l'arborescence des appels récursifs de la procédure PROFONDEUR-RÉCURSIF(s) s'identifie avec une arborescence couvrante de G de racine s. Montrer que si $\{x, y\}$ est une arête n'appartenant pas à l'arborescence, l'un des deux sommets x ou y est ascendant de l'autre dans l'arborescence. Montrer que la complexité en temps de la procédure est $O(\max\{n, m\})$, où n est le nombre de sommets et m le nombre d'arêtes du graphe G.

Chapitre 5

Tris

Ce chapitre traite des algorithmes de tri. La première section présente trois algorithmes de tri interne pour lesquels aucune hypothèse particulière n'est faite sur les clés : le tri rapide, le tri fusion et le tri par tas. La seconde section concerne le tri externe. On y présente un algorithme de construction des monotonies et deux algorithmes de répartition : le tri équilibré et le tri polyphasé.

Introduction

Dans ce chapitre, nous présentons trois algorithmes de tri parmi les plus efficaces lorsqu'aucune hypothèse particulière n'est faite sur la nature ou la structure des clés des éléments à trier. Le *tri rapide* place définitivement un élément appelé pivot, construit une liste «gauche» des éléments dont la clé est inférieure ou égale à celle du pivot, construit une liste «droite» des éléments dont la clé est strictement supérieure à celle du pivot et réalise enfin un appel récursif sur chacune des deux listes. Le *tri fusion* est également un tri dichotomique. Il réalise d'abord un appel récursif sur chacune des deux listes obtenues par scission de la liste initiale en deux listes de taille égale (à une unité près) et interclasse ensuite les deux listes triées. Le *tri par tas* place d'abord les éléments à trier dans un tableau organisé en file de priorité (tas), puis supprime un à un les éléments de priorité minimum du tas tout en les replaçant simultanément dans le tableau dans l'ordre inverse. Ces trois algorithmes de tri ont une complexité moyenne $O(n \log n)$. Dans le pire des cas, le tri rapide a pour complexité $O(n^2)$, alors que la complexité des deux autres tris est $O(n \log n)$. D'autres algorithmes moins performants comme par exemple les tris simples (tri bulle, tri par insertion, tri par sélection,...) ne sont pas présentés ici mais ont, pour la plupart d'entre eux, servi d'illustration à d'autres sujets traités dans le livre. Certains algorithmes, par exemple le *tri par champs*, sont plus performants mais exigent certaines hypothèses spécifiques sur les clés des éléments à trier. Nous proposons en exercice l'étude de certains algorithmes de ce type.

5.1 Tri interne

La donnée d'un problème de tri est constituée d'une permutation $L = (e_1, \ldots, e_n)$, d'un ensemble E de n éléments et d'une application $c : E \mapsto F$ de E dans un ensemble F totalement ordonné qui associe à chaque élément e une *clé* $c(e)$. Nous appellerons élément maximal de E un élément dont la clé est le plus grand élément de $c(E)$. Inversement, un élément minimal de E est un élément dont la clé est le plus petit élément de $c(E)$. Une *liste triée* de E est une permutation de E dont les éléments sont rangés dans un ordre compatible avec la relation d'ordre sur F. Trier la liste L, c'est déterminer une liste triée des éléments de L.

Exemple. Si $E = \{A, B, C, D\}$, $F = \{0, 1\}$, $c(A) = c(B) = 1$ et $c(C) = c(D) = 0$, la liste (C, D, B, A) est une liste triée.

Nous mesurerons la complexité d'un tri par le nombre de comparaisons de clés qu'il réalise et la taille d'un problème de tri sera définie par le nombre n d'éléments à trier. Nous supposerons également que la liste initiale L est implémentée par un tableau linéaire $T[i..j]$ où $n = j - i + 1$. Ces hypothèses sont réalistes dans le cas d'un *tri interne* où les éléments à trier sont et restent disponibles en mémoire centrale. La section 5.2 traitera le cas du tri externe pour lequel cette hypothèse n'est pas satisfaite.

5.1.1 Le tri rapide

Le tri rapide d'une liste L sur E est un algorithme comprenant deux phases. La première phase transforme la liste L en une liste $L' = G \cdot (\pi) \cdot D$ sur E où π est un élément de L appelé *pivot*, G est la *liste gauche* contenant g éléments ($g \leq n-1$), D est la *liste droite* contenant d éléments ($d \leq n-1$). De plus, la clé d'un élément quelconque de G est inférieure ou égale à la clé d'un élément quelconque de D.

Notons que chacune des listes G ou D peut être vide et qu'elles le sont toutes les deux si $n = 1$. Plus précisément, si le pivot est un élément maximal, alors la liste D est vide, si le pivot est un élément minimal, alors la liste G peut être vide ou non. Il est important de remarquer qu'après la première phase, le pivot est bien placé et que les tailles de la liste gauche et de la liste droite sont strictement plus petites que n.

La seconde phase consiste simplement en deux appels récursifs de l'algorithme, l'un pour la liste G, l'autre pour la liste D.

Un appel de l'algorithme de tri rapide est *terminal* si $n \leq 1$. Notons qu'un appel terminal n'effectue aucune comparaison de clés. Nous pouvons donc décrire l'algorithme de tri rapide par la procédure TRI RAPIDE ci-dessous. La clé de l'élément $T(k)$ est notée $c(k)$ et les listes associées aux appels récursifs correspondent à des sous-tableaux de $T[i..j]$. La première phase correspond à la fonction PIVOTER qui retourne l'indice du pivot.

```
procédure TRI RAPIDE(T[i..j]);
  si i < j alors
    k :=PIVOTER(T[i..j]);
    TRI RAPIDE(T[i..(k − 1)]);
    TRI RAPIDE(T[(k + 1)..j])
  finsi.
```

L'algorithme de pivotage

L'efficacité du tri rapide dépend directement de l'algorithme de pivotage. Ce dernier doit d'une part choisir le pivot et d'autre part réorganiser le tableau.

Nous conviendrons de choisir comme pivot le *premier élément du tableau*, c'est-à-dire l'élément $T(i)$ dont la clé $c(i)$ est notée conventionnellement γ. Ce choix, qui n'est pas le seul possible, n'est pas plus mauvais qu'un autre en l'absence d'hypothèse supplémentaire sur les données et présente le double avantage d'être simple à implémenter et de conduire à une analyse en moyenne rigoureuse.

Nous appelons *couple inversé* du tableau par rapport au pivot un couple d'indices (s, t) tel que :

$$i < s < t \leq j, \quad c(s) > \gamma, \quad c(t) \leq \gamma.$$

Le couple inversé (s, t) *précède* le couple inversé (s', t') si $s' \geq s$ et $t' \leq t$. Cette relation de précédence confère à l'ensemble des couples inversés une relation d'ordre total. Si cet ensemble n'est pas vide, il existe donc un premier couple inversé. La réorganisation du tableau $T[i..j]$ est un algorithme itératif où chaque itération concerne des sous-tableaux emboîtés de T et *toujours la même valeur* de γ.

La première itération concerne le tableau $T[i+1..j]$. Elle consiste à rechercher le premier couple inversé (s'il existe) par un appel à la fonction PREMIER-COUPLE-INVERSÉ qui retourne un couple (s, t). Si $s < t$, alors (s, t) est effectivement le premier couple inversé par rapport à γ, les éléments de rang s et t sont échangés dans le tableau T par la procédure ECHANGER et l'itération suivante concernera le sous-tableau $T[s'..t']$ où $s' = s + 1$ et $t' = t - 1$. L'algorithme se termine si $s' > t'$.

```
fonction PREMIER-COUPLE-INVERSÉ(T, k, l, γ) :couple;
  s := k; t := l;
  tantque s ≤ l etalors c(s) ≤ γ faire s := s + 1 fintantque;
  tantque t ≥ k etalors c(t) > γ faire t := t − 1 fintantque
  PREMIER-COUPLE-INVERSÉ :=(s, t).
```

La fonction PIVOTER ci-dessous implémente l'algorithme de pivotage.

```
fonction PIVOTER(T, i, j) :indice;
  γ :=c(i); s := i + 1; t := j;
  tantque s ≤ t faire
    (s, t) :=PREMIER-COUPLE-INVERSÉ(T, s, t, γ);
    si s < t alors
      ECHANGER(T, s, t); s := s + 1; t := t − 1
    finsi
  fintantque;
  PIVOTER :=t;
  ÉCHANGER(T, i, t).
```

La figure 1.1 montre une exécution de l'algorithme de pivotage. La première ligne contient la liste initiale.

6	(14)	3	1	10	5	1	6	4	5	2	11	9	[6]
6	6	3	1	(10)	5	1	6	4	5	[2]	11	9	14
6	6	3	1	2	5	1	6	4	[5]	(10)	11	9	14
5	6	3	1	2	5	1	6	4	6	10	11	9	14

◯ pointeur gauche
□ pointeur droit
pivot placé

Figure 1.1: *Un pivotage.*

Le lemme suivant établit un encadrement du nombre de comparaisons de clés réalisées lors d'un pivotage.

Lemme 1.1. *L'algorithme de pivotage réalise au moins $n-1$ et au plus $n+1$ comparaisons de clés.*

Preuve. Si tous les éléments de $T[i+1..j]$ ont une clé inférieure ou égale à γ, la valeur finale de s est $j+1$ et t n'a pas été modifié. Il en résulte $n-1$ comparaisons à partir de s et une comparaison à partir de t, soit en tout n comparaisons. Il y a de même n comparaisons si tous les éléments de $T[i+1..j]$ ont une clé strictement supérieure à γ. Si le tableau $T[i+1..j]$ contient un élément de clé inférieure ou égale à γ et un élément de clé strictement supérieure à γ, deux cas sont possibles. Si le *dernier* échange d'un couple inversé concerne deux éléments contigus dans T, il y a eu exactement $n-1$ comparaisons avant cet échange et il n'y en aura pas après. Sinon, lors de la terminaison du pivotage, les clés des éléments indicés par les dernières valeurs de s et t ont été comparées *deux fois* à γ, il y a donc eu $n+1$ comparaisons (voir figure 1.2). ∎

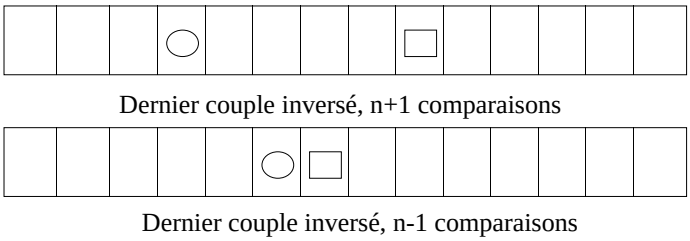


Figure 1.2: *Dernier échange d'un couple inversé.*

Complexité dans le pire des cas

Examinons l'arborescence $\mathcal{A}$ des appels récursifs d'une exécution de l'algorithme de tri rapide. L'arborescence associée à la liste initiale de la figure 1.1 est donnée sur la figure 1.3. Les indices i et j d'un appel sont inscrits à l'intérieur du sommet correspondant. Les sommets des appels terminaux sont cerclés de noir. Cette

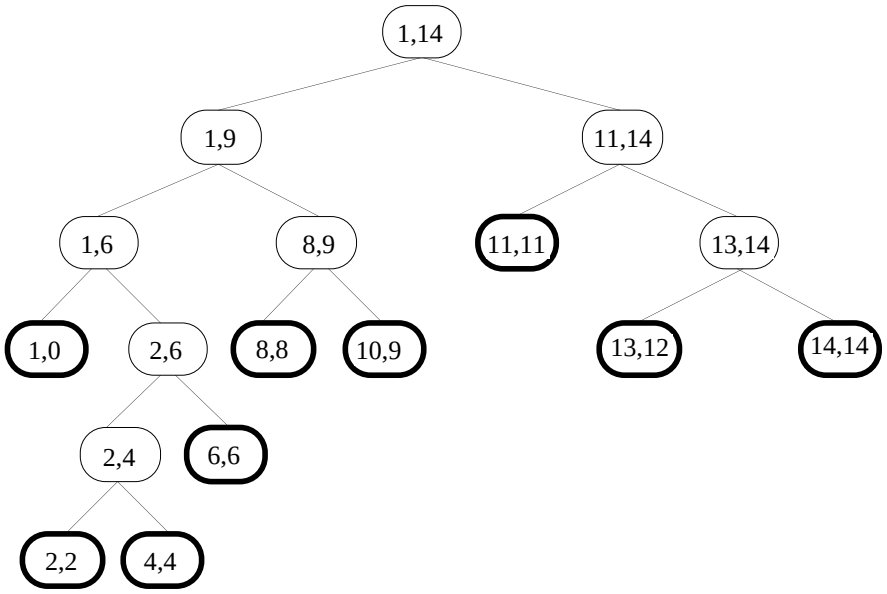


Figure 1.3: *Appels récursifs du tri rapide.*

arborescence est un arbre binaire complet dont les feuilles correspondent à un tableau contenant au plus un élément. Si les deux fils y et z d'un noeud x correspondent à deux tableaux de p et q éléments, le noeud x correspond lui à un tableau de $p+q+1$ éléments. D'après le lemme 1.1, le nombre maximum de comparaisons faites au noeud x , égal à $p+q+2$, est la somme du nombre maximum de comparaisons $(p+1)$ faites au noeud y et du nombre maximum de comparaisons $(q+1)$ faites au noeud z. Le nombre total de comparaisons faites pour tous les noeuds *d'un même niveau* de l'arbre est donc majoré par $n+1$ et le nombre total de comparaisons est lui-même majoré par $(n+1)h(\mathcal{A})$ où $h(\mathcal{A})$ est la hauteur en nombre de sommets de l'arborescence $\mathcal{A}$.

Si la liste initiale contient n éléments, la hauteur maximale est atteinte (voir exercices) lorsque chaque niveau (le niveau 0 de la racine excepté) est constitué de deux noeuds frères, un noeud terminal correspondant à une liste vide et un noeud correspondant à une liste de p éléments si son père est lui-même associé à une liste de $p+1$ éléments (voir figure 1.4). La complexité dans le pire des cas de l'algorithme de tri rapide est donc $O(n^2)$.

Cette complexité est effectivement atteinte lorsque la liste initiale L est une liste triée. En effet l'arborescence induite est alors le «peigne» représenté sur la figure 1.4. Dans ce cas, le nombre de comparaisons (voir lemme 1.1), inscrit à l'intérieur du sommet sur la figure, est *exactement* n pour le niveau 0, $n-1$ pour le niveau 1, $n-i$ pour le niveau i, 2 pour l'avant-dernier niveau et 0 pour le dernier niveau, soit en tout $\sum_{j=2}^{n} j{=}n(n+1)/2-1$ comparaisons.

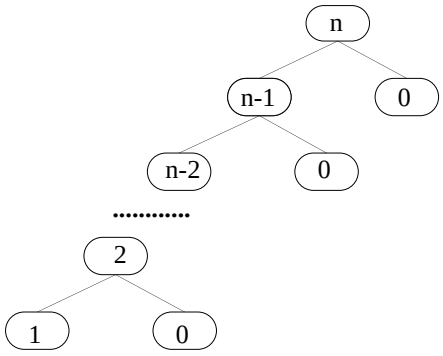


Figure 1.4: *Un «peigne».*

Complexité en moyenne du tri rapide

Nous faisons ici l'hypothèse que les clés des n éléments de la liste initiale L sont *distinctes* et que les $n!$ permutations qui représentent les rangs de ces clés dans L sont *équiprobables*. Nous noterons $\pi(L)$ la permutation des rangs des clés de la liste L.

Exemple. Si $L=(A,B,C,D,E,F)$, $c(A)=3$, $c(B)=1$, $c(C)=5$, $c(D)=10$, $c(E)=7$ et $c(F)=15$, on a $\pi(L)=(2,1,3,5,4,6)$. Le lemme suivant montre que l'algorithme de pivotage conserve les hypothèses probabilistes de la liste initiale pour les deux sous-listes gauche et droite.

Lemme 1.2. *Soit r le rang du pivot. Après l'algorithme de pivotage, les permutations de $\{1,\ldots,r-1\}$ associées aux listes gauches G sont équiprobables. Il en est de même pour les permutations de $\{r+1,\ldots,n\}$ associées aux listes droites D.*

Preuve. La preuve est immédiate si $r=1$ ou si $r=n$. Nous supposons donc $1<r<n$ et nous choisissons une permutation possible en sortie de l'algorithme

de pivotage, notée $\pi = \pi_1 \cdot (r) \cdot \pi_2$, où π_1 est une permutation de $\{1, \ldots, r-1\}$ et π_2 est une permutation de $\{r+1, \ldots, n\}$.

Supposons que l'algorithme de pivotage ait réalisé à partir d'une liste L, les q échanges de couples inversés $(s_1, t_1), (s_2, t_2), \cdots, (s_q, t_q)$ et que π soit la permutation des rangs de la liste obtenue après pivotage. On a alors :

$$1 \leq s_1 < s_2 < \cdots < s_q \leq r-1 \qquad r+1 \leq t_q < t_{q-1} < \cdots < t_1 \leq n. \qquad (1.1)$$

Si maintenant on opère sur la permutation π la transposition τ_0 qui consiste à échanger $\pi(1)$ et $\pi(r)$ puis les q transpositions τ_k, $k = 1, .., q$, où τ_k consiste à échanger $\pi(s_k)$ et $\pi(t_k)$, on retrouve la permutation des rangs de la liste L.

Réciproquement, si nous choisissons q entiers naturels s_k et q entiers naturels t_k satisfaisant (1.1), et si nous opérons sur π la transposition τ_0 suivie des q transpositions τ_k, nous obtenons une permutation π' qui après pivotage donne la permutation π. De plus, et c'est là le point crucial, deux choix distincts des entiers s_k et t_k satisfaisant (1.1) correspondent à deux permutations π' distinctes.

Pour r, π et q fixés, le nombre de choix possibles des entiers s_k et t_k satisfaisant 1.1 est $\binom{r-1}{q}\binom{n-r}{q}$. Comme les valeurs possibles pour q sont celles de l'ensemble $\{1, \ldots, \min\{r-1, n-r\}\}$, le nombre total de permutations qui, après pivotage, donne la permutation π est

$$\sum_{q=1}^{\min\{r-1,n-r\}} \binom{r-1}{q}\binom{n-r}{q}.$$

Ce nombre ne dépend que du rang r du pivot et non de la permutation π elle-même. Il en résulte que, pour r fixé, les permutations π après pivotage sont équiprobables. Il en est bien sûr de même des permutations π_1 et π_2. ∎

Le lemme précédent permet à l'analyse en moyenne de pouvoir profiter de la structure récursive de l'algorithme de tri rapide. En effet l'hypothèse d'équiprobabilité des permutations $\pi(L)$ est conservée pour les deux appels récursifs. Nous notons M_n le nombre *moyen* de comparaisons de clés réalisées pour une liste initiale de n éléments. En conditionnant par la valeur r du rang du pivot (les n valeurs de r sont équiprobables), les lemmes 1.2 et 1.1 nous permettent d'écrire pour $n \geq 2$:

$$M_n \leq \sum_{r=1}^{n} 1/n \, [n + 1 + M(r-1) + M(n-r)].$$

Les conditions initiales sont $M(0) = M(1) = 0$ puisqu'aucune comparaison n'est exécutée pour un appel terminal. En développant l'inégalité précédente et en remarquant que

$$\sum_{r=1}^{n} M(r-1) = \sum_{r=1}^{n} M(n-r) = \sum_{r=1}^{n-1} M(r)$$

il vient :

$$M_n \leq n + 1 + 2/n \sum_{r=1}^{n-1} M(r).$$

Pour obtenir une majoration de $M(n)$, nous résolvons l'équation de récurrence :

$$S_n = n + 1 + 2/n \sum_{r=1}^{n-1} S(r)$$

avec $S(0) = S(1) = 0$. La solution de cette équation de récurrence complète, développée dans la section 2.4 du chapitre 2, est donnée par

$$S_n = 2(n + 1)(H_{n+1} - 4/3),$$

où H_n est le $n^{ième}$ nombre harmonique. Il en résulte que $M_n = O(n \log n)$.

L'analyse de l'algorithme de pivotage a également montré que le nombre de comparaisons d'un pivotage est au moins $n-1$. On montre alors à partir de l'inégalité

$$M_n \geq \sum_{r=1}^{n} 1/n[n - 1 + M(r - 1) + M(n - r)]$$

et en suivant la même démarche que pour la majoration de M_n, que

$$M(n) \geq 2(n + 1)H_n - 4n.$$

Il en résulte que $M(n) = \theta(n \log n)$.

5.1.2 Le tri fusion

Le principe de l'algorithme de tri fusion est de scinder la liste initiale L de n éléments en deux sous-listes G et D, ayant respectivement $\lfloor n/2 \rfloor$ et $\lceil n/2 \rceil$ éléments, telles que $L = G \cdot D$. Un appel récursif est réalisé pour chacune des deux listes G et D. L'*interclassement* des deux listes triées fournit ensuite la liste triée résultat.

La complexité du tri fusion dépend donc de l'algorithme d'interclassement. Etant données deux listes triées L_1 et L_2 ayant respectivement n et m éléments, il est possible de réaliser leur interclassement dans un tableau résultat R en temps $O(n+m)$ (voir exercices). Dans le tri-fusion on peut profiter du fait que les deux listes se trouvent côte-à-côte dans le tableau initial pour construire un algorithme d'interclassement plus raffiné que celui du cas général mais qui a la même complexité en temps. La procédure INTERCLASSER ci-dessous recopie dans la partie gauche du tableau de travail la liste L_1 et dans la partie droite la liste L_2 inversée. Ce placement astucieux des deux listes permet alors de replacer dans le tableau initial T les $n + m$ éléments du plus petit au plus grand en répétant $n + m$ fois le même traitement. La liste L_1 occupe initialement dans T les positions de g à m et la liste L_2 les positions de $m + 1$ à d.

```
procédure INTERCLASSER(T, g, m, d);
  pour i de g à m faire R(i) := T(i);
  pour j de m + 1 à d faire R(j) := T(d + m + 1 − j);
  i := g; j := d;
  pour k de g à d faire
    si R(i) < R(j)
      alors T(k) := R(i); i := i + 1
      sinon T(k) := R(j); j := j − 1
    finsi
  finpour.
```

L'algorithme du tri-fusion est alors implémenté par la procédure TRI-FUSION ci-dessous. Cette procédure devra utiliser un tableau de travail *global* R pour réaliser les interclassements.

```
procédure TRI-FUSION(T, i, j);
  si i < j alors
    n := j − i + 1; g := i; m := i + ⌊n/2⌋ − 1; d := j;
    TRI-FUSION(T, g, m);
    TRI-FUSION(T, m + 1, d);
    INTERCLASSER(T, g, m, d)
  finsi.
```

La figure 1.5 montre l'arborescence des appels récursifs du tri fusion pour la liste de la figure 1.1. Dans chaque sommet sont inscrits en partie haute les bornes du sous-tableau et en partie basse le sous-tableau résultat de cet appel. Les sommets des appels terminaux sont arrondis.

Complexité du tri fusion dans le pire des cas

Soit n le nombre d'éléments du tableau initial. Le premier appel récursif de la procédure TRI-FUSION porte sur un tableau de $\lfloor n/2 \rfloor$ éléments et le second sur un tableau de $\lceil n/2 \rceil$ éléments. Comme le nombre maximum de comparaisons de la procédure INTERCLASSER pour deux sous-tableaux contenant en tout n éléments est $n-1$, le nombre maximum de comparaisons pour un tableau initial de taille n, noté $f(n)$, satisfait pour $n \geq 2$ l'inégalité :

$$f(n) \leq n - 1 + f(\lfloor n/2 \rfloor) + f(\lceil n/2 \rceil).$$

La condition initiale est $f(1) = 0$. Nous obtenons une majoration de f en résolvant l'équation de récurrence (de partitions)

$$g(n) = n - 1 + g(\lfloor n/2 \rfloor) + g(\lceil n/2 \rceil),$$

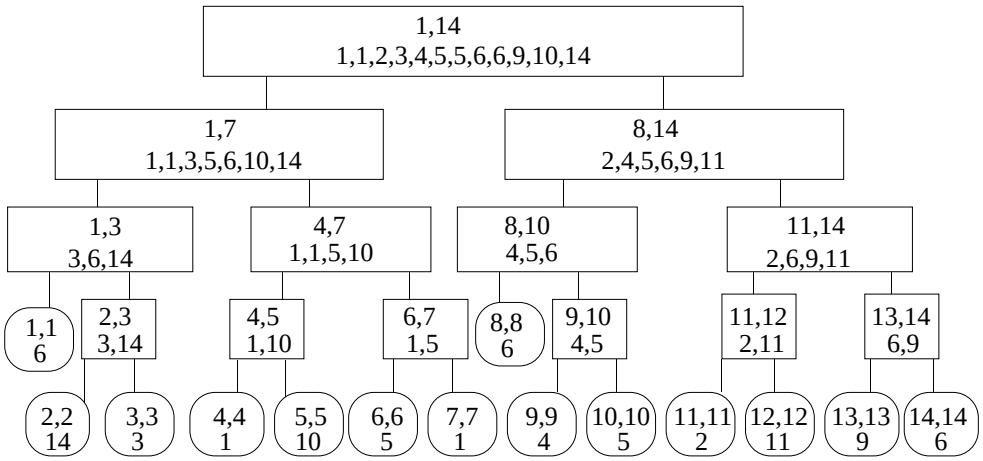


Figure 1.5: *Un tri fusion.*

avec $g(1) = 0$ comme condition initiale. Cette équation étudiée dans la section 2.3 du chapitre 2 (voir aussi l'exercice 2.4 du chapitre 2) a pour solution :

$$g(n) = \begin{cases} 0 & \text{si } n = 1 \\ n\lceil \log n \rceil + 1 - 2^{\lceil \log n \rceil} & \text{si } n \geq 2 \end{cases}$$

Il en résulte que la complexité $f(n)$ du tri fusion dans le pire des cas est $O(n \log n)$.

Le nombre minimum de comparaisons d'un interclassement de deux listes de taille m et n est $\min\{m, n\}$. Dans le cas de la procédure TRI-FUSION, ce nombre est $\lfloor n/2 \rfloor$. Le nombre minimum de comparaisons, noté $h(n)$ d'un tri fusion d'une liste de n éléments vérifie donc l'inégalité :

$$h(n) \geq \lfloor n/2 \rfloor + h(\lfloor n/2 \rfloor) + h(\lceil n/2 \rceil).$$

On obtient une minoration de h en résolvant l'équation de récurrence

$$g(n) = \begin{cases} 0 & \text{si } n = 1 \\ \lfloor n/2 \rfloor + g(\lfloor n/2 \rfloor) + g(\lceil n/2 \rceil) & \text{si } n \geq 2. \end{cases}$$

Les résultats sur les récurrences de partition permettent alors de montrer que $g(n) = \theta(n \log n)$. La complexité du tri fusion est donc $\theta(n \log n)$.

5.1.3 Le tri par tas

Rappelons qu'un tas est un arbre tournoi parfait dont les sommets contiennent les éléments d'un ensemble E muni d'une clé $c : E \mapsto F$ où F est totalement ordonné. Dans la section 3.4, des algorithmes ont été donnés pour l'insertion et pour la suppression d'un élément de plus petite clé. Ces algorithmes utilisent une implémentation du tas par un couple (T, p) où p est le nombre d'éléments contenus dans le tas et T est un tableau à p positions. Le tri par tas consiste simplement à

insérer un à un les n éléments de la liste initiale puis à réaliser n suppressions d'un élément de plus petite clé. Afin de pouvoir utiliser directement les algorithmes INSÉRERTAS et EXTRAIREMIN donnés dans la section 3.4, nous supposerons que la liste initiale est implémentée par le tableau $T[1..n]$. La procédure TRI-PAR-TAS ci-dessous utilise uniquement le tableau T pour toutes les opérations de mise à jour.

```
procédure TRI-PAR-TAS(T, n);
   p := 0; {création d'un tas vide}
   pour k de 1 à n faire INSÉRER(T(k), T);
   pour k de 1 à n faire EXTRAIREMIN(T).
```

Avant la $k^{ième}$ insertion, le tas est constitué des $k-1$ premiers éléments de T et les éléments non encore insérés occupent les $n-k$ positions suivantes. Après la $k^{ième}$ suppression, le tas occupe les $n-k$ premières positions du tableau et les éléments occupant les k dernières positions sont rangés par clé décroissante au sens large. En effet, l'échange des contenus de la racine et de la dernière feuille du tas, réalisé au début de EXTRAIREMIN(T), permet de « récupérer » l'élément supprimé lors de la $k^{ième}$ suppression dans la position $n-k+1$ du tableau.

Comme la complexité en temps des algorithmes INSÉRER et EXTRAIREMIN est $O(\log n)$ où n est le nombre d'éléments du tas, la complexité du tri par tas est $O(n \log n)$. La figure 1.6 montre l'exécution de l'algorithme de tri par tas. Une barre oblique à l'intérieur du tableau indique la séparation entre le tas en partie gauche et les éléments non contenus dans le tas en partie droite.

5.2 Tri externe

Les problèmes de tri étudiés jusqu'ici supposent que tous les objets à trier sont placés en mémoire centrale. Cependant en pratique, cette hypothèse ne peut être satisfaite en raison de la taille et du nombre des objets à trier. Ces objets sont alors placés sur des mémoires secondaires (disques, bandes magnétiques, etc.) et l'algorithme de tri exécutera des opérations de lecture et d'écriture sur le support utilisé. Comme la durée d'une opération d'entrée-sortie sur une mémoire externe est beaucoup plus grande qu'un temps d'accès à la mémoire centrale, le temps opératoire d'un algorithme de *tri externe* est essentiellement dû au nombre d'opérations d'entrée-sortie qu'il exécute. Ce nombre sera la mesure de complexité que nous utiliserons pour ce type d'algorithme.

Le mode d'accès (séquentiel, direct, etc.) est bien sûr un paramètre important pour la conception d'un algorithme de tri externe. Nous nous limiterons au cas de supports identiques à *accès séquentiel*.

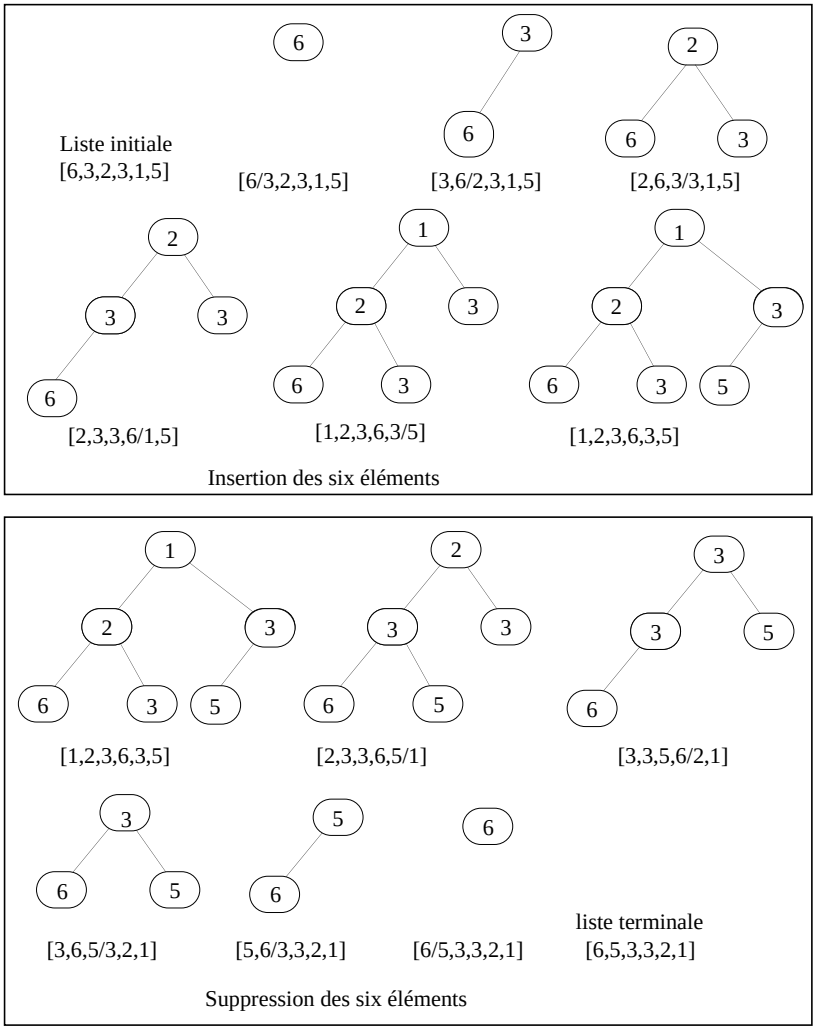


Figure 1.6: *Un tri par tas.*

La structure générale d'un algorithme de tri externe comprend la résolution de trois problèmes. Le premier est la *construction de monotonies*, c'est-à-dire de sous-listes disjointes triées. Le second est la *répartition dynamique* des monotonies sur les supports. Le troisième est l'*interclassement* des monotonies. Les solutions retenues pour ces trois problèmes doivent concourir à l'obtention sur un support de la liste triée de tous les objets en minimisant le nombre d'opérations d'entrée-sortie pour y parvenir. Il faut en particulier éviter des situations dites «de blocage» où toutes les monotonies se trouvent placées sur un même support alors que le tri n'est pas terminé.

Dans ce chapitre nous étudierons plus spécialement le problème de la répartition dynamique des monotonies et l'une de ses solutions, appelée *tri polyphasé*, qui, en utilisant les nombres de Fibonacci, conduit à un algorithme performant. Nous noterons s le nombre de supports, n le nombre d'objets à trier et m la place disponible en mémoire centrale, mesurée en nombre d'objets. On suppose que les

n objets sont initialement placés sur un même support noté S_0.

5.2.1 Construction des monotonies

Pour limiter le nombre d'entrées-sorties, il convient de construire un petit nombre de monotonies de grande taille. Deux techniques conduisant à un nombre comparable d'entrées-sorties répondent à cet objectif. L'une construit des monotonies de même taille m en utilisant un algorithme de tri interne; l'autre construit des monotonies en général plus longues par sélection et remplacement.

Monotonies de même taille

Chaque itération consiste à lire en mémoire centrale les m objets suivants du support S_0 (ou le nombre restant d'objets s'il est inférieur à m), à trier les objets contenus en mémoire centrale par un algorithme de tri interne et à écrire la liste obtenue sur un support qui dépend de la stratégie de répartition. L'algorithme réalise ainsi d'une part $2n$ opérations d'entrée-sortie (n lectures et n écritures) et d'autre part $\lceil n/m \rceil$ tris internes.

Sélection et remplacement

L'algorithme est initialisé par la lecture en mémoire centrale des m premiers objets de S_0. On note E l'ensemble des objets contenus en mémoire centrale et f l'objet suivant sur S_0. L'ensemble E contient un sous-ensemble M initialement vide d'objets dits *marqués* car ils ne peuvent appartenir à la monotonie en cours de construction. Le calcul de la monotonie suivante commence lorsque M est vide et se termine lorsque M est égal à E. Une itération consiste à calculer le plus petit élément e de $E-M$, à placer e à la suite de la monotonie courante en construction et à lire en mémoire centrale l'objet f de S_0 dans la place occupée par e. Si f est plus petit que e, il est ajouté à l'ensemble M car il ne pourra pas faire partie de la monotonie courante. Le pire des cas correspond aux objets rangés initialement dans l'ordre inverse de l'ordre requis. Après la lecture en mémoire centrale des m premiers objets, tout nouvel objet lu en mémoire est immédiatement marqué. Chaque monotonie est alors de taille m. Dans le cas général, la taille de chaque monotonie (sauf peut-être la dernière) est plus grande que m. Si les objets non marqués en mémoire centrale sont organisés en tas, la complexité des opérations élémentaires hors entrées-sorties est $O(n \log m)$ et le nombre d'opérations d' entrée-sortie est $2n$ puisque chaque élément est lu et écrit une seule fois. La figure 2.1 décrit les dix premières étapes d'un exemple de construction de monotonies par sélection et remplacement pour une mémoire de 5 cellules (m=5). La liste initiale sur support externe est S_0=$(4, 3, 6, 7, 10, 5, 1, 2, 8, 3, 5, 9, 4)$. L'élément de la ligne i et de la colonne j est la clé de l'objet contenu dans la cellule i à l'étape j. Si cette clé est suivie d'une étoile, la cellule i appartient à M. Si cette clé est

suivie d'une pointe de flèche, l'objet correspondant est le suivant de la monotonie en cours. Comme après l'étape 8 toutes les cellules sont marquées, la première monotonie est $(3, 4, 5, 6, 7, 8, 10)$. La seconde commence par $(1, 2, \ldots)$.

	1	2	3	4	5	6	7	8	9	10
1	4	4^	1*	1*	1*	1*	1*	1*	1^	4
2	3^	5	5^	2*	2*	2*	2*	2*	2	2^
3	6	6	6	6^	8	8^	5*	5*	5	5
4	7	7	7	7	7^	3*	3*	3*	3	3
5	10	10	10	10	10	10	10^	9*	9	9

changement de monotonie

Figure 2.1: *Sélection et remplacement.*

5.2.2 Répartition des monotonies

Une fois construite, chaque monotonie doit être placée sur l'un des s supports. La stratégie de répartition des monotonies sur les supports est ici essentielle pour l'efficacité des opérations ultérieures d'interclassement. Nous examinons deux stratégies de répartition. Le *tri équilibré* a le mérite d'être simple à implémenter, mais utilise mal les supports, réalise $2n$ opérations d'entrée-sortie par phase et surtout est contraint d'exécuter des recopies de monotonies. Le *tri polyphasé*, en fondant sa stratégie de répartition sur les nombres de Fibonacci, réalise un nombre limité d'opérations d'entrée-sortie à chaque phase, n'est jamais contraint à de simples recopies, mais doit par contre introduire des *monotonies fantômes* si le nombre total de monotonies initiales n'est pas l'un des éléments d'une suite spécifique déduite de la suite de Fibonacci d'ordre $s - 1$.

Le tri équilibré

Le tri équilibré consiste à définir deux sous-ensembles $\mathcal{L}$ (supports de lecture) et $\mathcal{E}$ (supports d'écriture) de q supports chacun. Lors de leur création, les monotonies sont réparties *uniformément* sur les supports de $\mathcal{L}$. Une *phase* de l'algorithme est décrite par la procédure PHASE-TRI-ÉQUILIBRÉ ci-dessous. L'itération fondamentale de cette procédure, appelée *fusion élémentaire* et implémentée par FUSION(σ, k), lit la première monotonie de chaque support de σ, interclasse ces monotonies en mémoire centrale et écrit la monotonie résultat sur le support d'écriture k.

```
procédure PHASE-TRI-ÉQUILIBRÉ;
  k := 0  (mod q);
  tantque toutes les monotonies de L n'ont pas été lues faire
    soit σ le sous-ensemble des supports non vides de L;
    FUSION(σ, k);
    k := k + 1  (mod q)
  fintantque;
  L := ensemble des supports de E;
  E := ensemble des supports de L.
```

Une phase du tri équilibré fait passer le nombre de monotonies de M à $\lceil M/q \rceil$ et les nouvelles monotonies restent uniformément réparties sur les supports. Si M_0 est le nombre initial de monotonies, le nombre maximal de phases est en $O(\log M_0)$. Chaque phase réalise $2n$ entrées-sorties car chaque objet est lu et écrit une fois. Il en résulte que le nombre d'opérations d'entrée-sortie du tri équilibré est en $O(n \log M_0)$. La figure 2.2 décrit un tri équilibré. Les M_i sont les monotonies initiales et f est l'opérateur de fusion élémentaire. Les sous-ensembles $\{S_1, S_2, S_3\}$ et $\{S_4, S_5, S_6\}$ sont alternativement utilisés en lecture et en écriture. Lors de la phase 1, la monotonie M_3 a simplement été recopiée.

	phase 0	phase 1	phase 2
S1	M1,M2,M3		f(f(M1,M4,M6),f(M2,M5,M7),M1)
S2	M4,M5		
S3	M6,M7		
S4		f(M1,M4,M6)	
S5		f(M2,M5,M7)	
S6		M3=f(M3)	

Figure 2.2: *Le tri équilibré.*

Comme nous l'avons souligné, le principal inconvénient du tri équilibré est de réaliser une lecture et une écriture de chaque objet lors d'une même phase. De plus, si en fin de phase tous les supports de lecture sont vides sauf un ne contenant qu'une seule monotonie, celle-ci sera simplement recopiée et le nombre de monotonies ne diminuera pas lors de cette fusion élémentaire.

Le tri polyphasé

Par rapport au tri équilibré, le tri polyphasé ne réalise au cours d'une phase qu'un nombre limité d'opérations d'entrée-sortie, n'exécute pas de simples recopies et

utilise beaucoup mieux l'ensemble des supports en se servant des nombres de Fibonacci pour répartir initialement les monotonies.

L'itération fondamentale du tri polyphasé, également appelée *phase*, consiste à choisir un support dit d'écriture, noté e et à réaliser k fusions élémentaires sur le support e. L'*état du système* est défini par la suite ordonnée par valeurs décroissantes $N = (N_1, \ldots, N_s)$ des nombres de monotonies sur les S supports, chaque rang de cette suite correspondant à un numéro logique de support. On passe des numéros logiques aux numéros physiques par une permutation et l'on appelle *total de* N le nombre $\tau(N) = \sum_{i\in\{1,\ldots,s\}} N_i$. Le lemme suivant détermine pour un état terminal N fixé, quel est l'état initial de plus grand total permettant d'atteindre N *en une seule phase*.

Lemme 2.1. *L'état* $(N_1 + N_2, N_1 + N_3, \cdots, N_1 + N_s, 0)$ *est un état de total maximal permettant d'atteindre l'état* N *en une seule phase.*

Preuve. Notons $\psi(N) = (N_1 + N_2, N_1 + N_3, \cdots, N_1 + N_s, 0)$. En prenant comme suppport d'écriture e celui qui ne contient aucune monotonie et en réalisant N_1 fusions élémentaires de l'ensemble des autres supports sur e, on construit une phase qui fait passer de l'état $\psi(N)$ à l'état $(N_2, N_3, \cdots, N_s, N_1)$ qui, à une permutation près, est l'état N. Remarquons de plus que $\tau(\psi(N)) = \tau(N)+(s-2)N_1$. Supposons maintenant que M soit un état initial permettant de passer en une seule phase de l'état M à l'état N. Considérons les numéros logiques des supports dans M. Soit e le numéro du support d'écriture choisi et $a_i, i \in \{1, \ldots, s\} - \{e\}$ le nombre d'occurrences du support numéro i dans les k fusions élémentaires de la phase. Le nombre de monotonies du support numéro i devient $M'_i = M_i - a_i, i \in \{1, \ldots, s\} - \{e\}$ et celui du support numéro e devient $M'_e = M_e + k$. Comme nous avons $k \leq M'_e$ et pour tout i dans $\{1, \ldots, s\} - \{e\}$, $a_i \leq k$, il vient :

$$\tau(M) = \tau(M') + (\sum_{i\in\{1,\ldots,s\}-\{e\}} a_i) - k \leq \tau(M') + (s-2)k \leq \tau(N) + (s-2)N_1$$

car $\tau(M') = \tau(N)$ et $k \leq N_1$. ∎

Soit $E^1 = (1, 0, \cdots, 0)$ l'état final associé à la liste des objets totalement triée sur un seul support. Il résulte du lemme précédent que $\psi^{(n)}(E^1)$ est une répartition initiale sur les supports qui permet d'obtenir l'état E^1 en n phases. Si m est le nombre initial de monotonies, le tri polyphasé utilise comme répartition initiale le vecteur $\psi^{(k)}(E^1)$ où k est le plus petit entier tel que $\tau(\psi^{(k)}(E^1)) \geq m$. Il faudra donc ajouter et répartir sur les supports $m - \tau(\psi^{(k)}(E^1))$ *monotonies fantômes*.

Répartition initiale des monotonies

Nous examinons dans cette section la nature et les propriétés essentielles de la suite $\psi^{(n)}(E^1)$, $n \in \mathbb{N}$ et la répartition initiale des monotonies fantômes. Notons que n est l'indice générique de la suite et ne représente pas dans cette section le

nombre d'éléments à trier. Nous notons $F_s^n = (f^n{}_1, \ldots, f^n{}_s)$ le vecteur $\psi^{(n)}(E^1)$ et $T_s(n)$ son total. Il résulte directement de la définition de ψ que F_s^{n+1} est obtenu à partir de F_s^n par la relation de récurrence suivante :

$$f_i^{n+1} = \begin{cases} f_1^n + f_{i+1}^n & \text{si } i \leq s-1 \\ 0 & \text{si } i = s \end{cases}$$

avec comme conditions initiales :

$$f_i^0 = \begin{cases} 1 & \text{si } i = 1 \\ 0 & \text{sinon} \end{cases}$$

Le tableau ci-dessous montre les valeurs de F_s^n pour $s = 6$ et $0 \leq n \leq 8$.

n	f_1^n	f_2^n	f_3^n	f_4^n	f_5^n	f_6^n	$T(n)$
0	1	0	0	0	0	0	1
1	1	1	1	1	1	0	5
2	2	2	2	2	1	0	9
3	4	4	4	3	2	0	17
4	8	8	7	6	4	0	33
5	16	15	14	12	8	0	65
6	31	30	28	24	16	0	129
7	61	59	55	47	31	0	253
8	120	116	108	92	61	0	497

Dans le cas particulier de trois supports, l'équation de récurrence précédente s'écrit pour $n \geq 2$:

$$f_1^{n+1} = f_1^n + f_2^n \qquad f_2^{n+1} = f_1^n.$$

Sa solution est immédiate car nous avons pour $n \geq 2$:

$$f_1^{n+1} = f_1^n + f_1^{n-1} \qquad f_2^{n+1} = f_1^{n-1} + f_2^{n-1} = f_2^n + f_2^{n-1}.$$

Compte tenu des conditions initiales, à savoir $f_1^0 = f_1^1 = 1$, $f_2^0 = 0$ et $f_2^1 = 1$, nous obtenons :

$$f_1^n = \varphi(n+1) \qquad f_2^n = \varphi(n) \qquad T_3(n) = \varphi(n+2)$$

où $\varphi(n)$ est l'élément de rang n de la suite de Fibonacci d'ordre deux définie par :

$$\varphi(n) = \begin{cases} \varphi(n-1) + \varphi(n-2) & \text{si } n \geq 2 \\ 1 & \text{si } n = 1 \\ 0 & \text{si } n = 0 \end{cases}$$

Dans le cas général, la suite de Fibonnaccci d'ordre p, notée $\Phi^p(n)$ est définie par :

$$\Phi^p(n) = \begin{cases} \sum_{k=1}^{p} \Phi^p(n-k) & \text{si } n \geq p \\ 1 & \text{si } n = p-1 \\ 0 & \text{si } n \in \{0, \ldots, p-2\} \end{cases}$$

On montre alors que dans le cas de s supports :

$$f_k^n = \Phi^{s-1}(n+(s-1)-2)+\Phi^{s-1}(n+(s-1)-3)+\cdots+\Phi^{s-1}(n+(s-1)-(s+1-k)$$

et que le nombre total de monotonies est donné par :

$$T_s(n) = (s-1)\Phi^{s-1}(n+(s-3)) + (s-2)\Phi^{s-1}(n+(s-4)) + \cdots + \Phi^{s-1}(n-1)$$

Supposons que le nombre initial m de monotonies soit compris strictement entre $T_s(n)$ et $T_s(n+1)$. Il faut donc créer $T_s(n+1)-m$ monotonies fantômes que l'on a avantage à répartir aussi uniformément que possible sur les supports. L'algorithme FIBONACCI(s,m) ci-dessous réalise cet objectif en construisant la répartition F_s^{n+1} et en donnant la priorité aux monotonies fantômes.

```
procédure FIBONACCI(s, m);
  {On suppose connu l'entier n tel que T_s(n) < m < T_s(n+1)}
  r := T_s(n+1) − T_s(n); F := T_s(n+1) − m;
  pour S de 1 à s − 1 faire d(S) := f_S^{n+1} − f_S^n;
  S := 0;
  pour j de 1 à r faire
    S := S + 1  (mod s − 1);
    si d(S) > 0 alors
      d(S) := d(S) − 1;
      si j < F
        alors écrire une monotonie fantôme sur le support S + 1
        sinon écrire la monotonie suivante sur le support S + 1;
      finsi
    finsi
  finpour;
  pour S de 1 à s − 1 faire
    écrire les f_S^n monotonies suivantes sur le support S
  finpour.
```

La figure 2.3 montre la répartition réalisée par l'algorithme précédent pour 19 monotonies initiales et 6 supports. Pour cet exemple, on a : $T_6(3)$=17, $T_6(4)$=33 et $F_6(4)$=$(8,8,7,6,4,0)$.

Analyse du tri polyphasé

La fonction génératrice $\hat{\Phi}^p$ de la suite de Fibonacci d'ordre p est donnée par :

$$\hat{\Phi}^p(z) = \sum_{n\in\mathbb{N}} \Phi^p(n) z^n = \frac{z^{p-1}}{1-z-z^2-\cdots-z^p}.$$

S1	f	f	f	m	m	m	m	m
S2	f	f	f	m	m	m	m	m
S3	f	f	f	m	m	m	m	
S4	f	f	f	m	m	m		
S5	f	f	m	m				

f: monotonie fantôme
m: monotonie réelle

Figure 2.3: *Répartition des monotonies fantômes.*

Il résulte alors des formules du paragraphe précédent donnant les f_p^n et $T_s(n)$ que les fonctions génératrices $\hat{f}_1$ et $\hat{T}_s$ des suites f_1^n et $T_s(n)$ sont respectivement :

$$\hat{f}_1(z) = \sum_{n\in\mathbb{N}} f_1(n)z^n = \frac{1}{1-z-z^2-\cdots-z^p}$$

et

$$\hat{T}_s(z) = \sum_{n\in\mathbb{N}} T_s(n)z^n = \frac{(s-1)z+(s-2)z^2+\cdots+z^{s-1}}{1-z-z^2-\cdots-z^p}.$$

Comme les zéros du polynôme $1-z-z^2-\cdots-z^p$ sont tous de module strictement plus petit que l'unité, on a $T_s(n) = O(a^n)$ où a est un rationnel strictement plus grand que l'unité. Si donc $m = T_s(n)$ est le nombre initial de monotonies, le *nombre de phases* du tri polyphasé est en $O(\log m)$.

Pour évaluer le nombre d'opérations d'entrée-sortie qui, rappelons-le, est notre mesure de complexité, nous supposons que toutes les monotonies initiales ont la même taille t et nous introduisons la notion de *transfert*. Un transfert correspond au «passage» de t objets d'un support sur un autre. Nous allons compter le nombre de transferts réalisés par un tri polyphasé à partir de la distribution initiale F_s^n. Notons que ce calcul prend en compte la taille des monotonies construites au cours de l'exécution du tri. Le tableau ci-dessous, construit pour un exemple à huit phases et six supports, montre l'évolution de la taille des monotonies sur chaque support, du nombre de monotonies sur chaque support, et du nombre de transferts t par phase.

n	S_1	S_2	S_3	S_4	S_5	S_6	t
1	$[1]^{61}$	$[1]^{59}$	$[1]^{55}$	$[1]^{47}$	$[1]^{31}$	[0]	253
2	$[5]^{31}$	$[1]^{30}$	$[1]^{28}$	$[1]^{24}$	$[1]^{16}$	[0]	5×31
3	$[9]^{16}$	$[5]^{15}$	$[1]^{14}$	$[1]^{12}$	$[1]^{8}$	[0]	9×16
4	$[17]^{8}$	$[9]^{8}$	$[5]^{7}$	$[1]^{6}$	$[1]^{4}$	[0]	17×8
5	$[33]^{4}$	$[17]^{4}$	$[9]^{4}$	$[5]^{3}$	$[1]^{2}$	[0]	33×4
6	$[65]^{2}$	$[33]^{2}$	$[17]^{2}$	$[9]^{2}$	$[5]^{1}$	[0]	65×2
7	$[129]^{1}$	$[65]^{1}$	$[33]^{1}$	$[17]^{1}$	$[9]^{1}$	[0]	129×1
8	$[253]^{1}$	[0]	[0]	[0]	[0]	[0]	253×1

Un élément $[a]^b$ du tableau où a et b sont deux entiers doit être interprété comme b monotonies de taille a. Les nombres de Fibonacci apparaissent partout dans ce tableau. On montre que d'une manière générale pour n phases et s supports, la première ligne du tableau central est :

$$([T_s(0)]^{f_1^n}, [T_s(0)]^{f_2^n}, \cdots, [T_s(0)]^{f_{s-1}^n})$$

et que la ligne associée à la phase p (hormis le nombre de transferts) est :

$$([T_s(p-1)]^{f_1^{n-p+1}}, [T_s(p-2)]^{f_2^{n-p+1}}, \cdots, [T_s(p-s+1)]^{f_{s-1}^{n-p+1}})$$

en adoptant la convention que $T_s(k) = 1$ si $k \leq 0$. Il résulte alors de la structure de la ligne générale du tableau central que le nombre de transferts associé à la phase p est donné par :

$$f_{s-1}^{n-p+1}[T_s(p-1) + T_s(p-2) + \cdots + T_s(p-s+1)]$$

ce qui s'écrit encore : $f_1^{n-p} \times T_s(p)$. Le nombre total de transferts, noté $\theta_s(n)$, est donc égal à $T_s(n) + \sum_{p=1}^{n} f_1^{n-p} \times T_s(p)$. Il en résulte que $\theta_s(n)$ est égal à la somme de $T_s(n)$ et du coefficient de z^n dans $\hat{f}_1(z)\hat{T}_s(z)$.

Lorsque n tend vers $+\infty$, le rapport $\theta_s(n)/T_s(n)$ est équivalent à une fonction linéaire du type $A_s \ln(T_s(n)) + B_s$ où A_s et B_s ne dépendent que du nombre s de supports. Ce comportement a également été observé expérimentalement pour la répartition issue de la procédure FIBONACCI(s, m) lorsque le nombre m de monotonies initiales (de même taille) tend vers $+\infty$ mais n'est pas nécessairement égal à l'une des valeurs de la suite $T_s(n)$.

Notes

Dans ce chapitre, nous nous sommes limités à la présentation de trois algorithmes classiques de tri interne et un algorithme de tri externe. L'ouvrage le plus complet du domaine est :
D. Knuth, *The Art of Computer Programming*, tome **3** : Sorting and Searching, Addison Wesley, 1973.

Exercices

5.1. Le *tri bulle* d'un tableau $t[i..j]$ de n éléments consiste, pour k variant de i à $j-1$, à faire «remonter» le plus petit élément du sous-tableau $t[k..j]$ en position k par une suite d'échanges. La procédure TRI-BULLE ci-dessous implémente cet algorithme :

```
procédure TRI-BULLE(t, i, j);
  pour k de i à j − 1 faire
    pour p de j − 1 à k pas −1 faire
      si c(p + 1) < c(p) alors ÉCHANGER(t, p + 1, p)
      {c(k) est la clé de l'élément t(k)}
    finpour
  finpour.
```

a) Déterminer le nombre de comparaisons exécutées par le tri bulle sur un tableau à n éléments.
b) Déterminer le nombre maximum et le nombre minimum d'échanges exécutés par le tri bulle sur un tableau à n éléments.
On suppose maintenant que les clés sont distinctes deux à deux et l'on note $\pi(t)$ la permutation de $\{1, \ldots, n\}$ associée au rang des éléments du tableau $t[i..j]$.
c) Démontrer que le nombre d'échanges réalisés par le tri bulle est égal au nombre d'inversions de π.
On note $I_{n,k}$ le nombre de permutations de $\{1, \ldots, n\}$ ayant k inversions. Exprimer $I_{n,k}$ en fonction des $I_{n-1,p}$ où $p \in \{\max\{0, k-n+1\}, \ldots, k\}$.
En déduire que la fonction génératrice $I_n(z) = \sum_{k\in\mathbb{N}} I_{n,k} z^k$ vérifie :

$$I_n(z) = (1 + z + z^2 + \ldots + z^{n-1}) I_{n-1}(z)$$

En supposant que les permutations $\pi(t)$ sont équiprobables, montrer que le nombre moyen d'échanges du tri bulle est $\frac{n(n-1)}{4}$.

5.2. Le *tri par sélection* d'un tableau $t[i..j]$ à n éléments consiste à déterminer l'élément minimum du tableau, à échanger cet élément avec le premier élément du tableau et à réaliser récursivement ces opérations sur le tableau $t[i+1..j]$. La procédure TRI-SÉLECTION ci-dessous implémente cet algorithme :

```
procédure TRI-SÉLECTION(t, i, j);
  si j > i alors
    soit c(k) = min{c(p) | p ∈ {i, ..., j}};
    ÉCHANGER(t, k, i);
    TRI-SÉLECTION(t, i + 1, j)
  finsi.
```

a) Déterminer, en fonction de n, le nombre d'échanges réalisés par la procédure TRI-SÉLECTION.
b) Déterminer, en fonction de n, le nombre minimum et le nombre maximum de comparaisons réalisées par la procédure TRI-SÉLECTION.

c) Déterminer le nombre moyen de comparaisons réalisées par la procédure TRI-SÉLECTION si les clés sont distinctes deux à deux et les permutations des rangs équiprobables.

5.3. Le *tri par insertion* d'un tableau $t[i..j]$ à n éléments consiste à réaliser un appel récursif pour le tableau $t[i..j-1]$ et à insérer le dernier élément du tableau à sa place dans le tableau en décalant d'une position vers la droite les éléments qui le suivent. La procédure TRI-INSERTION ci-dessous implémente cet algorithme :

```
procédure TRI-INSERTION(t, i, j);
  si j > i alors
    TRI-INSERTION(t, i, j − 1);
    k := j;
    tantque k > i etalors c(k) < c(k − 1) faire
      ÉCHANGER(t, k − 1, k)
    fintantque
    {c(k) est la clé de l'élément t(k)}
  finsi.
```

a) Déterminer la complexité du tri par insertion en nombre de comparaisons.
On suppose que les clés sont distinctes deux à deux et que les permutations des rangs sont équiprobables.
b) Démontrer que pour l'appel récursif, les permutations des rangs du tableau $t[i..j-1]$ sont équiprobables.
c) En déduire le nombre moyen de comparaisons du tri par insertion d'un tableau à n éléments.

5.4. Soit E un ensemble de n éléments dont les clés sont des entiers codés avec K chiffres de l'ensemble $\{0,\ldots,9\}$. Le *tri par champs* consiste à construire K listes $\{L_1,\ldots,L_K\}$ où la liste L_k est triée pour les k chiffres de poids faible. La liste L_K est donc une liste triée. On note $p^{(k)}$, $k \in \{1,\ldots,K\}$, le $k^{ième}$ chiffre de poids faible d'un entier p à K chiffres. La procédure TRI-CHAMPS ci-dessous implémente cet algorithme à partir de la liste L des n éléments à trier :

```
procédure TRI-CHAMPS(L);
  pour k de 1 à K faire
    pour i de 1 à n faire
      e :=DÉFILER(L);
      p := c(e)^(k);
      {c(e) est la clé de l'élément e}
      ENFILER(e, L_p)
    finpour;
    pour q de 0 à 9 faire L := L · L_q; L_q := () finpour
  finpour.
```

Les opérations DÉFILER et ENFILER correspondent à la gestion d'une file.
a) Montrer que la liste L obtenue en retour de la procédure TRI-CHAMPS est triée.
b) Exprimer en fonction de n et K la complexité (au sens classique) du tri par champs.
c) En déduire une condition pour que ce tri soit linéaire par rapport à n.

5.5. Soient L_1 et L_2 deux listes triées contenant respectivement n et m entiers et rangées dans deux tableaux T_1 et T_2. Ecrire un algorithme de complexité en temps $O(n+m)$ pour interclasser de ces deux listes dans un tableau T.

5.6. On utilise une variante de l'algorithme de tri rapide pour déterminer le $k^{ième}$ élément d'un ensemble de n éléments munis d'une clé. On suppose que les n clés sont distinctes deux à deux. On considère dans un premier temps la fonction SÉLECT ci-dessous :

```
fonction SÉLECT(t, i, j, k) : élément;
{on suppose que 1 ≤ k ≤ n}
  p :=PIVOTER(t, i, j);
  si k ≤ p
    alors SÉLECT :=SÉLECT(t, i, p, k)
    sinon SÉLECT :=SÉLECT(t, p + 1, j, k − p)
  finsi.
```

où la fonction PIVOTER retourne un entier $p \in \{i+1, \ldots, j-1\}$ et réorganise en temps $O(n)$ le tableau $t[i..j]$ en deux sous-tableaux $t[i..p]$ et $t[p+1..j]$ tels qu'un élément quelconque de $t[i..p]$ possède une clé inférieure ou égale à celle d'un élément quelconque de $t[p+1..j]$.
a) Démontrer que la procédure SÉLECT détermine le $k^{ième}$ élément du tableau.
b) Montrer que, sans hypothèse particulière sur l'algorithme de pivotage, la fonction SÉLECT peut réaliser $O(n^2)$ comparaisons.
c) Montrer que si l'algorithme de pivotage garantit que la taille du sous-tableau de l'appel récursif ne dépasse pas αn où $\alpha < 1$, la complexité en nombre de comparaisons de la fonction SÉLECT est $O(n)$.
On considère l'algorithme de choix du pivot suivant :

1. découper le tableau en $\lfloor n/5 \rfloor$ blocs $\{B_1, \ldots, B_{\lfloor n/5 \rfloor}\}$ de cinq éléments; les éléments restants (au plus 4) ne seront pas considérés dans la suite de l'algorithme;
2. déterminer les éléments médians m_k des B_k, $k \in \{1, \ldots, \lfloor n/5 \rfloor\}$;
3. utiliser la fonction SÉLECT pour déterminer l'élément d'ordre $\left\lfloor \frac{n+5}{10} \right\rfloor$ de la liste $(m_1, \ldots, m_{\lfloor n/5 \rfloor})$; (si $\lfloor n/5 \rfloor$ est pair, l'élément sélectionné est l'élément médian de la liste $(m_1, \ldots, m_{\lfloor n/5 \rfloor})$).

d) Montrer que le pivot choisi est strictement supérieur à au moins $3\left\lfloor \frac{n-5}{10} \right\rfloor$ éléments de t et est inférieur ou égal à au moins $3\left\lfloor \frac{n-5}{10} \right\rfloor$ éléments de t. En déduire

que pour $n \geq 75$, le sous-tableau de l'appel récursif est de taille au plus égale à $3n/4$.

On considère alors la fonction SÉLECT suivante :

```
fonction SÉLECT(t, i, j, k) : élément;
  si (j − i) ≤ 74 alors
    TRIER(t, i, j);
    extraire l'élément d'ordre k de t;
    exit
  sinon
    pour q de 1 à ⌊n/5⌋ faire
      m(q) :=MÉDIAN(t, 5(q − 1) + i, 5(q − 1) + i + 4)
      {m(q) est l'indice dans t de l'élément médian}
      {de {t(5(q − 1) + i), ..., t(5(q − 1) + i + 4)}}
    finpour;
    r :=SÉLECT(m, 1, ⌊n/5⌋ , ⌊(n+5)/10⌋);
    {r est l'indice dans t de l'élément}
    { d'ordre ⌊(n+5)/10⌋ de (t_m(1), ..., t_m(⌊n/5⌋))}
    p :=PARTITION(t, i, j, r);
    si p ≥ k
      alors SÉLECT :=SÉLECT(t, i, p, k)
      sinon SÉLECT :=SÉLECT(t, p + 1, j, k − p)
    finsi
  finsi.
```

La procédure TRIER est un algorithme de tri quelconque. La fonction MÉDIAN fournit l'élément de rang 3 d'une liste de 5 éléments. La procédure PARTITION réorganise le tableau t en prenant $t(r)$ comme pivot ; après son exécution, un élément quelconque de $t[i..p]$ a une clé inférieure ou égale à $t(r)$, un élément quelconque de $t[p+1..j]$ a une clé strictement supérieure à $t(r)$.

e) Démontrer la validité de la fonction SÉLECT.

f) Montrer que la complexité de la fonction SÉLECT est $O(n)$.

Chapitre 6

Arbres et ensembles ordonnés

Ce long chapitre contient la description de plusieurs familles d'arbres binaires de recherche équilibrés. Nous commençons par les arbres AVL, puis nous décrivons les arbres a–b, avec notamment des opérations plus élaborées comme la concaténation et la scission, et le coût amorti d'une suite d'opérations. Ensuite, nous présentons les arbres bicolores, et enfin une réalisation de structures persistantes sur ces arbres.

Introduction

La manipulation de grands volumes d'informations est une tâche fréquente en algorithmique. Ce chapitre présente plusieurs structures de données pour leur gestion. Les données peuvent être très variées, et organisées différemment d'une application à l'autre. Nous supposons que chaque donnée comporte un « champ » particulier, appelé sa *clé* qui peut être, par exemple, une chaîne de caractères ou un numéro. Les clés permettent d'identifier les données, et des données différentes ont donc des clés distinctes. Une clé permet d'accéder à la donnée qu'elle représente, selon un mécanisme qui dépend de la situation considérée, et que nous ne considérons pas ici. Les clés elles-mêmes, et c'est là leur intérêt, sont prises dans un ensemble totalement ordonné (appelé l'« univers »). La manipulation des données se fait donc à travers la manipulation des clés.

On peut distinguer trois catégories de structures de données, en fonction de leur comportement dans le temps. Une structure *statique* représente un ensemble qui ne varie pas (ou très peu) dans le temps. Un exemple typique est un lexique fixe. D'autres exemples sont fournis en géométrie algorithmique ; dans le problème de localisation, il s'agit de déterminer dans quelle région d'une subdivision fixe du plan se trouve un point donné. L'action principale sur une telle structure est la recherche d'informations, les mises à jour étant exceptionnelles et de ce fait négligeables. Un soin particulier (et donc un certain temps) peut être consacré à

la construction d'une structure « optimale ». Même si l'ensemble des données est statique, la structure de données peut évoluer dans le temps, pour améliorer le temps de réponse en fonction de la fréquence des interrogations.

Les structures *dynamiques*, qui font l'objet de ce chapitre, permettent en plus des modifications de l'ensemble des données représentées. Il s'agit principalement d'implémenter des *dictionnaires*, donc de réaliser la recherche, l'insertion et la suppression d'informations, ou des *listes concaténables* qui exigent comme opérations supplémentaires la concaténation et la scission. Les arbres binaires de recherche équilibrés réalisent ces opérations de manière efficace.

Ces structures sont éphémères dans le sens où une mise à jour détruit de façon définitive la version précédente. Une structure est *persistante* si elle est dynamique, et si elle conserve également les *versions antérieures* de la structure. On peut distinguer deux degrés de persistance : si l'on dispose de la *liste* des versions précédentes, on peut rechercher la présence d'éléments dans une version antérieure ; si de plus on est autorisé à modifier les versions antérieures, on aboutit à un *arbre* de versions, la plus élaborée des structures persistantes. Nous allons présenter une réalisation efficace de la liste des versions.

Dans ce chapitre, nous proposons des solutions au problème que voici : étant donné un ensemble S, sous-ensemble de l'univers noté U, effectuer de manière efficace les opérations suivantes, qui caractérisent un dictionnaire : rechercher un élément dans S, insérer un élément dans S, supprimer un élément dans S. Nous examinerons aussi d'autres opérations, comme la scission et la concaténation.

Eliminons tout de suite le cas d'un univers qui serait « petit ». On peut alors implémenter ces opérations de façon très simple. Un ensemble S est représenté par un tableau indicé par les éléments de U, qui donne sa fonction caractéristique, c'est-à-dire qui indique, pour chaque élément de U, s'il appartient ou non à S. Nous supposons dans la suite que l'univers U est très grand, trop grand en tout cas pour permettre cette réalisation, et que S est petit par rapport à U.

Il apparaît que les arbres sont une structure de données particulièrement bien adaptée à la réalisation efficace des trois opérations de base (recherche, insertion, suppression). Lorsque l'arbre est « équilibré » selon des critères à préciser, on peut éviter qu'il dégénère en une structure trop proche d'une liste, et on peut alors effectuer les opérations en temps logarithmique en fonction du nombre d'éléments de l'ensemble S. Bien évidemment, l'insertion et la suppression peuvent déséquilibrer l'arbre. Une partie substantielle des algorithmes est consacrée aux opérations de rééquilibrage qui doivent être réalisées avec soin.

Une première espèce d'arbres binaires équilibrés est constituée des arbres *AVL*, nommée ainsi d'après les initiales de leurs deux inventeurs (Adelson-Velskii et Landis). Nous considérons ensuite les arbres a–b qui sont des arbres où chaque nœud a entre a et b fils. Nous présentons la version balisée (dans le sens défini ci-dessous) de ces arbres, et montrons comment les utiliser pour la concaténation et la scission. Les arbres bicolores, que nous introduisons ensuite, sont une autre

famille d'arbres équilibrés. Ils ont été conçus à l'origine en vue de l'implémentation des arbres a–b, mais leur importance dépasse ce cadre. Ils nous serviront notamment pour la représentation de structures persistantes, présentées dans la dernière section.

6.1 Arbres de recherche

6.1.1 Définition

Nous allons parler d'arbres de recherche, et principalement d'arbres binaires de recherche. Il s'agit d'arbres binaires étiquetés aux sommets. L'étiquette d'un sommet x est la *clé* ou le *contenu* du sommet, et notée $c(x)$. Si A est un arbre binaire non vide, on note A_g et A_d ses sous-arbres gauche et droit. Pour tout sommet x, on note $A(x)$ le sous-arbre de racine x, et $A_g(x)$, $A_d(x)$ les sous-arbres de $A(x)$. Un *arbre binaire de recherche* est un arbre muni d'une fonction clé c sur l'ensemble de ses sommets vérifiant

$$c(y) < c(x) < c(z)$$

pour tout sommet x, et pour tous sommets y de $A_g(x)$ et z de $A_d(x)$. Il revient au même de dire que les clés sont croissantes si on les liste dans l'ordre symétrique.

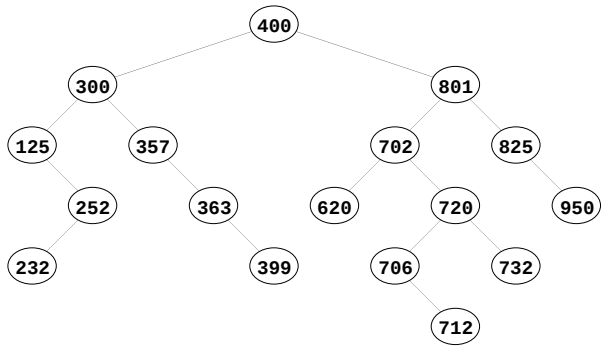


Figure 1.1: *Un arbre binaire de recherche.*

Il est commode de disposer de la même notion pour les arborescences ordonnées (un arbre binaire est un arbre positionné au sens du chapitre 4). Dans une *arborescence ordonnée de recherche*, chaque nœud x, à $d(x)$ fils, est muni de $d(x)-1$ clés $c_1(x), \ldots, c_{d(x)-1}(x)$, vérifiant la propriété suivante : soient $A_1(x), \ldots, A_{d(x)}(x)$ les sous-arbres de x, et soit y_i un sommet de $A_i(x)$ pour $1 \leq i \leq d(x)$; alors

$$c(y_1) < c_1(x) < c(y_2) < \cdots < c_{d(x)-1}(x) < c(y_{d(x)}).$$

Nous regroupons les notions d'arbre binaire de recherche et d'arborescence ordonnée de recherche sous le nom d'*arbre de recherche*. On rencontre en fait deux

catégories d'arbres de recherche, ceux où l'information est rangée aux sommets, et ceux où elle n'est rangée qu'aux feuilles. Comme nous l'avons dit plus haut, chaque clé est représentative d'une «donnée» en général plus volumineuse et plus lourde à manipuler que la clé elle-même. Chaque clé permet un accès, direct ou indirect, à cette information, et cet accès peut être rangé, avec la clé correspondante, à tout sommet de l'arbre, ou aux feuilles seulement. Dans le deuxième cas, les nœuds aussi contiennent des éléments de l'ensemble U des clés, mais elles ne servent qu'à organiser l'arbre. Nous les appelons des *balises*. Elles permettent de naviguer dans l'arbre, et en particulier elles pilotent la descente dans l'arbre lors d'une recherche. Chaque nœud contient une balise qui est supérieure ou égale aux clés de son sous-arbre gauche, et inférieure aux clés de son sous-arbre droit (en d'autres termes, les clés *et* les balises sont croissantes en parcours symétrique). Les arbres a–b sont des arbres balisés.

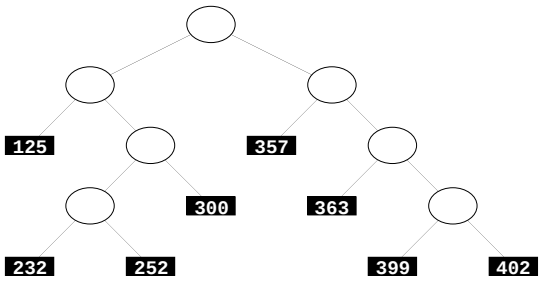


Figure 1.2: *Un arbre sans balises.*

La différence de structure entre un arbre binaire de recherche et un arbre binaire balisé de recherche, même si elle peut être importante dans certaines applications, est assez faible. Ainsi, pour passer d'un arbre binaire balisé à un arbre binaire de recherche, on peut procéder en deux étapes comme suit : dans un premier temps, on *rebalise* l'arbre, en choisissant comme balise d'un nœud la plus grande clé de son sous-arbre gauche.

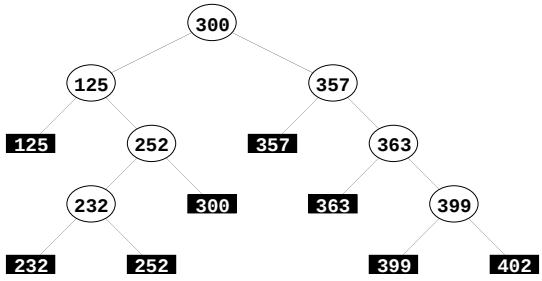


Figure 1.3: *L'arbre avec balises.*

Ensuite, on «oublie» les feuilles, et on insère dans l'arbre obtenu la plus grande clé de l'arbre d'origine. Cet algorithme s'implémente assez facilement en temps linéaire avec une file (voir exercices).

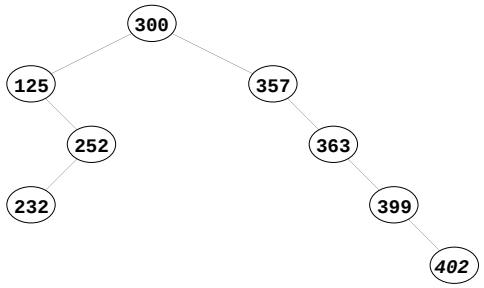


Figure 1.4: *L'arbre après insertion de la dernière clé.*

L'opération réciproque, de passage d'un arbre binaire de recherche à un arbre balisé, peut se faire en ajoutant d'abord des feuilles pour compléter l'arbre (au sens du chapitre 4), puis en munissant ces feuilles des clés appropriées : une feuille reçoit la clé du nœud pour lequel elle est la feuille la plus à droite dans le sous-arbre gauche. Il n'est pas difficile de réaliser cette opération avec une pile :

```
procédure BALISER(a);
   si EST-FEUILLE(a) alors
      FIXERCLÉ(a, SOMMET(P));
      DÉPILER(P)
   sinon
      EMPILER(CLÉ(a), P);
      BALISER(G(a));
      BALISER(D(a))
   finsi.
```

Les opérations de dictionnaire se réalisent, sur un arbre binaire balisé, de manière tout à fait semblable à celle exposée pour les arbres binaires de recherche dans le chapitre 3.

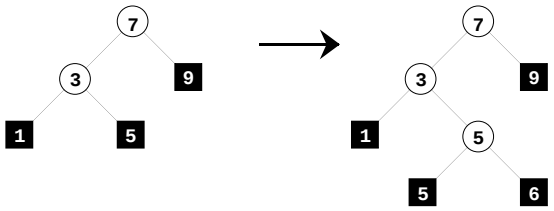


Figure 1.5: *Insertion de* 6 *dans l'arbre balisé.*

Ainsi, l'*insertion* d'une clé se fait comme dans un arbre binaire de recherche, sauf que l'on remplace la feuille découverte en descendant dans l'arbre par un nœud

qui devient le père de cette feuille et d'une nouvelle feuille contenant la clé (voir figure 1.5).

La *suppression* d'une clé est plus simple que dans un arbre binaire de recherche ordinaire; elle se fait en supprimant la feuille contenant la clé. Le père de cette feuille est remplacé par son autre fils (voir figure 1.6).

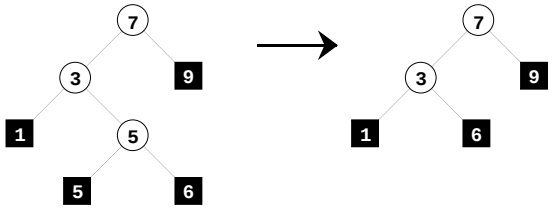


Figure 1.6: *Suppression de* 5 *dans l'arbre balisé.*

6.1.2 Rotations

Les opérations de rotation et de double rotation que nous introduisons maintenant sont intéressantes pour tous les arbres binaires. Elles servent notamment à rééquilibrer les arbres après une insertion ou une suppression. Soit A un arbre binaire non vide. Il est commode d'écrire $A = (x, B, C)$ pour exprimer que x est la racine de A et que B et C sont ses sous-arbres gauche et droit (voir figure 1.7). Nous utilisons cette notation dans la suite de cette section.

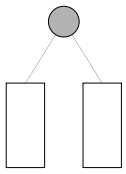


Figure 1.7: *L'arbre* $A = (x, B, C)$.

Soit donc $A = (x, X, B)$ un arbre non vide, supposons B non vide et posons $B = (y, Y, Z)$. La *rotation gauche* de A est l'opération :

$$A = (x, X, (y, Y, Z)) \quad \mapsto \quad \mathcal{G}(A) = (y, (x, X, Y), Z)$$

représentée dans la figure 1.8. La *rotation droite* est l'opération inverse :

$$A = (y, (x, X, Y), Z) \quad \mapsto \quad \mathcal{D}(A) = (x, X, (y, Y, Z))$$

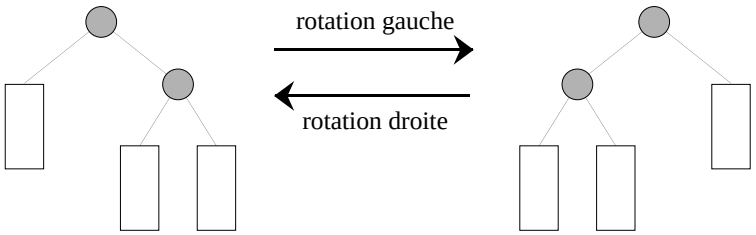


Figure 1.8: *Rotations gauche et droite.*

Proposition 1.1. *Si A est un arbre binaire de recherche, et si la rotation gauche (respectivement droite) est définie sur A, alors $\mathcal{G}(A)$ (respectivement $\mathcal{D}(A)$) est encore un arbre binaire de recherche.* ∎

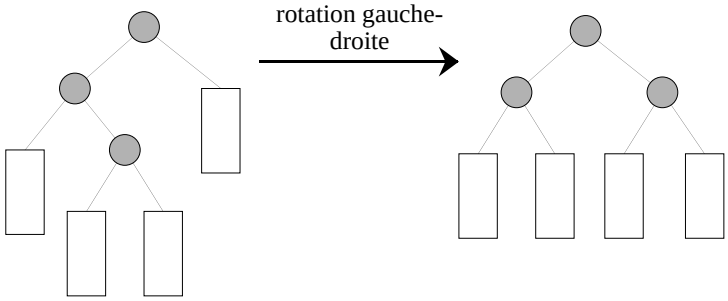


Figure 1.9: *Rotation gauche-droite.*

Deux *doubles rotations* sont utilisées, la rotation *gauche-droite* et la rotation *droite-gauche* : elles associent, à un arbre $A = (x, A_g, A_d)$ donné, respectivement l'arbre $\mathcal{D}(x, \mathcal{G}(A_g), A_d)$ et l'arbre $\mathcal{G}(x, A_g, \mathcal{D}(A_d))$ (voir figures 1.9 et 1.10). Ces opérations ne sont évidemment définies que si les sous-arbres requis ne sont pas vides. Ces opérations préservent bien sûr également les arbres binaires de recherche.

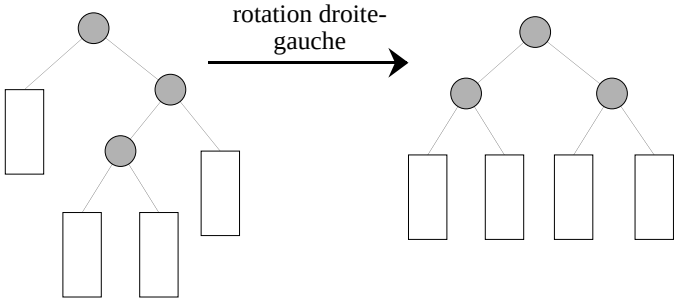


Figure 1.10: *Rotation droite-gauche.*

Un aspect essentiel de ces opérations est qu'elles s'implémentent en temps constant. Lorsque les arbres binaires sont déclarés par

```
arbre = ^sommet;
sommet = RECORD
            val: element;
            g, d: arbre
         END;
```

on obtient la procédure suivante qui réalise la rotation gauche, par un simple échange de pointeurs :

```
PROCEDURE rotationgauche (VAR a: arbre);
   VAR
      b: arbre;
   BEGIN
      b := a^.d; a^.d := b^.g; b^.g := a; a := b
   END;
```

Proposition 1.2. *Les opérations de rotation et de double rotation sur les arbres binaires se réalisent en temps constant.* ∎

6.2 Arbres *AVL*

6.2.1 Définition

Les arbres *AVL*, introduits par Adelson-Velskii et Landis en 1962, constituent une famille d'arbres binaires de recherche équilibrés en hauteur.

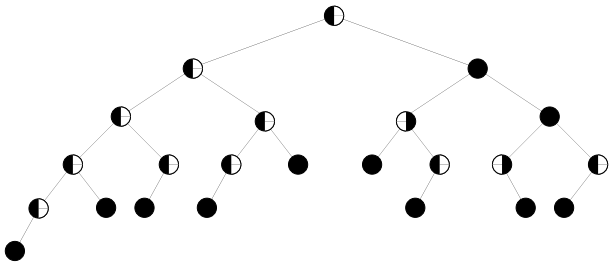


Figure 2.1: *Un arbre AVL.*

Un arbre binaire A est un *arbre AVL* si, pour tout sommet de l'arbre, les hauteurs des sous-arbres gauche et droit diffèrent d'au plus 1. Plus précisément, posons $\delta(A) = 0$ si A est l'arbre vide, et sinon

$$\delta(A) = hauteur(A_g) - hauteur(A_d)$$

où A_g et A_d sont les sous-arbres gauche et droit de A. Le nombre $\delta(A)$ est appelé l'*équilibre* de A (ou de sa racine). Alors A est un arbre *AVL* si pour tout sommet

x, on a $\delta(A(x)) \in \{-1, 0, 1\}$. Par abus, on écrira aussi $\delta(x)$. En particulier, un arbre réduit à un seul sommet a un équilibre nul.

Les arbres *AVL* ont une hauteur logarithmique en fonction du nombre de sommets. Plus précisément :

Proposition 2.1. *Soit A un arbre AVL ayant n sommets et de hauteur h. Alors*

$$\log_2(1+n) \leq 1 + h \leq 1,44 \log_2(2+n).$$

Preuve. Pour une hauteur h donnée, l'arbre ayant le plus de sommets est l'arbre complet qui a $2^{h+1}-1$ sommets. Donc $n \leq 2^{h+1}-1$, et par conséquent $\log(1+n) \leq 1+h$. Pour l'autre inégalité, soit $N(h)$ le minimum des nombres de sommets des arbres *AVL* de hauteur h. On a $N(-1) = 0$, $N(0) = 1$, $N(1) = 2$, et

$$N(h) = 1 + N(h-1) + N(h-2), \qquad h \geq 2$$

car pour obtenir le minimum, on choisit l'un des sous-arbres minimum de hauteur $h-1$, et l'autre minimum de hauteur $h-2$. Si l'on pose $F(h) = N(h) + 1$, on a $F(0) = 2$, $F(1) = 3$, et

$$F(h) = F(h-1) + F(h-2), \qquad h \geq 2$$

donc $N(h) = F_{h+3}$, où F_n est le n-ième nombre de Fibonacci (voir chapitre 2). Pour tout arbre *AVL* à n sommets et de hauteur h, on a par conséquent

$$n + 1 \geq F(h) = \frac{1}{\sqrt{5}} \left(\phi^{h+3} - \bar{\phi}^{h+3}\right) > \frac{1}{\sqrt{5}} \phi^{h+3} - 1$$

d'où

$$h + 3 < \frac{\log_2(n+2)}{\log_2 \phi} + \log_\phi \sqrt{5} \leq 1,44 \log_2(n+2) + 2$$

parce que $1/\log_2 \phi \leq 1,44$ et $\log_\phi \sqrt{5} \leq 2$. ∎

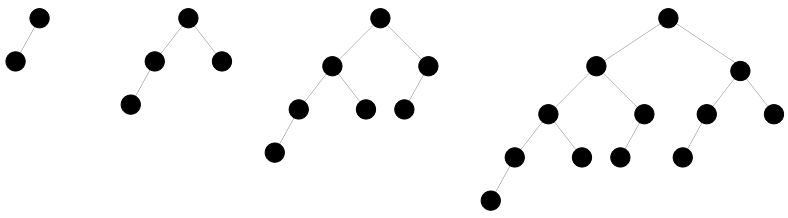


Figure 2.2: *Arbres de Fibonacci Φ_k pour $k = 2, 3, 4, 5$.*

La borne de la proposition est essentiellement atteinte par les *arbres de Fibonacci* qui sont définis comme suit : l'arbre Φ_0 est l'arbre vide, l'arbre Φ_1 est réduit à un seul sommet; l'arbre Φ_{k+2} a un sous-arbre gauche égal à Φ_{k+1}, et un sous-arbre droit égal à Φ_k. La hauteur de Φ_k est $k-1$, et Φ_k a $F_{k+2} - 1$ sommets.

L'*implémentation* des arbres *AVL* se fait avantageusement comme celle des arbres binaires de recherche (voir chapitre 3). Pour rendre efficaces les opérations de rééquilibrage dont il sera question plus loin, il convient d'ajouter un champ supplémentaire à chaque sommet qui contient la hauteur du sous-arbre. Pour un arbre binaire de recherche *AVL*, les types s'écrivent donc

```
arbre = ^sommet;
sommet = RECORD
            haut: integer;
            val: element;
            g, d: arbre
         END;
```

Certains auteurs recommandent de ne pas conserver la hauteur, mais seulement l'équilibre en chaque sommet. Comme l'équilibre peut se coder sur deux bits, il en résulte un gain de place. En revanche, les algorithmes sont plus difficiles à mettre en œuvre. Le gain de place est en fait réduit en codant la hauteur sur un octet. On peut alors représenter des arbres de hauteur 255, donc avec environ 2^{170} sommets, ce qui suffit en pratique.

6.2.2 Insertion

Les arbres *AVL* sont utilisés comme implémentation de dictionnaires. On considère donc des arbres binaires de recherche qui sont en plus des arbres *AVL*. Les opérations de recherche, d'insertion et de suppression se font, dans un arbre binaire de recherche, en temps proportionnel à la hauteur de l'arbre. Cette hauteur est logarithmique en fonction du nombre de sommets dans un arbre *AVL*. Ainsi, la recherche se fait en temps logarithmique. Pour l'insertion et la suppression, il convient de maintenir le caractère *AVL* après l'opération, ce qui se traduit par un rééquilibrage. Comme on verra, le coût du rééquilibrage est, lui aussi, proportionnel à la hauteur, ce qui garantit un coût total logarithmique.

Soit donc à insérer une clé c dans un arbre binaire de recherche *AVL* A. Si A est vide, le résultat est un arbre A' formé d'un seul sommet de contenu c, et cet arbre est *AVL*. Supposons donc A non vide.

Dans une *première phase*, on utilise l'algorithme d'insertion habituel (voir chapitre 3) dans un arbre binaire de recherche : on descend dans A à partir de sa racine r, et on crée une nouvelle feuille s contenant la clé c. Soit A' l'arbre ainsi obtenu.

Dans une *deuxième phase*, on remonte le chemin γ entre r et s, en progressant de s vers r, et on teste si l'arbre A' est encore *AVL*. Seuls les sous-arbres dont les racines sont sur le chemin γ peuvent changer de hauteur; cette hauteur ne peut qu'augmenter, et elle ne peut augmenter que de 1.

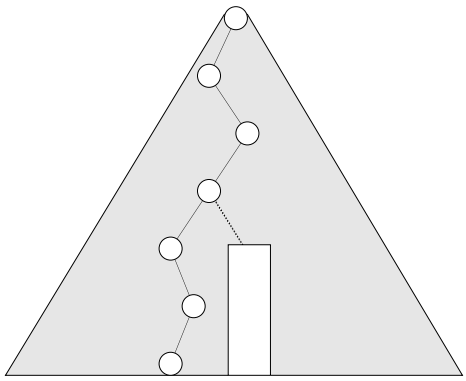


Figure 2.3: *Le chemin γ de rééquilibrage.*

Si les racines de ces sous-arbres vérifient les conditions d'équilibre des arbres *AVL*, l'arbre A' tout entier est *AVL*. Sinon, il existe un premier sommet x (en partant de s) sur le chemin γ dont l'équilibre, dans A' vaut 2 ou -2. Supposons, pour fixer les idées, que le fils gauche u de x est aussi sur γ (voir figure 2.3) ; alors l'équilibre de x est 2. Posons $h = hauteur(A(x))$. Alors

$$h - 1 = hauteur(A(u)), \quad h = hauteur(A'(u)), \quad h - 2 = hauteur(A_d(x)).$$

L'arbre $A_d(x)$ peut être vide, voir figure 2.4.

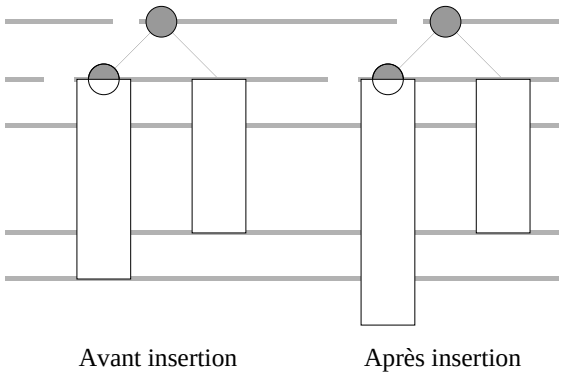


Figure 2.4: *L'arbre de racine x avant et après insertion.*

Posons $A'(u) = (u, X, Y)$ et $Z = A_d(x)$. Les arbres $A'(u)$ et Z sont *AVL*.

Deux cas se présentent alors :

(1) L'insertion s'est faite dans le sous-arbre *gauche* de u. On a alors $hauteur(X) = h - 1$ et $hauteur(Y) = h - 2$ (car $A'(y)$ est *AVL*). Dans ce cas, on effectue une rotation droite de $A'(x)$, ce qui donne l'arbre $A'' = (u, X, (x, Y, Z))$, voir figure 2.5. L'arbre A'' est un arbre *AVL*, et la hauteur de A'' est égale à la hauteur de $A(x)$, c'est-à-dire de x dans l'arbre avant insertion. Il en résulte qu'après cette rotation, l'arbre tout entier est *AVL*, et que la phase de rééquilibrage est terminée.

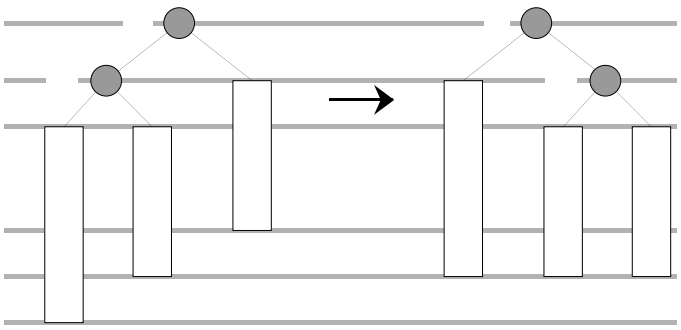


Figure 2.5: *Cas* (1) *: une rotation simple rétablit l'équilibre.*

(2) L'insertion s'est faite dans le sous-arbre *droit* de u. On a alors $hauteur(X) = h-2$ et $hauteur(Y) = h-1$. Posons $Y = (v, V, W)$. L'un des deux arbres V ou W est de hauteur $h-2$, et l'autre a pour hauteur $h-3$. On effectue alors une rotation gauche-droite de $A'(x)$, ce qui donne l'arbre $A'' = (v, (u, X, V), (x, W, Z))$, voir figure 2.6. Comme dans le premier cas, l'arbre A'' est un arbre *AVL* de même hauteur que $A(x)$. Le rééquilibrage s'arrête donc après cette double rotation.

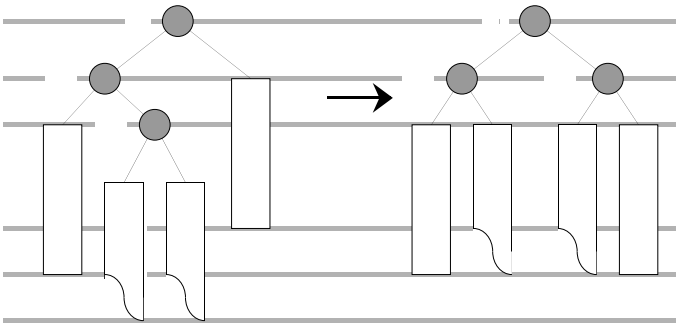


Figure 2.6: *Cas* (2) *: une rotation double rétablit l'équilibre.*

Proposition 2.2. *L'insertion dans un arbre AVL à n sommets se réalise en temps $O(\log n)$. Il suffit d'au plus une rotation ou double rotation pour rééquilibrer l'arbre après insertion.*

Preuve. Il reste à constater que les comparaisons de hauteurs se font, pour chaque sommet du chemin de remontée, en temps constant, ce qui est évident. ∎

6.2.3 Suppression

Considérons maintenant la suppression d'une clé c dans un arbre binaire de recherche *AVL* A. Si A n'a qu'un seul sommet, de contenu c, le résultat est l'arbre vide. On suppose donc que A a au moins deux sommets.

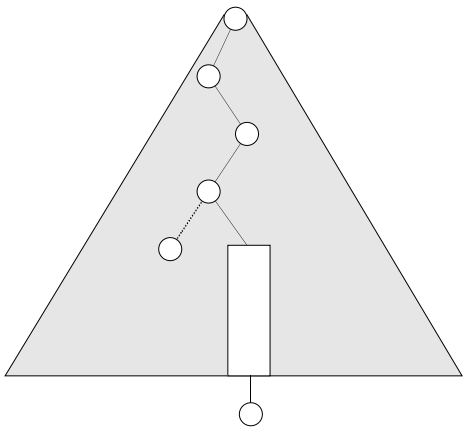


Figure 2.7: *Le chemin de rééquilibrage.*

Dans une *première phase*, on utilise l'algorithme habituel de suppression dans un arbre binaire de recherche (voir chapitre 3) : on descend dans A à partir de la racine r à la recherche du sommet t contenant la clé c; si t est une feuille, on la supprime, sinon on remplace le contenu de t par le contenu de la feuille la plus à droite du sous-arbre gauche de t (respectivement de la feuille la plus à gauche du sous-arbre droit de t), et on supprime cette feuille. Dans tous les cas, on supprime donc une feuille dans A. Notons-la s, soit A' l'arbre après suppression, et soit γ le chemin de r à s.

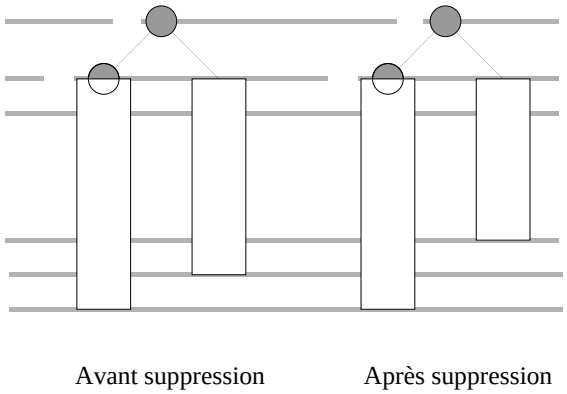


Figure 2.8: *L'arbre de racine x avant et après suppression.*

Dans une *deuxième phase*, on remonte le chemin γ de s vers r et on teste si l'arbre A' est encore *AVL*. Seuls les sous-arbres dont les racines sont sur le chemin γ peuvent changer de hauteur, et cette hauteur peut diminuer de 1. Si les racines de ces sous-arbres vérifient la condition d'équilibre, l'arbre A' est *AVL*. Sinon, il existe un premier sommet x (en partant du père de s) dont l'équilibre vaut 2 ou -2. Supposons, pour fixer les idées, que le fils gauche u de x n'*est pas* sur le chemin γ (voir figure 2.7), donc que la suppression s'est faite dans le sous-arbre droit de x. Après suppression, ce sous-arbre est transformé en un arbre *AVL* éventuellement

vide que nous notons Z. L'équilibre de x est 2. Posons $h = hauteur(A(x))$. Alors (voir figure 2.8)

$$h-1 = hauteur(A(u)), \quad h-2 = hauteur(A_d(x)), \quad h-3 = hauteur(Z).$$

Posons $A(u) = (u, X, Y)$. Deux cas se présentent alors :

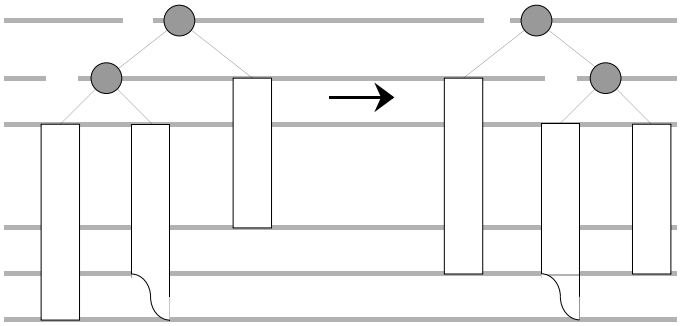


Figure 2.9: *Une rotation simple sur le chemin de rééquilibrage.*

(1) La hauteur de X est $h-2$. Dans ce cas, la hauteur de Y est $h-2$ ou $h-3$. On effectue une rotation droite de $A'(x)$, ce qui donne l'arbre $A'' = (u, X, (x, Y, Z))$ qui est *AVL*, voir figure 2.9. L'arbre A'' a hauteur h ou $h-1$, selon que Y a hauteur $h-2$ ou $h-3$. Dans le premier cas, A'' a donc la même hauteur que $A(x)$, et le processus de rééquilibrage s'arrête; dans le deuxième cas, il faut continuer en remontant vers la racine.

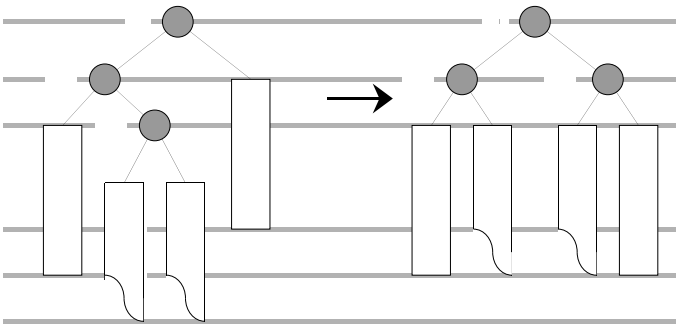


Figure 2.10: *Une rotation double sur le chemin de rééquilibrage.*

(2) La hauteur de X est $h-3$. Alors la hauteur de Y est $h-2$, parce que $A(u)$ est un arbre *AVL*. Posons $Y = (v, V, W)$. On effectue une rotation gauche-droite donnant l'arbre $A'' = (v, (u, X, V), (X, W, Z))$, voir figure 2.10. Cet arbre est *AVL*, mais sa hauteur est $h-1$. En d'autres termes, le processus de rééquilibrage doit continuer, en remontant vers la racine.

Proposition 2.3. *La suppression dans un arbre AVL à n sommets se réalise en temps $O(\log n)$.* ∎

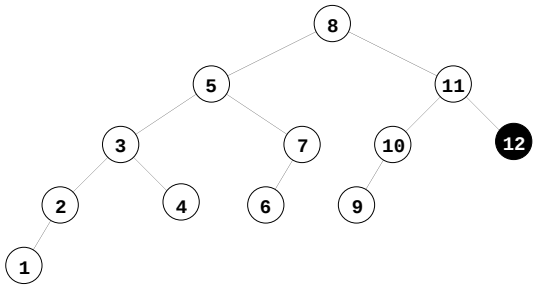


Figure 2.11: *Arbre de Fibonacci avant la suppression de la clé* 12.

Exemple. Considérons l'arbre (de Fibonacci) de la figure 2.11. La suppression de la clé 12 amène une première rotation autour du sommet de clé 11, puis une deuxième autour de la racine (figure 2.12).

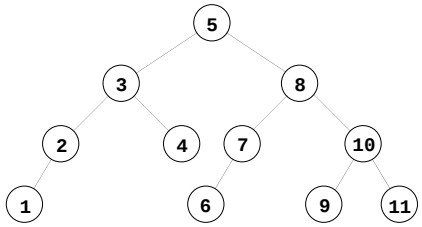


Figure 2.12: *Après suppression et rééquilibrage.*

6.2.4 Arbres balisés

Comme nous l'avons déjà dit dans l'introduction, il existe deux variantes des arbres binaires de recherche : ceux où tous les sommets contiennent des clés (et les informations associées), et ceux où seules les feuilles contiennent des clés. Les nœuds de ces arbres contiennent alors des *balises* (qui sont en général aussi des éléments de l'ensemble des clés) permettant de naviguer dans l'arbre.

Les arbres *AVL* balisés nous seront utiles plus loin. Chaque feuille d'un tel arbre contient donc une clé, et les clés sont croissantes de la gauche vers la droite. Nous avons déjà décrit comment on réalise l'insertion et la suppression dans un arbre balisé. Il n'est pas difficile de se convaincre que le rééquilibrage par rotations se transpose sans grand changement aux arbres balisés, puisque seuls les nœuds changent de fils. L'analyse du rééquilibrage dans les arbres *AVL* se transpose également dans les arbres balisés. On peut donc, au choix, utiliser des arbres *AVL* ou des arbres *AVL* balisés.

6.3 Arbres a–b

Les arbres a–b sont des arbres dont toutes les feuilles ont même profondeur, et le nombre de fils d'un nœud varie entre a et b. Nous montrons ici comment réaliser les opérations de recherche, d'insertion, de suppression, de scission et de concaténation en temps logarithmique en fonction du nombre de sommets de l'arbre. Les rééquilibrages sont en fait assez rares, puisqu'en moyenne deux opérations de rééquibrage insertion et suppression suffisent dans le cas des arbres 2–4. Les arbres a–b constituent ainsi une implémentation efficace des listes concaténables.

6.3.1 Définition

Soient a et b deux entiers, avec $a \geq 2$, et $b \geq 2a - 1$. Un *arbre a–b* est un arbre A vérifiant les conditions suivantes :

(i) les feuilles ont toutes la même profondeur ;
(ii) la racine a au moins 2 et au plus b fils ;
(iii) les autres nœuds ont au moins a et au plus b fils.

On note $d(x)$ le nombre de fils d'un nœud x, et $A_i(x)$ le i-ième sous-arbre de x, pour $i = 1, \ldots, d(x)$.

Soit S un ensemble de clés. Un arbre A est un *arbre a–b pour S* si les éléments de S sont rangés aux feuilles de A en ordre croissant de la gauche vers la droite, et si de plus, chaque nœud x de A contient une suite de $d(x)-1$ clés $k_1 < \cdots < k_{d(x)-1}$, appelées les *balises* de x et vérifiant les conditions :

- les clés des feuilles de $A_i(x)$ sont inférieures ou égales à k_i, pour $i = 1, \ldots, d(x) - 1$;
- les clés des feuilles de $A_i(x)$ sont strictement supérieures à k_{i-1}, pour $i = 2, \ldots, d(x)$.

En d'autres termes, si c_i est une clé d'une feuille de $A_i(x)$, on a

$$c_1 \leq k_1 < c_2 \leq k_2 < \cdots \leq k_{d(x)-1} < c_{d(x)}.$$

Il est commode de noter $k_i(x)$ la i-ième balise du nœud x. Lorsque $b = 2a - 1$, un arbre a–b est appelé un B-arbre d'ordre $a - 1$. L'arbre vide est un arbre a–b, de même que l'arbre formé d'un seul sommet qui est une feuille.

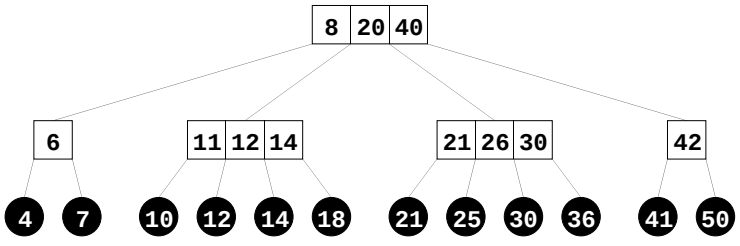


Figure 3.1: *Un arbre 2–4.*

Proposition 3.1. *Soit A un arbre a–b avec n feuilles ($n > 0$) de hauteur h. Alors*

$$2a^{h-1} \leq n \leq b^h$$

ou encore

$$\log n / \log b \leq h \leq 1 + \log(n/2) / \log a.$$

Preuve. Tout nœud a au plus b fils, donc A a au plus b^h feuilles. Tout nœud autre que la racine a au moins a fils, et la racine au moins 2; au total, il y a au moins $2a^{h-1}$ feuilles. Le deuxième encadrement en découle. ∎

6.3.2 Recherche d'un élément

Soit A un arbre a–b pour un ensemble S. La *recherche* d'une clé c dans A est dite *positive* si $c \in S$, elle est *négative* sinon. Si A est vide, la recherche est négative; dans les autres cas, elle se fait comme suit :

```
RECHERCHER (c, A);
  x =RACINE(A);
  tantque x est un nœud faire
    i := 1;
    tantque c > k_i(x) etalors i < d(x) faire i := i + 1;
    poser x := le i-ième fils de x
  fintantque;
  si c = cle(x) alors retourner(recherche positive)
  sinon retourner(recherche negative).
```

La recherche du sous-arbre $A_i(x)$ susceptible de contenir la clé c se fait en comparant c aux balises $k_i(x)$. Ceci se réalise en temps constant, c'est-à-dire indépendant de la taille de A. On a donc

Proposition 3.2. *Soit A un arbre a–b pour un ensemble S à n éléments ($n > 0$). La recherche d'une clé dans A se fait en temps $O(\log n)$.* ∎

6.3.3 Insertion d'un élément

L'*insertion* d'une clé c dans un arbre a–b pose, comme nous allons le voir, le problème du rééquilibrage. Il se peut en effet qu'après insertion, un nœud de l'arbre ait $b + 1$ fils. Dans ce cas, on *éclate* ce nœud en deux nœuds frères qui se partagent ces $b + 1$ fils. Il se peut que leur père, à son tour, ait trop de fils; on

répète alors cette procédure. Plus précisément, l'insertion se fait en deux phases. On suppose que A a au moins 2 sommets, les autres cas étant faciles.

Dans la *première* phase, on descend dans l'arbre à partir de la racine à la recherche d'une feuille y susceptible de contenir la clé c. Si la recherche est positive, la clé c figure déjà dans A et il n'y a pas lieu de l'ajouter ; sinon, soit x le père de y. Soient $c_1, \ldots, c_d$ les clés des fils de x, avec $d = d(x)$, et soient $k_1, \ldots, k_{d-1}$ les balises du nœud x. On a donc

$$c_1 \leq k_1 < c_2 \leq \cdots \leq k_{d-1} < c_d.$$

La recherche de c parmi les fils de x détermine un indice i tel que $k_{i-1} < c \leq k_i$ (si $i = 1$, on a seulement $c \leq k_1$, et si $i = d$, seulement $k_{d-1} < c$).

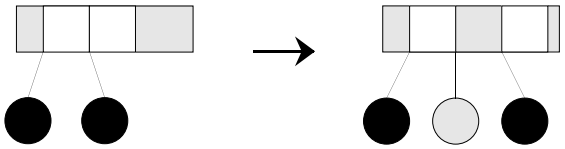


Figure 3.2: *Insertion d'une feuille, cas $c < c_i$.*

On crée alors une feuille de clé c, et on l'insère comme i-ième fils de x si $c < c_i$, et comme $i+1$-ième fils de x si $c_i < c$. On insère également la clé $\min(c, c_i)$ comme i-ième balise au nœud x. Les figures 3.2 et 3.3 décrivent les deux transformations.

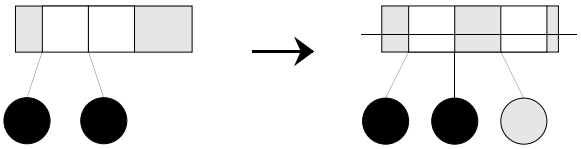


Figure 3.3: *Insertion d'une feuille, cas $c > c_i$.*

Dans la *deuxième* phase, on rééquilibre l'arbre si c'est nécessaire. Si le nœud x a au plus b fils après insertion, aucun rééquilibrage n'est nécessaire, et l'algorithme est terminé. Sinon, $d(x) = b + 1$, et on éclate le nœud x.

L'*éclatement* de x est l'opération suivante : on crée deux nœuds frères x' et x'', et on partage les $b+1$ fils de x en deux groupes, respectivement de $\lfloor (b+1)/2 \rfloor$ et $\lceil (b+1)/2 \rceil$ éléments, le premier fournissant les fils de x', le second les fils de x''. Comme $b \geq 2a - 1$, on a $\lfloor (b+1)/2 \rfloor \geq a$, donc x' et x'' vérifient les contraintes d'un arbre a–b. Si x a un père, soit z, les frères x' et x'' sont déclarés fils de z à la place de x. Sinon, on crée un nouveau nœud z (qui devient la nouvelle racine de l'arbre) dont x' et x'' sont les seuls fils. C'est de cette manière que l'arbre prend de la hauteur.

Les b balises $k_1(x), \ldots, k_b(x)$ du nœud x sont réparties comme suit : les balises d'indice 1 à $\lfloor (b+1)/2 \rfloor - 1$ deviennent les balises de x', les balises d'indice $\lfloor (b+1)/2 \rfloor + 1$ à b deviennent les balises de x'' ; quant à la balise dont l'indice est

$\lfloor (1+b)/2 \rfloor$, elle devient la balise qui, dans le père commun z de x' et x'', sépare x' de x''.

L'éclatement peut se représenter de manière schématique par la règle suivante :

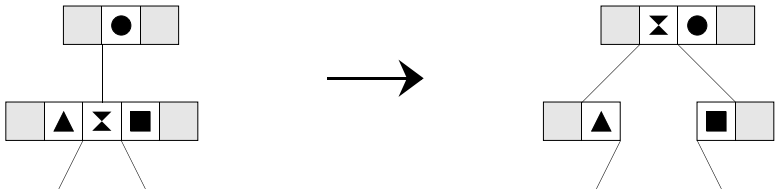


Figure 3.4: *Règle d'éclatement.*

Revenons à l'insertion. L'éclatement d'un nœud x augmente le nombre de fils de son père. Si nécessaire, on répète l'éclatement sur le père, puis sur le père de celui-ci, etc. A la fin, l'arbre est à nouveau a–b.

Proposition 3.3. *Soit A un arbre a–b pour un ensemble S à n éléments ($n > 0$). L'insertion d'une clé dans A se fait en temps $O(\log n)$.*

Preuve. La localisation de l'endroit où insérer la clé prend un temps proportionnel à la hauteur h de A. L'insertion de la feuille est suivie d'au plus h éclatements de nœuds. Chaque éclatement prend un temps constant. ∎

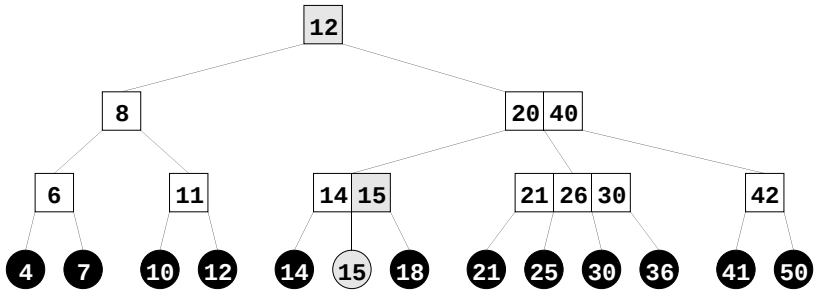


Figure 3.5: *L'arbre après insertion de* 15.

Exemple. L'insertion de la clé 15 dans l'arbre de la figure 3.1 provoque d'abord l'éclatement du deuxième fils de la racine, puis de la racine elle-même. L'arbre entier prend donc la forme donnée dans la figure 3.5.

6.3.4 Suppression d'un élément

La *suppression* d'une clé dans un arbre a–b est plus compliquée. Le rééquilibrage fait appel à l'opération inverse de l'éclatement, la *fusion*. Il est utile d'introduire

une opération supplémentaire, appelée *partage* qui, même si elle n'est pas indispensable, facilite la suppression.

Soient x' et x'' deux nœuds frères d'un arbre a–b, et soit z leur père. On suppose de plus que x' et x'' sont des fils consécutifs de z, disons x' à gauche de x''. Si le nombre total de fils de x' et x'' est au plus b, on peut *fusionner* x' et x'' en un seul nœud x, fils de z, qui prend comme fils l'ensemble des fils de x' et x''. Schématiquement, la fusion, opération inverse de l'éclatement, peut se représenter par la règle suivante :

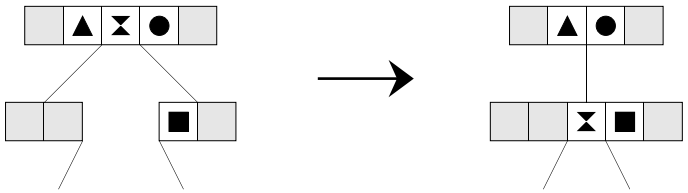


Figure 3.6: *Règle de fusion.*

Pour la deuxième opération, le *partage*, considérons à nouveau deux frères adjacents x' et x'', fils d'un nœud z, avec disons x' à gauche de x''. Si x' a moins de b fils, et x'' a plus de a fils, le partage consiste à transférer le sous-arbre le plus à gauche de x'', soir $A_1(x'')$, vers le nœud x', qui le reçoit donc comme sous-arbre le plus à droite : $A_{1+d(x')}(x') := A_1(x'')$. Ce transfert se schématise comme suit :

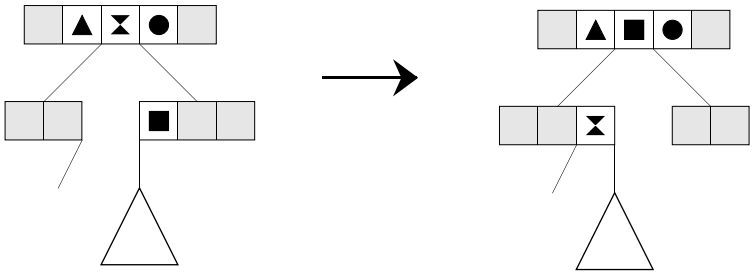


Figure 3.7: *Règle du partage.*

L'algorithme de *suppression* d'une clé c dans un arbre a–b A est le suivant : on cherche la feuille contenant la clé c. Soit x le père de cette feuille; on supprime la feuille et la clé, dans x, de même indice (ou d'indice $d(x) - 1$ si la feuille est le fils le plus à droite de x).

Considérons d'abord le cas où x est la racine de l'arbre. Si x a plus de deux fils, l'arbre est un arbre a–b, sinon x a un fils unique; on supprime alors x et son fils unique devient la racine de l'arbre (sa hauteur diminue donc de 1).

Si x n'est pas la racine, et si x a au moins a fils après cette suppression, l'arbre résultant est encore un arbre a–b et l'opération est terminée. Sinon, x a $a-1$ fils, et il faut rééquilibrer l'arbre. Soit z le père de x.

Si x a un frère gauche ou un frère droit y qui a exactement a fils, on fusionne x et y en un fils de z qui a $2a - 1 \leq b$ fils.

Si x a un frère gauche ou droit y qui a plus de a fils, on fait un partage entre x et y : le nœud x reçoit un fils supplémentaire, donc en a maintenant a, et y en perd un, mais en a encore au moins a, et z garde les mêmes fils : l'arbre est a–b.

Dans le cas d'une fusion, le nœud z perd un fils ; il se peut qu'il n'ait plus que $a-1$ fils. Il convient alors de continuer le rééquilibrage. En revanche, une opération de partage met fin au rééquilibrage. En résumé, les cas suivants se présentent, en fonction du nombre de fils des frères gauche et droit du nœud x (en supposant bien sûr que x ait deux frères adjacents) :

(i) les deux frères ont a fils : seule une fusion est possible ;
(ii) l'un des deux frères a a fils, l'autre en a plus de a : un partage et une fusion sont possibles ;
(iii) les deux frères ont plus de a fils : un partage est possible, et une fusion aussi, si le nombre total de fils ne dépasse pas b.

Dans certaines situations, on peut donc choisir entre une fusion ou un partage. Comme un partage arrête le rééquilibrage, on a intérêt à privilégier les partages.

Un partage s'apparente à une fusion suivie d'un éclatement (modifié). En effet, si un partage est possible, une fusion donne un nœud qui a assez de fils pour être éclaté, et on obtient le résultat du partage en éclatant le sommet, mais avec une répartition des fils qui n'est pas aussi équilibrée que celle décrite plus haut. On peut donc remplacer un partage par une fusion suivie d'un éclatement, et c'est en ce sens que le partage n'est pas indispensable au rééquilibrage.

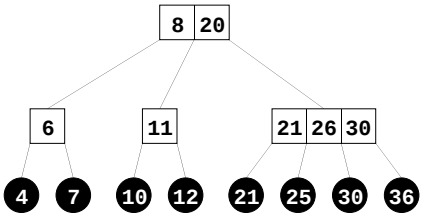


Figure 3.8: *Un arbre 2–4.*

Exemple. Considérons l'arbre de la figure 3.8. Après suppression de 12, l'arbre n'est plus équilibré (figure 3.9).

Le nœud sans balise peut être rééquilibré par un partage avec son frère droit. On obtient l'arbre de la figure 3.10.

Il peut aussi être rééquilibré par fusion avec son frère gauche. On obtient alors l'arbre de la figure 3.11.

Proposition 3.4. *Soit A un arbre a–b pour un ensemble S à n éléments ($n > 0$). La suppression d'une clé dans A se fait en temps $O(\log n)$.* ∎

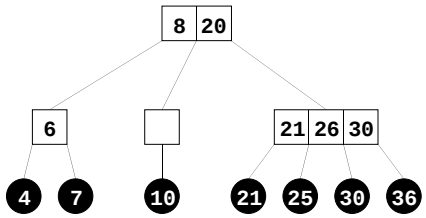


Figure 3.9: *Après suppression de* 12 *et avant rééquilibrage.*

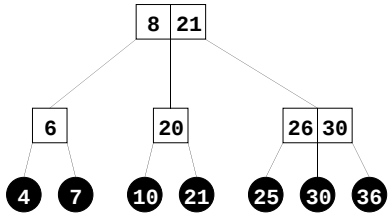


Figure 3.10: *Après partage avec le frère droit.*

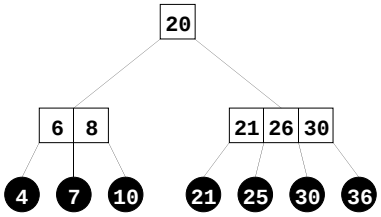


Figure 3.11: *Après fusion avec le frère gauche.*

D'autres opérations simples sur les ensembles de clés peuvent être considérées, comme par exemple :

MIN(S) ;
: détermine le plus petit élément de S ;

SUPPRIMER-MIN(S) ;
: supprime le plus petit élément de S.

La première opération se réalise en descendant dans la branche gauche de l'arbre, et pour la deuxième, il suffit d'appliquer ensuite l'opération de suppression. On traite de manière analogue MAX(S) et SUPPRIMER-MAX(S).

Théorème 3.5. *Si un ensemble S est représenté par un arbre a–b, les opérations* RECHERCHER(c, S), INSÉRER(c, S), SUPPRIMER(c, S), MIN(S), MAX(S), SUPPRIMER-MIN(S) *et* SUPPRIMER-MAX(S) *se réalisent toutes en temps* $O(\log(1 + |S|))$.

6.3.5 Concaténation et scission

On considère maintenant deux opérations supplémentaires très utiles sur les ensembles ordonnés qui se réalisent de façon efficace à l'aide d'arbres a–b. Ce sont :

CONCATÉNER(S_1, S_2, S) ;
est défini lorsque $\max(S_1) < \min(S_2)$, le résultat est $S = S_1 \cup S_2$.

SCINDER(S, c, S_1, S_2) ;
définit une décomposition $S = S_1 \cup S_2$, avec $S_1 = \{u \in S \mid u \leq c\}$ et $S_2 = \{u \in S \mid u > c\} = S - S_1$.

Proposition 3.6. *Soient S_1 et S_2 deux ensembles représentés par des arbres a–b. L'opération* CONCATÉNER(S_1, S_2, S) *peut se réaliser en temps $O(\log(1 + \max(|S_1|, |S_2|)))$. Pour un ensemble S représenté par un arbre a–b, l'opération* SCINDER(S, c, S_1, S_2) *peut se réaliser en temps $O(\log(1 + |S|))$.*

Preuve. Soit A_i un arbre a–b pour S_i, et soit h_i la hauteur de A_i, pour $i = 1, 2$. On suppose $h_1 \geq h_2$, l'autre cas se traite de la même manière.

Soit c une clé telle que $\max(S_1) \leq c < \min(S_2)$. On peut calculer une telle clé en temps $O(h_1)$, en calculant MAX(S_1) sur A_1. La concaténation se réalise comme suit.

Partant de la racine de A_1, on descend sa branche la plus à droite jusqu'au nœud x de profondeur $h_1 - h_2$. Ainsi, le sous-arbre $A_1(x)$ de racine x a la même hauteur que A_2. On fusionne alors ce sommet x et la racine r_2 de A_2 en un nœud unique disons z, en ajoutant la clé c=MAX(S_1) comme clé centrale dans la liste des balises du nœud z.

Si z a moins de b fils, c'est terminé; sinon, z a au plus $2b$ fils et un éclatement donne deux nœuds ayant chacun au plus b fils. Une remontée vers la racine permet de rééquilibrer l'arbre par des éclatements successifs.

Moyennant la connaissance de c, l'opération de concaténation se fait donc en temps $O(1+|h_1-h_2|)$. Pour déterminer c, un temps $O(\log(1+|S_1|))$ est suffisant. Dans la suite, la clé c sera disponible au moment de la concaténation.

Venons-en à la *scission*. Soit A un arbre a–b pour l'ensemble S, et soit c une clé; on descend dans l'arbre à la recherche de la clé c; soit y la feuille détectée. En cas de recherche positive, la clé de y est c; en cas de recherche négative, la clé de y est la plus grande clé dans S qui est inférieure à c (ou la plus petite clé de S supérieure à c). En d'autres termes, l'ensemble S_1 est formé des clés de toutes les feuilles à gauche de y, et S_2 est formé des clés des feuilles à droite de y. La clé de y appartient à S_1 ou S_2, selon les cas.

On considère maintenant le chemin menant de la racine à la feuille y. On supprime les arêtes de ce chemin et on éclate les sommets sur le chemin. Toutefois, l'éclatement est modifié comme suit : si l'arête supprimée dans un nœud x est la k-ième, le nœud x est éclaté en deux nœuds x' qui reçoit les $k-1$ premiers fils et

les $k-1$ premières balises, et x'' qui reçoit les autres fils et les balises de droite, y compris la k-ième. Chacun de ces nœuds a donc une balise supplémentaire qui servira ensuite, et chacun des arbres est a–b, sauf peut-être en sa racine. Enfin, si $k = 1$, seul le nœud x'' est créé, et de même si $k = d(x)$, seul x' est créé.

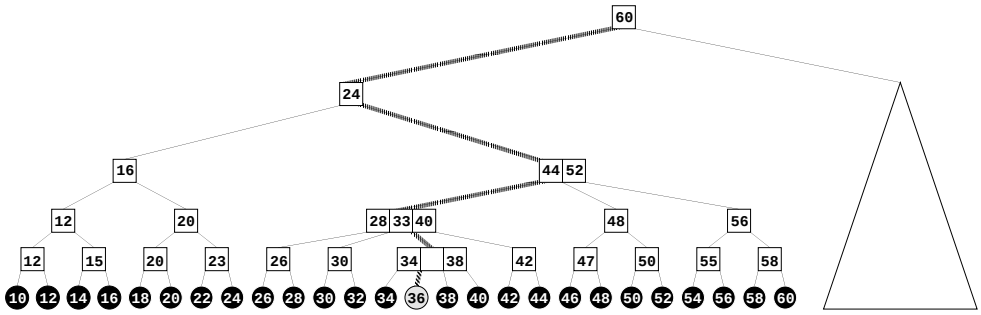


Figure 3.12: *Un arbre* 2–4 *avant la scission par la clé* 36.

On obtient alors deux forêts F_1 et F_2, où F_1 est la suite d'arbres à gauche du chemin, et F_2 la suite d'arbres à droite du chemin.

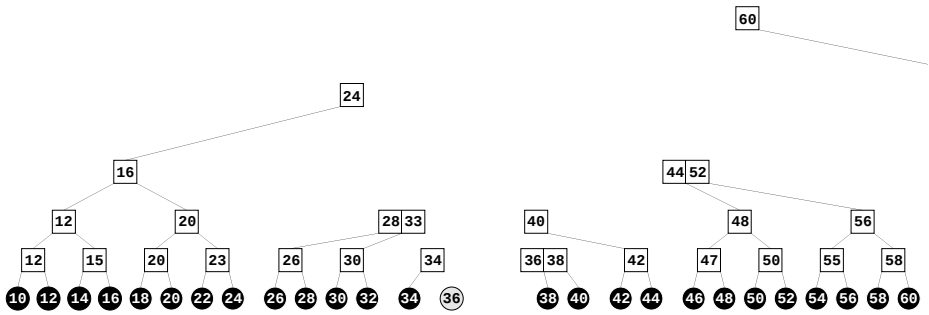


Figure 3.13: *Les deux forêts* F_1 *et* F_2 *après la séparation.*

Soient $D_1, \ldots, D_m$ les arbres constituant F_1, numérotés de la gauche vers la droite. Notons que $m \leq h$, où h est la hauteur de A. Chaque arbre D_i est un arbre a–b sauf peut-être pour la racine qui peut n'avoir qu'un fils. Par ailleurs, $\max(D_i) < \min(D_{i+1})$, et l'on connaît une clé c_i telle que $\max(D_i) \leq c_i < \min(D_i)$. C'est la balise qui se trouve dans le champ le plus à droite de la racine de D_i. Enfin, on a $h_i > h_{i+1}$, où h_i est la hauteur de D_i. On effectue alors la séquence

$$\begin{aligned}
&\textsc{Concaténer}(D_{m-1}, D_m, D'_{m-1});\\
&\textsc{Concaténer}(D_{m-2}, D'_{m-1}, D'_{m-2});\\
&\qquad\vdots\\
&\textsc{Concaténer}(D_1, D'_2, D'_1).
\end{aligned}$$

Concaténer des arbres qui sont des arbres a–b sauf éventuellement pour la racine qui n'a qu'un seul fils donne un arbre de même nature. Si D'_1 a une racine n'ayant qu'un fils, on supprime la racine, et on obtient un arbre a–b.

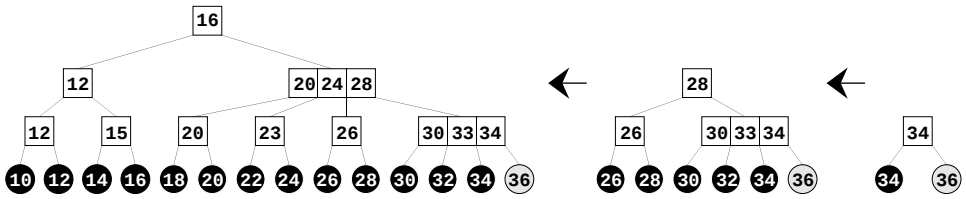


Figure 3.14: *Reconstitution de l'arbre à partir de la forêt F_1.*

Analysons le coût de la reconstruction. Soit h'_i la hauteur de D'_i. Le coût est

$$O(|h_{m-1} - h_m| + \sum_{i=1}^{m-2} |h_i - h'_{i+1}| + m).$$

Or, on a

$$h_{i+1} \leq h'_{i+1} \leq 1 + h_{i+1} \leq h_i \,.$$

En effet, seule l'inégalité centrale doit être prouvée. Pour cela, observons que $h'_{m-1} \leq 1 + \max(h_{m-1}, h_m) = 1 + h_{m-1}$, et, en raisonnant par récurrence descendante, $h'_{i+1} \leq 1 + \max(h_{i+1}, h'_{i+2}) = 1 + h_{i+1}$. Il en résulte que

$$\begin{aligned} |h_{m-1} - h_m| + \sum_{i=1}^{m-2} |h_i - h'_{i+1}| + m &\leq h_{m-1} - h_m + \sum_{i=1}^{m-2} (h_i - h'_{i+1}) + m \\ &\leq h_{m-1} - h_m + \sum_{i=1}^{m-2} (h'_i - h'_{i+1}) + m \leq h'_1 - h'_{m-1} + h_{m-1} - h_m + m \\ &\leq 2h = O(\log |S|). \end{aligned}$$

Ceci achève la preuve. ∎

6.3.6 Coût amorti des arbres 2–4

Nous analysons maintenant le coût du rééquilibrage non pas sur une seule modification d'un arbre a–b, mais sur une séquence d'insertions et de suppressions. Dans ce cas, une analyse fine du coût est possible, car les opérations de rééquilibrage se répercutent sur plusieurs insertions ou suppressions consécutives. Le résultat que nous établissons montre que le coût total des rééquilibrages est linéaire en fonction du nombre d'insertions et de suppressions, si l'on commence avec un arbre vide. Un corollaire de ce résultat remarquable est qu'en moyenne, le coût du rééquilibrage est constant dans un arbre 2–4.

Proposition 3.7. *On considère une suite quelconque de i insertions et de d suppressions dans un arbre 2–4 initialement vide, et on pose $n = i + d$; alors $P \leq d \leq n$ et $E + F \leq n + (i - d - 1)/2$, où P est le nombre de partages, E le nombre d'éclatements, et F le nombre de fusions de sommets.*

En particulier, la proposition précédente admet le corollaire suivant :

Théorème 3.8. *On considère une suite quelconque de n insertions ou suppressions dans un arbre 2–4 initialement vide. Alors le nombre total d'opérations de rééquilibrage est au plus $3n/2$.*

Ce résultat montre en particulier qu'il y a en moyenne $3/2$ opérations de rééquilibrage par insertion ou suppression. Ce nombre est indépendant de la taille de la structure, et est remarquablement faible.

Preuve du théorème. En vertu de la proposition, on a

$$E + F + P \leq n + (i - d - 1)/2 + d = n + (i + d - 1)/2 \leq 3n/2$$

parce que $i + d = n$. ∎

La preuve de la proposition repose sur une *mesure d'équilibre* d'un arbre 2–4 que nous allons introduire maintenant. Cette mesure indique, pour tout nœud de l'arbre, s'il est « bien » équilibré, c'est-à-dire en fait s'il a 3 fils. Cette mesure sera définie et étudiée pour des arbres qui ne sont pas des arbres 2–4, mais des arbres en cours de rééquilibrage. Un couple (A, s), où s est un sommet de l'arbre A, est un arbre 2–4 *partiellement équilibré* si

$$\begin{array}{l} 1 \leq d(s) \leq 5, \\ 2 \leq d(x) \leq 4 \quad \text{pour } x \neq s. \end{array}$$

L'*équilibre* d'un sommet x d'un arbre partiellement équilibré est le nombre $e(x)$ défini par $e(x) = \min(d(x) - 2, 4 - d(x))$. Il est immédiat que

$$e(x) = \begin{cases} -1 & \text{si } d(x) = 1 \text{ ou } d(x) = 5, \\ 0 & \text{si } d(x) = 2 \text{ ou } d(x) = 4, \\ 1 & \text{si } d(x) = 3. \end{cases}$$

L'*équilibre* de l'arbre A est la somme des équilibres de ses nœuds :

$$e(A) = \sum_{x \in \text{noeuds}(A)} e(x).$$

L'idée de la construction, et de la preuve de l'estimation, est que toute opération d'insertion ou de suppression de feuille dans l'arbre le déséquilibre, au sens de la mesure introduite, alors qu'une opération de fusion, d'éclatement ou de partage le rééquilibre.

Proposition 3.9. *Soit A un arbre 2–4 et soit A' obtenu par l'insertion ou la suppression d'une feuille sans rééquilibrage; alors $e(A') \geq e(A) - 1$.* ∎

Proposition 3.10. *Soit A un arbre 2–4 à m feuilles; alors $0 \leq e(A) \leq (m-1)/2$.*

Preuve. On note m_i le nombre de nœuds ayant i fils. Alors $e(A) = m_3$. Par ailleurs, le nombre d'arêtes de l'arbre A est $2m_2 + 3m_3 + 4m_4$, et il est aussi égal à $m + m_2 + m_3 + m_4 - 1$, puisque chaque sommet, à l'exception de la racine, a un père. Il résulte de l'égalité de ces expressions que

$$m_2 + 2m_3 + 3m_4 = m - 1$$

d'où le résultat. ∎

Lemme 3.11 (éclatement)**.** *Soit (A, s) un arbre 2–4 partiellement équilibré, et supposons que le sommet s a 5 fils; soit A' l'arbre obtenu par éclatement du sommet s. Alors $e(A') \geq 1 + e(A)$.*

Preuve. Notons d'abord que A' est partiellement équilibré. Si s n'est pas la racine de A, soit x le père de s dans A. Dans l'arbre A', le sommet x a deux fils s' et s'' à la place de s; ces deux sommets se partagent les 5 fils de s (voir figure 3.15).

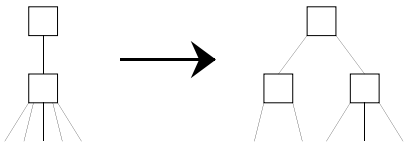


Figure 3.15: *Eclatement de x.*

Notons e' l'équilibre de sommets dans l'arbre A'. Alors $e'(x) \geq e(x) - 1$, et $e'(s') = 0$, $e'(s'') = 1$ ou vice-versa, d'où

$$e'(x) + e'(s') + e'(s'') = e'(x) + 1 \geq e(x) = 1 + e(x) + e(s)$$

et comme l'équilibre des autres sommets n'est pas modifié, ceci montre que $e(A') \geq 1 + e(A)$. Si s est la racine de A, alors A' a une nouvelle racine, soit r, ayant deux fils s' et s'' qui se partagent les 5 fils de s. On obtient

$$e'(r) + e'(s') + e'(s'') = 1 > 1 + e(s)$$

donc ici encore $e(A') \geq 1 + e(A)$. ∎

Lemme 3.12 (partage)**.** *Soit (A, s) un arbre 2–4 partiellement équilibré dont le sommet s a un seul fils, et supposons que s possède un frère voisin t ayant au moins 3 fils. Soit A' l'arbre obtenu par partage entre s et t. Alors $e(A') \geq e(A)$.*

Preuve. Notons e' l'équilibre de sommets dans l'arbre A'. Comme $e'(t) \geq e(t) - 1$, $e'(s) = 0$, $e(s) = -1$, on obtient $e'(t) + e'(s) \geq e(t) - 1 = e(t) + e(s)$. Les autres sommets ne sont pas modifiés; il en résulte que $e(A') \geq 1 + e(A)$. ∎

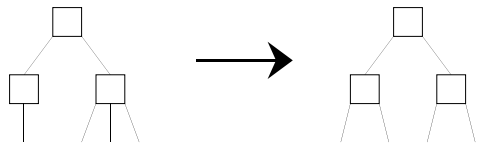


Figure 3.16: *Partage entre les frères s et t.*

Lemme 3.13 (fusion)**.** *Soit (A, s) un arbre 2–4 partiellement équilibré dont le sommet s a un seul fils. Soit A' l'arbre obtenu en supprimant s, si s est la racine, et l'arbre obtenu par fusion de s et t, si s n'est pas la racine, et si s possède un frère voisin t ayant 2 fils. Alors $e(A') \geq 1 + e(A)$.*

Preuve. Considérons d'abord le cas où s est la racine. Si s est le sommet unique de A, alors l'arbre A' est vide, et on a $e(A') = 0 = 1 + e(A)$. Sinon, soit t le fils de s; alors A' est l'arbre de racine t, et on a encore $e(A') = 1 + e(A)$.

Si s n'est pas la racine de A, soit x le père de s et t; dans A', les deux fils s et t de x sont remplacés par un unique fils s' réunissant les fils de s et t (figure 3.17).

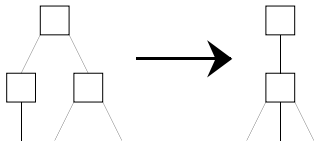


Figure 3.17: *Fusion des frères s et t.*

Notons e' l'équilibre de sommets dans A'. Comme $e'(x) \geq e(x) - 1$ et $e'(s') = 1$, on obtient $e'(x) + e'(s') \geq e(x) \geq 1 + e(x) + e(s) + e(t)$. Les autres sommets n'étant pas modifiés, on a $e(A') \geq 1 + e(A)$. ∎

Preuve de la proposition 3.7. Il y a au plus un partage par suppression et aucun par insertion; les inégalités sur P sont donc évidentes.

Pour prouver l'autre inégalité, soit A l'arbre obtenu, à partir de l'arbre vide après les n opérations d'insertion ou suppression. L'arbre vide a pour équilibre 0. Par ailleurs, chacune des n insertions ou suppressions diminue l'équilibre d'au plus 1 (Prop 3.9), chaque éclatement ou fusion l'augmente d'au moins 1, et chaque partage l'augmente ou le laisse invariant. Il en résulte que $e(A) \geq F + E - n$. Or, d'après la proposition 3.10, on a $e(A) \leq (i - d - 1)/2$, parce que A a $i - d$ feuilles. La proposition 3.7 résulte de ces deux inégalités. ∎

6.4 Arbres bicolores

6.4.1 Présentation

Les arbres bicolores, dus à Guibas et Sedgewick, constituent une structure d'arbres binaires de recherche particulièrememt efficace. Traditionnellement appelés arbres rouge-noir, nous les colorons en blanc et noir pour des raisons évidentes. Ces arbres sont efficaces pour plusieurs raisons. C'est une structure souple car elle permet de réaliser « aisément » de nombreuses opérations telles que recherche, insertion, suppression, mais aussi des opérations plus sophistiquées comme la scission ou la concaténation.

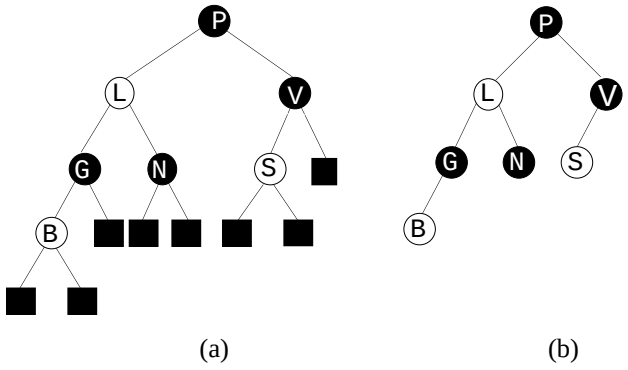


Figure 4.1: *Un arbre bicolore : (a) avec ses feuilles, (b) sans feuilles.*

Un *arbre bicolore* est un couple (A, c) où A est un arbre binaire complet et c est une application qui associe à chaque sommet une couleur (*blanc* ou *noir*) de façon que :

(a) toutes les feuilles sont noires ;
(b) la racine est noire ;
(c) le père d'un sommet blanc est noir ;
(d) les chemins issus d'un même sommet et se terminant en une feuille ont le même nombre de sommets noirs.

Un *arbre bicolore pour un ensemble ordonné* E est un arbre bicolore qui est un arbre de recherche pour E, avec la restriction que seuls les nœuds contiennent des éléments de E.

Notons que la condition (b) n'est pas toujours exigée, mais il est plus pratique pour la suite de ne considérer que des arbres bicolores ayant des racines noires.

Exemple. Dans la figure 4.1, les feuilles sont représentées par des carrés. On conviendra en général de ne pas les représenter. Ainsi l'arbre de la figure 4.1(a) devient celui de la figure 4.1(b).

Remarque. Notons que dans un arbre bicolore « effeuillé » , le grand-père d'un sommet est nécessairement un sommet complet, ainsi l'arbre, d'une certaine façon, est « presque » complet.

Nous allons donner immédiatement une propriété caractéristique des arbres bicolores, car selon les circonstances, la définition donnée ci-dessus ou la propriété ci-dessous sera plus adaptée.

Une *fonction rang* sur un arbre binaire complet A est une application rg de l'ensemble S des sommets de A dans l'ensemble $\mathbb{N}$ des entiers naturels vérifiant les conditions suivantes (p désignant la fonction père sur S) :

(i) pour toute feuille x, $rg(x) = 0$ et si x a un père $rg(p(x)) = 1$;
(ii) pour tout sommet x, $rg(x) \leq rg(p(x)) \leq rg(x) + 1$;
(iii) pour tout sommet x, $rg(x) < rg(p^2(x))$.

Proposition 4.1. *Tout arbre bicolore possède une fonction rang.*

Preuve. Soit (A, c) un arbre bicolore, et définissons le rang d'un sommet comme le nombre de sommets noirs situés sur un chemin quelconque allant de ce sommet (exclu) à une feuille de A. Notons que cette définition du rang est consistante en vertu de la condition (d) dans la définition d'un arbre bicolore . Démontrons que l'application rg que nous venons de définir satisfait les propriétés (i), (ii), (iii).

Par définition du rang d'un sommet on a immédiatement que (a)$\Longrightarrow$ (i). Il est clair aussi que $rg(x) \leq rg(p(x))$ pour tout sommet x, et que par ailleurs si x est blanc alors $rg(p(x)) = rg(x)$, et si x est noir alors $rg(p(x)) = rg(x) + 1$. Donc $rg(p(x)) \leq rg(x) + 1$. Ainsi (ii) est vérifié.

Reste à prouver (iii). On a $rg(x) \leq rg(p(x)) \leq rg(p^2(x))$ pour tout x. Supposons que $rg(x) = rg(p^2(x))$. Alors x est blanc ainsi que $p(x)$, ce qui contredit (c). Donc (c)$\Longrightarrow$(iii). ∎

Proposition 4.2. *Si A est un arbre binaire possède une fonction rang, alors il existe une affectation de couleurs c aux sommets de A telle que (A, c) soit un arbre bicolore.*

Preuve. Réalisons l'affectation des couleurs aux sommets de A de la façon suivante : tout sommet ayant même rang que son père est déclaré blanc, les autres étant déclarés noirs (donc aussi la racine). Montrons que (A, c) est bicolore.

Par définition même de la fonction c, les conditions (a) et (b) sont immédiates. Vérifions (c). Soit x un sommet blanc, x n'est donc pas la racine, et son père $p(x)$ a même rang que lui. Si $p(x)$ était blanc, alors $p(x)$ aurait un père de même rang, ce qui contredirait (iii). Donc (iii) $\Longrightarrow$(c).

Il reste à prouver (d). Pour ce faire, on prouve par induction sur la hauteur du sommet x la propriété (d') : tous les chemins issus de x et menant à une feuille ont même nombre de sommets noirs (x non compris) et ce nombre est égal à $rg(x)$.

Si x est une feuille, (d') est vraie pour x. Soit x un nœud qui n'est pas une feuille, alors par hypothèse d'induction, (d') est vraie pour les fils y et z de x. Il y a trois cas à envisager :

(α) $rg(x) = rg(y) = rg(z)$
(β) $rg(x) = rg(y) + 1 = rg(z) + 1$
(γ) $rg(x) = rg(y) + 1 = rg(z)$.
Sur la figure 4.2 on a représenté les trois cas, le sommet dont la couleur n'est pas connue est représenté par deux demi-disques de couleurs différentes.

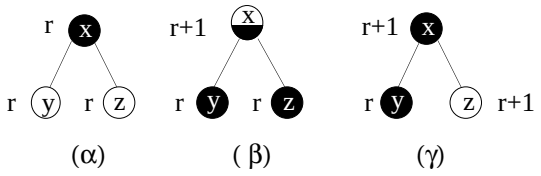


Figure 4.2: *Affectation d'une couleur à un nœud en fonction du rang.*

Dans le cas (α), y et z sont blancs, et puisque y et z vérifient (d'), x vérifie (d').

Dans le cas (β), y et z sont noirs, donc pour les mêmes raisons qu'en (α), x vérifie (d').

Dans le cas (γ), y est noir et z est blanc. Mais puisque y et z vérifient (d') et que $rg(y) + 1 = rg(z)$, tous les chemins issus de x et allant à une feuille ont le même nombre de sommets noirs (x non compris) et ce nombre est égal à $rg(x)$, donc x vérifie (d'). ∎

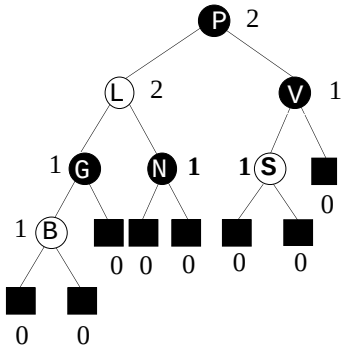


Figure 4.3: *Attribution d'un rang aux sommets de l'arbre de la première figure.*

Nous avons représenté sur la figure 4.3 les rangs des sommets de l'arbre bicolore de la figure 4.1. On laisse au lecteur le soin de vérifier en exercice que la correspondance que l'on vient d'établir entre les arbres bicolores et les arbres binaires ayant une fonction rang est une bijection, à savoir que (A, c) étant donné, la seule fonction rg sur A qui vérifie (i), (ii), (iii) est celle que nous avons définie, et de même pour la réciproque, la seule coloration possible est celle qui a été donnée.

6.4.2 Hauteur d'un arbre bicolore

L'efficacité des opérations élémentaires, en particulier de la recherche, dépend essentiellement de la hauteur de l'arbre binaire de recherche. C'est pourquoi il est

nécessaire d'évaluer de façon précise la hauteur d'un arbre bicolore en fonction du nombre de ses sommets.

Lemme 4.3. *Soit A un arbre bicolore à n sommets, et soit x un sommet dans A de rang k et de hauteur h. Alors $h \leq 2k$ et $n \geq 2^k$; de plus, si x est blanc, les deux inégalités sont strictes.*

Preuve. Par induction sur la hauteur du nœud x. Si x est une feuille, la propriété est trivialement vraie car x est noir et k est nul. Supposons que la hauteur h de x soit strictement positive. Soient y et z les fils de x. Les trois cas possibles aux symétries près sont représentés sur la figure 4.4. On notera $h(x)$ et $n(x)$ respectivement la hauteur et le nombre de sommets du sous-arbre de A de racine x.

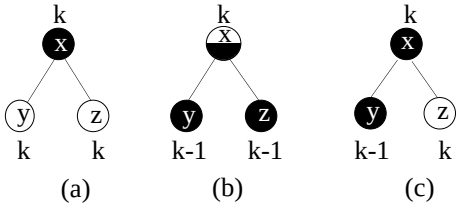


Figure 4.4: *Les trois cas possibles.*

Cas (a) Ici, x est noir et y et z sont blancs. Par hypothèse d'induction, on a : $h(y) < 2k$, $h(z) < 2k$, $n(y) > 2^k$ et $n(z) > 2^k$. Donc $h(x) \leq 2k$ et $n(x) = 1+n(y)+n(z) \geq 2^k$. Or x est noir et de rang k. Les inégalités sont donc correctes.

Cas (b) Les fils y et z sont noirs et x est blanc ou noir. Par induction, on a $h(y) \leq 2(k-1)$, $h(z) \leq 2(k-1)$, $n(y) \geq 2^{k-1}$ et $n(y) \geq 2^{k-1}$. Donc $h(x) \leq 2(k-1)+1 < 2k$ et $n(x) \geq 2^k + 1 > 2^k$. Comme x est blanc ou noir, le lemme est vérifié dans ce cas. Reste le dernier cas :

Cas (c) Un des fils, disons y est noir et l'autre blanc et le père x est noir. On a alors $h(y) \leq 2(k-1)$, $h(z) < 2k$, $n(y) \geq 2^{k-1}$ et $n(z) > 2^k$. Donc $h(x) \leq 2k$ et $n(x) > 2^{k-1} + 2^k + 1 > 2^k$. Comme x est noir, le lemme est vérifié. ∎

Proposition 4.4. *Un arbre bicolore ayant n sommets ($n > 0$) et de hauteur h vérifie :*

$$\lfloor \log n \rfloor \leq h \leq 2\lfloor \log n \rfloor$$

Preuve. La preuve découle directement du lemme précédent et du fait qu'un arbre binaire à $n > 0$ sommets est de hauteur au moins $\lfloor \log n \rfloor$. ∎

Corollaire 4.5. *La recherche d'un élément dans un arbre bicolore prend un temps $O(\log n)$ où n est le nombre de sommets de l'arbre.*

6.4.3 Insertion dans un arbre bicolore

Soit m un nouvel élément à insérer dans un arbre bicolore A. Comme dans un arbre binaire de recherche ordinaire, on descend dans l'arbre en partant de la racine. On crée un nouveau sommet blanc x contenant m dont les fils sont des feuilles noires. Soit A' l'arbre obtenu ; cet arbre n'est plus nécessairement un arbre bicolore (figure 4.5).

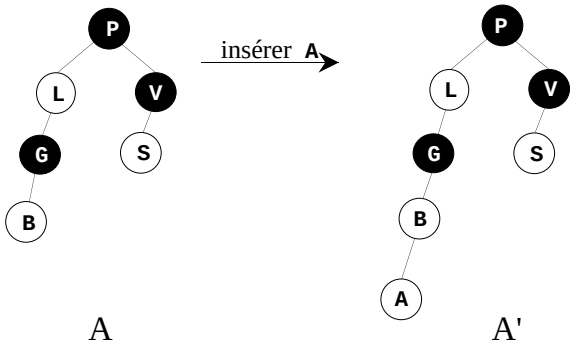


Figure 4.5: *Première phase de l'insertion de* A *dans A.*

En fait, seules les propriétés (b) ou (c) de la définition peuvent ne plus être respectées dans A', à savoir : la racine est blanche (parce que x est la racine) ou bien : un sommet et son père sont blancs (parce que le père de x est blanc).

Procédure de rééquilibrage

Nous allons donner 3 règles (et leurs symétriques) qui permettent de rééquilibrer l'arbre après insertion.

Si A était vide avant l'insertion (cas où (b) n'est pas respecté), il suffit de colorer x en noir, et c'est terminé (règle (γ) de la figure 4.6).

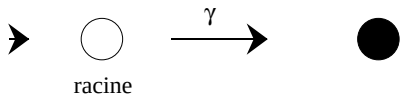


Figure 4.6: *Rééquilibrage après une insertion : règle* γ.

Sinon, le père y de x est blanc, donc y n'est pas la racine et son père z est noir. Supposons, pour fixer les idées, que y est fils gauche de z, et notons t le fils droit de z. Appliquons l'algorithme suivant, où x est le sommet courant qui au départ est le sommet nouvellement créé (le sommet courant est indiqué par une flèche sur la figure) :

• si y a un frère blanc on applique α_1 ou α_2 selon «l'éloignement» de x et t, i.e. selon que x est un fils gauche ou droit (si y est un fils droit, il faut prendre la règle

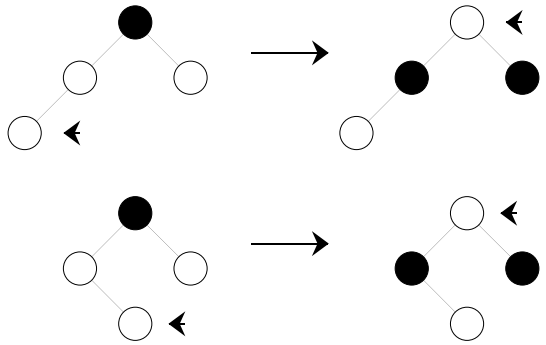


Figure 4.7: *Rééquilibrage après une insertion : règles α.*

symétrique qui revient à considérer effectivement l'éloignement des deux sommets x et t) ; le sommet courant devient z (α_1 ou α_2 ne sont que des changements de couleur). Alors si l'arbre obtenu n'est pas bicolore, c'est qu'à nouveau l'une des propriétés (b) ou (c) n'est pas respectée au sommet z, on itère donc le processus.

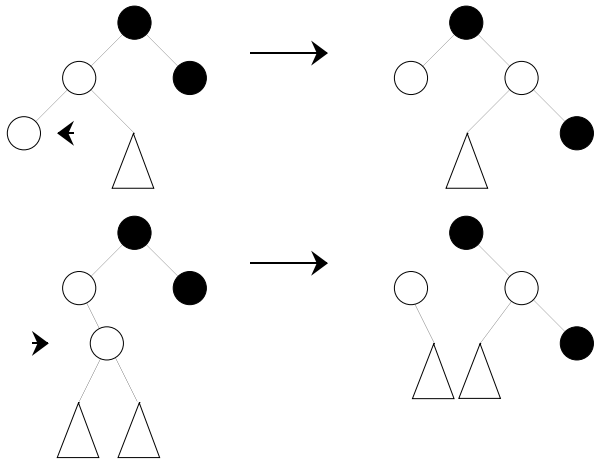


Figure 4.8: *Rééquilibrage après une insertion : règles β.*

- si y a un frère noir t, on fait une rotation simple (β_1) dans le cas où x et t sont «éloignés» et une rotation double (β_2) dans le cas où ils sont proches ; dans les deux cas, l'arbre obtenu est bicolore et le processus de rééquilibrage est terminé.

Puisque les règles α diminuent de deux la profondeur du sommet courant et que les règles γ et β sont des règles d'arrêt, il est clair que l'itération se termine et fournit un arbre de recherche bicolore.

Proposition 4.6. *Une insertion dans un arbre bicolore à n sommets prend un temps $O(\log n)$ et nécessite au plus deux rotations.*

Preuve. La descente avant insertion prend un temps $O(\log n)$. Puis dans la procédure de rééquilibrage, à chaque opération élémentaire, le sommet courant

a une profondeur qui diminue strictement, sauf en cas de règle d'arrêt, donc le nombre total d'opérations élémentaires est $O(\log n)$, par ailleurs chaque opération élémentaire prend un temps $O(1)$, d'où la première assertion. Par ailleurs, les règles nécessitant des rotations sont terminales, il s'agit de (β_1) (une rotation) et (β_2) (une rotation double) donc le résultat est démontré. ∎

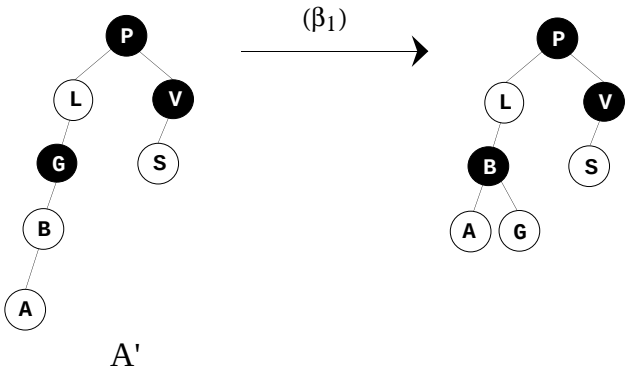


Figure 4.9: *Rééquilibrage après insertion de* A.

6.4.4 Suppression d'un élément dans un arbre bicolore

Considérons maintenant le problème de la suppression. On procède en deux étapes. La *première phase* consiste à appliquer l'algorithme de suppression d'un élément dans un arbre de recherche binaire classique. En descendant dans l'arbre on trouve le sommet x dans lequel est rangé l'élément m que l'on veut supprimer.

• si le fils droit de x est une feuille, alors on supprime x et on le remplace par son sous-arbre gauche de racine y; notons que ce sous-arbre gauche, du fait qu'on a affaire à un arbre bicolore, est soit une feuille, soit un arbre de hauteur 1 dont la racine est blanche et dont les fils sont des feuilles;

– si x était blanc, l'arbre reste un arbre bicolore et on peut remarquer que dans ce cas y est une feuille (figure 4.10 (a));

– si x était noir, il y a deux cas à envisager selon la couleur du sommet y qui a remplacé x :

– si y est blanc, ses fils sont des feuilles, on colore y en noir (figure 4.10 (b)) et l'algorithme est terminé;

– si y est noir, y est une feuille et y devient « *dégradé*» ce que l'on note sur la figure par un signe $-$, nous expliquerons dans un instant ce que cela signifie précisément (figure 4.10 (c));

• si le fils droit de x n'est pas une feuille, alors on recherche le sommet y contenant le prédécesseur l de m; ce sommet y a pour fils droit une feuille; on remplace m par l dans x et on supprime le sommet y par l'algorithme ci-dessus.

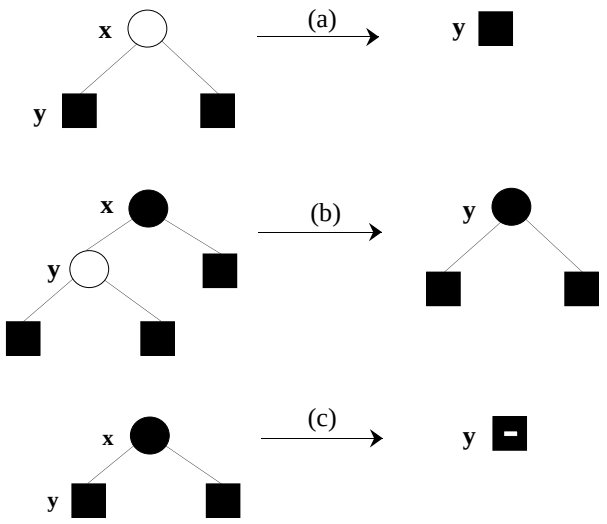


Figure 4.10: *Première phase de suppression d'un élément dans un arbre bicolore .*

Précisons la signification d'un sommet *dégradé* : lorsqu'un sommet x est dégradé, cela signifie que le nombre de sommets noirs sur un chemin de x à une feuille quelconque, x **compris**, est inférieur d'une unité au nombre de sommets noirs sur un chemin de y (frère de x) à une feuille quelconque, y **compris**, et donc le père de x, s'il existe, ne respecte pas la condition (d) de la définition des arbres bicolores.

La *deuxième phase* consiste à rééquilibrer l'arbre dans le cas où il contient un sommet dégradé. Les schémas des figures 4.11 et 4.12 donnent l'algorithme à appliquer.

Le sommet courant x est le sommet dégradé, il est indiqué par une flèche et on suppose que si x a un père y alors x est un fils gauche, et on appelle z son frère (on obtient toutes les règles par symétrie, on notera par des lettres primées les règles symétriques des règles indiquées). Plusieurs remarques sont à faire pour justifier la validité de l'algorithme. Tout d'abord, si z existe, alors, par définition d'un sommet dégradé, puisque x est dégradé, z ne peut être une feuille, donc z a deux fils. Ensuite, on peut noter que tous les cas sont examinés :

– soit x est la racine (règle (a_1)) ;

– soit x a un frère noir z qui a lui même deux fils noirs (règles (a_2) ou (a_3) selon que le père y est noir ou blanc) ;

– soit x a un frère noir z dont l'un des deux fils est blanc (règles (b_2) ou (b_3) selon qu'il s'agit d'un fils gauche ou droit ; notons que si les deux fils de z sont blancs on peut appliquer l'une ou l'autre règle) ;

– soit x a un frère blanc (règle (b_1)).

On peut noter que la profondeur du sommet courant diminue strictement à chaque application d'une règle sauf pour la règle (b_1), mais cette règle est suivie de

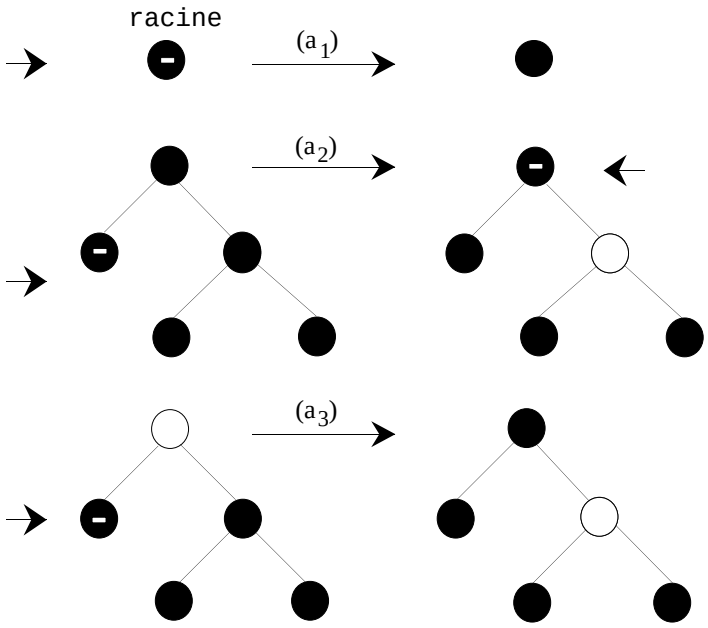


Figure 4.11: *Rééquilibrage après une suppression : règles* $(a_1), (a_2), (a_3)$.

l'application d'une règle (a_3), (b_2) ou (b_3) qui sont des règles d'arrêt. Donc le nombre de règles appliquées est au plus $O(\log n)$ et chacune prend un temps $O(1)$. Enfin, il est facile de vérifier que les règles appliquées sont correctes, i.e. que si le sommet courant est dégradé avant application de (a_2) ou (b_1), alors le sommet courant après application de la règle est bien un sommet dégradé , et que, après application de l'une des autres règles, qui sont des règles d'arrêt, l'arbre obtenu est un arbre bicolore.

Proposition 4.7. *Une suppression dans un arbre bicolore à n sommets ($n > 0$) prend un temps $O(\log n)$ et nécessite au plus trois rotations.*

Preuve. La première assertion découle immédiatement des remarques ci-dessus, chaque phase prenant un temps $O(\log n)$. Quant aux rotations, elles apparaissent dans les règles (b_1) (une rotation), (b_2) (une rotation) et (b_3)) (une rotation double). Or la règle (b_1) est suivie de l'une des règles $(a_3), (b_2), (b_3)$, qui sont toutes les trois terminales, la preuve est donc terminée. ∎

Exemple. Dans l'exemple de la figure 4.13, on supprime le contenu p de la racine.

La première phase consiste à mettre dans la racine la plus grande clé du sous-arbre gauche, à savoir n, et à supprimer le sommet qui contient cette clé, ce qui dégrade une feuille. On doit alors procéder au rééquilibrage de l'arbre. On applique alors la règle (b'_1), puis la règle (b'_3).

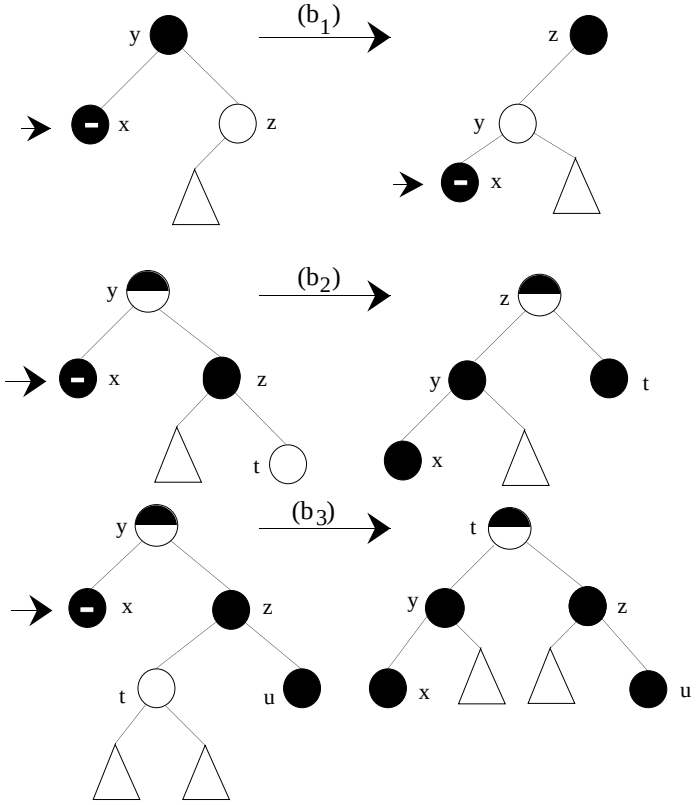


Figure 4.12: *Rééquilibrage après une suppression : règles* $(b_1), (b_2), (b_3)$.

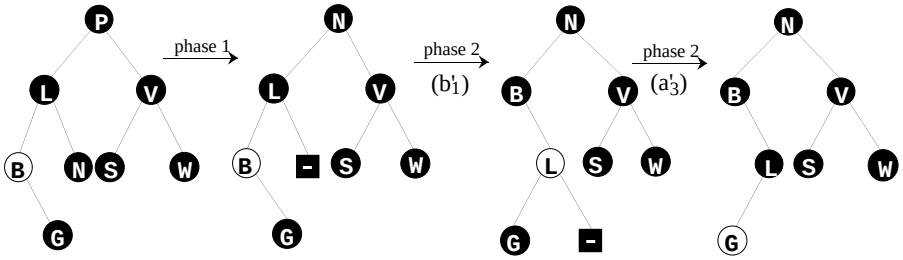


Figure 4.13: *Suppression de p.*

6.5 Enrichissement

Dans les sections précédentes, nous avons décrit trois familles d'arbres équilibrés qui implémentent, de façon efficace, les opérations d'un dictionnaire (recherche, insertion et suppression). Nous avons également constaté que la scission et la concaténation pouvaient être réalisées en temps logarithmique.

Dans cette section, nous montrons comment implémenter d'autres opérations, comme la recherche du k-ième élément, ou du médian, en temps logarithmique. Pour cela, il convient d'augmenter la structure (de l'*enrichir*) par quelques données additionnelles aux sommets. Le but de cette section est de montrer qu'un

arbre équilibré enrichi de cette manière possède à la fois la souplesse d'un arbre et d'une liste chaînée.

Père

Soit A un arbre, et considérons la fonction PÈRE(x : sommet; A : arbre) qui donne le père de x dans A, et qui est indéfini si x est la racine de A. Le calcul de cette fonction n'est pas facile si l'on ne dispose pas d'un champ supplémentaire dans les sommets. Enrichissons donc la structure en associant, à chaque sommet, un pointeur supplémentaire qui donne son père (ce pointeur vaut NIL pour la racine). La valeur de ce pointeur est facile à maintenir lors d'une insertion ou d'une suppression d'un sommet. On se convainc aisément que la mise à jour de ces pointeurs, lors d'une rotation ou d'une double rotation, se fait en temps constant.

Voisins

Soit A un arbre binaire de recherche pour un ensemble de clés S. Le parcours en ordre symétrique (voir chapitre 4) de l'arbre fournit ses clés en ordre croissant. On peut donc définir les fonctions SUIVANT et PRÉCÉDENT sur les sommets de A, comme dans une liste :

SUIVANT(x : sommet; A : arbre) : sommet;
: Donne le sommet contenant la clé qui suit la clé de x; indéfini si la clé de x est la plus grande clé dans A.

PRÉCÉDENT(x : sommet; A : arbre) : sommet;
: Donne le sommet contenant la clé qui précède la clé de x; indéfini si la clé de x est la plus petite clé dans A.

La réalisation en temps *constant* de ces deux opérations sur un arbre balisé (qu'il soit binaire, ou a–b) est particulièrement facile. On munit pour cela chaque feuille de deux pointeurs supplémentaires, l'un pointant vers son voisin de gauche (dont la clé est justement la clé précédente), l'autre vers son voisin de droite. Cela revient à plaquer, sur la suite des feuilles, une structure de liste doublement chaînée. La gestion de ces pointeurs, lors d'une insertion ou d'une suppression, se fait comme pour les listes, et le surcoût par opération est donc constant. Dans le cas particulier des arbres a–b (et plus généralement des arbres dont toutes les feuilles sont à la même profondeur), on peut continuer cette construction et créer une liste doublement chaînée non seulement aux feuilles, mais à tous les niveaux de l'arbre. La structure qui en résulte est un arbre à *liaisons par niveau* («level-linked tree»).

Ces arbres ont des propriétés remarquables; en particulier, la recherche d'une feuille qui se trouve à distance d d'une feuille donnée peut se faire en temps $O(\log d)$ (exercice).

Lorsque l'arbre n'est pas balisé, donc lorsque les informations se trouvent sur les sommets, la même structure de liste doublement chaînée est utilisée (voir figure

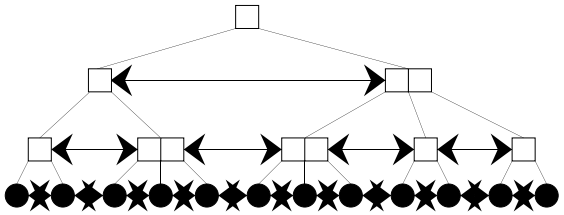


Figure 5.1: *Un arbre* 2–3 *à liaisons par niveau.*

5.2). On se convainc facilement que les rotations et doubles rotations laissent cette liste inchangée. En ce qui concerne la suppression d'une clé, on est amené, rappelons-le, à supprimer une feuille. C'est cette feuille qui est aussi supprimée dans la liste.

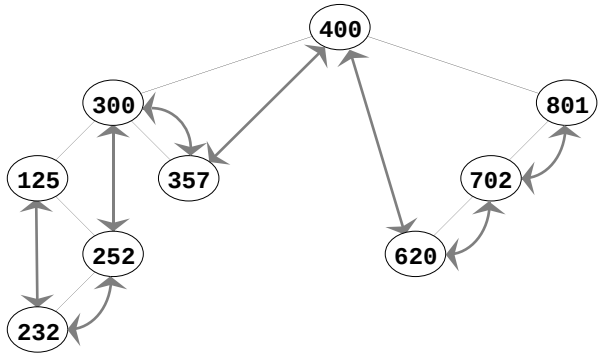


Figure 5.2: *Une liste doublement chaînée sur les sommets d'un arbre.*

Proposition 5.1. *Dans un arbre de recherche équilibré, on peut, moyennant des pointeurs supplémentaires, implémenter les opérations* SUIVANT *et* PRÉCÉDENT *en temps constant.* ∎

Numéro d'ordre

Le *numéro d'ordre* ou le rang d'une clé dans un ensemble S est le nombre de clés qui lui sont inférieures ou égales. C'est le numéro que reçoit la clé quand elles sont numérotées, à partir de 1, en ordre croissant.

Considérons le problème de la recherche de la clé de numéro d'ordre k dans un ensemble représenté par un arbre binaire de recherche A. Bien entendu, il ne s'agit pas de faire un parcours symétrique de l'arbre. Dotons chaque sommet x d'un champ supplémentaire dont la valeur $n(x)$ est le nombre de sommets dans le sous-arbre $A(x)$ de racine x (la *taille*). En particulier, pour la racine r, le nombre $n(r)$ est le nombre de sommets de A. Notons $\sigma(x)$ le numéro d'ordre de x dans la numérotation infixe. Posons

$$n_g(x) = \begin{cases} n(y) & \text{si } y \text{ est le fils gauche de } x\,; \\ 0 & \text{si } x \text{ n'a pas de fils gauche.} \end{cases}$$

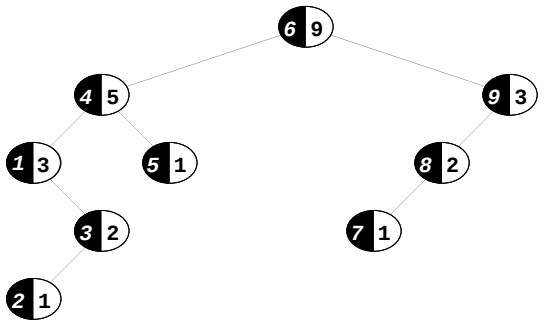


Figure 5.3: *Les numéros d'ordre (en blanc) et les tailles (en noir).*

et de même pour n_d. On cherche le k-ième sommet comme suit : on compare k au nombre $m = 1 + n_g(r)$, où r est la racine de l'arbre. L'entier m est le numéro d'ordre de la racine. Si $k < m$, le sommet cherché est dans le sous-arbre gauche ; si $k > m$, il est dans le sous-arbre droit, et il est le sommet de numéro $k - m$ dans la numérotation de ce sous-arbre. D'où la procédure :

```
procédure SOMMET-D'ORDRE(k, A) ;
  m := 1 + n_g(RACINE(A)) ;
  si m = k alors retourner RACINE(A)
  sinon
    si k < m alors retourner SOMMET-D'ORDRE(k, A_g)
    sinon retourner SOMMET-D'ORDRE(m − k, A_d)
  finsi.
```

Il reste à étudier les répercussions des opérations d'insertion, de suppression, et de rééquilibrage. Considérons par exemple les arbres *AVL*. Une insertion ou suppression doit être suivie d'une remontée vers la racine, pour ajuster les tailles $n(x)$ sur chaque nœud x du chemin entre la racine et la feuille ajoutée ou supprimée. Quant aux rotations et doubles rotations, l'ajustement se réduit à une ou deux additions ou soustractions.

Dans le cas d'un arbre a–b, on procède essentiellement de la même manière : en plus des balises, on range dans chaque nœud le nombre de feuilles dans le sous-arbre repéré par la balise. On procède ensuite comme pour les arbres binaires. On a donc :

Proposition 5.2. *Dans un arbre de recherche on peut, moyennant un champ supplémentaire, trouver la clé de numéro d'ordre donné en temps logarithmique en fonction du nombre de clés présentes.* ∎

Médian

Etant donné un ensemble de n clés S, on cherche à le partager en deux parties S_1 et S_2 de même taille (à 1 près) telles que les clés de S_1 soient inférieures aux clés de S_2. Pour cela, il suffit de connaître la clé *médiane*, c'est-à-dire la clé dont le numéro d'ordre est $\lfloor n/2 \rfloor$, puis d'appliquer un algorithme de scission. Si, pour chaque sommet, on connaît le nombre de sommets de son sous-arbre, on peut calculer le numéro d'ordre de la clé médiane en temps constant. En temps logarithmique, on repère ensuite le sommet médian, puis à nouveau en temps logarithmique, on effectue la scission. On a donc

Corollaire 5.3. *Dans un arbre a–b, on peut effectuer la recherche du médian en temps logarithmique, donc aussi scinder un arbre en deux parts égales (à 1 près) en temps logarithmique.* ∎

6.6 Arbres persistants

6.6.1 Ensembles ordonnés persistants

Le problème à résoudre est le suivant : il s'agit de gérer un ensemble totalement ordonné E qui évolue dans le temps en ayant pour souci de ne pas perdre d'information sur les états passés de cet ensemble ordonné. En particulier, on veut être en mesure de savoir si à un instant t passé l'ensemble E possédait un élément donné x. On peut prendre connaissance du passé bien sûr, mais on ne peut pas modifier ce passé, par contre on peut modifier l'état présent de l'ensemble c'est-à-dire sa version la plus récente.

Nous formaliserons le problème de la façon suivante : le temps ou plutôt une *chronologie* est représentée par une suite $T = (t_0, \cdots, t_m)$ strictement croissante de nombres réels appelés *instants*. A l'instant t, on peut effectuer deux types d'actions sur l'ensemble E qui consistent à ajouter ou supprimer un élément dans E :

INSÉRER(p, E) :
: consiste à insérer l'élément p dans l'ensemble ordonné E (ne peut être réalisé que si p n'y est pas déjà) ;

SUPPRIMER(p, E) :
: consiste à supprimer l'élément p de E (ne peut être réalisé que si p figure dans E).

Etant donnée une chronologie $T = (t_0, \cdots, t_m)$, et une suite d'actions $h = (a_0, \ldots, a_m)$, on appelle *ensemble ordonné persistant E, d'histoire h et de chronologie T* la suite d'ensembles $(E_0, E_1, \ldots, E_m)$ telle que $E_0 = \emptyset$ et pour tout $i < m$, l'ensemble E_{i+1} est obtenu en exécutant l'action a_i sur l'ensemble E_i (la suite d'actions doit bien sûr être exécutable). L'entier m est la longueur de l'histoire

h. L'instant *présent* est l'instant t_m. Pour alléger les notations, on note encore E au lieu de E_m «l'état» de l'ensemble ordonné à l'instant t_m.

A cet instant présent, on peut poser des questions sur le passé ou avancer dans la chronologie, c'est à dire faire une nouvelle mise à jour de E. Les questions sur le passé sont de la forme suivante :

CHERCHER(a, E, t_j) :
retourne l'élément de clé a contenu dans E_j à l'instant $t_j, j \leq m$ s'il y figure, un message d'échec sinon.

En revanche, la mise à jour de E, c'est-à-dire l'insertion ou la supprssion d'un élément, se fait en ajoutant :

- un nouvel instant t_{m+1} à la chronologie T;
- une nouvelle action a_{m+1} à l'histoire h.

La structure de données que nous allons présenter pour implémenter ces opérations est due à N. Sarnak et R. Tarjan. Ses performances sont les suivantes :

Si l'histoire est de longueur m, et si E_i contient à chaque instant au plus n éléments (notons que nécessairement $n \leq m$), alors la recherche se fait en temps $O(\log m)$, la mise à jour de E en temps et en espace $O(\log n)$. Nous donnerons ensuite une version améliorée de l'algorithme qui garde la même complexité en temps, et a une complexité amortie linéaire en fonction de m en espace, c'est-à-dire que la complexité amortie en espace d'une mise à jour est $O(1)$.

Nous nous plaçons dans le cadre naturel où l'ensemble E_i est représenté par un arbre B_i. L'idée la plus simple qui vient à l'esprit pour la mise à jour à l'instant t_{m+1} est de recopier l'arbre binaire B_m avant de le transformer en arbre B_{m+1} qui représente l'ensemble E à l'instant t_{m+1}.

On réalise alors CHERCHER(x, E, t_i) comme suit : grâce à un tableau de pointeurs (figure 6.1) sur les racines des arbres B_i on peut (par une recherche dichotomique) trouver en temps $O(\log m)$ le pointeur sur l'arbre B_i, puis en temps $O(\log n)$ trouver si l'élément x figure dans B_i.

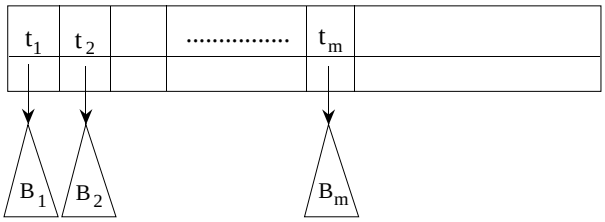


Figure 6.1: *Persistance par recopie intégrale.*

Bien sûr, la mise à jour prend sous cette forme un temps $O(n)$ pour la recopie de l'arbre, et un espace $O(n)$, donc ne répond pas aux exigences formulées plus haut.

Notons ici que si la chronologie est représentée par des entiers consécutifs (ce qui est fréquent) la recherche de l'arbre B_i peut se faire en temps $O(1)$ si on utilise un tableau de pointeurs sur les arbres B_i (figure 6.1).

Une autre idée, opposée à la première, est de ne jamais recopier un sommet. A chaque mise à jour de la structure, un sommet concerné reçoit l'information nécessaire (changement de clé, de fils) pour représenter la nouvelle version. Un sommet grossit alors, à chaque modification qui le concerne, d'un nombre constant de cellules, les sommets n'étant pas concernés par la modification restant inchangés.

L'inconvénient majeur de cette technique réside dans le temps considérable que prend la recherche. En effet, à chaque sommet, il convient de parcourir la liste des versions pour déterminer les informations relevantes à la version considérée, et le temps de ce parcours n'est pas indépendant du nombre de versions existantes.

Nous allons présenter ici deux méthodes efficaces pour représenter la liste des versions; la première est inspirée de la méthode de recopie, mais à chaque mise à jour, on ne recopie que les sommets sur le chemin de la racine du sommet concerné. La deuxième est inspirée de la méthode du grossissement des sommets. Toutefois, on se limite à un grossissement borné, et en cas de débordement, on dédouble les sommets.

Les deux méthodes sont intéressantes et efficaces, la première plus simple, et la deuxième plus économique en place.

6.6.2 Duplication de chemins

Pour gagner du temps et de l'espace dans la mise à jour, on observe que l'arbre B_{i+1} diffère peu de l'arbre B_i; on ne recopie que ce qui est nécessaire, ainsi B_{i+1} et B_i vont «partager» une structure commune. C'est la technique que nous allons exposer maintenant et que l'on cite habituellement comme la méthode de la *duplication de chemins*. Nous la développons avec des arbres bicolores mais elle peut s'effectuer sur les autres arbres binaires de recherche équilibrés.

La stratégie consiste à dupliquer tout sommet dont le contenu a été modifié ou dont un des fils a été dupliqué (ou modifié). Ainsi l'ensemble des sommets dupliqués constitue effectivement un ensemble de chemins partant de la racine de la nouvelle version. Notons que les couleurs des sommets sont seulement utiles pour la mise à jour de l'arbre et donc ne concernent que la dernière mise à jour, ainsi il n'est pas nécessaire de dupliquer un sommet dont on change simplement la couleur.

La règle de duplication est donc la suivante :

Règle de duplication. *Dupliquer tout sommet dont le contenu (qu'on appelle encore la clé) a été modifié. Dupliquer le père d'un sommet dupliqué. Le père*

dupliqué a deux pointeurs gauche et droit, un sur le fils dupliqué, l'autre sur le fils qui n'a pas été dupliqué.

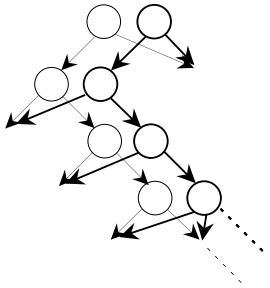


Figure 6.2: *Duplication de chemin.*

La duplication du père d'un sommet dupliqué se répercute donc jusqu'à la racine. Le but de cette règle est de ne pas modifier B_m, sauf éventuellement les couleurs des sommets, mais celles-ci sont désormais inutiles ; en partant du pointeur associé à l'instant t_m, on retrouve toujours le même arbre. Examinons en détail comment on procède à une insertion ou une suppression sur la version courante (c'est-à-dire la dernière version) de E. Mais auparavant, pour comprendre pourquoi le mécanisme de rééquilibrage est plus complexe dans le cas d'une suppression que dans celui d'une insertion, il faut observer quels sont les pointeurs modifiés dans une opération de rotation.

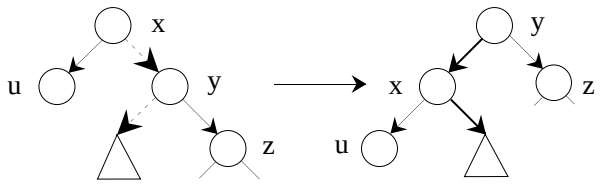


Figure 6.3: *Modifications de pointeurs dans une rotation gauche.*

La figure 6.3 montre à gauche en pointillés les pointeurs qui disparaissent, et à droite en gras les pointeurs nouvellement créés. Si l'on se reporte alors aux règles de rééquilibrage dans les arbres bicolores, on observe que dans le cas d'une insertion, les règles qui font intervenir des rotations (les règles β) ne modifient que des pointeurs issus de sommets qui, dans la structure persistante, sont des sommets déjà dupliqués, le rééquilibrage s'effectue donc «simplement». Par contre, pour ce qui est d'une suppression dans un arbre bicolore, les règles qui font intervenir des rotations (règles b_1, b_2, b_3) modifient des pointeurs de sommets non encore dupliqués, ces sommets doivent donc être dupliqués, nous verrons cela en détail un peu plus loin.

Cas d'une insertion

Soit B_m la dernière version de E et p l'élément à insérer à l'instant t_{m+1}. L'insertion se fait en deux phases.

Phase 1

La descente dans l'arbre B_m à partir de sa racine permet de créer une feuille x coloriée en rouge pour y placer p. Cette feuille appartient à B_{m+1} mais pas à B_m bien sûr. Puis on duplique tous les sommets situés sur le chemin allant de la racine à x non compris, chaque sommet dupliqué ayant un pointeur sur son fils situé dans B_m et un pointeur sur son fils nouvellement créé (figure 6.4).

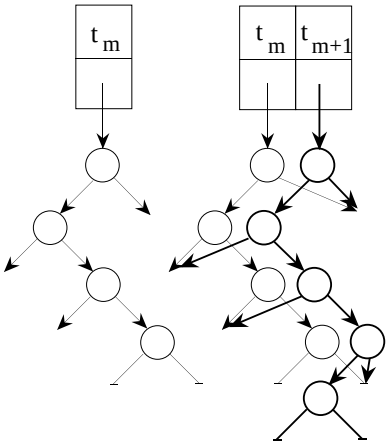


Figure 6.4: *Première phase de l'insertion.*

Expliquons l'exemple de la figure 6.5.

Nous nous autorisons ici à confondre les nœuds et leur contenu (car cela ne porte pas à conséquence), et nous parlerons d'ancien et de nouveau pour un sommet et son dupliqué. L'insertion de `J` modifie le fils droit de `I` (qui était vide). On duplique donc le fils droit de `I` en un sommet `J`, et l'on duplique tout le chemin de `J` à la racine. On a alors un nouveau chemin (`O`, `G`, `K`, `I`, `J`). Le nouvel `O` est racine de l'arbre B_{m+1} que l'on complète en donnant comme fils manquant à chaque nouveau sommet du chemin dupliqué, le fils correspondant de l'ancien sommet associé : par exemple le nouveau `K` a pour fils droit le fils droit de l'ancien `K`, i.e. `M`.

Phase 2

On a ainsi créé un nouvel arbre B_{m+1} dont la racine est la dupliquée de la racine de B_m. Il ne reste plus qu'à rééquilibrer l'arbre B_{m+1} en appliquant toujours la règle de duplication ci-dessus; mais il se trouve que dans le cas d'une insertion, le rééquilibrage ne nécessite pas de nouvelle duplication car les pointeurs modifiés dans les rotations sont issus de sommets déjà dupliqués, donc les pointeurs issus des nœuds de l'arbre B_m ne sont pas modifiés. On aura par exemple dans le cas

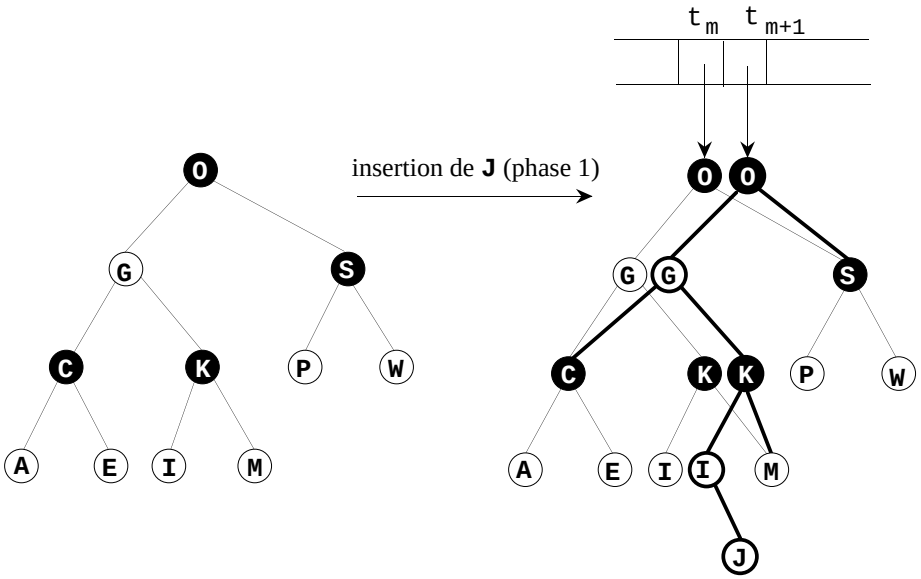


Figure 6.5: *Insertion d'un nouvel élément – phase* 1.

d'une rotation gauche le schéma suivant (les sommets dupliqués sont indiqués par des lettres primées) :

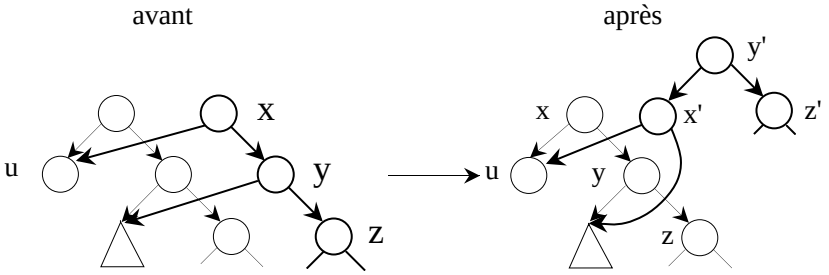


Figure 6.6: *Rééquilibrage par rotation gauche lors d'une insertion.*

La phase 2 consiste donc simplement en un rééquilibrage de l'arbre B_{m+1}. Notons que dans cette phase, la couleur d'un sommet commun à B_m et B_{m+1} peut être modifiée. Dans l'exemple donné (figure 6.7), la phase 2 consiste en un rééquilibrage de l'arbre B_{m+1} par application de la règle α_2 suivie de la règle β_2. La couleur de `M` a été modifiée, mais cela n'est pas gênant car la couleur des nœuds ne sert que dans la dernière version.

Cas d'une suppression

Soient q l'élément à supprimer dans la dernière version B_m de E et x le sommet contenant q.

Phase 1

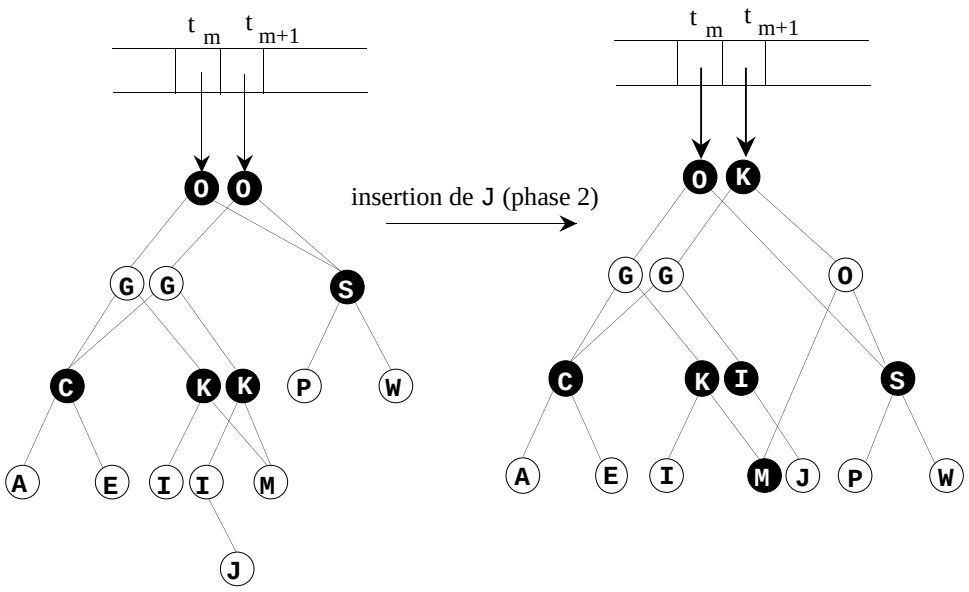


Figure 6.7: *Insertion d'un nouvel élément – phase* 2.

On applique dans un premier temps l'algorithme classique de suppression dans un arbre binaire de recherche.

Si x a un fils gauche, on détermine le sommet y contenant le prédécesseur de q, soit p. Ce sommet y n'a pas de fils droit autre qu'une feuille. On duplique tous les sommets allant de la racine à y non compris, le dupliqué de x contenant p, et chaque sommet dupliqué ayant un pointeur sur son fils qui est dans B_m et un pointeur sur son fils dupliqué. Soit z le père de y. Son dupliqué a un pointeur droit (figure 6.8 (a)) sur le fils gauche t de y si $z \neq x$, et un pointeur gauche (figure 6.8 (b)) sur le fils gauche t de y sinon. Si y est rouge, la couleur de t est inchangée, si y est noir alors si t est rouge il devient noir sinon il est *dégradé*.

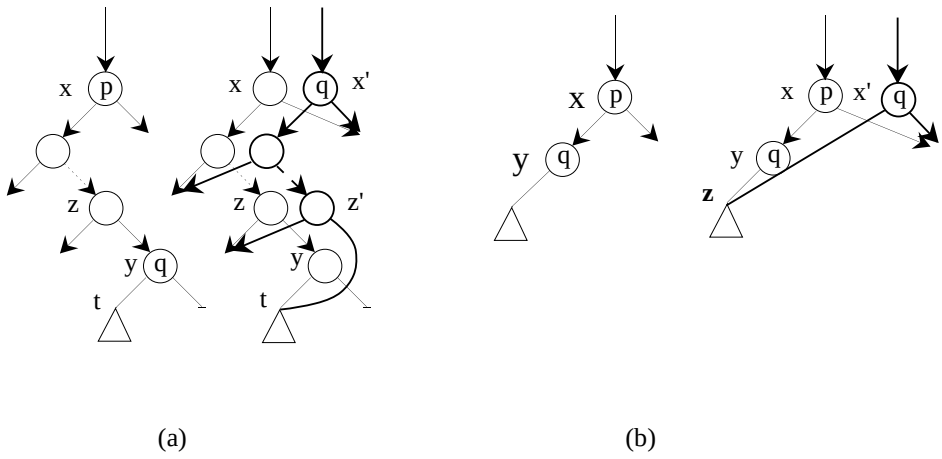


Figure 6.8: *Suppression d'un élément - phase 1.*

Si x a un fils droit, la procédure est symétrique de la précédente.

Si x n'a pas de fils (autre que des feuilles), on applique la procédure précédente où x joue le rôle de y. La figure 6.9 donne un exemple d'exécution de cette première phase.

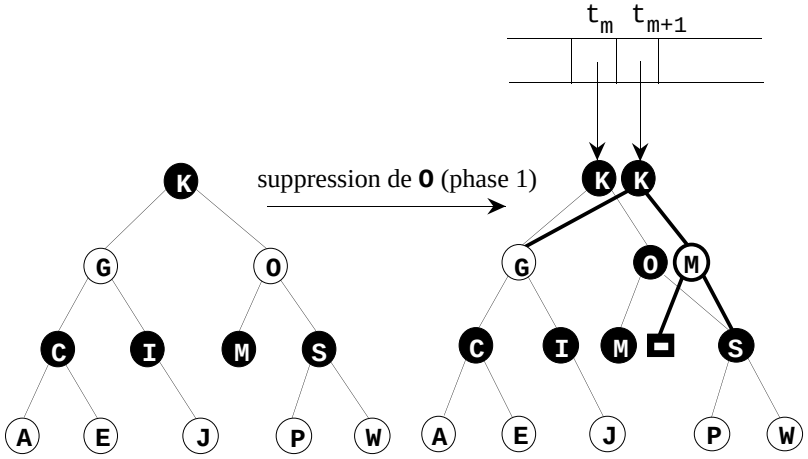


Figure 6.9: *Suppression de* `O` *- phase 1.*

Phase 2

Il reste à rééquilibrer l'arbre B_{m+1} obtenu dans la phase précédente. Mais les procédures de rééquilibrage des arbres bicolores utilisées dans la suppression modifient éventuellement des pointeurs sur des sommets de B_m qu'il faut donc dupliquer. Ainsi le rééquilibrage s'effectue selon les nouvelles règles décrites dans les figures 6.10 et 6.11.

On a représenté les sommets de B_m par un cercle et ceux de B_{m+1} qui sont des copies de sommets de B_m par un double cercle. Les nœuds affectés d'un signe $-$ sont les nœuds dégradés.

La deuxième phase de suppression appliquée à l'exemple de la figure 6.9 donne le résultat de la figure 6.12, après application de la règle (d).

Il est à noter que dans la structure décrite, chaque sommet a des pointeurs vers ses fils seulement et non vers ses pères. En effet, par nature même de l'arbre persistant, un sommet peut avoir plusieurs pères correspondant à des instants différents de l'histoire de l'ensemble persistant, ce qui rend l'implémentation de pointeurs vers les pères difficile. Donc, pour pouvoir réaliser les procédures de rééquilibrage, il est nécessaire d'avoir mémorisé le chemin dupliqué lors de la descente.

Théorème 6.1. *La structure d'arbre persistant représentant un ensemble E de taille au plus n et ayant une histoire de longueur m, utilisant la duplication de chemins permet d'effectuer chacune des opérations* INSERER(p, E) *et*

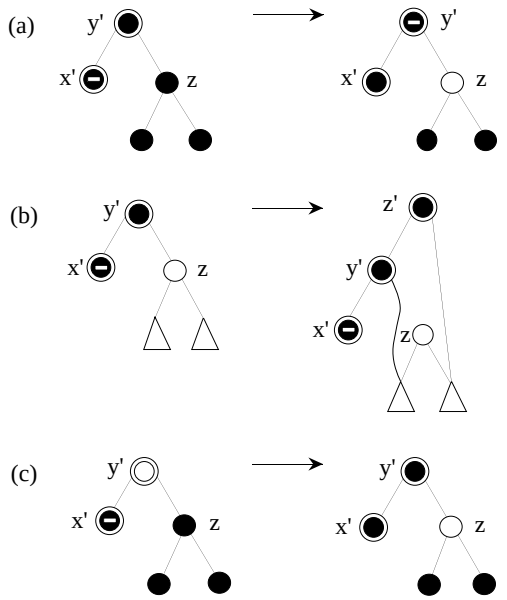


Figure 6.10: *Rééquilibrage dans une suppression, règles (a), (b), (c).*

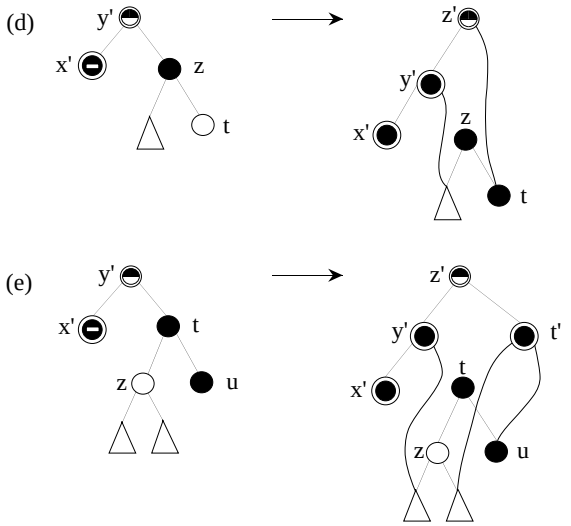


Figure 6.11: *Rééquilibrage dans une suppression, règles (d), (e).*

SUPPRIMER(p, E) *en temps et espace* $O(\log n)$ *et l'opération* CHERCHER(p, E, t_i) *en temps* $O(\log m)$. *Si la chronologie est la suite* $(0, 1, \cdots, m)$ *alors* CHERCHER (p, E, t_i) *se calcule en temps* $O(\log n)$.

Preuve. Il est clair que l'insertion ou la suppression d'un élément dans l'arbre persistant prend un temps proportionnel à celui pris par une insertion ou une suppression dans un arbre bicolore ordinaire. Quant au nombre de sommets dupliqués, il est majoré par la hauteur de l'arbre B_m augmenté dans le cas d'une suppression du nombre de rotations qui est au plus trois. Le résultat est donc prouvé pour la première assertion. Pour ce qui est de la recherche, une première recherche

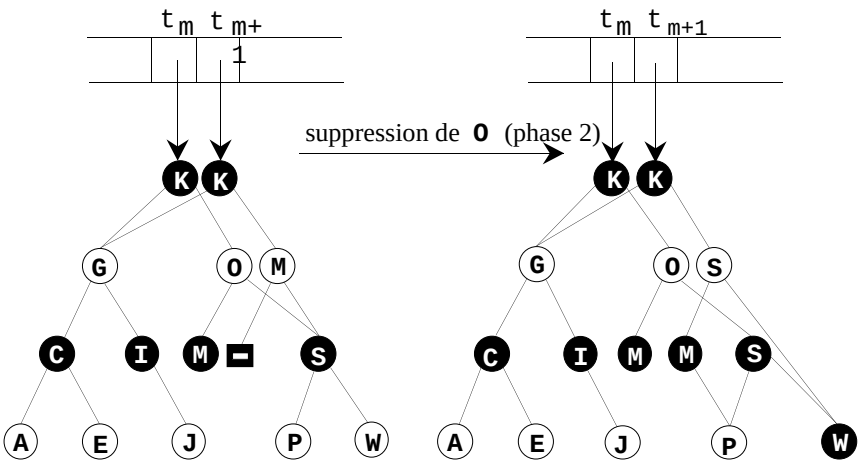


Figure 6.12: *Suppression d'un élément-phase 2 (règle (d)).*

dichotomique permet d'obtenir en temps $O(\log m)$ le pointeur sur l'arbre B_i dans lequel la recherche doit s'effectuer. Cette recherche prend alors un temps $O(\log n)$, mais $n \leq m$. Si les instants sont les entiers de 1 à m, alors on a accès en temps $O(1)$ à la racine de l'arbre B_i. ∎

6.6.3 Méthode de duplication des sommets pleins

Nous allons voir maintenant comment rendre la structure performante en espace, c'est-à-dire linéaire en fonction du nombre de mises à jour.

Il suffit de remarquer que dans une mise à jour d'un arbre bicolore ordinaire, qui peut être une insertion ou une suppression, le nombre de nœuds pour lesquels le contenu est modifié (modifications autres que celle de la couleur) et le nombre de pointeurs modifiés est $O(1)$.

La solution consiste à munir chaque sommet d'un nombre fixe k de pointeurs supplémentaires «libres» qui vont être utilisés pour une nouvelle mise à jour et éviter des duplications. Ces pointeurs sont munis d'un champ contenant l'instant auquel ils sont activés, c'est à dire auquel ils cessent d'être libres, et d'un champ précisant s'il s'agit d'un pointeur gauche ou droit (notons qu'on peut se passer de ce dernier champ car la nature gauche ou droite du pointeur peut se déterminer en comparant les clés des sommets reliés par ce pointeur, mais ce qu'on gagne en place d'un côté est perdu en temps de l'autre). Lorsque tous les pointeurs libres d'un nœud sont activés (le sommet est alors «plein») et qu'un pointeur du nœud doit être modifié, ou lorsque le contenu du sommet lui-même est modifié, alors seulement le sommet est dupliqué en un nouveau sommet qui dispose à nouveau de k pointeurs libres.

Nous allons détailler la structure ainsi obtenue pour $k = 1$; elle ne diffère pas dans son principe du cas général. Nous renvoyons le lecteur intéressé par le problème

général aux notes de fin de chapitre.

Chaque sommet est donc muni en tout de trois pointeurs, possédant un champ qui indique s'il s'agit d'un pointeur gauche ou droit, et d'un champ indiquant la date d'activation du pointeur qu'on appellera simplement date du pointeur. A un instant donné tout sommet a au moins deux pointeurs activés, un gauche et un droit, le troisième pouvant être activé ou libre. Ce troisième pointeur est représenté sur les figures par une flèche lorsqu'il est libre et débute par un triangle bicolore dont le côté noir indique s'il s'agit d'un pointeur gauche ou droit, ceci pour une plus grande lisibilité des figures

Compte tenu de la nouvelle structure des sommets, nous décomposons les procédures d'insertion et de suppression persistantes en opérations élémentaires que nous décrivons ci-dessous.

Phase 1

Dans la phase 1, les opérations élémentaires réalisées sont

(1) descente dans un sommet (lors d'une insertion, d'une recherche, ou d'une suppression) ;
(2) modification du contenu d'un sommet (lors d'une suppression ou d'une insertion) ;
(3) modification d'un pointeur (lors d'une duplication).

(1) Descente dans un sommet

Soit p l'élément à insérer ou à supprimer dans la version la plus récente, ou que l'on recherche dans la version B_k à l'instant t_k.

Si x est le sommet courant (au départ x est la racine de l'arbre correspondant à l'instant voulu), l'opération générique est la suivante :

– s'il s'agit d'une mise à jour, descendre dans la direction appropriée par le pointeur le plus récent ;

– s'il s'agit d'une recherche, descendre dans la direction appropriée par le pointeur le plus récent parmi ceux qui dans cette direction ont une date d'activation inférieure ou égale à t_k.

Exemple. Supposons que l'on cherche à déterminer si l'élément `F` figure dans la version B_4. Arrivée au sommet contenant `L` qui est plein, la descente doit se poursuivre à gauche ; or il y a deux pointeurs gauches, l'un activé à l'instant 3, l'autre à l'instant 7 ; la descente à gauche se fait donc vers `C` par le pointeur de date 3 (figure 6.13).

(2) Modification du contenu d'un sommet

Cette opération est réalisée lors d'une suppression ou d'une insertion à l'instant t_{m+1}. Rappellons qu'on ne considère pas comme modification du contenu la modification de la couleur d'un sommet.

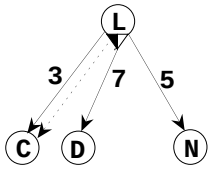


Figure 6.13: *Recherche de* `F` *à l'instant 4.*

On duplique le sommet x en un sommet x' ayant même couleur et ayant un nouveau contenu. Le pointeur gauche (respectivement droit) de x' pointe sur le même sommet que le pointeur gauche (respectivement droit) de x le plus récent. Ces deux pointeurs ont pour date d'activation t_{m+1}. Le sommet x' a un pointeur libre (figure 6.14). Si x n'est pas une racine, alors y père de x dans la dernière version doit subir l'opération modification de pointeur décrite ci-après .

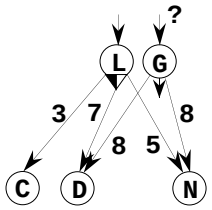


Figure 6.14: *Le sommet contenant* `L` *a éte dupliqué pour contenir* `G`*.*

(3) Modification d'un pointeur

Soit x un sommet dont un des fils y (le fils gauche par exemple) est dupliqué au cours de la mise à jour à l'instant t_{m+1}.

Si x n'est pas plein alors le pointeur libre de x est activé, pointe à gauche sur le nouveau fils y' et a pour date d'activation t_{m+1} (figure 6.15 (a)).

Si x est plein, on duplique x en un sommet x' dont le pointeur gauche pointe sur y' et le pointeur droit sur le fils droit le plus récent de x, ces deux pointeurs ont pour date d'activation t_{m+1}. Si x n'est pas une racine, alors le père de x doit à nouveau subir l'opération de modification de pointeur (figure 6.15 (b)).

Phase 2

La phase 2 procède au rééquilibrage de la nouvelle version. Il suffit ici de préciser comment s'effectue l'opération élémentaire de rotation. Dans une rotation, certains pointeurs changent de destination. Si leur date d'activation est précisément la date de mise à jour, on change leur destination, en gardant leur date d'activation. Par contre si leur date d'activation est antérieure à la date de mise à jour, alors il faut faire un traitement spécial, i.e. appliquer la règle **(3)** de la phase 1. Prenons le cas d'une rotation droite, les autres cas s'en déduisent. L'algorithme

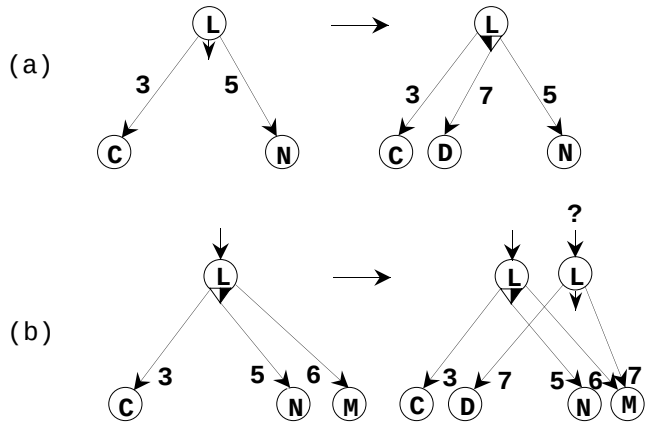


Figure 6.15: *Le fils gauche de* `L` *est dupliqué à l'instant* t_7.

à appliquer est donné dans les figures 6.16 (le nœud z n'est pas plein) et 6.17 (le nœud z est plein).

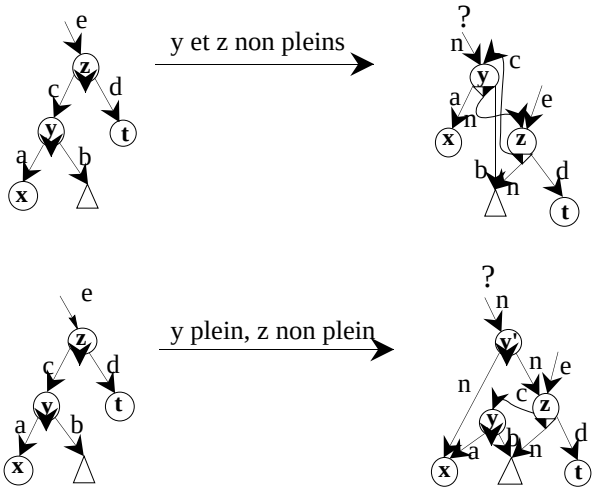


Figure 6.16: *Cas où z n'est pas plein.*

Ne sont indiqués à gauche que les pointeurs les plus récents dans une direction donnée. Les nouveaux pointeurs sont en traits épais. Soit n la date de mise à jour, et d la date d'activation du pointeur sur le nœud z avant rééquilibrage. L'exemple est traité dans le cas où toutes les dates d'activation sont strictement inférieures à n. Le point d'interrogation signifie qu'il faut appliquer au père de l'ancien z dans l'arbre gauche le traitement «modification de pointeur», qui donnera un résultat différent selon que ce père est plein ou non. Ainsi, si le nœud z (resp. y) est plein, z (resp. y) est dupliqué en z' (resp. y'). Si certaines dates autres que d sont égales à n, l'algorithme est plus simple car les pointeurs de date n ont simplement à changer de destination s'il y a lieu. Si $d = n$, il suffit de supprimer dans tous les

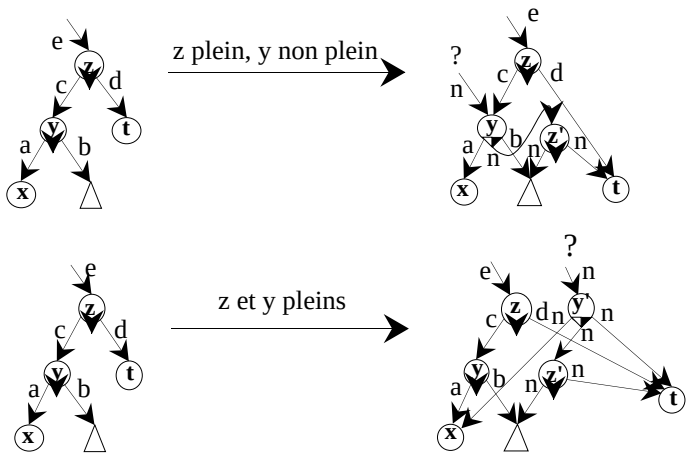


Figure 6.17: *Cas où z est plein.*

arbres obtenus à droite le pointeur sur z de date d.

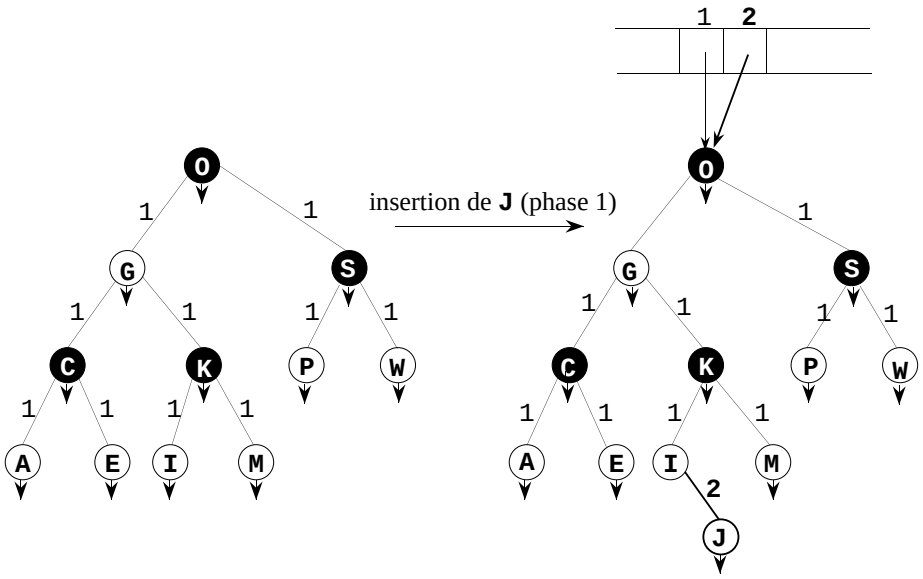


Figure 6.18: *Insertion d'un nouvel élément-phase 1.*

Les figures 6.18, 6.19, 6.20traitent l'exemple des figures 6.5, 6.7, 6.9, 6.12 par la méthode de duplication des nœuds pleins, puis nous donnons les résultats de la suite d'opérations «insertion de J, suppression de O» par la méthode de duplication de chemin (a), et par la méthode de duplication des nœuds pleins (b) pour l'arbre initial de la figure 6.5, en appelant les instants successifs 1, 2, 3 (figure 6.21).

Il nous reste à calculer la complexité en temps et en espace d'une mise à jour dans cette nouvelle structure de données.

Il est clair que la complexité en temps d'une insertion, suppression ou recherche est équivalente à celle de la structure plus simple précédemment étudiée, et reste

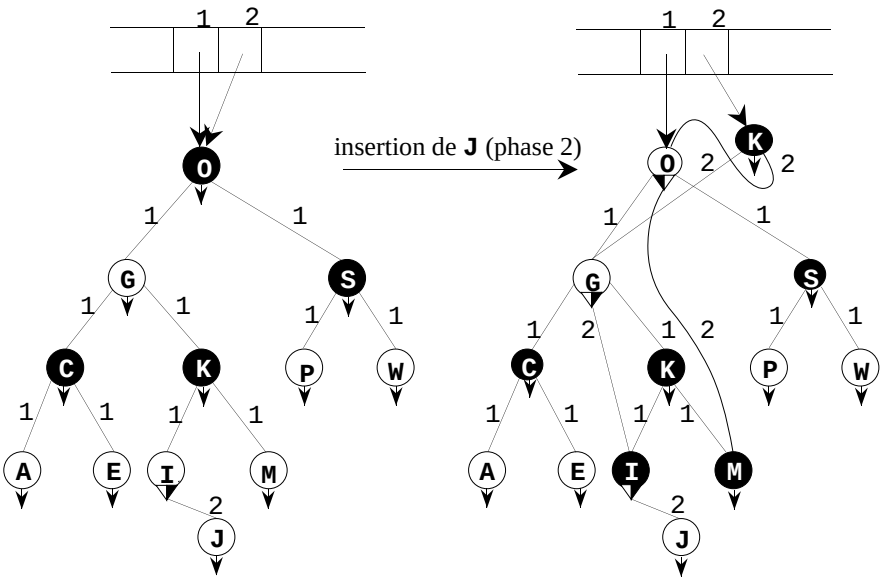


Figure 6.19: *Insertion d'un nouvel élément-phase 2.*

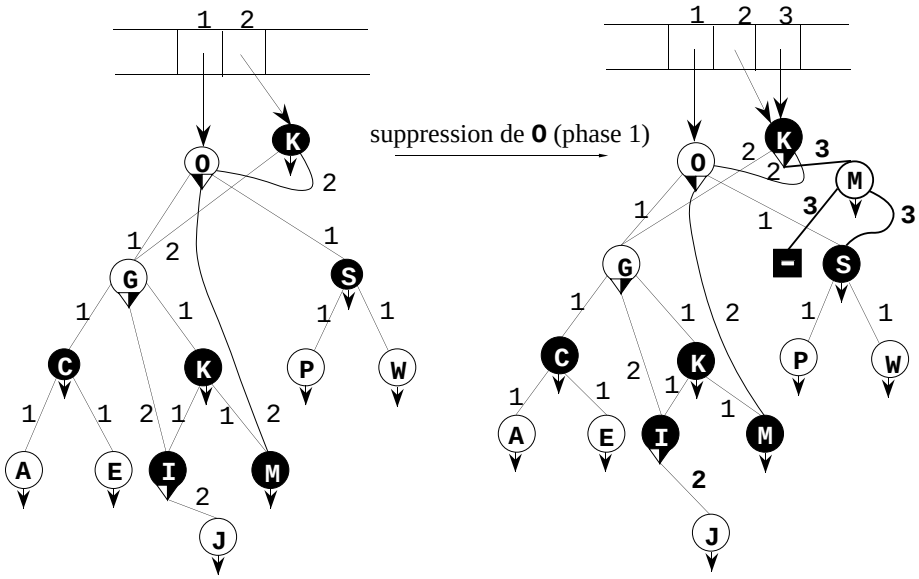


Figure 6.20: *Suppression d'un élément-phase 1.*

$O(\log m)$.

Complexité amortie en espace d'une mise à jour

Nous allons définir un *potentiel* pour chaque état de la structure de données qui va nous permettre de calculer la complexité amortie en espace d'une mise à jour.

Soit t_m l'instant de la dernière mise à jour de l'ensemble persistant E. On appelle *sommet actif* tout sommet accessible à partir de la racine de l'arbre B_m par un

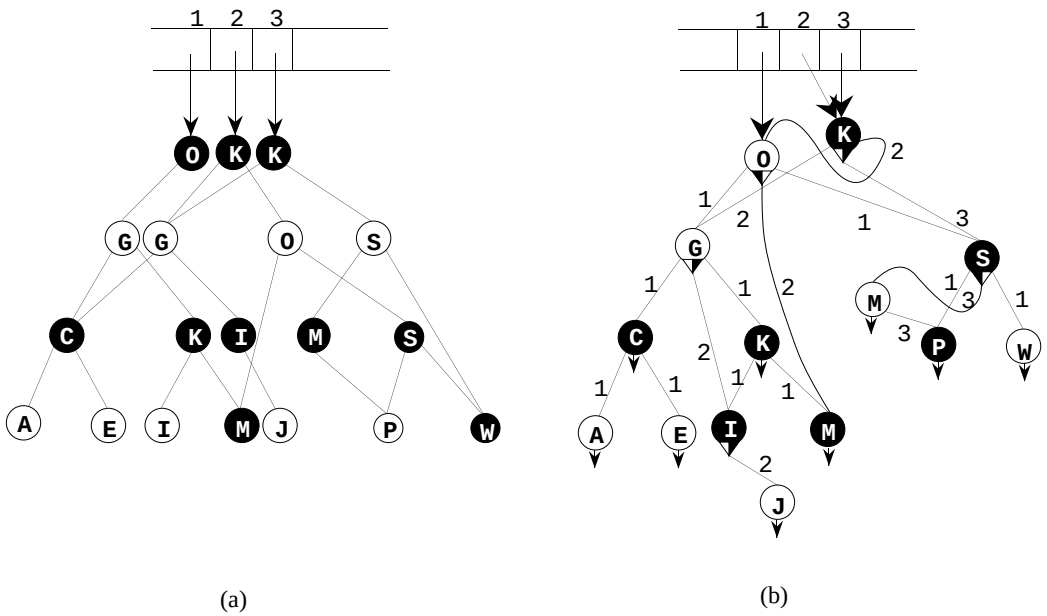


Figure 6.21: *Comparaison des deux méthodes.*

pointeur valide, i.e. de date d'activation la plus récente dans la direction correspondante (ce sont les sommets de l'arbre B_m). Les sommets non actifs sont dits *passifs*. Au cours du temps, un sommet passif reste passif, un sommet actif peut devenir passif. Le *potentiel* P_m de E à l'instant t_m (instant de la dernière mise à jour) est le nombre α_m de sommets actifs diminué du nombre λ_m de pointeurs libres dans les sommets actifs :

$$P_m = \alpha_m - \lambda_m \text{ pour } m > 0$$

Initialement, le potentiel P_0 de la structure vide est nul. Comme chaque sommet a au plus un pointeur libre, on a clairement : $\alpha_m \geq \lambda_m$ et donc $P_m \geq 0$. On définit le *coût amorti en espace* γ_m de la mise à jour à l'instant t_m comme étant le nombre (éventuellement négatif) c_m de sommets créés par l'action a_m, augmenté de la différence de potentiel de E entre l'instant t_m et t_{m-1} :

$$\gamma_m = c_m + (P_m - P_{m-1})$$

Ainsi le nombre de sommets créés au cours de l'histoire $(a_{t_0}, \cdots, a_{t_m})$ de E est :

$$\sum_{i=1}^{m} c_i = \sum_{i=1}^{m} \gamma_i - \sum_{i=1}^{m} (P_i - P_{i-1}) = \sum_{i=1}^{m} \gamma_i - P_m \leq \sum_{i=1}^{m} \gamma_i$$

Donc le coût total en espace est majoré par la somme des coûts amortis de chaque mise à jour.

Calculons la complexité amortie d'une mise à jour, en évaluant la complexité amortie de chaque opération élémentaire. Dans le tableau ci-dessous sont évaluées

successivement les quantités c_m, $\Delta(\alpha_m)$, $\Delta(\lambda_m)$, γ_m (où Δ représente la variation de la quantité exprimée), pour chaque opération élémentaire :

- la création d'un nœud qui s'effectue au cours d'une insertion : c_m augmente d'une unité, ainsi que α_m et λ_m car le nœud créé a un pointeur libre;
- la suppression d'un nœud qui s'effectue au cours d'une suppression : c_m diminue d'une unité ainsi que α_m, quant à λ_m le résultat varie selon que le nœud supprimé était plein ou non;
- la duplication d'un nœud (ne prend en compte que la duplication simple sans la création de pointeur vers le nœud dupliqué qu'elle implique) : c_m augmente d'une unité, α_m ne varie pas car le nœud qui a été dupliqué est devenu passif, λ_m augmente d'une unité car le nouveau nœud a un pointeur libre;
- l'activation d'un pointeur qui rend un nœud plein : c_m et α_m ne changent pas, par contre λ_m diminue d'une unité;
- la rotation simple : là encore, c_m et α_m ne changent pas, et la variation du nombre de pointeurs libres dépend des cas de figure selon que les nœuds concernés par la rotation sont pleins ou non.

opération élémentaire	$\Delta(c_m)$	$\Delta(\alpha_m)$	$\Delta(\lambda_m)$	γ_m
création d'un nœud	+1	+1	+1	+1
suppression d'un nœud	-1	-1	0 ou -1	-2 ou -1
duplication d'un nœud	+1	0	+1	0
activation d'un pointeur	0	0	-1	+1
rotation simple	0	0	de -1 à +3	de -1 à +3

Une insertion a pour opérations élémentaires de coût amorti non nul : une création de sommet, une activation de pointeur (si le sommet créé n'est pas la racine) et au plus deux rotations. Son coût amorti est donc $O(1)$.

Une suppression a pour opérations élémentaires de coût amorti non nul : une suppression de sommet, deux activations de pointeurs (une pour la modification de contenu du sommet contenant l'élément à supprimer, l'autre faisant suite à la suppression d'un sommet), et au plus trois rotations au cours du rééquilibrage. Le coût amorti d'une suppression est donc aussi $O(1)$.

Théorème 6.2. *La structure d'arbre persistant représentant un ensemble E de taille au plus n et ayant une histoire de longueur m, utilisant la méthode de duplication des nœuds pleins permet d'effectuer chacune des opérations* Inserer (p, E) *et* Supprimer(p, E) *en temps $O(\log n)$ et l'opération* Chercher (p, E, t_i) *en temps $O(\log m)$. Le coût amorti en espace d'une mise à jour est $O(1)$. Si les instants sont représentés par les entiers $1, 2, \cdots, m$, alors* Chercher (p, E, t_i) *se calcule en temps $O(\log n)$.*

Notes

La littérature sur les arbres de recherche est très vaste. De nombreux compléments figurent notamment dans :

K. Mehlhorn, *Data Structures and Algorithms Vol. 1*, Springer-Verlag, 1984.

G. H. Gonnet, R. Baeza-Yates, *Handbook of Algorithms and Data Structures*, Addison-Wesley, deuxième édition, 1991

contient des programmes en Pascal et en *C*. Une synthèse sur les structures de données avec une abondante bibliographie, est

K. Mehlhorn, A. Tsakalidis, Data structures, in : J. van Leeuwen, ed., *Handbook of Theoretical Computer Science, Vol. A*, North-Holland, 1990, 301–341.

Les structures de données persistantes sont traitées, avec de nombreuses variantes, dans l'article original de leurs inventeurs :

J. Driscoll, N. Sarnak, D. Sleator, R. Tarjan, Making data structures persistent, *J. Comput. Syst. Sci.* **38** (1989), 86–124.

Les files binomiales sont dues à J. Vuillemin ; un exposé détaillé est donné dans :

T.H. Cormen, C.E. Leiserson, R.L. Rivest, *Introduction to Algorithms*, MIT Press, McGraw-Hill, 1990.

Les arbres évasés ont été introduits et étudiés par Sleator et Tarjan. On peut consulter à ce propos le livre de Mehlhorn.

Exercices

6.1. Décrire les opérations de dictionnaire sur les arbres binaires de recherche balisés, et réaliser les programmes correspondants.

6.2. Décrire la procédure qui permet de passer d'un arbre binaire balisé à un arbre binaire de recherche.

6.3. Développer les opérations de dictionnaire sur les arbres *AVL* balisés.

6.4. Décrire les opérations de passage entre arbres de recherche et arbres de recherche balisés dans le cas d'arborescences ordonnées. Appliquer ces opérations pour traduire les algorithmes des arbres *a–b* en leur version non balisée.

6.5. Soit A un arbre 2–3 de hauteur h, dont tous les nœuds ont 3 fils. Soit c une clé strictement plus grande que les clés figurant dans A.

a) Montrer que l'insertion de c dans A, suivie de la suppression de c, redonne l'arbre A.

b) En déduire qu'une suite de n insertions/suppressions de c dans A exige $O(nh)$ opérations de rééquilibrage, et en conclure que le coût amorti du rééquilibrage dans un arbre 2–3 n'est pas constant.

6.6. Un *arbre à pointeur* («finger tree») est un arbre balisé muni d'un pointeur vers une feuille. Montrer que dans un arbre à pointeur à liaisons par niveau, la recherche d'une clé qui est à distance d de la feuille pointée peut se faire en temps $O(\log d)$.

6.7. On considère un arbre binaire de recherche A qui, pour chaque sommet, dispose d'un pointeur vers son père, ainsi que du nombre de sommets dans son sous-arbre. Montrer comment on peut calculer son numéro d'ordre à partir de ces informations.

6.8. On peut implémenter les arbres 2–4 au moyen d'arbres bicolores balisés comme suit : à chaque nœud d'un arbre 2–4 on associe un nœud noir s'il a 2 fils, un nœud noir dont un fils est blanc s'il a trois fils, et un nœud noir dont les deux fils sont blancs s'il a 4 fils (figure 6.22).

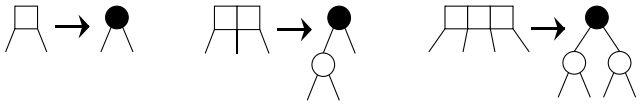


Figure 6.22: *Transformation d'un arbre* 2–4 *en arbre bicolore.*

a) Décrire comment les opérations d'éclatement, de fusion et de partage se transportent sur les arbres bicolores.

b) Décrire la réalisation de l'algorithme de scission sur les arbres bicolores.

6.9. Un *arbre* 1–2 *fraternel* («1–2 brother tree») est un arbre dont tous les nœuds ont 1 ou 2 fils, et tel qu'un nœud à 1 fils possède un frère qui a deux fils. De plus, toutes les feuilles sont à la même profondeur.

a) Montrer que si l'on supprime les nœuds à 1 fils dans un arbre fraternel, on obtient un arbre *AVL*, et montrer que cette correspondance est une bijection des arbres 1–2 fraternels sur les arbres *AVL*.

b) Un arbre fraternel de recherche est un arbre fraternel dont les sommets, sauf ceux ayant un fils unique, sont munis d'une clé, les clés étant croissantes en ordre symétrique. Décrire et implémenter les opérations de dictionnaire en temps logarithmique sur les arbres fraternels.

6.10. Un *arbre évasé* («splay tree») est un arbre binaire de recherche A muni d'une opération définie comme suit :

EVASER(x, A) ;
: donne un arbre A' qui représente le même ensemble de clés que A ; si x est une clé qui figure dans A, alors x est la clé de la racine de A' ; sinon, la racine de A' est soit x^- soit x^+, où x^- est la plus grande clé dans A inférieure à x, et x^+ est la plus petite clé supérieure à x.

Cette opération est réalisée en localisant, par une descente classique dans A, le sommet contenant x (ou x^- ou x^+ si x ne figure pas dans A). Ce sommet est

ensuite «remonté» vers la racine en appliquant des doubles rotations gauche-gauche, gauche-droite, droite-gauche, droite-droite (et éventuellement une simple rotation à la fin).

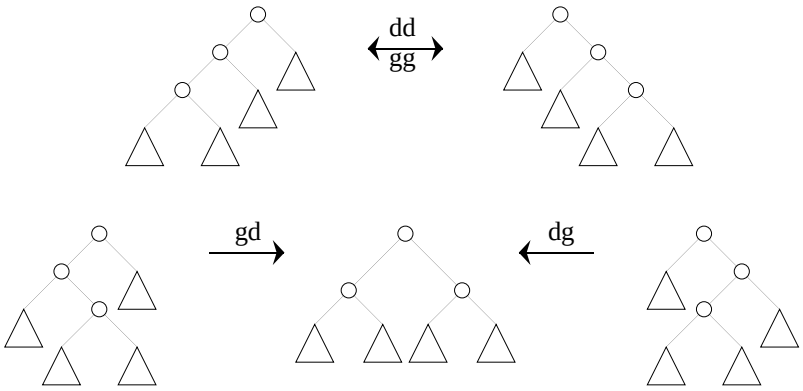


Figure 6.23: *Les quatre double-rotations.*

a) Montrer comment réaliser les opérations INSÉRER(x, A) et SUPPRIMER(x, A) en temps constant lorsqu'elles sont précédées de EVASER(x, A).

b) Montrer que CONCATÉNER(S_1, S_2, S_3) ainsi que SCINDER(S, x, S_1, S_2) se réalisent également en temps constant par des arbres évasés si elles sont précédées d'un appel approprié d'EVASER.

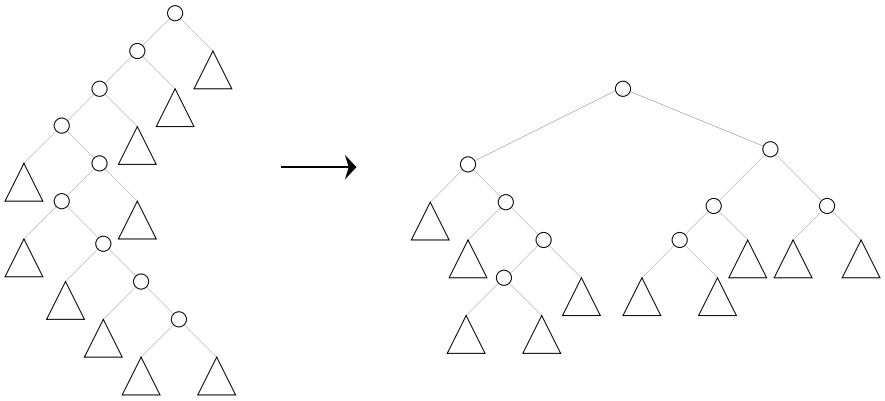


Figure 6.24: *Effet de l'opération* EVASER(a, A).

On suppose que chaque clé x est munie d'un *poids* positif $p(x)$.

c) Pour tout sommet s, on appelle *rang* de s le nombre $r(s) = \log n(s)$, où $n(s)$ est la somme des poids des clés dans le sous-arbre de racine s, et on définit le *potentiel* de A comme la somme des rangs des sommets de A. Montrer que le coût amorti de l'opération EVASER(x, A), relativement au potentiel, est majoré par

$1 + 3(r(A) - r(s))$, où $r(A)$ est le rang de la racine de A, et où s est le sommet localisé par la descente dans A.

d) On considère une suite de m opérations EVASER dans un arbre à n sommets; montrer que le coût total de ces opérations est majoré par $O((m+n)\log(m+n))$.

e) Pour chaque clé x, soit $q(x)$ le nombre de fois où l'évasion porte sur x. Montrer qu'alors le temps total est majoré par

$$O\left(m + \sum_{i=1}^{n} q(x) \log \frac{m}{q(x)}\right)$$

6.11. (Tri adaptatif.) On considère une suite $x_1, \ldots, x_n$ de nombres distincts à trier en ordre croissant; on suppose la suite «presque triée», c'est-à-dire que le *nombre d'inversions* $F = \text{Card}\{(i,j) \mid i < j \text{ et } x_i > x_j\}$ est «petit». Plus précisément, on demande de prouver que l'algorithme ci-dessous trie la suite en temps $O(n + n\log((1+F)/n))$.

L'algorithme insère successivement $x_n, x_{n-1}, \ldots, x_1$ dans un arbre 2–4 initialement vide. Pour l'insertion de x_i, on remonte la branche gauche de l'arbre contenant déjà $x_{i+1}, \ldots, x_n$ jusqu'à rencontrer un nœud s_i dont la balise gauche est supérieure à x_i, puis on insère x_i dans le sous-arbre gauche de s_i.

a) Montrer que la hauteur de s_i est $O(\log(1+f_i))$, où $f_i = \text{Card}\{j > i \mid x_i > x_j\}$.

b) En déduire que le coût total de l'algorithme est majoré par $O(\sum_{i=1}^{n} \log(1 + f_i) + E)$, où E est le nombre total d'éclatements de nœuds, et conclure.

6.12. (Arbres binomiaux.) On définit une suite $(B_n)_{n\geq 0}$ d'arborescences ordonnées par récurrence sur n comme suit : B_0 est l'arbre (arborescence) à un seul sommet; B_{n+1} est formé de la réunion de deux copies disjointes $B_n^{(g)}$ et $B_n^{(d)}$ de l'arbre B_n; sa racine est la racine de $B_n^{(d)}$, et il y a un arc de la racine de $B_n^{(d)}$ vers la racine de $B_n^{(g)}$. Cet arc est le plus petit (le plus à gauche) des arcs issus de la racine. L'arbre B_n est l'*arbre binomial d'ordre n*.

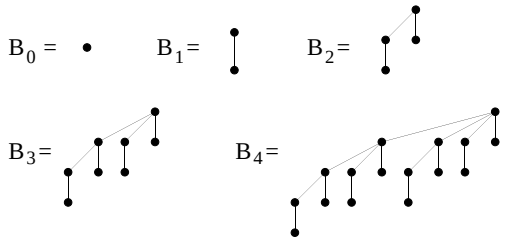


Figure 6.25: *Arbres binomiaux.*

a) Démontrer que dans B_n, il y a exactement $\binom{n}{k}$ sommets de profondeur k.

b) Démontrer qu'un seul sommet de B_n a n fils, et que 2^{n-k-1} sommets ont k fils pour $0 \leq k < n$.

On numérote les sommets de B_n de 0 à $2^n - 1$ de la gauche vers la droite en ordre postfixe (fils avant le père).

c) Montrer qu'un sommet est de profondeur $n - k$ si et seulement si son numéro a k "1" en écriture binaire.

d) Montrer que le nombre de fils d'un sommet est égal au nombre de "1" qui suivent le dernier "0" dans l'écriture binaire de son numéro.

Etant donné un entier n, soit

$$n = \sum_{i \geq 0} b_i 2^i, \qquad b_i \in \{0, 1\}$$

sa décomposition en base 2, soit $I_n = \{i \mid b_i = 1\}$, et soit $\nu(n) = \text{Card}(I_n)$. La *forêt binomiale* d'ordre n est l'ensemble $F_n = \{B_i \mid i \in I_n\}$. La *composante d'indice* i de F_n est B_i, si $i \in I_n$, et $\emptyset$ sinon.

e) Montrer que F_n a $n - \nu(n)$ arêtes.

Une *file binomiale* est une forêt binomiale dont chaque sommet x est muni d'une clé $c(x)$, élément d'un ensemble totalement ordonné, vérifiant : si y est fils de x, alors $c(y) > c(x)$.

f) Montrer que la recherche du sommet de clé minimale dans une file binomiale F_n peut se faire en $\nu(n) - 1$ comparaisons.

6.13. (Suite.) Soient F_n et $F_{n'}$ deux files binomiales ayant des ensembles de clés disjointes. On définit l'opération

$$\textsc{Union}(F_n, F_{n'})$$

qui retourne une file binomiale ayant pour clés l'union des ensembles de clés comme suit :

(1) Si $n = n' = 2^p$, alors $F_n = \{B_p\}$, $F_{n'} = \{B'_p\}$, et $\textsc{Union}(B_p, B'_p)$ est l'arbre B_{p+1} pour lequel $B_p^{(g)} = B_p$ et $B_p^{(d)} = B'_p$ si la clé de la racine de B_p est plus grande que la clé de la racine de B'_p; et $B_p^{(g)} = B'_p$ et $B_p^{(d)} = B_p$ dans le cas contraire.

(2) Dans le cas général, on procède en commençant avec les composantes d'indice minimal des deux files, et en construisant une suite d'*arbres report*. L'arbre report R_0 est vide, et à l'étape $k > 0$, l'arbre report R_k est soit vide, soit un arbre binomial d'ordre k. Etant donnés l'arbre report R_k, la composante C_k de F_n et la composante C'_k de $F_{n'}$, on définit la composante C''_k d'ordre k de $\textsc{Union}(F_n, F_{n'})$ et l'arbre report R_{k+1} comme suit :
(1) Si $R_k = C_k = C'_k = \emptyset$, alors $C''_k = R_{k+1} = \emptyset$;
(2) Si l'un exactement des trois opérandes est non vide, il devient C''_k, et $R_{k+1} = \emptyset$;
(3) Si deux opérandes sont non vides, alors $C''_k = \emptyset$ et R_{k+1} est l'UNION des deux opérandes;

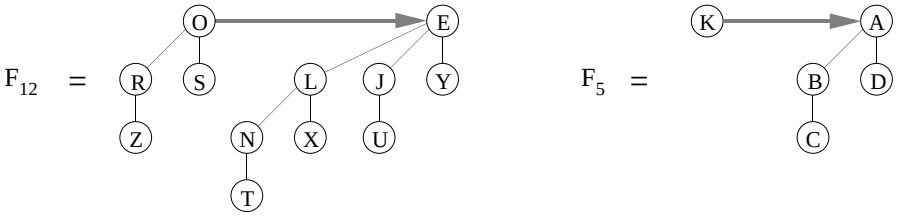


Figure 6.26: *Deux files binomiales,*

(4) Si les trois opérandes sont non vides, l'un devient C''_k, et R_{k+1} est l'UNION des deux autres.

Par exemple, l'union des deux files binomiales de la figure 6.26 donne la file de la figure 6.27.

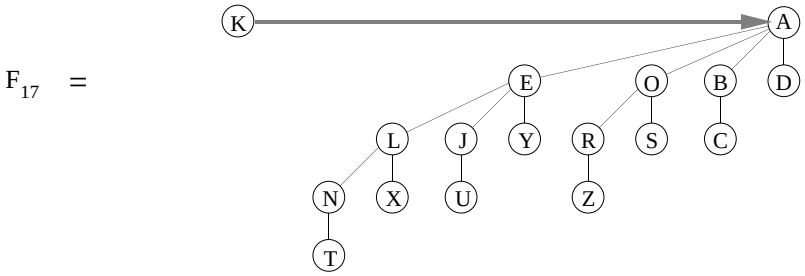


Figure 6.27: ... *et leur union.*

a) En supposant que l'UNION de deux arbres binomiaux de même ordre prend un temps constant, donner une majoration logarithmique en n et n' du temps de UNION$(F_n, F_{n'})$.

b) Proposer une structure de données qui permet de faire l'union de deux arbres binomiaux en temps constant.

c) Montrer que le nombre de comparaisons de clés pour construire UNION$(F_n, F_{n'})$ est $\nu(n)+\nu(n')-\nu(n+n')$. (On pourra prouver que ce nombre est égal au nombre de reports dans l'addition usuelle en base 2 de n et de n'.)

d) Montrer comment l'insertion, dans une file binomiale, d'une clé qui n'y figure pas déjà peut se ramener à une union.

e) En déduire un algorithme de construction d'une file binomiale pour un ensemble de n clés, et montrer que cela demande au total $n - \nu(n)$ comparaisons.

f) Montrer que la suppression de la plus petite clé dans une file binomiale F_n peut se faire en $O(\log n)$ opérations, y compris les opérations nécessaires pour reconstruire une file binomiale F_{n-1} pour les clés restantes.

g) En déduire un algorithme de tri d'une suite de n clés en temps $O(n \log n)$.

6.14. Appelons *arbre relativement équilibré* un arbre binaire de recherche tel que le facteur d'équilibre en tout sommet soit compris entre -2 et $+2$. Soit N_h le nombre de sommets minimal d'un arbre relativement équilibré de hauteur h.

a) Montrer que la suite $(N_h)_{h\in\mathbb{N}}$ satisfait une équation de récurrence linéaire que l'on résoudra.

b) En déduire que la hauteur d'un arbre relativement équilibré à n sommets est $\theta(\log n)$.

c) Elaborer des algorithmes d'insertion, suppression et recherche qui se réalisent en temps $O(\log n)$, où n est le nombre de sommets de l'arbre.

6.15. Construire une variante des arbres bicolores en remplaçant la condition «tout sommet blanc a un père noir» par «tout sommet blanc dont le père est blanc a un grand-père noir».

6.16. On considère la version des arbres persistants avec recopie des nœuds pleins. On modifie la nature des nœuds de la façon suivante : chaque nœud possède non pas un seul pointeur supplémentaire mais k pointeurs supplémentaires. Donner les algorithmes d'insertion, suppression et recherche adaptés, et analyser leur complexité.

6.17. Montrer que la méthode de duplication de chemins des arbres persistants donnée dans ce chapitre s'adapte aux arbres AVL et donner les algorithmes d'insertion, suppression, recherche correspondants.

Chapitre 7

Graphes valués

Ce chapitre traite deux problèmes fondamentaux de l'optimisation combinatoire. La première section est consacrée à la recherche d'un arbre couvrant de coût minimum dans un graphe valué non orienté. Nous présentons un algorithme très général fondé sur une règle des cycles et une règle des cocycles puis nous développons deux implémentations particulières conduisant aux algorithmes de Kruskal et de Prim. La seconde section décrit la recherche des chemins de coût minimum issus d'un sommet dans un graphe orienté valué. Nous exposons l'itération fondamentale de Ford, l'algorithme de Dijkstra pour des coûts positifs et sa variante A∗, l'algorithme de Bellman pour un graphe sans circuit et enfin l'algorithme Paps *lorsque aucune hypothèse particulière n'est faite sur le graphe valué.*

Introduction

La notion d'arbre de coût minimum intervient chaque fois que l'on doit relier au moindre coût des objets entre eux de telle sorte que tous les objets soient connectés directement ou non. La notion de chemin de coût minimum intervient chaque fois que l'on doit trouver un cheminement orienté d'un point à un autre au moindre coût. Ces deux problèmes sont bien résolus par des algorithmes polynomiaux efficaces qui utilisent des propriétés structurelles fortes des solutions optimales. Ces propriétés concernent les cycles et les cocycles pour le problème de l'arbre couvrant de coût minimum, les arborescences partielles pour le problème des chemins de coût minimum.

7.1 Arbre couvrant de coût minimum

7.1.1 Définition du problème

Soit $G = (S, A)$ un graphe non orienté connexe et $c : A \mapsto \mathbb{R}$ une valuation de ses arêtes. Si $H = (S, F)$ est un arbre couvrant de G, son *coût* noté $c(H)$ est défini par $\sum_{f \in F} c(f)$. Le problème est de déterminer dans l'ensemble non vide (car G est connexe) des arbres couvrants de G un arbre couvrant de coût minimum. La figure 1.1 représente un énoncé du problème qui nous servira à illustrer les propriétés et les algorithmes qui vont suivre.

Pour simplifier l'écriture, nous identifierons dans ce chapitre un graphe partiel (S, U) de G avec l'ensemble U de ses arêtes, on parlera alors du graphe partiel U.

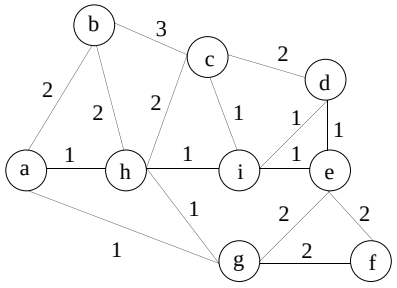


Figure 1.1: *Un graphe valué.*

7.1.2 Propriétés des arbres optimaux

On appelle *approximant* d'un arbre optimal un couple (X, Y) formé par deux sous-ensembles d'arêtes X et Y tels qu'il existe un arbre couvrant de coût minimum H contenant X et disjoint de Y. L'ensemble X contient les arêtes *admises*, l'ensemble Y contient les arêtes *écartées* et l'ensemble $Z = A - (X \cup Y)$ contient les arêtes *libres*. Etant donné un approximant (X, Y), nous allons montrer que :

a) l'existence dans G d'un *cycle candidat* ne contenant aucune arête de Y permet d'écarter une arête supplémentaire appartenant au cycle, c'est-à-dire de déterminer un meilleur approximant (X, Y') où Y' contient une arête de plus que Y.

b) l'existence dans G d'un *cocycle candidat* ne contenant aucune arête de X permet d'admettre une arête supplémentaire appartenant au cocycle, c'est-à-dire de déterminer un meilleur approximant (X', Y) où X' contient une arête de plus que X.

Une fois ces propriétés établies, la construction d'un arbre couvrant de coût minimum sera réalisée en maintenant un approximant et en l'améliorant à chaque itération. La figure 1.2 illustre les notions de cycle et de cocycle candidats pour un

approximant (X, Y) où les arêtes de X sont épaisses et les arêtes de Y en pointillé. Le cocycle $\omega(\{e\})$ est candidat, donc $X' = X \cup \{\{e, d\}\}$. Le cycle (h, b, c, h) est candidat, donc $Y' = Y \cup \{\{b, c\}\}$.

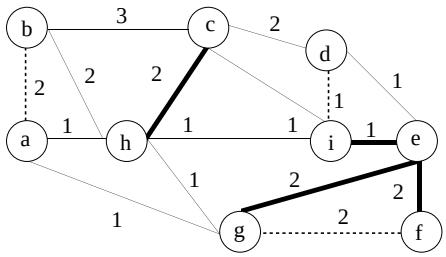


Figure 1.2: *Cycle et cocycle candidats.*

Lemme 1.1. *Soient (X, Y) un approximant, ω un cocycle de G ne contenant aucune arête de X et e une arête libre de coût minimal dans ω ; le couple $(X \cup \{e\}, Y)$ est un approximant.*

Preuve. Soit H un arbre couvrant minimal contenant X et disjoint de Y. Soit T tel que $\omega = \omega(T)$. Si ω ne contient aucune arête de X, il contient au moins une arête libre de Z car dans le cas contraire, l'arbre H ne serait pas connexe puisque ne contenant aucune arête incidente à T. Si H contient l'arête e, alors l'arbre H lui-même répond aux conditions. Sinon l'arête e ferme une chaîne élémentaire de H qui contient nécessairement une arête e' de ω distincte de e. Le graphe K obtenu à partir de H en substituant e à e' est un arbre couvrant de G de coût inférieur ou égal à $c(H)$ et donc égal à $c(H)$ puisque H est optimal. De plus K contient les arêtes de $X \cup \{e\}$ et est disjoint de Y. ∎

Lemme 1.2. *Soient (X, Y) un approximant, γ un cycle élémentaire de G ne contenant pas d'arête de Y et f une arête libre de coût maximal dans γ ; le couple $(X, Y \cup \{f\})$ est un approximant.*

Preuve. Soit H un arbre couvrant minimal contenant X et disjoint de Y. Le cycle γ ne contenant aucune arête de Y contient au moins une arête libre de Z car dans le cas contraire, l'arbre H contiendrait un cycle. Soit f l'arête libre de coût maximal de ce cycle. Si H ne passe pas par f, l'arbre H lui-même répond à la question. Sinon le graphe $H - \{f\}$ contient deux composantes connexes dont les graphes induits sont des arbres. La chaîne élémentaire obtenue à partir de γ après suppression de f contient une arête f' non contenue dans H et incidente à chacune des deux composantes connexes. Le graphe K obtenu à partir de H par substitution de f' à f est un arbre couvrant de G de coût inférieur ou égal à $c(H)$ et donc de coût égal $c(H)$ puisque H est un arbre optimal. De plus K contient X et est disjoint de $Y \cup \{f\}$. ∎

7.1.3 Algorithme général

Les deux lemmes précédents nous permettent d'énoncer deux règles, la *règle des cycles* et la *règle des cocycles*, et de construire un algorithme glouton qui détermine un arbre couvrant de coût minimum en admettant ou en rejetant un arête supplémentaire à chaque itération. Comme dans le paragraphe précédent, (X, Y) est un approximant d'un arbre optimal H.

Règle des Cycles : Soit ω un cocycle candidat ; choisir dans ω une arête libre e de coût minimal ; $Z := Z - \{e\}$; $X := X \cup \{e\}$.

Règle des Cocycles : Soit γ un cycle candidat ; choisir dans γ une arête libre e de coût maximal ; $Z := Z - \{e\}$; $Y := Y \cup \{e\}$.

L'algorithme ARBREMIN(G, c) induit par ces deux règles est alors le suivant :

```
procédure ARBREMIN(G, c) ;
   (X, Y) := (∅, ∅) ; Z := A ;
   tantque G possède un cocycle ou un cycle candidat
      appliquer la règle correspondante
   fintantque.
```

La terminaison de l'algorithme est assurée puisqu'à chaque itération l'ensemble $X \cup Y$ contient un élément de plus. Nous montrons qu'à l'issue de la dernière itération le graphe partiel X est un arbre couvrant de coût minimum.

Théorème 1.3. *L'algorithme* ARBREMIN(G, c) *détermine un arbre couvrant de coût minimum du graphe G pour la valuation c.*

Preuve. (Induction sur le numéro d'itération) Notons X_k et Y_k les sous-ensembles X et Y à l'issue de l'itération k. En utilisant les deux lemmes précédents, il est aisé de montrer par induction sur k que (X_k, Y_k) est un approximant. Notons maintenant K le numéro de la dernière itération. Si $K = m$, le graphe partiel X_K est un arbre couvrant de coût minimum car aucune arête n'est libre. Si $K < m$, il subsiste dans Z_K une arête libre e. L'arête e ne peut fermer une chaîne d'une composante connexe du graphe partiel X_K car la *règle des cycles* aurait pu être appliquée une fois de plus ; l'arête e ne peut non plus lier deux composantes connexes du graphe partiel X_K car la *règle des cocycles* aurait pu être appliquée une fois de plus. D'où la contradiction. ∎

7.1.4 Algorithmes spécifiques

Nous avons choisi de présenter, parmi les nombreux algorithmes de recherche d'un arbre couvrant de coût minimum, l'algorithme de Kruskal et l'algorithme de Prim.

Ces deux algorithmes relèvent de l'algorithme général présenté au paragraphe précédent mais utilisent chacun une stratégie spécifique pour l'ordre d'application de la *règle des cocycles* ou de la *règle des cycles*. Nous constaterons également que moyennant l'utilisation de structures de données adéquates, leur complexité dans le plus mauvais cas est assez faible.

Comme dans le paragraphe précédent, le couple (X, Y) est un approximant. Pour chacun des deux algorithmes, une itération consiste à introduire une arête supplémentaire dans X ou dans Y. Nous noterons à cet effet X_k, Y_k et Z_k les valeurs de X, Y et Z à l'issue de la $k^{ième}$ itération. Par convention nous aurons : $X_0 = Y_0 = \emptyset$ et $Z_0 = A$.

Algorithme de Kruskal

Le principe général de l'algorithme de Kruskal est d'établir une liste des arêtes ordonnée par coût croissant au sens large et d'introduire successivement chaque arête de la liste dans l'ensemble X ou dans l'ensemble Y en appliquant soit la *règle des cocycles* soit la *règle des cycles*.

```
procédure KRUSKAL(G, c);
  déterminer une liste (e_1, ..., e_m) des arêtes ordonnée par coût croissant;
  (X, Y) := (∅, ∅);
  pour i de 1 à m faire
    si e_i lie deux composantes connexes du graphe partiel X
      alors X := X ∪ {e_i}
      sinon Y := Y ∪ {e_i}
    finsi
  finpour.
```

Nous prouvons maintenant que la procédure KRUSKAL(G, c) détermine effectivement un arbre couvrant de coût minimum de G.

Théorème 1.4. *Soit $G = (S, A)$ un graphe connexe et $c : E \mapsto \mathbb{R}$ une valuation de ses arêtes, la procédure* KRUSKAL(G, c) *détermine un arbre couvrant de coût minimum de G pour la valuation c.*

Preuve. Nous montrons par induction sur le numéro d'itération k que chaque itération consiste à appliquer soit la *règle des cocycles* soit la *règle des cycles* si l'une des deux est appliquable. A l'issue de la première itération, $e_1 = \{a, b\}$ lie deux composantes connexes du graphe partiel vide d'arêtes et est une arête libre de coût minimal du cocycle $\omega(\{a\})$. La première itération applique donc la *règle des cocycles*. Supposons que (X_{k-1}, Y_{k-1}) soit un approximant. Si e_k lie deux composantes connexes C' et C'' du graphe partiel X_{k-1}, l'arête e_k est libre et de

coût minimal dans le cocycle candidat $\omega(C')$ puisque $X_{k-1} \cup Y_{k-1} = \{e_1, \ldots, e_{k-1}\}$, l'itération k applique donc la *règle des cocycles*. Sinon, l'arête e_k lie deux sommets d'une même composante connexe C du graphe partiel X_{k-1} et le graphe induit par C est un arbre. L'arête e_k ferme donc une chaîne du graphe partiel X_{k-1} et est la seule arête libre (donc de coût maximal) du cycle ainsi créé; l'itération k applique alors la *règle des cycles*. Puisque $X_m \cup Y_m = A$ et $Z_m = \emptyset$, le graphe partiel X_m est un arbre couvrant de coût minimum de G pour la valuation c. ∎

La figure 1.3 montre l'arbre couvrant (arêtes épaisses) de coût minimum calculé par l'algorithme de Kruskal.

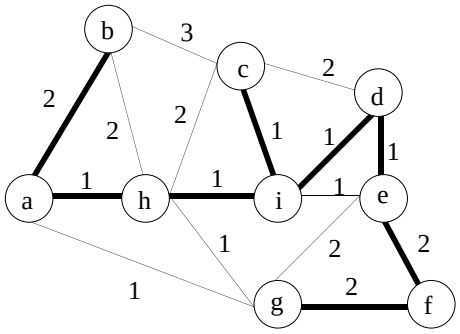


Figure 1.3: *Arbre couvrant de coût minimum.*

Implémentation de l'algorithme de Kruskal

L'algorithme de Kruskal manipule essentiellement les composantes connexes du graphe partiel X à travers les opérations de recherche de la composante connexe à laquelle appartient un sommet (chaque extrémité de l'arête e_i) et d'union de deux composantes connexes (si l'arête e_i lie ces deux composantes). Il s'agit donc de gérer ces deux opérations sur une collection d'ensembles disjoints. Une structure de données adaptée consiste à représenter chaque ensemble par une anti-arborescence dont les sommets représentent les éléments de l'ensemble et dont la racine est le représentant de l'ensemble. Si l'on note $\mathcal{V}$ cette collection, l'opérateur CRÉER$(v, \mathcal{V})$ ajoute le nouvel ensemble $\{v\}$ à $\mathcal{V}$; l'opérateur TROUVER$(v, \mathcal{V})$ détermine le représentant de l'ensemble de $\mathcal{V}$ auquel appartient v; enfin si u et v sont les représentants de deux ensembles distincts, l'opérateur UNIR$(u, v, \mathcal{V})$ remplace l'ensemble représenté par u par l'union des ensembles représentés par u et par v, et détruit l'ensemble représenté par v. La dernière section du chapitre 3 traite de l'implémentation efficace de ces opérateurs et montre en particulier que la complexité amortie d'une suite de m opérations comportant n opérations CRÉER est $O(m\alpha(m, n))$ où $\alpha(m, n)$ est une fonction réciproque de la fonction d'Ackermann.

En utilisant cette structure de données, une implémentation de l'algorithme de Kruskal est la suivante :

```
procédure KRUSKAL(G, c);
  déterminer une liste (e_1, ..., e_m) des arêtes ordonnée par coût croissant;
  {On note a_i et b_i les extrémités de e_i }
  V := ∅; X := ∅;
  pour tout s de S faire CRÉER(s, V) finpour;
  pour i de 1 à m faire
    u :=TROUVER(a_i, V); v :=TROUVER(b_i, V);
    si (u ≠ v) alors UNIR(u, v, V); X := X ∪ {e_i} finsi
  finpour.
```

La complexité du tri initial est $O(m \log n)$. Suivent alors m opérations CRÉER et au plus $2m$ TROUVER et m UNIR. Il résulte des propriétés de la fonction $\alpha(m, n)$ que la complexité globale est celle du tri des arêtes, soit $O(m \log n)$.

L'algorithme de Prim

Le principe de l'algorithme de Prim est d'appliquer $n-1$ fois la *règle des cocycles* à une suite de $n-1$ cocycles candidats associés à des sous-ensembles emboîtés de sommets. Le graphe partiel X ainsi obtenu, qui est sans cycles par construction et possède $n-1$ arêtes, est un arbre couvrant de coût minimum.

```
procédure PRIM (G, c);
  T := {a}; X := ∅; Z := A;
  {a est un sommet quelconque de S}
  pour i de 1 à n − 1 faire
    choisir une arête {x, y} de coût minimal dans ω(T);
    {on suppose : x ∈ T et y ∈ S − T}
    X := X ∪ {{x, y}};
    T := T ∪ {y}
  finpour;
  retourner(X).
```

Théorème 1.5. *Soit $G = (S, A)$ un graphe connexe et $c : A \mapsto \mathbb{R}$ une valuation de ses arêtes, la procédure* PRIM(G, c) *détermine un arbre couvrant de coût minimum de G pour la valuation c.*

Preuve. Durant son exécution, l'algorithme n'écarte aucune arête. Donc, lors de chaque itération, le cocycle $\omega(T)$ est candidat pour la *règle des cocycles* puisque G est connexe et X est inclus dans l'ensemble des arêtes du sous-graphe induit par T; l'arête $\{x, y\}$ est alors l'arête libre de coût minimal du cocycle $\omega(T)$. Chaque itération applique donc la *règle des cocycles*. ∎

La figure 1.4 montre l'arbre couvrant de coût minimum (arêtes épaisses) calculé par l'algorithme de Prim.

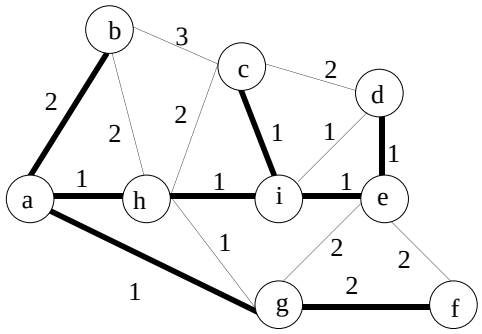


Figure 1.4: *Arbre couvrant de coût minimum.*

Implémentation de l'algorithme de Prim

L'opération fondamentale de l'algorithme de Prim est la recherche d'une arête de coût minimal d'un cocycle. Cette opération est appliquée à une suite de cocycles pour laquelle chaque sous-ensemble contient un sommet de plus que le précédent.

Une implémentation efficace de ces opérations utilise un tas (voir chapitre 3, section 3.4) pour gérer les sommets de la bordure $\mathcal{B}(T)$ de T où chaque sommet y de $\mathcal{B}(T)$ est affecté d'une priorité notée $p(y)$ égale au coût minimal d'une arête de $\omega(T)$ d'extrémité y. Un sommet de $\mathcal{B}(T)$ est *promouvable* s'il est de priorité minimale.

Nous utilisons les opérateurs suivants : Créer, Insérer, Extrairemin, Priorité, Est-élément qui sont respectivement les opérateurs de création d'un tas, d'insertion d'un élément, de suppression d'un élément de priorité minimum, d'affectation d'une nouvelle priorité à un élément et de test d'appartenance d'un élément.

Soit $\omega(T)$ le cocycle courant à l'issue d'une itération de l'algorithme de Prim et $\mathcal{T}$ le tas associé à $\mathcal{B}(T)$. Nous supposons que le nœud de $\mathcal{T}$ associé à un sommet y de $\mathcal{B}(T)$ contient outre la priorité $p(y)$ de y une arête de $\omega(T)$ d'extrémité y et de coût $p(y)$. Une arête de coût minimal de $\omega(T)$ est ainsi stockée à la racine de $\mathcal{T}$.

Soit $\{x, y\}$ l'arête sélectionnée lors de l'itération suivante ($x \in T, y \in \mathcal{B}(T)$), la mise à jour de T et de $\mathcal{B}(T)$ lors de la promotion du sommet y est réalisée par la procédure Promotion ci-dessous :

```
procédure PROMOTION(y, T);
  pour tout voisin z de y dans S − T faire
    si EST-ÉLÉMENT(z, 𝒯) alors
      si p(z) > c({y, z}) alors PRIORITÉ(z, c({y, z}), 𝒯)) finsi
    sinon INSÉRER(z, c({z, y}), 𝒯)
    finsi
  finpour;
  T := T ∪ {y}.
```

L'utlisation d'un tableau booléen permet de réaliser l'opération EST-ÉLÉMENT en temps constant. L'opération PRIORITÉ se réalise facilement si la file de priorité est implémentée par un tas. Elle prend alors un temps $O(\log n)$ où n est le nombre d'éléments du tas. Il est alors possible d'implémenter l'algorithme de Prim en utilisant la promotion comme suit :

```
procédure PRIM-AVEC-PROMOTION(G, c);
  CRÉER(𝒯);
  choisir un sommet a de S; T := {a};
  pour tout voisin v de a faire INSÉRER(v, c({a, v}), 𝒯) finpour;
  pour i de 1 à n − 1 faire
    y :=EXTRAIREMIN(𝒯);
    soit {x, y} l'arête rangée au nœud de 𝒯 contenant y;
    X := X ∪ {{x, y}};
    PROMOTION(y, T)
  finpour.
```

Chaque itération sur i supprime la racine (complexité $O(\log n)$), met X à jour (complexité $O(1)$) et réalise la promotion d'un sommet. Pour cette dernière, et dans le plus mauvais cas, il faudra exécuter pour chaque voisin le test d'appartenance au tas et soit une insertion soit une affectation de priorité. Donc le nombre total de mises à jour du tas $\mathcal{T}$ dues aux promotions est en $O(m)$. Il en résulte que la complexité globale de la procédure PRIM(G, c) est $O(m \log n)$.

7.2 Chemins de coût minimum

Lorsque les arcs d'un graphe orienté sont étiquetés par des nombres réels qui peuvent représenter des coûts, des distances, des durées ..., on définit de manière naturelle le coût d'un chemin comme la somme des étiquettes des arcs successivement empruntés par ce chemin. Un problème important est le calcul efficace d'un

chemin de coût minimum entre deux sommets donnés. Comme on peut l'imaginer aisément, ce problème intervient dans de très nombreuses applications et joue de plus un rôle central dans la résolution de certains problèmes plus difficiles tels que les flots, les ordonnancements, la programmation dynamique. Dans cette section, nous montrerons, suivant en cela la démarche unificatrice de Tarjan, que les principaux algorithmes de résolution sont issus d'une même itération fondamentale, que leur efficacité est liée à l'ordre dans lequel ces itérations sont exécutées et que pour obtenir une complexité faible, cet ordre doit dépendre des hypothèses faites sur les données. La plupart des algorithmes de résolution déterminent en fait pour tout sommet accessible à partir de l'origine un chemin de coût minimum de l'origine à ce sommet. Nous spécifierons donc le problème sous cette forme. Dans cette section, le seul algorithme présenté spécifique à un sommet origine et un sommet destination est l'algorithme appelé A^*. Il s'agit d'une variante de l'algorithme de Dijkstra qui utilise une évaluation par défaut supposée connue a priori du coût minimum d'un chemin de tout sommet au sommet destination. Cet algorithme est très utilisé pour des graphes de grande taille définis de manière implicite (problèmes de recherche en intelligence artificielle, programmation dynamique), car il permet d'obtenir la solution sans pour autant développer tous les sommets accessibles à partir de l'origine.

7.2.1 Définition du problème

Soient $G = (S, A)$ un graphe orienté, $c : A \mapsto \mathbb{R}$ une *valuation* des arcs de G et s un sommet du graphe G appelé *origine*. On suppose que tout sommet est accessible à partir de s. On appelle coût de l'arc (x, y) la valeur $c(x, y)$. Le *coût du chemin* $\gamma = (z_0, \ldots, z_q)$ est la somme $c(\gamma) = \sum_{k=1}^{q} c(z_{k-1}, z_k)$. L'objet du problème est de déterminer, pour chaque sommet x, le coût minimum noté $l(s, x)$ de s à x. La figure 2.1 représente un graphe valué muni d'un sommet origine (en grisé). Etant donnés deux chemins $\beta = (x_0, \ldots, x_p)$ et $\gamma = (y_0, \ldots, y_q)$ tels que

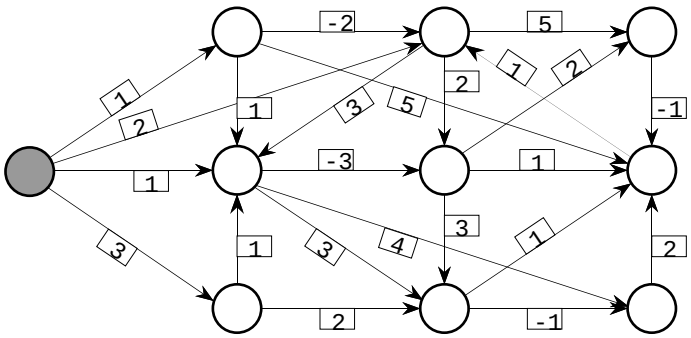


Figure 2.1: *Un graphe valué.*

$y_0 = x_p$, nous appelons concaténation des deux chemins β et γ le chemin noté $\beta\gamma$=$(x_0, \ldots, x_{p-1}, y_0, \ldots, y_q)$.

7.2.2 Existence d'une solution

La question de l'existence d'une solution repose sur la notion de *circuit absorbant*, c'est-à-dire ici de circuit de coût strictement négatif. Si un tel circuit existe, alors pour tout sommet x accessible à partir d'un sommet du circuit, il existe un chemin de s à x de coût arbitrairement petit obtenu en parcourant le circuit autant de fois que nécessaire. Le circuit $(1, 2, 3, 1)$ est absorbant pour le graphe de la figure 2.2. La proposition suivante précise la condition d'existence d'une solution.

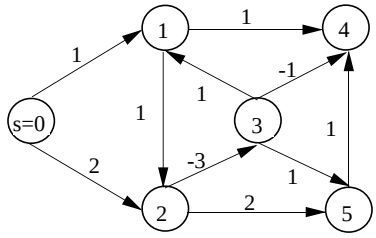


Figure 2.2: *Un circuit absorbant.*

Proposition 2.1. *Il existe un chemin de coût minimum de l'origine à tout autre sommet si et seulement si le graphe ne possède pas de circuit de coût strictement négatif.*

Preuve. La condition est bien sûr nécessaire. Réciproquement, soit x un sommet quelconque. L'ensemble des chemins *élémentaires* de s à x est dominant pour le problème car le coût d'un chemin élémentaire extrait d'un chemin γ de s à x est au plus égal au coût de γ. Les chemins élémentaires de s à x étant en nombre fini, il existe un chemin élémentaire de coût minimum. ∎

La proposition précédente n'est pas constructive et ne met pas en lumière la propriété structurelle fondamentale d'un chemin de coût minimum.

Proposition 2.2. *Soit $\gamma = (z_0, \ldots, z_q)$ un chemin de coût minimum de $x = z_0$ à $y = z_q$, tout chemin $(z_k, z_{k+1}, \ldots, z_l)$, $0 \leq k \leq l \leq q$, est un chemin de coût minimum de z_k à z_l.*

Preuve. Notons ν le chemin $(z_k, z_{k+1}, \ldots, z_l)$ et soit μ un chemin de z_k à z_l tel que $c(\mu) < c(\nu)$. Le chemin obtenu à partir de γ en remplaçant ν par μ est de coût strictement inférieur au coût de γ. Contradiction. ∎

La propriété précédente qui est élémentaire pour les chemins de coût minimum, est la base de la «programmation dynamique», une technique fondamentale pour résoudre certains problèmes d'optimisation combinatoire et de contrôle optimal. Ici, cette propriété induit une seconde condition nécessaire et suffisante fondée sur l'existence d'une arborescence partielle constituée de chemins de coût minimum. A cet effet, nous appellerons *arborescence de chemins minimaux* une arborescence partielle de G dont la racine est s et dont chaque chemin de s à x est un chemin de coût minimum de s à x dans G.

Proposition 2.3. *Si G n'a pas de circuits absorbants, G possède une arborescence des chemins minimaux.*

Preuve. Appelons *pré-arborescence des chemins minimaux* un sous-graphe $\mathcal{A} = (T, B)$ de G qui est une arborescence de racine s telle que pour tout sommet t de T, le chemin dans $\mathcal{A}$ de s à t est de coût minimum *dans* G. Nous montrons la condition par construction itérative d'une arborescence des chemins minimaux, à partir de la pré-arborescence des chemins minimaux $\mathcal{A}_0 = (\{s\}, \emptyset)$. On détermine à l'étape k une pré-arborescence des chemins minimaux $\mathcal{A}_k$ qui couvre au moins un sommet de plus que $\mathcal{A}_{k-1}$. Soient donc (voir figure 2.3)

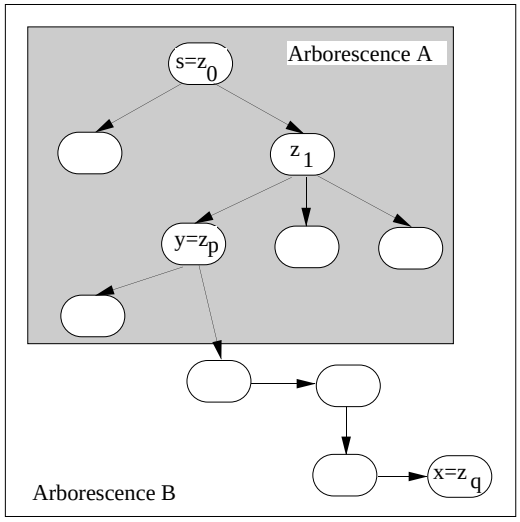


Figure 2.3: *Construction d'une arborescence des chemins minimaux.*

$\mathcal{A} = (T, B)$ une pré-arborescence des chemins minimaux telle que $S - T \neq \emptyset$,
x un sommet de $S - T$,
$\gamma = (z_0, \ldots, z_q)$ un chemin (élémentaire) de coût minimum de $s = z_0$ à $x = z_q$ dans G,
$y = z_p$ le *dernier* sommet de γ appartenant à T,
γ' le sous-chemin $(z_0, \ldots, z_p)$,
γ'' le sous-chemin $(z_p, \ldots, z_r)$ $(r \in \{p+1, .., q\})$,
β le chemin de s à y dans $\mathcal{A}$.

Les deux chemins γ' et β ont le même coût d'après la proposition 2.2. Donc, pour tout $r \in \{p+1, .., q\}$, le chemin $\beta\gamma''$ est de coût minimum dans G. De plus si $T' = T \cup \{z_{p+1}, \ldots, z_q\}$ et $B' = B \cup \{(z_j, z_{j+1}) \mid j = p, .., (q-1)\}$, alors le sous-graphe $\mathcal{A}' = (T', B')$ est une pré-arborescence des chemins minimaux qui couvre au moins un sommet de plus que $\mathcal{A}$. Il en résulte qu'après au plus $n - 1$ itérations, on obtient une arborescence des chemins minimaux. ∎

La plupart des algorithmes de recherche d'un chemin de coût minimum cherchent à construire par ajustements successifs une arborescence des chemins minimaux.

En chaque sommet x, ils font décroître une évaluation par excès $\Delta(x)$ du coût minimum d'un chemin de s à x qui est égale à la valeur d'un chemin de s à x. Le prédécesseur de x sur ce chemin, appelé *père de* x et noté $p(x)$, est également maintenu. Lors de la terminaison de l'algorithme, les arcs de l'ensemble $\{(p(x), x) \mid x \in S - \{s\}\}$ constituent une arborescence des chemins minimaux de G. Ces algorithmes utilisent comme test d'arrêt la caractérisation suivante d'une arborescence des chemins minimaux :

Proposition 2.4. *Soit $\mathcal{A}$ une arborescence partielle de racine s et soit $\lambda(x)$ le coût du chemin de s à x dans $\mathcal{A}$. L'arborescence $\mathcal{A}$ est une arborescence des chemins minimaux si et seulement si pour tout arc (x, y) de G on a : $\lambda(x) + c(x, y) \geq \lambda(y)$.*

Preuve. La condition nécessaire est immédiate. Réciproquement, soit γ un chemin de s à y. En sommant les inégalités (de la proposition) vérifiées par λ pour tous les arcs de γ, il vient : $\lambda(y) - \lambda(s) \leq c(\gamma)$ ou encore, puisque $\lambda(s) = 0 : \lambda(y) \leq c(\gamma)$. Le chemin de s à y dans $\mathcal{A}$ est donc de coût minimum dans G. ∎

7.2.3 Itération fondamentale

Soit $\Delta : S \mapsto \mathbb{R} \cup \{+\infty\}$ une fonction telle que si $\Delta(x) \in \mathbb{R}$, $\Delta(x)$ soit la valeur d'un chemin élémentaire de s à x. Un sommet x est dit *évalué* si $\Delta(x) \in \mathbb{R}$. Un arc (x, y) est dit *candidat*si $\Delta(x) + c(x, y) < \Delta(y)$ et l'on note C l'ensemble des arcs candidats. L'itération fondamentale, due à Ford, choisit un arc candidat (x, y) et exécute la mise à jour des fonctions p et Δ :

procédure AJUSTER-ARBORESCENCE(x, y) ;
$\Delta(y) := \Delta(x) + c(x, y)$; $p(y) := x$.

La figure 2.4 résume la transformation élémentaire réalisée à chaque itération.

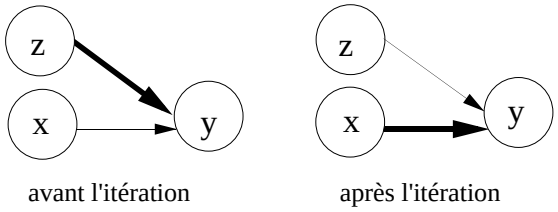


Figure 2.4: *L'itération de Ford.*

Avant cette itération on a : $\Delta(x) + c(x, y) < \Delta(y)$, $p(y) = z$ et $\Delta(z) + c(z, y) = \Delta(y)$. Après cette itération on a : $\Delta(x) + c(x, y) = \Delta(y)$, $p(y) = x$ et $\Delta(z) + c(z, y) > \Delta(y)$. La procédure AJUSTER-CANDIDATS(x, y) ci-dessous définit le nouvel ensemble d'arcs candidats.

```
procédure AJUSTER-CANDIDATS( x, y);
  C := C − {(x, y)};
  pour tout successeur z de y faire
    si Δ(y) + v(y, z) < Δ(z) alors
      C := C ∪ {(y, z)}
  finpour.
```

7.2.4 Algorithme de Ford

L'algorithme de Ford fondé sur la proposition précédente peut être implémenté par la procédure suivante :

```
procédure FORD(G, s);
  Δ(s) := 0; p(s) := ∅; C := ∅;
  pour tout sommet x de S − {s} faire Δ(x) := +∞; p(x) := ∅ finpour;
  pour tout successeur z de s faire C := C ∪ {(s, z)};
  tantque C ≠ ∅ faire
    choisir un arc (x, y) dans C;
    AJUSTER-ARBORESCENCE( x, y);
    AJUSTER-CANDIDATS( x, y)
  fintantque;
  retourner(p, Δ).
```

Le théorème suivant garantit la validité de la procédure FORD lorsque G ne possède pas de circuit absorbant.

Théorème 2.5. *Si le graphe G ne possède pas de circuit absorbant, la procédure* FORD*(G, s) se termine et à l'issue de la dernière itération, la fonction p définit une arborescence de chemins minimaux.*

Preuve. Si le graphe G ne possède pas de circuit absorbant, on montre aisément les propriétés **a)** et **b)** suivantes par induction sur le numéro d'itération :

a) si $\Delta(x)$ est fini, il existe un chemin *élémentaire* de s à x de coût $\Delta(x)$;

b) la *restriction* de la fonction p aux sommets évalués est une arborescence partielle du sous-graphe de G induit par les sommets évalués telle que pour tout sommet évalué y distinct de la racine $\Delta(p(y)) + c(p(y), y) = \Delta(y)$.

Si une itération porte sur un arc candidat (x, y) tel que y est évalué, la valeur $\Delta(y)$ décroît d'une quantité α égale à la différence (strictement positive) des coûts

de deux chemins élémentaires distincts de s à y. La valeur α est donc minorée par l'écart positif minimum ϵ entre les coûts de deux chemins élémentaires de coûts distincts. Le réel ϵ est strictement positif car l'ensemble des chemins élémentaires de G est fini, le nombre total de mises à jour de la fonction Δ pour le sommet y est donc *fini*. Il en résulte que la procédure se termine.

Lors de la terminaison, tous les sommets sont évalués, car si un sommet y ne l'était pas, il resterait au moins un arc candidat sur tout chemin de s à y. De plus, aucun arc (x, y) n'étant candidat, la fonction père définit un arborescence des chemins minimaux de G. ∎

Le théorème précédent garantit la validité de l'algorithme de Ford si le graphe G ne possède pas de circuit absorbant. Cependant, en laissant totalement libre l'ordre des itérations, c'est-à-dire le choix des arcs candidats, la convergence peut être très lente. Pour certains graphes valués à m arcs, il est possible d'exécuter jusqu'à 2^m itérations.

Pour obtenir une complexité polynomiale, la plupart des algorithmes regroupent les arcs ayant la même origine. Une itération encore appelée examen d'un sommet consiste alors à choisir un sommet et à exécuter la procédure AJUSTER-ARBORESCENCE pour tous les arcs candidats incidents extérieurement à ce sommet.

Il sera commode, pour analyser les différents algorithmes, de distinguer trois états pour un sommet. On *ouvre* un sommet x chaque fois que son évaluation décroît, on le *ferme* à l'issue de chaque exécution de EXAMINER, il reste *libre* tant qu'il n'est pas évalué. Initialement, le sommet s est ouvert et tous les autres sommets sont libres. Nous noterons respectivement O, F et L les sous-ensembles des sommets ouverts, fermés et libres. Les procédures OUVRIR(x) et FERMER(x) réalisent les mises à jour des ensembles O, F et L induites par l'ouverture et la fermeture du sommet x. La procédure INIT(G, s) ouvre le sommet s et libère tous les autres sommets.

La figure 2.5 représente l'automate des transitions possibles entre ces trois états. La procédure EXAMINER(x) implémente l'examen d'un sommet qui constitue

Figure 2.5: *Le graphe des états d'un sommet.*

maintenant une itération.

```
procédure EXAMINER(x);
  pour tout successeur y de x faire
    si Δ(x) + c(x, y) < Δ(y) alors
      AJUSTER-ARBORESCENCE( x, y); OUVRIR(y)
    finsi
  finpour;
  FERMER(x).
```

Pour obtenir un algorithme efficace, il convient de faire en sorte qu'un sommet fermé ne puisse plus (ou du moins le moins souvent possible) redevenir ouvert.

Nous allons analyser deux cas particuliers pour lesquels les itérations peuvent être ordonnées de telle sorte que tout sommet fermé reste fermé.

7.2.5 Graphe sans circuit

Soient G un graphe sans circuit et $L = (s_1, \ldots, s_n)$ une liste topologique des sommets de G telle que $s_1 = s$.

Proposition 2.6. *L'algorithme* BELLMAN(G, s) *examine les sommets une fois et une seule dans l'ordre de la liste.*

```
procédure BELLMAN(G, s);
  INIT( G, s);
  pour k de 1 à n faire EXAMINER(s_k).
```

Preuve. Lors de l'itération k ($k \geq 2$), le sommet s_k n'est pas libre car le sommet s_k (accessible à partir de s dans G) possède un prédécesseur s_i ($i < k$) placé avant lui dans la liste et qui a été examiné. Après l'exécution de EXAMINER(s_k), le sommet s_k est fermé et tous les sommets s_i ($i < k$) restent fermés puisque seuls les $\Delta(s_j)$ ($j > k$) ont pu décroître lors de cette itération. ∎

7.2.6 Algorithme de Dijkstra

Si les coûts sont positifs ou nuls, la règle qui consiste à examiner un sommet ouvert d'évaluation minimale conduit à un algorithme efficace dû à Dijkstra. L'algorithme est alors décrit par la procédure ci-dessous où INIT(G, s) initialise O, L et F.

```
procédure DIJKSTRA(G, s);
  INIT(G, s);
  répéter n − 1 fois
    choisir un sommet ouvert x d'évaluation minimale;
    EXAMINER(x)
  finrépéter.
```

La validité de cet algorithme repose sur sur la propriété suivante :

Théorème 2.7. *L'évaluation du sommet ouvert x choisi à chaque itération est le coût minimum d'un chemin de s à x dans G.*

Preuve. (Induction sur le numéro d'itération.) Le coût minimum d'un chemin de s à s est nul puisque le coût de tout circuit est positif ou nul. La propriété est donc vraie pour la première itération. Soit x le sommet ouvert choisi lors de l'itération i ($i > 1$) et F l'ensemble des sommets examinés avant l'itération i. Remarquons que d'après l'induction, tout sommet t de F a été choisi lors d'une itération antérieure à i et est resté fermé depuis en conservant une évaluation égale au coût minimum d'un chemin de s à t dans G. Il résulte alors de la procédure EXAMINER que pour tout sommet y de $S - F$, on a avant l'itération i :

$$\Delta(y) = \begin{cases} l(s,y) & \text{si } y \in F \\ \min\{l(s,z) + c(z,y) \mid z \in F \text{ et } (z,y) \in A\} & \text{si } y \in O \\ +\infty & \text{si } y \in L. \end{cases}$$

Soit $\gamma = (z_0, \ldots, z_q)$ un chemin quelconque de s à x et (z_{k-1}, z_k) le dernier arc de γ tel que $z_{k-1} \in F$ et $z_k \notin F$. On a donc avant l'itération i :

$$c(\gamma) \geq \Delta(z_{k-1}) + c(z_{k-1}, z_k) \geq \Delta(z_k) \geq \Delta(x)$$

d'après l'induction, la positivité des coûts, la relation précédente et le choix de x. Il en résulte que $\Delta(x)$ est le coût minimum d'un chemin de s à x dans G et que pour l'un des ces chemins, tous les sommets, sauf le dernier, appartiennent à F. ∎

Proposition 2.8. *L'algorithme* DIJKSTRA *sélectionne les sommets dans l'ordre croissant (au sens large) des coûts minimaux.*

Preuve. Raisonnons par induction sur le numéro d'itération et notons t le *dernier* sommet fermé avant l'itération i. Nous avons par induction pour tout sommet z de T : $\Delta(z) \leq \Delta(t)$ et $\Delta(x) \geq \Delta(t)$ puisque le sommet x n'a pas été choisi avant l'itération i . ∎

Les figures 2.6 et 2.7 décrivent une exécution de l'algorithme. Les sommets fermés sont noirs, les sommets ouverts sont gris, le sommet ouvert sélectionné est doublement cerclé, les sommets libres sont blancs et l'évaluation d'un sommet figure

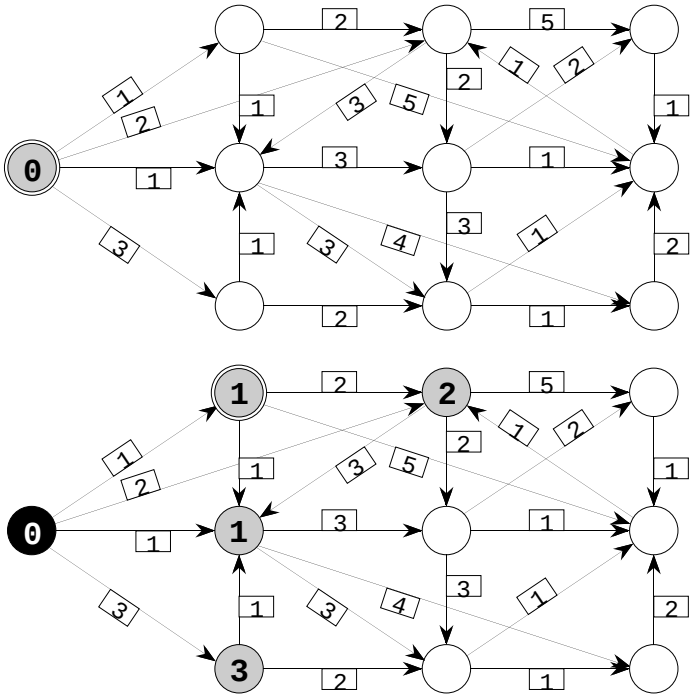


Figure 2.6: *Graphe initial et première itération.*

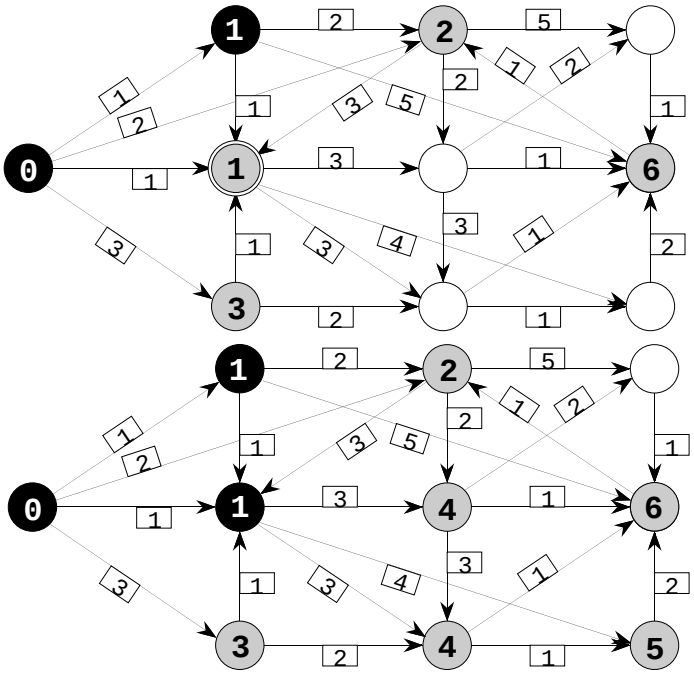


Figure 2.7: *Les deux itérations suivantes.*

à l'intérieur de ce sommet si elle est finie. L'évolution complète de l'ensemble O, du sommet sélectionné x et de la fonction Δ est donnée dans le tableau ci-dessous

(la dernière colonne est le coût minimum $l(0,x)$ de l'origine 0 au sommet x) :

O	x	Δ	
$\{0\}$	0	$\Delta(0)=0$	0
$\{1,2,3,4\}$	1	$\Delta(1)=1, \Delta(2)=1, \Delta(3)=3, \Delta(4)=2$	1
$\{2,3,4,8\}$	2	$\Delta(4)=2, \Delta(2)=1, \Delta(3)=3, \Delta(8)=6$	1
$\{3,4,8,5,6,9\}$	4	$\Delta(4)=2, \Delta(8)=6, \Delta(3)=3, \Delta(5)=4, \Delta(6)=4, \Delta(9)=5$	2
$\{3,8,5,6,9,7\}$	3	$\Delta(7)=7, \Delta(8)=6, \Delta(3)=3, \Delta(5)=4, \Delta(6)=4, \Delta(9)=5$	3
$\{8,5,6,9,7\}$	5	$\Delta(7)=7, \Delta(8)=6, \Delta(5)=4, \Delta(6)=4, \Delta(9)=5$	4
$\{8,6,9,7\}$	6	$\Delta(7)=6, \Delta(8)=5, \Delta(6)=4, \Delta(9)=5$	4
$\{8,9,7\}$	8	$\Delta(7)=6, \Delta(8)=5, \Delta(9)=5$	5
$\{9,7\}$	9	$\Delta(7)=6, \Delta(9)=5$	5
$\{7\}$	7	$\Delta(7)=6$	6

Proposition 2.9. *L'algorithme de Dijkstra détermine les chemins de coût minimum du couple (G,s) en temps $O(m \log n)$.*

Preuve. Si nous appelons *bordure* de F l'ensemble noté $\mathcal{B}(F)$ des sommets (nécessairement ouverts) de $S-F$ qui possèdent au moins un prédécesseur dans F, et si nous associons à chaque élément x de $\mathcal{B}(F)$ une priorité égale à son évaluation $\Delta(x)$, nous constatons qu'à chaque itération de l'algorithme, il faut essentiellement déterminer un élément z de $\mathcal{B}(F)$ de priorité minimale, supprimer z de $\mathcal{B}(F)$, insérer dans $\mathcal{B}(F)$ les successeurs de z qui ne lui appartiennent pas déjà et modifier éventuellement la priorité des successeurs de z qui appartiennent à $\mathcal{B}(F)$. La gestion de $\mathcal{B}(F)$ par un tas, tout à fait analogue à celle étudiée pour implémenter l'algorithme de Prim, démontre la proposition. ∎

7.2.7 Algorithme A^*.

Nous présentons dans ce paragraphe une variante de l'algorithme de Dijkstra adaptée au cas où il s'agit de déterminer dans un graphe dont les arcs sont valués par des coûts positifs ou nuls un chemin de coût minimum entre un sommet origine s et un sommet destination p.On suppose que l'on dispose initialement pour chaque sommet x d'une *évaluation par défaut* notée $h(x)$ du coût minimum $l(x,p)$ d'un chemin de x à p. L'idée fondamentale de l'algorithme A^* est alors d'associer à chaque sommet x une approximation de $l(s,p)$, notée $f(x)$, qui tient compte de la cible p par le biais de l'évaluation par défaut $h(x)$ et d'examiner en priorité un sommet ouvert d'approximation minimale.

La procédure A*(G,s,p,h) ci-dessous décrit cet algorithme.

```
procédure A*(G, s, p, h);
  INIT(G, s); f(s) = h(s); O := {s}; x := s;
  tantque x ≠ p faire
    EXAMINER*(x);
    x := sommet ouvert d'approximation f(x) minimale
  fintantque.
```

La procédure EXAMINER*(x) ne se distingue de la procédure EXAMINER(x) que par la mise à jour de l'approximation des nouveaux sommets ouverts.

```
procédure EXAMINER*(x);
  pour tout successeur y de x faire
    si Δ(x) + c(x, y) < Δ(y) alors
      AJUSTER-ARBORESCENCE( x, y);
      OUVRIR(y);
      f(y) := Δ(y) + h(y)
    finsi
  finpour;
  FERMER(x).
```

Le tableau suivant montre l'évolution de l'ensemble des sommets ouverts sur le graphe valué de la figure 2.8 pour lequel $s = 1$ et $p = 7$. Sur cette figure, l'évaluation par défaut initiale et les valeurs successives de l'approximation sont inscrites à côté de chaque sommet.

k	1	2	3	4	5	6	7
O	$\{1\}$	$\{2,3,4\}$	$\{2,4,6\}$	$\{3,4,6\}$	$\{4,6\}$	$\{4,7\}$	$\emptyset$

La validité de l'algorithme A^* repose sur le fait que l'approximation du sommet

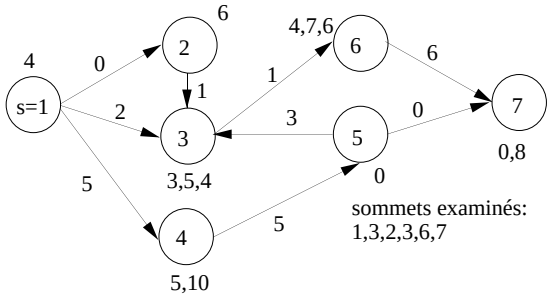


Figure 2.8: *Un graphe valué et la fonction h.*

examiné à chaque itération est une *évaluation par défaut* de $l(s,p)$. Lorsque le sommet p lui-même est examiné, son approximation vaut $f(p) = \Delta(p)$ et est donc égale à $l(s,p)$.

Proposition 2.10. *Lors de chaque itération, l'approximation $f(x)$ du sommet examiné est une évaluation par défaut de $l(s,p)$.*

Preuve. Soit $\gamma = (x_0, \ldots, x_r)$ un chemin *de coût minimum* de s à p. Nous montrons d'abord par induction qu'à l'issue de chaque itération, il existe un entier k tel que :

a) tout sommet x_i du sous-chemin $(x_0, \ldots, x_k)$ est fermé et satisfait $\Delta(x_i) = l(s, x_i)$;
b) le sommet x_{k+1} est ouvert et satisfait $\Delta(x_{k+1} = l(s, x_{k+1})$.

Remarquons d'abord que le sommet $s = x_0$ reste fermé car tout circuit passant par s est de coût positif ou nul. Il en résulte que l'invariant est vrai à l'issue de la première itération. Soit maintenant z le sommet examiné lors de l'itération i, et k l'entier associé par l'invariant à l'issue de l'itération $i-1$. Tout sommet x_i du sous-chemin $(x_0, \ldots, x_k)$ reste fermé puisque $\Delta(x_i) = l(s, x_i)$. Si $z \neq x_{k+1}$, le sommet x_{k+1} reste ouvert et l'invariant est vrai pour l'entier k à l'issue de l'itération i. Si $z = x_{k+1}$, le sommet x_{k+1} devient fermé. Considérons alors le sous-chemin $(x_{k+1}, \ldots, x_r)$ à l'issue de l'itération en cours. Ses sommets n'étant pas tous fermés puisque x_r est ouvert, notons x_j son premier sommet ouvert. Tous les sommets x_i de $(x_{k+1}, \ldots, x_{j-1})$ sont fermés et vérifient par définition $\Delta(x_i) = l(s, x_i)$. On a également $\Delta(x_j) = l(s, x_j)$ puisque le sommet x_{j-1} a été examiné. L'invariant est donc vrai pour l'entier $j-1$. L'invariant est donc conservé.

Considérons alors le sommet x_{k+1} où k est l'entier associé par l'invariant à l'itération en cours. On a :

$$f(x_{k+1}) = \Delta(x_{k+1} + h(x_{k+1}) = l(s, x_{k+1}) + h(x_{k+1}) \leq l(s,p).$$

Le sommet x examiné étant un sommet ouvert d'approximation minimale, il satisfait aussi $f(x) \leq l(s,p)$. ∎

La proposition précédente nous permet de conclure que l'algorithme A*(G, s, p, h) converge vers la solution. En effet, sa terminaison est assurée puisque son principe est l'itération de Ford et qu'il n'existe pas de circuits absorbants. De plus lorsque le sommet ouvert p est examiné, son approximation $f(p)$ satisfait $l(s,p) \geq f(p) = \Delta(p) \geq l(s,p)$.

L'algorithme A^* ne diffère de l'algorithme de Dijkstra que par le choix du sommet ouvert à examiner. En particulier si la fonction d'évalution par défaut h est nulle en tout sommet, les deux algorithmes sont identiques. De manière plus fine, on peut montrer que si la fonction h satisfait l'hypothèse de *consistance* :

$$\forall x, y \in S, \quad h(x) \leq l(s,y) + h(y)$$

alors comme dans l'algorithme de Dijkstra, le nouveau sommet examiné est celui d'évaluation minimale et donc restera fermé.

L'algorithme A^* peut être efficace dans le cas de graphes de grande taille définis de manière implicite pour lesquels on dispose d'une bonne fonction d'évaluation par défaut. En effet, dans ce cas, certains sommets ouverts dont l'approximation est de valeur élevée n'auront pas été examinés lors de la terminaison de l'algorithme.

7.2.8 Algorithme « PAPS »

Nous présentons dans ce paragraphe un algorithme pour le cas où les coûts sont des nombres réels quelconques. Sa particularité essentielle est d'examiner le sommet ouvert *le plus ancien*. Cet algorithme fournit bien entendu la solution du problème en l'absence de circuits absorbants mais permet également de déceler la présence d'un tel circuit lorsque le problème n'a pas de solution. L'ensemble des sommets ouverts est géré dans l'ordre « premier arrivé premier servi » au moyen d'une file. La procédure générale PAPS(G, s) dans laquelle la file est notée Φ implémente cet algorithme.

```
procédure PAPS(G, s);
   INIT(G, s);
   FILEVIDE(Φ); ENFILER(s, Φ);
   n(Φ) := 1; k := 0;
   tantque Φ n'est pas vide et k ≤ n − 1 faire
      ÉTAPE; k := k + 1
   fintantque;
   si Φ n'est pas vide alors afficher « circuit absorbant ».
```

La procédure ÉTAPE consiste à examiner tous les sommets présents dans la file à l'issue de l'étape précédente. On suppose que le nombre d'éléments dans la file, noté $n(\Phi)$ est maintenu lors de chaque mise à jour.

```
procédure ÉTAPE;
   pour j de 1 à n(Φ) faire
      x :=DÉFILER(Φ);
      EXAMINER(x)
   finpour.
```

Le tableau suivant montre l'évolution de l'ensemble des sommets ouverts pour le graphe valué de la figure 2.9. Sur cette figure, les évaluations successives d'un sommet sont inscrites à coté de ce sommet.

k	1	2	3	4	5	6
O	$\{0\}$	$\{1,2,3\}$	$\{5,1,6,4\}$	$\{6,5,3\}$	$\{6\}$	$\emptyset$

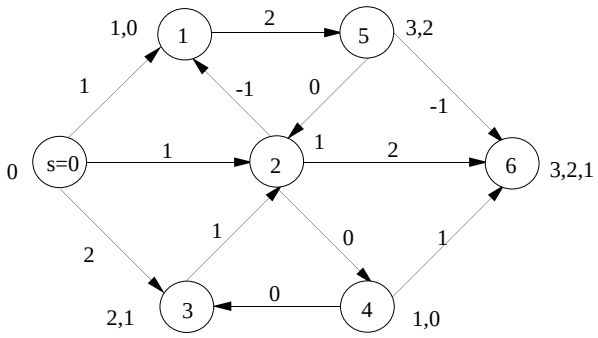


Figure 2.9: *Les évaluations de l'algorithme* PAPS.

Pour analyser l'algorithme PAPS(G, s), nous allons considérer l'ensemble noté O_k des sommets ouverts au début de l'étape k. Nous notons alors $C_k(x)$ l'ensemble (fini) des chemins de s à x de longueur inférieure ou égale à k et $\lambda_k(x)$ le coût minimum d'un chemin de $C_k(x)$ (par convention $\lambda_k(x) = +\infty$ si $C_k(x)$ est vide). L'invariant suivant est alors conservé :

Théorème 2.11. *Au début de l'étape k, l'évaluation $\Delta(x)$ d'un sommet x est égale au coût minimum d'un chemin de $C_k(x)$. Un sommet x est ouvert si et seulement si tout chemin de coût minimum de $C_k(x)$ possède exactement k arcs.*

Preuve. (Induction sur k.) La proposition est vraie par définition pour $k = 0$. Soit alors x un sommet tel que $C_{k+1}(x)$ ne soit pas vide. Au début de l'étape $k + 1$, on a :

$$\Delta_{k+1}(x) = \min\{\Delta_k(x), \min\{\Delta_k(y) + c(y, x) | y \in O_k \cap \Gamma^-(x)\}\}.$$

S'il existe un chemin de longueur inférieure ou égale à k de coût minimum dans $C_{k+1}(x)$, on a d'après l'induction $\Delta_{k+1}(x) = \Delta_k(x) = \lambda_{k+1}(x)$. Supposons maintenant que tout chemin de coût minimum dans $C_{k+1}(x)$ soit de longueur $k + 1$. Soit γ un tel chemin et (y, x) le dernier arc de γ. Tout chemin de coût minimum de $C_k(y)$ est de longueur k, donc d'après l'induction $y \in O_k \cap \Gamma^-(x)$ et $\Delta_{k+1}(x) = \lambda_{k+1}(x)$. Enfin si $C_{k+1}(x) = \emptyset$ alors $\Delta_{k+1}(x) = \lambda_{k+1}(x) = +\infty$.

Un sommet x appartient à O_{k+1} si $\Delta_{k+1}(x) < \Delta_k(x)$. Soit alors γ un chemin de coût minimum dans $C_{k+1}(x)$. Si γ est de longueur inférieure ou égale à k, nous avons $\Delta_k(x) \geq c(\gamma) = \Delta_{k+1}(x)$. Il en résulte que tout chemin de coût minimum dans $C_{k+1}(x)$ est de longueur $k + 1$. Réciproquement, si tout chemin de coût minimum de $C_{k+1}(x)$ est de longeur $k + 1$, alors, d'après l'induction, on a $\Delta_{k+1}(x) < \Delta_k(x)$ et x est bien un sommet ouvert au début de l'étape $k + 1$ (c'est-à-dire $x \in O_{k+1}$). ∎

Nous considérons maintenant le cas où l'on ne sait pas a priori si le graphe valué (G, c) contient ou non un circuit absorbant. La proposition suivante permet alors de déceler l'existence éventuelle d'un tel circuit.

Proposition 2.12. *Si au début de l'étape n, l'ensemble des sommets ouverts n'est pas vide, le graphe valué (G, c) possède un circuit absorbant.*

Preuve. D'après le théorème précédent, O_n est l'ensemble des sommets x pour lesquels tout chemin de coût minimum de $C_n(x)$ possède exactement n arcs. Si le graphe valué (G, c) ne possédait pas de circuit absorbant, il existerait un chemin *élémentaire* de coût minimum de s à x et donc de coût minimum dans $C_n(x)$. ∎

Le tableau suivant montre l'évolution de l'ensemble des sommets ouverts pour le graphe valué de la 2.10. Le circuit $(3, 2, 4, 3)$ est absorbant.

k	0	1	2	3	4	5	6	7
O	$\{0\}$	$\{1,2,3\}$	$\{5,1,6,4\}$	$\{6,5,3\}$	$\{2,6\}$	$\{1,4\}$	$\{3,5\}$	$\{2,6\}$

La complexité d'une étape quelconque de l'algorithme PAPS(G, s) est en $O(m)$

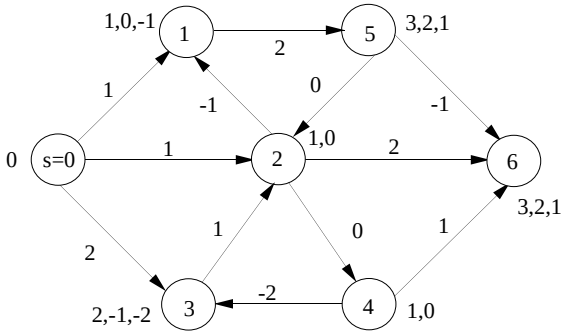


Figure 2.10: *L'algorithme* PAPS *avec un circuit absorbant.*

puisque un sommet est examiné au plus une fois. Comme l'algorithme exécute au maximum n étapes, sa complexité est $O(mn)$.

Notes

La présentation unifiée des algorithmes concernant les arbres couvrants de coût minimum par l'emploi de la *règle des cocycles* et de la *règle des cycles* est due à R.E. Tarjan. Il en est de même de la présentation des algorithmes de recherche des chemins de coût minimum en tant que variantes de l'algorithme «générique» de Ford :

R.E. Tarjan, *Data Structures and Network Algorithms*, SIAM Philadelphia, 1983, chapitres 6 et 7, 71–95.

Les algorithmes sur les arbres couvrant de coût minimum sont assez anciens. Par contre leur complexité a plus récemment été améliorée par l'utilisation de structures de données adaptées comme la gestion des partitions d'un ensemble pour l'algorithme de Kruskal et les files de priorité pour l'algorithme de Prim.

Si l'on restreint l'ensemble des arbres couvrants admissibles en ajoutant des contraintes supplémentaires, le problème devient en général beaucoup plus difficile. Si, par exemple, on limite le degré des sommets, le problème est alors NP-difficile sauf si le degré d'un seul sommet est limité.

Les algorithmes de recherche de chemins de coût minimum ont été très étudiés. Ils interviennent en effet de manière centrale dans beaucoup d'autres domaines comme les flots ou les ordonnancements. Ajouter des contraintes au problème de base le rend souvent beaucoup plus complexe. Par exemple, rechercher un chemin élémentaire de coût minimum entre deux sommets d'un graphe valué quelconque est un problème NP-difficile.

Exercices

7.1. Soit $G = (S, A)$ un graphe non orienté connexe valué par une fonction coût $c : A \mapsto \mathbb{R}$. Soit $H = (S, B)$ un graphe partiel non connexe de G. On appelle *arête de liaison* entre deux composantes connexes C et C' de H un arête qui a l'une de ses extrémités dans C et l'autre dans C'. Soit C une composante connexe de H, $\omega(C)$ le cocycle associé à C et e une arête de coût minimal de $\omega(C)$.
a) Démontrer qu'il existe un arbre couvrant de coût minimum de G qui contient e.
On considère l'algorithme de Sollin décrit par la procédure ci-dessous :

```
procédure SOLLIN(G, c);
  H := (S, ∅);
  tantque H n'est pas connexe faire
    soient C_1, ..., C_p les composantes connexes de H;
    {les C_k ne sont pas marquées au départ};
    tantqu'il existe une composante connexe C_k non marquée faire
      marquer(C_k);
      soit e = {x, y} une arête de coût minimum dans ω(C_k);
      {x est l'extrémité de e qui appartient à C_k}
      soit C_l la composante connexe contenant y;
      marquer(C_l);
      ajouter l'arête e à H
    fintantque
  fintantque.
```

b) Démontrer que l'algorithme précédent applique la *règle des cocycles* à chaque itération de la boucle *tantque* interne.
c) En déduire que lors de la terminaison, H est un arbre couvrant de coût minimum.

7.2. Soit $G = (S, A)$ un graphe orienté complet à n sommets et valué par une fonction coût $c : A \mapsto \mathbb{N}$ telle que :

1. $\forall x, y \in S, \quad c(x,y) = c(y,x)$;
2. $\forall x, y, z \in S, \quad c(x,y) + c(y,z) \geq c(x,z)$.

Un *tour* $t = (t_1, \ldots, t_n)$ de G est une permutation de S et le coût $c(t)$ du tour t est la somme :

$$c(t_n, t_1) + \sum_{k=1}^{n-1} c(t_k, t_{k+1}).$$

On note G' la version non orientée de G où le coût $c'(x,y)$ de l'arête $\{x,y\}$ est égal à $c(x,y)$. Soient H un arbre couvrant de coût minimum de G', $P' = (p_1, \ldots, p_n)$ un parcours en profondeur de G' à partir du sommet s et t le tour de G associé à P'. On note $c'(H)$ le coût de l'arbre H et $c^*(G)$ le coût minimum d'un tour de G.
a) Démontrer que $c^*(G) > c'(H)$.
b) Démontrer que $c(t) \leq 2c'(H)$.
c) Obtient-on le même résultat avec un parcours en largeur de H ?

7.3. On considère le problème de transport monoproduit défini par m fournisseurs F_i et n clients C_j. Le fournisseur F_i dispose d'un stock $a_i \in \mathbb{N}$, le client C_j a une demande $b_j \in \mathbb{N}$ et l'on suppose que $\sum a_i = \sum b_j$. Le coût *unitaire* de transport du fournisseur F_i au client C_j est un entier naturel c_{ij}. Le graphe valué G associé au problème a pour sommets les m fournisseurs et les n clients et ses arcs sont les couples (F_i, C_j) valués par c_{ij}. Un plan de transport $X = (x_{ij})$ est une solution du système :

$$\begin{aligned} \sum_{j \in \{1,\ldots,n\}} x_{ij} &= a_i \quad i \in \{1, \ldots, m\} \\ \sum_{i \in \{1,\ldots,m\}} x_{ij} &= b_j \quad j \in \{1, \ldots, n\} \\ x_{ij} \geq 0 \quad i \in \{1, \ldots, m\} &\quad j \in \{1, \ldots, n\} \end{aligned}$$

et le coût $c(X)$ du plan X est $\sum_{i \in \{1,\ldots,m\}, j \in \{1,\ldots,n\}} c_{ij} x_{ij}$. La matrice du système est notée M.
a) Démontrer qu'il existe un plan de transport optimal entier.
b) Démontrer que le rang de M est $m + n - 1$.

On appelle *solution de base* un ensemble de $m + n - 1$ colonnes indépendantes de M.

c) Montrer que le graphe partiel des arcs (F_i, C_j) associés à ces colonnes est un arbre couvrant de G. La réciproque est-elle vraie ?

Soit H un arbre couvrant de coût minimum de G. Soit $B(H)$ la solution de base associée à H (si elle existe).

d) Montrer qu'il existe un seul plan de transport tel que $x_{ij} = 0$ pour $(F_i, C_j) \notin B(H)$. Montrer que ce plan n'est pas nécessairement optimal.

7.4. Déterminer un graphe valué à m arcs muni d'un sommet origine s tel que l'algorithme « générique » de Ford réalise dans le plus mauvais cas $O(2^m)$ itérations.

7.5. Soit $G = (S, A)$ un graphe, soit s un sommet origine et soit $c : A \mapsto \mathbb{N}$ une fonction capacité. La capacité d'un chemin est la plus petite capacité des arcs de ce chemin. Adapter l'algorithme de Dijkstra à la recherche des chemins de capacité maximale de s à tous les autres sommets de G.

7.6. On considère l'algorithme A^* présenté dans la section 7.2.6. Démontrer que si la fonction d'évaluation par défaut h satisfait :

$$\forall (x, y) \in A, \quad h(x) \leq c(x, y) + h(y),$$

un sommet examiné restera fermé jusqu'à la terminaison de l'algorithme.

7.7. On considère l'algorithme «générique» de Ford et l'on note E l'ensemble des sommets examinés. Démontrer que les invariants suivants sont conservés à chaque itération :

> si $\Delta(x)$ est fini, il existe un chemin *élémentaire* de s à x de coût $\Delta(x)$;
> la *restriction* de la fonction père aux sommets de E est une arborescence partielle du sous-graphe de G induit par E telle que pour tout sommet y de E distinct de la racine $\Delta(p(y)) + c(p(y), y) = \Delta(y)$.

7.8. On appelle *graphe conjonctif* un graphe orienté $G = (S, A)$ valué par $c : A \mapsto \mathbb{R}$, muni d'une racine α et d'une anti-racine ω, et tel qu'il existe pour tout sommet x un chemin positif ou nul de α à x et de x à ω. Un *ensemble de potentiels* d'un graphe conjonctif G est une application $t : S \mapsto \mathbb{R}$ satisfaisant $t(\alpha) = 0$ et :

$$\forall (x, y) \in A, \quad t(y) - t(x) \geq c(x, y)$$

On note $T(G)$ l'ensemble des ensembles de potentiels de G.
a) Démontrer que $T(G)$ est non vide si et seulement si G ne possède pas de circuit strictement positif.
On suppose maintenant que $T(G)$ n'est pas vide et l'on note $r(x)$ le coût *maximum* d'un chemin de α à x.

b) Démontrer que r est le plus petit élément de $T(G)$ pour la relation d'ordre :

$$t \leq t' \Longleftrightarrow \forall x \in S, \quad t(x) \leq t'(x).$$

On considère l'ensemble $D(G)$ des ensembles de potentiels tels que $t(\omega) = r(\omega)$.

c) Démontrer que la fonction f définie par $f(x) = r(\omega) - C(x, \omega)$ où $C(x, \omega)$ est le coût maximum d'un chemin de x à ω est le plus grand élément de $D(G)$.

7.9. Soit $G = (S, A)$ un graphe orienté valué par $c : A \mapsto \mathbb{Q}$. On note $S = \{1, \ldots, n\}$, $S_k = \{1, \ldots, k\}$, G_k le graphe induit par S_k et l'on suppose que $s = 1$ est un sommet origine pour tous les sous-graphes G_k. On note enfin $\alpha_k(i)$ la valeur minimale d'un chemin de 1 à i dans G_k *passant au plus une fois par le sommet* k. On considère alors l'algorithme de Rao décrit ci-dessous :

```
procédure RAO(G, c);
(1)  k := 1; R := ∅; abs := 0;
(2)  si c(1,1) < 0 alors abs := 1
(3)  sinon
(4)     tantque abs = 0 et k ≤ n faire
(5)        k := k + 1; R := R ∪ {k}; T := {k};
(6)        α_k(k) := min{α_{k-1}(j) + c_{jk} | j ∈ Γ⁻_{G_k}(k)};
(7)        tantque abs = 0 et T ≠ R faire
(8)           U(T) := ω⁺_{G_k}(T);
(9)           si U(T) ≠ ∅ alors
(10)             calculer d_pq = min{d_ij | (i,j) ∈ U(T)}
                 {où d_ij = α_k(i) + c(i,j) − α_{k-1}(j)}
(11)          finsi;
(12)          si U(T) = ∅ ou d_pq > 0 alors
(13)             pour tout i dans R − T faire
(14)                α_k(i) := α_{k-1}(i); T := T ∪ {k}
(15)             finpour
(16)          sinon
(17)             α_k(q) := α_k(p) + c(p,q); T := T ∪ {q};
(18)             si (q,k) ∈ A et α_k(q) − α_k(k) + c(q,k) < 0 alors
(19)                afficher(« circuit absorbant » );abs := 1
(20)             finsi
(21)          finsi
(22)       fintantque
(23)    fintantque
(24) finsi.
```

a) Démontrer que la valeur de $\alpha_k(k)$ calculée à la ligne (6) est bien le coût minimum d'un chemin de 1 à k dans G_k passant au plus une fois par k.

b) Démontrer que si $U(T)$ est vide ou si d_{pq} est positif ou nul, le coût minimum d'un chemin de 1 à i dans G_k passant au plus une fois par k est $\alpha_{k-1}(i)$.

c) Démontrer que si d_{pq} est strictement négatif, alors $\alpha_k(p) + c(p,q)$ est le coût minimum d'un chemin de 1 à q dans G_k passant au plus une fois par k.

d) En déduire que l'algorithme RAO détermine les chemins de coût minimum de 1 à tous les autres sommets de G ou détecte la présence d'un circuit absorbant.

Chapitre 8

Flots

Dans ce chapitre nous définissons d'abord la notion de flot compatible et donnons la condition d'existence d'un tel flot. Nous présentons ensuite les propriétés qui interviennent de manière cruciale dans la conception des algorithmes sur les flots, à savoir la décomposition en cycles, l'absence de cycles améliorants dans un flot de coût minimum et la dominance des flots entiers. Nous définissons ensuite le problème du flot maximum d'une source vers un puits. Après avoir défini la notion de graphe d'écart et démontré le théorème de Ford et Fulkerson, nous analysons les algorithmes les plus performants pour ce problème, à savoir l'algorithme primal des distances estimées au puits et l'algorithme dual du préflot. Deux variantes efficaces de l'algorithme du préflot sont décrites, l'algorithme de Karzanov et l'algorithme des excès échelonnés. Nous considérons ensuite le problème du flot de coût minimum. Nous définissons le problème dual et donnons la condition nécessaire et suffisante d'optimalité d'un flot et d'un ensemble de potentiels, exprimée avec les coûts réduits des arcs du graphe d'écart. Nous présentons alors un algorithme primal dû à Goldberg et Tarjan fondé sur la notion d'ϵ-optimalité et sur les circuits de coût moyen minimum. Nous montrons ensuite comment transformer un problème de flot compatible de coût minimum en problème de flot maximum de coût minimum et nous présentons un algorithme dual pour résoudre ce dernier problème. Nous présentons enfin l'algorithme efficace d'Edmonds et Karp pour la recherche d'un plan de transport de coût minimum.

Introduction

La notion de flot modélise de manière naturelle une circulation discrète (voitures, électrons,...) ou continue (fluide,...), le long des arcs d'un graphe orienté. Cette circulation est soumise à des contraintes d'équilibre de charge aux sommets du graphe et de limitation de capacité sur les arcs. De nombreux problèmes d'optimisation se ramènent à la recherche de flots particuliers sur un graphe, les

plus connus sont les problèmes de transport (par exemple entre des fournisseurs et des clients), les problèmes de couplage et d'affectation (par exemple de tâches à des machines) ou encore les problèmes de chemins de coût minimum. Mais ce modèle intervient aussi souvent dans d'autres domaines comme la résolution de certains problèmes d'ordonnancement, le placement de tâches ou encore la maximisation de certaines fonctions booléennes. La théorie des flots occupe une place très intéressante en optimisation combinatoire car elle se situe à la frontière de la programmation linéaire continue et de la programmation en nombres entiers. En effet, les problèmes de flot que nous étudierons peuvent être formulés comme des programmes linéaires continus mais la matrice sous-jacente est telle que les solutions entières sont dominantes.

8.1 Préliminaires

Dans tout le chapitre, les graphes considérés sont orientés mais ne sont pas nécessairement simples, c'est-à-dire que des arcs distincts peuvent avoir la même origine et la même extrémité. Soit u un arc d'un graphe $G = (S, A)$, il sera commode de noter u^- son sommet origine et u^+ son sommet extrémité.

Soit E un ensemble fini. L'ensemble des fonctions de E dans $\mathbb{Z}$ est noté $\mathcal{F}(E)$. Le *produit scalaire* de deux éléments f et g de $\mathcal{F}(E)$ est par définition le nombre

$$f \star g = \sum_{e \in E} f(e)g(e).$$

L'ordre partiel sur $\mathcal{F}(E)$ défini par $\forall e \in E, f(e) \leq g(e)$ est noté $f \leq g$. La relation $f < g$ est définie par $f \leq g$ et $f \neq g$. La *fonction caractéristique* d'un sous-ensemble F de E est notée χ_F.

Soient $G = (S, A)$ un graphe et T un sous-ensemble de sommets. L'ensemble $\omega^-(T)$ des arcs *entrants dans* T et l'ensemble $\omega^+(T)$ des arcs *sortants de* T sont définis par :

$$\begin{aligned} \omega^-(T) &= \{u \in A \mid u^- \in S - T, u^+ \in T\} \\ \omega^+(T) &= \{u \in A \mid u^- \in T, u^+ \in S - T\}. \end{aligned}$$

Soient f une fonction de $\mathcal{F}(A)$ et T un sous-ensemble de sommets, les nombres $f \star \chi_{\omega^-(T)}$ et $f \star \chi_{\omega^+(T)}$ sont notés respectivement $f^-(T)$ et $f^+(T)$. L'*excès* de la fonction f pour la partie T est défini par $e_f(T) = f^-(T) - f^+(T)$. Lorsque T est le singleton $\{s\}$, les notations précédentes sont simplifiées en $f^-(s)$, $f^+(s)$ et $e_f(s)$.

Remarquons que pour toute fonction f de $\mathcal{F}(A)$, l'excès de S est nul et que l'excès d'une partie non vide T de S est égal à la somme des excès des sommets de T, c'est-à-dire $e_f(T) = f^-(T) - f^+(T) = \sum_{s \in T} e_f(s)$.

Comme la notion de chaîne n'a été introduite que pour les graphes non orientés (voir section 4.1.3), nous en définissons ici une extension pour les graphes orientés.

Une *chaîne* γ du graphe $G = (S, A)$ est une suite :

$$\gamma = ((s_0, u_1, s_1), (s_1, u_2, s_2), \cdots, (s_{q-1}, u_q, s_q))$$

telle que pour tout $i \in \{1, \ldots, q\}$ les deux extrémités de l'arc u_i sont s_{i-1} et s_i. Une chaîne est *élémentaire* si tous les sommets s_i pour $i \in \{0, \ldots, q\}$ sont distincts. Un arc u_i de la chaîne γ est dit *avant* si $u_i^- = s_{i-1}$ et $u_i^+ = s_i$, dans le cas contraire il est dit *arrière*. Un *cycle élémentaire* est une chaîne telle que tous les sommets $s_i, i \in \{0, \ldots, q-1\}$ sont distincts et $s_0 = s_q$. Sur la figure 1.1 sont représentés à gauche une chaîne élémentaire $[(1, a, 2), (2, b, 3), (3, c, 4), (4, d, 5)]$ et à droite un cycle élémentaire $[(1, a, 2), (2, b, 3), (3, c, 4), (4, d, 5), (5, e, 6), (6, f, 1)]$.

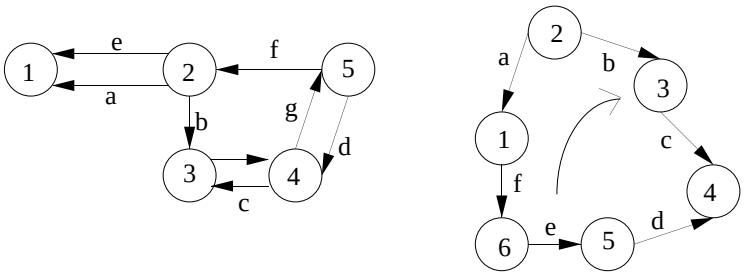


Figure 1.1: *Chaîne et cycle élémentaires.*

8.2 Flots d'un réseau

Un *réseau* $R = (G, a, b)$ est un graphe connexe sans boucles $G = (S, A)$, muni de deux fonctions a et b de $\mathcal{F}(A)$ vérifiant $a \leq b$. Les entiers $a(u)$ et $b(u)$ sont respectivement appelés *capacité minimale* et *capacité maximale* de l'arc u.

Un *réseau valué* $R = (G, a, b, c)$ est un réseau (G, a, b) muni d'une fonction c de $\mathcal{F}(A)$. Le nombre $c(u)$ est appelé *coût* de l'arc u.

Un *flot* f du réseau $R = (G, a, b)$ est une fonction $f : A \mapsto \mathbb{Q}$ vérifiant la *condition d'équilibre* (E) ci-dessous :

$$\forall s \in S \quad e_f(s) = 0 \qquad (E)$$

Un flot *compatible* est un flot qui vérifie également les *conditions de limitation de capacité* (L) suivantes :

$$a \leq f \leq b \qquad (L)$$

Remarque. Le choix de flux rationnels (nous aurions pu également considérer des flux réels) a été fait pour mieux mettre en évidence par la suite la propriété de dominance des flots entiers vis-à-vis du critère coût.

Soient f un flot du réseau R et T un sous-ensemble de sommets. Le nombre $f(u)$ est appelé *flux* de l'arc u, les quantités $f^-(T)$ et $f^+(T)$ sont appelées respectivement *flux entrant dans* T et *flux sortant de* T.

Remarquons que si la condition d'équilibre est satisfaite en tout sommet, elle est également vérifiée pour toute partie non vide T de sommets :

$$e_f(T) = f^-(T) - f^+T) = 0.$$

Le *coût* du flot f d'un réseau valué est par définition le nombre $f \star c$.

Les diverses quantités introduites jusqu'ici s'expriment facilement en utilisant la matrice M d'incidence sommets-arcs du graphe G. La ligne du sommet s représente la fonction $\chi_{\omega^+(s)} - \chi_{\omega^-(s)}$. Un flot compatible d'un réseau $R = (G, a, b)$ à m arcs est donc un vecteur f de $\mathbb{Q}^m$ satisfaisant :

$$Mf = 0 \quad \text{et} \quad a \leq f \leq b.$$

Un flot compatible de coût minimum du réseau valué $R = (G, a, b, c)$ est une solution optimale du programme linéaire :

$$\begin{array}{rl} f & \leq b \\ -f & \leq -a \\ Mf & = 0 \\ MIN & \sum_{u \in A} f(u)c(u) \end{array}$$

où les variables $f(u), u \in A$ sont rationnelles.

L'ensemble des flots d'un réseau constitue un *espace vectoriel* sur $\mathbb{Q}$. Soient f et g deux flots et α et β deux nombres rationnels; si $a \leq \alpha f + \beta g \leq b$ alors $\alpha f + \beta g$ est un flot compatible du réseau.

8.2.1 Existence d'un flot compatible de coût minimum

Les contraintes imposées par les limitations de capacité rendent la question de l'existence d'un flot compatible dans un réseau $R = (G, a, b)$ non triviale. Une condition nécessaire et suffisante d'existence due à Hoffman exprime simplement que pour tout sous-ensemble non vide T de sommets, le flux entrant minimal $a^-(T)$ doit être inférieur ou égal au flux sortant maximal $b^+(T)$. Cette condition de *consistance*, notée (C), s'exprime par :

$$\forall T \subset S, \quad a^-(T) \leq b^+(T). \qquad (C)$$

Soient $R = (G, a, b)$ un réseau et g un flot de ce réseau. La distance $\Delta_g(u)$ de $g(u)$ à l'intervalle $[a(u), b(u)]$ étant définie par :

$$\Delta_g(u) = \begin{cases} g(u) - b(u) & \text{si } g(u) > b(u) \\ a(u) - g(u) & \text{si } g(u) < a(u) \\ 0 & \text{si } a(u) \leq g(u) \leq b(u) \end{cases}$$

l'*incompatibilité* du flot g est mesurée par $i(g) = \Delta_g \star \chi_A$. Le lemme 2.1 montre que si la condition de compatibilité est satisfaite, il existe un flot h dont l'incompatibilité est strictement plus petite que celle de g.

Lemme 2.1. *Soit g un flot entier tel que $i(g) > 0$. Si le réseau est consistant, il existe un flot h tel que $i(h) < i(g)$.*

Preuve. Si $i(g)$ est strictement positif, il existe un arc v vérifiant par exemple $g(v) > b(v)$. On notera respectivement x et y l'origine et l'extrémité de v. Notons $H = (S, B)$ le graphe dont les arcs sont colorés en rouge ou en noir par la procédure COLORER ci-dessous :

```
procédure COLORER(G, a, b, v);
  pour tout arc u de G faire
    si g(u) > a(u) alors
        créer un arc rouge u' d'origine u⁻ et d'extrémité u⁺ dans H;
    si g(u) < b(u) alors
      créer un arc noir u'' d'origine u⁺ et d'extrémité u⁻ dans H.
```

La figure 2.1 montre le graphe H correspondant à un flot nul et à l'arc $v = (4, 1)$. Soit T l'ensemble des sommets accessibles à partir de y dans H et supposons que T

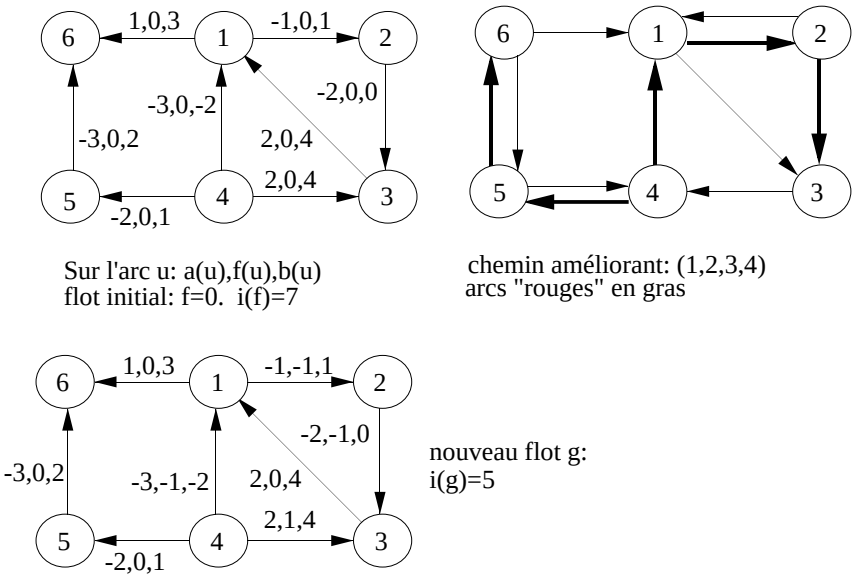


Figure 2.1: *Amélioration de la compatibilité d'un flot.*

ne contienne pas x. Dans le graphe G, un arc u de $\omega^-(T)$ satisfait nécessairement $g(u) \geq b(u)$ et un arc u de $\omega^+(T)$ satisfait nécessairement $g(u) \leq a(u)$. De plus l'arc v appartient à $\omega^-(T)$. Il en résulte que :

$$a^-(S - T) = a^+(T) \geq g^+(T) = g^-(T) > b^-(T) = b^+(S - T).$$

ce qui contredit la condition de compatibilité. Il existe donc un chemin rouge et noir de y à x dans H. Pour tout arc rouge u' de ce chemin posons $h(u) = g(u)-1$, pour tout arc noir u'' de ce chemin, posons $h(u) = g(u)+1$. Posons enfin $h(v) = g(v) - 1$ et $h(u) = g(u)$ pour tous les autres arcs de G. La fonction h est un flot du réseau R tel que $i(h) \leq i(g) - 1$ car on a $\Delta_h(v)=\Delta_g(v) - 1$ et pour tout arc u de G, $\Delta_h(u) \leq \Delta_g(u)$. Pour le flot g de la figure 2.1, le chemin $(1, 2, 3, 4)$ est améliorant et l'incompatibilité diminue de deux unités. ∎

Le théorème de Hoffman résulte alors directement du lemme 2.1.

Théorème 2.2 (de Hoffman)**.** *Un réseau possède un flot compatible si et seulement s'il est consistant.*

Preuve. La condition est suffisante car à partir d'un flot g (par exemple le flot nul), il est possible d'après le lemme 2.1 d'obtenir en au plus $i(g)$ itérations un flot compatible. La condition est nécessaire car si f est un flot compatible et T un sous-ensemble de sommets, la sommation des excès des sommets de T pour f conduit à :

$$0 = f^+(T) - f^-(T) \leq b^+(T) - a^-(T).$$

∎

Si la condition de consistance est satisfaite pour un réseau valué $R = (G, a, b, c)$, nous savons d'après le théorème de Hoffman que l'ensemble de ses flots compatibles n'est pas vide. Cet ensemble formé des vecteurs de $\mathbb{Q}^m$ solutions de $Mf = 0$ et $a \leq f \leq b$, est un *polyèdre convexe fermé et borné* de $\mathbb{Q}^m$. Il résulte alors d'un théorème important de la programmation linéaire continue que la fonction linéaire $f \mapsto f \star c$, où f parcourt l'ensemble des flots compatibles de R, atteint son minimum en un point extrême du polyèdre. Nous en concluons que la condition de consistance est aussi une condition nécessaire et suffisante d'existence d'un flot compatible de coût minimum.

Remarque. Dans la suite, tout réseau $R = (G, a, b)$ sera supposé consistant.

8.2.2 Quelques problèmes particuliers

Nous présentons dans ce paragraphe quelques problèmes d'optimisation dont les solutions peuvent être interprétées comme les flots de coût minimum d'un réseau valué.

Problèmes de transport.

Un *réseau de transport* est défini à partir d'un réseau valué $R = (G, a, b, c)$ et d'une fonction d de $\mathcal{F}(S)$ vérifiant $\sum_{s\in S} d(s) = 0$ appelée fonction *d'offre et de demande*. La valeur $c(u)$ représente le coût unitaire de transport sur l'arc u. Les sommets sont répartis en trois sous-ensembles, les *fournisseurs* pour lesquels $d(s) > 0$, les *clients* pour lesquels $d(s) < 0$ et enfin les *sommets de transit* vérifiant

$d(s) = 0$. Le problème est de trouver un *plan de transport* de coût minimum, c'est-à-dire une fonction g de $\mathcal{F}(A)$ satisfaisant $a \leq g \leq b$ telle que :

$$e_g(s) = \begin{cases} -d(s) & \text{si } s \text{ est un fournisseur} \\ d(s) & \text{si } s \text{ est client} \\ 0 & \text{si } s \text{ est un sommet de transit} \end{cases}$$

et dont le coût soit minimum. Le lecteur vérifiera aisément à partir d'un exemple que les plans de transport sont en bijection avec les flots compatibles d'un réseau valué. La figure 2.2 montre un problème de transport (capacités et coûts non représentés) et le réseau valué équivalent. Les coûts d'un plan de transport et de son flot image par la bijection étant égaux, on est ramené à la recherche d'un flot compatible de coût minimum.

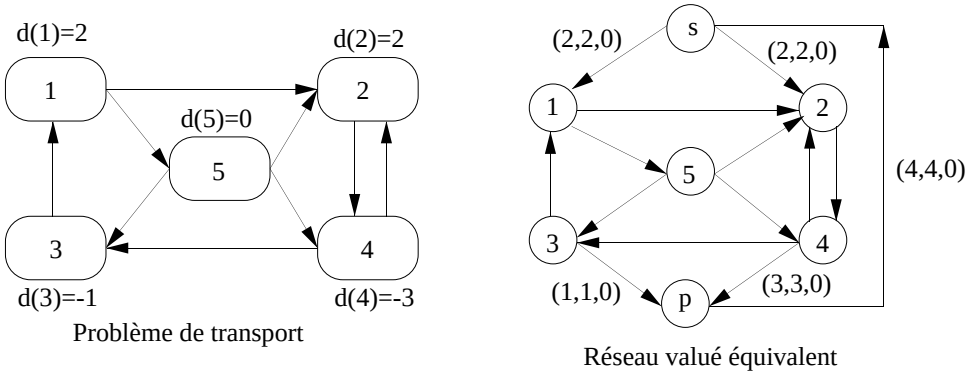


Figure 2.2: *Problème de transport et réseau valué équivalent.*

Chemins de coût minimum.

Soit G un graphe à n sommets dont les arcs sont valués par une fonction coût v et soit s une racine de G. On cherche à déterminer pour chaque sommet x de G un chemin de coût minimum de s à x. Posons $d(x) = -1$ si le sommet x est distinct de s, $d(s) = n - 1$ et pour tout arc u de G, $a(u) = 0, b(u) = n, c(u) = v(u)$. Un plan de transport entier de coût minimum du réseau de transport ainsi constitué correspond à l'envoi par le fournisseur s d'une unité vers chaque client le long d'un chemin de coût minimum.

Flot maximum.

Soit G un graphe dont deux sommets sont distingués, une source s et un puits p, et dont chaque arc u possède une capacité minimale nulle et une capacité maximale $b(u)$. Le problème du flot maximum de s à p est la recherche d'une fonction f, vérifiant $0 \leq f \leq b$ et $e_f(x) = 0$ pour tout sommet x distinct de la source et du puits, telle que la quantité transportée $e_f(p)$ soit maximum. On alloue alors un coût nul à chaque arc et on crée un *arc de retour* d'origine p, d'extrémité s, de coût -1 et de capacité maximale $b^-(p)$. Maximiser la quantité transportée de s à p revient alors à trouver un flot compatible de coût minimum.

Problème d'affectation.

Soient E et F deux ensembles de p éléments et $c(e, f)$ le coût attaché au couple (e, f). Le problème d'affectation de E sur F est la recherche d'une bijection $\varphi : E \to F$ dont le coût total $\sum_{e \in E} c(e, \varphi(e))$ soit minimum. Soit G le graphe complet biparti $(E \cup F, E \times F)$. Munissons chaque arc (e, f) d'une capacité minimale nulle, d'une capacité maximale unité et du coût $c(e, f)$. Posons pour tout $e \in E$, $d(e) = 1$ et pour tout $f \in F, d(f) = -1$. Un plan de transport entier de coût minimum correspond à une affectation optimale.

8.2.3 Trois propriétés fondamentales

L'ensemble des flots d'un réseau possède de nombreuses propriétés algébriques. Nous en présentons trois qui interviennent de manière directe dans les algorithmes de recherche de flots optimaux : la décomposition d'un flot entier en cycles élémentaires, une caractérisation d'un flot compatible de coût minimum et l'existence d'un flot compatible de coût minimum dont tous les flux sont entiers.

Décomposition en cycles

Ce paragraphe concerne une propriété générale des flots d'un réseau. Les fonctions capacité minimale et capacité maximale ne jouent aucun rôle.

Flot entier positif ou nul

Soient $R = (G, a, b)$ un réseau et f un flot entier *strictement positif* de ce réseau. Nous montrons l'existence d'un ensemble $C(f)$ de couples $(\mu, \lambda(\mu))$ où μ est un circuit élémentaire de G et $\lambda(\mu)$ un entier strictement positif, tel que :

$$f = \sum_{\mu \in C(f)} \lambda(\mu) \chi_\mu.$$

Un tel ensemble $C(f)$ est appelé *décomposition du flot* f sur les circuits de G. La fonction χ_μ est appelée par convention *flot canonique* du circuit μ.

Soit $G(f)$ le graphe partiel de G, appelé *graphe support* de f, composé des arcs de G de flux *strictement positif*. Le lemme 2.3 fournit une condition nécessaire et suffisante sur le graphe support pour que le flot f soit nul.

Lemme 2.3. *Un flot entier $f \geq 0$ est nul si et seulement si son graphe support $G(f)$ est sans circuits.*

Preuve. La condition est évidemment nécessaire. Pour la réciproque, nous montrons que si le graphe support est sans circuit, il n'a pas d'arcs. Dans le cas

contraire, il existe au moins un arc u de $G(f)$ dont l'extremité s n'a pas de successeur. Par définition de $G(f)$, on a $f^-(s) > 0$ et $f^+(s) = 0$. La condition d'équilibre n'est donc pas satisfaite pour le sommet s. ∎

Nous pouvons maintenant énoncer le théorème de décomposition.

Théorème 2.4. *Soit $R = (G, a, b)$ un réseau à m arcs. Tout flot entier strictement positif de R est une combinaison linéaire à coefficients entiers strictement positifs d'au plus m flots canoniques de G.*

Preuve. La preuve est constructive. Soit f un flot entier strictement positif et $G(f)$ le graphe support de f. Si $G(f)$ ne possède pas de circuits élémentaires, alors $f = 0$. Dans le cas contraire, nous choisissons un circuit élémentaire μ. La fonction χ_μ est un flot strictement positif de G. Notons ϵ la valeur minimum du flux d'un arc de μ ($\epsilon > 0$). La fonction $h = f - \epsilon\chi_\mu$ est un flot positif ou nul de G et $G(f)$ possède au moins un arc de plus que $G(h)$. En répétant au plus m fois l'opération précédente, on obtient un graphe support qui n'a pas de circuits. Le flot correspondant est donc nul. ∎

Le flot positif de la figure 2.3 se décompose sur les circuits élémentaires (S, A, D, F, P, S), (S, A, D, F, G, P, S), (S, A, D, G, P, S), (S, B, E, H, P, S), (S, C, H, P, S) et (A, D, F, A).

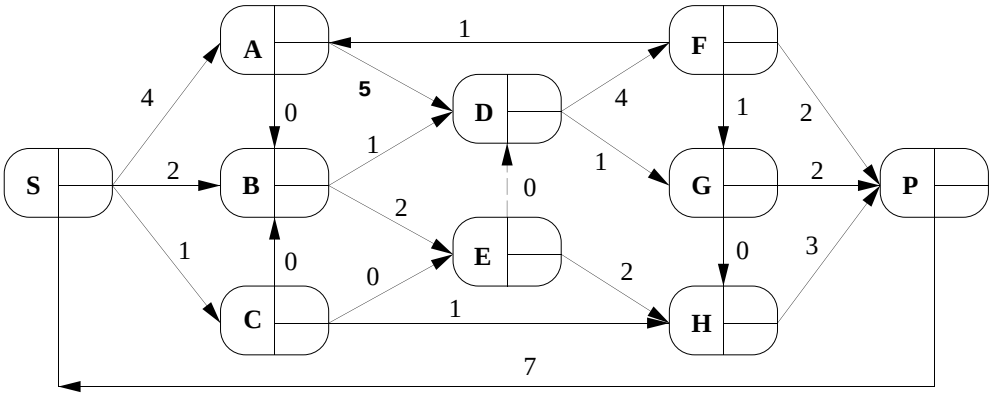


Figure 2.3: *Un flot entier positif.*

Flot entier quelconque

La propriété précédente se généralise au cas d'un flot entier f dont les flux sont de signe quelconque. Le flot f se décompose alors sur les *cycles élémentaires* de G.

Soit γ un cycle élémentaire. Nous notons respectivement γ^- et γ^+) les sous-ensembles des arcs «arrière» et «avant» de γ. La fonction ϕ_γ définie par $\phi(\gamma) = \chi_{\gamma^+} - \chi_{\gamma^-}$, qui satisfait les conditions d'équilibre en chaque sommet, est appelée *flot canonique* du cycle γ.

Théorème 2.5. *Soit f un flot entier d'un réseau $R = (G, a, b)$. Il existe un ensemble $C(f)$ d'au plus m cycles élémentaires de G tel que :*

a) *le flot f est une combinaison linéaire à coefficients entiers strictement positifs des flots canoniques des cycles de $C(f)$,*
b) *les flux (non nuls) alloués à un arc u de G par les flots canoniques de la décomposition ont tous le signe de $f(u)$.*

Preuve. Soient $G = (S, A)$ un graphe et soit B une partie de A. Nous notons alors $R' = (G', a', b')$ le réseau obtenu à partir de R en *remplaçant* chaque arc u de B par un arc opposé noté u' de capacité minimale $a'(u') = -b(u)$ et de capacité maximale $b'(u') = -a(u)$.

On note A' l'ensemble des arcs de G' et B' la partie des arcs de A' provenant de l'inversion d'un arc de B. La fonction f' de $\mathcal{F}(A')$ définie par :

$$f'(v') = \begin{cases} f(v) & \text{si } v' \in A' \setminus B' \\ -f(v) & \text{si } v' \in B' \end{cases}$$

est par construction un flot du réseau R'. De plus l'application définie par $\psi_{A,B}$: $f \in \mathcal{F}(A) \mapsto f' \in \mathcal{F}(A')$ est linéaire et satisfait :

$$\psi_{A',B'} \circ \psi_{A,B} = id_{\mathcal{F}(A)}.$$

Nous supposons qu'au moins un flux de f est strictement négatif et nous appliquons la transformation précédente à l'ensemble B des arcs de G dont le flux est strictement négatif. D'après le théorème 2.4, le flot f' se décompose sur les flots canoniques d'une famille $C'(f')$ d'au plus m *circuits élémentaires* de G'. A tout circuit μ' de G' correspond un cycle μ de G et l'on a $\psi_{A',B'}(\Phi_{\mu'})=\Phi_{\mu}$.

La propriété de décomposition de f résulte alors de la linéarité de $\psi_{A',B'}$. Si $f(u)$

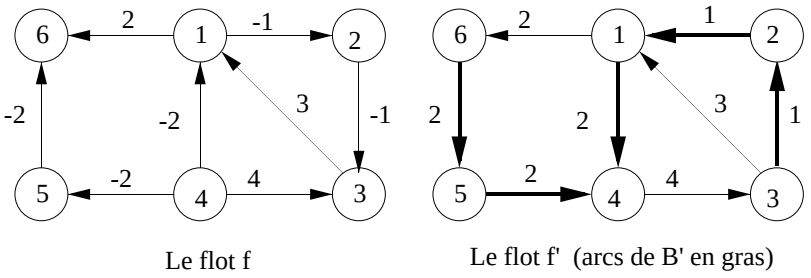


Figure 2.4: *L'application $\psi_{A,B}$.*

est strictement négatif (respectivement strictement positif), pour tout circuit de G' passant par l'arc u', on a $\psi_{A',B'}(\Phi_{\mu'})(u)=-1$ (respectivement $+1$). Il en résulte la condition *b)* du théorème.

Sur l'exemple de la figure 2.4, le flot positif f' est égal à $2\chi_{\mu_1} + \chi_{\mu_2} + \chi_{\mu_3}$ où $\mu_1=(1, 4, 3, 1)$, $\mu_2=(1, 6, 5, 4, 3, 1)$ et $\mu_3=(1, 6, 5, 4, 3, 2, 1)$ sont trois circuits de G'. Il en résulte que le flot f est égal à $2\phi_{\gamma_1} + \phi_{\gamma_2} + \phi_{\gamma_3}$ où γ_1, γ_2 et γ_3 sont les trois cycles de G associés aux circuits μ_1, μ_2 et μ_3 de G'. ∎

Conditions d'optimalité d'un flot compatible

Soit $R = (G, a, b, c)$ un réseau valué consistant. La notion de *cycle augmentant* est utile pour comparer les coûts de deux flots compatibles de ce réseau. Un cycle élémentaire μ est dit *augmentant* pour le flot compatible f s'il existe un entier positif k tel que $a \leq f + k\phi_\mu \leq b$. Si de plus le *coût unitaire* $c \star \phi_\mu$ est strictement négatif, le cycle μ est dit *améliorant*.

Proposition 2.6. *Soit R un réseau valué et soient f et g deux flots compatibles de ce réseau. Le flot $f - g$ se décompose sur un sous-ensemble d'au plus m cycles augmentant de g.*

Preuve. Soit $h = f - g$. D'après le théorème 2.5, il existe r ($r \leq m$) cycles élémentaires $\mu_1, \ldots, \mu_r$ de G et r nombres entiers strictement positifs $k_1, \ldots, k_r$ tels que : $h = \sum_{i=1}^{r} k_i \phi_{\mu_i}$. Comme $f = g + h$, nous avons :

$$a \leq g + \sum_{i=1}^{r} k_i \phi_{\mu_i} \leq b.$$

Soient μ_j l'un de ces cycles et u un arc de μ_j. D'après le théorème 2.5, tous les nombres $\phi_{\mu_i}(u), i \in \{1, \ldots, r\}$ sont de même signe. S'ils sont tous positifs ou nuls on a :

$$a(u) \leq g(u) + k_j \phi_{\mu_j}(u) \leq b(u)$$

car $g(u) \geq a(u)$ et $k_j \phi_{\mu_j}(u) \leq \sum_{i=1}^{r} k_i \phi_{\mu_i}$. S'ils sont tous négatifs ou nuls on a la même inégalité car $g(u) \leq b(u)$ et $k_j \phi_{\mu_j}(u) \geq \sum_{i=1}^{r} k_i \phi_{\mu_i}$. ∎

La proposition 2.6 nous permet maintenant de caractériser un flot compatible de coût minimum.

Théorème 2.7. *Un flot compatible f est de coût minimum si et seulement s'il ne possède pas de cycle améliorant.*

Preuve. La condition nécessaire est immédiate. Soit f un flot compatible et soit g un flot compatible de coût strictement inférieur. Posons $h = g - f$. D'après la proposition 2.6, il existe des cycles $\mu_1, \ldots, \mu_r$ augmentant pour f et des nombres entiers positifs $k_1, \ldots, k_r$ tels que :

$$c \star g = c \star f + \sum_{i=1}^{r} k_i (c \star \phi_{\mu_i}).$$

Il existe donc un j tel que $c \star \phi_{\mu_j}$ soit strictement négatif et le cycle μ_j est améliorant pour f. Contradiction. ∎

Soient f un flot compatible et μ un cycle de coût unitaire non nul. Si le flux de chaque arc du cycle n'est pas égal à l'une de ses deux bornes, le flot f n'est pas de coût minimum. Il est possible en effet soit d'augmenter soit de diminuer d'une

même quantité strictement positive tous les flux des arcs de μ et l'une de ces deux transformations conduit à un flot compatible strictement meilleur.

Un arc u de A est dit *libre* vis-à-vis du flot f si $a(u) < f(u) < b(u)$, dans le cas contraire il est dit *bloqué*. Un cycle élémentaire est *libre* si tous ses arcs sont libres.

Proposition 2.8. *Un réseau valué consistant possède un flot de coût minimum sans cycles libres. Le graphe partiel des arcs libres peut être complété par des arcs bloqués pour former un arbre (appelé arbre-base).*

Preuve. Soit R un réseau valué consistant et f un flot compatible de coût minimum. S'il existe un cycle γ libre, ce cycle est de coût nul. Posons alors $\alpha = \min\{f(u) - a(u) \mid u \in \gamma^+\}$, $\beta = \min\{b(u) - f(u) \mid u \in \gamma^-\}$, $\epsilon = \min\{\alpha, \beta\}$ et $g = f - \epsilon\Phi_\gamma$. La fonction g est un flot compatible de coût minimum de R qui possède au moins un arc libre de moins que f. En répétant cette transformation, on aboutit à un flot compatible de coût minimum sans cycle libre.

Dominance des flots entiers

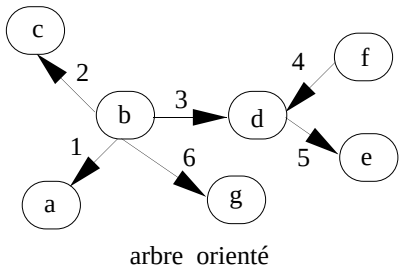
Soit $R = (G, a, b, c)$ un réseau valué consistant. Nous montrons qu'il existe un flot compatible de coût minimum dont tous les flux sont entiers. On dit encore que les flots entiers sont *dominants*. La preuve s'appuie sur une propriété de la matrice d'incidence sommets-arcs d'un *arbre orienté*, c'est-à-dire un graphe obtenu par orientation des arêtes d'un arbre (voir figure 2.5), et sur la proposition 2.8.

Proposition 2.9. *Soit $H = (T, B)$ un arbre orienté. Le graphe H possède une matrice d'incidence triangulaire inférieure dont tous les éléments diagonaux sont non nuls.*

Preuve. Nous raisonnons par récurrence sur $n = \mathrm{Card}(T)$. La propriété est vraie pour $n = 2$. Soit H un arbre orienté à n sommets ($n > 2$), t une feuille de H, u l'arc incident à cette feuille et H_t le sous-graphe induit par $T - \{t\}$. Le graphe H_t est un arbre orienté à $n - 1$ sommets qui possède (par induction) une matrice d'incidence triangulaire inférieure M_t satisfaisant la proposition. La matrice d'incidence M de H obtenue en ajoutant à M_t la colonne u et le sommet t comme première ligne et première colonne satisfait également la proposition. ∎

La figure 2.5 illustre la construction d'une telle matrice d'incidence.

Proposition 2.10. *Un réseau valué consistant possède un flot de coût minimum dont tous les flux sont entiers.*



arbre orienté

	6	5	4	3	2	1
g	-1	0	0	0	0	0
e	0	-1	0	0	0	0
f	0	0	1	0	0	0
d	0	1	-1	-1	0	0
c	0	0	0	0	-1	0
b	1	0	0	1	1	1
a	0	0	0	0	0	-1

Figure 2.5: *Matrice d'incidence d'un arbre orienté.*

Preuve. Soient R un réseau valué consistant. D'après la proposition 2.8, il existe un flot compatible f de coût minimum sans cycle libre et un arbre-base pour f dont les arcs constituent l'ensemble H. Nous notons alors K l'ensemble des arcs du coarbre. D'après la proposition 2.9, il existe une matrice d'incidence M de G composée d'une matrice d'incidence de H notée M_H vérifiant les propriétés de la proposition 2.9 et d'une matrice d'incidence de K notée M_K. La condition d'équilibre satisfaite par le flot compatible f s'écrit (en utilisant les notations habituelles) :

$$M_H f_H = -M_K f_K.$$

Le vecteur f_K est entier puisque les arcs de K sont bloqués pour le flot compatible f. Le vecteur f_H, solution unique de l'équation d'équilibre, est donc également entier. ∎

Il résulte de la proposition 2.10 que la recherche d'un flot compatible de coût minimum peut être réalisée dans le sous ensemble des flots compatibles entiers du réseau.

8.3 Problème du flot maximum

Un problème de *flot maximum* noté (G, b, s, p) est spécifié par la donnée d'un graphe orienté $G = (S, A)$ connexe sans boucles, de deux sommets distincts s et p appelés respectivement *source* et *puits* et d'une fonction capacité maximale b telle que pour tout arc u de A, la capacité $b(u)$ soit un entier strictement positif. Un *flot de s à p* est une fonction g de $\mathcal{F}(A)$ satisfaisant les contraintes (D) ci-dessous :

$$\begin{aligned} e_g(s) &\leq 0 \\ e_g(p) &\geq 0 \\ 0 \leq g &\leq b \\ \forall x \in S - \{s, p\}, \quad e_g(x) &= 0. \end{aligned} \qquad (D)$$

La *valeur* de g est définie par $\hat{g}{=}e_g(p)$ et le problème est la recherche d'un flot g de sà p de valeur maximum.

En munissant le graphe G d'un arc *de retour* noté u_0 d'origine p, d'extrémité s et de capacité maximale $b^-(p)$, on obtient un réseau associé au problème (G, b, s, p). Les flots compatibles de ce réseau sont en bijection avec les fonctions g solutions de (D). De plus si f est le flot du réseau associé à g dans la bijection, on a $e_g(p) = f(u_0)$. Le problème revient donc à déterminer un flot compatible du réseau associé dont le flux de l'arc de retour est maximum.

Nous emploierons dans la suite le terme flot pour désigner un flot compatible du réseau associé.

Le problème du flot maximum a été très étudié et a donné lieu à des algorithmes de plus en plus efficaces depuis les travaux de Ford et Fulkerson. On peut regrouper ces algorithmes en deux classes, les méthodes primales qui construisent une suite de flots de valeur croissante et les méthodes duales qui construisent une suite de flots approchés, appelés *préflots*, dont le dernier est un flot maximum. Nous présentons d'abord l'algorithme «générique» de Ford et Fulkerson, puis un algorithme efficace représentatif de chacune des classes, l'algorithme primal des distances estimées au puits et l'algorithme dual du préflot. Nous terminons par deux variantes très efficaces de l'algorithme du préflot, l'algorithme de Karzanov et l'algorithme des excès échelonnés.

Nous commençons par une simplification naturelle du réseau. Soit (G, b, s, p) un problème de flot maximum de s à p. Comme nous l'avons vu dans la section 8.2.2, le problème du flot de valeur maximum de s à p est un cas particulier du problème du flot compatible de coût minimum. Il existe donc d'après la proposition 2.10 un flot f^* de valeur maximum dont tous les flux sont entiers. D'après le théorème 2.4, le flot f^* est une combinaison linéaire à coefficients entiers strictement positifs des flots canoniques de circuits de G. Or seuls les circuits de la décomposition passant par l'arc de retour u_0 contribuent à la valeur de f^*. Il en résulte que le flot obtenu en ne retenant dans la décomposition de f^* que les circuits passant par u_0 est aussi de valeur maximum. Il existe donc un flot de valeur maximum qui est une combinaison linéaire à coefficients entiers strictement positifs de circuits passant par l'arc de retour u_0.

Soit x un sommet qui n'appartient pas à un chemin de s à p. Le graphe obtenu en supprimant x et tous les arcs adjacents à x contient encore tous les circuits de G passant par u_0. La valeur maximum d'un flot de s à p dans le nouveau graphe est donc la même que celle associée au graphe G. Nous pouvons donc supposer sans perte de généralité que tous les sommets du graphe G sont accessibles de s et co-accessibles de p.

La figure 3.1 présente le réseau associé à un problème de flot maximum et un flot f de ce réseau. A coté de chaque arc u sont inscrites sa capacité maximale $b(u)$ (entre parenthèses) et le flux $f(u)$.

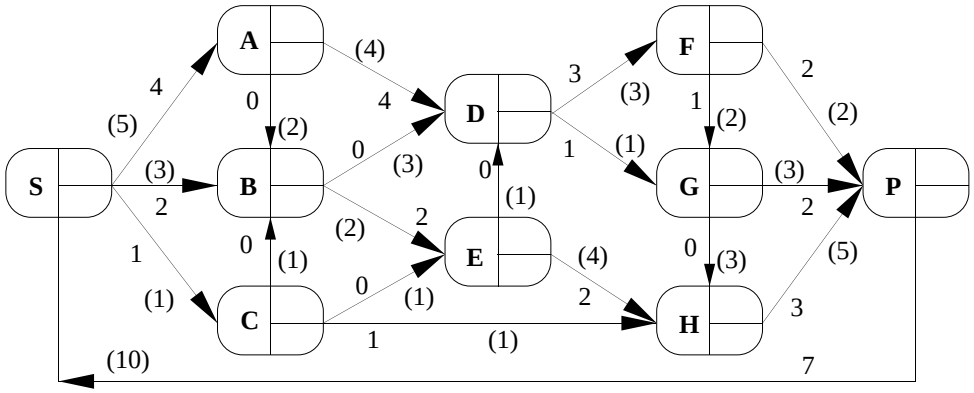


Figure 3.1: *Un problème de flot maximum et l'un de ses flots.*

Graphe d'écart d'un flot f

Les algorithmes de résolution du problème du flot maximum sont fondés sur la recherche de chaînes dites *améliorantes* le long desquelles sont réalisées des augmentations de flux. Comme nous le verrons, l'efficacité de certains algorithmes augmente si les améliorations de flot sont réalisées sur des chaînes de longueur minimum en nombre d'arcs. Pour ramener la recherche d'une chaîne améliorante à celle d'un chemin, on définit un outil très utile : le graphe d'écart. Soit R le réseau d'un problème de flot maximum (G, b, s, p) et soit f un flot de ce réseau. Le *graphe d'écart* d' $G_f = (S, A_f)$ du flot f est un graphe *valué* ayant les mêmes sommets que G et dont les arcs sont définis à partir des arcs de G par la procédure GRAPHE-D'ÉCART suivante :

```
procédure GRAPHE-D'ÉCART(G, f);
  A_f := ∅;
  pour tout arc u de G faire
    si f(u) < b(u)
      alors insérer dans A_f l'arc u' valué par r(u') = b(u) − f(u)
      d'origine u⁻ et d'extrémité u⁺
    finsi;
    si f(u) > 0
      alors insérer dans A_f l'arc u'' valué par r(u'') = f(u)
      d'origine u⁺ et d'extrémité u⁻;
    finsi;
  finpour;
  retourner(A_f).
```

Un arc u libre pour f produira donc les deux arcs u' et u'' dans G_f, un arc u saturé produira seulement l'arc u'' et un arc u vide produira seulement l'arc u'.

Un arc u' est appelé *représentant conforme* de l'arc u, un arc u'' est appelé *représentant non conforme* de l'arc u. Les arcs u' et u'' sont dits *conjoints*. La figure 3.2 montre le graphe d'écart du flot de la figure 3.1.

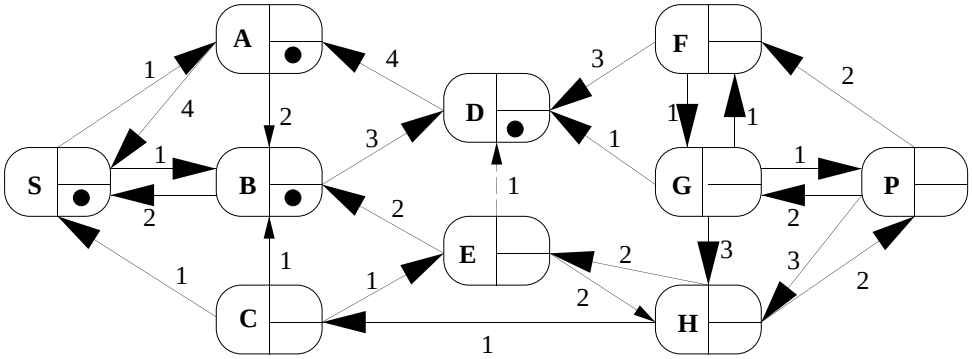


Figure 3.2: *Un graphe d'écart.*

Donnons quelques propriétés simples du graphe d'écart. Tout arc v du graphe d'écart est le représentant conforme ou non conforme d'un arc unique du graphe G appelé son *père*. La notation condensée $v = u'$ (respectivement $v = u''$) signifiera que l'arc v de G_f est le représentant conforme (respectivement non conforme) de l'arc u de G. Un arc u de G a donc pour le flot f un ou deux *représentants* dans G_f. Chacun de ces représentants a une valuation strictement positive et la somme des valuations des représentants de u est toujours égale à $b(u)$.

Soit u un arc de G. Faisons varier son flux $f(u)$ de 0 à $b(u)$. Si $f(u) = 0$, l'arc u' existe et l'arc u'' n'existe pas dans G_f. Si $f(u)$ augmente mais n'atteint pas la valeur $b(u)$, l'arc u'' apparait dans G_f, enfin lorsque $f(u) = b(u)$, l'arc u' disparait de G_f. Une augmentation de flux est donc susceptible de faire apparaître et/ou disparaître un arc du graphe d'écart. Il en est bien sûr de même d'une diminution de flux.

La fonction fondamentale du graphe d'écart G_f est de ramener la recherche d'un flot meilleur que f à celle d'un chemin de s à p dans G_f. Un tel chemin est appelé *chemin améliorant*. Soit μ un chemin améliorant et α la plus petite valuation des arcs de ce chemin, le lemme 3.1 montre que la nouvelle fonction f calculée par la procédure AUGMENTER-FLOT ci-dessous est un flot strictement meilleur.

```
procédure AUGMENTER-FLOT(f, μ);
  α :=min{r(v) | v arc de μ};
  pour chaque arc v de μ faire
    si v = u' alors f(u) := f(u) + α finfaire
    si v = u'' alors f(u) := f(u) − α finfaire
  finpour
  f(u_0) :=f(u_0) + α.
```

Lemme 3.1. *S'il existe un chemin améliorant dans G_f, le flot f n'est pas de valeur maximum.*

Preuve. Soit μ un chemin améliorant du graphe d'écart G_f. Notons α ($\alpha > 0$) la plus petite valuation des arcs de ce chemin. Appelons μ^+ (respectivement μ^-) les arcs conformes (respectivement non conformes) du chemin μ. Si nous notons g la nouvelle fonction f à l'issue de la procédure AUGMENTER-FLOT, g est un flot par définition des arcs conformes et non conformes et nous avons $\hat{g} = \hat{f} + \alpha > \hat{f}$ puique $\alpha > 0$. ∎

Il est important de remarquer que la procédure AUGMENTER-FLOT supprime au moins un arc du graphe d'écart G_f. En effet, si α est la valuation d'un arc $v = u'$ de μ^+, on a $\alpha = b(u) - f(u)$ et $g(u) = b(u)$. L'arc v de G_f n'appartient donc pas à G_g. Si α est la valuation d'un arc $v = u''$ de μ^-, on a $\alpha = f(u)$ et $g(u) = 0$. L'arc v de G_f n'appartient donc pas à G_g.

8.3.1 L'algorithme générique de Ford et Fulkerson

Tous les algorithmes de résolution du problème de flot maximum utilisent comme critère de terminaison un théorème important, dû à Ford et Fulkerson, que l'on peut rapprocher du théorème de la dualité en programmation linéaire continue.

Les objet duaux des flots sont appelés *coupes*. Une coupe est un sous-ensemble C d'arcs de G tel que $C = \omega^+(T)$ où T est un sous-ensemble de sommets contenant s et ne contenant pas p. La valeur d'une coupe est par définition $b(C)$. La figure 3.3 montre une coupe du réseau de la figure 3.1. Cette coupe, constituée des arcs de $\omega^+\{S, A, B, C\}$, a la valeur 10. Au problème de recherche d'un flot de valeur

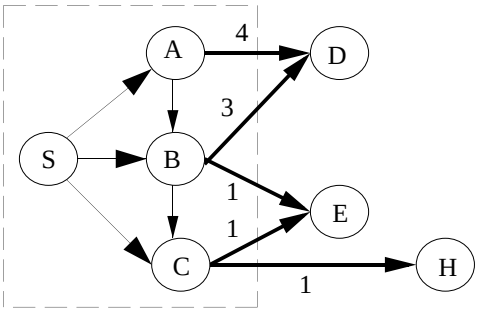


Figure 3.3: *Une coupe.*

maximum correspond par dualité le problème de recherche d'une coupe de valeur minimum. Le théorème de Ford et Fulkerson montre que ces deux valeurs sont égales.

Théorème 3.2 (de Ford et Fulkerson)**.** *La valeur maximum d'un flot est égale à la valeur minimum d'une coupe.*

Preuve. Nous montrons d'abord que la valeur $b(C)$ d'une coupe $C = \omega^+(T)$ $(s \in T, t \notin T)$ est supérieure ou égale à la valeur $\hat{f}$ d'un flot f. En effet, par sommation des équations d'équilibre sur les sommets de T, il vient :

$$f^-(T) = f(u_0) + \sum_{u \in \omega^-(T) - \{u_0\}} f(u) = f^+(T) \leq b(C).$$

Il en résulte : $\hat{f} \leq b(C)$.

Supposons maintenant que f soit un flot maximum. D'après le lemme 3.1, il n'existe pas de chemin de s à p dans le graphe d'écart G_f. Notons alors T l'ensemble des sommets accessibles à partir de s dans G_f. Un arc u de G sortant de T est saturé car dans le cas contraire, son représentant conforme serait un arc sortant de T dans G_f. Un arc u de G entrant dans T est vide car dans le cas contraire, son représentant non conforme serait, lui, un arc sortant de T dans G_f. Il en résulte que :

$$f^-(T) = f(u_0) = f^+(T) = b(C).$$

La coupe C est donc de valeur minimum. ∎

On déduit directement du lemme 3.1 et du théorème 3.2 la condition nécessaire et suffisante suivante :

Proposition 3.3. *Un flot f est maximum si et seulement s'il n'existe pas de chemin améliorant dans G_f.*

Le théorème précédent conduit très naturellement à un algorithme générique, dû à Ford et Fulkerson.

```
procédure FORD-FULKERSON (G, b, s, p);
   f := 0;
   tantque p est accessible à partir de s dans G_f faire
      déterminer un chemin améliorant μ de G_f;
      AUGMENTER-FLOT (f, μ)
   fintantque.
```

Cet algorithme se termine puisqu'à chaque itération la valeur du flot, majororée par $b^-(p)$, augmente au moins d'une unité. D'après le théorème de Ford et Fulkerson, le dernier flot obtenu est de valeur maximum. Le graphe d'écart de la figure 3.2 montre que le flot est maximum puisque l'ensemble des sommets accessibles à partir de S, pointés d'un rond noir, ne contient pas P.

En laissant totalement libre le choix du chemin améliorant, l'algorithme précédent peut réaliser un très grand nombre d'itérations. Sur l'exemple de la figure 3.4,

l'algorithme peut, à partir du flot nul, augmenter la valeur du flot alternativemnt sur le chemin $(1, 2, 4, 3)$ et sur la chaîne $(1, 4, 2, 3)$. Le flot maximum est alors obtenu après M itérations alors que deux itérations suffisent. Pour transformer cet algorithme «générique» en un algorithme efficace, il faut contraindre le choix du chemin améliorant. Un critère judicieux, dû à Edmonds et Karp, consiste à choisir un chemin améliorant de longueur minimum (en nombre d'arcs). Nous montrerons en effet que la longueur d'un *plus court chemin améliorant* ne peut pas croître d'une itération sur l'autre pourvu que l'augmentation de flot soit réalisée le long d'un tel chemin. De plus, après au plus m itérations (où m est le nombre d'arcs de G), cette diminution est stricte.

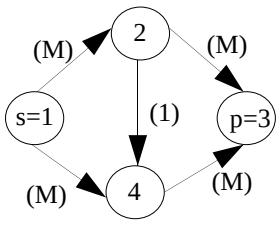


Figure 3.4: *Un problème de flot maximum.*

8.3.2 L'algorithme des distances estimées au puits

La faiblesse essentielle de l'algorithme générique est de ne retenir aucune information sur les chemins améliorants du graphe d'écart d'une itération sur l'autre alors que deux graphes d'écart successifs sont souvent très voisins. Le principe de l'algorithme des distances estimées au puits est de calculer, en chaque sommet du graphe d'écart, une évaluation par défaut de sa distance au puits. On peut alors calculer efficacement un *plus court chemin améliorant*.

Distance estimée et graphe d'admissibilité

Soient $G_f = (S, A_f)$ le graphe d'écart d'un flot f et n le nombre de sommets de G_f. Nous appelons *distance estimée* une application $\Delta : S \to \mathbb{N}$ telle que :

a) $\Delta(p) = 0$;

b) pour tout arc v de G_f, $\Delta(v^-) \leq \Delta(v^+) + 1$.

Si le sommet p est accessible à partir du sommet x dans G_f, la distance estimée du sommet x est une évaluation par défaut de la longueur minimum (en nombre d'arcs) d'un chemin de x à p dans G_f. Il en résulte que si $\Delta(y) \geq n$ pour $y \in S$, il n'existe pas de chemin de y à s dans G_f.

Un arc v de G_f est dit *admissible* si $\Delta(v^-) = \Delta(v^+) + 1$. Un *chemin admissible* est un chemin améliorant dont les arcs sont admissibles. Un chemin admissible est donc un plus court chemin améliorant. Nous appellerons *graphe d'admissibilité*

le graphe partiel de G_f formé des arcs admissibles. Le graphe d'admissibilité ne possède pas de circuits et aucun arc admissible n'a le sommet p comme origine.

Initialisation

Le flot initial est le flot nul et la distance estimée initiale d'un sommet x est la longueur minimum d'un chemin de x à p dans G_0. Notons que cette longueur est définie pour tout sommet x puisque tout sommet est par hypothèse co-accessible de p. Le *chemin courant* initial est réduit au sommet s. La procédure INITIALISER-DEP réalise cette initialisation.

```
procédure INITIALISER-DEP(G, b, s, p) ;
   μ := (s) ; f := 0 ; z := s ;
   pour tout sommet x de G faire Δ(x) := l(x, p, G_0).
```

On note $l(x, p, G_0)$ la longueur minimum d'un chemin de x à p dans G_0, μ le chemin courant, f le flot courant, et z l'extrémité de μ.

Itération courante

Soit μ le chemin courant et z son extrémité. Une itération de l'algorithme consiste à prolonger μ à partir de son extrémité dans le graphe d'admissibilité jusqu'à ce que l'extrémité z du chemin soit une sortie du graphe d'admissibilité.

Si $z = p$, le chemin μ est un plus court chemin améliorant. On améliore alors le flot le long de ce chemin, on ne modifie pas la distance estimée Δ et l'on réinitialise le chemin courant μ à (s).

Si $z \neq p$, on augmente *strictement* la distance estimée du sommet z et on supprime le dernier arc de μ qui n'est plus admissible.

La procédure ITÉRER-DEP ci-dessous réalise l'itération courante.

```
procédure ITÉRER-DEP ;
   PROLONGER(μ) ;
   soit z l'extrémité de μ ;
   si z = p
      alors AUGMENTER-FLOT(f, μ) ; μ := (s)
      sinon AUGMENTER-DISTANCE(z) ; SUPPRIMER-DERNIER-ARC(μ)
   finsi.
```

La procédure PROLONGER(μ) transforme le chemin μ en un chemin μ' du graphe d'admissibilité qui admet μ comme sous-chemin initial et dont l'extrémité z n'est pas l'origine d'un arc admissible. Sa complexité est de l'ordre du nombre d'arcs ajoutés.

Lorsque μ est un chemin améliorant, la procédure AUGMENTER-FLOT améliore le flot le long du chemin μ et met à jour le graphe d'écart. Comme le chemin μ est élémentaire, il comporte au plus $n-1$ arcs. La complexité de la procédure AUGMENTER-FLOT est donc $O(n)$.

Lorsque μ n'est pas un chemin améliorant, la procédure AUGMENTER-DISTANCE augmente la valeur de $\Delta(z)$ comme suit :

$$\Delta(z) := \begin{cases} n & \text{si } S_f(z) \text{ est vide} \\ \min_{y \in S_f(z)}\{1 + \Delta(y)\} & \text{sinon} \end{cases}$$

($S_f(z)$ est l'ensemble des successeurs de z dans G_f d'origine z) et met à jour le graphe d'admissibilité. La figure 3.5, où les distances estimées sont inscrites à côté de chaque sommet et où les arcs de μ sont épais, montre cette mise à jour. Si le sommet z a k successeurs dans G, la complexité de AUGMENTER-DISTANCE est $O(k)$.

Après l'exécution de AUGMENTER-DISTANCE, le dernier arc de μ n'est plus admissible. La procédure SUPPRIMER-DERNIER-ARC supprime le dernier arc de μ si la longueur de μ est strictement positive et laisse μ inchangé sinon. Le nouveau chemin μ est donc admissible. La complexité de la procédure SUPPRIMER-DERNIER-ARC est $O(1)$.

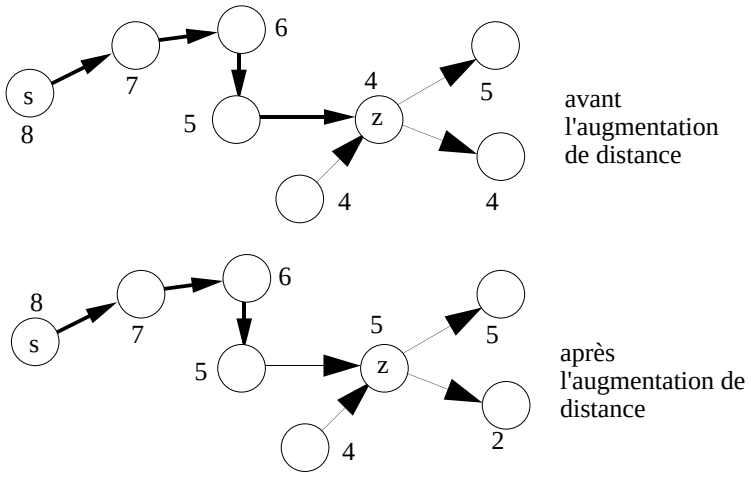


Figure 3.5: *Mise à jour de la distance estimée*

Terminaison

L'algorithme se termine lorsque la distance estimée du sommet s est supérieure ou égale à n. Par définition de la distance estimée, il n'existe plus alors de chemin

améliorant et le flot obtenu est maximum. Il en résulte la procédure DEP ci-dessous :

```
procédure DEP ;
   INITIALISER-DEP
   tantque Δ(s) < n faire
      ITÉRER-DEP
   fintantque.
```

Les figures 3.6, 3.7 et 3.8 illustrent le déroulement de l'algorithme des distances estimées au puits sur le réseau de la figure 3.1. La figure 3.6 montre en (A) le graphe d'écart du flot nul initial, les distances estimées (coin nord-est du sommet) et le graphe d'admissibilité (arcs épais). L'algorithme réalise ensuite une augmentation de flot pour le chemin admissible (S, C, H, P) puis augmente la distance estimée du sommet S. Le graphe d'écart à l'issue de cette première augmentation de la distance estimée est représenté en (B). Suivent ensuite trois augmentations de flot pour les chemins (S, A, D, F, P), (S, B, E, H, P) et (S, A, D, G, P) à l'issue desquelles la valeur du flot est 6 et le graphe d'écart celui de la figure 3.7. L'algorithme augmente alors la distance du sommet F et réalise ensuite un augmentation de flot sur le chemin (S, A, D, F, G, P). Le flot est alors optimal comme le montre le graphe d'écart de la figure 3.8 où les sommets accessibles à partir de S sont pointés. L'algorithme n'est cependant pas terminé car il reste encore des augmentations de distance à exécuter pour que la distance estimée de S devienne supérieure ou égale à $n = 10$.

Convergence et complexité

Après avoir décrit l'algorithme des distances estimées, nous montrons maintenant que cet algorithme se termine, qu'il calcule un flot de valeur maximum et que sa complexité est $O(n^2m)$.

Quelques propriétés seront utiles pour parvenir à ces résultats. Considérons la suite des fonctions Δ calculées par la procédure DEP. On note Δ_0 la fonction Δ initiale et Δ_k la fonction Δ à l'issue de l'itération k.

Proposition 3.4. *La suite Δ_k est une suite croissante de fonctions «distance estimée» . Si l'itération k réalise une augmentation de flot, alors $\Delta_k = \Delta_{k-1}$; si elle réalise une augmentation de distance, alors $\Delta_k > \Delta_{k-1}$.*

Preuve. Considérons le cas d'une augmentation de flot et notons g le nouveau flot. Si l'ensemble des arcs du nouveau graphe d'écart G_g est inclus dans l'ensemble des arcs de G_f, la fonction Δ est toujours une distance estimée pour G_g.

Supposons que le changement de flux du père u d'un arc $v = u'$ de μ crée l'arc u'' dans G_g. L'arc v étant admissible, on a $\Delta(v^-) = \Delta(v^+) + 1$ et donc $\Delta(v^+) =$

$\Delta(v^-) - 1 \leq \Delta(v^-) + 1$. Il en est de même si le changement de flux du père u d'un arc $v = u''$ de μ crée l'arc u' dans G_g. La fonction Δ respecte donc les deux conditions d'une distance estimée pour G_g.

Considérons le cas d'une augmentation de la distance estimée du sommet z due à la procédure AUGMENTER-DISTANCE et notons Δ' la nouvelle fonction Δ. Pour tout sommet distinct de z on a $\Delta'(z) = \Delta(z)$ et pour le sommet z nous avons :

$$\Delta'(z) := \begin{cases} n & \text{si } S_f(z) \text{ est vide} \\ \min_{y \in S_f(z)}\{1 + \Delta(y)\} & \text{sinon} \end{cases}$$

Il en résulte que $\Delta'(z)$ est strictement supérieur à $\Delta(z)$. De plus, pour tout sommet x successeur de z on a $\Delta'(z) \leq \Delta'(x) + 1$ et pour tout prédécesseur y de z on a $\Delta'(y) < \Delta'(z) + 1$. La fonction Δ' est donc une nouvelle distance estimée pour G_f strictement supérieure à Δ. ∎

Pour prouver que l'algorithme se termine nous utilisons le lemme 3.5 qui permet de majorer le nombre d'exécutions de la procédure AUGMENTER-FLOT entre deux exécutions de la procédure AUGMENTER-DISTANCE.

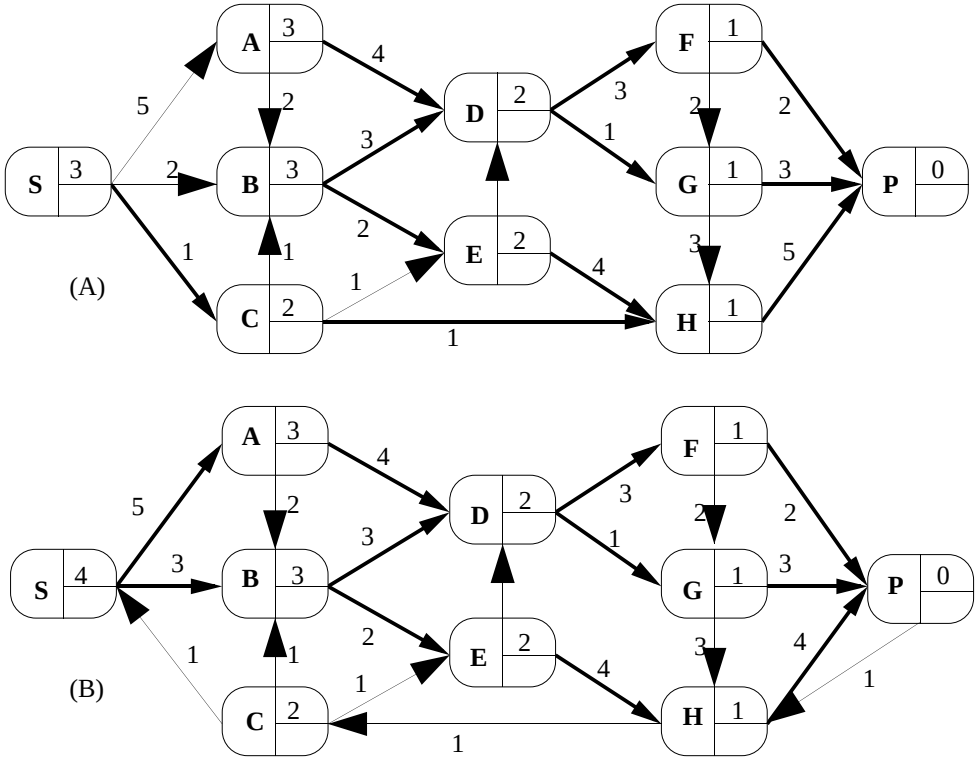


Figure 3.6: *Initialisation et première augmentation de distance.*

Lemme 3.5. *Entre deux suppressions successives d'un arc v du graphe d'écart, la distance estimée de v^+ a augmenté d'au moins deux unités.*

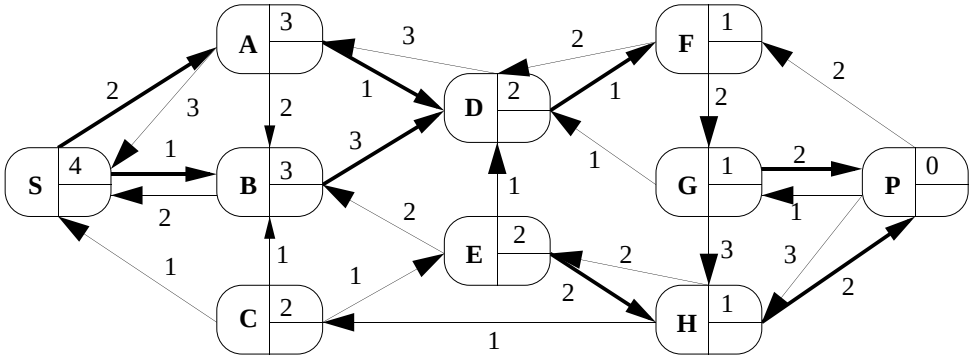


Figure 3.7: *Seconde augmentation de distance.*

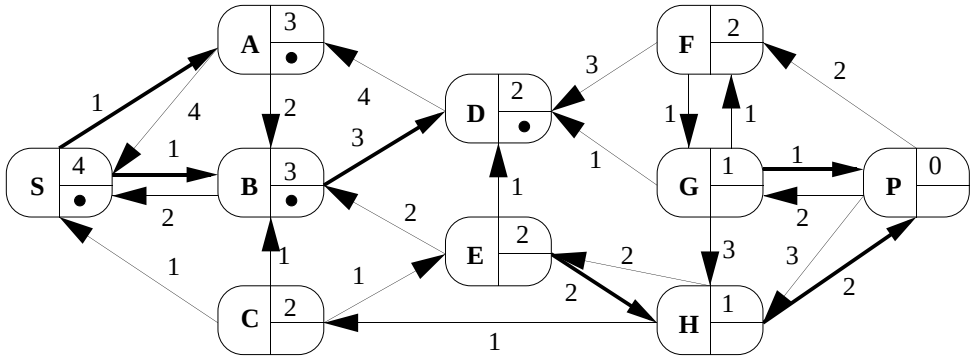


Figure 3.8: *Premier graphe d'écart d'un flot maximum.*

Preuve. Notons Δ_1 la distance estimée lors de la première suppression de l'arc v, on a $\Delta_1(v^-) = \Delta_1(v^+) + 1$ puisque v est admissible. Notons Δ_2 la distance estimée lors de la création suivante de v dans un graphe d'écart. Comme cette création ne peut avoir lieu que si l'arc conjoint de v appartient à un chemin admissible améliorant, on aura alors $\Delta_2(v^+) = \Delta_2(v^-) + 1$. Notons enfin Δ_3 la distance estimée lors de la suppression suivante de v, nous avons :

$$\Delta_3(v^+) \geq \Delta_2(v^+) = \Delta_2(v^-) + 1 \geq \Delta_1(v^-) + 1 = \Delta_1(v^+) + 2$$

■

Nous sommes maintenant en mesure de prouver la validité de l'algorithme des distances estimées au puits et de calculer sa complexité.

Théorème 3.6. *La procédure* DEP *calcule un flot maximum en temps* $O(n^2m)$.

Preuve. D'après le lemme 3.5, un même arc peut *être supprimé* d'un graphe d'écart au plus $\lceil n/2 \rceil$ fois. En effet, si l'arc v était supprimé au moins $\lceil n/2 \rceil + 1$ fois, la distance estimée de v^+ au début de la dernière augmentation de distance serait supérieure ou égale à $2\lceil n/2 \rceil \geq n$. Or comme v^+ est accessible à partir de s, on aurait $\Delta(s) \geq n$ en début d'itération. D'où la contradiction. Comme chaque

exécution de la procédure AUGMENTER-FLOT supprime au moins un arc, elle est appelée au plus $2m\lceil n/2\rceil$ fois.

Il en résulte que l'algorithme se termine puisque toute itération réalise soit une augmentation de flot soit une augmentation de distance. Lors de la terminaison, on a par définition $\Delta(s) \geq n$. Le dernier graphe d'écart ne contenant pas de chemin améliorant, le dernier flot est de valeur maximum.

Pour évaluer la complexité de la procédure DEP, nous allons suivre l'évolution de la longueur $L(\mu)$ du chemin courant μ. Cette longueur croît lors de chaque exécution de PROLONGER, décroit d'une unité au plus lors de l'exécution de AUGMENTER-DISTANCE, et devient nulle lors de l'exécution de AUGMENTER-FLOT. La complexité de la procédure PROLONGER est proportionnelle au nombre d'arcs ajoutés, c'est à dire à l'accroissement de $L(\mu)$. Comme le nombre d'exécutions de la procédure AUGMENTER-FLOT est au plus $2m\lceil n/2\rceil$ et la longueur $L(\mu)$ est au plus n, la somme des variations négatives de $L(\mu)$ dues aux exécutions de AUGMENTER-FLOT est majorée par $2nm\lceil n/2\rceil$. Comme AUGMENTER-DISTANCE est exécutée au plus n fois par sommet, la somme des variations négatives de $L(\mu)$ dues aux exécutions de AUGMENTER-DISTANCE est majorée par n^2. Il en résulte que la somme cumulée des variations négatives de $L(\mu)$ est majorée par $n^2 + 2nm\lceil n/2\rceil$. Comme la longueur initiale de μ est nulle et sa longueur terminale inférieure ou égale à n, la somme cumulée des variations positives de $L(\mu)$ est majorée par $n + n^2 + 2nm\lceil n/2\rceil$. La complexité de toutes les exécutions de la procédure PROLONGER est donc $O(n^2m)$. La complexité d'une exécution de AUGMENTER-FLOT est $O(n)$, donc la complexité de toutes les exécutions de la procédure AUGMENTER-FLOT est aussi $O(n^2m)$. La complexité d'une exécution de AUGMENTER-DISTANCE est de l'ordre du nombre de successeurs de z dans G et la distance estimée d'un sommet est augmentée au plus n fois, donc la complexité de toutes les exécutions de la procédure AUGMENTER-DISTANCE est $O(nm)$. Enfin la complexité $O(1)$ de la procédure SUPPRIMER-DERNIER-ARC est dominée par celle de AUGMENTER-DISTANCE. Il en résulte que la complexité de la procédure DEP est $O(n^2m)$. ∎

8.3.3 L'algorithme du préflot

L'algorithme des distances estimées au puits réalise des augmentations de flot le long de chemins améliorants et maintient de ce fait la satisfaction des conditions d'équilibre. A chaque étape on dispose ainsi d'un flot réalisable. Une idée *duale* consiste à saturer initialement les arcs sortant de s et à transporter *vers le puits* la plus grande partie des excès positifs ainsi créés en réalisant à chaque itération une réduction de l'excès d'un sommet. Dès que l'excès du sous-ensemble de sommets $S' = S - \{s, p\}$ est nul, le préflot obtenu est un flot de s à p de valeur maximum. Cet algorithme est une méthode duale car tant que l'algorithme n'est pas terminé, la fonction f n'est pas un flot de s à p.

Etant donné un problème de flot maximum (G, b, s, p), un *préflot* est une fonction de $\mathcal{F}(A)$ telle que :

$$\begin{aligned} e_f(s) &\leq 0 \\ e_f(p) &\geq 0 \\ 0 \leq f &\leq b \\ \forall x \in S - \{s, p\}, \quad e_f(x) &\geq 0 \end{aligned}$$

Pour un préflot f, le graphe d'écart G_f, la fonction distance estimée au puits et le graphe d'admissibilité sont définis comme dans le cas d'un flot de s à p (voir section 8.3. Un sommet *actif* est un sommet *distinct de s et de p* dont l'excès est strictement positif.

Initialisation

Le préflot initial f_0 est obtenu en saturant tous les arcs sortant du sommet s et en allouant un flux nul à tous les autres arcs. La distance estimée initiale d'un sommet x de S' est la longueur minimum en nombre d'arcs d'un chemin de x à p dans le graphe d'écart G_{f_0}, elle est notée $l(x, p, G_{f_0})$. La distance estimée initiale de s est n et celle de p est nulle. La procédure INITIALISER-PRÉFLOT réalise ces initialisations.

```
procédure INITIALISER-PRÉFLOT;
   f := 0;
   pour tout arc u sortant de s faire f(u) := b(u) finpour;
   Δ(p) := 0;
   Δ(s) := n;
   pour tout sommet x de S' faire Δ(x) := l(x, p, G_{f_0}) finpour.
```

L'initialisation joue un rôle important. En effet, comme les arcs sortant de s sont initialement saturés, il n'existe pas de chemin améliorant dans le graphe d'écart initial. De plus la distance estimée du sommet s n'est jamais modifiée. Il n'existera donc jamais de chemin améliorant dans les graphes d'écart des préflots successifs.

Itération courante

Une itération de l'algorithme du préflot consiste à choisir un sommet actif x. S'il n'existe pas d'arc admissible d'origine x, la distance estimée de x au puits est augmentée strictement comme dans la procédure AUGMENTER-DISTANCE de l'algorithme des distances estimées au puits. Sinon on choisit un arc *admissible* v d'origine x et l'on réduit l'excès du sommet origine x en modifiant le flux du père u de v. La procédure RÉDUIRE-EXCÈS ci-dessous réalise cette réduction.

```
procédure RÉDUIRE-EXCÈS(f, v);
  si v = u'
      alors ε := min{e_f(v⁻), b(u) − f(u)}; f(u) := f(u) + ε
      sinon ε := min{e_f(v⁻), f(u)}; f(u) := f(u) − ε
    finsi.
```

Nous dirons qu'une réduction d'excès est *saturante* si ϵ est égal à la valuation de l'arc v choisi dans G_f. Si l'arc v est le représentant conforme de l'arc u de G (c'est-à-dire si $v = u'$), l'arc u est saturé après la réduction et v n'appartient donc pas au nouveau graphe d'écart. Si l'arc v est le représentant non conforme de l'arc u de G (c'est-à-dire si $v = u''$), l'arc u est vide après la réduction et v n'appartient donc pas au nouveau graphe d'écart. Une réduction saturante supprime donc un arc dans l'ancien graphe d'écart.

Terminaison

Si l'ensemble des sommets actifs est vide à l'issue d'une itération, le dernier préflot est un flot de s à p et il n'existe pas de chemin améliorant dans le graphe d'écart correspondant. Le critère de terminaison de l'algorithme du préflot est l'absence de sommets actifs. Il en résulte la procédure PRÉFLOT ci-dessous :

```
procédure PRÉFLOT(G, b, s, p);
  INITIALISER-PRÉFLOT;
  tantque l'ensemble des sommets actifs est non vide faire
    choisir un sommet actif x;
    s'il existe un arc admissible d'origine x
      alors
        choisir un arc admissible v d'origine x;
        RÉDUIRE-EXCÈS(f, v)
      sinon AUGMENTER-DISTANCE(x)
    finsi
  fintanque.
```

Les figures 3.9, 3.10 et 3.11 illustrent une exécution de l'algorithme sur l'exemple de la figure 3.1.

La distance estimée est dans le coin nord-est du sommet et l'excès dans le coin sud-est. La figure 3.9 montre le graphe d'écart du préflot initial. Les premières réductions d'excès codées par (sommet actif, nature, arc admissible) sont les suiv-

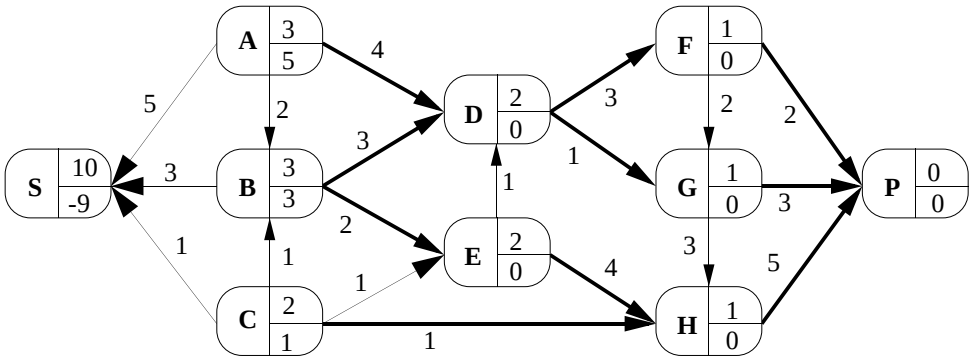


Figure 3.9: *Graphe d'écart du préflot initial.*

antes, ligne par ligne :

$(A, sat, (A, D))$	$(B, sat, (B, E))$	$(C, sat, (C, H))$
$(B, nonsat, (B, D))$	$(D, sat, (D, F))$	$(D, sat, (D, G))$
$(F, sat, (F, P))$	$(G, nonsat, (G, P))$	$(H, nonsat, (H, P))$
$(E, nonsat, (E, H))$	$(H, nonsat, (H, P))$	

Aucun des sommets actifs A, B ou F n'est alors l'origine d'un arc admissible. L'algorithme réalise une augmentation de distance pour F, les réductions $(F, nonsat, (F, G))$ et $(G, nonsat, (G, P))$, une augmentation de distance pour D et la réduction $(D, nonsat, (D, B))$. Il en résulte le graphe d'écart de la figure 3.10. La valeur maximum est atteinte mais certains excès sont encore positifs.

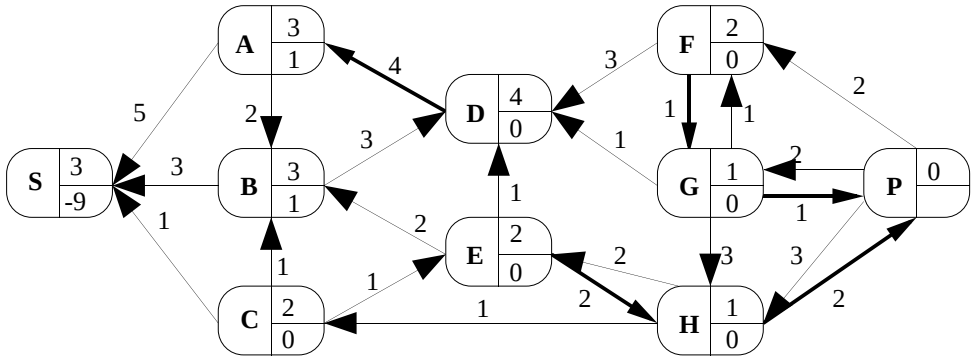


Figure 3.10: *La valeur maximum est atteinte.*

Tous ces excès doivent maintenant refluer vers l'entrée s. L'algorithme réalise alors des augmentations de distance jusqu'à ce que l'un des deux arcs (A,S) ou (B,S) devienne admissible. La figure 3.11 montre le dernier graphe d'écart lorsque les excès de tous les sommets de S' sont nuls.

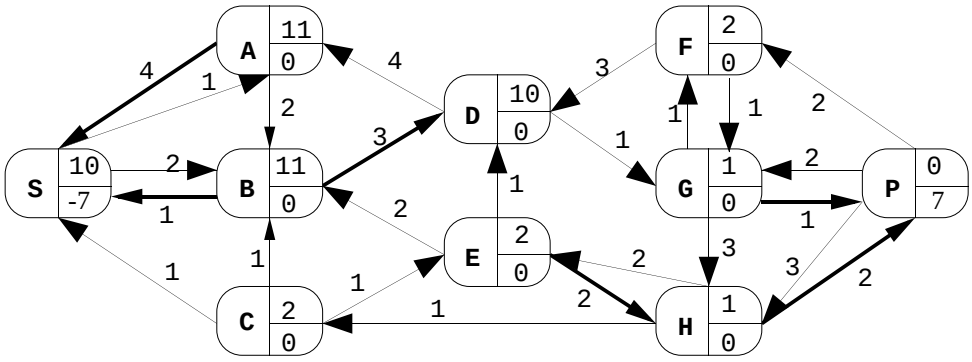


Figure 3.11: *Le dernier graphe d'écart.*

Convergence et complexité

Nous montrons dans cette section que l'algorithme du préflot se termine et nous calculons sa complexité. Deux lemmes préliminaires nous seront utiles.

Lemme 3.7. *La distance estimée d'un sommet actif est inférieure à* $2n$.

Preuve. Supposons le sommet y actif pour le préflot f et montrons qu'il existe alors un chemin élémentaire de s à y dans G dont tous les arcs ont un flux strictement positif. Considérons le graphe $H = (S \cup \{q\}, A \cup B)$ obtenu à partir de G en *ajoutant* un sommet q et les arcs suivants :

pour tout sommet actif x, ajouter un arc w_{xq} d'origine x et d'extrémité q;
ajouter un arc w_1 d'origine p et d'extrémité q;
ajouter un arc w_0 d'origine q et d'extrémité s.

Définissons la fonction g de $\mathcal{F}(A \cup B)$ par :

$$g(u) = \begin{cases} f(u) & \text{si } u \in A \\ e_f(x) & \text{si } u = w_{xq} \\ -e_f(s) & \text{si } u = w_0 \\ e_f(p) & \text{si } u = w_1 \end{cases}$$

La fonction g est un flot strictement positif du graphe H car :

$$e_g(x) = \begin{cases} e_f(x) - g(w_{xq}) = 0 & \text{si } x \text{ est actif dans } S' \\ e_f(x) = 0 & \text{si } x \text{ n'est pas actif dans } S' \\ e_f(p) - g(w_1) = 0 & \text{si } x = p \\ e_f(s) + g(w_0) = 0 & \text{si } x = s \end{cases}$$

Le flot g est donc une combinaison linéaire à coefficients entiers strictement positifs de circuits de H. L'un au moins de ces circuits passe par l'arc w_{yq} et ce circuit emprunte l'arc w_0. Soit μ le chemin élémentaire de s à y issu de ce circuit. Les arcs de μ sont des arcs de G dont les flux pour f sont strictement positifs. Il existe donc dans le graphe d'écart G_f un chemin de y à s formé par les représentants

non conformes des arcs de μ. Comme la distance estimée de s reste constante et égale à n, on a $\Delta(y) \leq \Delta(s) + (n-1) < 2n$. ∎

Le lemme 3.7 a deux conséquences importantes. Comme le nombre d'augmentations de la distance estimée *d'un même sommet* est inférieur à $2n$, le nombre total d'exécutions de la procédure AUGMENTER-DISTANCE est inférieur à $2n^2$. D'autre part le nombre d'exécutions de la procédure RÉDUIRE-EXCÈS entre deux augmentations de la distance est lui-aussi borné. En effet, cette procédure ne modifie pas la fonction distance estimée mais fait décroître d'au moins une unité le nombre $\sum_{x \in S'} e_f(x) \times \Delta(x)$. Il en résulte que l'algorithme du préflot se termine.

Nous allons maintenant évaluer le nombre d'exécutions de la procédure RÉDUIRE-EXCÈS en considérant séparément les réductions saturantes et non saturantes.

Lemme 3.8. *La procédure préflot réalise au plus $2nm$ réductions saturantes.*

Preuve. Comme dans le cas de l'algorithme des distances estimées au puits (Lemme 3.5), entre deux suppressions successives d'un même arc v du graphe d'écart, la distance estimée du sommet v^+ a augmenté d'au moins deux unités. Un même arc du graphe d'écart est donc supprimé au plus n fois (lemme 3.7). Comme une réduction saturante supprime au moins un arc, l'algorithme réalise au plus $2nm$ réductions saturantes. ∎

Lemme 3.9. *La procédure préflot réalise $O(n^2m)$ réductions non saturantes.*

Preuve. Nous notons T l'ensemble des sommets actifs et t un élément générique de T. Nous considérons comme fonction potentiel la somme $\phi = \sum_{t \in T} \Delta(t)$ des distances estimées *des sommets actifs*. Remarquons d'abord qu'à l'initialisation de l'algorithme, on a $\phi < n^2$ et qu'à sa terminaison ϕ est nul. Lors d'une itération, le sommet actif x est choisi et trois cas peuvent se présenter.

a) *Il n'existe pas d'arc admissible d'origine x.* L'ensemble T ne change pas mais la distance estimée de x augmente strictement. L'accroissement de ϕ est alors égal à celui de $\Delta(x)$. Il résulte alors du lemme 3.7 que la somme δ_1 des accroissements de ϕ dus aux augmentations de distance est majorée par $2n^2$.

b) *Une réduction saturante est réalisée sur v.* Notons y l'extrémité de l'arc v. La fonction distance estimée ne change pas mais le sommet y peut devenir actif. L'accroissement de ϕ est alors majoré par $2n$ (lemme 3.7). Comme l'algorithme exécute au plus $2nm$ réductions saturantes, l'accroissement δ_2 de ϕ dû aux réductions saturantes est majoré par $2n^2m$.

c) *Une réduction non saturante est réalisée sur v.* Notons y l'extrémité de l'arc v. La fonction distance estimée ne change pas, le sommet y peut devenir actif mais le sommet x n'est plus actif. L'accroissement de ϕ est majoré par $\Delta(y) - \Delta(x) = -1$. Donc si l'algorithme réalise N réductions non saturantes, l'accroissement correspondant δ_3 de ϕ est plus petit que $-N$.

En résumé, si nous notons δ l'accroissement total de ϕ, nous avons $\delta = \delta_1 + \delta_2 + \delta_3$ et $\delta \geq -n^2$. En utilisant les majorations précédentes il vient :

$$N \leq \delta_3 = -\delta + (\delta_1 + \delta_2) \leq 3n^2 + 2n^2 m.$$

■

Les lemmes 3.7, 3.8 et 3.9 conduisent à l'évaluation de la complexité de l'algorithme du préflot.

Théorème 3.10. *La procédure préflot calcule un flot de valeur maximum en temps $O(n^2 m)$.*

Preuve. La complexité de la procédure AUGMENTER-DISTANCE est $O(k)$ si k est le nombre de successeurs de x dans G. Comme pour un même sommet la distance estimée est augmentée au plus $2n$ fois, le temps opératoire des augmentations de distance est aussi en $O(nm)$. Si l'on associe à chaque sommet actif t un indicateur $i(t)$ qui est un arc admissible sortant de t s'il en existe ou un symbole spécial sinon, il suffit d'utiliser une structure de données (par exemple une liste doublement chaînée avec pointeurs inverses), on réalise en un temps $O(1)$ les opérations de mise à jour de l'ensemble des couples $(t, i(t))$ lors d'une réduction d'excès. Le temps opératoire global des réductions d'excès est alors en $O(n^2 m)$. Il en résulte une complexité $O(n^2 m)$ pour la procédure préflot. ■

L'algorithme du préflot peut être considéré comme un algorithme générique car il laisse libre le choix à chaque itération du nouveau sommet actif et de l'arc admissible. Nous allons présenter deux variantes efficaces de cet algorithme. L'algorithme de Karzanov choisit le sommet actif le plus éloigné du puits, l'algorithme des «excès échelonnés» réduit en priorité les sommets actifs dont les excès sont les plus grands.

8.3.4 L'algorithme de Karzanov

Cet algorithme sélectionne à chaque itération un sommet actif dont la distance estimée au puits est maximum. Il en résulte qu'entre deux exécutions consécutives de la procédure AUGMENTER-DISTANCE il y a au plus n réductions non saturantes. En effet, une réduction non saturante annule l'excès d'un sommet actif x et l'excès de x restera nul puisque, tant que la fonction distance estimée n'est pas mise à jour, les sommets actifs choisis après x ont un distance estimée inférieure ou égale à celle de x. Le nombre total de réductions non saturantes est donc en $O(n^3)$. Si l'on maintient pour chaque valeur possible r de la distance estimée une liste des sommets actifs dont la distance estimée au puits est r et un pointeur vers la liste non vide de plus grand r, la sélection d'un sommet actif de plus grande distance estimée est en $O(1)$. La complexité globale de l'algorithme de Karzanov est alors $O(n^3)$.

8.3.5 L'algorithme des excès échelonnés

L'idée de réaliser un *échelonnement* des excès et de réduire en priorité les plus grands excès conduit à une variante efficace de l'algorithme du préflot.

Cette variante découpe l'exécution de l'algorithme du préflot en phases. Si au début de la phase k on dispose d'un majorant E_{k-1} de l'ensemble des excès, cette phase sélectionnera tant qu'il en existe un sommet actif dont l'excès est supérieur ou égal à $E_{k-1}/2$. Lorsqu'il n'existe plus de tels sommets, la phase k se termine et l'on pose $E_k = E_{k-1}/2$ pour la phase $k+1$. Une phase consiste donc à réduire en priorité les plus gros excès. Il est cependant nécessaire d'affiner ce principe général qui pourrait trop souvent faire converger sur un même sommet une quantité de flux trop grande pour pouvoir être équilibrée. Une telle situation pour un sommet x force en effet l'algorithme à augmenter suffisamment la distance estimée du sommet x pour qu'il apparaisse dans le graphe d'écart un arc v *admissible* permettant de réduire l'excès du sommet x. La nouvelle règle de sélection d'un sommet actif est la suivante :

> Choisir un sommet actif x tel que
> $e_f(x) \geq E_{k-1}/2$ et $\Delta(x)$ est minimum.

La nouvelle règle de réduction de l'excès décrite par la procédure RÉDUIRE-EXCÈS-MODIFIÉ ci-dessous assure qu'au cours de la phase k les excès de tous les sommets actifs seront majorés par E_{k-1}.

```
procédure RÉDUIRE-EXCÈS-MODIFIÉ(f, v);
  si v = u'
    alors
      ε := min{e_f(v⁻), b(u) − f(u), E_{k−1} − e_f(v⁺)};
      f(u) := f(u) + ε
    sinon
      ε := min{e_f(v⁻), f(u), E_{k−1} − e_f(v⁺)};
      f(u) := f(u) − ε
  finsi.
```

L'algorithme est décrit par la procédure EXCÈS-ÉCHELONNÉS ci-dessous où l'on note B la capacité maximale d'un arc de G et $T(f, E)$ l'ensemble des sommets actifs pour le préflot f et dont l'excès est supérieur ou égal à E.

```
procédure EXCÈS-ÉCHELONNÉS( G, b, s, p);
  E := 2^⌈log B⌉ ; k := 1 + ⌈log B⌉ ;
  INITIALISER-PRÉFLOT;
  pour i de 1 à k faire
    tantque T(f, E) ≠ ∅ faire
      choisir un sommet x dans T(f, E) de distance estimée minimum;
      si x est une sortie du graphe d'admissibilité
        alors AUGMENTER-DISTANCE( x)
        sinon
          choisir un arc admissible v d'origine x;
          RÉDUIRE-EXCÈS-MODIFIÉ( f, v)
      finsi
    fintantque
    E := E/2
  finpour.
```

Les nouvelles règles de réduction et de sélection utilisées par l'algorithme induisent la propriété suivante :

Lemme 3.11. *Pendant la phase k, une réduction non saturante porte sur au moins $E_{k-1}/2$ unités de flux et l'excès de tout sommet reste inférieur à E_{k-1}.*

Preuve. Soit x un sommet actif choisi lors d'une réduction non saturante de la phase k et soit v l'arc admissible d'origine x et d'extrémité y sélectionné pour cette réduction. On a $\Delta(y)=\Delta(x)-1$ puisque v est admissible et par conséquent $e_f(y) < E_{k-1}/2$ d'après la nouvelle règle de sélection. La réduction n'étant pas saturante, la valuation de v est supérieure ou égale à $\min\{e_f(x), E_{k-1} - e_f(y)\}$. Comme $e_f(x) \geq E_{k-1}/2$ et $E_{k-1} - e_f(y) > E_{k-1}/2$, la variation de flux du père de v est supérieure ou égale à $E_{k-1}/2$.

Considérons maintenant une réduction quelconque de l'excès du sommet actif x à partir de l'arc admissible v d'origine x et d'extrémité y. Après cette réduction, seul le sommet y a pu voir son excès augmenter et cette augmentation est inférieure ou égale à $E_{k-1} - e_f(y)$. L'excès du sommet y inférieur à E_{k-1} au début de la phase k restera donc inférieur à cette valeur pendant toute cette phase. ∎

Théorème 3.12. *La complexité en temps de l'algorithme* EXCÈS-ÉCHELONNÉS *est $O(nm + n^2 \log B)$.*

Preuve. Evaluons le nombre de réductions non saturantes au cours de la phase k en considérant la fonction potentiel :

$$F = \frac{1}{E_{k-1}} \sum_{x \in S'} e_f(x)\Delta(x).$$

Au début de cette phase, F est majorée par $2n^2$ puisque pour tout sommet x, $e_f(x) < E_{k-1}$ et $\Delta(x) \leq 2n$ (Lemme 3.7). Lors de la sélection d'un sommet actif x, examinons les variations de F dues aux augmentations de distance et aux réductions saturantes

a) *Augmentation de distance.* L'algorithme augmente la distance estimée $\Delta(x)$ de d unités ($d \geq 1$). Il en résulte une augmentation de F majorée par d. La somme des augmentations de la distance estimée d'un sommet étant majorée par $2n$, l'augmentation totale de F due aux occurrences de ce premier cas est majorée par $2n^2$.

b) *Réduction saturante.* La fonction F décroit au moins de la valeur $1/2$ puisque, d'après le lemme 3.11, l'excès de x diminue d'au moins $E_{k-1}/2$ unités et l'excès de y augmente de la même quantité.

Comme F reste positif par définition et qu'une réduction quelconque fait décroitre F, le nombre de réductions non saturantes au cours de la phase k est majoré par $8n^2$.

Dans l'étude de l'algorithme du préflot, nous avons montré que la complexité des opérations autres que les réductions non saturantes est $O(nm)$. On peut d'autre part gérer l'ensemble des sommets actifs dont l'excès est supérieur ou égal à $E_{k-1}/2$ en associant à chaque valeur possible r de la distance estimée une liste doublement chaînée des sommets dont la distance estimée est r et l'excès supérieur à $E_{k-1}/2$. Un index sur la liste non vide de plus petit r est également maintenu. Les opérations d'ajout, de retrait et de sélection d'un sommet actif sont alors en $O(1)$ et la reconstitution des listes à chaque nouvelle phase en $O(n)$. Le nombre de phases étant égal à $1 + \lceil \log B \rceil$, la complexité globale de l'algorithme EXCÈS-ÉCHELONNÉS est $O(nm + n^2 \log B)$. ∎

8.4 Flot de coût minimum

Nous considérons dans cette section le problème de la recherche d'un flot compatible de coût minimum dans un réseau valué $R = (G, a, b, c)$. Nous présentons d'abord les propriétés fondamentales de dualité sur lesquelles s'appuient tous les algorithmes de résolution du problème. Nous décrivons ensuite l'algorithme primal de Golberg et Tarjan pour la recherche d'un flot de coût mimimum dans un réseau $R = (G, 0, b, c)$. Nous montrons ensuite que l'on peut ramener le problème général à la recherche d'un flot maximum de coût minimum dans un réseau muni d'une entrée, d'une sortie et dont les arcs sont valués par une capacité maximale et un coût. Nous décrivons un algorithme de type dual pour la résolution de ce problème lorsque les coûts sont positifs ou nuls. Nous considérons enfin le problème de la recherche d'un plan de transport de coût minimum et sa résolution par l'algorithme dual d'Edmonds et Karp.

8.4.1 Graphe d'écart et conditions d'optimalité

Soit f un flot d'un réseau valué $R = (G, 0, b, c)$. Le graphe d'écart du flot f est obtenu à partir du graphe d'écart G_f défini par la procédure GRAPHE-D'ÉCART de la section 8.3 en munissant simplement chaque arc v d'un coût, noté également $c(v)$, et défini par :

$$c(v) = \begin{cases} c(u) & \text{si } v = u' \\ -c(u) & \text{si } v = u'' \end{cases}$$

Soit $\mu = ((s_0, u_1, s_1), (s_1, u_2, s_2), \cdots, (s_{q-1}, u_q, s_q))$ un cycle améliorant pour le flot f. Par définition, un arc avant de μ n'est pas saturé et un arc arrière de μ n'est pas vide. Il en résulte que la suite $\nu = ((s_0, v_1, s_1), (s_1, v_2, s_2), \cdots, (s_{q-1}, v_q, s_q))$ où

$$v_k = \begin{cases} u'_k & \text{si } u_k \text{ est arc avant de } \mu \\ u''_k & \text{si } u_k \text{ est arc arrière de } \mu \end{cases}$$

est un circuit de coût négatif du graphe d'écart G_f. Réciproquement à un circuit de coût négatif du graphe d'écart G_f correspond un cycle améliorant de G.

La proposition 4.1, conséquence directe du théorème 2.7, énonce une condition nécessaire et suffisante d'optimalité qui porte uniquement sur le graphe d'écart G_f.

Proposition 4.1. *Un flot f d'un réseau $R = (G, 0, b, c)$ est de coût minimum si et seulement si son graphe d'écart G_f ne possède aucun circuit de coût strictement négatif.*

8.4.2 Problème dual et conditions d'optimalité

Nous avons établi que la recherche d'un flot de coût minimum correspond à la résolution d'un programme linéaire dont la matrice est la matrice d'incidence sommets-arcs du graphe G. Les algorithmes de résolution efficaces de ce problème vont d'une part exploiter la structure de cette matrice (c'est-à-dire travailler sur le graphe d'écart) et d'autre part tirer parti des propriétés de dualité issues de la programmation linéaire.

Le *programme linéaire primal* (PLP) associé à un problème de flot de coût minimum dans un réseau $R = (G, 0, b, c)$ s'écrit :

$$\begin{array}{rl} \forall u \in A, & f(u) \geq 0 \\ \forall u \in A, & f(u) \leq b(u) \\ \forall x \in S, & e_f(x) = 0 \\ MIN & \sum_{u \in A} c(u) f(u) \end{array} \qquad (PLP)$$

où les variables $f(u)$ sont rationnelles.

Le *programme linéaire dual* (PLD) associe à chaque sommet x du réseau la variable sans contrainte de signe $\pi(x)$ appelée *potentiel* du sommet x, et associe

à chaque arc u du réseau une variable positive ou nulle notée $\delta(u)$. Le programme dual (PLD) s'écrit :

$$\begin{array}{rl} \forall u \in A, \quad -\delta(u) + \pi(u^+) - \pi(u^-) & \leq c(u) \\ \forall u \in A, \quad \delta(u) & \geq 0 \\ MAX \quad -\sum_{u \in A} b(u)\delta(u) & \end{array} \qquad (PLD)$$

L'existence d'un sous-ensemble dominant des solutions du programme dual montre que les véritables inconnues du programme dual sont les potentiels des sommets de G. En effet si nous *fixons* le potentiel de chaque sommet, il est immédiat de vérifier que la meilleure solution du programme dual correspond aux valeurs des variables $\delta(u)$ calculées par la formule (Δ) ci-dessous :

$$\forall u \in A, \quad \delta(u) = \max\{0, -c(u) + \pi(u^+) - \pi(u^-)\}. \qquad (\Delta)$$

Nous ne considèrerons donc dans la suite que des *solutions dominantes du programme dual*, c'est-à-dire des couples (π, δ) pour lesquels les valeurs des $\delta(u)$ résultent de π par la formule (Δ). Une solution du dual sera donc complètement caractérisée par un ensemble de potentiels π.

Etant donnés un flot f solution du programme linéaire primal et une solution (π, δ) du programme linéaire dual, le théorème *des écarts complémentaires*, corollaire du théorème de la dualité, fournit une condition *nécessaire et suffisante* d'optimalité des deux solutions. Cette condition, notée (OPT), s'écrit :

$$\begin{array}{rl} \forall u \in A, \quad \delta(u)[f(u) - b(u)] & = 0 \\ \forall u \in A, \quad f(u)[c(u) + \delta(u) - \pi(u^+) + \pi(u^-)] & = 0 \end{array} \qquad (OPT)$$

Les conditions d'optimalité précédentes peuvent être exprimées de manière plus condensée à partir du graphe d'écart en utilisant la notion de coût réduit. Soit π une fonction potentiel définie sur les sommets de G et f un flot, le *coût réduit* relatif à π d'un arc v du graphe d'écart G_f est défini par $\bar{c}_\pi(v)=c(v)+\pi(v^-)-\pi(v^+)$. La Proposition 4.2 fournit une condition nécessaire et suffisante d'optimalité fondée sur les coûts réduits du graphe d'écart G_f.

Proposition 4.2. *Un flot f et un ensemble de potentiels π sont optimaux si et seulement si le coût réduit relatif à π de tout arc du graphe d'écart G_f est positif ou nul.*

Preuve. Soient f un flot et (π, δ) une solution dominante du dual qui satisfont la condition (OPT). Considérons un arc v du graphe d'écart G_f.

Si $v = u'$ (c'est-à-dire si v est le représentant conforme de l'arc u de G), l'arc u n'est pas saturé et l' on a d'après (OPT) : $\delta(u) = 0$. Il résulte alors de la formule (Δ) que $\pi(u^+) - \pi(u^-) - c(u) \leq 0$ et donc que $\bar{c}_\pi(v)=c(v) + \pi(v^-) - \pi(v^+) \geq 0$.

Si $v = u''$ (c'est-à-dire si v est le représentant non conforme de l'arc u de G), l'arc u n'est pas vide et l' on a d'après (OPT) : $0 \leq \delta(u) = -c(u) + \pi(u^+) - \pi(u^-)$. Comme $v^+ = u^-$ et $v^- = u^+$, nous avons $\bar{c}_\pi(v)=c(v) + \pi(v^-) - \pi(v^+) \geq 0$.

Supposons maintenant que le coût réduit de tout arc du graphe d'écart soit positif ou nul. Pour un arc u de G, trois cas sont possibles :

L'arc u est libre. Les deux arcs u' et u'' de G_f ont des coûts réduits positifs ou nuls, et donc nuls puisque $\bar{c}_\pi(u') = -\bar{c}_\pi(u'')$; il en résulte que $\delta(u) = 0$ et que la condition (OPT) est satisfaite pour l'arc u.

L'arc u est saturé. Le coût réduit de l'arc u'' étant positif ou nul, on a $-c(u) + \pi(u^+) - \pi(u^-) \geq 0$. On a donc d'après (Δ) : $\delta(u) = -c(u) + \pi(u^+) - \pi(u^-)$ et la condition (OPT) est satisfaite pour l'arc u.

L'arc u est vide. Le coût réduit de l'arc u' étant positif ou nul, on a $c(u) + \pi(u^-) - \pi(u^+) \geq 0$. On a donc d'après (Δ) : $\delta(u) = 0$. La condition (OPT) est satisfaite pour l'arc u. ∎

8.4.3 Un algorithme primal

Le problème considéré dans cette section est la recherche d'un flot compatible de coût minimum dans un réseau valué $R = (G, 0, b, c)$. Aucune restriction n'est faite ici sur le signe des coûts $c(u)$. L'algorithme que nous allons décrire, dû à Goldberg et Tarjan détruit systématiquement à chaque itération le cycle améliorant de G associé à un circuit de coût moyen minimum du graphe d'écart. Cet algorithme est de type primal puisqu'à chaque étape, il fournit un flot compatible tout en améliorant la satisfaction des conditions d'optimalité duales. La procédure GOLDBERG-TARJAN(R) ci-dessous réalise cet algorithme.

```
procédure GOLDBERG-TARJAN(R);
   f :=0;
   ε :=max_{u∈A} |c(u)|;
   tantqu'il existe un circuit négatif dans G_f faire
      ρ :=CIRCUIT-COÛT-MOYEN-MINIMUM(G_f);
      f :=AUGMENTER-FLOT-SUR-CIRCUIT(f, ρ)
   fintantque.
```

Soit f un flot et G_f le graphe d'écart associé. Le *coût moyen* d'un circuit ρ de q arcs dans G_f est la quantité $\sum_{v\in\rho} c(v)/q$. Remarquons que pour toute fonction potentiel π, le coût réduit moyen d'un circuit de G_f est égal à son coût moyen. Soit $\gamma(f)$ le coût moyen minimum d'un circuit de G_f. Remarquons qu'il existe un circuit élémentaire de coût $\gamma(f)$. L'algorithme de Karp, de complexité $O(nm)$, permet de calculer le coût moyen minimum d'un circuit dans un graphe valué de n sommets et m arcs (voir exercices). Le flot f est dit *ϵ-optimal* s'il existe une fonction potentiel π telle que :

$$\forall v \in A_f, \quad \bar{c}_\pi(v) \geq -\epsilon.$$

Il résulte de cette définition qu'un flot 0-optimal est de coût minimum. La proposition suivante montre que si ϵ est suffisamment petit, un flot ϵ-optimal est de coût minimum.

Proposition 4.3. *Si $0 \leq \epsilon < 1/n$, un flot ϵ-optimal f est de coût minimum.*

Preuve. Soit ρ un circuit élémentaire de G_f. On a $c(\rho) = \bar{c}_\pi(\rho) > -1$. La valeur de ρ est donc positive ou nulle. ∎

Etant donné un flot f, on appelle *déviation* de f la valeur minimum de ϵ pour laquelle le flot f est ϵ-optimal. La déviation de f, notée $\epsilon(f)$, est liée au coût moyen minimum d'un circuit de G_f.

Théorème 4.4. *Soit f un flot. La déviation de f est égale à* $\max\{0, -\gamma(f)\}$.

Preuve. Si le flot f est ϵ-optimal, alors pour un circuit quelconque ρ de q arcs dans G_f, on a $c(\rho) \geq -q\epsilon$. Il en résulte que $\epsilon \geq -\gamma(f)$. Nous montrons maintenant qu'il existe effectivement une fonction potentiel π pour laquelle le flot f est $-\gamma(f)$-optimal.

Supposons d'abord que le graphe G_f soit fortement connexe. Définissons pour chaque arc v de G_f la valuation $c'(v) = c(v) - \gamma(f)$ et choisissons un sommet source s. Pour tout circuit ρ de q arcs dans G_f, on a $c'(\rho) = c(\rho) - q\gamma(f) \geq 0$. Pour tout sommet x, il existe donc un chemin de coût minimum de s à x dans G_f pour la valuation c'. Si l'on définit le potentiel $\pi(x)$ du sommet x par la valeur de ce chemin, il vient :

$$\forall v \in A_f, \quad \pi(v^+) \leq \pi(v^-) + c(v) - \gamma(f).$$

Il en résulte que le flot f est $-\gamma(f)$-optimal.

Si le graphe G_f n'est pas fortement connexe, on note $\{C_1, \ldots, C_p\}$ ses composantes fortement connexes. Tous les circuits du graphe induit par C_i ont une valeur positive ou nulle pour la valuation c'. En considérant le graphe induit par C_i *seul*, nous pouvons associer à chaque sommet x de C_i une fonction $\pi'(x)$ qui vérifie l'ingalité précédente sur tous les arcs du graphe (fortement connexe) induit par C_i. La valuation a_{ij} d'un arc (C_i, C_j) du graphe réduit de G_f est définie par :

$$a_{ij} = \min\{c'(v) + \pi'(v^-) - \pi'(v^+) \mid v \in A_f, v^- \in C_i, v^+ \in C_j\}.$$

Le graphe réduit de G_f étant sans circuits, il existe pour tout sommet C_i un chemin de coût minimum α_i d'extrémité C_i. Par définition des valeurs α_i, nous avons $\alpha_j - \alpha_i \leq a_{ij}$ pour tout arc (C_i, C_j). Le potentiel d'un sommet x de C_i est alors défini par $\pi(x) = \pi'(x) + \alpha_i$. Pour tout arc v de G_f appartenant à un graphe induit par une composante fortement connexe, on a bien sûr $\bar{c}_\pi(v) \geq -\gamma(f)$. Soit maintenant un arc v tel que $v^- \in C_i$ et $v^+ \in C_j$, on a :

$$\pi(v^+) - \pi(v^-) = \pi'(v^+) - \pi'(v^-) + \alpha_j - \alpha_i \leq \pi'(v^+) - \pi'(v^-) + a_{ij} \leq c'(v).$$

Le flot f est donc $-\gamma(f)$-optimal pour le potentiel π. ∎

Il est utile dans la suite de considérer le graphe partiel de G_f constitué des arcs dont le coût réduit pour un potentiel π est strictement négatif. Nous appellerons ce graphe le *graphe d'inadmissibilité* de f pour la fonction potentiel π et nous le noterons $G_f^-(\pi)$. La proposition suivante montre que si un flot f est ϵ-optimal pour la fonction potentiel π et si $G_f^-(\pi)$ est sans circuits, la déviation de f est au plus $(1-1/n)\epsilon$.

Proposition 4.5. *Soit f un flot ϵ-optimal pour le potentiel π. Si le graphe d'inadmissibilité de f est sans circuit, alors le flot f est $(1-1/n)\epsilon$-optimal.*

Preuve. Soit ρ un circuit élémentaire de q arcs dans G_f. Le circuit ρ contient au moins un arc de coût réduit positif. Le coût moyen de ρ qui est au moins égal à $-(1-1/q)\epsilon$ est minoré par $-(1-1/n)\epsilon$. Il en résulte que pour un circuit de coût moyen minimum, on a $\gamma(f) \geq -(1-1/n)\epsilon$. On déduit alors le résultat du théorème 4.4. ∎

Convergence

La convergence de l'algorithme est assurée car le flot g résultant de la destruction d'un circuit ρ de coût moyen minimum dans G_f par la procédure AUGMENTER-FLOT-SUR-CIRCUIT(f, ρ) possède une déviation au plus égale à celle du flot f. Les deux lemmes suivants précisent les conditions de cette convergence.

Lemme 4.6. *Soit g le flot résultant d'un flot ϵ-optimal f après la destruction d'un circuit de coût moyen minimum dans G_f. Les déviations de f et g satisfont $\epsilon(g) \leq \epsilon(f)$.*

Preuve. D'après la définition de la déviation $\epsilon(f)$ du flot f, il existe un potentiel π tel que pour tout arc v de G_f on a $\bar{c}_\pi(v) \geq -\epsilon(f)$. Soit ρ le circuit détruit. Comme le coût moyen des arcs de ρ est égal à $-\epsilon(f)$, le coût réduit pour π d'un arc quelconque de ρ est égal à $-\epsilon(f)$. Soit v un arc de G_g qui n'existe pas dans G_f. Si $v = u'$, alors l'arc u'' est un arc de ρ. On a donc $\bar{c}_\pi(u'') = -\epsilon(f)$ et $\bar{c}_\pi(v) = \epsilon(f)$. On aboutit à la même conclusion si $v = u''$. Il en résulte que $\epsilon(g) \leq \epsilon(f)$. ∎

Nous noterons dans la suite m_e le nombre d'arcs du graphe d'écart G_f qui reste toujours inférieur ou égal à $2m$. Le lemme suivant évalue le gain obtenu lorsque l'algorithme exécute m_e itérations successives.

Lemme 4.7. *Soit f un flot et soit g le flot obtenu après m_e destructions de circuits. Si g n'est pas de coût minimum, sa déviation satisfait $\epsilon(g) \leq (1-1/n)\epsilon(f)$*
.

Preuve. Soit f un flot ϵ-optimal pour le potentiel π. La destruction d'un circuit ρ de coût moyen minimum dont tous les arcs ont un coût réduit négatif ne peut créer que des arcs de coût réduit positif ou nul et supprime au moins un arc de $G_f^-(\pi)$. Nous considérons alors deux cas pour le flot g obtenu après les m_e itérations successives :

Lors de chaque destruction, les arcs du circuit détruit ont un coût réduit négatif ou nul. D'après la remarque précédente, le graphe d'inadmissibilité $G_f^-(\pi)$ est vide et le flot g est donc de coût minimum.

Soit ρ le premier circuit supprimé contenant un arc de coût réduit positif. D'après le lemme 4.6, la déviation du flot h obtenu avant la suppression de ρ est inférieure ou égale à $\epsilon(f)$. D'après la preuve de la proposition 4.5, le coût moyen de ρ vaut au moins $-(1-1/n)\epsilon$. Il résulte alors du théorème 4.4 et du lemme 4.6 que la déviation des flots obtenus après h est inférieure ou égale à $(1-1/n)\epsilon$. ∎

Les lemmes 4.6 et 4.7 sont alors suffisants pour majorer polynomialement le nombre d'itérations de l'algorithme.

Théorème 4.8. *L'algorithme de Goldberg et Tarjan réalise $O(nm \log nC)$ itérations.*

Preuve. Soit f le flot courant. Lors de l'initialisation de l'algorithme, on a $\epsilon(f) \leq C$. D'après la proposition 4.3, l'algorithme se termine dès que la déviation du flot courant est strictement inférieure à $1/n$. Soit K le nombre total d'itérations et posons $K-1 = pq + r, (0 \leq r < p)$. La déviation obtenue après pq destructions est inférieure ou égale à $(1-1/n)^q C$ d'après le lemme 4.7 et supérieure ou égale à $1/n$ puisque l'algorithme n'est pas terminé. On a donc $(1-1/n)^q C \geq 1/n$ ou encore $(1-1/n)^{\lfloor (K-1)/p \rfloor} C \geq 1/n$. Comme pour $n > 1$ on a $\ln(1-1/n) < -1/n$, il vient :

$$\lfloor (K-1)/p \rfloor \leq -\ln nC / \ln(1-1/n) \leq n \ln nC.$$

Il en résulte que $K = O(mn \log nC)$. ∎

Si l'on utilise l'algorithme de Karp pour déterminer à chaque itération un circuit de coût moyen minimum, la complexité de l'algorithme de Goldberg et Tarjan est $O(n^2 m^2 \log nC)$. En fait Goldberg et Tarjan ont montré que le nombre total d'itérations est majoré par un polynôme en n et m. La preuve due à Tardos repose sur une propriété supplémentaire des flots ϵ-optimaux que nous énonçons sans démonstration.

Proposition 4.9. *Soit f un flot ϵ-optimal pour le potentiel π et v un arc de G_f de coût réduit supérieur ou égal à $2n\epsilon$. Le flux du père de v dans G est le même pour tous les flots ϵ-optimaux.*

Cette proposition permet d'établir que la complexité en temps de l'algorithme de Goldberg et Tarjan est $O(n^2 m^2 \min\{\log nC, m \log n\})$.

Une amélioration de l'algorithme

La version de l'algorithme de Goldberg et Tarjan décrite dans la section précédente est intéressante à un double point de vue, d'une part elle exploite directement la condition nécessaire et suffisante d'optimalité d'un flot compatible et d'autre part il s'agit d'un algorithme *fortement polynomial*, c'est-à-dire polynomial lorsque le coût d'une opération arithmétique élémentaire est polynomial en le nombre de bits nécessaires pour coder ses opérandes. Cependant cet algorithme est moins efficace que d'autres algorithmes fortement polynomiaux fondés sur des techniques d'échelonnement ou d'approximations.

Goldberg et Tarjan ont alors proposé une nouvelle version qui constitue l'un des meilleurs algorithmes connus aujourd'hui pour des réseaux pas trop denses dont les coûts (entiers) ne sont pas trop grands. Cette amélioration consiste à remplacer la destruction d'un circuit de coût moyen minimum (qui coûte $O(nm)$ opérations élémentaires) par une série d'au plus m destructions de circuits du graphe d'inadmissibilité suivie d'un ajustement de la fonction potentiel. Une structure de données ad-hoc, appelée *arbres dynamiques*, permet de réaliser une destruction de circuit ou un ajustement de potentiel en $O(\log n)$.

Soit f le flot courant et soit π le potentiel courant. Le plus petit ϵ, noté $\epsilon(f,\pi)$, pour lequel le flot f est ϵ-optimal *à π fixé*, est défini par :

$$\epsilon(f,\pi) = \max\{0, -\min\{\bar{c}_\pi(v) \mid v \in A_f\}\}.$$

Rappelons que le graphe d'inadmissibilité $G_f^-(\pi)$ est formé des arcs v de G_f dont le coût réduit $\bar{c}_\pi(v)$ par rapport à π est strictement négatif. L'algorithme peut alors être décrit comme suit :

```
procédure GOLDBERG-TARJAN-PLUS(R);
  f :=0; π :=0;
  tantque ε(f,π) > 1/n faire
    tantque G_f^-(π) possède un circuit faire
      DÉTRUIRE-CIRCUIT(G_f^-(π))
    fintantque
    AJUSTER-POTENTIEL(π)
  fintantque.
```

La procédure DÉTRUIRE-CIRCUIT détermine un circuit ρ de $G_f^-(\pi)$, et détruit ce circuit par un appel à la procédure AUGMENTER-FLOT-SUR-CIRCUIT. La procédure AJUSTER-POTENTIEL transforme π en un potentiel π' tel que $\epsilon(f,\pi') \leq \epsilon(f,\pi)$.

Si l'on définit une itération comme une série de destructions de circuits suivie d'un ajustement de la fonction potentiel (boucle tantque extérieure), le théorème suivant montre que cet algorithme converge après au plus $O(n \log nC)$ itérations.

Théorème 4.10. *La procédure* GOLDBERG-TARJAN-PLUS *calcule un flot de coût minimum en $O(n \log nC)$ itérations.*

Preuve. Considérons une itération. Chaque destruction de circuit dans $G_f^-(\pi)$ n'ajoute pas d'arcs à $G_f^-(\pi)$ mais en détruit au moins un. Le graphe $G_f^-(\pi)$ sera donc sans circuits après au plus m_e destructions de circuits. Si, après ces destructions, le graphe $G_f^-(\pi)$ est vide, le flot f est de coût minimum, sinon le flot f est $(1 - 1/n)\epsilon(f, \pi)$-optimal d'après le lemme 4.7. La convergence et le nombre maximum d'itérations résultent alors du lemme 4.3 et du théorème 4.4. ∎

Goldberg et Tarjan ont créé une structure de données spécifique, appelée *arbre dynamique* qui permet de réaliser chaque destruction de circuit et ajustement de potentiel en $O(\log n)$. Indiquons simplement qu'elle permet de gérer une famille d'arborescences dont chaque arc correspond à un arc v du graphe $G_f^-(\pi)$ valué par $b(u) - f(u)$ si $v = u'$ ou par $f(u)$ si $v = u''$. Cette structure conduit à un algorithme de complexité $O(nm(\log n) \min\{\log nC, m \log n\})$ qui est l'un des meilleurs connus aujourd'hui.

8.4.4 Flot maximum de coût minimum

Soit $R = (G, a, b, c)$ un réseau valué et f un flot compatible de ce réseau. Nous montrons que l'on peut ramener la recherche d'un flot compatible de coût minimum du réseau R à celle d'un flot maximum de coût minimum dans un réseau R' muni d'une entrée, d'une sortie, et dont les arcs sont valués par une capacité maximale et un coût. La construction du réseau R' repose sur le signe des excès $e_a(x)$ des sommets de G pour la fonction a. Nous noterons S_+ (respectivement S_-) l'ensemble des sommets de G dont l'excès pour a est strictement positif (respectivement négatif) et α la quantité $\sum_{x \in S_+} e_a(x) = -\sum_{x \in S_-} e_a(x)$. Nous supposerons α strictement positif.

Les sommets du réseau R' sont ceux du graphe G auxquels on adjoint un sommet source s et un sommet puits p. L'ensemble des arcs du réseau R' est formé :

(a) des arcs u du graphe G où chaque arc u est muni d'une capacité minimale nulle, d'une capacité maximale égale à $b(u) - a(u)$ et du coût $c(u)$;
(b) pour chaque sommet x de S_+ d'un nouvel arc u_{sx}, d'origine s, d'extrémité x, de capacité minimale nulle, de capacité maximale $e_a(x)$ et de coût nul ;
(c) pour chaque sommet x de S_- d'un nouvel arc u_{xp}, d'origine x, d'extrémité p, de capacité minimale nulle, de capacité maximale $-e_a(x)$ et de coût nul ;
(d) d'un arc de retour u_0 d'origine p, d'extrémité s, de capacité minimale nulle, de capacité maximale α et de coût nul.

La figure 4.1 montre en (A) un réseau valué R où sur chaque arc sont inscrits le triplet $(a(u), b(u), c(u))$ et le flux $f'(u)$ en gras et où à côté de chaque sommet l'excès pour la fonction a est souligné. Sur la partie (B) est représenté le réseau R' où sur chaque arc sont incrits le couple $(b(u), c(u))$ et en gras le flux $f(u)$.

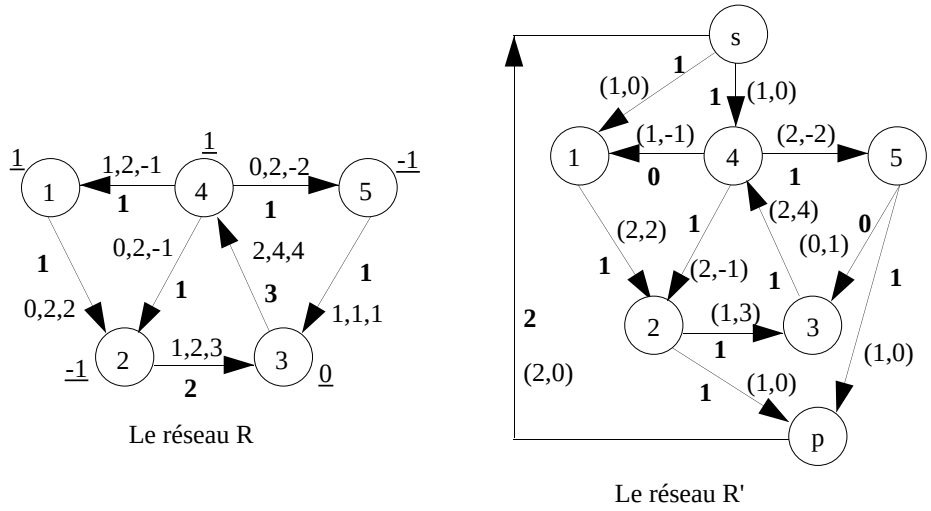


Figure 4.1: *Les réseaux R et R'.*

On appelle *flot saturant* du réseau R' un flot compatible de valeur α. La proposition 4.11 établit la correspondance entre les flots compatibles de R et les flots saturants de R'.

Proposition 4.11. *Il existe une bijection entre les flots compatibles du réseau R et les flots saturants du réseau R'. L'image d'un flot compatible de coût minimum du réseau R est un flot maximum de coût minimum du réseau R'.*

Preuve. Soit f un flot compatible de R, nous lui associons l'application f' définie sur les arcs du réseau R' par :

$$f'(u) = \begin{cases} e_a(x) & \text{si } u = u_{sx} \\ -e_a(x) & \text{si } u = u_{xp} \\ f(u) - a(u) & \text{si } u \in A \\ \alpha & \text{si } u = u_0 \end{cases}$$

La fonction f' satisfait les contraintes de capacité du réseau R', montrons que cette fonction est aussi un flot de R' en calculant les excès des différents sommets. Il est d'abord clair que $e_{f'}(s)$=$e_{f'}(p)$=0. Considérons un sommet x de S_+, nous avons :

$$e_{f'}(x) = (e_f(x) - e_a(x)) + f'(u_{sx}) = 0;$$

de même pour un sommet x de S_-, nous avons :

$$e_{f'}(x) = (e_f(x) - e_a(x)) - f'(u_{xp}) = 0.$$

La fonction f' est donc un flot saturant du réseau R'.

Réciproquement, soit g un flot saturant du réseau R'. La fonction f définie sur les arcs de G par $f(u) = g(u) + a(u)$ est le seul flot compatible de R vérifiant $f' = g$.

La correspondance $f \mapsto f'$ est donc bien une bijection. On remarque de plus que les coûts des flots f et f' diffèrent de la constante $a \star c$, ce qui implique la seconde partie de la proposition. ∎

Il résulte de la propriété précédente que la recherche d'un flot de valeur maximum de s à p dans le réseau R' permet de répondre au problème de l'existence d'un flot compatible pour le réseau R. En effet, si la valeur maximum d'un flot de s à p dans R' est inférieure strictement à α, le réseau R ne possède pas de flot compatible. Dans le cas contraire, un tel flot existe et il en est de même d'un flot compatible de coût minimum (voir section 2.3).

Un algorithme dual

Une donnée (G, b, s, p, c) d'un problème de flot maximum de coût minimum est spécifiée par un énoncé (G, b, s, p) d'un problème de flot maximum et une fonction c de $\mathcal{F}(A)$ définissant le coût sur chaque arc d'une unité de flux. Nous supposerons dans cette section la fonction c positive ou nulle. Si α est la valeur maximum d'un flot de s à p pour le problème (G, b, s, p), un flot maximum de coût minimum est un flot de valeur α dont le coût est minimum.

Un couple (f, π) où f est un flot du réseau R est dit *dual-réalisable* si les coûts réduits relatifs à π des arcs de G_f sont positifs ou nuls. Il est dit *primal-réalisable* si f est un flot maximum de s à p. Le principe de base de cet algorithme dual est de réaliser à chaque itération une augmentation de la valeur du flot à partir d'un chemin améliorant de coût réduit minimum dans le graphe d'écart. Comme nous le démontrerons, ce choix permet de déterminer un nouvel ensemble de potentiels pour lequel les arcs du nouveau graphe d'écart ont un coût réduit positif ou nul.

Initialisation

Le flot initial est le flot nul et le potentiel initial de chaque sommet est nul également. Comme la fonction c est positive ou nulle, les coûts réduits initiaux du premier graphe d'écart sont positifs ou nuls. Le couple $(0, 0)$ est donc dual-réalisable.

Itération courante

Soit (f, π) le couple courant dual-réalisable (et non primal-réalisable) résultat de l'itération précédente. Comme le flot f n'est pas de valeur maximum, il existe au moins un chemin améliorant de s à p dans le graphe d'écart G_f. Comme les coûts réduits des arcs du graphe d'écart G_f sont positifs ou nuls, il existe également un chemin améliorant μ de coût réduit minimum. L'itération courante réalise d'abord une augmentation de flot à partir de μ en exécutant la procédure AUGMENTER-FLOT(f, μ) et met ensuite à jour le potentiel de chaque sommet comme l'indique

la procédure MODIFIER-POTENTIELS(f, π) ci-dessous. Dans cette procédure, on note T l'ensemble des sommets accessibles à partir de s dans G_f et $\bar{l}_\pi(t)$ le coût réduit minimum d'un chemin de s à t dans G_f.

```
procédure MODIFIER-POTENTIELS(f, π);
   M :=max_{t∈T} l̄_π(t);
   pour tout sommet t de T faire π(t) :=π(t) + l̄_π(t);
   pour tout sommet z de S − T faire π(z) :=π(z) + M.
```

Cette procédure augmente le potentiel de tout sommet t de T du coût réduit minimum d'un chemin de s à t dans G_f et augmente le potentiel des autres sommets d'une même quantité M égale au plus grand coût réduit minimum d'un chemin d'origine s dans G_f. L'itération courante, imlémentée par la procédure CHANGER-DE-COUPLE ci-dessous réalise le changement de flot et la modification des potentiels :

```
procédure CHANGER-DE-COUPLE(f, π, μ);
   AUGMENTER-FLOT(f, μ);
   MODIFIER-POTENTIELS(f, π).
```

La proposition 4.12 montre que les coûts réduits du nouveau graphe d'écart et relatifs aux nouveaux potentiels sont positifs ou nuls.

Proposition 4.12. *Les coûts réduits du couple (f', π') calculés par la procédure* CHANGER-DE-COUPLE(f, π, μ) *sont positifs ou nuls.*

Preuve. Considérons d'abord le cas d'un arc v de $G_{f'}$ qui n'existait pas dans G_f. Si $v = u'$, l'arc u'' est un arc de μ. Comme μ est un chemin de coût réduit minimum pour le couple (f, π), on a :

$$\bar{l}_\pi(u^-) = \bar{l}_\pi(u^+) + \bar{c}_\pi(u'').$$

Comme u^- et u^+ appartiennent à T, nous avons :

$$\bar{c}_{\pi'}(u') = c(u') + \pi'(u^-) - \pi'(u^+) = -c(u'') + (\pi(u^-) + \bar{l}_\pi(u^-)) - (\pi(u^+) + \bar{l}_\pi(u^+)).$$

En regroupant les termes correspondant à $\bar{c}_\pi(u'')$, il vient :

$$\bar{c}_{\pi'}(u') = -\bar{c}_\pi(u'') + \bar{l}_\pi(u^-) - \bar{l}_\pi(u^+) = 0.$$

Si $v = u''$, un calcul analogue montre que le nouveau coût réduit de l'arc v est nul.

Considérons maintenant un arc v de $G_{f'}$ qui est aussi dans G_f. Il résulte de la définition de T que seuls les trois cas suivants sont possibles :

(1) $v^- \in T$, $v^+ \in T$. On a alors par définition de $\bar{l}_\pi$:

$$\bar{c}_{\pi'}(v) = \bar{c}_\pi(v) + \bar{l}_\pi(v^-) - \bar{l}_\pi(v^+) \geq 0.$$

(2) $v^- \in S - T$, $v^+ \in S - T$. On a alors puisque (f, π) est dual-réalisable :

$$\bar{c}_{\pi'}(v) = \bar{c}_\pi(v) + M - M \geq 0.$$

(3) $v^- \in S - T$, $v^+ \in T$. On a alors par définition de M :

$$\bar{c}_{\pi'}(v) = \bar{c}_\pi(v) + M - \bar{l}_\pi(v^+) \geq 0.$$

Il en résulte que tous les nouveaux coûts réduits sont positifs ou nuls. ∎

Terminaison

Chaque augmentation de la valeur du flot réalisée par la procédure CHANGER-DE-COUPLE porte sur au moins une unité de flux. L'algorithme se termine donc après au plus $B = \min\{b^-(p), b^+(s)\}$ itérations. A l'issue de la dernière itération, le flot f est de valeur maximum et les arcs de G_f ont un coût réduit positif ou nul par rapport au dernier ensemble de potentiels. La procédure FLOT-MAX-COÛT-MIN(G, b, s, p, c) ci-dessous décrit l'algorithme complet.

```
procédure FLOT-MAX-COÛT-MIN(G, b, s, p, c);
   f := 0; π =: 0; T := S;
   pour tout t de T faire calculer l̄_π(t);
   tantque p ∈ T faire
      soit μ un chemin améliorant de coût réduit l̄_π(p);
      CHANGER-DE-COUPLE(f, π, μ);
      T :=SOMMETS-ACCESSIBLES(f, π);
      pour tout t de T faire calculer l̄_π(t)
   fintantque.
```

Complexité

Le nombre d'itérations de l'algorithme est majoré par $B = \min\{b^-(p), b^+(s)\}$ puisque chaque changement de couple augmente la valeur du flot d'au moins une unité. La complexité d'une itération est dominée par le calcul des coûts réduits minimum des chemins de s aux sommets de T. Pour ce calcul on peut utiliser l'algorithme de Dijkstra dont la complexité est $O(m \log n)$ pour un graphe d'écart

quelconque. Il en résulte une complexité globale $O(mB \log n)$ pour l'algorithme FLOT-MAX-COÛT-MIN(G, b, s, p, c). Cet algorithme n'est donc pas polynomial, mais seulement pseudo-polynomial en raison du terme B. La faiblesse de cet algorithme tient clairement au fait qu'une itération complète est réalisée pour une augmentation insuffisante de la valeur du flot.

8.4.5 Plan de transport de coût minimum

Nous considérons dans cette section un problème de transport spécifié par un graphe $G = (S, A)$, une fonction de coût de $\mathcal{F}(A)$ positive ou nulle et une fonction d de $\mathcal{F}(S)$, définissant l'offre et la demande et vérifiant $\sum_{s \in S} d(s) = 0$. La fonction d permet de classer les sommets en sommets fournisseurs (si $d(s) > 0$), sommets clients (si $d(s) < 0$) et sommets de transit (si $d(s) = 0$). On note respectivement F, C et T les sous-ensembles des sommets fournisseurs, clients et de transit. On suppose de plus qu'il existe au moins un chemin de G liant un sommet fournisseur quelconque à un sommet client quelconque. Un *plan de transport* est une fonction $x : A \mapsto \mathbb{Q}^+$ telle que :

$$\begin{aligned} \forall u \in A, \quad & x(u) \geq 0 \\ \forall s \in S, \quad & e_x(s) = -d(s). \end{aligned}$$

Un plan de transport est de coût minimum si son coût $c \star x$ est minimum.

Ce problème est un cas particulier important du problème du flot maximum de coût minimum, il se pose en effet fréquemment dans les applications.

Le graphe d'écart G_x d'un plan de transport x est défini de la même manière que le graphe d'écart G_f d'un flot f, à cette remarque près que chaque arc de G possède un représentant conforme valué par $+\infty$. Tout chemin de G produit donc un chemin conforme dans G_x passant par les mêmes sommets. Un raisonnement tout à fait analogue à celui mené dans la section 8.4.4 permet d'établir une condition d'optimalité d'un couple (x, π) où x est un plan de transport et π un potentiel sur les sommets de S. Cette proposition est en fait un corollaire de la proposition 4.2.

Proposition 4.13. *Un plan de transport x et un potentiel π sont optimaux si et seulement si les coûts réduits des arcs du graphe d'écart G_x sont positifs ou nuls.*

L'algorithme de Edmonds et Karp que nous allons décrire est une extension de l'algorithme FLOT-MAX-COÛT-MIN du paragraphe précédent. Il s'agit donc d'un algorithme dual qui repose sur l'idée qu'il convient de satisfaire en priorité les couples (fournisseur,client) qui sont les plus déficitaires du point de vue de l'offre et de la demande. A cet effet, on appellera *pseudo-plan* une fonction x de $\mathcal{F}(A)$ positive ou nulle et l'on notera $\delta_x(s) = d(s) - e_x(s)$ le *déficit* du sommet s pour le pseudo-plan x. Si un couple (x, π) dual-réalisable est tel que tous les déficits sont nuls, alors x est un plan de transport de coût minimum.

Soit x un pseudo-plan et soit Δ un entier naturel. Nous notons $P_x(\Delta)$ (respectivement $N_x(\Delta)$) le sous-ensemble des sommets dont le déficit est supérieur ou égal à Δ (respectivement inférieur ou égal à $-\Delta$). Le pseudoplan x est dit Δ-*optimal* si au moins l'un des deux ensembles $P_x(\Delta)$ ou $N_x(\Delta)$ est vide. Une phase de l'algorithme de Edmonds et Karp consiste à transformer un pseudoplan Δ-optimal en un pseudoplan $\Delta/2$-optimal en exécutant au plus n fois l'algorithme de Dijkstra pour la recherche des chemins de coût réduit minimum.

Initialisation

L'algorithme est initialisé par le pseudoplan $x = 0$, le potentiel $\pi = 0$ et $\Delta = B$ où $B = \max_{s \in S} d(s)$. Remarquons que ce premier pseudoplan est Δ-optimal puisque $P_x(\Delta) = \emptyset$.

Phase courante

Soit x le pseudoplan courant 2Δ-optimal résultant de la phase précédente. Une itération de la phase suivante choisit d'abord un couple de sommets (s, t) où $s \in P_x(\Delta)$ et $t \in N_x(\Delta)$. Elle détermine ensuite un chemin μ de coût réduit minimal dans G_x de s à t, diminue les déficits des sommets s et t de la quantité Δ en augmentant de Δ les flux transportés par les arcs du chemin de G formé par les pères des arcs de μ. Elle modifie enfin les potentiels pour que le nouveau pseudoplan soit dual-réalisable. Cette itération est répétée tant que le pseudoplan x n'est pas Δ-optimal. La procédure ci-dessous réalise la phase courante de l'algorithme de Edmonds et Karp.

```
procédure PHASE-EDMONDS-KARP(Δ);
  P := {s | δ_x(s) ≥ Δ};
  N := {s | δ_x(s) ≤ −Δ};
  tantque (P ≠ ∅ et N ≠ ∅) faire
    choisir s dans P et t dans N;
    pour tout sommet z de S faire
      calculer l̄_π(s, z)
    finpour;
    soit μ un chemin de coût réduit minimum de s à t;
    MODIFIER-PSEUDOPLAN(x, μ);
    pour tout sommet z faire π(z) := π(z) + l̄_π(s, z);
  fintantque;
  Δ := Δ/2.
```

La procédure MODIFIER-PSEUDOPLAN est une variante simplifiée de la procédure AUGMENTER-FLOT qui augmente systématiquement de Δ les flux des pères des arcs de μ.

Terminaison

L'algorithme de Edmonds et Karp se termine dès que $\Delta < 1$. Tous les déficits sont alors nuls. D'où la procédure pour l'algorithme complet :

```
procédure EDMONDS-KARP(R);
   (x, π) := (0, 0); Δ := B;
   tantque Δ ≥ 1 faire
      PHASE-EDMONDS-KARP(Δ)
   fintantque.
```

Les figures 4.2 et 4.3 illustrent le comportement de l'algorithme sur un exemple. Pour chaque sommet, le coin sud-est contient le coût réduit minimum de l'origine au sommet, le coin nord-est contient le potentiel courant, le coin nord-ouest contient le nom du sommet et le coin sud-ouest contient le déficit de ce sommet. Sur chaque arc est inscrit le coût réduit par rapport au potentiel en cours. La partie supérieure de la figure 4.2 représente le réseau initial qui est aussi le premier réseau d'écart ; sa partie inférieure représente le réseau d'écart après la première phase ($\Delta = 2$). Cette première phase choisit le chemin (arcs épais) μ_1=(a, e, f, c) et augmente les flux de ses arcs de trois unités. La figure 4.3 correspond à la

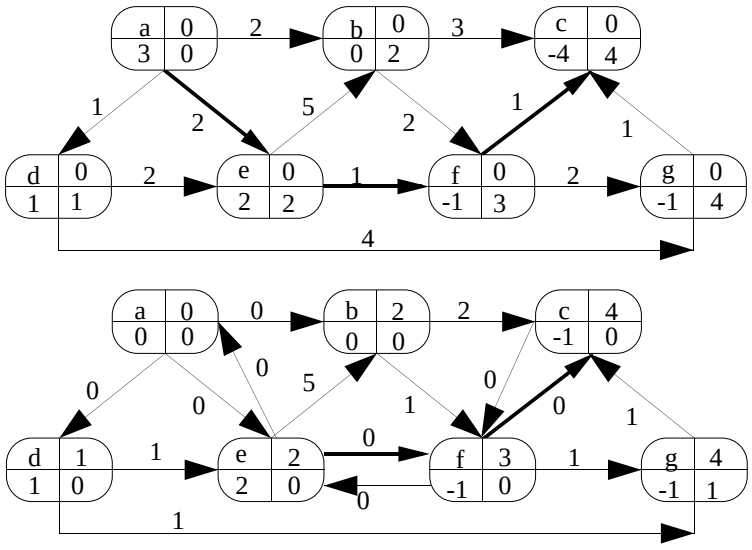


Figure 4.2: *Première phase.*

seconde phase ($\Delta = 1$). Les chemins successivement choisis sont μ_2=(e, f, c), μ_3=(d, g) et μ_4=(e, f), et chacun conduit à une réduction du déficit de une unité. Après ces trois réductions, le déficit total est nul et le plan $3\chi_{\mu_1} + \chi_{\mu_2} + \chi_{\mu_3} + \chi_{\mu_4}$ est optimal.

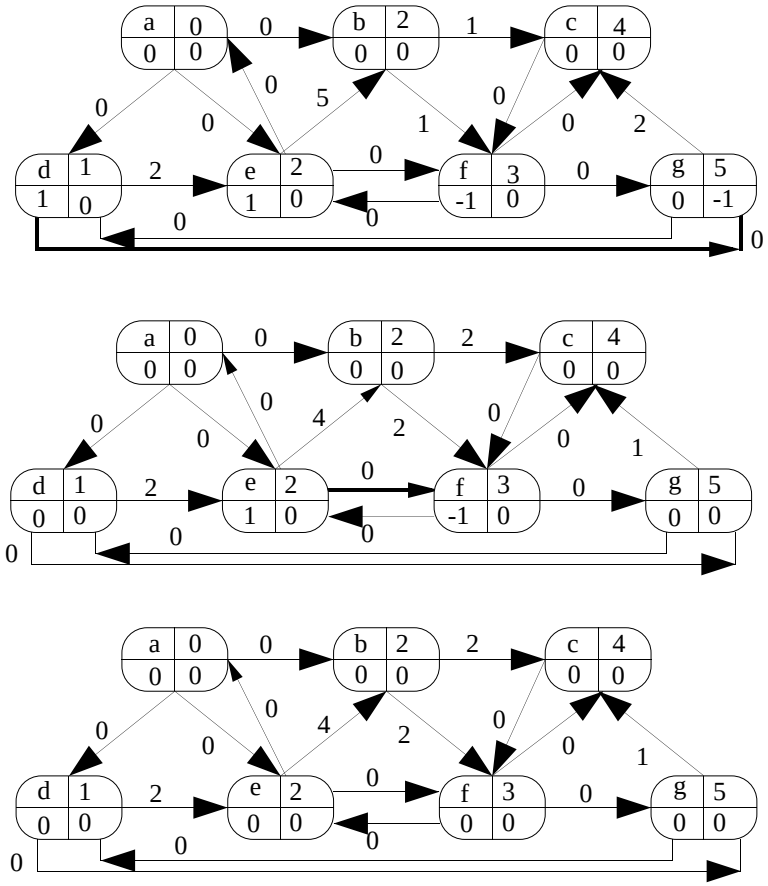


Figure 4.3: *Seconde phase.*

Convergence et Complexité

Comme l'algorithme exécute exactement $1 + \log B$ phases, il se termine si chaque phase réalise un nombre fini d'itérations. Comme nous allons montrer que chaque phase réalise au plus n itérations, l'algorithme se termine et le dernier pseudoplan est un plan de transport optimal car tous les déficits sont nuls.

Lemme 4.14. *Une phase exécute au plus n itérations.*

Preuve. Supposons qu'une phase se termine parce que l'ensemble $P_x(2\Delta)$ est vide. Considérons alors une itération quelconque de la phase suivante et soit μ le chemin de G_x d'origine s et d'extrémité t associé à cette itération. Le déficit du sommet s est réduit de Δ unités par construction et par conséquent le sommet s dont le déficit devient strictement inférieur à Δ ne sera plus candidat à une réduction de déficit au cours de cet même phase. Un raisonnement analogue peut être fait si c'est l'ensemble $N_x(2\Delta)$ qui est vide. Le lemme en résulte. ∎

La complexité de l'algorithme de Edmonds et Karp est $O(nT(m,n)\log B)$ où $T(m,n)$ est la complexité de l'algorithme de Dijkstra utilisé pour déterminer à chaque itération les chemins de coût réduit minimum.

Lorsque les capacités maximales du réseau initial R sont finies, il est possible (voir exercices) de le remplacer par un réseau «équivalent» R' sans limitations de capacités et comportant $n+m$ sommets et $2m$ arcs. En appliquant l'algorithme de Edmonds et Karp au réseau R' et en utilisant une version modifiée de l'algorithme de Dijkstra permettant le calcul des coûts réduits minimum pour le réseau R' avec la même complexité $T(m,n)$ que sur le réseau R, on aboutit à un algorithme de complexité $O(\min\{m \log B, m \log n\} T(m,n))$ qui est l'un des plus efficaces connus aujourd'hui.

Notes

Comme les flots constituent le modèle de base de nombreuses applications, ils ont fait l'objet d'une recherche très intense depuis les premiers résultats de Ford et Fulkerson en 1956. Ces recherches ont eu essentiellement pour objectif de développer des algorithmes de plus en plus efficaces. Citons deux ouvrages qui contiennent les résultats les plus récents du domaine :

R.E. Tarjan, *Data Structures and Network Algorithms*, SIAM Philadelphia, chapitre 7,1983

R.K. Ahuja, T.L. Magnanti, et J.B. Orlin, *Network Flows*, Sloan School of Management, Work Report, MIT, 1988

L'idée, due à Edmonds et Karp, de choisir comme chemins améliorants les plus courts en nombre d'arcs a conduit à un algorithme de complexité $O(m^2n)$. Indépendamment, E.A. Dinic a introduit les réseaux à niveaux qui sont les sous-graphes du réseau initial contenant les sommets et les arcs appartenant à au moins un plus court chemin améliorant. Par un calcul de flots bloquants dans ces réseaux à niveaux, il obtient un flot maximum en temps $O(n^2m)$.

E.A. Dinic, Algorithm for Solution of a Problem of Maximum Flow in a Network with Power Estimation, *Soviet Math. Dokl.*,11(1970),1277-1280

La notion de distance estimée au puits, due à Goldberg, est une variante du calcul des réseaux à niveaux de Dinic. Elle présente le triple avantage d'être de compréhension plus aisée, de manipulation plus simple et de conduire à des algorithmes plus performants. L'algorithme des distances estimées présenté dans la section 8.3.2 énumère exactement les mêmes chemins améliorants que l'algorithme de Dinic.

La première variante de l'algorithme du préflot est due à A.V. Karzanov qui l'a introduite pour les réseaux à niveaux.

A.V. Karzanov; Determining the Maximal Flow in a Network by the method of Preflows, *Soviet Math. Dokl.*,15(1974),434-437

La variante de l'algorithme du préflot obtenue par échelonnement des excès est due à J.B. Orlin et R.K. Ahuja.

Le problème du flot de coût minimum a aussi été très étudié. Citons quelques étapes depuis les travaux de Ford et Fulkerson en 1962 qui présentèrent les premiers algorithmes primal-dual comme l'algorithme «out of kilter» . Les améliorations sur des chemins du graphe d'écart de coût réduit minimum sont dues indépendamment à Jewell, Iri, Busacker et Gowen. L'utilisation simultanée des ensembles de potentiel pour obtenir des chemins dont les arcs sont de coût réduit positif est due à Edmonds et Karp. C'est dans ce cadre que nous avons présenté l'algorithme dual pour le problème du flot maximum de coût minimum.

Le concept d'ϵ-optimalité fut introduit par Bertsekas en 1979. L'algorithme primal de Goldberg et Tarjan de la section 8.4.3 ainsi que sa variante fortement polynomiale sont développés dans :

A.V.Goldberg et R.E.Tarjan; Finding Minimum Cost Circulations by Canceling Negative Cycles, *Proc. 20th ACM Symp. on the Theory of Computing*, 388-397,(1988).

Pour un problème de transport, l'idée de réaliser un échelonnement des déficits des sommets est due à Edmonds et Karp. L'algorithme qui en résulte pour la recherche d'un plan de transport de coût minimum est développé dans :

J. Edmonds et R.M. Karp, Theoretical Improvements in Algorithmic Efficiency for Network Flow Problems, *J. ACM*,19,(1972),248-264.

Cet algorithme a été étendu aux réseaux avec capacités maximales par J.B. Orlin qui a obtenu une complexité en temps de $O(m \log n(m + n \log n))$ dans :

J.B. Orlin; A Faster Strongly Polynomial Minimum-Cost Flow Algorithm, *Proc. 20th ACM Syp. on the Theory of Computing*, 377-387,(1988).

Exercices

8.1. Soit $R = (G, a, b)$ un réseau.

a) Construire à partir de R un réseau de transport R' muni d'une source s et d'un puits p tel que R est consistant si et seulement si R' possède un flot saturant.

b)En déduire un algorithme pour déterminer si un réseau est consistant.

8.2. Soit $G = (S, A)$ un graphe orienté biparti où $S = X \cup Y$, $X \cap Y = \emptyset$ et $A \subset X \times Y$. On appelle *couplage* de G une partie C de A telle que deux arcs quelconques de C n'ont pas d'extrémité commune. Un couplage C est dit parfait si tout sommet de X est l'origine d'un arc de C.

a) Démontrer que les couplages de G sont en bijection avec les flots entiers d'un réseau que l'on construira.

b) En déduire que G possède un couplage parfait si et seulement si :

$$\forall T \subset X, \quad \mathrm{Card}(\Gamma^+(T)) \geq \mathrm{Card}(T)$$

8.3. Soit $G = (S, A)$ un graphe orienté connexe à n sommets et m arcs et soit T un sous-ensemble de sommets. On appelle *cocycle associé* à T le sous-ensemble d'arcs $\omega(T)=\omega^+(T)\cup\omega^-(T)$. On associe au cocycle $\omega(T)$ le vecteur caractéristique $\vec{T}$ de $\{0, 1, -1\}^m$ défini par :

$$\vec{T}(u) = \begin{cases} 1 & \text{si } u \in \omega^+(T) \\ -1 & \text{si } u \in \omega^-(T) \\ 0 & \text{sinon} \end{cases}$$

a) Montrer que les vecteurs caractéristiques d'un cycle et d'un cocycle quelconques sont orthogonaux. Montrer que le sous-espace vectoriel de $\mathbb{R}^m$ engendré par les cocycles est de dimension $n - 1$.

8.4. Démontrer la proposition 2.8 . **8.5.** Soit $R = (G, b, s, p)$ le réseau d'un problème de flot maximum. Construire un réseau R contenant plusieurs coupes de valeur minimum. On note $\mathcal{C}(R)$ l'ensemble des coupes de valeur minimum du réseau R. Montrer que $\mathcal{C}(R)$ est stable pour l'union et l'intersection.

8.6. Démontrer la proposition 4.1.

8.7. Soit $G = (S, A)$ un graphe orienté fortement connexe et $c : A \mapsto \mathbb{Z}$ une fonction coût sur les arcs. On suppose qu'il n'existe pas de circuit de coût strictement négatif. On note $F_k(x)$ le coût minimum d'un chemin de k arcs de s à x dans G en posant $F_k(x) = +\infty$ si un tel chemin n'existe pas. Si γ est u circuit de G à p arcs, le *coût moyen* de γ est défini par : $\bar{c}(\gamma) = c(\gamma)/p$. Il s'agit alors de calculer par l'*algorithme de Karp* le coût moyen minimum d'un circuit de G, c'est-à-dire la valeur : $\lambda^* = \min\{\bar{c}(\gamma) \mid \gamma \in \mathcal{C}(G)\}$ où $\mathcal{C}(G)$ est l'ensemble des circuits de G.

a) Montrer que :

$$\min_{x\in S} \max_{k\in\{0,\ldots,n-1\}} \{\frac{F_n(x) - F_k(x)}{n-k}\} = 0 \Rightarrow \lambda^* = 0$$

On notera $\pi(x)$ le coût minimum d'un chemin de s à x dans G et on montrera que :

$$\max_{k\in\{0,\ldots,n-1\}} \{\frac{F_n(x) - F_k(x)}{n-k}\} \geq 0$$

et que l'égalité a lieu si et seulement si $F_n(x) = \pi(x)$. On montrera ensuite qu'il existe un sommet y tel que $F_n(y) = \pi(y)$.
En déduire que :

$$\lambda^* = \min_{x\in S} \max_{k\in\{0,\ldots,n-1\}} \{\frac{F_n(x) - F_k(x)}{n-k}\}$$

8.8. Soit (G, b, s, p) le réseau d'un problème de flot maximum. On suppose que :

$$2^{K-1} < B = \max_{u\in A} b(u) \leq 2^K - 1$$

On peut donc coder chaque capacité avec au plus K bits. Si p est un entier naturel codé sur K bits, on note $p^{(k)}$, $k \in \{1, \ldots, K\}$, l'entier associé aux k bits de poids fort de p (par exemple si $p = 7$ et $K = 5$, alors $p^{(3)} = 1$). On considère le problème $P_k = (G, b_k, s, p)$ où $b_k(u) = b(u)^{(k)}$.

a) Montrer que

$$b_{k+1}(u) \in \{2b_k(u), 2b_k(u) + 1\}$$

Soit f_k^* la valeur maximum d'un flot de P_k, montrer que

$$f_{k+1}^* \leq 2f_k^* + m$$

où m est le nombre d'arcs du réseau.

b) En déduire une variante polynomiale de l'algorithme «générique» de Ford et Fulkerson, de complexité $O(nm \log B)$.

8.9. Soit $R = (G, a, b, c)$ un réseau valué muni d'une fonction d'offre et de demande $d : S \mapsto \mathbb{Z}$ telle que $\sum_S d(s) = 0$. Soit u un arc de G dont la capacité minimale $a(u)$ est strictement positive.

a) Montrer que l'on peut transformer $d(u^-)$, $d(u^+)$, $a(u)$ et $b(u)$ en $d'(u^-)$, $d'(u^+)$, $a'(u) = 0$ et $b'(u)$ de sorte que les flots de R et de R' soient en bijection et que les coûts de deux flots associés dans la bijection diffèrent d'une constante.

Soit $R = (G, 0, b, c)$ un réseau valué muni d'une fonction d'offre et de demande $d : S \mapsto \mathbb{Z}$ telle que $\sum_S d(s) = 0$. Soit u un arc de G.

b) Montrer que l'on peut transformer l'arc u en deux arcs u_1 et u_2 de capacité minimale nulle et de capacité maximale infinie, tels que :

$u_1^- = u^-, \quad u_2^- = u^+, \quad u_1^+ = u_2^+ = \alpha$ où α est un nouveau sommet ;
les flots de R et du nouveau réseau R' se correspondent dans une bijection conservant le coût.

Chapitre 9

Automates

Dans ce chapitre, nous présentons les bases de la théorie des automates finis. Nous montrons l'équivalence entre les automates finis et les automates finis déterministes, puis nous étudions des propriétés de fermeture. Nous prouvons le théorème de Kleene qui montre que les langages reconnaissables et les langages rationnels sont une seule et même famille de langages. Nous prouvons l'existence et l'unicité d'un automate déterministe minimal reconnaissant un langage donné, et nous présentons l'algorithme de Hopcroft de minimisation.

Introduction

Les automates finis constituent l'un des modèles de calcul les plus anciens en informatique. A l'origine, ils ont été conçus et employés comme une tentative de modélisation des neurones ; les premiers résultats théoriques, comme le théorème de Kleene, datent de cette époque. Parallèlement, ils ont été utilisés en tant qu'outils de développement des circuits logiques. Les applications les plus courantes sont à présent le traitement de texte, dans son sens le plus général. L'analyse lexicale, la première phase d'un compilateur, est réalisée par des algorithmes qui reproduisent le fonctionnement d'un automate fini. La spécification d'un analyseur lexical se fait d'ailleurs souvent en donnant les expressions rationnelles des mots à reconnaître. Dans le traitement de langues naturelles, on retrouve les automates finis sous le terme de réseau de transitions. Enfin, le traitement de texte proprement dit fait largement appel aux automates, que ce soit pour la reconnaissance de motifs – qui constitue l'objet du chapitre suivant – ou pour la description de chaînes de caractères, sous le vocable d'expressions régulières. De nombreuses primitives courantes dans les systèmes d'exploitation modernes font appel, implicitement ou explicitement, à ces concepts.

La théorie des automates a également connu de grands développements du point de vue mathématique, en liaison étroite avec la théorie des monoïdes finis, princi-

palement sous l'impulsion de M. P. Schützenberger. Elle a aussi des liens profonds avec les théories logiques.

9.1 Mots et langages

Soit A un ensemble. On appelle *mot* sur A toute suite finie

$$u = (a_1, \ldots, a_n)$$

où $n \geq 0$, et $a_i \in A$ pour $i = 1, \ldots, n$. L'entier n est la *longueur* de u, notée $|u|$. Si $n = 0$, le mot est appelé le *mot vide*, et est noté 1 ou ε. Si

$$v = (b_1, \ldots, b_m)$$

est un autre mot, le *produit de concaténation* de u et v est le mot

$$uv = (a_1, \ldots, a_n, b_1, \ldots, b_m)$$

Tout mot étant le produit de concaténation de mots de longueur 1, on identifie les mots de longueur 1 et les éléments de A. On appelle alors A l'*alphabet*, et les éléments de A des *lettres*. On note A^* l'ensemble des mots sur A.

Si u, v et w sont des mots et $w = uv$, alors u est un *préfixe* et v est un *suffixe* de w. Si de plus $u \neq w$ (resp. $v \neq w$), alors u est un *préfixe propre* (resp. un *suffixe propre*) de w. Le mot v est un *facteur* d'un mot w s'il existe des mots u et u' tels que $w = uvu'$.

Un *langage* (formel) sur A est une partie X de A^*. Si X et Y sont des langages, le *produit* XY est défini par

$$XY = \{xy \mid x \in X,\, y \in Y\}$$

Ce produit est associatif, et le langage $\{1\}$ est élément neutre pour le produit. On définit les puissances de X par $X^0 = \{1\}$, et $X^{n+1} = X^n X$ pour $n \geq 0$. L'*étoile* de X est le langage

$$X^* = \bigcup_{n \geq 0} X^n$$

C'est l'ensemble de tous les produits $x_1 \cdots x_n$, pour $n \geq 0$ et $x_1, \ldots, x_n \in X$. Le mot vide appartient toujours à X^*. Il est facile de vérifier que

$$X^* = \{1\} \cup XX^* = \{1\} \cup X^*X$$

Il est commode de noter X^+ l'ensemble $XX^* = X^*X$. Souvent, on omettra des accolades autour de singletons; ainsi, on écrira w au lieu de $\{w\}$.

9.2 Automates finis

9.2.1 Définition

Un *automate fini* sur un alphabet fini A est composé d'un ensemble fini Q d'*états*, d'un ensemble $I \subset Q$ d'états *initiaux* , d'un ensemble $T \subset Q$ d'états *terminaux* ou *finals* et d'un ensemble $\mathcal{F} \subset Q \times A \times Q$ de *flèches*. Un automate est habituellement noté

$$\mathcal{A} = (Q, I, T, \mathcal{F})$$

Parfois, on écrit plus simplement $\mathcal{A} = (Q, I, T)$, lorsque l'ensemble des flèches est sous-entendu, mais cette notation est critiquable car l'ensemble des flèches est essentiel. L'*étiquette* d'une flèche $f = (p, a, q)$ est la lettre a. Un *calcul* de longueur n dans $\mathcal{A}$ est une suite $c = f_1 \cdots f_n$ de flèches consécutives $f_i = (p_i, a_i, q_i)$, c'est-à-dire telles que $q_i = p_{i+1}$ pour $i = 1, \ldots, n-1$. L'*étiquette* du calcul c est $|c| = a_1 \cdots a_n$. On écrit également, si $w = |c|$,

$$c : p_1 \to q_n \quad \text{ou} \quad c : p_1 \xrightarrow{w} q_n$$

Par convention, il existe un calcul vide $1_q : q \to q$ d'étiquette ε ou 1 (le mot vide) pour chaque état q. Les calculs peuvent être composés. Etant donnés deux calculs $c : p \to q$ et $d : q \to r$, le calcul $cd : p \to r$ est défini par concaténation. On a bien entendu $|cd| = |c|\,|d|$.

Un calcul $c : i \to t$ est dit *réussi* si $i \in I$ et $t \in T$. Un mot est *reconnu* s'il est l'étiquette d'un calcul réussi. Le *langage reconnu* par l'automate $\mathcal{A}$ est l'ensemble des mots reconnus par $\mathcal{A}$, soit

$$L(\mathcal{A}) = \{w \in A^* \mid \exists c : i \to t,\ i \in I,\ t \in T,\ w = |c|\}$$

Une partie $X \subset A^*$ est *reconnaissable* s'il existe un automate fini $\mathcal{A}$ sur A telle que $X = L(\mathcal{A})$. La famille de toutes les parties reconnaissables de A^* est notée $\mathrm{Rec}(A^*)$.

La terminologie qui vient d'être introduite suggère d'elle-même une représentation graphique d'un automate fini par ce qui est appelé son *diagramme d'états* : les états sont représentés par les sommets d'un graphe (plus précisément d'un multigraphe). Chaque flèche (p, a, q) est représentée par un arc étiqueté qui relie l'état de départ p à l'état d'arrivée q, et qui est étiqueté par l'étiquette a de la flèche. Un calcul n'est autre qu'un chemin dans le multigraphe, et l'étiquette du calcul est la suite des étiquettes des arcs composant le chemin. Dans les figures, on attribue un signe distinctif aux états initiaux et terminaux. Un état initial est muni d'une flèche qui pointe sur lui, un état final est repéré par une flèche qui le quitte. Parfois, nous convenons de réunir en un seul arc plusieurs arcs étiquetés ayant les mêmes extrémités. L'arc porte alors l'ensemble de ces étiquettes. En particulier, s'il y a une flèche (p, a, q) pour toute lettre $a \in A$, on tracera une flèche unique d'étiquette A.

9.2.2 Exemples

L'automate de la figure 2.1 est défini sur l'alphabet $A = \{a, b\}$.

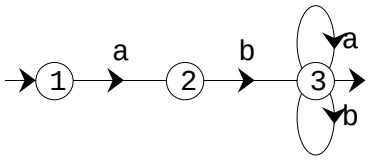


Figure 2.1: *Automate reconnaissant le langage abA^*.*

L'état initial est l'état 1, le seul état final est 3. Tout calcul réussi se factorise en

$$1 \xrightarrow{a} 2 \xrightarrow{b} 3 \xrightarrow{w} 3$$

avec $w \in A^*$. Le langage reconnu est donc bien abA^*. Toujours sur l'alphabet $A = \{a, b\}$, considérons l'automate de la figure 2.2.

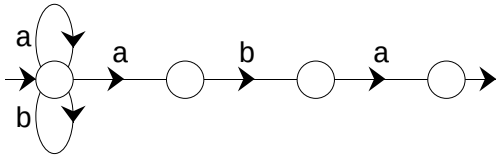


Figure 2.2: *Automate reconnaissant le langage A^*aba.*

Cet automate reconnaît l'ensemble A^*aba des mots qui se terminent par aba. L'automate de la figure 2.3 est défini sur l'alphabet $A = \{a, b\}$.

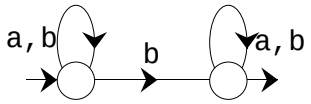


Figure 2.3: *Automate reconnaissant les mots contenant au moins un b.*

Tout calcul réussi contient exactement une fois la flèche $(1, b, 2)$. Un mot est donc reconnu si et seulement s'il contient au moins une fois la lettre b.

L'automate de la figure 2.4 reconnaît l'ensemble des mots contenant un nombre impair de a. Un autre exemple est l'automate vide, ne contenant pas d'états. Il reconnaît le langage vide. A l'inverse, l'automate de la figure 2.5 ayant un seul état qui est à la fois initial et terminal, et une flèche pour chaque lettre $a \in A$ reconnaît tous les mots.

Enfin, le premier des deux automates de la figure 2.6 ne reconnaît que le mot vide, alors que le deuxième reconnaît tous les mots sur A sauf le mot vide, c'est-à-dire le langage A^+.

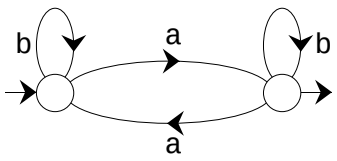


Figure 2.4: *Automate reconnaissant les mots contenant un nombre impair de a.*

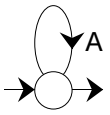


Figure 2.5: *Tous les mots sont reconnus.*

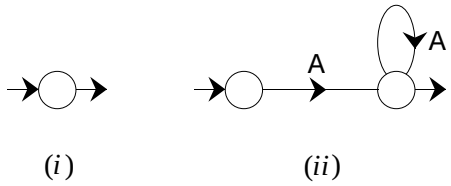


Figure 2.6: *Automates reconnaissant* (i) *le mot vide et* (ii) *tous les mots sauf le mot vide.*

9.2.3 Automates déterministes

Un état $q \in Q$ d'un automate $\mathcal{A} = (Q, I, T, \mathcal{F})$ est *accessible* s'il existe un calcul $c : i \to q$ avec $i \in I$. De même, l'état q est *coaccessible* s'il existe un calcul $c : q \to t$ avec $t \in T$. Un automate est *émondé* si tous ses états sont accessibles et coaccessibles. Soit P l'ensemble des états qui sont à la fois accessibles et coaccessibles, et soit $\mathcal{A}^0 = (P, I \cap P, T \cap P, \mathcal{F} \cap P \times A \times P)$. Il est clair que $\mathcal{A}^0$ est émondé. Comme tout calcul réussi de $\mathcal{A}$ ne passe que par des états accessibles et coaccessibles, on a $L(\mathcal{A}^0) = L(\mathcal{A})$.

Emonder un automate fini revient à calculer l'ensemble des états qui sont descendants d'un état initial et ascendants d'un état final. Cela peut se faire en temps linéaire en fonction du nombre de flèches (d'arcs), par les algorithmes du chapitre 4.

Dans la pratique, les automates déterministes que nous définissons maintenant sont les plus importants, notamment parce qu'ils sont faciles à implémenter.

Un automate $\mathcal{A} = (Q, I, T, \mathcal{F})$ est *déterministe* s'il possède un seul état initial (c'est-à-dire $|I| = 1$) et si

$$(p, a, q), (p, a, q') \in \mathcal{F} \Rightarrow q = q'$$

Ainsi, pour tout $p \in Q$ et tout $a \in A$, il existe au plus un état q dans Q tel que

$(p, a, q) \in \mathcal{F}$. On pose alors, pour $p \in Q$ et $a \in A$,

$$p \cdot a = \begin{cases} q & \text{si } (p, a, q) \in \mathcal{F}, \\ \emptyset & \text{sinon.} \end{cases}$$

On définit ainsi une fonction partielle

$$Q \times A \to Q$$

appelée la *fonction de transition* de l'automate déterministe. On l'étend aux mots en posant, pour $p \in Q$,

$$p \cdot 1 = p$$

et pour $w \in A^*$, et $a \in A$,

$$p \cdot wa = (p \cdot w) \cdot a$$

Cette notation signifie que $p \cdot wa$ est défini si et seulement si $p \cdot w$ et $(p \cdot w) \cdot a$ sont définis, et dans l'affirmative, $p \cdot wa$ prend la valeur indiquée. Avec cette notation on a, avec $I = \{i\}$,

$$L(\mathcal{A}) = \{w \in A^* \mid i \cdot w \in T\}.$$

Un automate est dit *complet* si, pour tout $p \in Q$ et $a \in A$, il existe au moins un état $q \in Q$ tel que $(p, a, q) \in \mathcal{F}$. Si un automate fini $\mathcal{A}$ n'est pas complet, on peut le compléter sans changer le langage reconnu en ajoutant un nouvel état non final s, et en ajoutant les flèches (p, a, s) pour tout couple (p, a) tel que $(p, a, q) \notin \mathcal{F}$ pour tout $q \in Q$. Rendre un automate déterministe, c'est-à-dire le *déterminiser*, est plus difficile, et donne lieu à un algorithme intéressant.

Théorème 2.1. *Pour tout automate fini $\mathcal{A}$, il existe un automate fini déterministe et complet $\mathcal{B}$ tel que*

$$L(\mathcal{A}) = L(\mathcal{B}).$$

Preuve. Soit $\mathcal{A} = (Q, I, T, \mathcal{F})$. On définit un automate déterministe $\mathcal{B}$ qui a pour ensemble d'états l'ensemble $\wp(Q)$ des parties de Q, pour état initial I, et pour ensemble d'états terminaux $V = \{S \subset Q \mid S \cap T \neq \emptyset\}$. On définit enfin la fonction de transition de $\mathcal{B}$ pour $S \in \wp(Q)$ et $a \in A$ par

$$S \cdot a = \{q \in Q \mid \exists s \in S : (s, a, q) \in \mathcal{F}\}.$$

Nous prouvons par récurrence sur la longueur d'un mot w que

$$S \cdot w = \{q \in Q \mid \exists s \in S : s \xrightarrow{w} q\}.$$

Ceci est clair si $w = 1$, et est vrai par définition si w est une lettre. Posons $w = va$, avec $v \in A^*$ et $a \in A$. Alors comme par définition $S \cdot w = (S \cdot v) \cdot a$, on a $q \in S \cdot w$ si et seulement s'il existe une flèche (p, a, q), avec $p \in S \cdot v$, donc telle qu'il existe un calcul $s \xrightarrow{v} p$ pour un $s \in S$. Ainsi $q \in S \cdot w$ si et seulement s'il existe un calcul $s \xrightarrow{w} q$ avec $s \in S$. Ceci prouve l'assertion.

Maintenant, $w \in L(\mathcal{A})$ si et seulement s'il existe un calcul réussi d'étiquette w, ce qui signifie donc que $I \cdot w$ contient au moins un état final de $\mathcal{A}$, en d'autres termes que $I \cdot w \cap T \neq \emptyset$. Ceci prouve l'égalité $L(\mathcal{A}) = L(\mathcal{B})$. ∎

La démonstration du théorème est constructive. Elle montre que pour un automate $\mathcal{A}$ à n états, on peut construire un automate fini déterministe reconnaissant le même langage et ayant 2^n états. La construction est appelée la *construction par sous-ensembles* (en anglais la «subset construction»).

Exemple. Sur l'alphabet $A = \{a, b\}$, considérons l'automate $\mathcal{A}$ reconnaissant le langage A^*ab des mots se terminant par ab (figure 2.7).

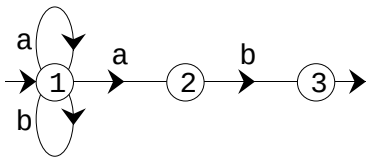


Figure 2.7: *Un automate reconnaissant le langage A^*ab.*

L'application stricte de la preuve du théorème conduit à l'automate déterministe à 8 états de la figure 2.8. Cet automate a beaucoup d'états inaccessibles. Il suffit

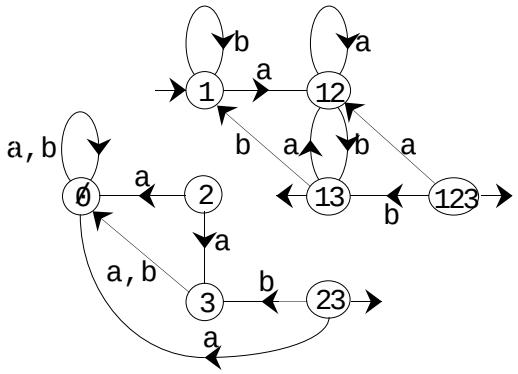


Figure 2.8: *L'automate déterminisé.*

de se contenter des états accessibles, et si l'on n'a pas besoin d'un automate complet, il suffit de considérer la partie émondée; dans notre exemple, on obtient l'automate complet à trois états de la figure 2.9.

En pratique, pour déterminiser un automate $\mathcal{A}$, on ne construit pas l'automate déterministe de la preuve en entier avant de l'émonder; on combine plutôt les deux étapes en une seule, en faisant une recherche des descendants de l'état initial de l'automate $\mathcal{B}$ pendant sa construction. On maintient pour cela un ensemble R d'états de $\mathcal{B}$ déjà construits, et un ensemble de *transitions* (S, a) qui restent à explorer et qui sont composées d'un état de $\mathcal{B}$ déjà construit et d'une lettre. A chaque étape, on choisit un couple (S, a) et on construit l'état $S \cdot a$ de $\mathcal{B}$ en se

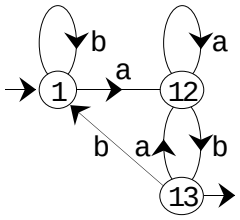


Figure 2.9: *L'automate déterministe et émondé.*

servant de l'automate de départ $\mathcal{A}$. Si l'état $S' = S \cdot a$ est connu parce qu'il figure dans R, on passe à la transition suivante; sinon, on l'ajoute à R et on ajoute à la liste des transitions à explorer les couples (S', a), pour $a \in A$.

Exemple. Reprenons l'exemple ci-dessus. Au départ, l'ensemble d'états de l'automate déterministe est réduit à $\{1\}$, et la liste E des transitions à explorer est $(\{1\}, a)$, $(\{1\}, b)$. On débute donc avec

$$R = \{\{1\}\} \qquad E = (\{1\}, a), (\{1\}, b)$$

Choisissons la première transition, ce qui donne le nouvel état $\{1, 2\}$ et les nouvelles transitions $(\{1, 2\}, a), (\{1, 2\}, b)$, d'où

$$R = \{\{1\}, \{1, 2\}\} \qquad E = (\{1\}, b), (\{1, 2\}, a), (\{1, 2\}, b)$$

Les étapes suivantes sont (noter que l'on a le choix de la transition à traiter, ici on les considère dans l'ordre d'arrivée) :

$$\begin{array}{ll} R = \{\{1\}, \{1, 2\}\} & E = (\{1, 2\}, a), (\{1, 2\}, b) \\ R = \{\{1\}, \{1, 2\}\} & E = (\{1, 2\}, b) \\ R = \{\{1\}, \{1, 2\}, \{1, 3\}\} & E = (\{1, 3\}, a), (\{1, 3\}, b) \\ R = \{\{1\}, \{1, 2\}, \{1, 3\}\} & E = (\{1, 3\}, b) \\ R = \{\{1\}, \{1, 2\}, \{1, 3\}\} & E = \emptyset \end{array}$$

Comme le suggère la construction, il existe des automates à n états pour lesquels tout automate déterministe équivalent possède de l'ordre de 2^n états. Voici, pour tout entier positif n fixé, un automate à $n + 1$ états (figure 2.10) qui reconnaît le langage $L_n = A^* a A^{n-1}$ sur l'alphabet $A = \{a, b\}$. Nous allons démontrer

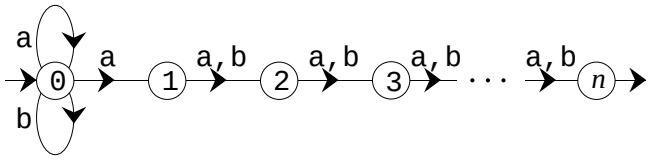


Figure 2.10: *Un automate à $n + 1$ états.*

que tout automate déterministe reconnaissant L_n a au moins 2^n états. Soit en effet $\mathcal{A} = (Q, i, T)$ un automate déterministe reconnaissant L_n. Alors on a, pour $v, v' \in A^n$:

$$i \cdot v = i \cdot v' \Rightarrow v = v' \tag{2.1}$$

En effet, supposons $v \neq v'$. Il existe alors des mots x, x', w tels que $v = xaw$ et $v' = x'bw$ ou vice-versa. Soit y un mot quelconque tel que wy est de longueur $n-1$. Alors $vy = xawy \in L_n$ et $v'y = x'bwy \notin L_n$, alors que l'égalité $i \cdot v = i \cdot v'$ implique que $i \cdot vy = i \cdot v'y$, et donc que vy et vy' soit sont tous deux dans L_n, soit sont tous deux dans le complément de L_n. D'où la contradiction. L'implication (2.1) montre que $\mathcal{A}$ a au moins 2^n états.

9.3 Opérations

9.3.1 Opérations booléennes

Proposition 3.1. *La famille des langages reconnaissables sur un alphabet A est fermée pour les opérations booléennes, c'est-à-dire pour l'union, l'intersection et la complémentation.*

Preuve. Soient X et X' deux langages reconnaissables de A^*, et soient $\mathcal{A} = (Q, i, T)$ et $\mathcal{A}' = (Q', i', T')$ deux automates finis déterministes complets tels que $X = L(\mathcal{A})$ et $X' = L(\mathcal{A}')$. Considérons l'automate déterministe complet

$$\mathcal{B} = (Q \times Q', (i, i'), S)$$

dont la fonction de transition est définie par

$$(p, p') \cdot a = (p \cdot a, p' \cdot a)$$

pour tout couple d'états (p, p') et toute lettre $a \in A$. Alors on a

$$(p, p') \cdot w = (p \cdot w, p' \cdot w)$$

pour tout mot w, comme on le vérifie immédiatement en raisonnant par récurrence sur la longueur de w. Il résulte de cette équation que pour $S = T \times T'$, on obtient $L(\mathcal{B}) = X \cap X'$, et pour $S = (T \times Q') \cup (Q \times T')$, on obtient $L(\mathcal{B}) = X \cup X'$. Enfin, pour $S = T \times (Q' - T')$, on a $L(\mathcal{B}) = X - X'$. ∎

Corollaire 3.2. *Un langage X est reconnaissable si et seulement si $X - \{1\}$ est reconnaissable.* ∎

9.3.2 Automates asynchrones

Nous introduisons maintenant une généralisation des automates finis, dont nous montrons qu'en fait ils reconnaissent les mêmes langages. L'extension réside dans le fait d'autoriser également le mot vide comme étiquette d'une flèche. L'avantage de cette convention est une bien plus grande souplesse dans la construction des

automates. Elle a en revanche l'inconvénient d'accroître le nondéterminisme des automates.

Un *automate fini asynchrone* $\mathcal{A} = (Q, I, T, \mathcal{F})$ est un automate dans lequel certaines flèches peuvent être étiquetées par le mot vide, donc tel que

$$\mathcal{F} \subset Q \times (A \cup 1) \times Q.$$

Les notions de calcul, d'étiquette, de mot et de langage reconnu s'étendent de manière évidente aux automates asynchrones. La terminologie provient de l'observation que, dans un automate asynchrone, la longueur d'un calcul réussi, donc la longueur d'un calcul, peut être supérieure à la longueur du mot qui est son étiquette. Dans un automate usuel en revanche, lecture d'une lettre et progression dans l'automate sont rigoureusement synchronisées.

Exemple. L'automate asynchrone de la figure 3.1 reconnaît le langage a^*b^*.

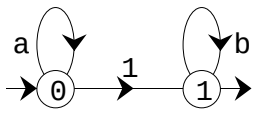


Figure 3.1: *Un automate asynchrone.*

Proposition 3.3. *Pour tout automate fini asynchrone $\mathcal{A}$, le langage $L(\mathcal{A})$ est reconnaissable.*

Preuve. Soit $\mathcal{A} = (Q, I, T)$ un automate fini asynchrone sur A. Soit $\mathcal{B} = (Q, I, T)$ l'automate fini dont les flèches sont les triplets (p, a, q) pour lesquels il existe un calcul $c : p \xrightarrow{a} q$ dans $\mathcal{A}$. On a

$$L(\mathcal{A}) \cap A^+ = L(\mathcal{B}) \cap A^+.$$

Si $I \cap T \neq \emptyset$, les deux langages $L(\mathcal{A})$ et $L(\mathcal{B})$ contiennent le mot vide et sont donc égaux. Dans le cas contraire, on a $L(\mathcal{A}) = L(\mathcal{B}) \cup 1$ ou $L(\mathcal{A}) = L(\mathcal{B})$, selon que le mot vide 1 est l'étiquette d'un calcul réussi dans $\mathcal{A}$ ou non. Ceci prouve que $L(\mathcal{A})$ est reconnaissable. ∎

Reprenons notre exemple de l'automate de la figure 3.1. La construction de la preuve conduit à l'automate «synchrone» de la figure 3.2.

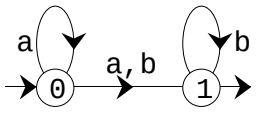


Figure 3.2: *L'automate précédent «synchronisé».*

En effet, il existe dans l'automate d'origine deux calculs, de longueur 2, de l'état 0 à l'état 1, l'un portant l'étiquette a, l'autre l'étiquette b. Notons que cet automate ne reconnaît pas le mot vide, donc ne reconnaît le langage a^*b^* qu'au mot vide près.

9.3.3 Produit et étoile

Proposition 3.4. *Si X est reconnaissable, alors X^* est reconnaissable; si X et Y sont reconnaissables, alors XY est reconnaissable.*

Preuve. Soit X un langage reconnaissable. Comme $X^* = (X-1)^*$, et que $X-1$ est reconnaissable, on peut supposer que X ne contient pas le mot vide. Soit $\mathcal{A} = (Q, I, T, \mathcal{F})$ un automate fini reconnaissant X, et soit $\mathcal{B} = (Q, I, T)$ l'automate asynchrone ayant pour flèches

$$\mathcal{F} \cup (T \times \{1\} \times I)$$

Notons que $T \cap I = \emptyset$ parce que le mot vide n'est pas dans X.

Montrons que l'on a $X^+ = L(\mathcal{B})$. Il est clair en effet que $X^+ \subset L(\mathcal{B})$. Réciproquement, soit $c : i \xrightarrow{w} t$ un calcul réussi dans $\mathcal{B}$. Ce calcul a la forme

$$c : i_1 \xrightarrow{w_1} t_1 \xrightarrow{1} i_2 \xrightarrow{w_2} t_2 \xrightarrow{1} \cdots i_n \xrightarrow{w_n} t_n$$

avec $i = i_1$, $t = t_n$, et où aucun des calculs $c_k : i_k \xrightarrow{w_k} t_k$ ne contient de flèche étiquetée par le mot vide. Alors $w_1, w_2, \ldots, w_n \in X$, et donc $w \in X^+$. Il en résulte évidemment que $X^* = X^+ \cup 1$ est reconnaissable.

Considérons maintenant deux automates finis $\mathcal{A} = (Q, I, T, \mathcal{F})$ et $\mathcal{B} = (P, J, R, \mathcal{G})$ reconnaissant respectivement les langages X et Y. On peut supposer les ensembles d'états Q et P disjoints. Soit alors $\mathcal{C} = (Q \cup P, I, R)$ l'automate dont les flèches sont

$$\mathcal{F} \cup \mathcal{G} \cup (T \times \{1\} \times J)$$

Alors on vérifie facilement que $L(\mathcal{C}) = XY$. ∎

9.4 Langages rationnels

Dans cette section, nous définissons une autre famille de langages, les langages rationnels, et nous prouvons qu'ils coïncident avec les langages reconnaissables. Grâce à ce résultat, dû à Kleene, on dispose de deux caractérisations très différentes d'une même famille de langages.

9.4.1 Langages rationnels : définitions

Soit A un alphabet. Les *opérations rationnelles* sur les parties de A^* sont les opérations suivantes :

union	$X \cup Y$;
produit	$XY = \{xy \mid x \in X,\ y \in Y\}$;
étoile	$X^* =$ le sous-monoïde engendré par X.

Une famille de parties de A^* est *rationnellement fermée* si elle est fermée pour les trois opérations rationnelles. Les *langages rationnels*indexrationnel, langage de A^* sont les éléments de la plus petite famille rationnellement fermée de A^* qui contient les singletons (c'est-à-dire les langages réduits à un seul mot) et le langage vide. Cette famille est notée $\mathrm{Rat}(A^*)$. Une expression d'un langage comme combinaison finie d'unions, de produits et d'étoiles de singletons est une *expression rationnelle*. Une étude plus formelle des expressions rationnelles est entreprise au paragraphe 3 de cette section. Considérons quelques exemples.

Le langage abA^*, avec $A = \{a, b\}$, est rationnel : il est le produit des singletons a et b par l'étoile de A qui est lui-même l'union des deux lettres a et b. De même, le langage A^*aba est rationnel. Toujours sur $A = \{a, b\}$, le langage L des mots qui contiennent un nombre pair de a est rationnel : c'est le langage $L = (ab^*a \cup b)^*$.

9.4.2 Le théorème de Kleene

Le théorème suivant est dû à Kleene :

Théorème 4.1 (de Kleene)**.** *Soit A un alphabet fini. Alors les langages rationnels et reconnaissables sur A coïncident :* $\mathrm{Rat}(A^*) = \mathrm{Rec}(A^*)$.

Cet énoncé est remarquable dans la mesure où il donne deux caractérisations très différentes d'une même famille de langages : l'automate fini est un moyen de calcul, et se donner un langage par un automate fini revient à se donner un algorithme pour vérifier l'appartenance de mots au langage. Au contraire, une expression rationnelle décrit la structure syntaxique du langage. Elle permet en particulier des manipulations, et des opérations entre langages.

La démonstration du théorème de Kleene est en deux parties. Une première partie consiste à montrer que tout langage rationnel est reconnaissable, c'est-à-dire à prouver l'inclusion $\mathrm{Rat}(A^*) \subset \mathrm{Rec}(A^*)$. On l'obtient en montrant que la famille $\mathrm{Rec}(A^*)$ est fermée pour les opérations rationnelles. L'inclusion opposée $\mathrm{Rec}(A^*) \subset \mathrm{Rat}(A^*)$ se montre en exhibant, pour tout automate fini, une expression rationnelle (au sens du paragraphe suivant) du langage qu'il reconnaît. Nous commençons par la proposition suivante :

Proposition 4.2. *Soit A un alphabet. Tout langage rationnel de A^* est reconnaissable :* $\mathrm{Rat}(A^*) \subset \mathrm{Rec}(A^*)$.

Preuve. La famille des langages sur A reconnus par des automates finis est fermée par union, produit et étoile. Par ailleurs, elle contient les singletons ; en effet, pour tout mot $w = a_1 a_2 \cdots a_n$ de longueur n, l'automate de la figure 4.1 reconnaît ce mot. Ainsi, $\mathrm{Rec}(A^*)$ est fermée pour les opérations rationnelles. Ceci montre que tout langage rationnel sur A est reconnaissable. ∎

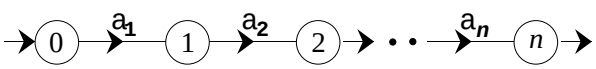


Figure 4.1: *Automate reconnaissant exactement le mot* $w = a_1 a_2 \cdots a_n$.

Proposition 4.3. *Soit A un alphabet fini. Tout langage reconnaissable de A^* est rationnel :* $\mathrm{Rec}(A^*) \subset \mathrm{Rat}(A^*)$.

Preuve. Soit $\mathcal{A} = (Q, I, T)$ un automate fini sur A, et soit X le langage reconnu. Numérotons les états de manière que $Q = \{1, \ldots, n\}$. Pour i, j dans Q, soit $X_{i,j} = \{w \mid i \xrightarrow{w} j\}$, et pour $k = 0, \ldots, n$, soit $X_{i,j}^{(k)}$ l'ensemble des étiquettes des calculs de longueur strictement positive de la forme

$$i \to p_1 \to \ldots \to p_s \to j, \qquad s \geq 0, \quad p_1, \ldots, p_s \leq k$$

Observons que $X_{i,j}^{(0)} \subset A$, et donc que chaque $X_{i,j}^{(0)}$ est une partie rationnelle, parce que l'alphabet est fini. Ensuite, on a

$$X_{i,j}^{(k+1)} = X_{i,j}^{(k)} \cup X_{i,k+1}^{(k)} \left(X_{k+1,k+1}^{(k)}\right)^* X_{k+1,j}^{(k)} \tag{4.1}$$

Cette formule montre, par récurrence sur k, que chacun des langages $X_{i,j}^{(k)}$ est rationnel. Or

$$X_{i,j} = \begin{cases} X_{i,j}^{(n)} & \text{si } i \neq j \\ 1 \cup X_{i,j}^{(n)} & \text{si } i = j \end{cases} \tag{4.2}$$

et

$$X = \bigcup_{\substack{i \in I \\ t \in T}} X_{i,t} \tag{4.3}$$

et par conséquent les langages $X_{i,j}$ sont rationnels. Donc X est également rationnel. ∎

Les formules (4.1)–(4.3) permettent de calculer effectivement une expression (expression rationnelle) pour le langage reconnu par un automate fini. Cette méthode est appelée l'algorithme de MacNaughton et Yamada, d'après ses créateurs. Il est très simple, mais pas très efficace dans la mesure où il faut calculer les $O(n^3)$ expressions $X_{i,j}^{(k)}$ pour un automate à n états. Une façon parfois plus commode — surtout pour les calculs à la main — de déterminer l'expression rationnelle du langage reconnu par un automate consiste à résoudre un système d'équations linéaires naturellement associé à tout automate fini.

Soit $\mathcal{A} = (Q, I, T, \mathcal{F})$ un automate fini sur A. Le *système d'équations* (linéaire droit) associé à $\mathcal{A}$ est

$$X_p = \bigcup_{q \in Q} E_{p,q} X_q \cup \delta_{p,T} \qquad p \in Q$$

où

$$E_{p,q} = \{a \in A \mid (p,a,q) \in \mathcal{F}\}, \qquad p,q \in Q$$
$$\delta_{p,T} = \begin{cases} \{1\} & \text{si } p \in T \\ \emptyset & \text{sinon} \end{cases}$$

Considérons par exemple l'automate de la figure 4.2.

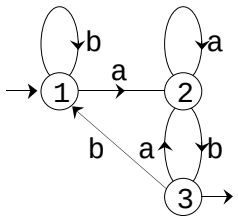


Figure 4.2: *Quel est le langage reconnu par cet automate?*

Le système d'équations associé à cet automate est

$$\begin{aligned} X_1 &= bX_1 \cup aX_2 \\ X_2 &= \qquad aX_2 \cup bX_3 \\ X_3 &= bX_1 \cup aX_2 \qquad \cup 1 \end{aligned}$$

Une *solution* du système d'équations est une famille $(L_p)_{p\in Q}$ de langages qui vérifie le système d'équations.

Proposition 4.4. *Soit $\mathcal{A} = (Q, I, T)$ un automate fini sur A, et posons*

$$L_p(\mathcal{A}) = \{w \in A^* \mid \exists c : p \to t \in T,\ |c| = w\}$$

Alors la famille $(L_p(\mathcal{A}))_{p\in Q}$ est l'unique solution du système d'équations associé à $\mathcal{A}$.

Le langage $L_p(\mathcal{A})$ est l'ensemble des mots reconnus par $\mathcal{A}$ en prenant p pour état initial, de sorte que

$$L(\mathcal{A}) = \bigcup_{p\in I} L_p(\mathcal{A})$$

Nous commençons par le cas particulier d'un système d'équations réduit à une seule équation. La démarche sera la même dans le cas général.

Lemme 4.5 (Lemme d'Arden)**.** *Soient E et F deux langages. Si $1 \notin E$, alors l'équation $X = EX \cup F$ possède une solution unique, à savoir le langage E^*F.*

Preuve. Posons $L = E^*F$. Alors $EL \cup F = EE^*F \cup F = (EE^* \cup 1)F = E^*F = L$, ce qui montre que L est bien solution de l'équation. Supposons qu'il y ait deux solutions L et L'. Comme

$$L - L' = (EL \cup F) - L' = EL - L' \subset EL - EL'$$

et que $EL - EL' \subset E(L - L')$, on a $L - L' \subset E(L - L') \subset E^n(L - L')$ pour tout $n > 0$. Mais comme le mot vide n'appartient pas à E, cela signifie qu'un mot de $L - L'$ a pour longueur au moins n pour tout n. Donc $L - L'$ est vide, et par conséquent $L \subset L'$ et de même $L' \subset L$. ∎

Preuve de la proposition. Pour montrer que les langages $L_p(\mathcal{A})$, $(p \in Q)$, constituent une solution, posons

$$L'_p = \bigcup_{q \in Q} E_{p,q} L_q(\mathcal{A}) \cup \delta_{p,T}$$

et vérifions que $L_p(\mathcal{A}) = L'_p$ pour $p \in Q$. Si $1 \in L_p(\mathcal{A})$, alors $p \in T$, donc $\delta_{p,T} = \{1\}$ et $1 \in L'_p$, et réciproquement. Soit $w \in L_p(\mathcal{A})$ de longueur > 0; il existe un calcul $c : p \to t$ pour un $t \in T$ d'étiquette w. En posant $w = av$, avec a une lettre et v un mot, le calcul c se factorise en $p \xrightarrow{a} q \xrightarrow{v} t$ pour un état q. Mais alors $a \in E_{p,q}$ et $v \in L_q(\mathcal{A})$, donc $w \in L'_p$. L'inclusion réciproque se montre de la même façon.

Pour prouver l'unicité de la solution, on procède comme dans la preuve du lemme d'Arden. Soient $(L_p)_{p \in Q}$ et $(L'_p)_{p \in Q}$ deux solutions du système d'équations, et posons $M_p = L_p - L'_p$ pour $p \in Q$. Alors

$$M_p = \bigcup_{q \in Q} E_{p,q} L_q \cup \delta_{p,T} - L'_p \subset \bigcup_{q \in Q} (E_{p,q} L_q - E_{p,q} L'_q) \subset \bigcup_{q \in Q} E_{p,q} M_q$$

A nouveau, ces inclusions impliquent que $M_p = \emptyset$ pour tout p. Supposons en effet le contraire, soit w un mot de longueur minimale dans

$$M' = \bigcup_{p \in Q} M_p$$

et soit p un état tel que $w \in M_p$. Alors

$$w \in \bigcup_{q \in Q} E_{p,q} M_q$$

donc w est le produit d'une lettre et d'un mot v de M'; mais alors v est plus court que w, une contradiction. ∎

Pour *résoudre* un système d'équations, on peut procéder par élimination de variables (c'est la méthode de Gauss). Dans notre exemple, on substitue la troisième équation dans la deuxième, ce qui donne

$$X_2 = (a \cup ba) X_2 \cup b^2 X_1 \cup b$$

qui, par le lemme d'Arden, équivaut à l'équation

$$X_2 = (a \cup ba)^* (b^2 X_1 \cup b)$$

Cette expression pour X_2 est substituée dans la première équation du système; on obtient

$$X_1 = (b \cup a(a \cup ba)^* b^2) X_1 \cup a(a \cup ba)^* b$$

d'où, à nouveau par le lemme d'Arden,

$$X_1 = (b \cup a(a \cup ba)^* b^2)^* a(a \cup ba)^* b$$

Il n'est pas du tout évident que cette dernière expression soit égale à $\{a, b\}^* ab$. Pour le montrer, observons d'abord que

$$(a \cup ba)^* = a^*(ba^+)^*$$

(rappelons que $X^+ = XX^* = X^*X$ pour tout X) en utilisant la règle $(U \cup V)^* = U^*(VU^*)^*$, d'où

$$a(a \cup ba)^* b = a^+(ba^+)^* b = (a^+ b)^+$$

Il en résulte que

$$b \cup a(a \cup ba)^* b^2 = [1 \cup (a^+ b)^+]b = (a^+ b)^* b$$

d'où

$$X_1 = [(a^+ b)^* b]^* (a^+ b)^* (a^+ b) = (a^+ b \cup b)^* (a^+ b) = (a^* b)^* a^* ab = \{a, b\}^* ab$$

9.4.3 Expressions rationnelles

Le théorème de Kleene, qui établit l'égalité entre langages rationnels et reconnaissables, possède des preuves constructives. L'une est l'algorithme de McNaughton et Yamada, la deuxième revient à résoudre un système d'équations linéaires. Dans les deux cas, la rationalité du langage obtenu est évidente parce qu'elle provient d'une *expression rationnelle* fournie pour le langage; plus précisément, le langage est exprimé à l'aide des opérations rationnelles.

Dans ce paragraphe, nous étudions les expressions rationnelles en elles-mêmes, en faisant une distinction entre une expression et le langage qu'elle décrit, un peu comme l'on fait la différence entre une expression arithmétique et la valeur numérique qu'elle représente. L'approche est donc purement syntaxique, et le langage représenté par une expression peut être considéré comme résultant d'une évaluation de l'expression.

Les manipulations (algébriques ou combinatoires) d'expressions permettent de construire des automates efficaces reconnaissant le langage représenté par des opérations qui sont proches de l'analyse syntaxique, et qui ne font pas intervenir le langage lui-même. Elles se prêtent donc bien à une implémentation effective. Une autre application est la recherche de motifs dans un texte, comme elle est réalisée dans un traitement de texte par exemple, et comme elle sera traitée au chapitre suivant.

Soit A un alphabet fini, soient 0 et 1 deux symboles ne figurant pas dans A, et soient $+, \cdot, ^*$ trois symboles de fonctions, les deux premiers binaires, le dernier unaire. Les *expressions* sur $A \cup \{0, 1, +, \cdot, ^*\}$ sont les termes bien formés sur cet ensemble de symboles.

Exemple. Sur $A = \{a, b\}$ le terme $a(a + a \cdot b)^* b$ est une expression. Comme souvent, on peut représenter une expression par un arbre qui, sur cet exemple, est l'arbre de la figure 4.3.

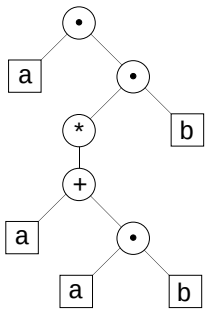


Figure 4.3: *L'expression rationnelle* $a(a + a \cdot b)^* b$.

Les parenthèses ne sont utilisées que lorsqu'elles sont nécessaires, en convenant que l'opération * a priorité sur $\cdot$ qui a priorité sur $+$; on omet également le signe de multiplication quand cela ne prête pas à confusion.

Les *expressions rationnelles* sur A sont les éléments de l'algèbre quotient obtenue en convenant que

(i) l'opération $+$ est idempotente, associative et commutative,

(ii) l'opération $\cdot$ est associative et distributive par rapport à l'opération $+$,

(iii) les équations suivantes sont vérifiées pour tout terme e :

$$\begin{aligned} 0 + e &= e = e + 0 \\ 1 \cdot e &= e = e \cdot 1 \\ 0 \cdot e &= 0 = e \cdot 0 \end{aligned}$$

(iv) on a :

$$0^* = 1^* = 1$$

Exemple. Par exemple, $(0 + a(1 + b \cdot 1))^*$ et $(ab + a)^*$ sont la *même* expression rationnelle, alors que $1 + a^* a$ et a^* sont deux expressions différentes (même si elles décrivent le même langage).

On note $\mathcal{E}(A)$ l'algèbre des expressions rationnelles ainsi obtenue. La convention d'identifier certaines expressions selon les règles que nous venons d'édicter allège très considérablement l'écriture et l'exposé; tout ce qui suit pourrait se faire également en ne raisonnant que sur les termes ou les arbres.

L'important est toutefois que l'*égalité* de deux expressions est *facilement décidable* sur les expressions elles-mêmes. Pour ce faire, on met une expression, donnée disons sous forme d'arbre, en «forme normale» en supprimant d'abord les feuilles 0 ou 1 quand c'est possible en appliquant les égalités (iii) ci-dessus, puis en distribuant le produit par rapport à l'addition pour faire «descendre» au maximum les produits. Lorsque les deux expressions sont en forme normale, on peut décider récursivement si elles sont égales : il faut et il suffit pour cela qu'elles aient même symbole aux racines, et si les *suites* d'expressions des fils obtenues en faisant jouer l'associativité de $\cdot$ et les *ensembles* d'expressions des fils modulo l'associativité, l'idempotence et la commutativité de $+$ sont égaux.

Exemple. En appliquant les règles de simplification à l'expression $(0 + a(1 + b \cdot 1))^*$, on obtient $(a(1+b))^*$. Par distributivité et simplification, elle donne $(a+ab)^*$, et les deux expressions fils de la racine sont les mêmes dans cette expression, et dans $(ab + a)^*$.

Remarque. Dans ce qui précède ne figure pas l'opération $e \mapsto e^+$. C'est pour ne pas alourdir l'exposé que nous considérons cette opération comme une abréviation de ee^* ; on pourrait aussi bien considérer cette opération comme opération de base.

Il y a une application naturelle de l'algèbre des expressions rationnelles sur A dans les langages rationnels sur A. C'est l'application

$$L : \mathcal{E}(A) \to \wp(A^*)$$

définie par récurrence comme suit :

$$\begin{array}{l} L(0) = \emptyset, \quad L(1) = \{1\}, \quad L(a) = \{a\}, \\ L(e + e') = L(e) \cup L(e'), \quad L(e \cdot e') = L(e)L(e'), \\ L(e^*) = L(e)^* \end{array}$$

Pour une expression e, le langage $L(e)$ est le langage *décrit* ou *dénoté* par e. Deux expressions e et e' sont dites *équivalentes* lorsqu'elles dénotent le même langage, c'est-à-dire lorsque $L(e) = L(e')$. On écrit alors $e \approx e'$.

Proposition 4.6. *Pour toutes expressions rationnelles e et f, on a les formules suivantes :*

$$\begin{aligned} (ef)^* &\approx 1 + e(fe)^* f \\ (e + f)^* &\approx e^*(fe^*)^* \\ e^* &\approx (1 + e + \ldots + e^{p-1})(e^p)^* \qquad p > 1 \end{aligned}$$

Cette proposition est un corollaire immédiat du lemme suivant :

Lemme 4.7. *Quelles que soient les parties X et Y de A^*, on a*

$$(XY)^* = 1 \cup X(YX)^*Y \tag{4.4}$$
$$(X \cup Y)^* = X^*(YX^*)^* \tag{4.5}$$
$$X^* = (1 \cup X \cup \cdots \cup X^{p-1})(X^p)^* \qquad p > 1 \tag{4.6}$$

Preuve. Pour établir (4.4), montrons d'abord que $(XY)^* \subset 1 \cup X(YX)^*Y$. Pour cela, soit $w \in (XY)^*$. Alors $w = x_1y_1 \cdots x_ny_n$ pour $n \geq 0$ et $x_i \in X$, $y_i \in Y$. Si $n = 0$, alors $w = 1$, sinon $w = x_1vy_n$, avec $v = y_1 \cdots x_n \in (YX)^*$. Ceci prouve l'inclusion. L'inclusion réciproque se montre de manière similaire.

Pour prouver (4.5), il suffit d'établir

$$(X \cup Y)^* \subset 1 \cup X^*(YX^*)^*$$

l'inclusion réciproque étant évidente. Soit $w \in (X \cup Y)^*$. Il existe $n \geq 0$ et $z_1, \ldots, z_n \in X \cup Y$ tels que $w = z_1 \cdots z_n$. En groupant les z_i consécutifs qui appartiennent à X, ce produit peut s'écrire

$$w = x_0y_1x_1 \cdots y_mx_m$$

avec $x_0, \ldots, x_m \in X^*$ et $y_0, \ldots, y_m \in Y$. Par conséquent $w \in 1 \cup X^*(YX^*)^*$.

Considérons enfin (4.6). On a

$$(1 \cup X \cup \cdots \cup X^{p-1})(X^p)^* = \bigcup_{k=0}^{p-1} \bigcup_{m \geq 0} X^{k+mp} = X^* \qquad \blacksquare$$

9.5 Automate minimal

Déterminer un automate ayant un nombre minimum d'états pour un langage rationnel donné est très intéressant du point de vue pratique. Il est tout à fait remarquable que tout langage rationnel possède un automate déterministe minimal unique, à une numérotation des états près. Ce résultat n'est plus vrai si l'on considère des automates qui ne sont pas nécessairement déterministes.

Il y a deux façons de définir l'automate déterministe minimal reconnaissant un langage rationnel donné. La première est intrinsèque; elle est définie à partir du langage par une opération appelée le *quotient*. La deuxième est plus opératoire; on part d'un automate déterministe donné, et on le réduit en identifiant des états appelés *inséparables*. Les algorithmes de minimisation utilisent la deuxième définition.

9.5.1 Quotients

Soit A un alphabet. Pour deux mots u et v, on pose

$$u^{-1}v = \{w \in A^* \mid uw = v\}, \qquad uv^{-1} = \{w \in A^* \mid u = wv\}$$

Un tel ensemble, appelé *quotient gauche* respectivement *droit* est, bien entendu, soit vide soit réduit à un seul mot qui, dans le premier cas est un suffixe de v, et dans le deuxième cas un préfixe de u.

La notation est étendue aux parties en posant, pour $X, Y \subset A^*$

$$X^{-1}Y = \bigcup_{x\in X}\bigcup_{y\in Y} x^{-1}y, \qquad XY^{-1} = \bigcup_{x\in X}\bigcup_{y\in Y} xy^{-1}$$

Les quotients sont un outil important pour l'étude des automates finis et des langages rationnels. On utilise principalement les quotients gauches par un mot, c'est-à-dire les ensembles

$$u^{-1}X = \{w \in A^* \mid uw \in X\}$$

Notons qu'en particulier, $1^{-1}X = X$ pour tout ensemble X, et que $(uv)^{-1}X = v^{-1}(u^{-1})X$. Pour toute partie X de A^*, on pose

$$Q(X) = \{u^{-1}X \mid u \in A^*\} \tag{5.1}$$

Exemple. Sur $A = \{a, b\}$, soit X l'ensemble des mots qui contiennent au moins une fois la lettre a. Alors on a :

$$1^{-1}X = X, \quad a^{-1}X = A^*, \quad b^{-1}X = X, \quad a^{-1}A^* = b^{-1}A^* = A^*$$

donc $Q(X) = \{X, A^*\}$.

Proposition 5.1. *Soit $X \subset A^*$ le langage reconnu par un automate fini déterministe, accessible et complet $\mathcal{A} = (Q, i, T)$; pour $q \in Q$, soit $L_q(\mathcal{A}) = \{w \in A^* \mid q \cdot w \in T\}$. Alors*

$$\{L_q(\mathcal{A}) \mid q \in Q\} = Q(X) \tag{5.2}$$

Preuve. Montrons d'abord que $Q(X)$ est contenu dans $\{L_q(\mathcal{A}) \mid q \in Q\}$. Soit $u \in A^*$, et soit $q = i \cdot u$ (cet état existe parce que $\mathcal{A}$ est complet). Alors $u^{-1}X = L_q(\mathcal{A})$, puisque

$$w \in u^{-1}X \iff uw \in X \iff i \cdot uw \in T \iff q \cdot w \in T \iff w \in L_q(\mathcal{A})$$

Pour montrer l'inclusion réciproque, soit $q \in Q$ et soit $u \in A^*$ tel que $q = i \cdot u$ (un tel mot existe parce que l'automate est accessible). Alors $L_q(\mathcal{A}) = u^{-1}X$. Ceci prouve la proposition. ∎

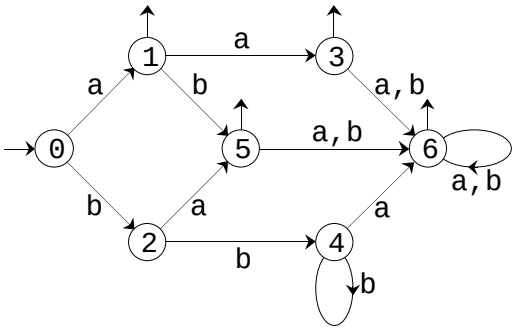


Figure 5.1: *Un automate reconnaissant* $X = b^*a\{a,b\}^*$.

Exemple. Considérons l'automate $\mathcal{A}$ donné dans la figure 5.1. Un calcul rapide montre que

$$L_1(\mathcal{A}) = L_3(\mathcal{A}) = L_5(\mathcal{A}) = L_6(\mathcal{A}) = \{a,b\}^*$$
$$L_0(\mathcal{A}) = L_2(\mathcal{A}) = L_4(\mathcal{A}) = b^*a\{a,b\}^*$$

L'équation (5.2) est bien vérifiée.

On déduit de cette proposition que, pour un langage reconnaissable X, l'ensemble des quotients gauches $Q(X)$ est fini; nous allons voir dans un instant que la réciproque est vraie également. L'équation (5.2) montre par ailleurs que tout automate déterministe, accessible et complet reconnaissant X possède au moins $|Q(X)|$ états. Nous allons voir que ce minimum est atteint, et même, au paragraphe suivant, qu'il existe un automate unique, à un isomorphisme près, qui a $|Q(X)|$ états et qui reconnaît X.

Soit $X \subset A^*$. On appelle *automate minimal* de X l'automate déterministe

$$\mathcal{A}(X) = (Q(X), X, T(X))$$

dont l'ensemble d'états est donné par (5.1), ayant X comme état initial, l'ensemble

$$T(X) = \{u^{-1}X \mid u \in X\} = \{u^{-1}X \mid 1 \in u^{-1}X\}$$

comme ensemble d'états terminaux, la fonction de transition étant définie, pour $Y \in Q$ et $a \in A$ par

$$Y \cdot a = a^{-1}Y$$

Notons que si $Y = u^{-1}X$, alors $Y \cdot a = a^{-1}(u^{-1}X) = (ua)^{-1}X$, donc la fonction de transition est bien définie, et l'automate est complet.

Exemple. L'automate $\mathcal{A}(X)$ pour le langage $X = b^*a\{a,b\}^*$ a les deux états X et $\{a,b\}^*$, le premier est initial, le deuxième est final. L'automate est donné dans la figure 5.2.

Proposition 5.2. *Le langage reconnu par l'automate* $\mathcal{A}(X)$ *est* X.

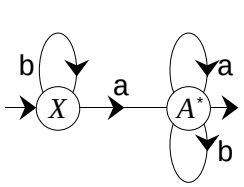


Figure 5.2: *Automate minimal pour* $X = b^*a\{a,b\}^*$

Preuve. Montrons d'abord que, pour tout $w \in A^*$ et pour tout $Y \in Q(X)$, on a $Y \cdot w = w^{-1}Y$. En effet, ceci est vrai si w est une lettre ou le mot vide. Si $w = ua$, avec a une lettre, alors $Y \cdot ua = (Y \cdot u) \cdot a = (u^{-1}Y) \cdot a = a^{-1}(u^{-1}Y) = (ua)^{-1}Y$. Ceci prouve la formule. Il en résulte que

$$w \in L(\mathcal{A}(X)) \iff X \cdot w \in T \iff w^{-1}X \in T \iff w \in X$$

Donc $\mathcal{A}(X)$ reconnaît X. ∎

Corollaire 5.3. *Une partie $X \subset A^*$ est reconnaissable si et seulement si l'ensemble $Q(X)$ est fini.*

Preuve. Si X est reconnaissable, alors (5.2) montre que $Q(X)$ est fini. La réciproque découle de la proposition précédente. ∎

On peut calculer l'ensemble $Q(X)$ pour un langage rationnel, à partir d'une expression rationnelle pour X, à l'aide des formules de la proposition suivante. Cela ne résout pas complètement le problème du calcul de l'automate minimal, parce qu'une difficulté majeure demeure, à savoir tester si deux expressions sont équivalentes. On en parlera plus loin.

Proposition 5.4. *Soit a une lettre, et soient X et Y des langages. Alors on a*

$$\begin{aligned} a^{-1}\emptyset &= a^{-1}1 = \emptyset \\ a^{-1}a &= 1 \\ a^{-1}b &= \emptyset \quad (b \neq a) \\ a^{-1}(X \cup Y) &= a^{-1}X \cup a^{-1}Y \\ a^{-1}(XY) &= (a^{-1}X)Y \cup (X \cap 1)a^{-1}Y \qquad (5.3) \\ a^{-1}X^* &= (a^{-1}X)X^* \qquad (5.4) \end{aligned}$$

Preuve. Seules les deux dernières formules demandent une vérification. Si $w \in a^{-1}(XY)$, alors $aw \in XY$; il existe donc $x \in X$, $y \in Y$ tels que $aw = xy$. Si $x = 1$, alors $aw = y$, donc $w \in (X \cap 1)a^{-1}Y$; sinon, $x = au$ pour un préfixe u de w, et $u \in a^{-1}X$, d'où $w \in (a^{-1}X)Y$. L'inclusion réciproque se montre de la même manière.

Considérons la dernière formule. Soit $w \in a^{-1}X^*$. Alors $aw \in X^*$, donc $aw = xx'$, avec $x \in X$, $x \neq 1$, et $x' \in X^*$. Mais alors $x = au$, avec $u \in a^{-1}X$, donc $w \in (a^{-1}X)X^*$. Réciproquement, on a par la formule (5.3), l'inclusion $(a^{-1}X)X^* \subset a^{-1}(XX^*)$, donc $(a^{-1}X)X^* \subset a^{-1}X^*$. ∎

Exemple. Calculons, à l'aide de ces formules, le langage $a^{-1}(b^*aA^*)$. On a

$$a^{-1}(b^*aA^*) = a^{-1}(b^*(aA^*)) = (a^{-1}b^*)aA^* \cup a^{-1}(aA^*)$$

par (5.3) ; or le premier terme de l'union est vide par (5.3), d'où

$$a^{-1}(b^*aA^*) = (a^{-1}a)A^* \cup (a \cap 1)a^{-1}A^* = (a^{-1}a)A^* = A^*$$

9.5.2 Equivalence de Nerode

Dans ce paragraphe, tous les automates considérés sont déterministes, accessibles et complets.

Soit $\mathcal{A} = (Q, i, T)$ un automate sur un alphabet A reconnaissant un langage X. Pour tout état q, on pose

$$L_q(\mathcal{A}) = \{w \in A^* \mid q \cdot w \in T\}$$

C'est donc l'ensemble des mots reconnus par l'automate $\mathcal{A}$ en prenant q comme état initial. Bien entendu, le langage X reconnu par $\mathcal{A}$ coïncide avec $L_i(\mathcal{A})$. On a vu, au paragraphe précédent, que

$$\{L_q(\mathcal{A}) \mid q \in Q\} = Q(X)$$

La correspondance s'établit par

$$L_q(\mathcal{A}) = u^{-1}X \qquad \text{si } i \cdot u = q$$

Notons que, plus généralement,

$$L_{q\cdot v}(\mathcal{A}) = v^{-1}L_q(\mathcal{A}) \tag{5.5}$$

En effet, $w \in L_{q\cdot v}(\mathcal{A})$ si et seulement si $(q \cdot v) \cdot w \in T$ donc si et seulement si $vw \in L_q(\mathcal{A})$. Lorsque l'automate est fixé, on écrira L_q au lieu de $L_q(\mathcal{A})$. Deux états $p, q \in Q$ sont dits *inséparables* si $L_p = L_q$, ils sont *séparables* sinon. Ainsi, p et q sont séparables si et seulement s'il existe un mot w tel que $p \cdot w \in T$ et $q \cdot w \notin T$ ou vice-versa. Si w est un mot ayant cette propriété, on dit qu'il *sépare* les états p et q. L'*équivalence de Nerode* sur Q (ou sur $\mathcal{A}$) est la relation $\sim$ définie par

$$p \sim q \iff p \text{ et } q \text{ sont inséparables}$$

Proposition 5.5. *Dans l'automate minimal, deux états distincts sont séparables, et l'équivalence de Nerode est l'égalité.*

Preuve. Soit $Y = u^{-1}X$ un état de l'automate minimal $\mathcal{A}(X)$ du langage X. Comme $Y = X \cdot u$, on a $L_Y = u^{-1}X = Y$. Par conséquent, deux états distincts ne sont pas équivalents. ∎

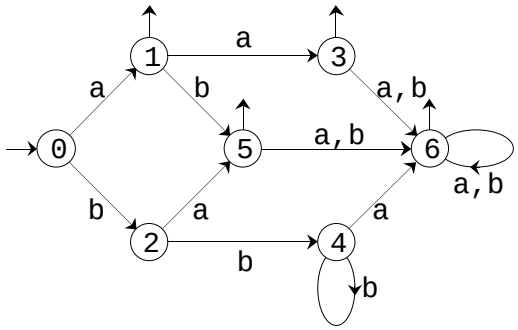


Figure 5.3: *Un automate qui n'est pas minimal.*

Exemple. Reprenons l'automate donné dans la figure 5.3 ci-dessous, et déjà considéré dans le paragraphe précédent. Puisque

$$L_1 = L_3 = L_5 = L_6 = \{a, b\}^*$$
$$L_0 = L_2 = L_4 = b^*a\{a, b\}^*$$

l'équivalence de Nerode a donc les deux classes $\{0, 2, 4\}$ et $\{1, 3, 5, 6\}$. Le mot vide sépare deux états pris dans des classes distinctes.

Proposition 5.6. *L'équivalence de Nerode est une relation d'équivalence régulière à droite, c'est-à-dire vérifiant*

$$p \sim q \Rightarrow p \cdot u \sim q \cdot u \qquad (u \in A^*)$$

De plus, une classe de l'équivalence ou bien ne contient pas d'états terminaux, ou bien ne contient que des états terminaux.

Preuve. Soit $\mathcal{A} = (Q, i, T)$ un automate. Il est clair que la relation $\sim$ est une relation d'équivalence. Pour montrer sa régularité, supposons $p \sim q$, et soit $u \in A^*$. En vue de (5.5), on a $L_{q\cdot u} = u^{-1}L_q = u^{-1}L_p = L_{p\cdot u}$, donc $p \cdot u \sim q \cdot u$. Supposons enfin $p \sim q$ et p terminal. Alors $1 \in L_p$, et comme $L_p = L_q$, on a $1 \in L_q$, donc q est terminal. ∎

L'équivalence de Nerode sur un automate $\mathcal{A} = (Q, i, T)$ étant régulière à droite, on peut définir un *automate quotient* en confondant les états d'une même classe. Plus précisément, notons $[q]$ la classe d'un état q dans l'équivalence. L'automate quotient est alors défini par

$$\mathcal{A}/\!\sim = (Q/\!\sim, [i], \{[t] : t \in T\})$$

avec la fonction de transition

$$[q \cdot a] = [q] \cdot a \tag{5.6}$$

qui justement est bien définie (indépendante du représentant choisi dans la classe de q) parce que l'équivalence est régulière à droite.

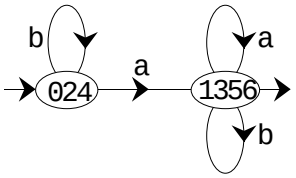


Figure 5.4: *Un automate quotient.*

Dans l'exemple ci-dessus, l'automate quotient a deux états : l' état $\{0, 2, 4\}$ est initial, et $\{1, 3, 5, 6\}$ est l'état final. La formule (5.6) permet de calculer les transitions. On obtient l'automate de la figure 5.4.

Comme le suggère cet exemple, l'automate quotient est en fait l'automate minimal, à une renumérotation des états près. Nous avons besoin, pour le prouver, de la notion d'isomorphisme d'automates. Soient $\mathcal{A} = (Q, i, T)$ et $\mathcal{A}' = (Q', i', T')$ deux automates sur A. Ils sont dits *isomorphes* s'il existe une bijection

$$\alpha : Q \to Q'$$

telle que $\alpha(i) = i'$, $\alpha(T) = T'$, et $\alpha(q \cdot a) = \alpha(q) \cdot a$ pour tout $q \in Q$ et $a \in A$.

Proposition 5.7. *Soit $\mathcal{A} = (Q, i, T)$ un automate sur un alphabet A reconnaissant un langage X. Si l'équivalence de Nerode de $\mathcal{A}$ est l'égalité, alors $\mathcal{A}$ et l'automate minimal $\mathcal{A}(X)$ sont isomorphes.*

Preuve. Soit $\alpha : Q \to Q(X)$ définie par $\alpha(q) = L_q(\mathcal{A})$. Comme l'équivalence de Nerode est l'égalité, α est une bijection. Par ailleurs, $\alpha(i) = L_i = X$, et $t \in T$ si et seulement si $1 \in L_t(\mathcal{A})$, donc si et seulement si $L_t(\mathcal{A})$ est état final de $\mathcal{A}(X)$. Enfin, on a

$$L_{q \cdot a}(\mathcal{A}) = L_q(\mathcal{A}) \cdot a$$

montrant que α est bien un morphisme. ∎

De cette propriété, on déduit une conséquence importante :

Théorème 5.8. *L'automate minimal d'un langage X est l'automate ayant le moins d'états parmi les automates déterministes complets qui reconnaissent X. Il est unique à un isomorphisme près.*

Preuve. Soit $\mathcal{B}$ un automate reconnaissant X et ayant un nombre minimal d'états. Alors son équivalence de Nerode est l'égalité, sinon on pourrait passer au quotient par son équivalence de Nerode et trouver un automate plus petit. Par la proposition précédente, $\mathcal{B}$ est isomorphe à $\mathcal{A}(X)$. ∎

L'unicité de l'automate minimal ne vaut que pour les automates déterministes. Il existe des automates non déterministes, non isomorphes, ayant un nombre minimal d'états, et reconnaissant un même langage (voir exercices).

9.6 Calcul de l'automate minimal

Le calcul de l'automate minimal peut se faire par un procédé appelé la construction de Moore et que nous exposons dans le premier paragraphe. Nous verrons dans la suite une implémentation sophistiquée de cette construction.

9.6.1 Construction de Moore

Soit $\mathcal{A} = (Q, i, T)$ un automate déterministe, accessible et complet sur un alphabet A. Pour calculer l'automate minimal, il suffit de calculer l'équivalence de Nerode définie, rappelons-le, par

$$p \sim q \iff L_p = L_q$$

où $L_p = \{w \in A^* \mid p \cdot w \in T\}$. Pour calculer cette équivalence, on procède par approximations successives. On considère pour ce faire l'équivalence suivante, où k est un entier naturel :

$$p \sim_k q \iff L_p^{(k)} = L_q^{(k)}$$

avec

$$L_p^{(k)} = \{w \in L_p \mid |w| \leq k\}$$

L'équivalence de Nerode est l'intersection de ces équivalences. La proposition suivante exprime l'équivalence $\sim_k$ au moyen de l'équivalence $\sim_{k-1}$. Elle permettra de prouver que l'on obtient bien «à la limite» l'équivalence de Nerode, et elle donne aussi un procédé de calcul.

Proposition 6.1. *Pour tout entier $k \geq 1$, on a*

$$p \sim_k q \iff p \sim_{k-1} q \quad \text{et} \quad (\forall\, a \in A, \quad p \cdot a \sim_{k-1} q \cdot a)$$

Preuve. On a

$$\begin{aligned} L_p^{(k)} &= \{w \mid p \cdot w \in T \text{ et } |w| \leq k\} \\ &= \{w \mid p \cdot w \in T \text{ et } |w| \leq k-1\} \\ &\qquad \cup \bigcup_{a \in A} a\{v \mid (p \cdot a) \cdot v \in T \text{ et } |v| \leq k-1\} \\ &= L_p^{(k-1)} \cup \bigcup_{a \in A} a L_{p \cdot a}^{(k-1)} \end{aligned}$$

L'observation n'est qu'une traduction de ces égalités. ∎

Corollaire 6.2. *Si les équivalences $\sim_k$ et $\sim_{k+1}$ coïncident, les équivalences $\sim_{k+l}$ ($l \geq 0$) sont toutes égales, et égales à l'équivalence de Nerode.*

Preuve. De la proposition, il résulte immédiatement que l'égalité des équivalences $\sim_k$ et $\sim_{k+1}$ entraîne celle des équivalences $\sim_{k+1}$ et $\sim_{k+2}$. D'où la première assertion. Comme $p \sim q$ si et seulement si $p \sim_k q$ pour tout $k \geq 0$, la deuxième assertion s'en déduit. ∎

Proposition 6.3. *Si $\mathcal{A}$ est un automate à n états, l'équivalence de Nerode de $\mathcal{A}$ est égale à $\sim_{n-2}$.*

Preuve. Si, pour un entier $k > 0$, les équivalences $\sim_{k-1}$ et $\sim_k$ sont distinctes, le nombre de classes de l'équivalence $\sim_k$ est $\leq k + 2$. ∎

9.6.2 Scinder une partition

La construction de l'automate minimal par la méthode de Moore appliquée directement à un automate à n états sur un alphabet à m lettres fournit un algorithme qui a un temps de calcul dans le pire des cas en $O(mn^2)$. Nous présentons ici un algorithme dû à Hopcroft, dont la preuve a été simplifiée par D. Gries, et dont la complexité en temps est $O(mn \log n)$.

Soient $\mathcal{R}$ et $\mathcal{R}'$ deux relations d'équivalence sur un ensemble Q. On dit que $\mathcal{R}$ *est plus fine que* $\mathcal{R}'$, ce que l'on note $\mathcal{R} \leq \mathcal{R}'$, si : $p\mathcal{R}q \Rightarrow p\mathcal{R}'q$.

Soient $\mathcal{P}$ et $\mathcal{P}'$ les partitions associées à $\mathcal{R}$ et $\mathcal{R}'$, on a alors :

$$\mathcal{R} \leq \mathcal{R}' \text{ si et seulement si } \forall P \in \mathcal{P},\ \exists P' \in \mathcal{P}',\ P \subset P'.$$

Soit A un ensemble opérant à droite sur Q par une application de $Q \times A$ dans Q où l'image du couple (q, a) est notée $q \cdot a$ (c'est le cas de la fonction de transition d'un automate déterministe complet). On note alors $a^{-1}S = \{q \in Q \mid q \cdot a \in S\}$. Pour toute partie S de Q, on note $\bar{S}$ la partie $Q - S$. Remarquons que $Q = a^{-1}S \cup a^{-1}\bar{S}$, et que cette union est disjointe.

On peut alors établir un certain nombre de propriétés :

Soient $\mathcal{P}$ une partition de Q, P et S deux éléments de $\mathcal{P}$ et a un élément de A.

On dit que P est *stable pour* (S, a) si on a :

$$P \cdot a \subset S \text{ ou } P \cdot a \subset \bar{S}$$

ou encore de manière équivalente si :

$$P \subset a^{-1}S \text{ ou } P \subset a^{-1}\bar{S}$$

Si P n'est pas stable pour (S, a), on dit que P est *brisée* par (S, a) ; cela signifie que les ensembles

$$P \cap a^{-1}S \text{ et } P \cap a^{-1}\bar{S}$$

sont tous les deux non vides. On définit une opération SCINDER de la manière suivante : SCINDER$(P,(S,a))$ encore noté $P \underset{a}{\lhd} S$ est défini par :

$$P \underset{a}{\lhd} S = \begin{cases} \{P\} & \text{si } P \text{ est stable pour } (S,a) \\ \{P \cap a^{-1}S, P \cap a^{-1}\bar{S}\} & \text{sinon} \end{cases}$$

On peut noter aussi que l'on a : $P \underset{a}{\lhd} S = P \underset{a}{\lhd} \bar{S}$.

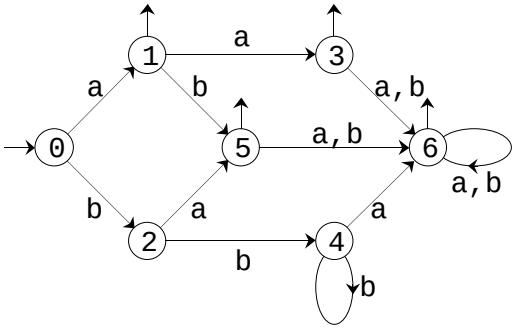


Figure 6.1: *Un exemple illustrant l'opération* $\lhd$.

Exemple. Considérons l'automate déterministe complet de la figure 6.1. La partie Q n'est pas stable pour (T,b) car $Q \cdot b = \{2,4,5,6\}$; or $2 \notin T$, et $5 \in T$. On a alors :

$$Q \underset{b}{\lhd} T = \{\{1,3,5,6\},\{0,2,4\}\}.$$

Par contre $\bar{T}$ est stable pour (T,a) et pour (T,b), car $\bar{T} \cdot a \subset T$ et $\bar{T} \cdot b \subset \bar{T}$.

Lemme 6.4. *Soit A un ensemble opérant à droite sur Q, soient P,S,T des parties de Q et $a,b \in A$.*

i) l'opération SCINDER *est «commutative», i.e. :*

$$P \underset{a}{\lhd} S \underset{b}{\lhd} T = P \underset{b}{\lhd} T \underset{a}{\lhd} S$$

ii) l'opération SCINDER *est «monotone» , i.e. si $\{S_1,S_2\}$ est une partition de S alors :*

$$P \underset{a}{\lhd} S \underset{a}{\lhd} S_1 = P \underset{a}{\lhd} S_2 \underset{a}{\lhd} S_1.$$

Preuve. i) $P \underset{a}{\lhd} S$ est formé des ensembles non vides parmi $P \cap a^{-1}S$, $P \cap a^{-1}\bar{S}$, et $P \underset{a}{\lhd} S \underset{b}{\lhd} T$ est composé des ensembles non vides parmi :

$$P \cap a^{-1}S \cap b^{-1}T, \quad P \cap a^{-1}\bar{S} \cap b^{-1}T, \quad P \cap a^{-1}S \cap b^{-1}\bar{T}, \quad P \cap a^{-1}\bar{S} \cap b^{-1}\bar{T}.$$

On constate aisément que la permutation de S et T et de a et b donne le même résultat.

ii) Comme l'opération SCINDER est commutative, il suffit de comparer $P \lhd_a S_1 \lhd_a S$ à $P \lhd_a S_1 \lhd_a S_2$. Il est facile de vérifier que les opérations $\bullet \lhd_a S_2$ et $\bullet \lhd_a S$ ne modifient pas la partition $P \lhd_a S_1$, d'où le résultat cherché. ∎

Une partition $\mathcal{P}$ est *stable pour* (S, a) si les classes qui la composent le sont. On définit SCINDER$(\mathcal{P}, (S, a))$ que l'on note aussi $\mathcal{P} \lhd_a S$ comme étant la partition $\mathcal{P}'$ obtenue en remplaçant dans $\mathcal{P}$ chaque classe P par SCINDER$(P, (S, a))$.

L'opération SCINDER agissant sur une partition a les propriétés suivantes :

Lemme 6.5. *i) Si une partition $\mathcal{P}$ de Q est stable pour (S, a), alors toute partition plus fine que $\mathcal{P}$ l'est également.*

ii) La partition $\mathcal{P}' = \mathcal{P} \lhd_a S$ est stable pour (S, a), plus fine que $\mathcal{P}$, et si $\mathcal{P}$ n'est pas stable pour (S, a), alors $\mathcal{P}'$ est strictement plus fine que $\mathcal{P}$.

Preuve. Les preuves sont faciles et laissées au lecteur à titre d'exercice. ∎

Remarques.

– Il est facile de vérifier que les propriétés énoncées au lemme 6.4 restent vraies si l'on remplace P par une partition $\mathcal{P}$.

– La notation $P \lhd_a S \lhd_b T$ peut s'étendre à une liste quelconque $\mathcal{L} = ((S_1, a_1), \ldots, ((S_k, a_k))$ de couples, et on écrira alors $P \lhd \mathcal{L}$ ou bien SCINDER$(P, \mathcal{L})$.

Notons que la régularité à droite d'une relation d'équivalence sur Q peut s'énoncer en termes de stabilité de la façon suivante :

Lemme 6.6. *Une relation d'équivalence sur Q est régulière à droite si et seulement si la partition $\mathcal{P}$ associée est stable pour tous les couples (P, a) où $P \in \mathcal{P}$ et $a \in A$.* ∎

9.6.3 Algorithme de Hopcroft

Revenons à l'équivalence de Nerode. Soit $\mathcal{A} = (Q, i, T)$ un automate déterministe complet à n états sur un alphabet A à m lettres. Rappelons que l'équivalence de Nerode sur Q est définie par :

$$p \sim q \Longleftrightarrow L_p = L_q$$

où L_p est l'ensemble des étiquettes des chemins de i à p.

Note 9.6.2

On appelle *partition de Nerode* la partition de Q associée à cette relation d'équivalence. On a vu (proposition 5.6) que l'équivalence de Nerode est régulière à droite.

Nous supposons désormais que l'ensemble T d'états terminaux de $\mathcal{A}$ est non vide et distinct de Q. Considérons la partition de Q, $\mathcal{P}_0 = \{T, \bar{T}\}$, et $\mathcal{R}_0$ la relation d'équivalence associée. Compte tenu des définitions données, on peut énoncer la proposition suivante :

Proposition 6.7. *L'équivalence de Nerode est l'équivalence régulière à droite la plus grossière qui soit plus fine que $\mathcal{R}_0$.*

Preuve. Soit $\mathcal{R}$ l'équivalence de Nerode. D'après la proposition 5.6, il est clair que $\mathcal{R}$ est régulière à droite et plus fine que $\mathcal{R}_0$.

Soit $\mathcal{R}'$ une relation d'équivalence régulière à droite et plus fine que $\mathcal{R}_0$. Supposons que $\mathcal{R}'$ ne soit pas plus fine que $\mathcal{R}$. Dans ce cas il existe $p, q \in Q$ tels que $p\mathcal{R}'q$ et $L_p \neq L_q$. Alors on peut supposer par exemple qu'il existe $u \in L_p$ et $u \notin L_q$. Soit $t = p \cdot u$ et $q' = q \cdot u$. On a $t \in T$, $q' \notin T$ et $t\mathcal{R}'q'$ puisque $p\mathcal{R}'q$. Ce qui induit une contradiction puisque $\mathcal{R}_0 \leq \mathcal{R}'$. ∎

Une partition $\mathcal{P}$ de l'ensemble Q sera dite *admissible* si la relation d'équivalence associée est plus fine que $\mathcal{R}_0$ et moins fine que l'équivalence de Nerode. La proposition 6.7 devient :

Lemme 6.8. *Une partition de l'ensemble Q est la partition de Nerode de l'automate $\mathcal{A}$ si et seulement si elle est admissible et stable.* ∎

Par ailleurs on déduit facilement de ce qui précède la propriété suivante :

Lemme 6.9. *Si $\mathcal{P}$ est une partition admissible, non stable pour le couple (P, a), alors $\mathcal{P} \underset{a}{\lhd} P$ est une partition admissible.* ∎

Pour calculer l'automate minimal de l'automate $\mathcal{A} = (Q, i, T)$, nous allons calculer son équivalence de Nerode et, pour cela, calculer la partition de Nerode associée à cette relation d'équivalence.

Le principe de l'algorithme est le suivant. On part de la partition $\mathcal{P}_0 = \{T, \bar{T}\}$ que l'on raffine par bris successifs; plus précisément, tant que $\mathcal{P}$ (au début $\mathcal{P} = \mathcal{P}_0$) n'est pas stable, on brise une classe de la partition.

Première écriture de l'algorithme

Soient $\mathcal{A} = (Q, i, T)$ un automate sur A et $\mathcal{P}_0 = \{T, \bar{T}\}$ la partition initiale de Q. Considérons l'algorithme suivant :

```
procédure MOORE1 (A);
  P := P_0;
  tantque ∃P ∈ P et a ∈ A tels que P n'est pas stable pour (P, a) faire
    P := P ◁_a P
  fintantque.
```

La boucle «tantque» est exécutée au plus $n-1$ fois, car à chaque exécution le nombre d'éléments de $\mathcal{P}$ augmente strictement et est majoré par n. La partition $\mathcal{P}$ obtenue est stable. Par ailleurs, «*P est admissible*» est un invariant de la boucle «tantque» en raison du lemme 6.9.

Comme $\mathcal{P}_0$ est admissible, cela implique que la partition obtenue à l'arrêt de la boucle est admissible, c'est donc la partition de Nerode. D'où :

Proposition 6.10. *La procédure* MOORE1 *calcule l'équivalence de Nerode.*

∎

Nous modifions la première procédure en transformant l'itération de la façon suivante : à chaque entrée dans la boucle on dresse la liste $\mathcal{L}$ des couples (P, a) pour lesquels $\mathcal{P}$ n'est pas stable, et on remplace $\mathcal{P}$ par $\text{SCINDER}(\mathcal{P}, \mathcal{L})$. Notons qu'on ne fait plus la même suite de scissions que dans la procédure MOORE1, car si $\mathcal{L} = ((P_1, a_1), \ldots, (P_k, a_k))$, et si l'on suppose que P_2 a été scindé lors de la scission relative à (P_1, a_1), alors dans la procédure MOORE1 on ne fera pas de scission relativement à (P_2, a_2) parce que P_2 n'appartient plus à la partition. Il est facile de voir que cette modification n'altère pas la validité de l'algorithme. On peut énoncer le résultat ainsi :

Lemme 6.11. *Si $\mathcal{P}$ est une partition admissible et $\mathcal{L}$ est la liste des couples (P, a) ($P \in \mathcal{P}$, $a \in A$) pour lesquels $\mathcal{P}$ n'est pas stable, alors* $\text{SCINDER}(\mathcal{P}, \mathcal{L})$ *est encore une partition admissible.*

Preuve. Il suffit de prouver que si $\mathcal{P}$ est admissible et que P est une union d'éléments de $\mathcal{P}$ alors $\mathcal{P} \lhd_a P$ est encore admissible. La preuve est similaire à celle de la Proposition 6.10 et est laissée au lecteur. ∎

On peut donc remplacer la procédure MOORE1 par la suivante : Note 9.6.3

```
procédure MOORE2 (A);
  P := P_0;
  tantque P n'est pas stable faire
    calculer la liste L des couples (P, a) qui brisent P;
    PP ◁ L
  fintantque.
```

Donnons un exemple pour montrer que la suite des scissions n'est pas la même dans les procédures MOORE1 et MOORE2.

Exemple. Considérons l'automate de la figure 6.2.

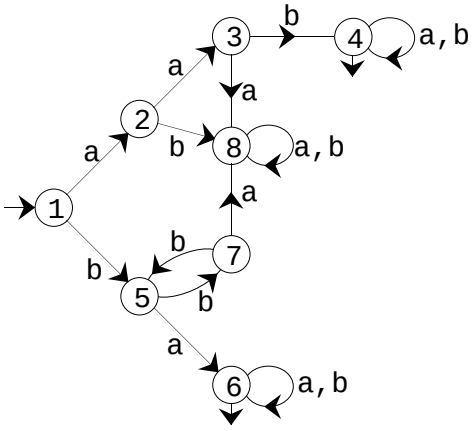


Figure 6.2: *Exemple.*

Note 9.6.4

La procédure MOORE1 donne la suite de scissions suivantes :

$\mathcal{P} : 46 - 123578$
Scission relative à $(46, a)$ — $\mathcal{P} : 46 - 12378 - 5$
Scission relative à $(5, b)$ — $\mathcal{P} : 46 - 17 - 28 - 3 - 5$
Scission relative à $(28, a)$ — $\mathcal{P} : 46 - 1 - 7 - 2 - 8 - 3 - 5$
Terminé

La procédure MOORE2 donne la suite de scissions suivantes :

$\mathcal{P} : 46 - 123578$
$\mathcal{L} = ((46, a), (46, b), (123578, a), (123578, b))$
Scission relative à $(46, a)$ — $\mathcal{P} : 46 - 12378 - 5$
Scission relative à $(46, b)$ — $\mathcal{P} : 46 - 1278 - 3 - 5$
Scission relative à $(123578, a)$ — $\mathcal{P}$: pas de changement
Scission relative à $(123578, b)$ — $\mathcal{P}$: pas de changement
$\mathcal{L} = ((1278, a)(1278, b), (3, a), (5, b))$
Scission relative à $(1278, a)$: — $\mathcal{P} : 46 - 178 - 2 - 3 - 5$
Scission relative à $(1278, b)$: — $\mathcal{P} : 46 - 17 - 8 - 2 - 3 - 5$
Scission relative à $(3, a)$: — $\mathcal{P}$: pas de changement
Scission relative à $(5, b)$: — $\mathcal{P} : 46 - 1 - 7 - 8 - 2 - 3 - 5$
$\mathcal{L} = \emptyset$
Terminé

L'automate minimal obtenu est donc le suivant et reconnaît le langage $\{a^2b, b(b^2)^* a\}\{a, b\}^*$:

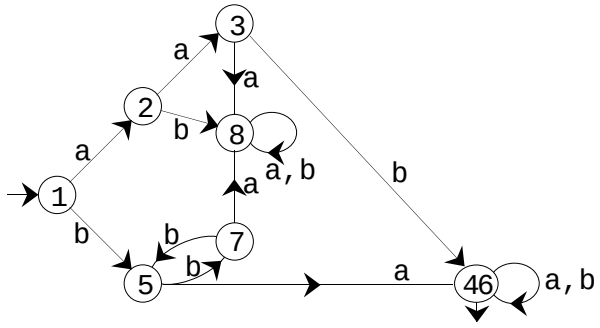


Figure 6.3: *Automate minimal obtenu à partir de celui de la figure 6.2.*

Version définitive de l'algorithme

Contrairement à ce que laisserait croire l'exemple traité, l'écriture de l'algorithme sous cette forme va nous permettre de mieux exploiter les propriétés de l'opération SCINDER pour améliorer les performances de l'algorithme.

Dans un premier temps, au lieu de calculer les couples (P, a) pour lesquels $\mathcal{P}$ n'est pas stable, on peut prendre $\mathcal{L} = \mathcal{P} \times A$; cela revient à ajouter des couples pour lesquels $\mathcal{P}$ est stable et pour lesquels l'opération SCINDER n'aura aucun effet. Ceci fait, en vertu du lemme 6.5 iv), on peut remplacer à la première itération $\mathcal{L} := \{\bar{T}, T\} \times A$ par $\mathcal{L} := \{T\} \times A$ ou $\mathcal{L} := \{\bar{T}\} \times A$. On choisira entre T et $\bar{T}$, la partie de plus faible cardinal, pour des raisons qui s'éclairciront plus tard mais dont on devine d'ores et déjà l'intérêt (réduction de la taille des ensembles relativement auxquels on fait la scission). Ceci nous amène à écrire la version quasiment définitive de l'algorithme. Ecrivons l'algorithme sous la forme suivante :

procédure HOPCROFT $(\mathcal{A})$;
  $\mathcal{P} := \{T, \bar{T}\}$;
  si Card$(T) <$ Card$(\bar{T})$ alors $\mathcal{L} := \{T\} \times A$ sinon $\mathcal{L} := \{\bar{T}\} \times A$;
  tantque $\mathcal{L} \neq \emptyset$ faire
  (a) enlever un couple (P, a) de $\mathcal{L}$;
  (b) déterminer l'ensemble $\mathcal{B}$ des éléments de $\mathcal{P}$ brisés par (P, a);
  (c) pour tout $B \in \mathcal{B}$ calculer $\{B', B''\} = B \underset{a}{\lhd} P$;
  (d) pour tout $B \in \mathcal{B}$ et pour tout $b \in A$ faire
        si $(B, b) \in \mathcal{L}$ alors
  (e)   enlever (B, b) et ajouter (B', b) et (B'', b) à $\mathcal{L}$
  (f)   sinon si Card$(B') <$ Card(B'') alors ajouter (B', b) à $\mathcal{L}$
              sinon ajouter (B'', b) à $\mathcal{L}$
        finsi
  fintantque;
  retourner $(\mathcal{P})$.

Dans cette version de l'algorithme, la liste $\mathcal{L}$ représente certains des couples (P, a) pour lesquels il faut scinder $\mathcal{P}$. Mais à un instant donné, $\mathcal{L}$ ne représente pas la liste de *tous* les couples $(P, a),\ P \in \mathcal{P}$ pour lesquels $\mathcal{P}$ n'est pas stable. Néanmoins la propriété suivante (I) est un invariant de la boucle «tantque» :

$$\text{si } P \in \mathcal{P} \text{ et } (P, a) \notin \mathcal{L} \text{ pour un certain } a, \text{ alors}$$

a) $\mathcal{P}$ est stable pour (P, a)

$$\text{ou} \qquad (I)$$

b) à l'arrêt de la boucle «tantque», la partition $\mathcal{P}$ est stable pour (P, a).

Prouvons donc que (I) est un invariant de la boucle «tantque» en utilisant les lemmes 6.4 et 6.5 .

Appelons $\mathcal{P}_k$ la partition obtenue avant la k-ième itération de la boucle, et $\mathcal{P}_N$ la partition finale.

Avant d'entrer dans la boucle «tantque» (I) est vrai : supposons que **a)** soit faux; comme $(P, a) \notin \mathcal{L}$, alors $(\bar{P}, a) \in \mathcal{L}$, et les lemmes 6.4 et 6.5 assurent que les transformations de $\mathcal{L}$ qui ont lieu en (a), (e), ou (f) sont telles que $\mathcal{P}_N$ est stable pour $(\bar{P}, a)$ et donc pour (P, a).

Supposons maintenant qu'avant la k-ième itération, (I) soit vrai et soient respectivement $\mathcal{L}_k$ et $\mathcal{L}_{k+1}$ l'état de $\mathcal{L}$ avant et après cette k-ième itération. Montrons que (I) est vrai pour $\mathcal{P}_{k+1}$ à la sortie de la k-ième itération de la boucle «tantque».

Soit un couple $(P, a) \notin \mathcal{L}_{k+1}$, avec $P \in \mathcal{P}_{k+1}$;

- ou bien $P \in \mathcal{P}_k$;

si $(P, a) \notin \mathcal{L}_j$, par hypothèse de récurrence $\mathcal{P}_k$ est stable pour (P, a) et donc aussi $\mathcal{P}_{k+1}$, ou bien $\mathcal{P}_N$ est stable pour (P, a) (cqfd);

si $(P, a) \in \mathcal{L}_j$, il est facile de voir que nécessairement la k-ième itération est une scission relativement à (P, a) et que P n'a pas été brisé lors de cette scission. La partition $\mathcal{P}_{k+1}$ est donc stable pour (P, a) ainsi que toutes les suivantes (cqfd);

- ou bien $P \notin \mathcal{P}_k$;

alors P a été obtenu lors de la k-ième itération par «bris» d'une partie R en deux parties P et P', avec $R \in \mathcal{P}_k$;

si $(R, a) \notin \mathcal{L}_k$, alors à la sortie de la boucle on a $(P, a) \notin \mathcal{L}_{k+1}$ par hypothèse, et donc $(P', a) \in \mathcal{L}_{k+1}$. Ainsi on est sûr que $\mathcal{P}_N$ sera stable pour (P', a) et par hypothèse de récurrence que $\mathcal{P}_k$ est stable pour (R, a) ou que $\mathcal{P}_N$ sera stable pour (R, a). Donc par application du lemme 6.4 ii), $\mathcal{P}_N$ sera stable pour (P, a) (cqfd);

si $(R, a) \in \mathcal{L}_k$, comme $(P, a) \notin \mathcal{L}_{k+1}$, c'est que la scission faite lors de la k-ième itération est une scission relative à (R, a) (sinon, puisque $(R, a) \in \mathcal{L}_k$, on aurait (P, a) et $(P', a) \in \mathcal{L}_{k+1}$, ce qui contredit $(P, a) \notin \mathcal{L}_{k+1}$). Donc $\mathcal{P}_N$ sera stable pour (R, a), et puisque $(P, a) \notin \mathcal{L}_{k+1}$ c'est que $(P', a) \in \mathcal{L}_{k+1}$, et donc $\mathcal{P}_N$ sera stable pour (P', a). D'où l'on déduit que $\mathcal{P}_N$ sera stable pour (P, a) (cqfd).

On peut donc énoncer le lemme :

Lemme 6.12. *L'algorithme* HOPCROFT *s'arrête et calcule la partition de Nerode.*

Preuve. Il reste uniquement à vérifier que la boucle «tantque» s'arrête. Or, on ne peut ajouter deux fois un même couple (P, a) dans $\mathcal{L}$, car chaque couple (P, a) ajouté dans la boucle en (e) ou (f) est constitué d'une partie P strictement contenue dans une classe de la partition courante, et comme chaque passage dans la boucle supprime un élément de $\mathcal{L}$, la boucle «tantque» s'arrête. ∎

Nous énonçons un dernier lemme utile à l'étude de la complexité de l'algorithme :

Lemme 6.13. *Le nombre d'itérations de la boucle «tantque» est majoré par* $2mn$.

Preuve. Il suffit de prouver que le nombre de couples (P, a) introduits dans $\mathcal{L}$ est au plus $2mn$, puisqu'à chaque itération on enlève un couple de $\mathcal{L}$. Pour chaque couple (P, a) introduit dans $\mathcal{L}$, P est élément de la partition courante. Considérons l'arbre binaire représentant les scissions successives opérées sur $\mathcal{P}$. La racine est la partie Q toute entière et ses fils sont T et $\bar{T}$ (si $T = Q$, on part de T), et chaque nœud P admet pour fils P' et P'', résultats d'une scission appliquée à P. Un tel arbre binaire complet a au plus n feuilles et donc au plus $2n - 1$ sommets. D'où le résultat. ∎

9.6.4 Complexité de l'algorithme

Avant de décrire les structures de données nécessaires à l'implémentation de l'algorithme, nous allons détailler l'écriture des instructions (b), (c) et (d) de la procédure HOPCROFT.

Détaillons l'écriture des instructions (b), (c) et (d) de la procédure HOPCROFT :

On remplace (b) par :

> (b) calculer $a^{-1}P$;
> dresser la liste CLASSES-RENCONTRÉES des classes $B \in \mathcal{P}$ telles que $B \cap a^{-1}P \neq \emptyset$;

Notons que cette liste contient toutes les classes susceptibles d'être brisées, mais certaines d'entre elles peuvent être stables pour (P, a). C'est à l'étape suivante que l'on détermine les classes devant être brisées. Ainsi (c-f) est remplacé par :

```
(c)  pour tout B ∈Classes-Rencontrées faire
       si B · a ⊄ P alors
         créer une nouvelle classe Jumeau(B);
         enlever de B et mettre dans Jumeau(B)
         tous les états p tels que p · a ∈ P;
(d)      pour tout b ∈ A faire
           si avant scission (B, b) ∈ L alors
(e)          ajouter (Jumeau(B),b) à L
(f)        sinon si après scission Card(B) ≤ Card(Jumeau(B)) alors
                     ajouter (B, b) à L
                   sinon ajouter (Jumeau(B),b) à L
           finsi
         finpour
       finsi
     finpour.
```

Considérons une itération de la boucle (c) pour laquelle $B \cdot a \not\subset P$. A la fin de cette itération, B et Jumeau(B) sont exactement les ensembles B' et B''. D'où l'algorithme complet :

```
procédure HOPCROFT (A);
P := {T, T̄};
si Card(T) < Card(T̄) alors L := {T} × A sinon L := {T̄} × A;
tantque L ≠ ∅ faire
(a)  enlever un couple (P, a) de L;
(b)  INVERSE:= a⁻¹P;
     dresser la liste CLASSES-RENCONTRÉES des classes
     B ∈ P telles que B∩INVERSE≠ ∅;
(c)  pour tout B ∈CLASSES-RENCONTRÉES faire
       si B · a ⊄ P alors
         créer une nouvelle classe JUMEAU(B);
         déplacer de B dans JUMEAU(B) tous les états p | p · a ∈ P;
(d)      pour tout b ∈ A faire
           si avant scission (B, b) ∈ L alors
             ajouter (JUMEAU(B),b) à L
           sinon si après scission Card(B) ≤ Card(JUMEAU(B)) alors
                     ajouter (B, b) à L
                   sinon ajouter (JUMEAU(B),b) à L
           finsi
         finpour
       finsi
     finpour
fintantque.
```

Structures de données

• Les *états* de l'automate sont les entiers de 1 à n. Les lettres sont les entiers de 1 à m. Une partition ayant au plus n éléments, un élément d'une partition aura pour nom un entier entre 1 et n, ainsi un couple (P, a) sera représenté comme un élément de $\{1, \ldots, n\} \times \{1, \ldots, m\}$, où la première composante est le nom de la partie P. Comme le nombre de classes grossit au cours du temps, une variable COMPT initialisée à 2 (ou à 1 si $T = Q$) indique quels sont les entiers déjà utilisés pour nommer les classes, à savoir les entiers de 1 à COMPT.

Une partition de Q sera gérée grâce à quatre tableaux indicés par les entiers de 1 à n : CLASSE, PART, CARD, PLACE.

Pour tout état i, CLASSE[i] est le nom de la classe contenant i. Pour toute classe de nom P, PART[P] est un pointeur sur une liste doublement chaînée des éléments de P, et CARD[P] est le nombre d'éléments de la classe P. Enfin, pour tout état i appartenant à la classe de nom P, PLACE[i] est un pointeur sur la «place» de i dans la liste doublement chaînée PART[P] des éléments de la classe de nom P. Cela permet en temps $O(1)$ de supprimer un élément d'une classe et de l'insérer dans une autre.

Exemple. $Q = \{1, \cdots, 6\}$, $\mathcal{P}$:$35 - 1 - 246$

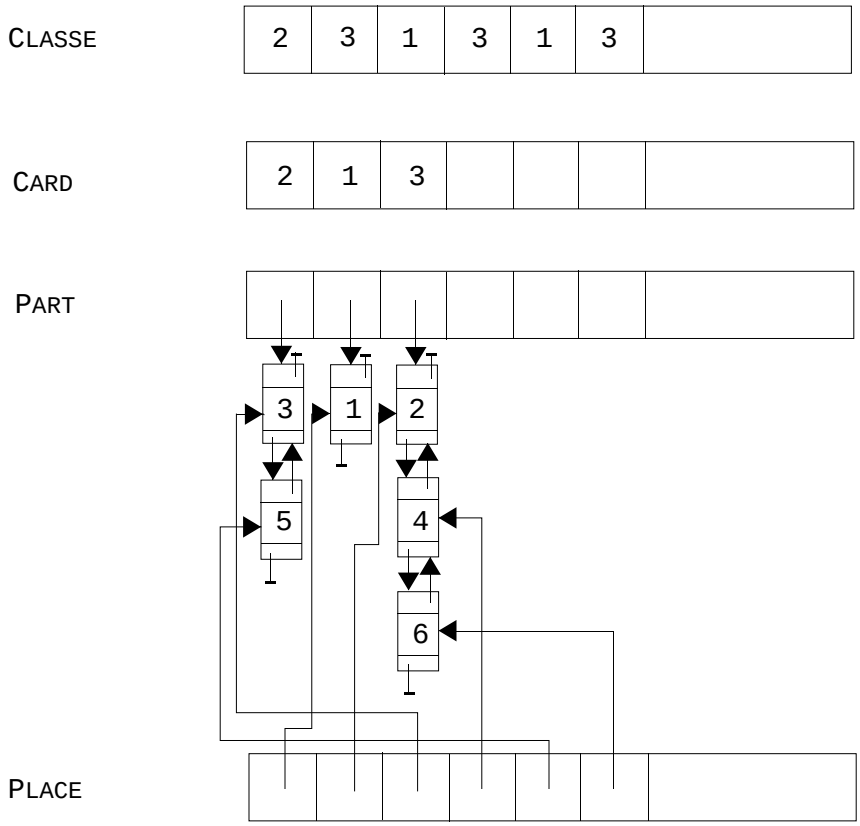


Figure 6.4: *Un exemple.*

• Pour calculer INVERSE:= $a^{-1}P$, on dispose d'un tableau INV indicé par $\{1, \ldots, n\} \times \{1, \ldots, m\}$ tel que INV$[p, a]$ est un pointeur sur une liste chaînée des éléments q tels que $q \cdot a = p$. Ainsi INVERSE sera non pas construit «physiquement», mais «lu» en temps proportionnel à sa taille en parcourant la liste PART$[P]$ et, pour chaque élément p de cette liste, en parcourantINV$[p, a]$. Ainsi l'expression «pour tout q dans INVERSE» est programmée par : pour tout $p \in P$ et pour tout $q \in$INV$[p, a]$.

• La liste $\mathcal{L}$ doit être gérée de telle sorte que les opérations d'adjonction, de recherche, et de suppression d'un élément *arbitraire* soient aisées (arbitraire fait référence ici à la ligne (1) de la procédure HOPCROFT, c'est-à-dire qu'on veut supprimer un élément de $\mathcal{L}$, peu importe lequel). Comme la taille de $\mathcal{L}$ est majorée par mn, on la représentera par un tableau de booléens LISTE indicé par $\{1, \ldots, n\} \times \{1, \ldots, m\}$ tel que LISTE$[P, a]$ est vrai si $(P, a) \in \mathcal{L}$, et simultanément par une liste chaînée. Ainsi, l'adjonction se fait en temps $O(1)$, (la mise à jour dans le tableau et dans la liste chaînée prenant un temps constant) ; la recherche se fait en temps $O(1)$ en utilisant le tableau ; enfin la suppression d'un élément arbitraire (ligne (a) de la procédure) se fait en deux temps, on enlève le premier élément de la liste chaînée puis on met à jour le tableau, ce qui prend un temps

$O(1)$, de même le test $\mathcal{L} \neq \emptyset$ se fait en temps $O(1)$ grâce à la structure de liste chaînée.

• Le traitement de Classes-Rencontrées et de Jumeau est relativement délicat.

Pour dresser la liste de Classes-Rencontrées, et pour gérer les scissions, on utilise trois structures de données :

une liste chaînée Classes-Rencontrées des noms des classes à scinder,

un tableau d'entiers Partage indicé par $\{1, \ldots, n\}$, tel que pour toute classe B de nom i, Partage$[i]$ est le cardinal de $B \cap a^{-1}P$ (le tableau Partage doît être à 0 à chaque entrée dans la boucle «tantque»)

et un tableau Jumeau indicé par $\{1, \ldots, n\}$. L'élément Jumeau$[i]$ est le nom de la nouvelle classe créée pour briser la classe de nom i. Ce tableau est initialisé à 0 au début de l'algorithme.

Analyse de la complexité

Analysons le temps d'exécution de la procédure Hopcroft avec les structures de données proposées ci-dessus. Notons au passage que pour obtenir une faible complexité en temps il a été nécessaire de ne pas lésiner sur l'espace mémoire utilisé.

L'initialisation qui précède la boucle «tantque» prend un temps $O(mn)$. Le nombre d'itérations de la boucle «tantque» étant majoré par $2mn$, le temps global d'exécution de la ligne (a) est $O(mn)$.

On a vu dans la preuve du lemme 6.13 que le nombre de classes créées est majoré par $2n-1$, donc le temps global d'exécution du bloc (d) prend un temps $O(mn)$.

Il reste à évaluer le temps global d'exécution des blocs (b) et (c). Soit N le nombre de passages dans la boucle «tantque». Dans un premier temps, évaluons la somme des tailles des N listes «abstraites» Inverse parcourues.

Proposition 6.14. *Soient $a \in A$, $p \in Q$. Le nombre de fois où l'on supprime de la liste $\mathcal{L}$ un couple (P, a) tel que $p \in P$, est majoré par $\log_2 n$.*

Preuve. On dira qu'un couple (P, a) est *marqué* si $p \in P$ (a et p sont fixés). Avant d'entrer dans la boucle «tantque», $\mathcal{L}$ contient au plus un couple marqué, et ceci reste vrai à tout instant. Soit (P, a) le premier couple marqué qui soit supprimé de $\mathcal{L}$. Le prochain couple (P'', a) marqué qui sera introduit dans $\mathcal{L}$ dans la boucle «tantque» est tel que P'' est une classe résultat de Scinder(P', b), pour un $b \in A$ où P' est une partie de P et $\mathrm{Card}(P'') \leq \mathrm{Card}(P')/2$. Tant qu'il n'y a pas de suppression dans $\mathcal{L}$ d'un couple marqué, $\mathcal{L}$ contient un unique couple marqué dont la première composante est un sous-ensemble de P'', donc sa taille est inférieure ou égale à celle de P''. Ainsi, le prochain couple marqué qui sera supprimé dans $\mathcal{L}$

aura une taille inférieure ou égale à $\mathrm{Card}(P)/2$. En itérant ce processus, et tenant compte du fait que le premier couple marqué introduit (éventuellement) dans $\mathcal{L}$ vérifie $\mathrm{Card}(P) \leq n/2$, il s'ensuit qu' on ne peut supprimer de $\mathcal{L}$ plus de $\log_2 n$ fois un tel couple. ∎

Corollaire 6.15. *La somme des tailles des N listes* INVERSE *parcourues est majorée par $mn \log_2 n$.*

Preuve. Soit un triplet fixé (p, a, q) tel que $q = p \cdot a$. Le nombre de fois où p est «lu» dans la liste INVERSE parce que $q = p \cdot a$, est exactement égal au nombre de couples (P, a) tels que $q \in P$, qui sont supprimés de la liste $\mathcal{L}$. Or par la proposition 6.14 ce nombre est majoré par $\log_2 n$. D'où le résultat. ∎

Il reste à constater que le temps global d'exécution des blocs (b) et (c) est proportionnel à la somme des tailles des N listes INVERSE.

En réalité, les blocs (b) et (c) s'exécutent en parcourant deux fois la liste INVERSE de la manière suivante :

```
(b) créer une liste vide CLASSES-RENCONTRÉES;
    pour tout p ∈ P et pour tout q ∈INV[p, a] faire
      i :=CLASSE[q];
      si PARTAGE[i] = 0 alors
        PARTAGE[i] := 1;
        ajouter i à CLASSES-RENCONTRÉES
      sinon
        PARTAGE[i] :=PARTAGE[i] + 1
      finsi
    finpour;
```

```
(c) pour tout p ∈ P et pour tout q ∈INV[p, a] faire
      i :=CLASSE[q];
      si PARTAGE[i] <CARD[i] alors
        si JUMEAU[i] = 0 alors
          COMPT:=COMPT+1;JUMEAU[i] :=COMPT
        finsi;
        supprimer q de sa classe et l'insérer dans
        la classe de nom JUMEAU[i]
      finsi
    finpour;
    pour tout j appartenant à CLASSES-RENCONTRÉES faire
      PARTAGE[j] := 0;JUMEAU[j] := 0;
```

En écrivant ainsi les blocs (b) et (c) et compte-tenu des structures de données choisies, il est clair que le temps d'exécution de chacun de ces blocs prend un temps proportionnel à la somme des tailles des N listes INVERSE et donc un temps $O(mn \log n)$.

En conclusion, on peut énoncer le :

Théorème 6.16. *Si A est un alphabet à m lettres et $\mathcal{A}$ un automate à n états sur cet alphabet, la procédure* HOPCROFT *calcule, dans le pire des cas, en temps $O(mn \log n)$ l'automate minimal de $\mathcal{A}$.*

Notes

La théorie des automates, dont nous avons décrit ici les bases, a connu des développements considérables. Un traité substantiel est l'ouvrage de S. Eilenberg :

S. Eilenberg, *Automata, Languages, and Machines*, Vol. A Academic Press, 1974.

Pour les développements récents, notamment en rapport avec l'algorithmique des mots, voir :

D. Perrin, Finite Automata, in : J. van Leeuwen, ed., *Handbook of Theoretical Computer Science, Vol. B*, North-Holland, 1990, 1–57.

Une théorie des langages rationnels et reconnaissables de mots infinis a vu le jour en même temps que la théorie portant sur les mots finis. On pourra consulter le chapitre de W. Thomas dans ce même volume.

Exercices

9.1. Montrer les égalités suivantes pour $X, Y, Z \subset A^*$:

$$\begin{aligned} (XY)^{-1}Z &= Y^{-1}(X^{-1}Z) \\ X^{-1}(YZ^{-1}) &= (X^{-1}Y)Z^{-1} \\ (XZ^{-1})Y^{-1} &= X(YZ)^{-1} \end{aligned}$$

9.2. Décrire un algorithme qui permet de tester si le langage reconnu par un automate donné est vide, fini non vide, ou infini.

9.3. (Lemme de l'étoile) Démontrer que pour tout langage reconnaissable L, il existe un entier N tel que tout mot w de L de longueur $|w| \geq N$ admet une factorisation $w = uxv$ avec $0 < |x| \leq N$ et vérifiant $ux^*v \subset L$.

9.4. Utiliser le lemme de l'étoile pour prouver que les langages $\{a^n b^n \mid n \geq 0\}$, $\{a^n b^p \mid n \geq p \geq 0\}$, $\{a^n b^p \mid n \neq p\}$ ne sont pas reconnaissables.

9.5. Démontrer que si L est un langage reconnaissable, alors l'ensemble des facteurs des mots de L est encore un langage reconnaissable. (Même question pour les préfixes, suffixes.)

9.6. Soient A et B deux alphabets. Un *morphisme* de A^* dans B^* est une application $f : A^* \to B^*$ vérifiant $f(uv) = f(u)f(v)$ pour $u, v \in A^*$.

a) Démontrer que si K est un langage rationnel sur A, alors $f(K)$ est un langage rationnel sur B.

b) Démontrer que $f^{-1}(L)$ est un langage rationnel sur A pour tout langage rationnel L sur B.

9.7. Une *substitution* de A^* dans B^* est une application s de A^* dans l'ensemble des parties de B^* qui vérifie $s(uv) = s(u)s(v)$ pour $u, v \in A^*$, et $s(1) = \{1\}$. Démontrer que si $s(a)$ est un langage rationnel pour tout $a \in A$, alors $s(K)$ est un langage rationnel pour tout langage rationnel K sur A.

9.8. Montrer que les deux automates de la figure 6.5 reconnaissent le même langage.

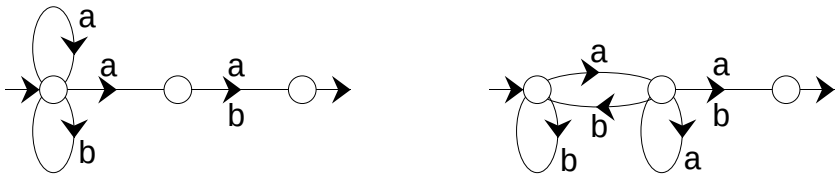


Figure 6.5: *Deux automates non déterministes reconnaissant le même langage.*

Quel est l'automate minimal déterministe reconnaissant ce langage? Plus généralement, montrer que les langages A^*wA^n, avec $A = \{a, b\}$, $n > 0$ sont reconnus par au moins $|w| + 1$ automates non déterministes non isomorphes ayant strictement moins d'états que l'automate (déterministe) minimal de ce langage.

9.9. Soient $\mathcal{A} = (Q, i, T)$ et $\mathcal{A}' = (Q', i', T')$ deux automates déterministes sur un alphaber A. On se propose de tester si ces deux automates sont *équivalents*, c'est-à-dire reconnaissent le même langage.

1) Donner un algorithme résolvant ce problème d'équivalence en utilisant la minimisation des automates déterministes. Quelle est sa complexité?

2) a) Soit $\mathcal{A}'' = (Q'', \{i, i'\}, T \cup T')$ l'automate union des automates $\mathcal{A}$ et $\mathcal{A}'$ (les ensembles Q et Q' sont supposés disjoints). L'automate $\mathcal{A}''$ est déterministe à ceci près qu'il a deux états initiaux. Pour tout $a \in A$, on note δ''_a la restriction à $Q'' \times \{a\}$ de la fonction de transition de l'automate $\mathcal{A}''$. Soit $\mathcal{P}$ la partition la plus fine de l'ensemble Q'' telle que i et i' soient dans la même classe, et qui soit compatible avec la fonction de transition de l'automate $\mathcal{A}''$ i.e. vérifiant $p \sim q \Rightarrow \delta''_a.p \sim \delta''_b.q$ pour tout $p, q \in Q''$ et tout $a \in A$, où $\sim$ désigne la relation d'équivalence associée à la partition considérée.

b) Montrer que les automates $\mathcal{A}$ et $\mathcal{A}'$ sont équivalents si et seulement si l'ensemble $T \cup T'$ est saturé pour la relation d'équivalence associée à la partition $\mathcal{P}$.

c) En déduire un algorithme résolvant le problème d'équivalence de deux automates déterministes. On utilisera pour ce faire l'algorithme « union-find ». Calculer sa complexité

Cet algorithme est dû à Hopcroft et Karp.

9.10. Soit $\mathcal{A} = (Q, F, I, T)$ un un automate fini sur un alphabet A reconnaissant un langage L. Soit $\mathcal{A}^r = (Q, F^r, T, I)$ l'automate obtenu en renversant les flèches de $\mathcal{A}$ (i.e. $(p, a, q) \in F^r \Leftrightarrow (q, a, p) \in F$) et en échangeant états initiaux et terminaux.

1) Montrer que $\mathcal{A}^r$ reconnaît l'image miroir de L notée L^r, i.e. l'ensemble :

$$L^r = \{a_1 \cdots a_n \in A^* \mid \text{ pour tout } \ a_i \in A \text{ et } a_n \cdots a_1 \in L\}$$

2) Soit $\mathcal{A}^r_d$ l'automate déterminste émondé obtenu à partir de $\mathcal{A}^r$ par la construction donnée dans ce chapitre. Montrer que $\mathcal{A}^r_d$ est l'automate déterministe *minimal* reconnaissant L^r.

La preuve de ce résultat est due à J. Brzozowski.

3) En déduire un algorithme de calcul de l'automate déterministe minimal d'un automate donné, et en donner sa complexité.

Chapitre 10

Motifs

Dans ce chapitre, nous considérons d'abord le problème de la recherche d'une ou de toutes les occurrences d'un mot x dans un texte t. Nous présentons l'algorithme naïf, l'algorithme de Morris et Pratt, et sa variante due à Knuth, Morris et Pratt, l'implémentation par automate fini et l'algorithme de Simon, et pour finir l'algorithme de Boyer et Moore, dans sa version de Horspool et dans la version complète. Ensuite, nous considérons la recherche d'une occurrence de plusieurs motifs, et décrivons l'algorithme de Aho et Corasick. Dans la dernière section, nous étudions la recherche d'occurrences de mots décrits par une expression rationnelle.

Introduction

La recherche de motifs dans un texte est un problème important qui apparaît dans de nombreux domaines scientifiques. En informatique, on le rencontre naturellement en traitement des données, dans l'édition de textes, en analyse syntaxique ou en recherche d'informations ; en particulier, tous les éditeurs de textes et de nombreux langages de programmation offrent des possibilités de recherche de motifs.

Dans sa forme la plus simple, le problème se ramène à localiser une occurrence d'un mot, le *motif*, dans une chaîne de caractères, le *texte*. Par exemple, le texte `rechercher` contient deux occurrences du motif `cher`. Même pour ce problème simple, il existe de nombreux algorithmes intéressants, plus ou moins sophistiqués, et qui sont plus efficaces que la méthode naïve qui vient immédiatement à l'esprit. Le problème devient plus complexe lorsque l'on autorise des ensembles de motifs ou des motifs représentés par des expressions rationnelles. Par exemple, dans plusieurs traitements de textes courants l'expression `ch.*r` dénote les mots qui commencent par `ch` et qui se terminent par `r`. Dans le texte `rechercher`, les mots `cher` et `chercher` sont des occurrences de mots décrits par cette expression.

Le résultat effectif de l'algorithme dépend de l'application considérée. Dans le cas où les données sont organisées en lignes par exemple, comme dans un dictionnaire ou un listage de programme, nous pouvons être intéressés par les lignes qui correspondent au motif. En compilation, on vise plutôt à partitionner la chaîne de caractères d'entrée en une suite de lexèmes, tels que commentaires, identificateurs, opérateurs, où la forme des lexèmes est déterminée par une expression rationnelle. Dans l'édition de textes, on cherche des occurrences de motifs en vue notamment de les remplacer par d'autres.

Dans ce chapitre, nous considérons trois problèmes : le plus important, et qui sera traité en détail, est la recherche d'une ou de toutes les occurrences d'un mot x dans un texte t. Nous présentons trois algorithmes pour ce problème. Ensuite, nous considérons la recherche d'une occurrence de plusieurs motifs, et enfin la recherche d'occurrences de mots décrits par une expression rationnelle. Dans tous les cas, c'est le motif recherché qui subit une analyse préalable, et jamais le texte. En effet, le motif est considéré comme fixe, et le texte est changeant et inconnu. Une situation duale, où le texte est fixe et le motif peut varier, se présente par exemple dans la recherche d'entrées dans un dictionnaire. Une recherche efficace demande alors d'organiser et de coder de manière convenable le dictionnaire. Ce problème ne sera pas considéré ici.

10.1 Recherche d'un motif

Le problème que nous abordons dans cette section est le suivant : étant donnés un *texte* $t \in A^*$ et un *motif* $x \in A^*$, où A est un alphabet, déterminer si x est un facteur de t, et le cas échéant trouver une occurrence de x dans t. L'efficacité d'un algorithme de recherche se mesure en nombre de comparaisons de caractères.

Posons $t = t_1 \cdots t_n$ et $x = x_1 \cdots x_m$, où les t_j et les x_i sont des lettres. Les algorithmes que nous présentons sont bâtis sur le schéma que voici : le motif x est comparé aux facteurs $t_{k+1} \cdots t_{k+m}$ de longueur m de t jusqu'à trouver une coïncidence, si elle existe. L'algorithme naïf fait cette comparaison pour toutes les positions $k = 0, \ldots, n-m$. Les améliorations que nous examinons ensuite ne font la comparaison que pour un sous-ensemble des positions, en tirant profit de la connaissance du texte t accumulée lors des comparaisons précédentes et d'un prétraitement du motif x. Le schéma général est donc :

```
RECHERCHE-D'UN-MOTIF(x, t) ;
  k := 0 ;
  tester l'égalité x = t_{k+1} ··· t_{k+m} ;
  s'il y a égalité, reporter k et arrêter
  sinon augmenter k et recommencer.
```

De façon imagée, la méthode consiste à faire «glisser» le motif x le long de la chaîne t, et à comparer x au bloc de t couvert (voir figure 1.1). Les divers algorithmes déterminent selon des critères différents la position suivante où le motif x a des chances de se trouver dans le texte t. L'efficacité d'une méthode spécifique dépend grandement du prétraitement appliqué au motif x. L'information recueillie sur le motif peut être employée pour former un analyseur, voire un automate fini, qui est ensuite appliqué au texte.

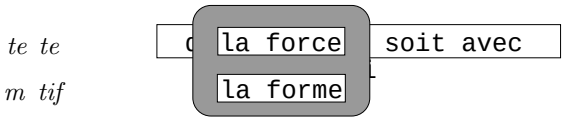


Figure 1.1: *Le motif glissant sur le texte.*

10.1.1 Un algorithme naïf

Posons $t = t_1 \cdots t_n$ et $x = x_1 \cdots x_m$. L'algorithme «naïf» consiste à comparer le motif x à chaque facteur de t de longueur m. Si une occurrence est rencontrée, on la signale; sinon, on recommence avec le facteur suivant de t. Le but est de calculer un entier k où commence une occurrence de x, c'est-à-dire tel que

$$x_1 \cdots x_m = t_{k+1} \cdots t_{k+m}$$

L'algorithme «naïf» suivant réalise cette recherche :

```
procédure RECHERCHE-NAÏVE(x, t);
  i := 1; j := 1;
  tantque i ≤ m et j ≤ n faire
    si t[j] = x[i] alors i := i + 1; j := j + 1
    sinon j := j − i + 2; i := 1 finsi
  fintantque;
  si i > m alors
    occurrence de x à la position j − m
  sinon
    pas d'occurrence
  finsi.
```

Dans le cas le plus défavorable, le nombre de comparaisons est $(n - m + 1)m = nm - m^2 + m$. Ce cas est réalisé par exemple pour $x = a^{m-1}b$, $t = a^{n-1}b$. Pour m petit devant n, ce nombre est de l'ordre de $nm = |x||t|$. Toutefois, le comportement moyen de l'algorithme naïf est plutôt bon, comme le montre l'observation suivante :

Proposition 1.1. *Si l'alphabet comporte au moins deux lettres, et dans l'hypothèse d'une distribution de probabilité uniforme et indépendante sur les lettres, le nombre moyen de comparaisons pour rechercher un motif dans un texte de longueur n par l'algorithme naïf est au plus $2n$.*

Bien entendu, les textes et les motifs n'ont, dans la pratique, aucune raison d'être équiprobables, ce qui limite quelque peu la portée de la proposition.

Preuve. Soit $q = |A|$. Considérons un motif fixé $x = x_1 \cdots x_m$. Nous montrons que le nombre moyen de comparaisons de caractères faites dans la comparaison de x à $t_{k+1} \cdots t_{k+m}$ est majoré par 2. Ce nombre est

$$\sum_{i=1}^{m} i c_i$$

où c_i est la probabilité pour que l'on fasse exactement i comparaisons. Or

$$\sum_{i=1}^{m} i c_i = \sum_{i=1}^{m} d_i$$

où $d_i = c_i + \cdots + c_m$ est la probabilité pour que l'on fasse au moins i comparaisons. Maintenant, on fait au moins i comparaisons lorsque

$$x_1 \cdots x_{i-1} = t_{k+1} \cdots t_{k+i-1}$$

de sorte que $d_i = 1/q^{i-1}$. Le nombre moyen de comparaisons est donc

$$1 + 1/q + \cdots + 1/q^{m-1} \leq 1 + \frac{1}{q-1} \leq 2$$

Il en résulte que le nombre moyen de comparaisons de l'algorithme naïf sur un texte t de longueur n est majoré par $2n$. ∎

10.1.2 L'algorithme de Morris et Pratt

La lenteur de l'algorithme naïf, dans le cas le plus défavorable, s'explique par le fait qu'en cas d'échec, il recommence à zéro la comparaison du motif x au facteur suivant de t, sans exploiter l'information contenue dans la réussite partielle de la tentative précédente. Si l'échec s'est produit à la i-ième lettre du motif x, on a (voir figure 1.2) pour un entier k

$$x_1 \cdots x_{i-1} = t_{k+1} \cdots t_{k+i-1} \quad \text{et} \quad x_i \neq t_{k+i}$$

Toute recherche ultérieure qui commence dans le facteur $t_{k+1} \cdots t_{k+i-1}$ peut tirer profit du fait que le mot $t_{k+1} \cdots t_{k+i-1}$ est un préfixe du motif x. Si une nouvelle recherche commence en position $\ell + 1$ (avec $k < \ell < k + i - 1$), elle compare

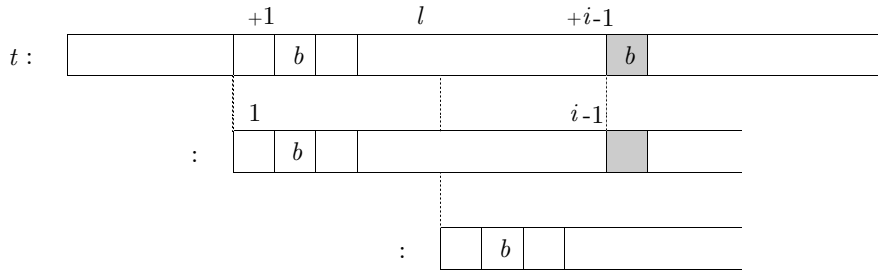


Figure 1.2: *Echec à la i-ième lettre du motif.*

$t_{\ell+1}t_{\ell+2}\cdots$ à $x_1x_2\cdots$. Or $t_{\ell+1}t_{\ell+2}\cdots$ est un suffixe de $t_{k+1}\cdots t_{k+i-1}$ qui lui est égal à $x_1\cdots x_{i-1}$. En d'autres termes, on compare $x_1x_2\cdots$ à un suffixe de $x_1\cdots x_{i-1}$, donc à un morceau du motif lui-même!

Ces comparaisons sont indépendantes du texte t, puisqu'elles ne concernent que des morceaux du motif. On peut donc les faire avant de considérer t, et on peut en attendre un gain de temps substantiel dans la mesure où ce prétraitement sur le motif ne sera fait qu'une seule fois et qu'il permettra d'éviter, lors de l'examen du texte, de répéter des comparaisons identiques à plusieurs endroits différents du texte.

Le *prétraitement* sur le motif x que nous allons réaliser permettra de reconnaître rapidement les seules configurations où la recherche d'une occurrence vaut la peine d'être continuée. Pour cela, il s'agit de déterminer les indices i où le mot $x_1\cdots x_{i-1}$ se termine par un préfixe de x. Introduisons une définition. Soit u un mot quelconque non vide; un *bord* de u est un mot distinct de u qui est à la fois préfixe et suffixe de u.

Exemple. Le mot $u = abacaba$ possède les trois bords ε, a, et aba. Le mot $u = abcabcab$ possède les bords ε, ab et $abcab$.

On note Bord(x) et on appelle *bord maximal* le bord le plus long d'un mot non vide x. Si x est de longueur m, on définit une fonction

$$\beta : \{0, 1, \ldots, m\} \to \{-1, \ldots, m-1\}$$

dépendant de x par $\beta(0) = -1$ et pour $i > 0$, par

$$\beta(i) = |\mathrm{Bord}(x_1 \cdots x_i)|$$

Bien entendu, $\beta(i) \leq i - 1$. Voici par exemple les bords maximaux et leurs longueurs, pour les préfixes du mot *abacabac* :

		a	b	a	c	a	b	a	c
Indice	0	1	2	3	4	5	6	7	8
Bord	$-$	ε	ε	a	ε	a	ab	aba	$abac$
β	-1	0	0	1	0	1	2	3	4

Revenons à l'amélioration de l'algorithme naïf. Si (voir figure 1.3)

$$x_1 \cdots x_{i-1} = t_{k+1} \cdots t_{k+i-1} \quad \text{et} \quad x_i \neq t_{k+i},$$

alors le mot $t_{k+p+1} \cdots t_{k+i-1}$ ne peut être début d'une occurrence de x que s'il est un bord de $x_1 \cdots x_{i-1}$. Il suffit donc, pour chercher une occurrence, de «décaler» x

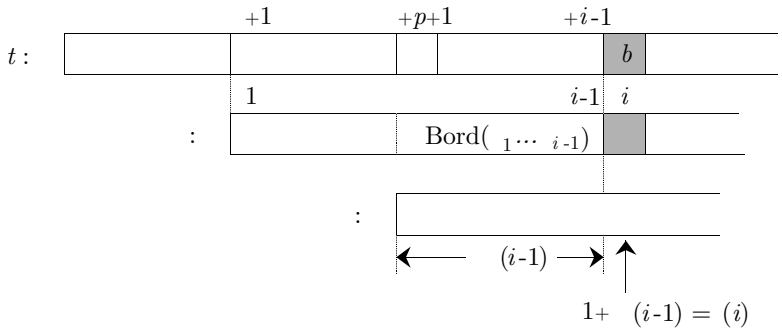


Figure 1.3: *Décalage d'une position.*

d'une longueur appropriée pour superposer $x_1 \cdots x_{i-1}$ avec son plus grand bord. Le décalage se réalise en poursuivant les comparaisons entre la lettre t_{k+i} et la lettre $x_{1+\beta(i-1)}$. Dans la pratique, on utilise à la place de β une fonction

$$s : \{1, \ldots, m\} \to \{0, \ldots, m\}$$

définie pour $i = 1, \ldots, m$ par

$$s(i) = 1 + \beta(i-1)$$

de sorte que le décalage consiste à remplacer i par $s(i)$. La fonction s est appelée la *fonction de suppléance* du motif x. Voici par exemple les fonctions β et s pour le mot *abacabac* :

		a	b	a	c	a	b	a	c
	0	1	2	3	4	5	6	7	8
β	-1	0	0	1	0	1	2	3	4
s		0	1	1	2	1	2	3	4

Avec la fonction de suppléance, on obtient l'algorithme ci-dessous, dû à Morris et Pratt. Dans cet algorithme, j est l'indice de la lettre courante du texte t, et i l'indice de la lettre courante du motif x. A chaque tour dans la boucle *tantque*, on a

$$x_1 \cdots x_{i-1} = t_{j-i+1} \cdots t_{j-1}$$

Si $t_j = x_i$, on progresse dans l'analyse, et sinon on décale x en remplaçant i par $s(i) = 1 + \beta(i-1)$.

```
procédure MORRIS-PRATT(x, t);
  i := 1; j := 1;
  tantque i ≤ m et j ≤ n faire
    si i ≥ 1 etalors t[j] ≠ x[i] alors i := s(i)
    sinon i := i + 1; j := j + 1 finsi
  fintantque;
  si i > m alors
    occurrence de x à la position j − m
  sinon
    pas d'occurrence de x dans t
  finsi.
```

Proposition 1.2. *L'algorithme de Morris et Pratt calcule une occurrence d'un motif x dans un texte t en au plus $2|t| - 1$ comparaisons de caractères, si l'on dispose de la fonction de suppléance sur x.*

Preuve. Posons $n = |t|$. Appelons test positif un test pour lequel $t[j] = x[i]$, et test négatif un test pour lequel $t[j] \neq x[i]$. Chaque test positif incrémente i et j, et chaque test négatif diminue i, éventuellement de plus d'une unité. Comme chaque test positif augmente j, il y a au plus n tests positifs. Il n'y a n tests positifs que si i ne s'annule pas.

Pour compter le nombre de tests négatifs, considérons l'entier $j - i$. Au début, $j - i = 0$. Un test positif ne change pas la valeur de $j - i$, un test négatif l'augmente strictement. Donc le nombre de tests négatifs est majoré par la valeur qu'a $j - i$ à la fin du calcul, donc par n ou par $n - 1$ selon que i s'annule ou non. ∎

Exemple. La table suivante donne, pour la recherche du motif $x = abacabac$ dans le texte $t = babacacabacaab$, la suite des valeurs que prend l'indice i dans l'algorithme :

j	1	2	3	4	5	6	7	8	9	10	11	12	13	14
	b	a	b	a	c	a	c	a	b	a	c	a	a	b
i	1	1	2	3	4	5	6	1	2	3	4	5	6	2
	0						2						2	
							1						1	
							0							

La figure 1.4 montre les décalages successifs du motif. Les différences sont constatées aux positions sombres du texte. Le nombre total de comparaisons est 18. Lorsque $j = 7$ et $i = 6$, on a $t_j \neq x_i$, et on compare t_7 à la lettre qui suit le bord maximal de $x_1 \cdots x_5$, à savoir x_2. Le même schéma se répète pour $j = 13$, et en fait chaque fois qu'une lettre t_j est comparée à x_6. A chaque fois, on compare t_j à x_2, et cette comparaison est inutile parce que $x_6 = x_2$. Une version de l'algorithme, due à Knuth, Morris et Pratt et que nous présentons plus loin permet d'éliminer en partie ces comparaisons redondantes.

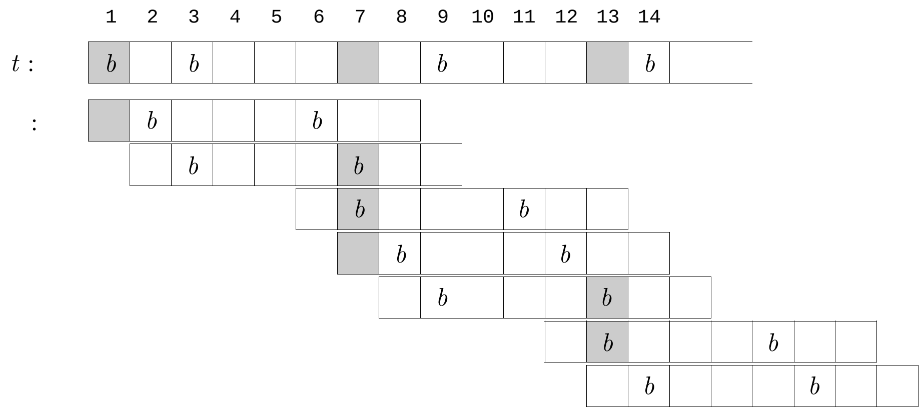


Figure 1.4: *Décalages successifs du motif.*

10.1.3 Bords

Pour compléter l'exposé de l'algorithme de Morris et Pratt, il faut indiquer comment calculer la fonction β ou, de manière équivalente, la fonction de suppléance. Cela demande une étude plus approfondie des bords d'un mot.

Proposition 1.3. *Soit x un mot non vide, et soit k le plus petit entier tel que* $\mathrm{Bord}^k(x) = \varepsilon$.

(1) *Les bords de x sont les mots* $\mathrm{Bord}(x), \mathrm{Bord}^2(x), \ldots, \mathrm{Bord}^k(x)$.

(2) *Soit a une lettre. Alors $\mathrm{Bord}(xa)$ est le plus long préfixe de x qui est dans l'ensemble* $\{\mathrm{Bord}(x)a, \mathrm{Bord}^2(x)a, \ldots, \mathrm{Bord}^k(x)a, \varepsilon\}$.

Preuve. (1) Un bord de $\mathrm{Bord}(x)$ est aussi un bord de x, donc les mots $\mathrm{Bord}(x)$, $\mathrm{Bord}^2(x)$, …, $\mathrm{Bord}^k(x)$ sont tous des bords de x. Réciproquement, soit z un bord de x. Ou bien $z = \mathrm{Bord}(x)$, ou alors z est un bord de $\mathrm{Bord}(x)$. Par récurrence, z est un des mots $\mathrm{Bord}(x)$, $\mathrm{Bord}^2(x)$, …, $\mathrm{Bord}^k(x)$.

(2) Soit z un bord de xa. Si $z \neq \varepsilon$, alors $z = z'a$, où z' est un bord de x. Donc, par (1), z est un des mots de l'ensemble

$$B = \{\mathrm{Bord}(x)a, \mathrm{Bord}^2(x)a, \ldots, \mathrm{Bord}^k(x)a, \varepsilon\}$$

Réciproquement, tout mot de B est suffixe de xa, donc est un bord de xa s'il est préfixe de xa. ∎

On déduit de la proposition la caractérisation suivante du plus long bord :

Corollaire 1.4. *Soit x un mot non vide et soit a une lettre. Alors*

$$\mathrm{Bord}(xa) = \begin{cases} \mathrm{Bord}(x)a & \text{si } \mathrm{Bord}(x)a \text{ est préfixe de } x, \\ \mathrm{Bord}(\mathrm{Bord}(x)a) & \text{sinon.} \end{cases}$$

Preuve. Si $\mathrm{Bord}(x)a$ est préfixe de x, alors $\mathrm{Bord}(xa) = \mathrm{Bord}(x)a$. Dans le cas contraire, posons $y = \mathrm{Bord}(x)$. Alors $\mathrm{Bord}(xa)$ est préfixe de y, et par le (2) de la proposition 1.3, $\mathrm{Bord}(xa)$ est le plus long préfixe de x, donc de y, dans l'ensemble $\{\mathrm{Bord}(y)a, \mathrm{Bord}^2(y)a, \ldots, \mathrm{Bord}^{k-1}(y)a, \varepsilon\}$. A nouveau par 1.3(2), cela signifie que $\mathrm{Bord}(xa) = \mathrm{Bord}(ya)$. ∎

Voici quelques exemples :

$x = ababb$	$\mathrm{Bord}(x) = \varepsilon$	$\mathrm{Bord}(xa) = \mathrm{Bord}(x)a$
$x = babbaa$	$\mathrm{Bord}(x) = \varepsilon$	$\mathrm{Bord}(xa) = \mathrm{Bord}(a) = \varepsilon$
$x = abaaab$	$\mathrm{Bord}(x) = ab$	$\mathrm{Bord}(xa) = \mathrm{Bord}(x)a = aba$
$x = abbaab$	$\mathrm{Bord}(x) = ab$	$\mathrm{Bord}(xa) = \mathrm{Bord}(aba) = a$

Un autre corollaire de la proposition 1.3 indique comment calculer efficacement la fonction β. On a en effet :

Corollaire 1.5. *Soit x un mot de longueur m. Pour $j = 0, \ldots, m-1$, on a*

$$\beta(1+j) = 1 + \beta^k(j)$$

où $k \geq 1$ est le plus petit entier tel que l'une des deux conditions suivantes est vérifiée

(*i*) $1 + \beta^k(j) = 0$;
(*ii*) $1 + \beta^k(j) \neq 0$ *et* $x_{1+\beta^k(j)} = x_{1+j}$. ∎

Ce corollaire se traduit en une procédure qui calcule la fonction β pour un mot x, sous la forme d'un tableau :

```
procédure BORDS-MAXIMAUX(x, β);
  β[0] := −1;
  pour j de 1 à m faire
    i := β[j − 1];
    tantque i ≥ 0 etalors x[j] ≠ x[i + 1] faire i := β[i] fintantque;
    β[j] := i + 1
  finpour.
```

Une procédure tout à fait semblable calcule la fonction de suppléance :

```
procédure SUPPLÉANCE(x, s);
  s[1] := 0;
  pour j de 1 à m − 1 faire
    i := s[j];
    tantque i > 0 etalors x[j] ≠ x[i] faire i := s[i] fintantque;
    s[j + 1] := i + 1
  finpour.
```

Une preuve analogue à celle de la proposition 1.2 montre que le calcul de la fonction β d'un motif de longueur m se fait en $2m-3$ comparaisons de caractères.

Corollaire 1.6. *La recherche d'une occurrence d'un motif x de longueur m dans un texte t de longueur n par l'algorithme de Morris et Pratt demande au plus $2(n+m)-4$ comparaisons de caractères.* ∎

10.1.4 L'algorithme de Knuth, Morris et Pratt

L'algorithme de Knuth, Morris et Pratt que nous présentons maintenant est une amélioration de l'algorithme précédent, basée sur l'élimination de situations qu'il est inutile d'examiner.

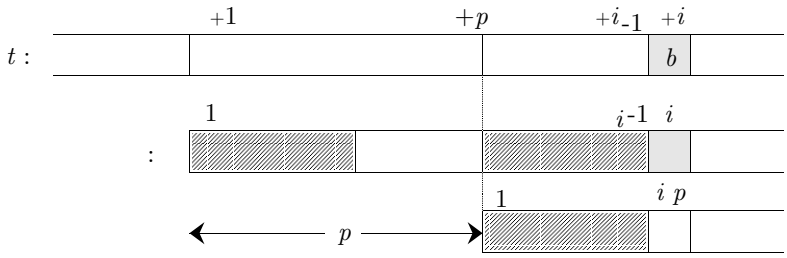


Figure 1.5: *Lorsque $b \neq a$, le décalage est inutile si $c = a$.*

Revenons à la configuration où une recherche du motif $x = x_1 \cdots x_m$ dans un texte $t = t_1 \cdots t_n$ a échoué parce que

$$x_1 \cdots x_{i-1} = t_{k+1} \cdots t_{k+i-1} \qquad \text{et} \qquad x_i \neq t_{k+i}$$

Le décalage proposé par l'algorithme de Morris et Pratt est déterminé par la longueur du bord $\mathrm{Bord}(x_1 \cdots x_{i-1})$; la nouvelle position du motif débute à la lettre d'indice $p+1$, avec $p = i - 1 - \beta(i-1)$ (voir figure 1.5). Ce décalage n'est utile que si l'on est assuré de pouvoir réellement progresser, c'est-à-dire si la lettre t_{k+i} est égale à la lettre de x qui va lui être comparée, à savoir la lettre x_{i-p} ; sinon, il faut chercher un autre bord. Or, cette condition ne peut pas être traduite en une condition sur le mot x seul, donc ne peut pas être incluse dans le prétraitement de x. Mais on peut la remplacer par une condition plus faible et exiger de ne pas se retrouver dans la même situation que précédemment, à savoir que

$$x_{i-p} \neq x_i.$$

C'est cette condition supplémentaire qui est testée dans l'algorithme de Knuth, Morris et Pratt. Pour la mettre en œuvre, on définit une fonction analogue à la fonction β de Morris et Pratt, et qui va tenir compte de cette condition. Auparavant, introduisons une définition.

Soit $x = x_1 \cdots x_m$. Deux préfixes u et v de x sont *disjoints* (dans x) si $x_{1+|u|} \neq x_{1+|v|}$ ou si l'un des deux mots est x tout entier. Le préfixe u est un *bord disjoint* du préfixe v si u est un bord de v, et si u et v sont des préfixes disjoints de x. Si v possède un bord disjoint dans x, on note $\mathrm{DBord}(v)$ le plus long bord disjoint de v, appelé *bord disjoint maximal* de v. Contrairement à la notion de bord, le concept de bord disjoint n'est défini que sur les préfixes d'un mot x.

Exemple. Soit $x = abcababcac$. Le préfixe $v = abcababca$ de x possède les trois bords ε, a, $abca$ qui sont tous les trois disjoints de v. Les préfixes ab et $abcabab$ de x ne sont pas disjoints car ils sont tous les deux suivis de la lettre c. Le préfixe $w = abcababc$ enfin a les deux bords ε et abc, dont aucun n'est disjoint de w; il n'a donc pas de bord maximal disjoint.

On définit la fonction

$$\gamma = \gamma_x : \{0, \ldots, m\} \to \{-1, \ldots, m-1\}$$

par $\gamma(0) = -1$ et, pour $j = 1, \ldots, m$,

$$\gamma(j) = \begin{cases} |\mathrm{DBord}(x_1 \cdots x_j)| & \text{si } x_1 \cdots x_j \text{ a un bord disjoint dans } x; \\ -1 & \text{sinon.} \end{cases}$$

Exemple. Voici les fonctions β et γ tabulées pour le motif $x = abcababcac$. Leur comparaison illustre le gain que l'on peut attendre de l'usage de γ à la place de β.

		a	b	c	a	b	a	b	c	a	c
j	0	1	2	3	4	5	6	7	8	9	10
$\beta(j)$	-1	0	0	0	1	2	1	2	3	4	0
$\gamma(j)$	-1	0	0	-1	0	2	0	0	-1	4	0

Soit v un préfixe du mot x. Les bords de v sont, par la proposition 1.3, les mots $\mathrm{Bord}(v), \mathrm{Bord}^2(v), \ldots, \mathrm{Bord}^k(v), \varepsilon$, où k est le plus grand entier tel que $\mathrm{Bord}^k(v) \neq \varepsilon$. Les longueurs des bords de v sont donc les nombres $\beta(j)$, $\beta^2(j)$, …, $\beta^k(j)$, 0, avec $j = |v|$. Il en résulte que si $\gamma(j) \neq -1$, alors $\gamma(j) = \beta^d(j)$, où d est le plus petit entier tel que $\mathrm{Bord}^d(v)$ est disjoint de v.

Proposition 1.7. *Soit $x = x_1 \cdots x_m$ un mot non vide; la fonction $\gamma = \gamma_x$ vérifie, pour $j \geq 1$*

$$\gamma(j) = \begin{cases} \beta(j) & \text{si } j = m \text{ ou } x_{1+j} \neq x_{1+\beta(j)}, \\ \gamma(\beta(j)) & \text{sinon.} \end{cases}$$

Preuve. Soit $j \geq 1$. Si $j = m$, alors $\gamma(m) = \beta(m)$. Supposons donc $j < m$, et soit $x_1 \cdots x_i$ le bord maximal de $x_1 \cdots x_j$. On a donc $i = \beta(j)$. Si $x_{1+i} \neq x_{1+j}$, alors $\gamma(j) = \beta(j)$. Si $x_{1+i} = x_{1+j}$, les bords disjoints de $x_1 \cdots x_i$ sont exactement les bords disjoints de $x_1 \cdots x_j$ car si $x_1 \cdots x_k$ est un tel bord, on a $x_{1+k} \neq x_{1+j}$ si et seulement si $x_{1+k} \neq x_{1+i}$, donc $\gamma(j) = \gamma(i)$. ∎

Cette proposition permet de calculer γ à partir de la fonction β. Réciproquement, β peut s'exprimer en fonction de γ, et la combinaison de ces deux relations permet d'évaluer γ sans calculer préalablement la fonction β.

Proposition 1.8. *Soit x un mot de longueur m. Pour $j = 0, \ldots, m-1$, on a*

$$\beta(1+j) = 1 + \gamma^k(\beta(j))$$

où $k \geq 0$ est le plus petit entier tel que l'une des deux conditions suivantes est vérifiée :

(i) $1 + \gamma^k(\beta(j)) = 0$;

(ii) $1 + \gamma^k(\beta(j)) \neq 0$ *et* $x_{1+\gamma^k(\beta(j))} = x_{1+j}$.

Preuve. Posons $i = \beta(j)$ et $y = x_1 \cdots x_i$, et soit $a = x_{1+i}$, et $b = x_{1+j}$. Si $a = b$, alors $\beta(1+j) = 1 + \beta(j)$, et la formule est vraie avec $k = 0$. Sinon, $\beta(1+j)$ est soit nul, soit l'un des nombres $1 + \beta^r(j)$. Or, parmi les mots y, $\mathrm{Bord}(y), \ldots$, il suffit d'examiner ceux qui sont suivis d'une lettre autre que b. Il suffit donc de chercher le bord maximal *disjoint* de y, dont la longueur est $\gamma(i)$. ∎

En vertu des propositions 1.7 et 1.8, on obtient la procédure que voici pour le calcul de γ. La boucle *tantque* calcule en fait $\beta(j)$:

```
procédure BORDS-DISJOINTS-MAXIMAUX(x, γ);
  γ[0] := −1; i := −1;
  pour j de 1 à m faire          {ici i = β(j − 1)}
    tantque i ≥ 0 etalors x[j] ≠ x[i + 1] faire i := γ[i] fintantque;
    i := i + 1;                  {ici i = β(j)}
    si x[1 + j] ≠ x[1 + i] alors γ[j] := i sinon γ[j] := γ[i]
  finpour.
```

Introduisons une *deuxième fonction de suppléance* r définie par

$$r(i) = 1 + \gamma(i-1)$$

On peut alors calculer r par

```
procédure DEUXIÈME-SUPPLÉANCE(x, r);
  r[1] := 0; i := 0;
  pour j de 1 à m − 1 faire
    tantque i > 0 etalors x[j] ≠ x[i] faire i := r[i] fintantque;
    i := i + 1;
    si x[1 + j] ≠ x[i] alors r[1 + j] := i sinon r[1 + j] := r[i]
  finpour.
```

Avec cette deuxième fonction de suppléance, l'algorithme de Knuth, Morris et Pratt s'écrit exactement comme l'algorithme de Morris et Pratt. La seule modification est le remplacement de s par r. Nous changeons très légèrement sa structure, en remplaçant le double test sur i et j par deux boucles *tantque* imbriquées, ceci pour pouvoir mettre en évidence la notion de délai entre l'examen de deux caractères :

```
procédure KNUTH-MORRIS-PRATT(x, t);
  i := 1; j := 1;
  tantque i ≤ m et j ≤ n faire
    tantque i > 0 etalors t[j] ≠ x[i] faire i := r[i] fintantque;
    i := i + 1; j := j + 1
  fintantque;
  si i > m alors
    occurrence de x à la position j − m
  sinon
    pas d'occurrence de x dans t
  finsi.
```

Revenons sur un exemple précédent, où l'on cherche le motif $x = abacabac$ dans le texte $t = babacacabacaab$. Les deux fonctions de suppléance s et r sont les suivantes :

	a	b	a	c	a	b	a	c
	1	2	3	4	5	6	7	8
s	0	1	1	2	1	2	3	4
r	0	1	0	2	0	1	0	2

La figure 1.6 montre les décalages successifs du motif. Les différences sont constatées aux positions sombres du texte. Le nombre total de comparaisons est 16.

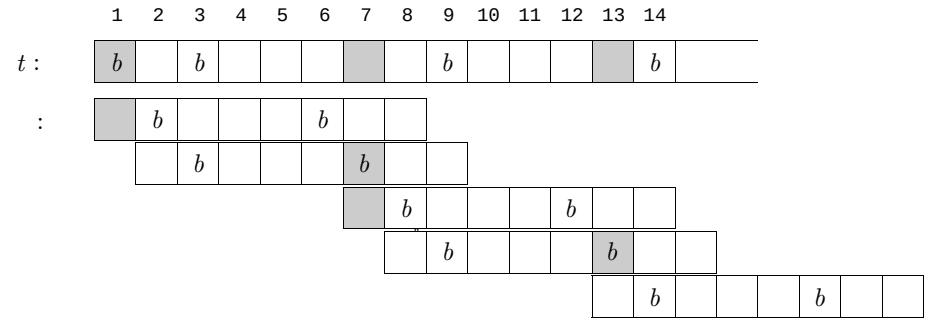


Figure 1.6: *Décalages successifs du motif.*

Le tableau donne, pour chaque lettre, la suite des valeurs que prend l'indice i dans l'algorithme :

j	1	2	3	4	5	6	7	8	9	10	11	12	13	14
	b	a	b	a	c	a	c	a	b	a	c	a	a	b
i	1	1	2	3	4	5	6	1	2	3	4	5	6	2
	0						1						1	
							0							

Lorsque $j = 7$ et $i = 6$, on a $t_j \neq x_i$, et on compare cette fois-ci t_7 directement à la lettre x_1. La différence entre les deux fonctions de suppléance apparaît bien si on trace leur graphe (figure 1.7). Il est clair que l'algorithme de Knuth, Morris et

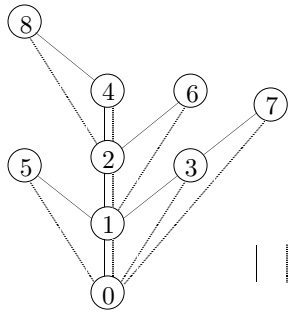


Figure 1.7: *Les graphes des deux fonctions de suppléance.*

Pratt est plus efficace que l'algorithme de Morris et Pratt, même si, dans le cas le plus défavorable, leur complexité est la même. Une différence de comportement notable entre ces deux algorithmes concerne le *délai*, c'est-à-dire le nombre maximum de comparaisons de caractères faites sur *un* caractère du texte à analyser. Dans l'algorithme ci-dessus, c'est le nombre de tours effectués dans la boucle *tant-que* interne. Le délai est le temps que l'on doit attendre avant de pouvoir passer au caractère suivant du texte t, et il mesure donc jusqu'à quel point l'algorithme est différent d'un algorithme *en temps réel*, c'est-à-dire est capable de traiter un symbole par unité de temps. Il a été prouvé que le délai de l'algorithme de Morris et Pratt peut atteindre la longueur m du motif, alors que pour Knuth, Morris et Pratt, il ne dépasse jamais $1 + \log_\phi m$, où $\phi = (1 + \sqrt{5})/2$. En ce sens aussi, l'algorithme de Knuth, Morris et Pratt, sans être plus difficile à programmer, est plus efficace.

10.1.5 L'automate des occurrences

Dans cette section, nous montrons comment l'algorithme de Knuth, Morris et Pratt s'interprète en termes d'automates finis. Soit A l'alphabet sur lequel sont écrits le texte et le motif. Soit $x \in A^*$ le motif dont on cherche à déterminer les

occurrences dans le texte t. Comme x a une occurrence dans t si et seulement si $t \in A^*xA^*$, déterminer les occurrences de x dans t équivaut à trouver les préfixes de t qui appartiennent au langage (rationnel) A^*x. Pour cela, il suffit de construire l'automate reconnaissant A^*x, et de lui faire lire le texte t. Or, un automate reconnaissant A^*x est vite construit. C'est l'automate

$$\mathcal{A} = (P, \varepsilon, x, \mathcal{F})$$

où P est l'ensemble des préfixes de x, l'état initial est le mot vide, l'unique état final est le motif x, et dont les flèches sont

$$\mathcal{F} = \{(\varepsilon, a, \varepsilon) \mid a \in A\} \cup \{(p, a, pa) \mid p, pa \in P, a \in A\}$$

Exemple. Pour $A = \{a, b, c\}$, et pour $x = abcababcac$, l'automate $\mathcal{A}$ est donné dans la figure 1.8, où les états sont représentés par leur longueur.

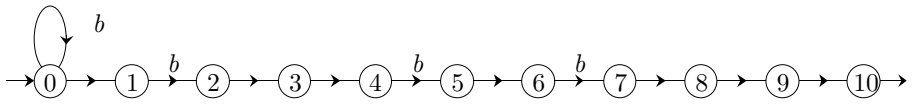


Figure 1.8: *Automate reconnaissant le langage $A^*abcababcac$.*

Dans la mesure où cet automate n'est pas déterministe, il n'est pas facilement exploitable. Nous allons voir que l'automate minimal déterministe reconnaissant A^*x n'est pas difficile à calculer et a autant d'états que l'automate ci-dessus, à savoir $1 + m$, où m est la longueur de x. Pour le caractériser, nous considérons la fonction f_x qui à tout mot u associe

$$f_x(u) = \text{ le plus long suffixe de } u \text{ qui est préfixe de } x$$

Par exemple, pour $x = abcababcac$, on a $f_x(abacababc) = abc$; si p est préfixe de x, on a évidemment $f_x(p) = p$.

Proposition 1.9. *Soit x un mot et soit P l'ensemble de ses préfixes. L'automate minimal reconnaissant A^*x est l'automate déterministe $\mathcal{A}(x) = (P, \varepsilon, x)$ dont la fonction de transition est définie par $p \cdot a = f_x(pa)$.*

Preuve. Nous allons vérifier que pour tout $u \in A^*$,

$$u^{-1}(A^*x) = f_x(u)^{-1}(A^*x)$$

Ceci prouve que les états de l'automate minimal s'identifient aux préfixes de x, et que la fonction de transition est bien celle indiquée. Soit donc $u \in A^*$, et soit u' tel que $u = u' f_x(u)$. On a

$$u^{-1}(A^*x) = f_x(u)^{-1}u'^{-1}(A^*x) \supset f_x(u)^{-1}(A^*x)$$

Pour prouver l'inclusion réciproque, soit $w \in u^{-1}(A^*x)$. Alors $uw \in A^*x$, et il existe donc un mot v tel que $uw = vx$. Si x est suffixe de w, alors $w \in A^*x$, donc $f_x(u)w \in A^*x$ et $w \in f_x(u)^{-1}(A^*x)$. Si en revanche w est suffixe de x, appelons z le mot tel que $x = zw$. Alors on a aussi $u = vz$, donc z est suffixe de u et préfixe de x. Par définition de $f_x(u)$, il existe y tel que $f_x(u) = yz$. Mais alors $f_x(u)w = yzw = yx$, montrant que $w \in f_x(u)^{-1}(A^*x)$. Ceci prouve l'inclusion et achève la démonstration. ∎

L'automate $\mathcal{A}(x)$ est l'*automate des occurrences*. On obtient immédiatement l'algorithme suivant de recherche de motifs, où le texte $t = t_1 \cdots t_n$ est de longueur n. Dans la mesure où l'automate des occurrences $\mathcal{A}(x)$ est disponible et rangé dans une table (de taille $O(|A|(|x|+1))$), le calcul de l'état suivant se fait en temps constant, et le temps d'exécution de l'algorithme est $O(n)$ pour un texte de longueur n.

```
procédure RECHERCHE-AUTOMATE(x, t);
  q :=état initial;
  pour j de 1 à n faire
    q := q · t[j];
    si q est état final alors j est une fin d'occurrence finsi
  finpour.
```

L'automate $\mathcal{A}(x)$ pour $x = abcababcac$ est donné dans la figure 1.9. Les états sont représentés par leurs longueurs. L'expression donnée ci-dessous fait le lien avec

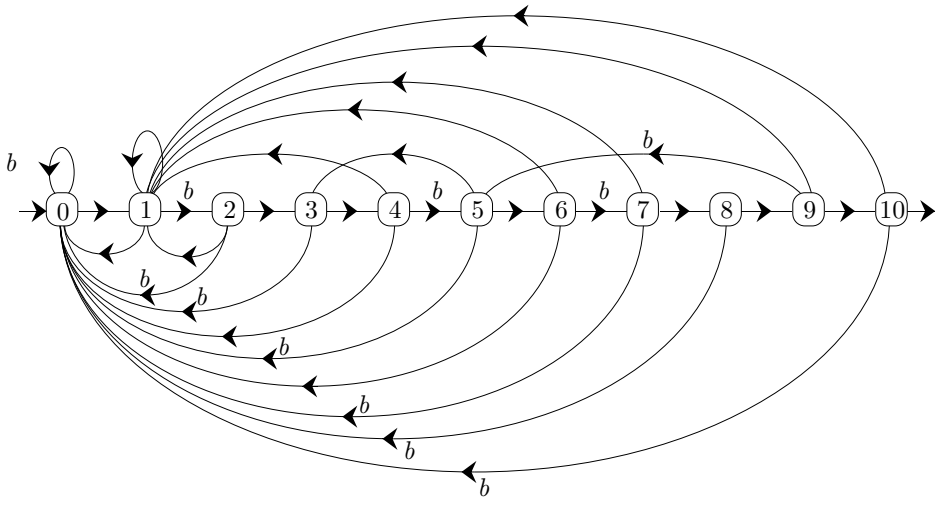


Figure 1.9: *Automate déterministe* $\mathcal{A}(abcababcac)$.

l'algorithme de Morris et Pratt.

Proposition 1.10. *La fonction de transition de l'automate des occurrences $\mathcal{A}(x)$ vérifie, pour tout préfixe p non vide de x et toute lettre a,*

$$p \cdot a = \begin{cases} pa & \textit{si } pa \textit{ est préfixe de } x\,; \\ \mathrm{Bord}(pa) & \textit{sinon.} \end{cases}$$

Preuve. Si pa est préfixe de x, alors $f_x(pa) = pa$; sinon, $f_x(pa)$ est le plus long suffixe de pa qui est préfixe de x, donc préfixe de pa, c'est-à-dire Bord(pa). ∎

Le calcul de la fonction de transition de l'automate des occurrences est en fait assez facile, et est réalisable en temps linéaire. Ceci résulte du corollaire suivant :

Corollaire 1.11. *La fonction de transition de l'automate des occurrences $\mathcal{A}(x)$ vérifie, pour tout préfixe p non vide de x et toute lettre a*

$$p \cdot a = \begin{cases} pa & \textit{si } pa \textit{ est préfixe de } x\,; \\ \mathrm{Bord}(p) \cdot a & \textit{sinon.} \end{cases}$$

Preuve. L'énoncé résulte immédiatement de la proposition si pa est préfixe de x. Si pa n'est pas préfixe de x, alors en vertu de 1.4

$$p \cdot a = \mathrm{Bord}(pa) = \begin{cases} \mathrm{Bord}(p)a & \text{si Bord}(p)a \text{ est préfixe de } p, \\ \mathrm{Bord}(\mathrm{Bord}(p)a) & \text{sinon,} \end{cases}$$

donc $\mathrm{Bord}(pa) = \mathrm{Bord}(p) \cdot a$. ∎

Corollaire 1.12. *La fonction de transition de l'automate des occurrences $\mathcal{A}(x)$ vérifie, pour tout préfixe p de x et toute lettre a*

$$p \cdot a = \begin{cases} pa & \textit{si } pa \textit{ est préfixe de } x\,; \\ \mathrm{DBord}(p) \cdot a & \textit{si } \mathrm{DBord}(p) \textit{ existe;} \\ \varepsilon & \textit{sinon.} \end{cases}$$

Preuve. Si $p \neq \varepsilon$ et pa n'est pas préfixe de x, alors $p.a = \mathrm{Bord}(pa)$. Si $\mathrm{Bord}(pa) \neq \varepsilon$, alors $\mathrm{Bord}(pa) = qa$, où q est le plus long bord de p tel que qa est préfixe de x. Comme pa n'est pas préfixe de x, le mot q est *un* bord disjoint de p; d'où la formule par récurrence. ∎

Exemple. Pour l'automate de la figure 1.9, les fonctions β et γ prennent les valeurs suivantes :

		a	b	c	a	b	a	b	c	a	c
	0	1	2	3	4	5	6	7	8	9	10
β	-1	0	0	0	1	2	1	2	3	4	0
γ	-1	0	0	-1	0	2	0	0	-1	4	0

On en déduit par exemple que $5 \cdot b = 2 \cdot b = 0$, et de façon similaire $9 \cdot a = 4 \cdot a (= 1 \cdot a) = 0 \cdot a = 1$, selon que l'on utilise la formule du premier ou du deuxième corollaire. Enfin, $3 \cdot b = 3 \cdot c = 0$.

La recherche d'un motif x avec un automate fini se fait en *temps réel*. La fonction de transition de l'automate se calcule, à partir de β ou de γ, en temps linéaire, c'est-à-dire en temps $O(|x| \cdot |A|)$, où A est l'alphabet de base. L'inconvénient majeur est l'encombrement de l'automate : sa taille, qui est également $O(|A|(|x|+1))$, devient prohibitive si A est par exemple l'alphabet des 256 caractères Ascii étendus.

10.1.6 L'algorithme de Simon

I. Simon a proposé un algorithme qui est un compromis entre l'algorithme de Knuth, Morris et Pratt, et l'implémentation par automate. Simon part de l'automate minimal $\mathcal{A}(x)$ associé à un motif x de longueur m, et remarque qu'il n'y a que peu de flèches «significatives» : il y a d'abord les flèches «avant», qui font passer d'un état i à l'état $i+1$, et les flèches «arrière», qui font passer d'un état i à un état $j < i$, avec $j \neq 0$. Les autres flèches mènent à l'état 0, et ne sont pas «significatives». Nous prouverons que les flèches significatives (nous dirons actives) sont en nombre au plus $2m$, et si l'on ne stocke que celles-ci, on est ramené à un algorithme en place linéaire, indépendamment de la taille de l'alphabet. En contrepartie, la recherche de la «bonne» flèche pour effectuer une transition ne se fait plus en temps constant (donc l'algorithme n'est pas en temps réel), mais le rangement des transitions peut être organisé de manière à rendre cette recherche toujours au moins aussi efficace que dans l'algorithme de Knuth, Morris et Pratt.

Soit donc $x \in A^*$ un motif de longueur m, et soit $\mathcal{A}(x) = (P, \varepsilon, x)$ l'automate minimal reconnaissant A^*x. On identifiera souvent un état de P, c'est-à-dire un préfixe p de x, à sa longueur. Une flèche (p, a, q) de l'automate est une flèche *active* si $q \neq \varepsilon$, elle est *passive* si $q = \varepsilon$. Une flèche active (p, a, q) est une flèche *avant* si $q = pa$, c'est une flèche *arrière* si $q \neq pa$ (et $q \neq \varepsilon$).

Exemple. La figure 1.10 donne l'automate de la figure 1.9, où l'on n'a conservé que les flèches actives. Il y a 9 flèches arrière.

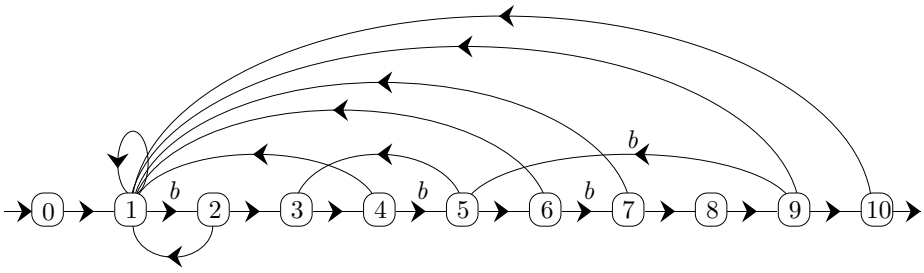


Figure 1.10: *Automate $\mathcal{A}(abcababcac)$, sans ses flèches passives.*

Le motif x étant de longueur m, l'automate $\mathcal{A}(x)$ a m flèches avant. Nous allons prouver qu'il a au plus m flèches arrière. Pour cela, nous avons besoin d'une définition qui a par ailleurs son intérêt propre. Soit $w = a_1 \cdots a_n$ un mot, avec $a_1, \ldots, a_n$ des lettres. Une *période* de w est un entier $p > 0$ tel que

$$a_i = a_{p+i} \quad \text{pour tout } i = 1, \ldots, n-p$$

Notons que $|w|$ est toujours une période de w. Par exemple, le mot $w = abcababca$, de longueur 9, a les périodes 9, 8, 5. On peut voir une période comme un «décalage» qui conserve la coïncidence des lettres qui se superposent. Plus précisément, on a le lemme suivant :

Lemme 1.13. *Soit w un mot de longueur n, et soit $p > 0$. Alors p est une période de w si et seulement si w possède un bord de longueur $n - p$.*

Preuve. Soit $w = a_1 \cdots a_n$. Alors p est une période si et seulement si

$$a_1 \cdots a_{n-p} = a_{1+p} \cdots a_n,$$

donc si et seulement si le préfixe de longueur $n - p$ de w est aussi suffixe de w. ∎

Notons pér(w) la plus petite période de w. Il résulte du lemme 1.13 que

$$\text{pér}(w) + \beta(w) = |w| \tag{1.1}$$

où $\beta(w)$ est la longueur du plus long bord de w. Revenons à l'automate $\mathcal{A}(x)$.

Proposition 1.14. *L'automate $\mathcal{A}(x)$ a au plus $|x|$ flèches arrière.*

Preuve. Nous allons d'abord prouver l'assertion que voici : si (p, a, q) et (p', a', q') sont deux flèches arrière distinctes, alors pér$(pa) \neq$ pér$(p'a')$.

Soient en effet (p, a, q) et (p', a', q') deux flèches arrière. Par la proposition 1.10, on a $q = \text{Bord}(pa)$, $q' = \text{Bord}(p'a')$, et $q \neq \varepsilon$, $q' \neq \varepsilon$. Supposons, en raisonnant par l'absurde, que pa et $p'a'$ ont même période, disons $t =$ pér(pa) et, pour fixer les idées, supposons $|p| \leq |p'|$. Si $k = |pa|$, on a $t < k$ par l'équation 1.1. Notons b la lettre d'indice k dans $p'a'$. Alors $a \neq b$. Ceci est clair si $|p'| > |p|$ parce que p' est préfixe de x, alors que pa ne l'est pas. C'est vrai aussi si $|p'| = |p|$ parce qu'alors $b = a' \neq a$. Comme t est une période de pa, les lettres d'indice k et $k - t$ de pa sont les mêmes. Comme p est préfixe de x, la lettre d'indice $k - t$ de pa est x_{k-t}, donc $x_{k-t} = a$. Or t est aussi une période de $p'a'$, et donc les lettres d'indice k et $k - t$ de $p'a'$ sont les mêmes. On a donc aussi $x_{k-t} = b$, ce qui est impossible puisque $a \neq b$. Ceci prouve l'assertion.

Il résulte de l'assertion que l'application qui à une flèche arrière $(p, a, \text{Bord}(pa))$ associe pér(pa) est injective. Or pér$(pa) = |pa| - \text{Bord}(pa) < |pa|$ d'après l'équation 1.1, donc $1 \leq$ pér$(pa) \leq |x|$, ce qui montre la proposition. ∎

L'*implémentation de Simon* de l'automate $\mathcal{A}(x)$ consiste à associer, à chaque état p, la liste des flèches avant et arrière issues de p, ordonnées par numéro décroissant d'état d'arrivée. Chaque flèche $u = (p, a, q)$, pour p fixé, est représentée par un couple $(\mathit{lettre}(u), \mathit{état}(u)) = (a, q)$. Avec cette structure de données, le calcul de l'état suivant se fait par la fonction :

```
fonction ETAT-SUIVANT-PAR-SIMON(p, a);
  u :=tête-liste(p);
  tantque u ≠ vide faire
    si lettre(u) = a alors retourner état(u)
    sinon u := suivant(u)
  fintantque;
  retourner(0).
```

et l'algorithme de recherche du motif x, de longueur m, dans un texte t de longueur n est, comme pour tout automate (voir la procédure RECHERCHE-AUTOMATE) :

```
procédure SIMON(x, t);
  p := 0;
  pour j de 1 à n faire
    p :=ETAT-SUIVANT-PAR-SIMON(p, t[j]);
    si p = m alors j est une fin d'occurrence
  finpour.
```

Exemple. L'implémentation de l'automate $\mathcal{A}(abcababcac)$ est représentée dans la figure 1.11 ; les préfixes sont représentés par leur longueur.

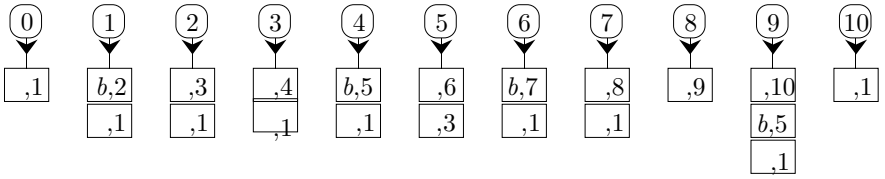


Figure 1.11: *Implémentation de l'automate* $\mathcal{A}(abcababcac)$.

Proposition 1.15. *L'algorithme de Simon calcule les occurrences d'un motif x dans un texte t en un nombre de comparaisons de caractères inférieur ou égal à celui de l'algorithme de Knuth, Morris et Pratt.*

Preuve. Soit p un état, et soit a une lettre. Le nombre de comparaisons faites par l'algorithme de Simon pour déterminer l'état suivant est égal au nombre de comparaisons faites dans la liste associée à l'état p. La première comparaison concerne la flèche avant, les suivantes des flèches arrière ; si toutes les comparaisons sont négatives, c'est une flèche passive qui est utilisée.

Soit $(p, b, p \cdot b)$ une flèche arrière. Alors par le corollaire 1.11, $p \cdot b = qb$, et q est un bord disjoint de p. En d'autres termes, si

$$(p, a_1, q_1a_1), \ldots, (p, a_k, q_ka_k)$$

est la suite ordonnée des flèches arrière, les états $q_1, \ldots, q_k$ sont des bords disjoints de p, et mutuellement disjoints. Ils figurent donc dans l'ensemble

$$\{\mathrm{DBord}(p), \mathrm{DBord}^2(p), \ldots\}$$

Comme les états sont rangés en ordre décroissant, ces bords disjoints sont calculés successivement dans l'algorithme de Knuth, Morris et Pratt, alors que l'algorithme de Simon n'en sélectionne qu'un sous-ensemble. ∎

Il reste à préciser comment réaliser, en temps linéaire, l'implémentation de Simon, c'est-à-dire le calcul de la suite ordonnée des flèches actives pour chaque état. Notons $F(p)$ la suite ordonnée des flèches actives issues de l'état p ; chaque flèche est représentée par le couple (lettre, état d'arrivée), comme dans la figure 1.11.

Soit $x = x_1 \cdots x_m$ un motif. Pour l'état initial ε de l'automate $\mathcal{A}(x)$, la suite $F(\varepsilon)$ des flèches actives est réduite à l'élément $(x_1, 1)$. Soit p un autre état, et soit $i = |p|$ sa longueur. La suite $F(p)$ contient tout d'abord la flèche $(x_{i+1}, i+1)$, sauf lorsque $i = m$. Pour déterminer les autres flèches, nous utilisons le corollaire 1.12.

Si $\mathrm{DBord}(p)$ existe, c'est-à-dire si $\gamma(i) \neq -1$, alors on a $p \cdot a = \mathrm{DBord}(p) \cdot a$ pour toute lettre $a \neq x_{i+1}$. Ainsi, la suite $F(p)$ se prolonge par la suite $F(\mathrm{DBord}(p))$ des flèches actives de $\mathrm{DBord}(p)$, purgée de la flèche dont la lettre est x_i, si cette flèche y figure.

Si en revanche $\mathrm{DBord}(p)$ n'existe pas, la suite $F(p)$ est réduite à son premier élément.

Il est clair que la copie d'une liste, avec purge éventuelle d'un élément, se fait en temps linéaire. L'implémentation de Simon se calcule donc en temps linéaire, puisque la fonction γ se calcule, elle aussi, en temps linéaire.

Exemple. La fonction γ_x pour $x = abcababcac$ a déjà été donnée plus haut. Ses valeurs sont :

		a	b	c	a	b	a	b	c	a	c
j	0	1	2	3	4	5	6	7	8	9	10
$\gamma(j)$	-1	0	0	-1	0	2	0	0	-1	4	0

La liste $F(8)$ est réduite à $(a, 9)$ parce que $\gamma(8) = -1$; la liste $F(9)$ est la concaténation de $(c, 10)$ et de $F(4)$; enfin, la liste $F(5)$ est composée de $(a, 6)$ et de la liste $F(2)$, purgée de son élément $(a, 1)$, donc réduite à $(c, 3)$.

10.2 L'algorithme de Boyer et Moore

L'algorithme de Boyer et Moore se rattache au schéma général des algorithmes de recherche que nous avons exposé au début de la section précédente : on compare le motif x à certains facteurs du texte t, en faisant glisser x de gauche à droite le long du texte t.

La différence principale entre l'algorithme de Boyer et Moore et les algorithmes précédents réside dans la manière de comparer le motif x aux facteurs du texte. Alors que l'algorithme naïf et l'algorithme de Knuth, Morris et Pratt comparent les lettres du motif x aux lettres du facteur de t de la gauche vers la droite, l'algorithme de Boyer et Moore les compare de la droite vers la gauche. Cette modification apparemment anodine est très rentable puisqu'en pratique, l'algorithme de Boyer et Moore est le plus rapide des algorithmes connus.

10.2.1 Algorithme de Horspool

Dans sa version la plus simple, l'algorithme (appelé algorithme de Boyer-Moore-Horspool) compare le motif $x = x_1 \cdots x_m$ à un facteur $t_{k+1} \cdots t_{k+m}$ du texte. Si une différence entre x et ce facteur est constatée, on décale le motif vers la droite de manière à faire coïncider la dernière lettre du facteur, c'est-à-dire la lettre t_{k+m}, avec son occurrence la plus à droite dans x.

Exemple. La figure 2.1 montre le fonctionnement de l'algorithme de Boyer-Moore-Horspool sur le texte $t = aabbbababacaabbaba \cdots$ et le motif $x = aababab$. Les différences sont constatées aux positions 4, 11, 18 du texte (cases sombres du

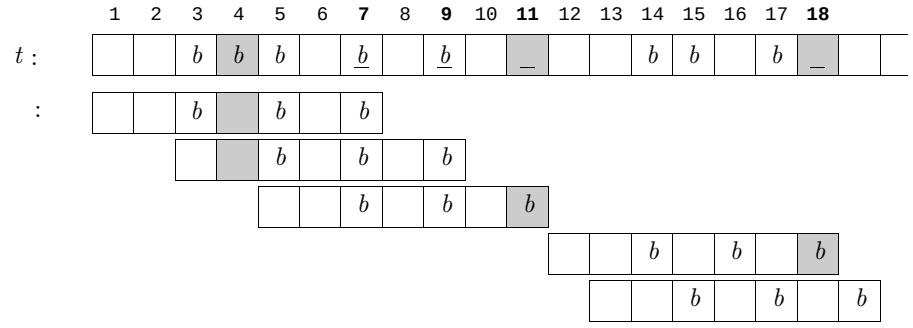


Figure 2.1: *Algorithme de Horspool.*

texte et de l'occurrence du motif). Les lettres qui déterminent le décalage sont soulignées (les indices correspondants sont gras). La première différence constatée conduit à décaler le motif de 2 vers la droite pour faire coïncider la lettre b à la cinquième position dans x avec la septième lettre du texte. La deuxième différence entraîne à nouveau un décalage de 2. La troisième différence constatée doit entraîner un décalage pour faire coïncider la lettre c avec une lettre du motif.

Comme il n'y a pas d'occurrence de la lettre c dans le motif, le motif est décalé de toute sa longueur. Le dernier décalage est d'une unité, parce qu'il y a une occurrence de a en avant-dernière position dans x.

Soit x un mot de longueur m. Pour toute lettre a de l'alphabet, soit $d(a)$ la distance entre la dernière occurrence de a dans x (la dernière place exceptée) et la dernière lettre de x. Plus précisément

$$d(a) = \begin{cases} |u| & \text{si } au \text{ est suffixe de } x,\ u \neq \varepsilon \text{ et } a \text{ ne figure pas dans } u, \\ |x| & \text{si } a \text{ ne figure pas dans } x. \end{cases}$$

La fonction d est la *fonction de dernière occurrence*. Par exemple, pour le mot $x = aababab$, la fonction de dernière occurrence vaut :

	a	b	c
d	1	2	7

L'algorithme esquissé plus haut est le suivant (x est un mot de longueur m, et t un texte de longueur n). La version que nous donnons calcule toutes les occurrences de x dans t :

```
Algorithme BOYER-MOORE-HORSPOOL(x, t);
  j := m;
  tantque j ≤ n faire
    i := m;
    tantque i > 0 etalors t_{j-m+i} = x_i faire i := i − 1 fintantque;
    si i = 0 alors
      j est une fin d'occurrence de x;
      j := j + 1
    sinon
      j := j + d[t_j]
    finsi
  fintantque.
```

Sur notre exemple, les valeurs successives que prend l'indice j dans la boucle externe sont 7, 9, 11, 18, 19. Le nombre de comparaisons de caractères n'est que 12.

Cet exemple illustre l'efficacité des algorithmes du type Boyer et Moore : en pratique, on constate que le nombre total de comparaisons est très souvent inférieur à la longueur du texte. En d'autres termes, ces algorithmes n'examinent même pas tous les caractères, par opposition à tous les algorithmes opérant de la gauche vers la droite qui, eux, examinent au moins une fois chaque caractère. En revanche, il n'est pas difficile de voir que l'algorithme est en temps $O(|x||t|)$ dans le cas le plus défavorable. Considérons le motif $x = ba^{m-1}$ dont on cherche une occurrence

dans a^n. La fonction de décalage n'apporte pas d'amélioration puisque $d(a) = 1$. On fait donc $n - m$ décalages, et chaque test demande (s'il est fait de droite à gauche) m comparaisons de caractères.

Le comportement en moyenne de l'algorithme a fait l'objet d'études expérimentales et est tout à fait excellent. On peut prouver, mais cela dépasse le cadre de ce texte, que l'algorithme est sous-linéaire en moyenne (voir les notes).

La comparaison entre x et $t_{j-m+1} \cdots t_j$ est faite de droite à gauche, mais on pourrait aussi bien la faire de gauche à droite; ce qui importe c'est le calcul du décalage en fonction de la dernière lettre du facteur.

Il reste à calculer la fonction de dernière occurrence d pour le motif x. Cela se fait très simplement comme suit :

```
procédure DERNIÈRE-OCCURRENCE (x, d);
   pour toute lettre a de A faire d[a] := m finpour;
   pour i de 1 à m − 1 faire d[x_i] := m − i finpour.
```

Bien entendu, ce calcul de d peut être inclus au début de la procédure pour l'algorithme de Boyer, Moore et Horspool.

10.2.2 Algorithme de Boyer et Moore

La fonction de dernière occurrence peut être employée différemment : alors que dans la version de Horspool, le décalage est déterminé par la dernière lettre du facteur examiné, on peut utiliser la lettre du facteur où la différence a été constatée. Ainsi, la superposition du motif $x = aababab$ au facteur $aabcbab$ conduit, dans Horspool, à un décalage de 2 à cause de la lettre b terminant le facteur, et ceci indépendamment du fait que c'est à l'occurrence de la lettre c dans le facteur que la différence a été constatée. Dans une variante de l'algorithme de Boyer et Moore (pour être historiquement exact, c'est Horspool qui a introduit une variante de l'algorithme de Boyer et Moore), c'est la lettre où la différence se manifeste qui détermine le décalage. Plus précisément, lorsqu'une coïncidence partielle du texte et du motif est constatée :

$$x_i \neq t_j, \quad x_{i+1} \cdots x_m = t_{j+1} \cdots t_{m+j-i}$$

on décale le motif pour faire coïncider la lettre t_j avec $x_{d(t_j)}$, c'est-à-dire avec la dernière occurrence de t_j dans x, puis on recommence les comparaisons sur le nouveau facteur du texte; toutefois, ce décalage n'est fait que si cela fait réellement avancer le motif, c'est-à-dire lorsque $d(t_j) > m - i$; sinon, on décale le motif de 1. En d'autres termes, la comparaison est reprise pour les nouvelles valeurs

$$j := j + \max(d(t_j), m - i + 1); \ i := m$$

Voici l'algorithme obtenu :

```
Algorithme BOYER-MOORE-SIMPLIFIÉ(x, t);
  j := m;
  tantque j ≤ n faire
    i := m;
    tantque i > 0 etalors t_j = x_i faire
      i := i − 1; j := j − 1 fintantque;
    si i = 0 alors fin d'une occurrence de x en j;
    j := j + max(d[t_j], m − i + 1);
  fintantque.
```

Exemple. Reprenons l'exemple précédent de la recherche du motif $x = aababab$ dans le texte $t = aabbbababacaabbaba\cdots$ avec l'algorithme de Boyer-Moore simplifié. La figure 2.2 montre les décalages successifs du motif. Les différences sont

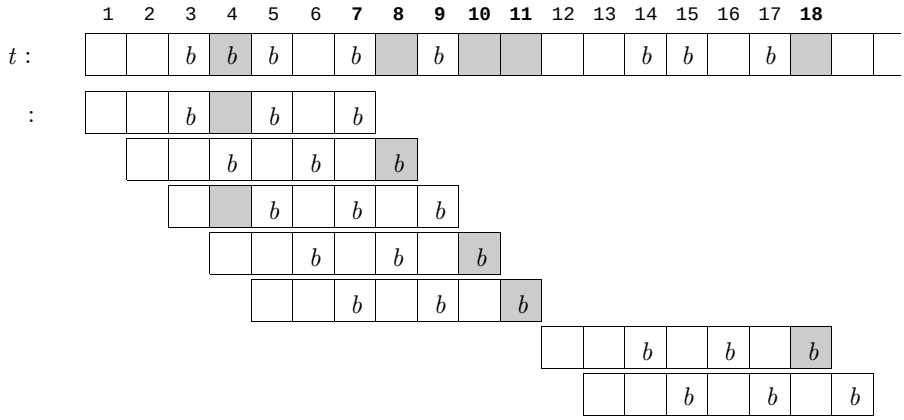


Figure 2.2: *Algorithme de Boyer-Moore simplifié.*

constatées aux positions sombres du texte. Les valeurs successives que prend l'indice j en début de la boucle *tantque* extérieure sont indiquées en gras. Le nombre total de comparaisons de caractères est 14.

La version complète de l'algorithme de Boyer et Moore fait intervenir, en plus de la fonction de dernière occurrence d, une deuxième fonction qui prend en compte le suffixe où la différence entre le motif et le texte a été constatée. La fonction de dernière occurrence est employée de la façon que nous venons d'indiquer. Avant la description formelle, donnons une description sommaire de la deuxième fonction de décalage. Supposons que la comparaison, de droite à gauche, du motif $x = x_1 \cdots x_m$ à un facteur du texte t ait mis en évidence une coïncidence avec un suffixe propre $u = x_{i+1} \cdots x_m$ de x, mais que la lettre x_i soit différente de la lettre correspondante t_j du texte. On a donc (voir figure 2.3) :

$$x_i \neq t_j, \quad x_{i+1} \cdots x_m = t_{j+1} \cdots t_{j+m-i}$$

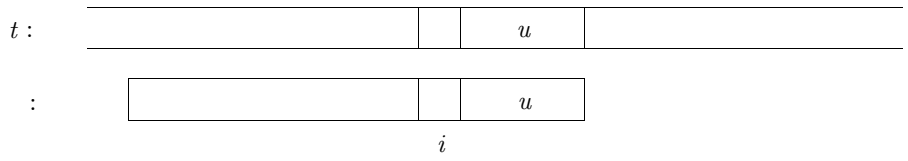


Figure 2.3: *Coïncidence partielle du motif et du texte.*

Un décalage qui tient compte de cette information va aligner x sur la dernière occurrence de u dans x qui est précédée d'une lettre différente de x_i (voir figure 2.4). Le décalage (noté $d_2(i)$ dans la figure) est la quantité dont l'indice j va

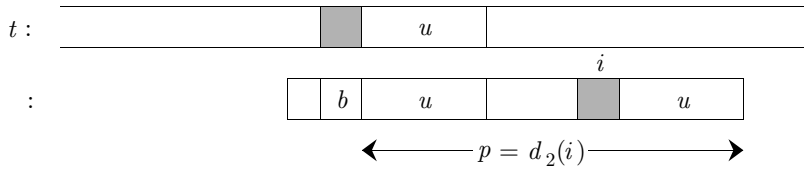


Figure 2.4: *Décalage : premier cas.*

être augmentée; il est égal à la longueur p du plus court suffixe v de x qui a u comme bord, et qui n'est pas précédé de la lettre x_i. Il se peut que x n'ait pas

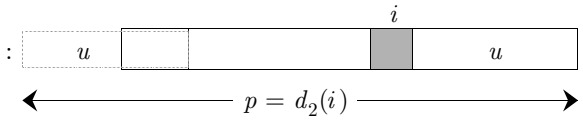


Figure 2.5: *Décalage : deuxième cas.*

d'occurrence du suffixe u précédée d'une lettre différente de x_i. Dans ce cas, on cherche le plus long suffixe de u qui est un préfixe de x; le décalage (encore noté $d_2(i)$) dépasse alors $|x|$ (voir figure 2.5). Cette situation se produit en particulier si $x = u$. Dans ce cas, $i = 0$ et $d_2(0) = |x| + |\mathrm{Bord}(x)|$.

Il est commode d'appeler, par analogie avec la notion de bord disjoint, *bord disjoint droit* d'un suffixe v de x un bord u de v tel que les suffixes u et v ne sont pas précédés de la même lettre dans x. Pour tout suffixe u de x on pose

$$V(u) = \{v \mid v \text{ est suffixe de } x, \ \ u \text{ est bord disjoint droit de } v\}$$

et

$$W(u) = \{w \mid x \text{ est suffixe de } w, \ \ u \text{ est bord de } w, \ |w| \leq |ux|\}$$

L'ensemble $V(u)$ contient les suffixes de x pour lesquels le premier type de décalage est possible, et $W(u)$ contient les mots intervenant dans le deuxième

cas; on peut clairement se limiter aux mots de longueur majorée par $|ux|$. La deuxième fonction de décalage, que nous appellerons la *fonction du bon suffixe* est traditionnellement notée d_2. Elle est définie, pour $i = 0, \ldots, m$, en posant $u = x_{i+1} \cdots x_m$ par

$$d_2(i) = \min_{v \in V(u) \cup W(u)} |v|$$

Notons que les mots de $W(u)$ sont de longueur supérieure à $|x|$; on les considère donc uniquement lorsque $V(u) = \emptyset$.

Exemple. Considérons le mot $x = aababab$. La fonction du bon suffixe est :

	0	1	2	3	4	5	6	7
x		a	a	b	a	b	a	b
$d_2(i)$	14	13	12	6	10	6	8	1

Pour $i = 5$, le suffixe $u = ab$ est bord des suffixes $abab$ et $ababab$. Seul le deuxième est disjoint de u, donc $V(u) = \{ababab\}$, et $d_2(5) = 6$. Pour $i = 4$, le suffixe est $u = bab$; on a $V(u) = \emptyset$ et $W(u) = \{ux\}$, donc $d_2(4) = 10$. Pour $i = 7$, le mot vide est bord disjoint de b, donc $d_2(1) = 1$.

L'algorithme de Boyer et Moore procède comme suit : Le motif $x = x_1 \cdots x_m$ est comparé aux facteurs de longueur m du texte t. Pour une position donnée, on compare les lettres de la droite vers la gauche. Lorsqu'une différence est constatée, c'est-à-dire lorsque $x_i \neq t_j$ pour des indices i et j, l'indice j est incrémenté, l'indice i remis à m, et la comparaison recommence sur les lettres t_j et x_i. Pour l'incrémentation de j, on peut choisir l'une ou l'autre des fonctions de décalage; en fait, on choisit celle qui fournit la plus grande valeur. D'où l'algorithme :

```
Algorithme BOYER-MOORE(x, t);
  j := m;
  tantque j ≤ n faire
    i := m;
    tantque i > 0 etalors t_j = x_i faire
      i := i − 1; j := j − 1 fintantque;
    si i = 0 alors
      j est une fin d'occurrence de x;
      j := j + d_2[i]
    sinon
      j := j + max(d[t_j], d_2[i])
    finsi
  fintantque.
```

Exemple. Reprenons le même exemple du texte $t = aabbbababacaabbaba \cdots$ et du motif $x = aababab$ avec l'algorithme de Boyer et Moore complet. La

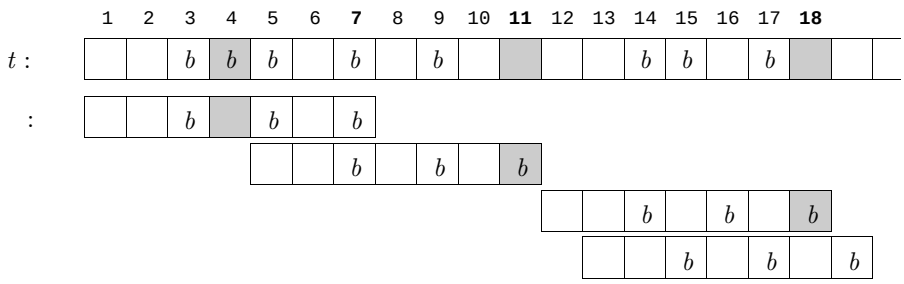


Figure 2.6: *Algorithme de Boyer-Moore complet.*

figure 2.6 montre les décalages successifs du motif. Les différences sont constatées aux positions sombres du texte. Les valeurs successives que prend l'indice j en début de la boucle *tantque* extérieure sont indiquées en gras. Le nombre total de comparaisons de caractères est 8.

Malgré son efficacité expérimentale, l'algorithme de Boyer et Moore peut prendre un temps en $O(|x||t|)$ lorsque l'on cherche *toutes* les occurrences d'un motif dans un texte (prendre $x = a^m$ et $t = a^n$). Une modification de l'algorithme a été proposée qui permet, au moyen d'un prétraitement plus compliqué, d'obtenir un temps linéaire dans tous les cas. R. Cole a prouvé récemment qu'avec cet algorithme, le nombre de comparaisons est toujours borné par $3|t|$. La preuve est compliquée et ne sera pas donnée ici.

10.2.3 Fonction du bon suffixe

Pour compléter l'exposé de l'algorithme de Boyer et Moore, nous allons calculer la fonction du bon préfixe d_2. Il s'avère que l'on peut se ramener à des concepts familiers en *retournant* le motif (c'est-à-dire en prenant son image miroir). Nous allons en fait calculer une *fonction du bon préfixe* du motif.

Soit $x = x_1 \cdots x_m$ un mot, et soit u un préfixe de x. Posons

$$V^{\sim}(u) = \{v \mid v \text{ est préfixe de } x \text{ et } u \text{ est bord disjoint de } v\}$$

Par ailleurs, soit (en conformité avec la notation de la section 10.2.5) $f_u(x)$ le plus long suffixe de x qui est préfixe de u et $w_u = g_u(x)u$, où $g_u(x)$ est défini par $g_u(x)f_u(x) = x$. On définit la fonction du bon préfixe δ pour $i = 0, \ldots, m-1$ et $u = x_1 \cdots x_i$ par

$$\delta(i) = \min_{v \in V^{\sim}(u) \cup \{w_u\}} |v|$$

Si d_2 est la fonction du bon suffixe de $x^{\sim} = x_m \cdots x_1$, alors par construction

$$d_2(i) = \delta(m-i) \qquad i = 1, \ldots, m-1$$

Pour déterminer la fonction du bon préfixe, il faut évaluer l'ensemble $V^{\sim}(u)$ et le mot w_u pour chaque préfixe u de x. Les mots w_u se calculent bien à l'aide de la

fonction $\beta = \beta_x$ qui donne, pour chaque préfixe $x_1 \cdots x_j$ de x, la longueur $\beta(j)$ du bord maximal de $x_1 \cdots x_j$. Nous avons déjà expliqué plus haut comment calculer cette fonction. Considérons alors les nombres

$$m,\ \beta(m),\ \beta^2(m), \ldots,\ \beta^r(m) = 0$$

et soit $u = x_1 \cdots x_i$ un préfixe de x. Si i est dans l'intervalle $[\beta^k(m), \beta^{k-1}(m)[$, on a $|f_u(x)| = \beta^k(m)$, puisque $f_u(x)$ est un bord de x. Il en résulte que

$$|w_u| = |g_u(x)| + |u| = m - \beta^k(m) + i$$

Ceci conduit aux instructions suivantes qui calculent les longueurs $|w_u|$ pour les préfixes de x :

```
j := m;
tantque j > 0 faire
   pour i de β[j] à j − 1 faire δ[i] := m − β[j] + i;
   j := β[j]
fintantque
```

Venons-en aux mots de $V^{\sim}(u)$. Posons $u = x_1 \cdots x_i$. Alors

$$V^{\sim}(u) = \{x_1 \cdots x_j \mid \beta(j) = i \text{ et } x_{j+1} \neq x_{i+1}\} \cup \Delta_i$$

où Δ_i est un terme correctif défini par

$$\Delta_i = \begin{cases} \{x\} & \text{si } \beta(m) = i; \\ \emptyset & \text{sinon.} \end{cases}$$

Pour évaluer correctement ces mots, il semble à première vue nécessaire de déterminer tous les bords disjoints de tous les préfixes de x, ce qui conduit à un algorithme qui n'est plus linéaire en temps. En fait, le morceau de programme suivant convient :

```
(1) pour j de 1 à m − 1 faire
(2)    i := β[j];
(3)    tantque i ≥ 0 etalors x_{j+1} ≠ x_{i+1} faire
(4)       δ[i] := min(δ[i], j);
(5)       i := β[i]
(6)    fintantque
(7) finpour
```

Expliquons pourquoi ce programme est correct. Si l'on remplace les lignes (3)−(6) par

```
(3′)   tantque i ≥ 0 faire
(4′)      si x_{j+1} ≠ x_{i+1} alors δ[i] := min(δ[i], j);
(5′)      i := β[i]
(6′)   fintantque
```

le programme calcule tous les bords disjoints, et choisit le plus petit. Le premier programme ne calcule qu'un sous-ensemble des bords disjoints; pour un indice j donné, il calcule uniquement la suite de bords disjoints *consécutifs* à partir du plus grand bord de $x_1 \dots x_j$. En d'autres termes, dès que l'on rencontre un bord, disons $x_1 \cdots x_k$, qui n'est pas disjoint de $x_1 \dots x_j$, le calcul des bords est interrompu. Ceci se justifie par la double observation suivante : un bord de $x_1 \cdots x_k$ est disjoint de $x_1 \cdots x_k$ si et seulement s'il est disjoint de $x_1 \cdots x_j$, et les bords disjoints de $x_1 \cdots x_k$ (donc les bords disjoints manquants de $x_1 \cdots x_j$) ont déjà été calculés lorsque l'indice de la boucle *pour* a pris la valeur k.

En composant les deux parties que nous venons de développer, on obtient le programme complet calculant la fonction du bon préfixe :

```
procédure Bon-Préfixe(x, δ);
  j := m;
  tantque j > 0 faire
    pour i de β[j] à j − 1 faire δ[i] := m − β[j] + i;
    j := β[j]
  fintantque;
  pour j de 1 à m − 1 faire
    i := β[j];
    tantque i ≥ 0 etalors x_{j+1} ≠ x_{i+1} faire
      δ[i] := min(δ[i], j); i := β[i]
    fintantque
  finpour.
```

Cette procédure calcule la fonction du bon préfixe en temps linéaire. La fonction du bon suffixe s'obtient en trois étapes; on prend l'image miroir du motif, on en calcule la fonction du bon préfixe, et on en déduit d_2 :

```
procédure Bon-Suffixe(x, d_2);
  pour i de 1 à m faire y[i] := x[m + 1 − i];
  Bords-maximaux(y, β);
  Bon-Préfixe(y, δ);
  d_2[0] := m + β[0];
  pour i de 1 à m faire d_2[i] := δ[m − i].
```

On peut ne pas passer par l'image miroir et on peut incorporer le calcul de β à l'intérieur de la procédure, mais le programme devient moins facile à comprendre.

10.3 L'algorithme de Aho et Corasick

Soit X un ensemble fini de mots. Nous considérons ici le problème de localisation des occurrences des mots de X dans un texte t.

On peut faire cette recherche séquentiellement, et chercher successivement les occurrences de chaque motif dans le texte, à l'aide d'un des algorithmes présentés précédemment. Cette démarche n'est pas efficace, puisqu'elle oblige à lire le texte t autant de fois qu'il y a de motifs. En revanche, un algorithme qui effectue la recherche parallèlement pour chaque motif semble prometteur. C'est essentiellement le fonctionnement de l'algorithme de Aho et Corasick que nous présentons ici. Il généralise l'algorithme de Knuth, Morris et Pratt au cas de plusieurs motifs à rechercher dans un texte. L'algorithme se présente comme une généralisation de l'automate des occurrences. On construit un automate déterministe reconnaissant l'ensemble A^*X, sans toutefois expliciter complètement sa fonction de transition, puis on utilise une fonction de suppléance similaire à celle décrite précédemment pour guider le cheminement dans l'automate.

Soit donc X un ensemble fini de mots sur un alphabet A, et soit P l'ensemble des préfixes des mots de X. On construit un automate

$$\mathcal{A} = (P, \varepsilon, P \cap A^*X)$$

reconnaissant A^*X, dont P est l'ensemble d'états, ε est l'état initial et $P \cap A^*X$ l'ensemble d'états terminaux. La fonction de transition est définie, pour $p \in P$ et $a \in A$, par

$$p \cdot a = f_X(pa)$$

où f_X est une extension de la fonction f_x utilisée dans l'algorithme de Knuth, Morris et Pratt. Elle est définie par

$$f_X(u) = \text{ le plus long suffixe de } u \text{ qui est dans } P.$$

En particulier, $p \cdot a = pa$ si $pa \in P$. La partie de l'automate formée seulement des flèches (p, a, pa), avec p et pa dans P, est appelée le *squelette* de l'automate.

Exemple. Considérons l'ensemble $X = \{aba, bab, acb, acbab, cbaba\}$ de 5 mots sur l'alphabet $A = \{a, b, c\}$. Le squelette de l'automate $\mathcal{A}$ est donné dans la figure 3.1. Les états sont numérotés de façon arbitraire. La façon d'obtenir que 4 est état terminal sera expliqué plus loin.

Pour poursuivre l'analogie avec l'automate des occurrences, définissons, pour tout mot u non vide, le mot $\mathrm{Bord}_X(u)$ comme le plus long suffixe propre de u qui est dans P. Voici la fonction Bord_X pour l'ensemble ci-dessus. On a remplacé les préfixes par leurs numéros.

i =	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$\mathrm{Bord}_X(i)$ =	−	0	13	14	15	8	0	13	14	1	2	3	15	0	6	7

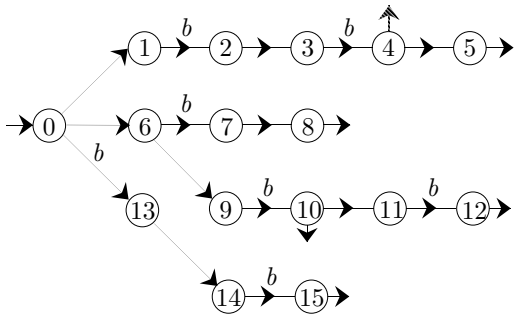


Figure 3.1: *Automate pour l'ensemble X.*

La fonction de transition de l'automate s'exprime à l'aide des bords. En effet :

$$p \cdot a = \begin{cases} pa & \text{si } pa \in P; \\ \text{Bord}_X(pa) & \text{sinon}; \end{cases}$$

et de plus,

$$\text{Bord}_X(pa) = \begin{cases} \text{Bord}_X(p) \cdot a & \text{si } p \neq \varepsilon; \\ \varepsilon & \text{sinon.} \end{cases}$$

Ces formules permettent d'évaluer la fonction de transition comme suit :

```
fonction TRANSITION(p, a);
   tantque pa ∉ P et p ≠ ε faire
      p := Bord_X(p)
   fintantque;
   si pa ∈ P alors retourner(pa) sinon retourner(ε).
```

Pour mettre en œuvre cette fonction, on doit résoudre deux problèmes : il faut connaître la fonction Bord_X, et il faut savoir tester rapidement si pa est dans P. Considérons d'abord la réalisation du test. On peut calculer une table g indicée par $P \times A$, avec

$$g[p, a] = \begin{cases} pa & \text{si } pa \in P; \\ \text{échec} & \text{sinon,} \end{cases}$$

mais cette façon de faire est irréaliste, car la table occupe trop de place dès que l'alphabet est grand (par exemple, si A est l'ensemble des caractères ASCII). Si, dans une application donnée, un tableau de cette taille ne crée pas de problème, alors mieux vaut remplir avec la fonction de transition complète de l'automate les cases qui contiendraient «échec», ce qui accélère ensuite la recherche.

De façon plus concrète, on range l'ensemble P des préfixes des motifs de X dans un arbre; tester si, pour un préfixe p et une lettre a, le mot pa est préfixe, revient à tester s'il y a un arc sortant de p et étiqueté par a. Ce test peut prendre un temps proportionnel à $|A|$ (ou $\log(|A|)$ si l'on programme plus soigneusement, en

rangeant les fils d'un sommet dans un arbre binaire équilibré). La place prise par l'arbre est proportionnelle à $\mathrm{Card}(P) \leq m$, où $m = \sum_{x \in X} |x|$ est la somme des longueurs des motifs. La recherche des occurrences des mots de X dans un texte $t = t_1 \dots t_n$ se fait par l'algorithme suivant, où l'on a incorporé l'évaluation des transitions :

```
procédure AHO-CORASICK(X, t)
  q := ε ;      {état initial}
  pour i de 1 à n faire
    tant que q t_i ∉ P et q ≠ ε faire q := Bord_X[q] fintantque;
    si q t_i ∈ P alors q := q t_i sinon q := ε finsi;
    si q est un état final alors afficher(q) finsi
  finpour.
```

Une table représentant la fonction Bord_X se calcule à l'aide d'un parcours en largeur du squelette de $\mathcal{A}$. Simultanément, on calcule les états terminaux autres que ceux correspondant aux mots de X. L'algorithme est le suivant :

```
procédure BORDS-DE-X(P, Bord_X)
  pour a ∈ A faire Bord_X[a] := ε finpour;
  pour p ∈ P \ ε et a ∈ A tels que pa ∈ P faire
    q := Bord_X[p];
    tantque qa ∉ P et q ≠ ε faire q := Bord_X[q] fintantque;
    si qa ∈ P alors Bord_X[pa] := qa sinon Bord_X[pa] := ε finsi;
    si qa est terminal alors ajouter pa aux états terminaux finsi;
  finpour.
```

Il n'est alors pas difficile de voir que l'algorithme de calcul des occurrences des mots de X dans t est en temps $O(n+m)$ et en place $O(m)$ (plus précisément en temps $O((n+m)|A|)$ ou $O((n+m)\log|A|)$ si l'on tient compte de la taille de l'alphabet).

10.4 Recherche d'expressions

Le problème que nous considérons dans cette section est le suivant : étant donnés une expression rationnelle e et un texte t, déterminer s'il existe un mot dans le langage $X = L(e)$ dénoté par e qui figure dans le texte t, et dans l'affirmative rapporter le mot et son occurrence. Ce problème apparaît couramment dans les éditeurs de textes, dès qu'ils ont des possibilités de recherche un peu sophistiquées.

Comme pour la recherche de motifs, la question revient à chercher le ou les préfixes du texte t qui appartiennent au langage A^*X ; la démarche est aussi la même : on construit d'abord, en « prétraitement », un automate reconnaissant A^*X, puis on cherche les préfixes de t reconnus par cet automate.

La construction d'un automate déterministe pour le langage A^*X n'est pas difficile, mais il peut avoir un nombre considérable d'états, même pour une expression courte : l'exemple donné dans le chapitre précédent montre que le nombre d'états peut être exponentiel en fonction de la taille de l'expression. Si l'on se contente d'un automate qui n'est pas nécessairement déterministe, c'est en revanche la reconnaissance d'un mot qui s'en trouve compliquée. Nous proposons un compromis entre ces deux possibilités : on construit d'abord un automate asynchrone particulier pour le langage dénoté par une expression rationnelle e, et dont le nombre d'états et de flèches est linéaire en fonction de la taille de e. On montre ensuite que la recherche des occurrences des mots du langage dans un texte t de longueur n peut se faire en temps $O(nm)$, où m est la taille de l'expression e.

10.4.1 Calcul efficace d'un automate

Nous étudions ici un procédé de calcul efficace d'un automate reconnaissant le langage dénoté par une expression rationnelle. Soit A un alphabet, et soit e une expression rationnelle sur A. La *taille* de e, notée $|e|$, est le nombre de symboles figurant dans e. Plus précisément, on a

$$|0| = |1| = |a| = 1 \quad \text{pour} \quad a \in A$$
$$|e + f| = |e \cdot f| = 1 + |e| + |f|, \quad |e^*| = 1 + |e|$$

Nous allons construire, pour toute expression e, un automate reconnaissant $L(e)$ qui a des propriétés particulières. Un automate asynchrone $\mathcal{A}$ est dit *normalisé* s'il vérifie les conditions suivantes :

(i) il existe un seul état initial, et un seul état final, et ces deux états sont distincts ;

(ii) aucune flèche ne pointe sur l'état initial, aucune flèche ne sort de l'état final ;

(iii) tout état est soit l'origine d'exactement une flèche étiquetée par une lettre, soit l'origine d'au plus deux flèches étiquetées par le mot vide ε.

Notons que le nombre de flèches d'un automate normalisé est au plus le double du nombre de ses états.

Proposition 4.1. *Pour toute expression rationnelle e de taille m, il existe un automate normalisé reconnaissant $L(e)$, et dont le nombre d'états est au plus $2m$.*

Preuve. Elle est constructive, et elle constitue en fait une autre démonstration de ce que tout langage rationnel est reconnaissable. On procède par récurrence

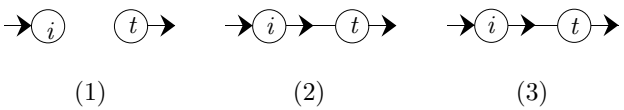


Figure 4.1: *Automates normalisés reconnaissant* (1) *l'ensemble vide,* (2) *le mot vide,* (3) *la lettre a.*

sur la taille de e. Pour $e = 0$, $e = 1$, et $e = a$, où $a \in A$, les automates de la figure 4.1 donnent des automates normalisés ayant 2 états. Si $e = e' + e''$, soient $\mathcal{A}' = (Q', i', t')$ et $\mathcal{A}'' = (Q'', i'', t'')$ deux automates normalisés reconnaissant des langages $X' = L(e')$ et $X'' = L(e'')$. On suppose Q' et Q'' disjoints.

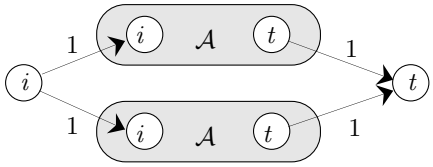


Figure 4.2: *Automate pour l'union.*

L'automate

$$\mathcal{A} = (Q' \cup Q'' \cup \{i, t\}, i, t)$$

où i et t sont deux nouveaux états distincts et dont les flèches sont, en plus de celles de $\mathcal{A}'$ et de A'', les quatre flèches $(i, 1, i')$, $(i, 1, i'')$, $(t', 1, t)$, $(t'', 1, t)$ reconnaît $X' \cup X'' = L(e)$ (voir figure 4.2). L'automate est normalisé, et $|Q| \leq 2|e|$. Si $e = e' \cdot e''$, considérons l'automate

$$\mathcal{A} = ((Q' \setminus t') \cup Q'', i', t'')$$

obtenu en «identifiant» t' et i'', c'est-à-dire en remplaçant toute flèche aboutissant en t' par la même flèche, mais aboutissant en i'' (voir figure 4.3).

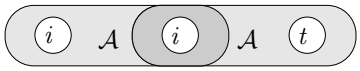


Figure 4.3: *Automate pour le produit.*

Cet automate reconnaît le langage $X'X''$; clairement, son nombre d'états est majoré par $2|e|$. Enfin, si $e = e'^*$, l'automate

$$\mathcal{A} = (Q' \cup \{i, t\}, i, t)$$

de la figure 4.4 qui, en plus des flèches de $\mathcal{A}'$, possède les quatre flèches $(i, 1, i')$, $(i, 1, t)$, $(t', 1, i')$, $(t', 1, t)$ reconnaît le langage $X'^* = L(e)$. Il est normalisé et a 2 états de plus que $\mathcal{A}'$. ∎

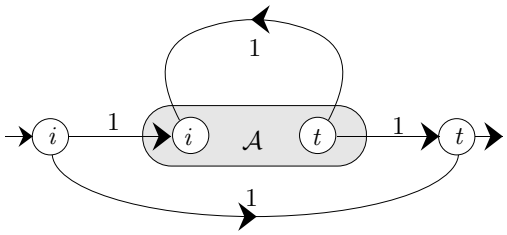


Figure 4.4: *Automate pour l'étoile.*

Exemple. Pour l'expression $(a+b)^*b(1+a)(1+a)^*$, la construction de la preuve produit l'automate de la figure 4.5, dans laquelle les flèches non marquées portent comme étiquette le mot vide. Observons qu'il existe de nombreux chemins dont

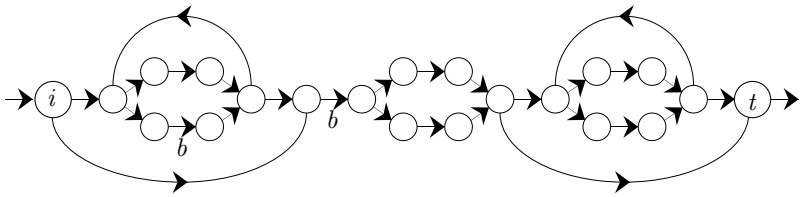


Figure 4.5: *Un automate pour l'expression* $(a+b)^*b(1+a)(1+a)^*$.

l'étiquette est vide, par exemple de l'état marqué d'une étoile à l'état terminal.

10.4.2 Recherche d'occurrences

Soit e une expression rationnelle; la recherche d'occurrences de mots dénotés par l'expression e dans un texte t se fait en deux étapes; dans un premier temps, on construit un automate normalisé $\mathcal{A}$ reconnaissant le langage $A^*L(e)$, à partir de l'expression $(a_1 + \cdots + a_h)^*e$, où $A = \{a_1, \ldots, a_h\}$. Cette étape constitue le prétraitement. Dans un deuxième temps, le texte t est lu lettre par lettre. Pour chaque préfixe de t, on calcule l'ensemble des états de $\mathcal{A}$ accessibles à partir de l'état initial par ce préfixe, et on affiche les préfixes pour lesquels l'état final est accessible. Cette deuxième étape est réalisable en temps $O(Nn)$ où N est le nombre d'états de $\mathcal{A}$ et n est la longueur de t; la raison en est la forme particulière de $\mathcal{A}$ qui permet de calculer chaque nouvel ensemble d'états accessibles en temps linéaire en N à partir du précédent. Comme $N = O(|e|)$, on obtient un algorithme en $O(nm)$, où $m = |e|$.

Le calcul d'un automate normalisé à partir d'une expression rationnelle peut se faire en temps et en place linéaires en fonction de la taille de l'expression. Une façon économique de représenter un automate normalisé est d'utiliser deux tables indicées par son ensemble d'états : la première contient, pour l'indice q, l'unique couple (a, q'), s'il existe, tel que (q, a, q') est une flèche de l'automate avec a une

lettre; la deuxième contient l'ensemble des états qui sont extrémités de flèches d'origine q et étiquetées par le mot vide; cet ensemble a au plus 2 éléments.

Nous en venons maintenant à la reconnaissance. Soit $\mathcal{A} = (Q, i, t)$ un automate normalisé. Comment vérifier si un mot w est reconnu par l'automate $\mathcal{A}$? Evidemment, on ne veut pas déterminiser l'automate, pour ne pas perdre le profit du nombre réduit d'états et de flèches. La méthode consiste à simuler la déterminisation de l'automate $\mathcal{A}$ en conservant, à chaque étape, l'ensemble des états accessibles de l'état initial par un chemin dont l'étiquette est la partie déjà lue du mot d'entrée. La seule difficulté est le calcul des états accessibles à partir d'un état par un chemin d'étiquette 1.

Soit P un ensemble d'états. Notons $\varepsilon(P)$ l'ensemble des états $q \in Q$ pour lesquels il existe $p \in P$ et un chemin $c : p \to q$ d'étiquette 1. Pour toute lettre a, notons $P \cdot a$ l'ensemble des états $q \in Q$ tels qu'il existe une flèche (p, a, q) dans $\mathcal{A}$, avec $p \in P$. Pour vérifier si un mot $w = a_1 \cdots a_n$ est reconnu par $\mathcal{A}$, on construit une suite $P_0, \ldots, P_n$ d'ensembles d'états par

$$\begin{aligned} P_0 &= \varepsilon(\{i\}); \\ P_k &= \varepsilon\left(P_{k-1} \cdot a_k\right), \qquad k = 1, \ldots, n. \end{aligned}$$

Il est immédiat que P_k est l'ensemble des états accessibles de i par un chemin d'étiquette $a_1 \cdots a_k$. Par conséquent, w est accepté par $\mathcal{A}$ si et seulement si l'état final t appartient à P_n. On en déduit donc l'algorithme suivant (où $w = a_1 \cdots a_n$, $\textsc{trans}(P, a) = P \cdot a$, et $\textsc{epsilon}(P) = \varepsilon(P)$) :

```
fonction RECONNAISSANCE(w : mot) : booléen;
  P :=EPSILON({i});
  pour k de 1 à n faire
    P :=EPSILON(TRANS(P, a_k))
  finpour;
  si (t ∈ P) alors retourner(vrai) sinon retourner(faux) finsi.
```

Pour la mise en œuvre de cet algorithme, nous représentons les ensembles d'états par deux structures de données, l'une qui permet l'adjonction et la suppression d'un élément en temps constant, comme une pile ou une file, et l'autre qui permet l'adjonction et le test d'appartenance en temps constant, comme un vecteur booléen. La première ou la deuxième structure est utilisée dans le calcul de la fonction de transition :

```
fonction TRANS(P : ensemble; a : lettre) : ensemble;
(1)  R := ∅;
(2)  pour q dans P faire
(3)     s'il existe q' tel que (q, a, q') est une flèche alors
(4)        R := R ∪ q
(5)     finsi
(6)  finpour;
(7)  retourner(R).
```

L'initialisation à la ligne (1) se fait en temps $O(N)$, si $\mathcal{A}$ a N états; le test à la ligne (3) est en temps constant, donc la boucle est en temps $O(N)$.

Le calcul de ε se fait bien entendu par un parcours de graphe, puisqu'il s'agit de calculer les états accessibles par des chemins composés uniquement de flèches étiquetées par le mot vide. On peut l'implémenter par exemple comme suit, en utilisant deux représentations d'un même ensemble, la première (notée T) permettant de tester en temps constant si cet ensemble est vide, et d'ajouter et de supprimer des éléments (pile ou file par exemple), la deuxième (notée R) permettant de tester l'appartenance en temps constant (un tableau booléen par exemple) :

```
fonction EPSILON(P : ensemble) : ensemble;
(1)  T := P; R := P;
(2)  tant que T ≠ ∅ faire
(3)     choisir q dans T;
(4)     pour chaque flèche (q, 1, q') d'origine q faire
(5)        si q' ∉ R alors R := R ∪ q'; T := T ∪ q' finsi
(6)     finpour
(7)  fintantque;
(8)  retourner(T).
```

La ligne (1) se fait en temps linéaire en N (nombre d'états de l'automate $\mathcal{A}$), la boucle (2) n'est pas exécutée plus de N fois, puisqu'à chaque tour on enlève un élément à T, et que l'on n'ajoute que des états non encore visités. Le nombre de flèches examinées dans la boucle (4) est au plus 2 (et ceci est essentiel pour la linéarité de l'algorithme), et les opérations de la ligne (5) se font en temps constant. Ceci montre que l'algorithme est en temps $O(N)$.

Pour le test d'occurrences, on procède comme pour la reconnaissance, sauf que l'on affiche toutes les positions où se termine un mot reconnu par l'automate. Voici l'algorithme (on a posé $t = t_1 \cdots t_n$) :

```
procédure OCCURRENCES(t : mot) : booléen;
  P :=EPSILON({i});
  pour k de 1 à n faire
    P :=EPSILON(TRANS(P, t_k));
    si l'état terminal est dans P alors afficher(k) finsi;
  finpour.
```

Notes

Parmi les différents algorithmes de recherche d'un motif qui ont été exposés, l'algorithme de Boyer et Moore est expérimentalement le plus rapide, et dans la variante de Horspool, il est simple à programmer. La fonction `egrep` de Unix utilise par exemple un algorithme de recherche d'expressions et, pour des chaînes de caractères, l'algorithme de Boyer-Moore dans la variante de Horspool. L'éditeur de texte Emacs fait amplement appel à la recherche d'expressions, par exemple dans le traitement du courrier. Les performances des divers algorithmes dans le cas le plus défavorable sont bien connues, pour l'efficacité en moyenne (sous l'hypothèse de distribution uniforme) certains des résultats sont très récents, et d'autres manquent encore. Voici une table de valeurs connues :

	pire cas	moyenne
Naïf	$\|x\|\,\|t\|$	$1 + 1/(q-1)$
Morris-Pratt	$2\|t\|$	$1 + 1/q - 2/q^2$
Knuth-Morris-Pratt	*id.*	—
Simon	*id.*	—
Horspool	$\|x\|\,\|t\|$	$2/(1+q)$
Boyer-Moore simple	*id.*	—
Boyer-Moore (variante)	$3\|t\|$	—

Les estimations en moyenne sont de M. Régnier et de Baeza-Yates, Régnier. Elles désignent le nombre moyen de comparaisons par caractère du texte, et valent lorsque la longueur du motif et la taille q de l'alphabet sont «grandes». L'estimation dans le cas le plus défavorable de l'algorithme de Boyer-Moore, due à R. Cole, s'applique à une variante de cet algorithme.

L'exposé de ce chapitre a profité des notes du cours de M. Crochemore à l'Université Paris 7. L'histoire des algorithmes de recherche de motifs jusqu'en 1977 est contée dans :

D.E. Knuth, J.H. Morris Jr, V.R. Pratt, Fast pattern matching in strings, *SIAM J. Comput.* **6** (1977), 323–350.

La version simplifiée de l'algorithme, due à Morris et Pratt, et qui date de 1970, n'a jamais été publiée. La référence la plus complète est :

A. V. Aho, Algorithms for finding patterns in strings, in : J. van Leeuwen, ed., *Handbook of Theoretical Computer Science, Vol. A*, North-Holland, 1990, 255–300.

Ce chapitre contient d'autres algorithmes, comme l'algorithme de Commentz-Walter qui est une version à la Boyer et Moore de la recherche d'un ensemble de motifs dans un texte. Les recherches continuent activement sur l'élaboration et l'évaluation d'algorithmes de recherche de motifs, en particulier d'algorithmes qui minimisent à la fois le nombre de comparaisons, le délai, et la place requise pour conserver le prétraitement. D'autres algorithmes pour une recherche en parallèle, ou pour une recherche avec symboles indifférents, ont aussi été proposés.

Exercices

10.1. Décrire comment il convient de modifier l'algorithme de Knuth, Morris et Pratt pour qu'il détecte *toutes* les occurrences d'un motif dans un texte.

10.2. Deux mots u et v sont *conjugués* s'il existe x, y tels que $u = xy$, $v = yx$. Donner un algorithme qui teste en temps $O(n)$ si deux mots de longueur n sont conjugués.

10.3. Les *mots de Fibonacci* sont définis par $f_0 = a$, $f_1 = ab$, et $f_{n+2} = f_{n+1}f_n$ pour $n \geq 0$. Montrer que $\mathrm{Bord}(f_{n+2}) = f_n$.

10.4. Donner des exemples de mots x pour lesquels l'automate $\mathcal{A}(x)$ a exactement $|x|$ flèches arrière.

10.5. Montrer que tout chemin d'étiquette x dans l'automate $\mathcal{A}(x)$ utilise au plus une flèche arrière.

10.6. Montrer que l'algorithme de Boyer et Moore fait de l'ordre de $3|t|$ opérations en cherchant une occurrence de $x = (ba^k)^2$ dans $t = a^{k+2}(ba^{k+2})^p$, quels que soient les entiers positifs k et p.

10.7. Montrer que l'automate construit dans l'algorithme de Aho et Corasick n'est en général pas l'automate minimal reconnaissant A^*X.

10.8. Soit A un alphabet, et soit $\bullet$ une lettre qui n'est pas dans A. Si $u = a_1 \cdots a_m$ et $v = b_1 \cdots b_m$ sont deux mots, avec $a_i, b_i \in A \cup \{\bullet\}$, on pose $u \sqsubseteq v$ si $a_i \neq \bullet$ implique $a_i = b_i$ pour $1 \leq i \leq m$.

Soit $x = x_1 \cdots x_m$ un mot de longueur m sur A. L'*automate de Boyer et Moore* de x, noté $\mathcal{A}(x)$, a pour états les mots y de longueur m sur $A \cup \{\bullet\}$ tels que $y \sqsubseteq x$ (accessibles à partir de l'état initial comme expliqué ci-dessous). L'état initial est $\bullet^m$, l'état final est x. La fonction de transition est définie comme suit. Soit $q = u\bullet v$, avec $v = x_{k+1} \cdots x_m \in A^*$, soit $a \in A$, et soit $p = uav = y_1 \cdots y_m$. Alors

$$q \cdot a = \begin{cases} p & \text{si } a = x_k\,; \\ y_{1+d} \cdots y_m \bullet^d & \text{sinon,} \end{cases}$$

où d est le plus petit entier tel que $y_{1+d} \cdots y_m \sqsubseteq x_1 \cdots x_{m-d}$. Par exemple, l'automate de Boyer et Moore de *aba* est donné dans la figure ci-dessous.

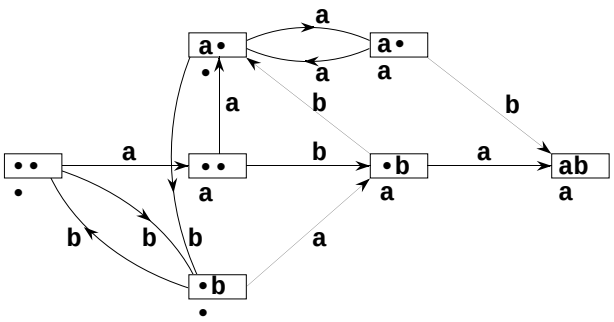


Figure 4.6: *L'automate de Boyer et Moore pour aba.*

a) Ecrire un programme qui prend en argument un mot x et qui calcule l'automate de Boyer et Moore.

b) Montrer que si toutes les lettres de x sont distinctes, l'automate a exactement $m(m+1)/2$ états.

c) Montrer que si $x = a^i b a^j$ avec $i + j + 1 = m$, alors $\mathcal{A}(x)$ a $\theta(m^3)$ états.

Chapitre 11

Géométrie algorithmique

La première section de ce chapitre contient un rappel de quelques notations classiques en géométrie euclidienne, ainsi que la mise en place d'outils élémentaires qui nous seront utiles dans les sections suivantes. La deuxième section est consacrée aux algorithmes de calcul de l'enveloppe convexe d'un ensemble fini de points du plan. Nous terminons par un algorithme dynamique plus sophistiqué. Quelques problèmes de localisation d'un point dans le plan divisé en régions sont étudiés dans la troisième section; on y utilise des structures de données assez complexes, en particulier les ensembles ordonnés persistants. La dernière section est consacrée aux diagrammes de Voronoï de points et de segments.

11.1 Notions préliminaires

Le développement récent de l'algorithmique géométrique est dû aux progrès de la technologie. En effet, la synthèse d'images par ordinateur était encore, il y a quelques années, fort coûteuse. Les ordinateurs actuels ont une rapidité et une puissance de calcul qui rendent l'interactivité possible en infographie. Il est inutile de souligner l'importance de cet outil comme moyen de communication entre la machine et l'utilisateur.

La géométrie joue un rôle fondamental dans un grand nombre de domaines tels que vision par ordinateur, robotique, conception assistée par ordinateur, « design » industriel, image de synthèse, etc. Cela explique l'importance algorithmique des problèmes géométriques. Grossièrement, on peut, si l'on y tient, placer une frontière entre les mathématiques et l' informatique de la façon suivante. Le mathématicien prouve l'existence d'un objet : « l'enveloppe convexe d'un ensemble fini de points du plan est un polygone convexe » , l'informaticien le calcule (si c'est possible), et cherche un algorithme efficace (s'il en existe).

Dans ce chapitre, nous présentons des algorithmes pour quelques problèmes fondamentaux de géométrie plane. Il est intéressant de noter que certains principes algo-

rithmiques classiques tels que la dichotomie se révèlent souvent particulièrement adaptés à un certain nombre de problèmes géométriques. D'autre part, de nouvelles méthodes spécifiques à la géométrie émergent. Parmi elles, on peut citer le principe de balayage de l'espace par un hyperplan. Ce principe permet de transformer un problème statique en dimension k en un problème dynamique en dimension $k-1$, et exploite le fait qu'en cours de balayage, la «situation» étudiée n'est modifiée de manière significative qu'en un nombre fini de positions de l'hyperplan et que par ailleurs deux situations consécutives sont «peu différentes»; on dispose donc d'une grande partie de l'information déjà calculée, en utilisant par exemple la structure d'ensemble ordonné persistant. Une autre approche algorithmique fréquente consiste à subdiviser l'espace en «régions» où les solutions au problème posé sont les mêmes. C'est le cas par exemple dans la recherche des voisins : on calcule le diagramme de Voronoï de l'ensemble des sites.

11.1.1 Notations

Les objets considérés appartiennent à un plan affine euclidien orienté. Parmi les bases orthonormées directes, on en privilégie une notée $(\vec{\imath}, \vec{\jmath})$, définissant deux directions appelées horizontale $(\vec{\imath})$ et verticale $(\vec{\jmath})$.

On notera :

$\overrightarrow{pq}$	le vecteur d'origine p et d'extrémité q;
$d(p, \vec{u})$	la droite passant par p, de vecteur directeur $\vec{u}$ non nul;
$\vec{d}(p, \vec{u})$	la droite orientée par $\vec{u}$ passant par p;
$d(p, q)$	la droite passant par p et q $(p \neq q)$;
$\vec{d}(p, q)$	la droite orientée de p vers q passant par p et q $(p \neq q)$;
$[p, q]$	le segment d'extrémités p et q;
$[p, \vec{u}($	la demi-droite d'origine p de vecteur directeur $\vec{u}$;
dét $(\vec{u}, \vec{v})$	le déterminant du couple de vecteurs $(\vec{u}, \vec{v})$ dans une base orthonormée directe;
$(\vec{u}, \vec{v})$	l'angle de $\vec{u}$ avec $\vec{v}$; on notera encore $(\vec{u}, \vec{v})$ lorsqu'il n'y a pas ambiguïté le secteur angulaire associé à l'angle $(\vec{u}, \vec{v})$.

Un angle$(\vec{u}, \vec{v})$ est dit *convexe* (resp. *réflexe*, *plat*) s'il est de mesure strictement inférieure à π (resp. strictement supérieure à π, égale à π).

Etant donnée une suite de points $(x_1, ..., x_n)$, on appelle *suite circulaire* et on note $((x_1, ..., x_n))$ l'ensemble des suites $(x_i, x_{i+1}, \cdots, x_n, x_1, \cdots, x_{i-1})$ conjuguées de la suite $(x_1, ..., x_n)$. Dans une suite circulaire $((x_1, ..., x_n))$, chaque élément a un *successeur*et un *prédécesseur* unique. Le successeur de x_i est x_{i+1}, avec la convention que $x_{n+1} = x_1$. De même, le prédécesseur de x_i est x_{i-1}, avec la convention que $x_0 = x_n$.

11.1.2 Lignes polygonales, polygones

Une *ligne polygonale* est une suite $L = (S_1, ..., S_n)$ de segments $S_i = [x_i,\ x_{i+1}]$ de longueur non nulle. Les segments S_i sont les *côtés* de la ligne polygonale, et les points x_i sont ses *sommets*. On convient aussi de représenter la ligne polygonale L par la suite $(x_1, ..., x_{n+1})$ de ses sommets.

La ligne polygonale L est *fermée* si $x_{n+1} = x_1$. Elle est *simple* si deux segments non consécutifs ont une intersection vide et si deux segments consécutifs ont une intersection réduite à une extrémité et ont pour support des droites distinctes. Dans le cas où la ligne polygonale est fermée, on considère que les segments S_n et S_1 sont consécutifs.

Un *polygone* (simple) est une ligne polygonale fermée (simple) ayant au moins trois côtés.

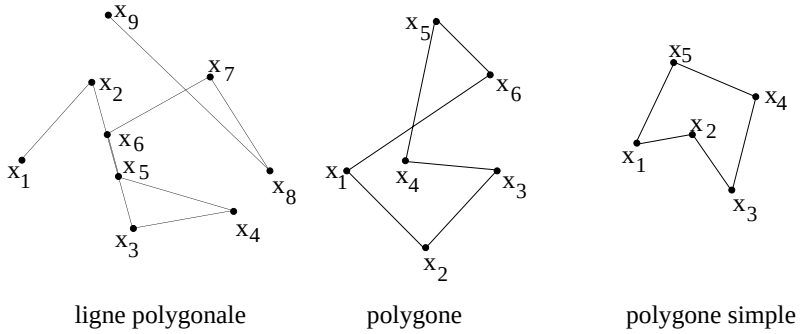


Figure 1.1: *Lignes polygonales.*

Un polygone simple détermine deux régions du plan. L'une bornée, est appelée *l'intérieur*, l'autre non bornée est appelée *l'extérieur*. La frontière de ces deux régions est l'union des côtés du polygone. Pour ne pas alourdir l'écriture, il est d'usage d'appeler encore polygone la région fermée et bornée définie par l'intérieur du polygone et sa frontière.

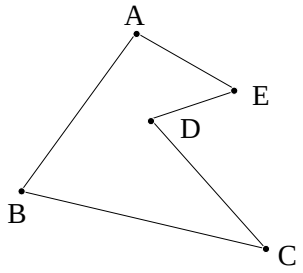


Figure 1.2: *Le contour positif du polygone est* $((A, B, C, D, E))$.

Soit $P = (x_1, ..., x_n, x_{n+1} = x_1)$ un polygone simple. Le plan étant orienté, P a un *contour positif* (ou direct) et un *contour négatif* qu'on conviendra de représenter

par les suites circulaires $((x_1, ..., x_n))$ et $((x_n, \ldots, x_1))$. Le contour positif est tel que l'intérieur du polygone se trouve à sa gauche (voir figure 1.2).

Exemple. Le contour positif du polygone P de la figure 1.2 est $((A, B, C, D, E))$ et son contour négatif est $((A, E, D, C, B))$.

11.1.3 Ordre polaire, circuit polaire

Ordre polaire

On note $m\text{mes}(\vec{u}, \vec{v})$ la mesure en radian de l'angle $(\vec{u}, \vec{v})$ dans l'intervalle $[0, 2\pi[$. Soit D une demi-droite du plan d'origine O. On définit dans le plan privé de O un ordre total appelé *ordre polaire relatif à la demi-droite* D par :

$$p \leq q$$

si et seulement si :

$$\text{mes}(D, \overrightarrow{Op}) < \text{mes}(D, \overrightarrow{Oq}) \text{ ou } (\text{mes}(D, \overrightarrow{Op}) = \text{mes}(D, \overrightarrow{Oq}) \text{ et } Oq \leq Op)$$

Autrement dit, l'ordre polaire relatif à D est l'ordre lexicographique pour les coordonnées polaires des points par rapport à D, à ceci près que l'ordre relatif à la distance à O est inversé.

Exemple. Pour les points de la figure 1.3, l'ordre polaire de S par rapport à D est $(p_6, p_5, p_4, p_2, p_1, p_3, p_7)$.

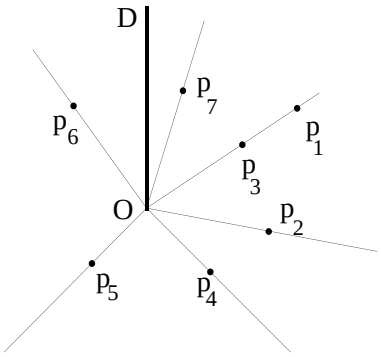


Figure 1.3: *Ordre polaire.*

Lemme 1.1. *Soit S un ensemble de points, $O \notin S$, et deux demi-droites D et D' d'origine O. Les suites obtenues en rangeant les points de S pour les deux ordres polaires relatifs à D et D' sont conjuguées l'une de l'autre.*

Ce lemme nous permet de définir une notion qui ne dépend que de S et O, celle de *circuit polaire de S relativement à O.*

Circuit polaire

Soit S un ensemble de points et $O \notin S$, on appelle *circuit polaire de S relativement à O* la suite circulaire de l'ordre polaire des points de S par rapport à une demi-droite quelconque D d'origine O. Dans l'exemple de la figure 1.3, la suite circulaire $((p_1, p_3, p_7, p_6, p_5, p_4, p_2))$ est le circuit polaire de $S = \{p_1, ..., p_7\}$ par rapport à O.

Nous introduisons ici une nouvelle notion qui permettra de calculer efficacement le circuit polaire d'un ensemble de points relativement à un point fixé. Soit O un point fixé du plan. On considère dans l'ensemble des points du plan privé de O une relation appelée *relation de précédence circulaire convexe relativement à O* notée $\preceq_O$ et définie par :

$$p \preceq_O p' \iff 0 < \text{mes}(\overrightarrow{Op}, \overrightarrow{Op'}) \leq \pi \text{ ou } (\text{mes}(\overrightarrow{Op}, \overrightarrow{Op'}) = 0 \text{ et } Op \geq Op')$$

On notera $\prec_O$ la relation stricte associée. Il est clair que $\preceq_O$ n'est pas une relation d'ordre car elle n'est ni transitive ni anti-symétrique (mais presque...). Dans l'exemple de la figure 1.4 ci-dessous, on a :

$$p_1 \prec_O p_2,\ p_2 \prec_O p_3,\ p_3 \prec_O p_1$$

Néanmoins, la relation de précédence circulaire convexe possède certaines pro-

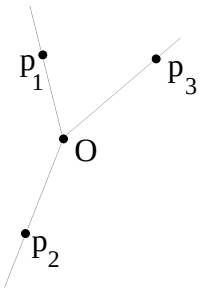


Figure 1.4: $p_1 \prec_O p_2,\ p_2 \prec_O p_3,\ p_3 \prec_O p_1$

priétés utiles, comme nous allons le voir. Notons que cette relation *se calcule en temps constant*, en effectuant uniquement des opérations élémentaires telles que multiplications, additions ou soustractions de nombres réels, sans que l'on soit obligé de calculer les mesures des angles. En effet, on a :

$$p \preceq_O p' \iff \begin{cases} \text{dét}\,(\overrightarrow{Op}, \overrightarrow{Op'}) > 0 \\ \textit{ou} \\ \text{dét}\,(\overrightarrow{Op}, \overrightarrow{Op'}) = 0,\ \overrightarrow{Op} \text{ et } \overrightarrow{Op'} \text{ de même sens, et } Op \geq Op' \\ \textit{ou} \\ \text{dét}\,(\overrightarrow{Op}, \overrightarrow{Op'}) = 0,\ \overrightarrow{Op} \text{ et } \overrightarrow{Op'} \text{ de sens contraire.} \end{cases}$$

On définit par ailleurs la relation ternaire p_1 *est entre* p_0 *et* p_2 *par rapport à* O que l'on notera $entre_O(p_0, p_1, p_2)$ par :

$$\exists i \in \{0, 1, 2\}\ \ p_i \preceq_O p_{i+1} \text{ et } p_{i+1} \preceq_O p_{i+2}$$

(les indices sont définis modulo 3.) C'est une notion étroitement liée à la notion de circuit polaire qui sera fort utile dans la suite, en effet :

Lemme 1.2. *Soit p_0, p_1, p_2 trois points distincts et distincts de O. On a $entre_O(p_0, p_1, p_2)$ si et seulement si le circuit polaire de $\{p_0, p_1, p_2\}$ relativement à O est $((p_0, p_1, p_2))$.*

On peut remarquer que dire que c est entre a et b par rapport à O signifie simplement que le point c appartient au secteur angulaire $(\overrightarrow{Oa}, \overrightarrow{Ob})$ dont une partie de la frontière est exclue, ce qui est précisé dans la figure 1.5. Les traits en gras font partie du secteur angulaire.

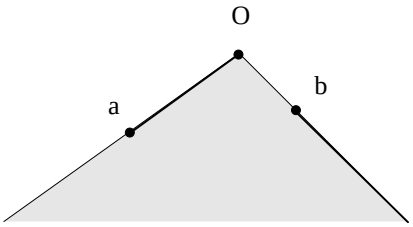


Figure 1.5: *Un secteur angulaire.*

On peut alors redéfinir les notions de circuit polaire et d'ordre polaire grâce à la relation $entre_O$:

Proposition 1.3. *La suite circulaire $((p_1, ..., p_n))$ est un circuit polaire par rapport à O si et seulement si pour tout i, le point p_i est entre p_{i-1} et p_{i+1}.*

Proposition 1.4. *Soit p_0 un point distinct de O, et soit $\leq$ l'ordre polaire par rapport à la demi-droite $[O, \overrightarrow{Op_0}($. On a l'équivalence suivante :*

$$p \leq q \iff p \text{ est entre } p_0 \text{ et } q$$

Cet énoncé fournit un algorithme de calcul du circuit polaire d'un ensemble S par rapport à un point O qui ne nécessite pas le calcul de mesures d'angles. Cela est appréciable car d'une part ces calculs sont coûteux, d'autre part ils utilisent des fonctions réciproques de fonctions trigonométriques qui engendrent des erreurs d'approximation.

Proposition 1.5. *Soit S un ensemble de N points et O un point n'appartenant pas à S. Le circuit polaire de S par rapport O se calcule en $O(NlogN)$ comparaisons entre les points pour la relation de précédence circulaire convexe.*

Preuve. Il suffit de choisir un point p_0 distinct de O, et de trier les points pour l'ordre polaire relatif à la demi-droite $[O, \overrightarrow{Op_0}($ en utilisant la propriété précédente. ∎

On peut remarquer que la relation d'ordre polaire se calcule encore plus simplement dans le cas particulier suivant :

Proposition 1.6. *Soit S un ensemble fini de points, et O un point fixé n'appartenant pas à S. Si tous les points de S appartiennent à un demi-plan ouvert dont la frontière passe par O, alors la relation de précédence circulaire convexe par rapport à O est un ordre total sur S ; c'est l'ordre polaire associé à toute demi-droite Ox non située dans ce demi-plan.*

Nous terminons par la définition de *tour gauche* ou *droit*. Soient p, q, r trois points distincts du plan. Le triplet (p, q, r) est un *tour gauche* (resp. *droit*) si le point r est situé strictement à gauche (resp. à droite) de la droite orientée $\overrightarrow{d}(p, q)$.

Proposition 1.7. *Soient p, q, r trois points non alignés du plan. Les cinq propriétés qui suivent sont équivalentes :*

- (*i*) *(p, q, r) est un tour gauche ;*
- (*ii*) *$((p, q, r))$ est le contour positif du triangle formé des trois points p, q, r ;*
- (*iii*) $\text{dét}(\overrightarrow{pq}, \overrightarrow{pr}) > 0$;
- (*iv*) *l'angle $(\overrightarrow{pq}, \overrightarrow{pr})$ est convexe non nul ;*
- (*v*) $q \prec_p r$.

Nous considérons maintenant la question de la fusion en un seul circuit de deux circuits polaires par rapport à un même point O. L'algorithme proposé est voisin de celui qu'on utilise habituellement pour la fusion de deux listes triées, mais il est adapté aux circuits qui sont de nature légèrement différente. Soient $P = ((p_0, \ldots, p_{n-1}))$, $Q = ((q_0, \ldots, q_{k-1}))$ deux circuits polaires par rapport à un point O. La suite $(p_0, \ldots, p_{n-1})$ représente la liste triée des points de P pour l'ordre polaire relativement à la demi-droite $[O, \overrightarrow{Op_0}($, de même $(q_0, \ldots, q_{k-1})$ représente la liste triée des points de Q pour l'ordre polaire relativement à la demi-droite $[O, \overrightarrow{Oq_0}($. Il suffit pour résoudre le problème de fusionner deux listes triées *pour le même ordre*, par exemple pour l'ordre polaire par rapport à $[O, \overrightarrow{Oq_0}($. Or on sait que la suite triée des points de P pour l'ordre polaire par rapport à $[O, \overrightarrow{Oq_0}($ est une suite conjuguée de la suite $(q_0, \ldots, q_{k-1})$, commençant disons par p_{i+1} (figure 1.6).

Ce point s'obtient en insérant q_0 dans le circuit P, car q_0 s'insère alors entre p_i et p_{i+1}. Il reste alors à fusionner les deux listes triées $(q_0, \ldots, q_{k-1})$ et $(p_{i+1}, \ldots, p_{n-1}, p_0, \ldots, p_i)$ pour l'ordre polaire par rapport à $[O, \overrightarrow{Oq_0}($.

Les circuits P et Q par rapport à O sont donnés sous forme de listes. L'algorithme FUSION(P, Q, O) calcule une liste qui représente le circuit fusion de P et Q par rapport au point O.

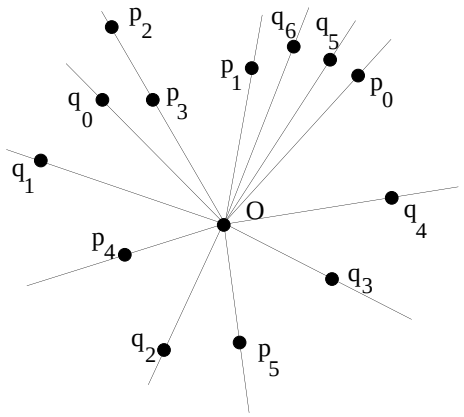


Figure 1.6: q_0 *s'insère entre* p_3 *et* p_4*, on fusionne alors* $(q_0, \ldots, q_6)$ *avec* $(p_4, p_5, p_0, \ldots, p_3)$.

Algorithme FUSION(P, Q, O);
(1) Insérer q_0 dans le circuit P; soit p_i le prédécesseur de q_0;
(2) Fusionner les listes triées $(q_0, \ldots, q_{k-1})$ et $(p_{i+1}, \ldots, p_{n-1}, p_0, \ldots, p_i)$ pour l'ordre polaire relatif à la demi-droite $[O, \overrightarrow{Oq_0}($.

L'étape (1) prend un temps $O(n)$. L'étape (2) prend un temps $O(n+k)$. Notons ici encore que la fusion des listes ne nécessite pas le calcul de l'angle polaire des points par rapport à $[O, \overrightarrow{Oq_0}($, la relation de précédence circulaire convexe suffit pour trier les points.

Proposition 1.8. *La fusion de deux circuits polaires de tailles respectives* n *et* k *se fait en temps* $O(n + k)$.

11.2 Enveloppe convexe

11.2.1 Généralités

La convexité est une propriété très intéressante en géométrie et elle intervient dans un nombre important d'algorithmes. C'est pourquoi un important travail de recherche s'est effectué sur ce thème pour s'efforcer d'obtenir des algorithmes de calcul d'enveloppe convexe qui soient performants tant en espace qu'en temps. Nous donnons ici les algorithmes les plus classiques en montrant les avantages et inconvénients de chacun, en terminant par un algorithme dynamique.

Un ensemble S est *convexe* si pour tout couple (x, y) de points de S, le segment $[x, y]$ est contenu dans S. L'*enveloppe convexe* d'un ensemble S, notée $EC(S)$, est le plus petit ensemble convexe contenant S.

Remarquons qu'une intersection quelconque d'ensembles convexes est convexe, et que le plan lui-même est convexe. Il s'ensuit que pour tout ensemble S, l'enveloppe convexe de S existe car c'est exactement l'intersection de tous les ensembles convexes contenant S. Nous nous limitons ici au calcul de l'enveloppe convexe d'un ensemble fini de points du plan. Dans ce cas, l'enveloppe convexe vérifie une propriété intéressante que l'on peut décrire par l'image suivante : on obtient l'enveloppe convexe de n points en entourant ces points avec un ruban élastique; lorsqu'on le lâche il prend la forme de l'enveloppe convexe .

Remarque. Dans toute cette section, nous ferons une légère entorse à la définition d'un polygone simple et admettrons qu'un polygone simple peut n'avoir que deux côtés, i.e. être d'intérieur vide, ceci pour donner des théorèmes aux énoncés plus simples, ainsi que des preuves sans cas particulier.

Théorème 2.1. *Soit $S \subset \mathbb{R}^2$ un ensemble de n points ($n > 1$). L'enveloppe convexe de S est un polygone convexe dont les sommets appartiennent à S.*

Avant de donner la preuve du théorème, précisons des notations que nous serons amenés à utiliser à plusieurs reprises par la suite.

Soit $((p_1, ..., p_n))$ le contour direct d'un polygone convexe P et p un point extérieur à P. Soient δ et δ' les deux demi-droites issues de p et tangentes à P, telles que l'angle (δ, δ') soit convexe. Ces demi-droites coupent le contour de P soit en un sommet de P soit en un côté de P (voir figure 2.1). Soit p_i (resp. p_j) le sommet de P le plus éloigné de p sur δ (resp. sur δ'). On dira que le secteur angulaire $(\overrightarrow{pp_i}, \overrightarrow{pp_j})$ est *le cône enveloppant P de sommet p*. Notons que p_i et p_j peuvent encore être déterminés de la façon suivante. Soit D une demi-droite ne coupant pas le polygone P, et considérons l'ordre polaire relatif à D. Alors p_i est le plus petit point pour l'ordre polaire relatif à D parmi les sommets de P; par ailleurs soit p'_j le plus grand point parmi les sommets de P, alors p_j est le sommet de P le plus loin de O sur la demi-droite $[O, \overrightarrow{Op'_j}($, c'est-à-dire le plus petit sur cette demi-droite pour la relation d'ordre que l'on considère. Les points p_i et p_j se calculent donc en temps $O(n)$ si n est le nombre de sommets de P.

Preuve. La preuve se fait par récurrence sur n. La propriété est vraie pour $n = 2$. Soit $S' \in \mathbb{R}^2$ un ensemble de n points ($n > 2$), $p \in S'$, et $S = S' - \{p\}$. Si p appartient à l'enveloppe convexe $EC(S)$ alors $EC(S') = EC(S)$. Sinon, soit $((p_1, ..., p_h))$ le contour direct de $EC(S)$, et soient p_i et p_j les sommets de $EC(S)$ tels que $(\overrightarrow{pp_i}, \overrightarrow{pp_j})$ soit le cône enveloppant $EC(S)$ de sommet p. Les sommets p_i et p_j coupent le contour direct de $EC(S)$ en deux listes telles que l'une, de p_i vers p_j, est rangée dans le sens positif pour un circuit polaire relativement à p, et l'autre, de p_j vers p_i, dans le sens négatif. En supprimant cette dernière liste, sauf les points p_i et p_j, et en la remplaçant par p on obtient le contour positif de $EC(S')$ (figure 2.2). ∎

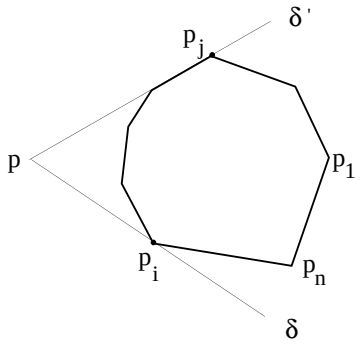


Figure 2.1: *Cône enveloppant.*

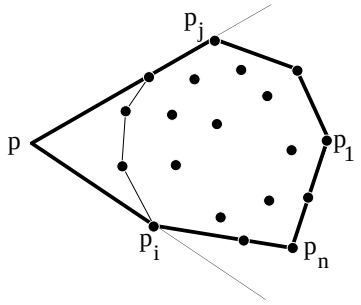


Figure 2.2: *Adjonction d'un point n'appartenant pas à l'enveloppe convexe.*

Problème 1. *Etant donné un ensemble fini S de points du plan, calculer $EC(S)$.*

Avant d'examiner quelques algorithmes classiques pour résoudre ce problème, nous pouvons donner une borne inférieure du temps de calcul.

Connaissant la nature de l'objet à calculer, précisons sous quelle forme nous voulons obtenir le résultat. Puisque $EC(S)$ est un polygone, nous demanderons que le résultat du calcul soit le contour positif du polygone, c'est à dire une suite circulaire de ses sommets.

Théorème 2.2. *Le calcul de l'enveloppe convexe de n points demande un temps $\Omega(n \log n)$.*

Preuve. Considérons un ensemble S de n points $p_i(x_i, x_i^2)_{i=1,\ldots,n}$ situés sur une parabole d'équation $y = x^2$. Le contour positif de $EC(S)$ permet donc d'obtenir, après une lecture, les nombres x_i triés par ordre croissant d'abscisses. Or le tri de n éléments d'un ensemble ordonné nécessite un temps $\Omega(n \log n)$. ∎

Notons que la preuve du théorème 2.1 donne un algorithme de calcul en temps $O(n^2)$.

11.2.2 Marche de Jarvis

La «marche de Jarvis» est un des algorithmes les plus naturels puisqu'il correspond à la technique d'emballage dite du «paquet cadeau» .

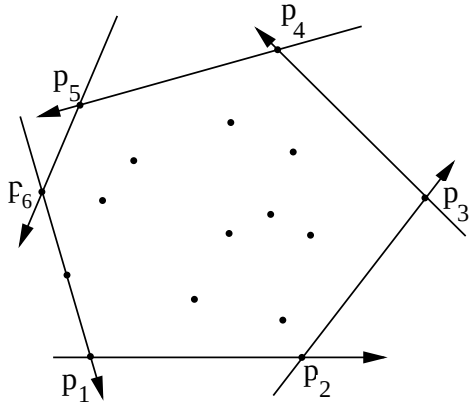


Figure 2.3: *Marche de Jarvis.*

Dans un premier temps, on peut chercher à déterminer non pas les sommets de l'enveloppe, mais ses arêtes. Or $[p, q]$ est une arête de l'enveloppe convexe si et seulement si tous les autres points sont situés sur le segment $[p, q]$ ou du même côté de la droite $d(p, q)$. On peut donc déterminer en temps $O(n)$ si un segment donné appartient à la frontière de l'enveloppe convexe. Or il y a $\binom{n}{2}$ segments à examiner. On obtient donc un algorithme en $O(n^3)$.

Jarvis a amélioré l'algorithme précédent en utilisant l'observation suivante : soit $((p_1, p_2, \ldots, p_n))$ le contour positif de l'enveloppe convexe des n points; le point p_{i+1} est le point minimal pour l'ordre polaire relatif à la demi-droite $d[p_i, \overrightarrow{p_{i-1}p_i}($, dans l'ensemble des sommets privé de p_i. Il suffit donc de déterminer deux points consécutifs p_1 et p_2 de l'enveloppe convexe, par exemple pour p_1 on peut choisir un point d'ordonnée minimale (et d'abscisse minimale, s'il y a plusieurs points d'ordonnée minimale) p_2 étant alors le point minimal dans $S - \{p_1\}$ pour l'ordre polaire relativement à la demi-droite $[p_1, \vec{\imath}($, ce qui prend un temps linéaire, puis de calculer les points p_i grâce à la remarque ci-dessus. D'où l'algorithme :

Algorithme JARVIS(S);
(1) Soit p_1 le point d'ordonnée minimale (et d'abscisse minimale s'il y a plusieurs points d'ordonnée minimale) de S;
(2) Soit p_2 le point minimal de $S - \{p_1\}$ pour l'ordre polaire par rapport à la demi-droite $[p_1, \overrightarrow{\imath}($;
(3) $k := 2$;
(4) répéter
(5) calculer le point q de S minimal pour l'ordre polaire par rapport à $[p_k, \overrightarrow{p_{k-1}p_k}($;
$k := k + 1$; $p_k := q$
jusqu'à ce que $p_k = p_1$.

Evaluons la complexité en temps de l'algorithme pour un ensemble S à n points. Clairement (1) et (2) prennent un temps $O(n)$. La boucle (4) est répétée $h - 1$ fois où h est le nombre de sommets de l'enveloppe convexe. La ligne (5) prend un temps $O(n)$. Donc l'algorithme est en temps $O(hn)$ où h est le nombre de sommets de l'enveloppe convexe, ce qui peut être intéressant si h est petit devant n, mais dans le pire des cas l'algorithme est en temps $O(n^2)$.

11.2.3 Algorithme de Graham

L'algorithme que nous exposons ici est dû à Graham; il repose sur les deux propriétés suivantes d'un polygone convexe :

Théorème 2.3. *Quelque soit le point O intérieur à un polygone convexe P, le contour positif de P est exactement le circuit polaire des sommets de P par rapport à O.*

Preuve. En effet si p, q, r sont trois points consécutifs du contour positif de P, alors on a $entre_O(p, q, r)$ du fait que P est convexe et que O est intérieur à P. ∎

On peut faire ici deux remarques :

— le résultat reste vrai si O est sur la frontière de P mais n'est pas un sommet de P;

— l'énoncé est faux pour un polygone quelconque, mais n'est pas caractéristique des polygones convexes (voir figure 2.4).

Théorème 2.4. *Un polygone simple est convexe si et seulement si pour tout triplet (p, q, r) de sommets consécutifs dans le sens direct, (p, q, r) est un tour gauche.*

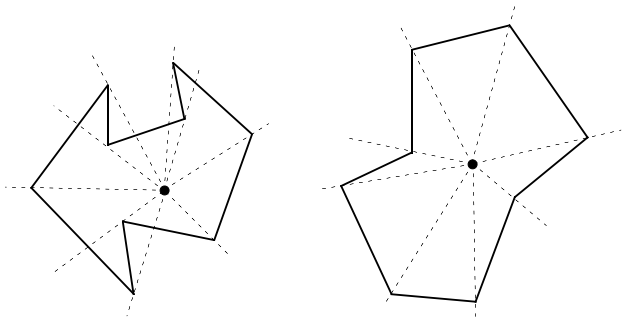


Figure 2.4: *Circuit polaire : exemple et contre-exemple pour un polygone non convexe.*

L'algorithme de Graham consiste à choisir un point intérieur à l'enveloppe convexe de S, mais n'appartenant pas à S, trier les points de S en un circuit polaire par rapport à O, et éliminer les points qui dans ce circuit ne constituent pas un tour gauche avec leurs deux voisins.

L'algorithme s'applique bien sûr dans le cas où les points de S ne sont pas alignés. Pour trouver un point intérieur à $EC(S)$ mais n'appartenant pas à S, il suffit de prendre l'isobarycentre O de trois points a, b, c de S non alignés, et de vérifier qu'il n'appartient pas à S. Si $O \in S$ alors on remplace O par le milieu de a et O, on itère ce processus au plus n fois. Donc trouver ce point O prend dans le pire des cas un temps $O(n)$.

On calcule ensuite le circuit polaire de S par rapport à O sous forme de liste doublement chaînée pour avoir accès en temps $O(1)$ au prédécesseur et au successeur d'un point du circuit. On réalise alors un parcours de cette liste $((p_0, \ldots, p_{n-1}))$ dans le sens croissant, en partant d'un point p_0 dont on est certain qu'il appartient à $EC(S)$, par exemple le point d'abscisse maximale (et d'ordonnée maximale en cas de non unicité) (fig.2.5). Si (p_{i-1}, p_i, p_{i+1}) est un tour droit, alors il est clair que p_i n'appartient pas à la frontière de $EC(S)$ puisque O est intérieur à $EC(S)$, donc p_i est intérieur au triangle $Op_{i-1}p_{i+1}$ et donc intérieur à $EC(S)$. L'algorithme consiste donc à parcourir la liste doublement chaînée en éliminant de tels points.

procédure GRAHAM(S) ;
(1) Déterminer un point O intérieur à S n'appartenant pas à S ;
(2) Calculer le circuit polaire de S par rapport à O ;
(3) Soit p_0 le point de S d'abscisse maximale (et d'ordonnée maximale s'il y en a plusieurs) ;
{ si p est un sommet, $succ(p)$ et $pred(p)$ désignent respectivement le successeur et le prédécesseur de p dans le circuit courant}
(4) $p := p_0$;
tantque $succ(p) \neq p_0$ faire
si $(p, succ(p), succ(succ(p))$ est un tour gauche alors $p := succ(p)$
sinon supprimer $succ(p)$; $p := pred(p)$
finsi
fintantque.

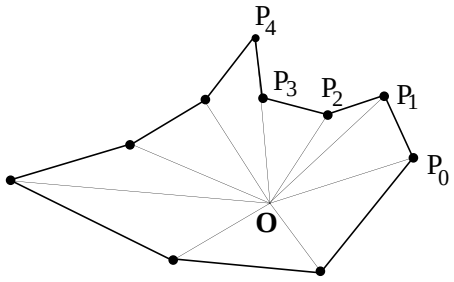


Figure 2.5: *Algorithme de Graham.*

Validité de l'algorithme

A la fin de la procédure on obtient une liste doublement chaînée de points de S telle que
(1) les points forment un circuit polaire par rapport à O ;
(2) trois points consécutifs quelconques forment un tour gauche.
On obtient donc la liste des sommets consécutifs d'un polygone convexe P en vertu des théorèmes 2.3 et 2.4, et puisque les points de S qui ont été supprimés n'appartenaient pas à la frontière de $EC(S)$, P est bien la frontière de $EC(S)$.

Théorème 2.5. *L'algorithme de Graham calcule l'enveloppe convexe de n points du plan en temps $O(n \log n)$ et en espace $O(n)$.*

Preuve. Les étapes 1 et 3 demandent, on l'a vu, un temps $O(n)$. L'étape 2 se réalise en temps $O(n \log n)$ (Proposition 1.5). Enfin il est aisé de voir que la boucle «tant que» (4) prend un temps linéaire. En effet, à chaque passage dans la boucle soit on avance d'un pas dans le parcours, soit on supprime un point. Or on ne peut avancer de plus de n pas, et on ne peut pas supprimer plus de n points.

Enfin l'espace nécessaire à l'exécution de la procédure est une liste doublement chaînée de taille n, c'est donc un espace linéaire. ∎

Traitons un exemple :

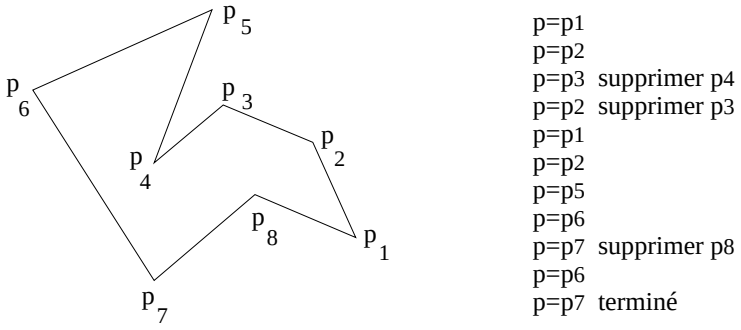


Figure 2.6: *Exemple.*

Bien que l'algorithme de Graham soit optimal dans le pire des cas, il est intéressant d'examiner d'autres algorithmes qui peuvent présenter des avantages variés.

11.2.4 Algorithme dichotomique

L'algorithme qui suit utilise le principe de dichotomie. Pour calculer l'enveloppe convexe $EC(S)$ d'un ensemble S, on divise S en deux parties S_1 et S_2 ayant le même nombre d'éléments à 1 près, on calcule leurs enveloppes convexes $EC(S_1)$, $EC(S_2)$, puis on détermine l'enveloppe convexe de $EC(S_1) \cup EC(S_2)$. L'efficacité de l'algorithme dépend de la complexité de la dernière étape. Or celle-ci consiste à calculer l'enveloppe convexe de l'union de deux polygones convexes, problème plus simple que celui d'origine que l'on peut espérer résoudre en temps linéaire. On est donc conduit à étudier le :

Problème 2. *Etant donnés deux polygones convexes P_1 et P_2, déterminer l'enveloppe convexe de leur union.*

Nous allons présenter un algorithme dont la complexité est donnée par le :

Théorème 2.6. *L'enveloppe convexe de l'union de deux polygones convexes se calcule en temps $O(N)$, où N est le nombre total de sommets.*

Preuve. On suppose qu'au moins un des deux polygones a plus de deux sommets (par exemple P_1, sinon on calcule en temps $O(1)$ l'enveloppe convexe des quatre points). Soit p un point intérieur à P_1, p est déterminé en temps $O(1)$.

Si p appartient à $EC(P_2)$ (ce qui se teste en temps $O(n_2)$, où n_2 est le nombre de sommets de P_2), alors les contours positifs de P_1 et P_2 sont respectivement les circuits polaires de P_1 et P_2 par rapport à p (théorème 2.3). En temps $O(n_1 + n_2)$

on peut fusionner ces deux circuits en un circuit polaire relativement à p. Il suffit ensuite d'appliquer l'étape 4 de l'algorithme de Graham qui est en temps $O(n_1 + n_2)$.

Si p est extérieur à P_2 (figure 2.7), on détermine en temps $O(n_2)$ deux sommets p_i et p_j de P_2 tels que (p_i, p, p_j) constitue le cône de sommet p qui enveloppe P_2 . On fusionne en temps $O(n_1 + n_2)$ le contour positif de P_1 et la liste des sommets de P_2 de p_j à p_i pour obtenir un circuit polaire de l'ensemble de ces points par rapport à p. Il n'y a plus qu'à appliquer l'étape 4 de l'algorithme de Graham à ce circuit, ce qui se réalise en temps $O(n_1 + n_2)$. ∎

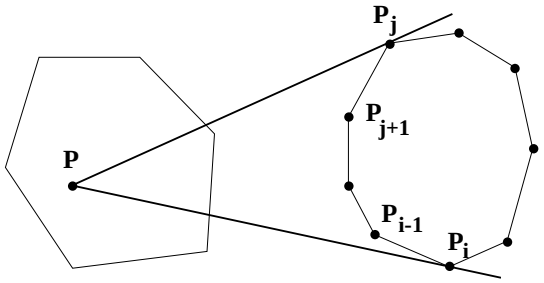


Figure 2.7: *Cas où p est extérieur à P_2.*

L'algorithme dichotomique s'écrit très simplement en procédant récursivement. Notons ENV_CONV_FUSION (P_1, P_2) la procédure qui calcule l'enveloppe convexe de l'union de deux polygones convexes P_1 et P_2. Alors :

```
fonction ENV_CONV_DICHO(S);
  si |S| ≤ 3 alors calculer directement l'enveloppe convexe de S
  sinon
    diviser S en deux ensembles S1 et S2 de même cardinal (à 1 près);
    P1 :=ENV_CONV_DICHO(S1);
    P2 :=ENV_CONV_DICHO(S2);
    ENV_CONV_FUSION(P1, P2)
  finsi.
```

Théorème 2.7. *En procédant par dichotomie, l'enveloppe convexe de N points se calcule en temps $O(N \log N)$.*

Preuve. Supposons que N est une puissance de 2. Soit $T(N)$ le temps de calcul de la fonction ENV_CONV_DICHO(S) pour un ensemble S à N éléments et $F(N)$ le temps de calcul de la procédure ENV_CONV_FUSION(P_1, P_2) où N est le nombre total de sommets de P_1 et P_2. On a alors

$$T(N) \leq 2T(N/2) + F(N) \tag{2.1}$$

Comme on l'a vu plus haut, $F(N)$ est $O(N)$. Donc de 2.1 on déduit que $T(N)$ est $O(N \log N)$. ∎

Notons que cet algorithme possède l'avantage de pouvoir être utilisé en parallèle en raison de son mécanisme de fusion.

11.2.5 Gestion dynamique d'une enveloppe convexe

Les algorithmes précédemment décrits opèrent tous de manière statique, c'est-à-dire qu'ils exigent de connaître l'ensemble S tout entier avant de commencer le calcul de l'enveloppe convexe de S. Si l'ensemble S est modifié par adjonction ou suppression de certains points, le calcul doit être repris complètement. La question qui se pose est donc la suivante : trouver un algorithme efficace pour calculer l'enveloppe convexe d'un ensemble dynamique de points. La réponse bien sûr n'est pas simple, comme on peut l'imaginer, en particulier la suppression d'un point de S peut faire apparaître sur la nouvelle frontière plusieurs points qui étaient auparavant internes à l'enveloppe convexe (figure 2.8). L'idéal serait

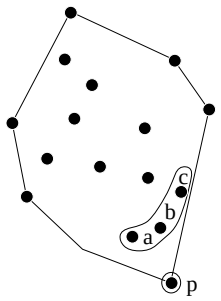


Figure 2.8: *Si p est supprimé, a, b et c apparaissent.*

d'obtenir un algorithme fournissant une représentation dynamique de l'enveloppe convexe d'un ensemble S de taille N tel que le temps de calcul de $EC(S)$ soit aussi bon que dans le cas statique, c'est-à-dire en temps $O(N \log N)$ et tel que la mise à jour de $EC(S)$ par adjonction ou suppression d'un point de S prenne un temps raisonnable, c'est-à-dire nettement inférieur à $N \log N$. Pour obtenir un temps de calcul de $EC(S)$ en $O(N \log N)$ il faudrait que la mise à jour par adjonction ou suppression d'un point prenne un temps $O(\log N)$. Aucun algorithme à ce jour ne donne ce résultat. Néanmoins l'algorithme que nous allons examiner offre un temps de mise à jour en $O(\log^2 N)$, ce qui donne un temps global de calcul de $EC(S)$ en $O(N \log^2 N)$ qui est bien sûr moins bon que dans un calcul statique, mais il répond à des exigences plus grandes.

La solution que nous proposons ici est due à Overmars et van Leeuwen. Une première chose à remarquer est la suivante : $EC(S)$ peut être considéré comme l'intersection de deux ensembles convexes non bornés (figure 2.9) qu'on appellera enveloppe convexe supérieure $ECS(S)$, et enveloppe convexe inférieure $ECI(S)$

que l'on peut décrire par abus comme les enveloppes convexes respectives des ensembles $S \cup \{P_{-\infty}(0, -\infty)\}$ et $S \cup \{P_{+\infty}(0, +\infty)\}$ respectivement.

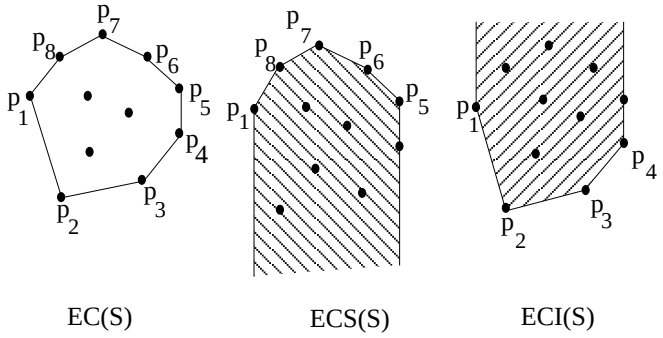


Figure 2.9: *Enveloppes convexes supérieure et inférieure.*

Les contours de ces deux enveloppes $ECI(S)$ et $ECS(S)$ donnent naissance à deux lignes polygonales qui convenablement concaténées définissent le contour direct de $EC(S)$. Dans l'exemple de la figure 2.9, on a :

$$\begin{aligned} ECS(S) &= (p_5, p_6, p_7, p_8, p_1) \\ ECI(S) &= (p_1, p_2, p_3, p_4) \\ EC(S) &= ((p_1, p_2, p_3, p_4, p_5, p_6, p_7, p_8)) \end{aligned}$$

Il nous suffit donc de nous limiter à l'étude de l'enveloppe convexe inférieure.

La structure de données choisie pour maintenir de manière dynamique l'enveloppe convexe $ECI(S)$ est, (ce n'est pas surprenant) une structure d'arbre, pour obtenir un coût de mise à jour réduit. On range les points de S aux feuilles d'un arbre binaire complet équilibré (c'est donc un arbre balisé au sens du chapitre 6). L'ordre pour la recherche est l'ordre lexicographique sur $\mathbb{R}^2$ (les points sont donc rangés par abscisses croissantes, puis en cas d'égalité par ordonnées croissantes). La balise pour piloter la recherche en un nœud v est un couple (x, y) représentant les coordonnées du point rangé dans la feuille la plus à droite du sous arbre gauche du nœud v. Chaque nœud v a pour vocation de représenter l'enveloppe convexe inférieure des points rangés dans les feuilles du sous-arbre de racine v, qu'on notera $Inf(v)$. Soient $filsG(v)$ et $filsD(v)$ les sommets de l'arbre qui sont respectivement fils gauche et fils droit du nœud v. Comment calculer $Inf(v)$ à partir de $I_1 = Inf(filsG(v))$ et $I_2 = Inf(filsD(v))$? La figure 2.10 donne le résultat.

Il existe un unique sommet p_1 de I_1 et un unique sommet p_2 de I_2 tels que le segment $[p_1, p_2]$ soit un côté de $Inf(v)$. On appellera *pont* ce segment, et p_1 et p_2 seront les deux *piliers* du pont. Le sommet p_1 scinde I_1 en deux lignes polygonales I_{11} et I_{12} (dans l'ordre), p_1 étant rangé dans I_{11}, et p_2 scinde I_2 dans l'ordre en deux lignes polygonales I_{21} et I_{22}, p_2 étant affecté à I_{22}. Alors $Inf(v)$ est la concaténation de I_{11} et I_{22}. Nous allons voir qu'il suffit de stocker en chaque sommet v la partie $Q(v)$ de l'enveloppe convexe inférieure $Inf(v)$ qui n'appartient pas à $Inf(père(v))$.

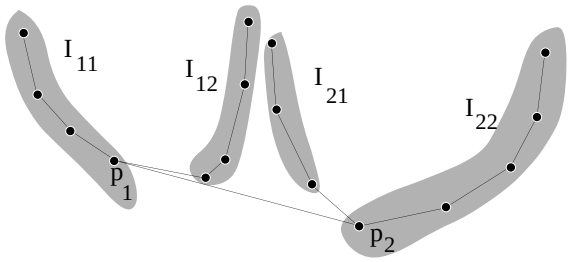


Figure 2.10: *Calcul de Inf(v) à partir de l'enveloppe convexe inférieure des feuilles des fils de v.*

Réciproquement, comment fabriquer *Inf*(*filsG*(v)) et *Inf*(*filsD*(v)) à partir de *Inf* (v) ? Il suffit pour cela de connaître les extrémités du pont $[p_1(v), p_2(v)]$ entre *Inf*(*filsG*(v)) et *Inf*(*filsD*(v)). A partir de *Inf*(v) et $p_1(v)$, on scinde *Inf*(v) en deux lignes polygonales qui, concaténées respectivement avec $Q($*filsG*$(v))$ et $Q($*filsD*$(v))$ deviennent les enveloppes *Inf*(*filsG*(v)) et *Inf*(*filsD*(v)) cherchées.

Ainsi nous pouvons décrire maintenant complètement la structure choisie. En chaque nœud v (différent d'une feuille) de l'arbre binaire sont stockées deux informations :

1) un pointeur vers la ligne polygonale $Q(v)$ stockée sous forme de liste concaténable (voir la section 6.5). Elle représente la partie de *Inf*(v) qui n'appartient pas à *Inf*($père(v)$) (si v est la racine, $Q(v) =$*Inf*(v)).

2) un couple $(p_1(v), p_2(v))$ représentant les piliers gauche et droit du pont de *Inf*(v).

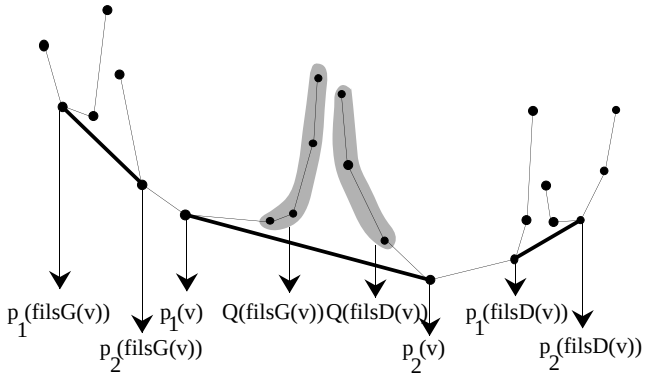


Figure 2.11: *Les fonctions p_1, p_2 et G.*

Cette représentation de l'enveloppe convexe inférieure d'un ensemble de points de taille N est intéressante car elle nécessite un espace $O(N)$. En effet, l'arbre a $2N-1$ sommets et les points rangés dans les listes $Q(v)$ en chaque nœud v sont au nombre total de N, puisque les listes $Q(v)$ forment en fait une partition de l'ensemble S.

Nous allons donner un algorithme qui calcule de manière efficace le pont $[p_1, p_2]$ entre I_1 et I_2 que l'on notera $\textsc{Pont}(I_1, I_2)$ et donc permet de calculer I à partir de I_1 et I_2.

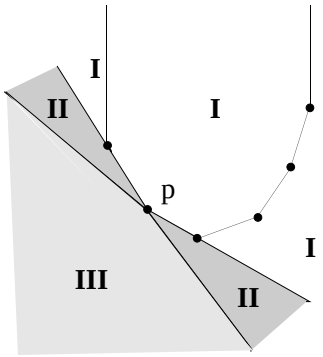


Figure 2.12: *Les 3 régions I, II, III auxquelles v peut appartenir.*

Il nous faut auparavant donner quelques définitions nécessaires à la mise en œuvre de l'algorithme. Soit p un sommet d'une enveloppe convexe inférieure $i = ECI(S)$ et v un point (fig. 2.12). Le point p est

- *concave* (relativement au segment $[p, v]$ et à i) si v est dans la région **(I)**;
- *tangent* si le point v est dans la région **(II)**;
- *réflexe* si le point v est dans la région **(III)**.

La région **(I)** est ouverte et la région **(III)** est fermée.

Soient I_1 et I_2 deux enveloppes convexes inférieures disjointes, i.e. séparées par une droite verticale d. Nous allons démontrer qu'avec la représentation précédemment définie, il est possible de calculer la fonction $\textsc{Pont}(I_1, I_2)$ efficacement .

Lemme 2.8. *Soient I_1 et I_2 deux enveloppes convexes inférieures disjointes. La fonction $\textsc{Pont}(I_1, I_2)$ se calcule en temps $O(\log N)$ où N est le nombre total de points.*

Preuve. Soient donc deux enveloppes convexes inférieures $I_1 = Inf(S_1)$ et $I_2 = Inf(S_2)$ disjointes (i.e. séparées par une droite verticale d). Nous pouvons alors déterminer le pont $[p_1, p_2]$ entre I_1 et I_2 qui permet de fabriquer l'enveloppe convexe $Inf(S_1 \cup S_2)$ comme étant l'unique segment $[p_1, p_2]$ reliant un sommet de I_1 à un sommet de I_2 et tel que p_1 soit un point tangent pour $[p_1, p_2]$ et I_2, et p_2 soit aussi un point tangent pour $[p_1, p_2]$ et I_1. Soit q_1 un sommet de I_1 et q_2 un sommet de I_2. Il y a 9 cas à envisager selon la nature respective des points q_1 et q_2 relativement au segment $[q_1, q_2]$ et relativement à I_2 et I_1 respectivement.

Ces 9 cas sont représentés sur la figure 2.13.

Sont hachurées les parties de I_1 et I_2 non candidates à contenir les piliers du pont. La figure est suffisamment explicite; seul le cas où q_1 et q_2 sont tous les

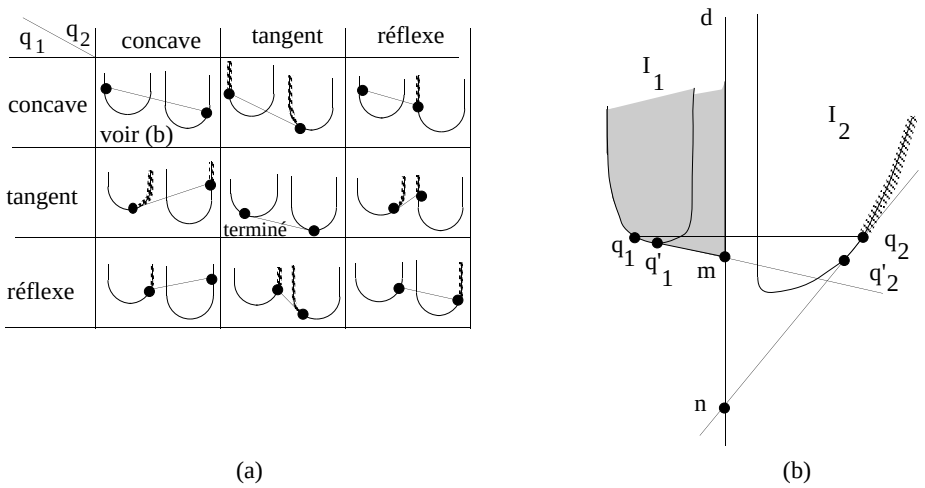


Figure 2.13: *Les* 9 *cas du lemme.*

deux concaves est un peu plus complexe. Il faut pour cela considérer la droite $d(q_1, q_1')$ où q_1' est le successeur de q_1 sur I_1 et la droite $d(q_2,\ q_2')$ où q_2' est le prédecesseur de q_2 sur I_2. Soit d une droite verticale séparant I_1 et I_2. La droite $d(q_1, q_1')$ (resp. $d(q_2, q_2')$) coupe d en m (resp. n). Si m est au-dessus au sens large de n (resp. au dessous au sens large) alors on peut hachurer la partie de I_2 à droite de q_2 (resp. la partie de I_1 à gauche de q_1). En effet pour tout point q'' de I_2 situé à droite de q_2 et tout point q' de I_1, le segment $[q', q'']$ est situé dans l'ensemble convexe limité par la partie de I_1 à gauche de q_1, les segments $[q_1, m]$, $[m, q_2]$ et la partie de I_2 à droite de q_2. Donc q'' est concave relativement à $[q', q'']$ et I_2, car $[q', q'']$ coupe d au-dessus de m alors que $[q_2, q_2']$ coupe d en n qui est au-dessous de m.

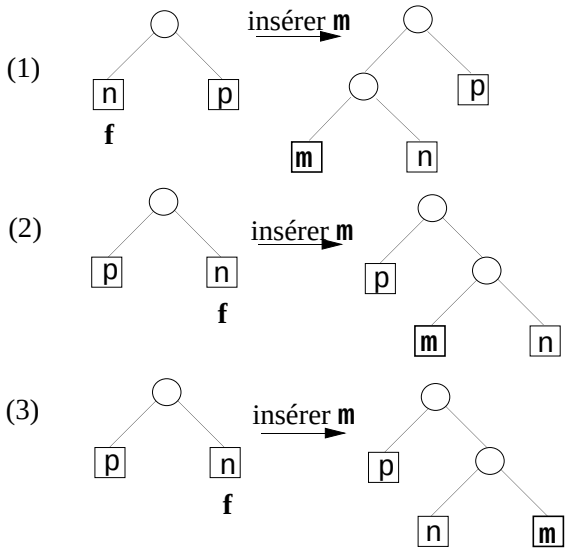


Figure 2.14: *Insertion d'un nouveau point* m.

Ainsi si le processus de recherche de $\textsc{Pont}(I_1, I_2)$ part des racines des arbres représentant les listes concaténables associées, ces arbres étant équilibrés, ils sont de hauteur au plus $\log N$ et $\textsc{Pont}(I_1, I_2)$ se calcule alors en temps $O(\log N)$. Notons qu'il faut prendre la précaution d'avoir accès dans la liste concaténable au voisin d'un sommet en temps $O(1)$. ∎

Il nous reste à décrire maintenant le processus de mise à jour de la structure de données décrivant de manière dynamique une enveloppe convexe inférieure, lors d'une insertion d'un nouveau point ou d'une suppression.

Prenons le cas d'une insertion. Les points sont rangés aux feuilles d'un arbre binaire complet équilibré. Il faut donc modifier les algorithmes classiques d'insertion : soit m le point à insérer et f la feuille à laquelle la descente aboutit en partant de la racine, f contenant le point n. Si f est une feuille gauche alors m est à gauche de n et on insère m selon le schéma (1) ; si f est une feuille droite, si m est à gauche de n alors on réalise (2) sinon on réalise (3) (figure 2.14).

```
procédure DESCENDRE(v, m);
  si v n'est pas une feuille alors
    (Q_G, Q_D) :=SCINDER(Inf(v), p_1(v), p_2(v));
    Inf(filsG(v)) :=CONCATÉNER(Q_G, Q(filsG(v)));
    Inf(filsD(v)) :=CONCATÉNER(Q(filsD(v)), Q_D);
    si m ≺ cle(v) alors DESCENDRE(filsG(v), m)
    sinon DESCENDRE(filsD(v), m)
    finsi;
  INSÉRER(m).
```

Préoccupons nous maintenant de la mise à jour des informations contenues dans les sommets de l'arbre en supposant pour l'instant, pour simplifier, que l'insertion réalisée ne demande pas de rééquilibrage.

La mise à jour se fait en deux temps. Lors de la descente dans l'arbre pour insérer une nouvelle feuille, pour chaque sommet v situé sur le chemin c de descente, on calcule $\mathit{Inf}(v)$ et $\mathit{Inf}(\mathit{frère}(v))$ inductivement et grâce aux points $p_1(v)$, $p_2(v)$ on transmet aux fils de v les lignes polygonales respectives qui leur manquent (figure 2.11) pour former l'enveloppe convexe qui leur est associée. Ainsi au moment de l'insertion du nouveau point, on dispose pour chaque sommet v du chemin c ainsi que pour ses frères, de l'enveloppe convexe correspondante.

Après insertion, la remontée à la racine (c'est ici que se situe un éventuel rééquilibrage) permet de recalculer inductivement $Q(v)$ pour tous les sommets v de c, et $p_1(v)$, $p_2(v)$ pour tous les sommets v de c ainsi que pour leurs frères (effectivement, on peut remarquer que la mise à jour des informations ne concerne que les sommets de c pour ce qui est de p_1 et p_2, mais concerne aussi les sommets frères pour ce qui est de Q). La mise à jour se réalise alors ainsi : si en un sommet v on

dispose de *Inf*(v) ainsi que de *Inf*$(frère(v))$ alors l'opération PONT permet de calculer $Q(v)$, $Q(frère(v))$ ainsi que *Inf*$(frère(v))$, $p_1(père(v))$ et $p_2(père(v))$, et donc inductivement permet la mise à jour des fonctions Q, p_1 et p_2. Ces deux opérations de descente et de montée sont décrites plus précisément dans les procédures DESCENDRE et MONTER, sans toutefois prendre en compte le rééquilibrage. Dans ces procédures, la variable v représente un sommet de l'arbre, et m est le point que l'on insère dans l'arbre.

```
procédure MONTER(v);
  si v ≠ racine alors
    (Q1, Q2, Q3, Q4, J1, J2) :=PONT(Inf(v), Inf(frère(v)));
    Q(v) := Q2;
    Q(frère(v)) := Q3; Inf(père(v)) :=CONCATÉNER(Q1, Q4);
    p1(père(v)) := J1, p2(père(v)) := J2;
    MONTER(père(v))
  sinon Q(v) :=Inf(v).
```

Examinons maintenant le mécanisme de rééquilibrage (par la méthode des arbres AVL). Prenons le cas d'une rotation droite par exemple (figure 2.15). Soient

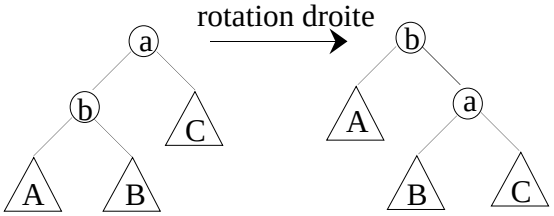


Figure 2.15: *Rotation droite.*

a', b', c' les racines des arbres A, B, C. La mise à jour des informations stockées dans les nœuds a, b, a', b', c' se fait par la suite d'instructions suivantes :

$(Q_1, Q_2, Q_3, Q_4, J_1, J_2)$ =PONT(*Inf*(B),*Inf*(C));
Inf(a) =CONCATÉNER(Q_1, Q_4); $Q(b') = Q_2$; $Q(c') = Q_3$;
$p_1(a) = J_1$; $p_2(a) = J_2$;
$(Q'_1, Q'_2, Q'_3, Q'_4, J'_1, J'_2)$ =PONT(*Inf*(A),*Inf*(a));
Inf(b) =CONCATÉNER(Q'_1, Q'_4); $Q(a') = Q'_2$; $Q(a) = Q'_3$;
$p_1(b) = J'_1$; $p_2(b) = J'_2$.

On peut remarquer que $Q(b)$ sera calculé au niveau de son père. Ainsi une rotation prend un temps $O(\log N)$.

Le procédé d'insertion est illustré par l'exemple ci-dessous (figures 2.16 et 2.17). La figure 2.16 représente l'arbre binaire équilibré dont les feuilles contiennent les points dont on doit calculer l'enveloppe convexe inférieure. On a représenté les points par des entiers qui correspondent à l'ordre dans lequel on a inséré les

points dans l'arbre. En chaque nœud v sont indiqués $Q(v)$ et $p_1(v)$. La figure 2.17 représente l'insertion d'un nouveau point. Les informations qui ont été recalculées au cours de cette insertion sont indiquées en gras et soulignées.

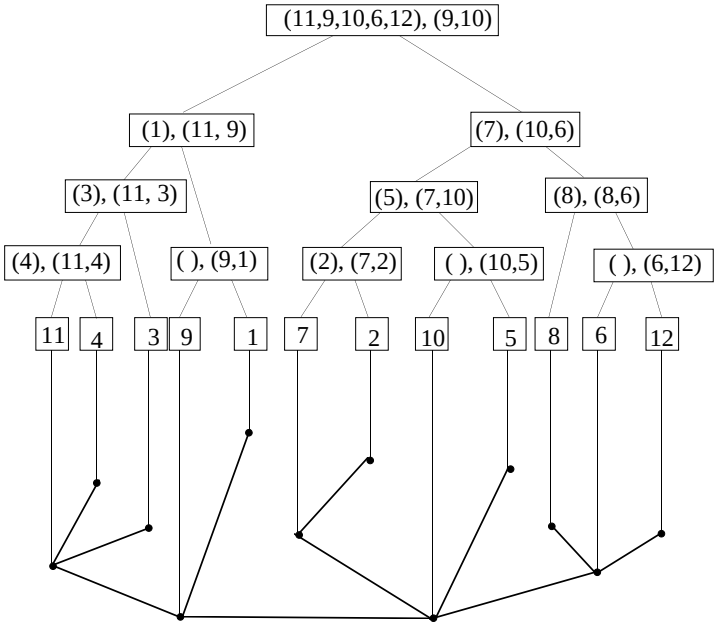


Figure 2.16: *Structure de l'arbre avant insertion du point* p_{13}.

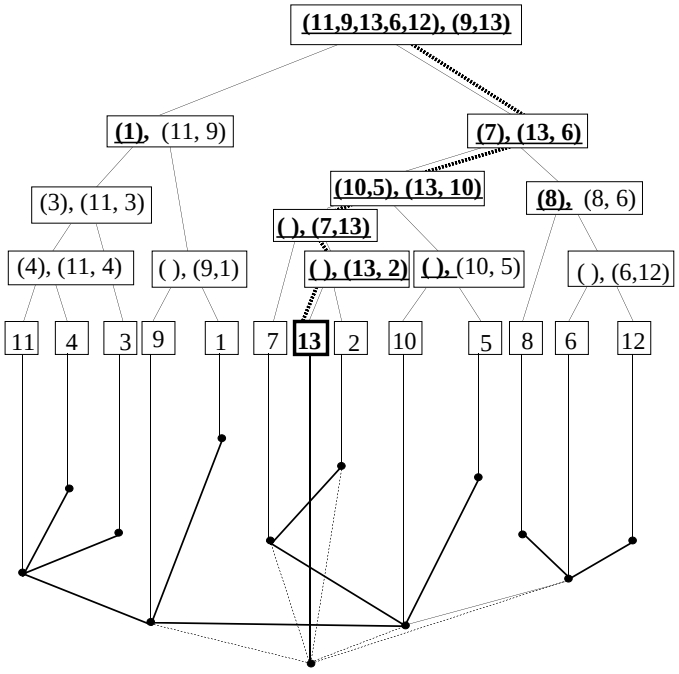


Figure 2.17: *Insertion du point* p_{13}.

Nous laissons au lecteur le soin de traiter le cas de la suppression d'un point

(exercice) qui est identique en ce qui concerne le mécanisme de mise à jour.

Analysons la complexité en temps de chacune de ces procédures. Rappelons que les opérations SCINDER et CONCATÉNER se réalisent en temps $O(\log k)$ où k est le nombre total d'éléments. Si N est le nombre de feuilles de l'arbre binaire, on a $k \leq N$; la hauteur de l'arbre étant $O(\log N)$, on en déduit que la procédure DESCENDRE, comme la procédure MONTER prennent un temps $O((\log N)^2)$. D'où le théorème :

Théorème 2.9. *Les enveloppes convexes inférieure et supérieure d'un ensemble de N points ont une mise à jour dynamique qui prend dans le pire des cas un temps $O((\log N)^2)$ par insertion ou suppression.*

Bien sûr cet algorithme donne un temps de calcul de l'enveloppe convexe de N points en temps $O(N(\log N)^2)$, ce qui est un temps moins bon que celui des algorithmes vus précédemment. Cela s'explique par le fait que la possibilité de mise à jour efficace par adjonction ou suppression d'un point nécessite l'utilisation de structures de données plus complexes, et donc un peu plus lourdes à manipuler.

11.3 Localisation de points dans le plan

Les problèmes de recherche en géométrie, bien que proches des problèmes de recherche classiques, présentent des particularités propres, liées précisément à leur nature géométrique et à la complexité des objets traités.

De manière très générale le problème de localisation est le suivant :

Données : L'espace est divisé en k régions $R_1, \ldots, R_k$.

Question : Soit m un point de l'espace. A quelle région appartient-il ?

La question est destinée à être posée pour un grand nombre de points. Il est donc intéressant de prétraiter les données de manière à obtenir un algorithme efficace pour la localisation de m. Les paramètres qui mesurent cette efficacité sont donc :

- le temps du prétraitement ;
- l'espace utilisé pour représenter les données ;
- le temps de réponse à la question.

Il y a encore peu de résultats sur ce problème en dimension 3 ou supérieure, par contre c'est un problème assez bien résolu dans le plan, du moins dans un cas simple, celui des subdivisions planaires. Il est certain que la nature de la partition détermine la complexité de la tâche.

Nous ne traiterons ici que le cas du plan. Considérons tout d'abord le problème élémentaire suivant :

Problème P_1 **:** Soit P un polygone simple (P divise le plan en deux régions). Etant donné un point z, z est-il intérieur à P ?

On peut encore simplifier le problème :

Problème P_2 **:** Soit P un polygone convexe. Un point z étant donné, le point z est-il intérieur à P ?

11.3.1 Cas d'un polygone simple

Un algorithme simple sans prétraitement permet de résoudre le problème P_1 en temps linéaire (en fonction du nombre de sommets du polygone).

Théorème 3.1. *Soit n le nombre de sommets d'un polygone simple P. On peut déterminer si un point est à l'intérieur de P sans prétraitement en temps $O(n)$.*

Preuve. On teste en temps $O(n)$ si z appartient au contour de P. Supposons donc que z n'appartient pas au contour de P. L'idée est simple : soit d une demi-droite d'origine z. Alors z est intérieur à P si et seulement si le nombre de points d'intersection de d avec le contour de P est impair ; en effet chaque fois que d coupe un côté de P, d passe de l'intérieur à l'extérieur de P ou viceversa et donc pour passer de z interne à P à « l'autre extrémité » de d qui est externe à P, il faut couper un nombre impair de côtés (figure 3.1). Tout le problème est de dénombrer

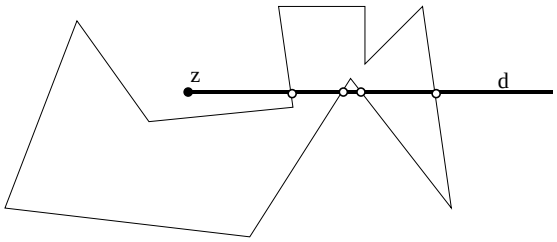


Figure 3.1: *Principe de l'algorithme.*

correctement les intersections en tenant compte des cas de dégénérescence. On voit clairement, dans les cas dégénérés indiqués dans la figure 3.2, quel est le nombre d'intersections qu'il faut compter. L'idée intuitive est qu'un déplacement « infinitésimal » de tous les points du polygone vers le haut ne modifie pas le résultat (i.e. le fait que z soit interne ou non à P). Ce déplacement infinitésimal permet d'éliminer les cas de dégénérescence. Les multiplicités indiquées sur le schéma sont réalisées lorsque l'on considère comme « intersectant d » uniquement les côtés de P non horizontaux qui rencontrent d mais dont l'extrémité inférieure a une ordonnée strictement inférieure à celle de z. D'où l'algorithme :

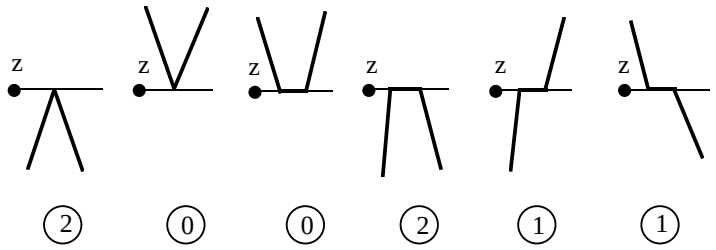


Figure 3.2: *Le décompte des intersections.*

procédure LOCALISER (z, P);
  Soit d une demi-droite horizontale d'origine z;
  $s := 0$;
  pour chaque côté non horizontal c de P faire
    si c rencontre d et l'extrémité inférieure de c est strictement
      inférieure à celle de z
    alors $s := s + 1$ finsi;
  si s est pair alors z est à l'extérieur de P sinon z est interne à P.

Il est clair que cet algorithme s'exécute en temps proportionnel au nombre de côtés de P.

11.3.2 Cas d'un polygone simple convexe

Nous allons voir maintenant comment traiter le problème P_2 en temps plus efficace grâce à un prétraitement qui utilise la convexité de P.

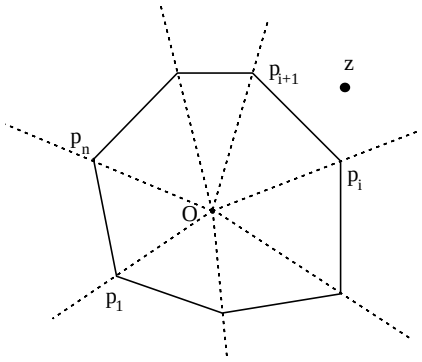


Figure 3.3: *Cas d'un polygone convexe.*

Théorème 3.2. *Soit P un polygone convexe ayant n sommets. Grâce à un prétraitement en temps $O(n \log n)$ on détermine si un point appartient à l'intérieur de P en temps $O(\log n)$ et en espace $O(n)$.*

Preuve. Soit $((p_1, \ldots, p_n))$ le contour direct de P. Si O est un point intérieur à P, le plan est divisé en n secteurs angulaires $(\overrightarrow{Op_i}, \overrightarrow{Op_{i+1}})$ (voir figure 3.3). Si on a déterminé à quel secteur angulaire z appartient, on détermine en temps $O(1)$ si z est intérieur à P ou non en repérant si z est à gauche ou à droite de la droite orientée $\overrightarrow{d}(p_i, p_{i+1})$. Le calcul du secteur angulaire se fait par dichotomie de la manière suivante :

```
procédure SECTEUR-ANGULAIRE(z, O, P);
  si z est entre p_n et p_1 alors terminé
  sinon i := 1; j := n;
    tantque j > i + 1 faire
      k := (i + j)div2;
      si z est entre p_i et p_k alors j := k
      sinon i := k
    fintantque
  finsi.
```

D'où l'algorithme complet :

```
procédure LOCALISER-POL-SIMPLE-CONVEXE(z, P);
(1) choisir un point intérieur à P;
(2) déterminer le secteur angulaire (Op_i, Op_{i+1}) contenant z;
(3) si z est à droite de p_i p_{i+1} alors
        z est externe à P
    sinon z est interne à P.
```

L'étape (1) prend un temps $O(1)$: il suffit de prendre le milieu de deux sommets non consécutifs de P.

L'étape (2) prend un temps $O(\log n)$ puisqu'on procède par dichotomie. La boucle «tant que» est exécutée $O(\log n)$ fois (on rappelle que la relation z est entre a et b équivaut à $(a \prec z$ et $z \prec b)$ ou $(z \prec b$ et $b \prec a)$ ou $(b \prec a$ et $a \prec z))$. L'étape (3) prend un temps $O(1)$. Enfin, l'implémentation de (2) peut se faire à l'aide d'un tableau et donc l'espace nécessaire est $O(n)$. ∎

11.3.3 Cas d'une subdivision planaire généralisée

Etudions maintenant le problème dans le cas d'une subdivision planaire quelconque.

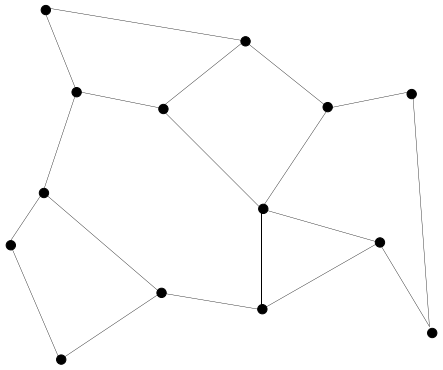


Figure 3.4: *Une subdivision planaire.*

Une *subdivision planaire* est une partition du plan en régions dont les contours sont des polygones simples, une et une seule de ces régions étant non bornée. Une *subdivision planaire généralisée* autorise les demi-droites et les droites comme côtés des polygones. La subdivision de la figure 3.5 possède 9 régions dont 3 sont bornées (deux côtés ne peuvent se couper qu'en une extrémité). Le problème de

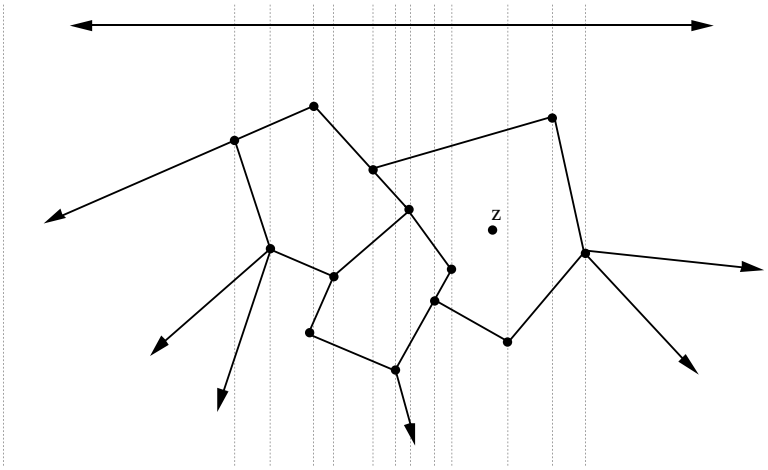


Figure 3.5: *Une subdivision planaire généralisée.*

localisation s'énonce donc de manière précise comme suit :

Données : Une subdivision planaire généralisée à n côtés.

Question : Si z est un point du plan, déterminer la région du plan à laquelle il appartient.

Le principe de l'algorithme que nous allons exposer est le suivant : par chaque sommet de la subdivision on fait passer une droite verticale; le plan est alors divisé en bandes verticales. Les côtés de la subdivision coupant une bande donnée peuvent être ordonnés de bas en haut (car deux côtés ne peuvent se couper à l'intérieur d'une bande). La localisation d'un point se ramène alors à deux recherches dichotomiques successives.

En temps $O(\log n)$ on détermine dans quelle bande est le point z, puis à l'intérieur de la bande on détermine en temps $O(\log n)$ entre quels segments le point z est situé, ce qui permet d'identifier la région.

Examinons alors soigneusement quel doit être le prétraitement. On suppose que la partition est donnée de telle sorte que pour chaque côté on connaisse en temps $O(1)$ les noms des deux régions situées de part et d'autre, et on suppose aussi que l'on connait le contour de chaque région.

Un prétraitement «brutal» consisterait à construire un arbre binaire de recherche pour chaque bande ce qui prendrait un espace $O(n^2)$ car chaque bande peut être coupée par $O(n)$ côtés. Mais on peut observer que les arbres binaires de recherche associés à deux bandes voisines comportent une certaine quantité d'information commune. Le prétraitement consiste donc à balayer le plan de gauche à droite par une droite verticale; la position initiale étant à $-\infty$, un arbre persistant contient les côtés de la subdivision coupés par la droite. La mise à jour se fait à chaque passage par le sommet suivant s de la subdivision (les sommets sont triés par ordre croissant d'abscisse). Au cours de cette mise à jour, on supprime les côtés d'extrémité droite s et on insère les côtés d'extrémité gauche s. Le prétraitement demande donc un temps $O(n \log n)$ et un espace $O(n)$.

Théorème 3.3. *Le problème de la localisation d'un point dans une subdivision à n côtés se résout en temps $O(\log n)$ par un prétraitement prenant un temps $O(n \log n)$ et un espace $O(n)$.*

On peut se demander si ce résultat est optimal. Si la partition se réduit à une bande horizontale formée de n rectangles, la localisation se ramène à la localisation d'un réel x dans une suite ordonnée $x_1 < \ldots < x_n$, ce qui prend un temps $\Omega(\log n)$ par une recherche binaire. Pour ce qui est du prétraitement, il nécessite clairement le rangement des données qui occupe un espace $O(n)$. Reste le temps du prétraitement qui peut être amélioré (pas dans cette version certainement puisqu'elle nécessite un tri préliminaire des sommets de la subdivision). Il existe un autre algorithme fondé sur la triangulation des polygones simples qui a un temps de prétraitement linéaire et qui conserve un espace linéaire et un temps de réponse logarithmique, mais il est beaucoup plus complexe et plus difficile à mettre en oeuvre.

11.4 Diagrammes de Voronoï

11.4.1 Diagrammes de Voronoï de points

Soit S un ensemble fini de points du plan P. Traditionnellement, les éléments de S sont appelés des *sites*.La *région de Voronoï* d'un site $a \in S$ est l'ensemble

$$R(a) = \{x \in P \mid d(a,x) \leq d(b,x) \text{ pour tout } b \in S\} \tag{4.1}$$

C'est donc l'ensemble des points du plan qui sont plus proches de a que de tout autre site. On peut donner une formulation plus compacte en introduisant les demi-plans

$$H_{a,b} = \{x \in P \mid d(a,x) \leq d(b,x)\}$$

pour $a, b \in S$, et $a \neq b$. Alors

$$R(a) = \bigcap_{b \neq a} H_{a,b}$$

Cette formule montre que la région de Voronoï $R(a)$ est convexe : c'est un *polytope convexe* (i. e. par définition une intersection finie de demi-plans). Une région de Voronoï est soit bornée (auquel cas c'est un polygone convexe), soit non bornée. Dans tous les cas, sa frontière est une ligne polygonale.

On appelle *diagramme de Voronoï* de S, et on note $\mathrm{Vor}(S)$, la réunion des frontières des régions de Voronoï de S. Le diagramme est formé de *sommets* qui sont les sommets des lignes polygonales des régions, et *d'arêtes*, qui sont les arêtes de ces lignes polygonales (voir figure 4.1). Comme cette terminologie le suggère,

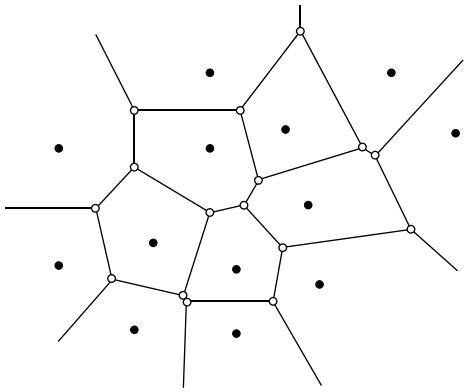


Figure 4.1: *Diagramme de Voronoï : les sites sont pleins.*

on peut aussi considérer le diagramme de Voronoï comme *graphe* abstrait, avec les sommets et les arêtes ainsi définis. Comme certaines frontières de régions de Voronoï ne sont pas bornées, les arêtes non bornées n'ont qu'une seule extrémité (mais on peut facilement remédier à cela en introduisant un sommet *à l'infini* dans la direction de l'arête).

Tout point x du diagramme de Voronoï est équidistant de deux sites au moins, et la distance de x à ces sites est minimale; donc $x \in \mathrm{Vor}(S)$ si et seulement s'il existe $a, b \in S$, $a \neq b$, tels que

$$d(a,x) = d(b,x) \leq d(c,x) \qquad \text{pour tout site} \quad c \in S$$

Un sommet du diagramme est un point qui est équidistant d'au moins trois sites à distance minimale. Revenons aux régions :

Proposition 4.1. *Une région de Voronoï $R(a)$ est non bornée si et seulement si a est un sommet de l'enveloppe convexe de S.*

Preuve. Supposons d'abord que $R(a)$ n'est pas bornée. Comme $R(a)$ est convexe, elle contient une demi-droite infinie D d'origine a. Si a n'est pas un sommet de l'enveloppe convexe de S, alors D intersecte l'enveloppe convexe en une arête $[b, c]$ de l'enveloppe convexe (voir figure 4.2). Mais alors tout point sur D qui est suffisamment loin, est plus proche de b (ou de c) que de a, contrairement à la définition de $R(a)$.

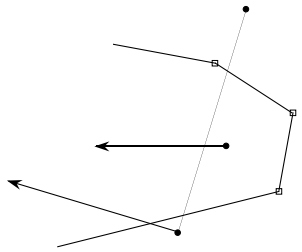


Figure 4.2: *Tout point suffisamment loin sur D est plus proche de b que de a.*

Réciproquement, si a est un sommet de l'enveloppe convexe de S, soient b et c ses voisins sur l'enveloppe. Tout point x qui est dans le secteur angulaire délimité par les demi-droites perpendiculaires aux segments $[b, a]$ et $[a, c]$ issues de a et orientées vers l'extérieur de l'enveloppe convexe (voir figure 4.3) appartient à la région de Voronoï $R(a)$, donc cette région n'est pas bornée. ∎

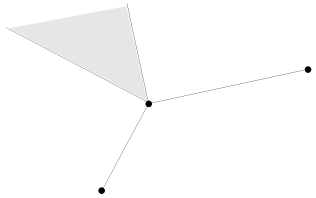


Figure 4.3: *La partie grisée est dans $R(a)$.*

Proposition 4.2. *Le diagramme de Voronoï* $\mathrm{Vor}(S)$ *d'un ensemble fini S est planaire en tant que graphe; de plus, si S ne contient pas quatre points cocycliques, tout sommet de* $\mathrm{Vor}(S)$ *est de degré* 3.

Preuve. Puisque $\mathrm{Vor}(S)$ est constitué des frontières des régions de Voronoï l'intersection de deux arêtes est par définition un sommet du graphe. Ceci prouve que le graphe est planaire.

Soit maintenant u un sommet de $\mathrm{Vor}(S)$, et soient $a_1, \ldots, a_k$ les sites tels que $d(u, a_1) = \cdots = d(u, a_k) \leq d(u, b)$ pour tout $b \in S - \{u, a_1, \ldots, a_k\}$. Les points $a_1, \ldots, a_k$ sont sur le cercle de centre u et de rayon $d(u, a_1)$, donc $k = 3$. ∎

Soit S un ensemble de sites, et soit x un point du plan. L'*amplitude*de x («clearance» en anglais) est le nombre

$$\delta(x) = \min\{d(x,a) \mid a \in S\}$$

C'est le rayon du plus grand disque de centre x et dont l'intérieur est disjoint de S. Ce disque est appelé le *disque de Delaunay* de x, et le cercle correspondant son *cercle de Delaunay*. On peut alors définir de manière équivalente le diagramme de Voronoï comme l'ensemble des points du plan dont le cercle de Delaunay contient au moins deux éléments de S, et les sommets du diagramme comme les points dont le cercle contient au moins trois sites.

Lorsque S ne contient pas quatre points cocycliques, on peut associer à S une triangulation appelée la triangulation de Delaunay, et que nous définissons maintenant. De manière générale, on appelle *triangulation* de S un ensemble de triangles vérifiant :

(1) la réunion des triangles est égale à l'enveloppe convexe de S;
(2) les intérieurs des triangles sont deux à deux disjoints;
(3) les sommets des triangles sont les éléments de S.

Soit p un sommet de $\mathrm{Vor}(S)$. Comme p est de degré 3, il existe exactement trois sites appartenant au cercle de Delaunay de p. Ces trois sites constituent les sommets d'un triangle $T(p)$ inscrit dans le cercle de Delaunay de p. Il est facile de vérifier que l'ensemble des triangles $T(p)$ constitue une triangulation de S appelée la *triangulation de Delaunay* de S.

11.4.2 L'algorithme de Fortune

Dans cette section, nous décrivons un algorithme pour calculer le diagramme de Voronoï d'un ensemble S fini de points (les «sites») dans le plan. L'algorithme est un algorithme de balayage. La construction est plus facile à comprendre lorsqu'on l'interprète dans l'espace $\mathbb{R}^3$. Le plan contenant S est le plan P d'équation $z = 0$. L'espace $\mathbb{R}^3$ est balayé par un plan d'équation $x + z = t$.

A chaque site $a \in S$ est associé un demi-cône C_a d'axe vertical (on parlera plus simplement de cône), défini par

$$C_a = \{(x,y,z) \mid d(a,(x,y)) = z\}$$

Ainsi, la hauteur z d'un point sur ce cône est égale à la distance de ce point à sa projection (x,y) sur le plan P.

Soient a et b deux sites, et C_a et C_b les cônes correspondants. Ces cônes s'intersectent en une branche d'hyperbole H. La hauteur z d'un point (x,y,z) sur cette hyperbole est égale à la fois à $d(a,(x,y))$ et à $d(b,(x,y))$; en d'autres termes, la projection (orthogonale) de H est la médiatrice du segment $[a,b]$. Autrement dit, cette hyperbole est l'intersection commune de C_a et de C_b avec le plan vertical défini par la médiatrice du segment $[a,b]$.

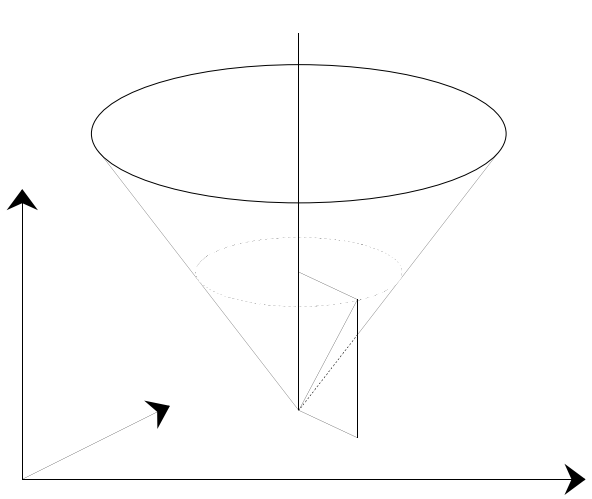


Figure 4.4: *Demi-cône associé au site a.*

Supposons maintenant que les cônes C_a pour $a \in S$, sont «opaques», et «regardons-les d'en bas». L'intersection de deux cônes associés à des sites a et b se projette en une droite, mais tous les points de cette droite ne sont pas visibles : un point p sera caché par un autre cône si le site de ce cône est plus proche de p que a et b ne le sont. Le segment de droite visible est donc délimité par des points projection de l'intersection de trois cônes.

Traduisons ceci en posant

$$C = \{(x, y, z) \mid z = \min_{a \in S} d(a, (x, y))\}$$

L'ensemble C est exactement la surface de tous les points visibles, puisque, pour chaque (x, y), on choisit le point de hauteur z minimale. Soit B l'ensemble des points de C qui appartiennent à au moins deux cônes. Alors on a :

Proposition 4.3. *La projection orthogonale de l'ensemble B des points qui appartiennent à au moins deux cônes de la famille C_a ($a \in S$) est égale au diagramme de Voronoï de S.*

Preuve. Soit $p = (x, y, z)$ un point de B. S'il appartient à la fois à C_a et à C_b pour deux sites distincts $a, b \in S$, alors le point $p' = (x, y)$ dans P est équidistant de a et de b, et cette distance est z. Donc (x, y) appartient au diagramme de Voronoï de S. Réciproquement, si $p' = (x, y)$ appartient au diagramme de Voronoï de S, on a $d(a, p') = d(b, p') = \min_{c \in S} d(c, p')$ pour deux points a, b de S. Le point $p = (x, y, d(a, p'))$ appartient à B. ∎

Front parabolique

L'algorithme de calcul du diagramme de Voronoï est un algorithme de balayage, par un plan oblique, qui est incliné de sorte à être tangent aux cônes quand il

passe par leur sommet. Plus précisément, à tout instant $t \in \mathbb{R}$, on considère le *plan de balayage* P_t d'équation $x + z = t$. Il intersecte donc le plan P contenant S en la droite Δ_t d'équation $x = t$. Appelons *ligne de balayage* cette droite.

Le plan balaye l'espace pour t croissant. L'intersection d'un cône C_a et du plan de balayage P_t est vide tant que le plan se trouve à gauche du cône, c'est-à-dire tant que t est plus petit que l'abscisse x_a de a. Au moment où le plan touche le cône, et est tangent au cône, l'intersection est une demi-droite passant par a; on peut voir cette demi-droite comme une parabole dégénérée. Ensuite, l'intersection est une parabole, et cette parabole s'«ouvre» au fur et à mesure que t croît. La projection de la parabole sur le plan P est une parabole $\pi_t(a)$ de foyer a, et de directrice Δ_t.

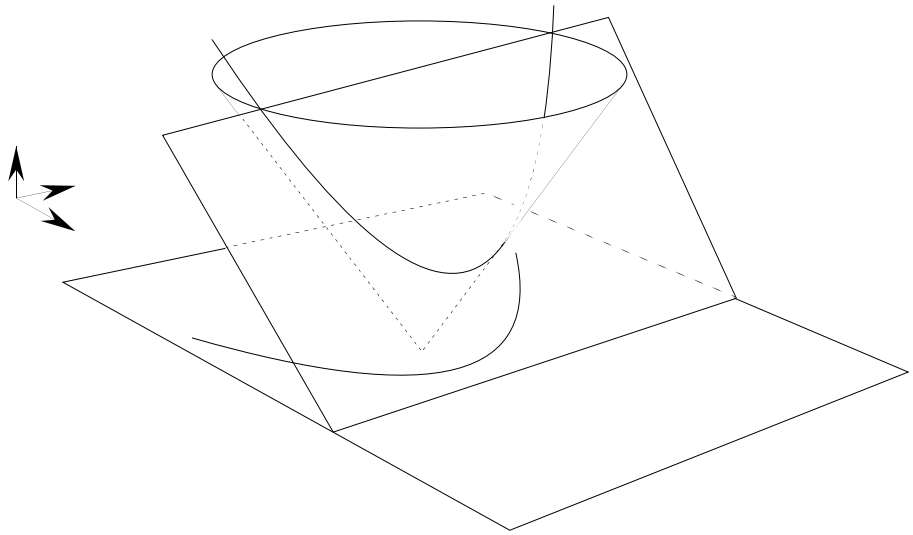


Figure 4.5: *Le plan de balayage.*

Supposons à nouveau les cônes C_a opaques, supposons de plus P_t opaque, et regardons à nouveau les surfaces d'en bas. Le plan P_t occulte tous les sites qui se trouvent à droite de Δ_t, donc les cônes associés à ces sites. Les cônes des sites déjà balayés donnent un ensemble de paraboles, toutes de même directrice, et de largeur variable. Seules certaines de ces paraboles sont en partie visibles, à savoir tant qu'elles ne sont pas entrées à l'intérieur d'autres cônes. Plus précisément, l'intersection de la surface C et du plan P_t se projette en une courbe F_t qui est une union d'arcs de parabole de même directrice Δ_t. Ces paraboles ont pour foyers des sites d'abscisse inférieure à t. La courbe F_t est appelée le *front parabolique* à

l'instant t.

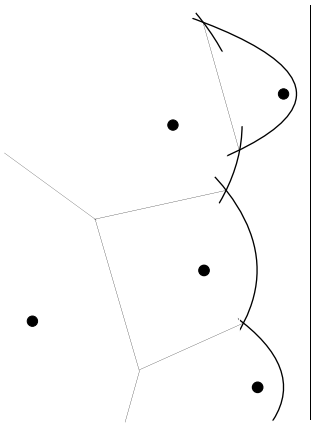


Figure 4.6: *Le front parabolique au temps t.*

Toute droite horizontale du plan P intersecte le front parabolique F_t en un point unique. Par ailleurs, F_t possède des *points anguleux*. Ce sont des points d'intersection de deux arcs de parabole consécutifs sur le front. Ils correspondent à la projection d'un point de B, et appartiennent donc au diagramme de Voronoï de S. La proposition suivante montre la réciproque.

Proposition 4.4. *Tout point du diagramme de Voronoï de S est point anguleux d'un front parabolique F_t pour un réel t.*

Preuve. Soit u une arête du diagramme de Voronoï de S, et soit p un point sur u. L'arête u est un intervalle sur la médiatrice de deux sites a et b. Posons $r = d(a,p) = d(b,p)$, et considérons la droite de balayage Δ_t située à droite de p et telle que $d(p,\Delta_t) = r$. En d'autres termes, Δ_t est tangente, en un point p', au cercle Π de centre p et de rayon r, cercle qui passe par a et b. Ce cercle ne contient, par définition du diagramme de Voronoï aucun autre site en son intérieur.

Le point p appartient aux paraboles $\pi_t(a)$ et $\pi_t(b)$. Supposons que p n'appartient pas au front parabolique F_t. Alors une autre parabole $\pi_t(c)$ passe strictement entre p et Δ_t, et coupe donc l'intervalle $]p,p'[$ en un point q. Or le site c est équidistant de p' et de q, donc se trouve sur le cercle Π' de centre q et de rayon qp'. Mais alors Π' est à l'intérieur de Π, et c est à l'intérieur de Π, ce qui est impossible. ∎

Il résulte de cette proposition qu'il «suffit» de balayer continûment l'espace par le plan P_t pour obtenir l'ensemble des points constituant le diagramme de Voronoï comme points anguleux des fronts paraboliques.

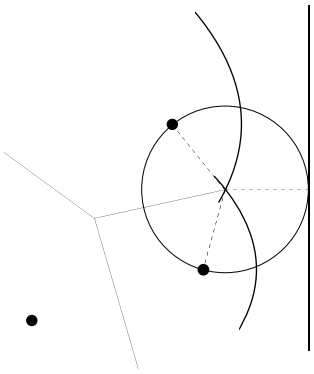


Figure 4.7: *Tout point du diagramme de Voronoï est point anguleux.*

Evénements

En fait, comme le diagramme de Voronoï est formé d'un nombre fini de points et de segments, on peut se contenter de calculer un nombre fini de données. De manière analogue, la suite des sites relatifs aux arcs du front parabolique ne change qu'en un nombre fini d'instants t, lorsque t varie de $-\infty$ à $+\infty$; ce sont ces instants qui constituent les «événements» dont nous parlons maintenant. Un *événement* est un instant t où un arc de parabole apparaît ou disparaît sur le front parabolique F_t.

Apparition d'un nouvel arc de parabole

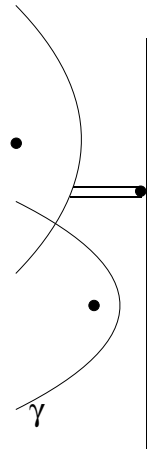


Figure 4.8: *Apparition d'un nouvel arc α.*

Lorsque la droite de balayage Δ_t passe par un site a, une nouvelle parabole $\pi_t(a)$ apparaît. Au départ, cette parabole est dégénérée, et est réduite à une demi-droite horizontale issue de a. Cette parabole (demi-droite) intersecte le front parabolique en un seul arc, ou alors en deux arcs consécutifs ; dans ce cas, le

point d'intersection est un point anguleux. Ces situations sont les seules où peut apparaître un nouvel arc parabolique :

Proposition 4.5. *Un nouvel arc parabolique apparaît à l'instant t dans le front parabolique si et seulement si Δ_t balaye un nouveau site.*

Preuve. Supposons le contraire. Alors un nouvel arc parabolique α apparaît dans le front parabolique à un instant t sans être créé par l'arrivée d'un nouveau site. La parabole π dont α est un arc existait donc déjà, mais était auparavant située entièrement à gauche du front parabolique. Au moment de l'apparition de l'arc α dans le front, la parabole π est

- soit tangente à un arc de parabole du front ;
- soit passe par un point anguleux du front.

Les deux situations mènent à une contradiction. En effet, si π est tangente à un arc β d'une parabole π', cela signifie que π a un rayon de courbure inférieur à celui de π', donc que son foyer est plus proche de Δ_t que celui de π'. Or π est contenue (au sens large) dans π', donc son sommet est plus éloigné de Δ_t que le sommet de π'. Ces deux conditions sont contradictoires.

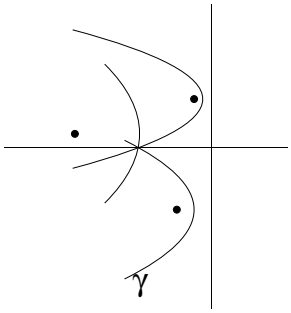


Figure 4.9: *Apparition de l'arc α : deuxième cas impossible.*

Pour montrer que le deuxième cas est également impossible, il nous faut un peu de calcul, et des notations (voir figure 4.9). Notons (x_s, y_s) les coordonnées d'un point s. Soient β et γ les deux arcs de parabole dont l'intersection définit le point anguleux $p = (x_p, y_p)$, à l'instant t où l'arc de parabole α apparaît. Soient b et c les foyers des paraboles supportant β et γ, et soit a le foyer de π. On suppose $y_b > y_p > y_c$. On a alors $x_a \leq x_b$ ou $x_a \leq x_c$, selon que $y_a \geq y_p$ ou $y_a \leq y_p$.

Considérons la droite horizontale H d'équation $y = y_p$. Au temps t, les paraboles $\pi_t(a)$, $\pi_t(b)$, et $\pi_t(c)$ intersectent H au point p. Au temps $t + \varepsilon$, pour $\varepsilon > 0$ assez petit, l'intersection de H avec $\pi_{t+\varepsilon}(a)$ se situe à droite de l'intersection de H avec $\pi_{t+\varepsilon}(b)$, et à droite de $\pi_{t+\varepsilon}(c)$. Ceci signifie qu'au temps t, la vitesse avec laquelle le point d'intersection $\pi_t(a) \cap H$ se déplace sur H est supérieure à la vitesse correspondante de $\pi_t(b) \cap H$. Calculons donc cette vitesse. L'équation d'une parabole $\pi_t(s)$ de foyer s et de directrice Δ_t est

$$(x - x_s)^2 + (y - y_s)^2 = (x - t)^2$$

ou encore $2x(t-x_s) = t^2 - x_s^2 - (y-y_s)^2$. Soit $x^{(a)}(t)$ l'abscisse du point $\pi_t(a) \cap H$. En dérivant, on obtient $2x^{(a)} + 2(dx^{(a)}/dt)(t - x_a) = 2t$, soit encore

$$\frac{dx^{(a)}}{dt} = \frac{t - x^{(a)}}{t - x_a}$$

et des formules analogues pour b et c. Au temps t, on a $x^{(a)}(t) = x^{(b)}(t) = x^{(c)}(t) = x_p$. Comme

$$\frac{dx^{(a)}}{dt} > \frac{dx^{(b)}}{dt}$$

on a donc $x_a < x_b$, et de même $x_a < x_c$, une contradiction. ∎

Disparition d'un arc de parabole

Au moment où un arc de parabole β disparaît, il est réduit à un point p qui est en fait l'intersection de trois arcs de parabole consécutifs α, β et γ. Par un argument déjà employé plus haut, on vérifie que α et γ n'appartiennent pas à la même parabole. Le point p est équidistant des foyers a, b et c des paraboles correspondantes, et de la droite de balayage Δ_t. En d'autres termes, le point p est un sommet du diagramme de Voronoï. Plus précisément, Δ_t est tangent au cercle circonscrit aux sites a, b, c de centre p, qui est un cercle de Delaunay. Nous venons de prouver :

Proposition 4.6. *Un arc disparaît du front parabolique à l'instant t si et seulement si Δ_t est tangent à un cercle de Delaunay centré sur un sommet du diagramme de Voronoï.*

Deux structures de données sont nécessaires pour mettre en œuvre l'algorithme de Fortune. Ces structures de données sont d'ailleurs toujours de même nature lorsque l'on veut réaliser un algorithme de balayage : une première structure gère les événements. Chaque événement correspond à un nombre réel, qui est l'abscisse de la droite de balayage. On extrait les éléments en ordre croissant, et on insère si nécessaire de nouveaux éléments ; il convient donc d'employer une *file de priorité*. La deuxième structure gère les objets actifs, ici les arcs du front parabolique. Ces objets sont naturellement ordonnés par ordonnées croissantes, et les opérations à effectuer sont celles d'un *dictionnaire*. On réalise donc l'implémentation en faisant appel à un arbre de recherche équilibré. Si le nombre d'événements est linéaire en fonction du nombre n de sites, il y a de fortes présomptions pour que l'algorithme coûte au total $O(n \log n)$ opérations, et une place $O(n)$. On verra qu'il en est ainsi aussi pour l'algorithme de Fortune.

File des événements

La file des événements, notée $\mathcal{E}$, contient deux types d'événements : les événements de type *site*, et les événements de type *cercle*. Chaque événement est décrit

par son type, et par un nombre réel, qui est l'instant t où il doit être extrait de la file et traité. Un événement de type site est l'arrivée de la ligne de balayage sur un nouveau site. La figure 4.10 montre un tel événement. Ces événements sont repérés et rangés par les abscisses croissantes des sites. Un événement de type cercle correspond à la disparition d'un arc de parabole. Comme nous l'avons vu plus haut, la disparition se produit au moment où trois arcs de parabole s'intersectent en un point anguleux. Ce point anguleux est alors le centre d'un cercle de Delaunay tangent à la droite de balayage à cet instant, et c'est un sommet du diagramme de Voronoï. C'est ainsi que sont créés les sommets du diagramme de Voronoï. Un événement de type cercle est repéré par la plus grande valeur t de la droite de balayage Δ_t tangente au cercle, puisque c'est à cet instant qu'un arc de parabole disparaît du front parabolique.

Au début de l'exploration, la file $\mathcal{E}$ est initialisée avec les n événements sites. Au fur et à mesure du balayage, des événements du type cercle sont créés et insérés dans la file. Plus précisément, des événements de type cercle sont créés (voir aussi plus bas) chaque fois que trois arcs de parabole deviennent consécutifs. Lors de la création de tels événements, on n'est pas certain qu'ils donneront naissance à un sommet du diagramme de Voronoï, car les sites à droite de la ligne de balayage ne sont pas encore connus à cet instant. On les insère quand même dans la file, quitte à les supprimer ultérieurement.

Voici donc une description de l'algorithme de Fortune, qui sera détaillée dans la suite :

```
procédure FORTUNE;  (description sommaire)
  Insérer les événements sites dans la file E;
  tantque la file E n'est pas vide, faire
    extraire un événement p;
    si l'événement est un site, alors
      créer un nouvel arc parabolique, et une arête de Voronoï;
      désactiver les événements cercle obsolètes;
      insérer les événements cercle;
    si l'événement est un cercle, alors
      supprimer l'arc parabolique, créer un sommet de Voronoï;
      désactiver les événements cercle obsolètes;
      insérer les événements cercle;
  fintantque.
```

Arbre du front parabolique

Les arcs de parabole constituant le front parabolique sont naturellement ordonnés par valeur croissante de leurs ordonnées. On les range aux feuilles d'un arbre

binaire de recherche équilibré $\mathcal{F}$, qui contient quelques informations complémentaires. Un nœud x de l'arbre contient un couple (a, b) de sites, à savoir les sites des arcs de parabole extrêmes de l'intervalle de paraboles aux feuilles du sous-arbre de racine x. Ces informations permettent de piloter la recherche de l'arc de parabole intersectant une droite horizontale d'ordonnée donnée. En fait, l'arbre $\mathcal{F}$ est enrichi comme suit :

- chaque feuille (représentant un arc du front) contient un pointeur vers le site associé ;
- les feuilles sont chaînées entre elles, mais indirectement : entre deux feuilles consécutives est placé un pointeur (que nous appelons *chaînon*) vers une arête du diagramme de Voronoï en cours de construction.

En effet, comme nous l'avons montré plus haut, le point anguleux entre deux arcs de parabole consécutifs du front est sur une arête du diagramme de Voronoï et c'est un pointeur vers cette arête, dont on ne connaît d'ailleurs en général pas les extrémités, qui est le chaînon entre deux feuilles successives. Cela signifie aussi que de chaque feuille, on peut accéder en temps constant aux voisins. Nous avons vu que ceci est une extension facile des arbres de recherche balisés. Une arête elle-même contient un couple de pointeurs vers (le ou) les sommets du diagramme de Voronoï qui sont ses extrémités. Comme on le verra, la création d'une arête et l'attribution des extrémités à l'arête ne se font pas nécessairement au même instant.

Gestion du front

Les événements sont extraits dans l'ordre de la file des événements $\mathcal{E}$. Voici les opérations à réaliser sur l'arbre.

Si l'événement est un site a, un arc parabolique dégénéré α est créé. On descend dans l'arbre $\mathcal{F}$ à la recherche de l'arc parabolique, disons β, qui est intersecté par α (le cas où α intersecte le front en un point anguleux sera considéré plus loin). L'arc β est coupé en deux arcs β' et β'' entre lesquels vient se placer α. Dans l'arbre $\mathcal{F}$, on substitue donc, à la feuille β, trois feuilles β', α, β'', et on insère les deux nœuds nécessaires. Une nouvelle arête v du diagramme de Voronoï est créée, et les deux chaînons entre β' et α d'une part, entre α et β'' d'autre part, pointent vers cette arête (on ne connaît pas encore les sommets du diagramme de Voronoï qui seront les extrémités de l'arête).

Si l'événement est du type cercle, cela signifie d'abord la suppression d'un arc de parabole β réduit à un point s qui est un sommet du diagramme de Voronoï. La suppression de la feuille β dans l'arbre $\mathcal{F}$ (avec le rééquilibrage éventuellement nécessaire) est donc accompagnée de la création d'un sommet du diagramme de Voronoï. De plus, les arêtes sur lesquelles pointent les deux chaînons voisins de β reçoivent leur extrémité correspondante : c'est le point s. La suppression de β et de ses deux chaînons est suivie de l'insertion d'un chaînon entre ses feuilles voisines. Ce chaînon pointe vers une nouvelle arête, disons w, dont une extrémité est le sommet s.

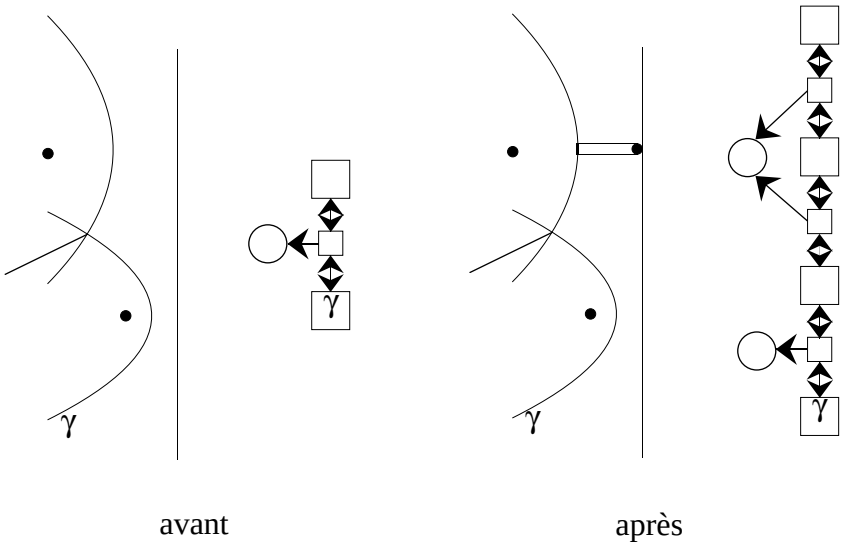


Figure 4.10: *Un événement de type site : cas «général».*

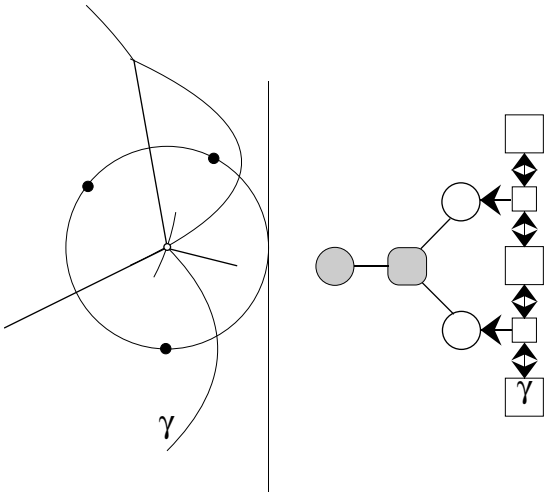


Figure 4.11: *Disparition de β et création du sommet s.*

Toutes ces opérations (sauf la recherche et le rééquilibrage) se font bien sûr en temps constant.

Reste le cas d'un événement de type site «spécial» : c'est le cas où l'arc parabolique dégénéré α créé par l'arrivée du site a intersecte le front parabolique en un point anguleux (voir figure 4.12). Dans ce cas, les deux opérations ci-dessus se succèdent : l'arc α est inséré dans l'arbre $\mathcal{F}$ entre les deux arcs qu'il intersecte, et le sommet s du diagramme de Voronoï est créé immédiatement.

Gestion des événements

Comme nous l'avons dit, les événements de type site sont créés au début des

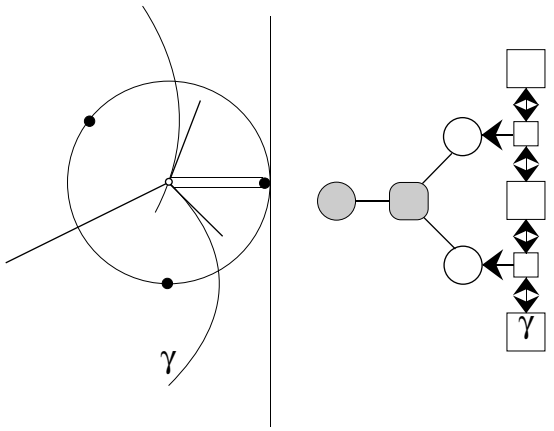


Figure 4.12: *Evénement site avec création d'un sommet.*

calculs. Les événements de type cercle signalent un sommet du diagramme de Voronoï, ou de manière équivalente, un cercle de Delaunay contenant 3 sites au moins. Un tel événement est créé chaque fois que l'on rencontre, pour la première fois, trois arcs consécutifs du front parabolique. Dans ce cas, le cercle circonscrit à leurs sites est calculé. Cet événement est repéré par l'abscisse maximale d'un point sur ce cercle. La gestion de ces événements est plus délicate : chaque fois qu'un nouveau site est introduit, il peut apparaître à l'intérieur de cercles calculés précédemment, et dont on avait supputé, au moment de leur création, qu'ils étaient des cercles de Delaunay. Pour tenir compte de cette situation, il faudra détruire, ou désactiver, des événement de type cercle introduits dans la file (on suppose qu'un événement de type cercle possède un champ qui signale s'il est actif ou non). Un événement de type cercle est déterminé par trois arcs consécutifs α, β, γ du front parabolique. On dira qu'il est *associé* à β; l'événement est actif tant qu'il continue à être associé à un arc parabolique.

La création d'événements de type cercle se fait lors du traitement des événements de $\mathcal{E}$. Considérons les deux cas.

Un événement de type site provient de la rencontre d'un site a; considérons d'abord le cas «général» : on crée un arc parabolique (dégénéré) α qui coupe en deux arcs β', β'' un arc β (voir ci-dessus). L'événement de type cercle associé à β, s'il existe, est détruit (désactivé), et deux nouveaux événements de type cercle sont crées, s'il y a assez d'arcs, l'un associé à β', et l'autre associé à β''. Notons que comme a est sur les cercles correspondants, ces événements se situent à droite de la ligne de balayage. Dans le cas d'un événement de type site «spécial», on supprime les événements de type cercle associés aux voisins β et γ de α, et on crée de nouveaux événements de type cercle pour β et γ (avec α comme voisin!).

Si l'événement est de type cercle, il y a disparition d'un arc β du front parabolique (précisément celui auquel est associé l'événement en cours de traitement!). Soient

α et γ les arcs qui précèdent et suivent β (à supposer qu'ils existent). Les éventuels événements de type cercle associés à ces arcs sont désactivés, et d'autres sont calculés : soit δ l'arc qui suit γ dans le front (s'il existe). Les trois arcs α, γ, δ sont maintenant consécutifs; soit C le cercle circonscrit à leurs sites. Si l'abscisse maximale d'un point de C est supérieure à la valeur t de balayage, l'événement ainsi créé est associé à γ et inséré dans la file; sinon, il est ignoré.

Expliquons pourquoi on peut ignorer l'événement si l'abscisse maximale d'un point de C est inférieure à t : dans ce cas, si C *est* un cercle de Delaunay, son centre est un sommet du diagramme de Voronoï, et il a donc déjà été traité antérieurement (en d'autres termes, un événement de type cercle déterminé par α, γ, et δ a déjà été créé et traité antérieurement).

Analyse

Il nous reste à analyser le temps pris par l'algorithme :

Théorème 4.7. *L'algorithme de Fortune calcule le diagramme de Voronoï de n sites en temps $O(n \log n)$ et en place $O(n)$.*

Preuve. Chaque événement de type site crée un arc parabolique dégénéré, et sauf dans un cas spécial, coupe en deux un arc parabolique. Un tel événement crée donc trois arcs, et en supprime un (celui qui est coupé en deux). Un événement de type cercle ne crée pas d'arc. Le nombre *total* d'arcs de parabole présents durant le balayage est $O(n)$. Le traitement d'un événement cercle crée un sommet du diagramme de Voronoï, et comme le diagramme est planaire, le nombre d'événements cercles traités est en $O(n)$. Mais tous les événements créés ne sont pas traités, puisque certains sont supprimés auparavant. Toutefois, le traitement d'un événement site ou cercle ne crée qu'un nombre constant d'événements, donc le nombre d'événements de type cercle est aussi en $O(n)$.

Ainsi, le nombre total d'événements créés est en $O(n)$. La place nécessaire pour gérer ces événements, et pour gérer l'arbre des arcs de parabole, est donc $O(n)$. Chaque événement requiert $O(\log n)$ opérations, pour l'insertion dans la file, et chaque gestion d'arc de parabole demande également un temps $O(\log n)$. D'où le temps total annoncé. ∎

11.4.3 Diagramme de Voronoï de segments

Dans cette section, on considère des diagrammes de Voronoï de segments. Il existe plusieurs définitions; nous choisissons de présenter celle qui est la mieux adaptée au problème de planification de trajectoires dont il sera question dans le chapitre suivant. Etant donné un segment $s = [a, b]$ du plan, on appellera, par abus de langage, *segment ouvert* le segment s privé de ses extrémités a et b.

Soit S un ensemble formé d'un nombre fini de *segments ouverts* deux à deux disjoints, et d'un ensemble fini de *points*. Les points qui sont les extrémités des

segments sont toujours supposés appartenir à S. Pour tout segment ouvert s, on notera $\bar{s}$ le segment formé de s et de ses extrémités. On appellera indistinctement *primitive* un point ou un segment ouvert de S. L'ensemble de la figure 4.13 ci-dessous est formé de 10 primitives, les 6 points a, b c, d, e, f et les 4 segment ouverts u, v, w, t.

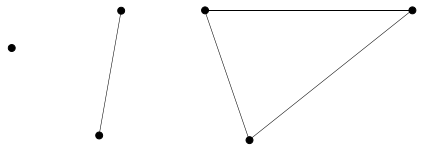


Figure 4.13: *Un ensemble de 4 segments ouverts et 6 points.*

Nous définissons le diagramme de Voronoï de S en faisant l'usage de l'amplitude d'un point et des disques de Delaunay. Soit p un point du plan. L'*amplitude* de p (relativement à S) est le nombre

$$\delta(p) = \min\{d(p,s) \mid s \in S\}.$$

Lorsque s est un segment, la distance de p à s est définie par

$$d(p,s) = \inf\{d(p,q) \mid q \in s\}.$$

Le *disque de Delaunay* du point p du plan est le disque $D(p)$ de centre p et de rayon $\delta(p)$. C'est le plus grand disque centré sur p et dont l'intérieur est disjoint de S. La frontière du disque, qui est le *cercle de Delaunay* de p, rencontre la réunion des objets de S en au moins un point. L'ensemble $Proche(p)$ des primitives *proches* de p est défini comme suit :

- un point $a \in S$ appartient à $Proche(p)$ si $d(a,p) = \delta(p)$, donc si a est sur le cercle de Delaunay de p;
- un segment ouvert s appartient à $Proche(p)$ si $d(a,s) = \delta(p)$, et si de plus la projection orthogonale de p sur la droite contenant s appartient à $\bar{s}$.

Le *diagramme de Voronoï* de S est l'ensemble $\mathrm{Vor}(S)$ formé des points p tels que $Proche(p)$ contient au moins deux primitives. Un *sommet* est un point p tel que $Proche(p)$ contient au moins trois primitives. Une *arête* est un ensemble connexe maximal de points tels que $Proche$ contient exactement deux primitives. Une arête peut être infinie des deux côtés, ou d'un seul côté, ou finie; ses extrémités, lorsqu'elles existent, sont des sommets du diagramme.

La forme et la structure du diagramme de Voronoï de segments sont plus complexes que pour un ensemble de points. Dans le cas du diagramme de Voronoï d'un ensemble de points, les arêtes sont des segments de droites, des demi-droites ou des droites. Il n'en est plus ainsi pour le diagramme de Voronoï de segments : les arêtes peuvent être également des arcs de parabole. Plus précisément, soit s un segment ouvert, d'extrémités a et b. La *bande délimitée par* s, notée $B(s)$, est

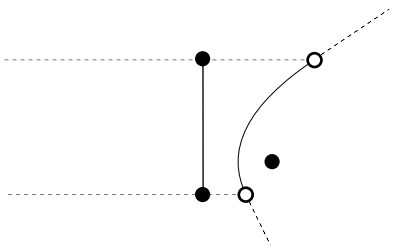


Figure 4.14: *Diagramme de Voronoï formé de 4 demi-droites et d'un arc de parabole.*

la région fermée du plan délimitée par les droites perpendiculaires à s et passant par a et par b respectivement. La bande de s divise le plan en trois régions : deux demi-plans ouverts $P(a)$ et $P(b)$, l'un ayant sur sa frontière a et l'autre b, et la bande $B(s)$ (voir figure 4.14). On a alors :

Lemme 4.8. *Soit p un point qui n'appartient pas au segment $\bar{s} = [a, b]$, et soit E l'ensemble des points équidistants de p et de $[a, b]$. L'intersection de E et de $P(a)$ est la partie de la médiatrice de $[p, a]$ contenue dans $P(a)$; l'intersection de E et de $B(s)$ est la partie de la parabole de foyer p et de directrice $d(a, b)$ contenue dans $B(s)$.* ∎

Ce lemme décrit donc entièrement le diagramme de Voronoï d'un ensemble formé d'un segment et d'un point. Ce diagramme est formé de quatre demi-droites (en pointillé sur la figure), et d'un arc de parabole.

La figure 4.15 donne le diagramme de Voronoï de deux segments. Il a six sommets, et onze arêtes. Nous avons étiqueté certaines arêtes par le couple de primitives qui forme l'ensemble *Proche*. Evidemment, deux arêtes concourantes ont une primitive commune. On peut voir que la partie «centrale» est formée de 7 arêtes.

La *région de Voronoï* d'une primitive r est l'ensemble $R(r)$ des points p tels que r appartient à $Proche(p)$. L'intérieur d'une région de Voronoï est constitué des points p du plan tels que $Proche(p)$ est réduit à une seule primitive. Par connexité, cette primitive est la même pour tous les points de la région. Ainsi, la région $R(s)$ du segment s sur la figure 4.15 est formée de la bande $B(s)$, à laquelle on a enlevé la portion située au-dessus des arêtes (s, d), (s, t), (s, c), (s, t).

Proposition 4.9. *Toute région de Voronoï $R(r)$ est étoilée par rapport à r, c'est-à-dire pour tout $p \in R(r)$, il existe $q \in r$ tel que $[p, q] \subset R(r)$.*

Nous allons prouver un résultat légèrement plus fort :

Proposition 4.10. *Soit p un point à l'intérieur d'une région de Voronoï $R(r)$, et soit $\bar{p}$ la projection de p sur r. Si $p \neq \bar{p}$, l'intersection de la droite D passant par p et $\bar{p}$ et de $R(r)$ est convexe.*

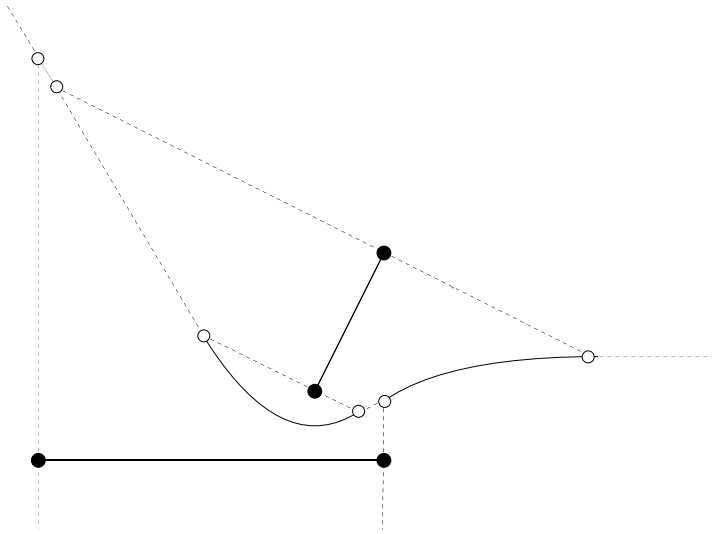


Figure 4.15: *Le diagramme défini par deux segments.*

Preuve. Comme p appartient à $R(r)$, l'intersection du cercle de Delaunay de centre p avec la réunion des primitives ne rencontre que r, et l'intersection est réduite au point $\bar{p}$. Soit q un point de $D \cap R(r)$, et t un point de $[\bar{p}, q]$. Le disque de Delaunay de t est contenu dans le disque de Delaunay de q, et est tangent à ce disque en $\bar{p}$, donc $t \in R(r)$, car l'intérieur du disque de Delaunay de q, et donc de t ne rencontre aucune primitive. Cela implique que l'intersection de D et de $R(r)$ est convexe. ∎

Bien entendu, une région de Voronoï n'est en général pas convexe, dans le cas de segments. La figure 4.16 donne un diagramme de Voronoï plus complexe. Lorsque les segments forment des polygones disjoints, on peut distinguer la partie *intérieure* et la partie *extérieure* du diagramme. Dans la figure 4.16, seule est tracée la partie du diagramme qui est intérieure au rectangle, et extérieure aux autres polygones. Cette configuration est celle qui est rencontrée dans le cadre de la planification de trajectoires.

Nous mentionnons sans preuve le résultat suivant :

Théorème 4.11. *Il existe un algorithme pour calculer le diagramme de Voronoï de n primitives en temps $O(n \log n)$.*

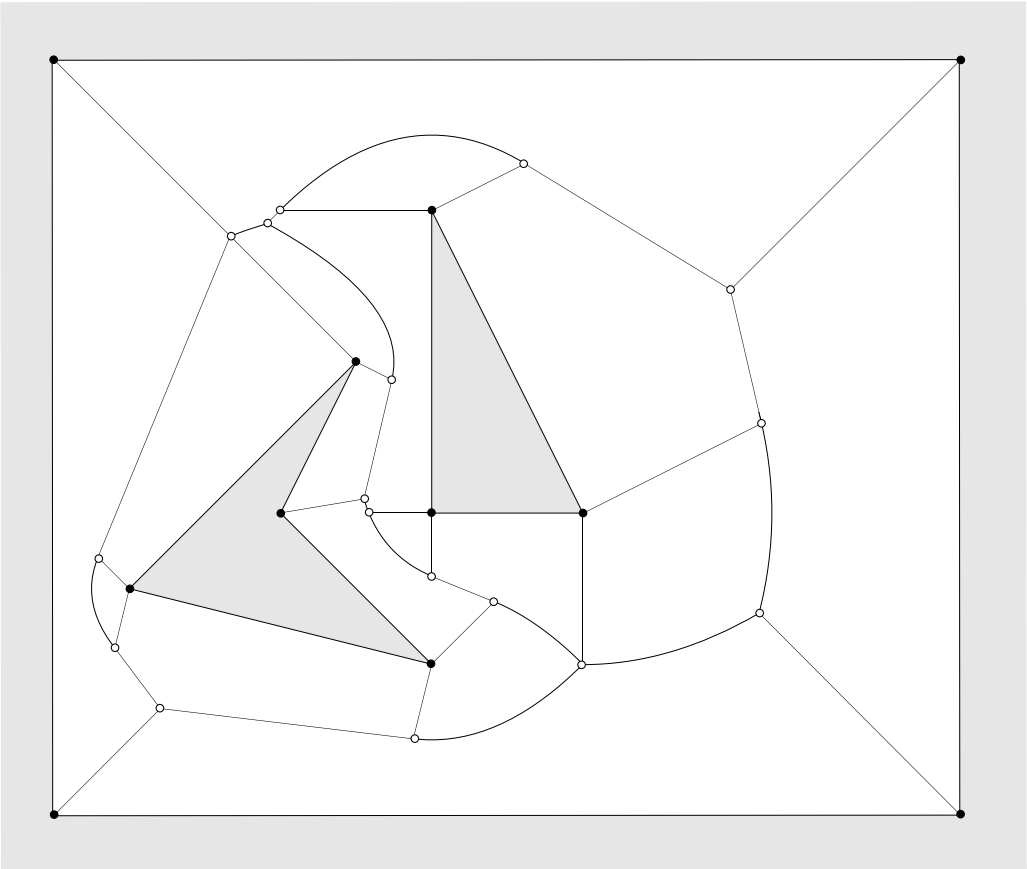


Figure 4.16: *Un diagramme de Voronoï.*

Notes

La littérature sur la géométrie algorithmique est relativement récente, pour les raisons évoquées dans la première section. Les algorithmes présentés dans ce chapitre sont pour la plupart exposés dans les ouvrages de F.P. Preparata et M.I. Shamos et de K. Melhorn :

F.P. Preparata, M.I. Shamos, *Computational Geometry*, Springer, Berlin, 1985.

K. Mehlhorn, *Data Structures and Algorithms : Vol.3 Multi-dimensional Searching and Computational Geometry*, Springer, Berlin, 1984.

Nous avons eu pour principe de donner une description détaillée de l'implémentation des algorithmes proposés, qui garantit la complexité annoncée et qui permet une programmation facile, avec la préoccupation implicite de minimiser les erreurs dues à la manipulation de nombres réels (nous n'avons pas du tout abordé les difficultés posées par la nature non discrète des problèmes géométriques).

Les algorithmes de calcul d'enveloppe convexe dans le plan sont nombreux; le premier algorithme en temps $O(n \log n)$ est dû à Graham et date de 1972.

L'algorithme de la localisation d'un point dans une subdivision planaire exposé est le plus performant à ce jour, en temps et en espace. Il est dû à N. Sarnak et R.E. Tarjan :

N. Sarnak, R.E. Tarjan, Planar point location using persistent search trees, *Comm. Assoc.Comput.Mach.* **29** (1986), 669–679.

L'étude des diagrammes de Voronoï est riche et variée. L'algorithme de calcul du diagramme de Voronoï d'un ensemble fini de points coplanaires exposé ici est celui de S.J. Fortune

S.J. Fortune, A sweepline algorithm for Voronoï diagrams, *Proc. 2nd Ann. ACM Symp. Comput. Geom.* (1986), 313–322.

D'autres algorithmes de même complexité existent, en particulier celui de D.G. Kirkpatrick qui procède par dichotomie. L'avantage de l'algorithme de Fortune est de pouvoir s'adapter au cas des segments ; nous ne l'avons pas présenté ici, car il est relativement sophistiqué.

Le chapitre 7 Computational Geometry de F.F. Yao du *Handbook Vol.A* contient des références complètes sur les sujets abordés dans ce chapitre.

Exercices

11.1. Soit D une droite, $O \in D$, et H un demi-plan ouvert de frontière D. Montrer que la restriction à H de la relation $\preceq_O$ est une relation d'ordre.

11.2. Soit O un point du plan. On insère dans un arbre binaire initialement vide, successivement n points, le critère de descente au sommet contenant p, lors de l'insertion du point q, étant :

si $q \preceq_O p$ alors descendre à gauche sinon descendre à droite.

a) Montrer que le parcours préfixe de l'arbre obtenu fournit un circuit polaire des n points relativement à O.

b) En est-il de même si l'on maintient une structure d'arbre AVL, i.e. si l'on procède à des rééquilibrages par rotations lors des insertions ?

c) Montrer qu'on peut définir une structure d'arbre AVL balisé, i.e. avec stockage des éléments aux feuilles, de façon à ce que l'insertion successive de n points dans un arbre initialement vide fournisse un circuit polaire des n points relativement à O ; on précisera quelles sont les balises aux nœuds et quel est le test de descente.

11.3. Dans l'algorithme de Graham, au lieu de choisir $O(0, y_0)$ à l'intérieur de l'enveloppe convexe, on choisit le « point » $O'(x_0, -\infty)$.

a) Que devient le circuit polaire des n points relativement à O' ?

b) Montrer qu'en appliquant l'algorithme de Graham avec O' au lieu de O, on obtient l'enveloppe convexe supérieure des n points.

c) En déduire un nouvel algorithme de calcul d'enveloppe convexe et donner sa complexité.

11.4. Montrer que dans une gestion dynamique d'enveloppe convexe, on peut réaliser la suppression d'un point en temps $O(\log^2 n)$, où n est le nombre de points considérés.

11.5. On dit que $P = (p_1, p_2, ..., p_n)$ est *une ligne polygonale cadrée dans la direction* δ si c'est une ligne polygonale simple et si de plus, il existe deux droites parallèles distinctes Δ et Δ' de direction δ passant respectivement par p_1 et p_n telles que tous les points p_i $(1 < i < n)$ soient situés à l'intérieur de la bande limitée par Δ et Δ' (figure 4.17(a)).

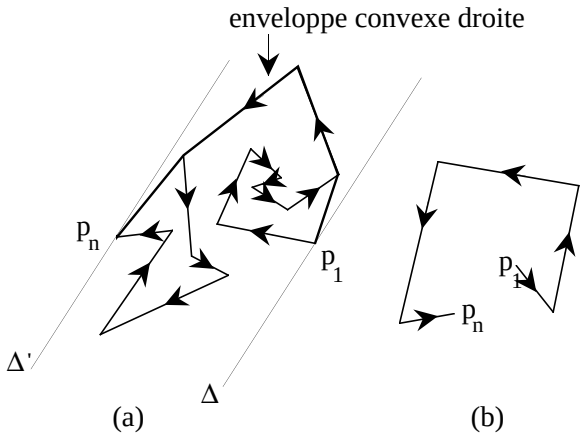


Figure 4.17: *Exemples.*

Il est clair qu'une ligne polygonale simple n'est pas nécessairement cadrée (figure 4.17(b)).

Si $P = (p_1, ..., p_n)$ est une ligne polygonale cadrée, alors p_1 et p_n sont des sommets de l'enveloppe convexe de P, et ils partitionnent le contour orienté de $EC(P)$ en deux lignes polygonales simples, l'une de p_1 vers p_n que nous appellerons *enveloppe convexe droite de* P et l'autre de p_n vers p_1 que nous appellerons *enveloppe convexe gauche*. Cette terminologie se justifie par le fait que l'enveloppe convexe gauche (resp. droite) de P ainsi définie est également le contour de l'enveloppe convexe de l'ensemble des sommets de P situés à gauche (resp. à droite), au sens large, du vecteur $\overrightarrow{p_1 p_n}$. Il suffit donc de calculer l'enveloppe convexe droite D de P, le calcul de l'enveloppe convexe gauche étant de même nature. Ainsi, D est constitué d'une sous-suite $(q_1, ..., q_R)$ de la suite P avec $q_1 = p_1$ et $q_R = p_n$. On peut décrire chaque segment $[q_i, q_{i+1}]$ comme étant la «fermeture» d'une «poche» $K_i = (p_{k_i}, ..., p_{k_{i+1}})$ de la ligne polygonale P (figure 4.18)

a) Déterminer un algorithme qui consiste à suivre la ligne polygonale $(p_1, .., p_n)$ et à construire successivement les fermetures des poches en temps $O(n)$.

b) En déduire un algorithme linéaire de calcul de l'enveloppe convexe d'un polygone simple.

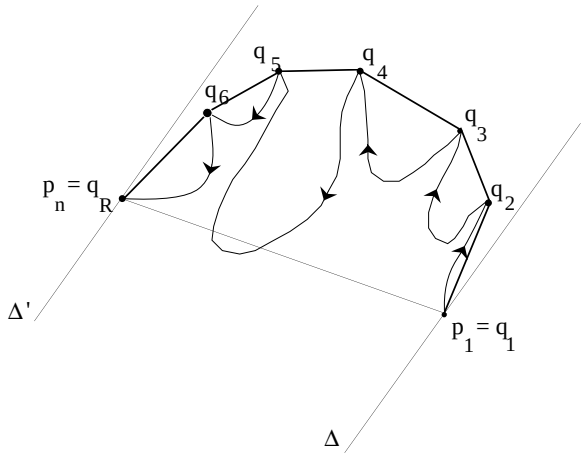


Figure 4.18: *Principe de l'algorithme.*

11.6. Soient P et Q deux polgones convexes ayant respectivement m et n sommets. Montrer qu'on peut calculer la distance de P à Q en temps $O(sup(m,n))$ dans le pire des cas.

11.7. Le problème *des bureaux de poste* :

Etant donnés n points distincts du plan (les bureaux de poste), déterminer pour un point arbitraire p le (ou un) bureau de poste le plus près de p.

Décrire un prétraitement des n points qui permet de résoudre le problème en temps $O(\log n)$.

11.8. Le problème du *plus grand cercle vide* : Etant donnés un ensemble S de n points distincts du plan, déterminer un cercle de plus grand rayon ne contenant aucun des n points à l'intérieur. Montrer que le centre d'un tel cercle est soit un sommet de $Vor(S)$ soit l'intersection d'une arête de $Vor(S)$ et d'un côté de $EC(S)$. En déduire un algorithme en temps $O(n \log n)$ pour résoudre ce problème (algorithme dû à G.T. Toussaint).

Chapitre 12

Planification de trajectoires

Dans ce chapitre, nous étudions un problème de planification de trajectoires appelé le problème du « déménageur de piano ». La première section explique la nature du problème et donne les notations et définitions nécessaires. Dans la deuxième section, un algorithme pour le calcul du mouvement de translation d'un segment dans un environnement polygonal est présenté; cet algorithme est dû à Schwartz et Sharir. Dans la dernière section, nous étudions le déplacement d'un disque dans un environnement polygonal et présentons un algorithme, dû à Ó'Dúnlaing et Yap, utilisant les diagrammes de Voronoï de segments.

Introduction

Le problème du « déménageur de piano » est un problème fondamental de la robotique. On peut le formuler de la façon suivante : soit un piano (ou un robot) et des obstacles; étant données deux positions permises du piano, existe-t-il un mouvement de ce piano entre ces deux positions sans collision avec les obstacles, et si oui, donner un tel mouvement. Les études faites sur ce sujet sont nombreuses et font appel à des outils mathématiques appartenant à des domaines variés tels que la géométrie classique, la topologie, la géométrie algébrique, l'algèbre et la combinatoire. Dans son aspect le plus général, le robot n'est pas un corps rigide, mais est constitué d'éléments possèdant un certain nombre de degrés de liberté. Des solutions théoriques existent mais qui ne sont pas efficaces. Une approche heuristique du problème en intelligence artificielle a été faite, et a donné lieu à des solutions qui sont implémentées et utilisées. Nous nous intéresserons ici à l'aspect algorithmique du problème, c'est-à-dire à la recherche de solutions exactes.

Nous avons limité notre choix à la présentation de deux problèmes de base classiques, qui font appel dans leur résolution à l'algorithmique géométrique et plus

généralement à divers outils algorithmiques développés dans cet ouvrage. Nous avons choisi deux approches différentes pour les deux problèmes, ce qui permet d'avoir un aperçu des différentes techniques utilisées pour résoudre ce genre de questions.

Notations et définitions

Le plan contient un certain nombre d'obstacles, et l'*espace libre du plan*, $\mathcal{E}$, est constitué des points extérieurs aux obstacles. L'objet $\mathcal{X}$ à déplacer est une partie de $\mathbb{R}^2$ de forme variable : polygone, disque, segment...

Etant donné un objet $\mathcal{X}$, une *position* de $\mathcal{X}$ est l'image de $\mathcal{X}$ dans un déplacement (appelé encore isométrie directe) du plan. Or tout déplacement p du plan peut se caractériser comme la composée d'une rotation d'origine O et d'angle de mesure $\theta \in [0, 2\pi[$ et d'une translation. Un repère orthonormé étant choisi, une position de l'objet $\mathcal{X}$, soit $p(\mathcal{X})$, où p est un déplacement, sera déterminée par un triplet $p = (x, y, \theta)$ de $\mathbb{R}^2 \times [0, 2\pi[$, où θ caractérise la rotation et (x, y) la translation. Il n'y a pas, en général, bijection entre l'ensemble des positions d'un objet et l'ensemble des déplacements du plan; plus précisément lorsque l'objet possède certaines propriétés de symétrie, plusieurs déplacements correspondent à la même position. Par exemple (figure 0.1), le segment $\mathcal{X}$ a la même image par les déplacements $(x, y, 0)$ et (x', y', π).

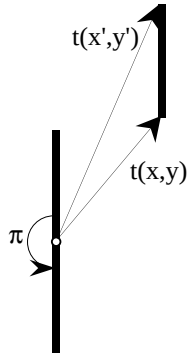


Figure 0.1: *La translation $t(x, y)$ définit la même position que la rotation d'angle plat suivie de $t(x', y')$.*

Il reste néanmoins que la donnée du déplacement détermine de manière unique la position de l'objet, ce qui justifie cette représentation.

Une position $p(\mathcal{X})$ est *libre* si $p(\mathcal{X}) \subset \mathcal{E}$, *semi-libre* si $p(\mathcal{X})$ «touche» un ou plusieurs obstacles i.e. n'a de points communs avec les obstacles que sur leur frontière (fig.0.2).

La figure 0.2 montre des positions différentes d'un objet qui est un segment, dont certaines sont libres, d'autres semi-libres, d'autres ni libres ni semi-libres. On

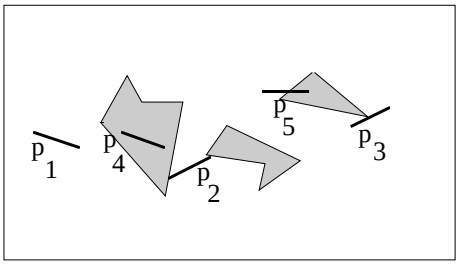


Figure 0.2: *p_1 est libre, p_2 et p_3 sont semi-libres, p_4 et p_5 ne le sont pas.*

notera $\mathcal{L}$ et $\mathcal{SL}$ l'ensemble des positions libres et semi-libres de $\mathcal{X}$ dans $\mathcal{E}$. Grâce à la représentation de la position d'un objet par un déplacement, on a :

$$\mathcal{L},\ \mathcal{SL} \subset\ \mathbb{R}^2 \times [0, 2\pi[$$

Notons que $\mathcal{L}$ et $\mathcal{SL}$ sont des ensembles dépendant à la fois de l'environnement $\mathcal{E}$ et de l'objet $\mathcal{X}$.

Etant données deux positions p_0 et p_1 de $\mathcal{X}$, on appelle *mouvement de $\mathcal{X}$ de la position p_0 vers la position p_1* une application continue $m :\ [0,1] \longrightarrow \mathbb{R}^2 \times [0, 2\pi[$ telle que $m(0) = p_0$ et $m(1) = p_1$. Soient p_0 et p_1 deux positions libres. Le mouvement est *libre* (resp. *semi-libre*) si pour tout $\alpha \in [0,1]$, la position $m(\alpha)$ est une position libre (resp. semi-libre). Le mouvement m est un *mouvement de translation* s'il existe $\theta \in [0, 2\pi[$ tel que pour tout $\alpha \in [0,1]$, $\ m(\alpha) \in \mathbb{R}^2 \times \{\theta\}$.

Dans les cas particuliers auxquels nous allons nous intéresser, ces définitions générales vont se simplifier.

12.1 Translation d'un segment

12.1.1 Présentation du problème

Nous supposons ici que l'objet $\mathcal{X}$ est un segment de longueur l, les obstacles des polygones simples, et que les mouvements autorisés sont des mouvements de translation.

L'ensemble des *obstacles* est constitué d'un nombre fini de polygones simples d'intérieur non vide, d'intersection finie deux à deux (c'est-à-dire pouvant se «toucher» mais non se chevaucher). Si les obstacles partagent des morceaux de frontière de longueur non nulle, on peut toujours redéfinir les obstacles (en les fusionnant) de manière à ce que les hypothèses que nous faisons soient vérifiées.

Pour rendre le traitement des données homogène, nous supposerons que le plan est limité à un rectangle R suffisamment grand pour qu'il ne modifie pas la nature du problème; pour cela il suffit que ses côtés soient par exemple à une distance strictement supérieure à l de tous les sommets des polygones constituant

les obstacles (l est la longueur du segment à déplacer). Ainsi, R est ajouté à l'ensemble des obstacles mais la partie «interdite» est la région extérieure à R. Dorénavant, *l'espace libre* $\mathcal{E}$ est l'ensemble des points intérieurs à R et extérieurs aux autres obstacles (figure 0.2).

Puisque l'objet est un segment et que nous limitons les déplacements à des translations, une position p de $\mathcal{X}$ est caractérisée plus simplement encore par un triplet $(x, y, \theta) \in \mathbb{R}^2 \times [0, \pi[$, où (x, y) représente le couple de coordonnées de l'extrémité P de p la plus basse, (et la plus à gauche si p est horizontal) Q étant l'autre extrémité, et θ est la mesure de l'angle $(\overrightarrow{PQ}, \vec{i})$ (figure 1.1); par définition de P, il est clair que θ est strictement inférieur à π .

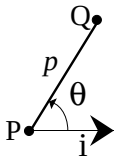


Figure 1.1: *P et θ caractérisent p.*

On note alors $\mathcal{L}$, $\mathcal{SL}$, $\mathcal{L}_\theta$, $\mathcal{SL}_\theta$ respectivement l'ensemble des positions libres, semi-libres, libres d'orientation θ, semi-libres d'orientation θ de $\mathcal{X}$ dans $\mathcal{E}$. On a bien sûr :

$$\mathcal{L},\ \mathcal{SL} \subset\ \mathbb{R}^2 \times [0, \pi[\quad \mathcal{L}_\theta, \mathcal{SL}_\theta \subset \mathbb{R}^2 \times \{\theta\}$$

On identifiera donc $\mathcal{L}_\theta$ et $\mathcal{SL}_\theta$ à des parties de $\mathbb{R}^2$ et même plus précisément à des parties de $\mathcal{E}$. Dans ce cadre restreint, un mouvement de la position p_0 à la position p_1 d'un segment $\mathcal{X}$ devient une application continue $m :\ [0, 1] \longrightarrow \mathbb{R}^2 \times \{\theta\}$ telle que $m(0) = p_0$ et $m(1) = p_1$. Le problème que nous allons résoudre se formule comme suit :

Données : L'ensemble $\mathcal{S}$ des segments constituant les côtés des obstacles, avec pour chaque segment la donnée de son successeur et de son prédécesseur dans le contour direct de l'obstacle auquel il appartient, et la longueur l du segment $\mathcal{X}$.

Questions : Pour une valeur donnée $\theta \in [0, \pi[$:

(1) Déterminer si une position p d'orientation θ est libre, c'est-à-dire appartient à l'ensemble $\mathcal{L}_\theta$

(2) Etant données deux positions libres p_0 et p_1, déterminer s'il existe un mouvement m de $\mathcal{X}$ de la position p_0 à la position p_1, et si oui en calculer un.

12.1.2 Présentation de l'algorithme

Nous supposons pour l'instant et pour simplifier l'exposé de l'algorithme, que l'ensemble des sommets ne contient pas trois points distincts alignés verticalement

et que les obstacles sont disjoints, et verrons ensuite comment l'algorithme que nous fournissons s'étend au cas général.

Nous prenons pour l'instant $\theta = \pi/2$, nous verrons ensuite comment en déduire le résultat pour une valeur de θ quelconque.

L'algorithme repose sur un principe de balayage du plan par une droite verticale. Donnons-en tout d'abord les grandes lignes :

Soit S l'ensemble des sommets des obstacles. On fait passer par chaque sommet une droite verticale en ne conservant que les morceaux de droite contenus dans l'espace libre $\mathcal{E}$. On décompose ainsi l'espace libre $\mathcal{E}$ en cellules trapézoïdales (éventuellement dégénérées en triangles) dont les côtés parallèles sont verticaux. Dans l'exemple de la figure 1.2, il y a trois obstacles (grisés) et le rectangle clôturant la scène. L'espace libre $\mathcal{E}$ est formé des points intérieurs au rectangle et extérieurs aux obstacles. Les polygones sont délimités par seize segments étiquetés de a à o. La décomposition indiquée fournit quinze cellules trapézoïdales numérotées de 1 à 15.

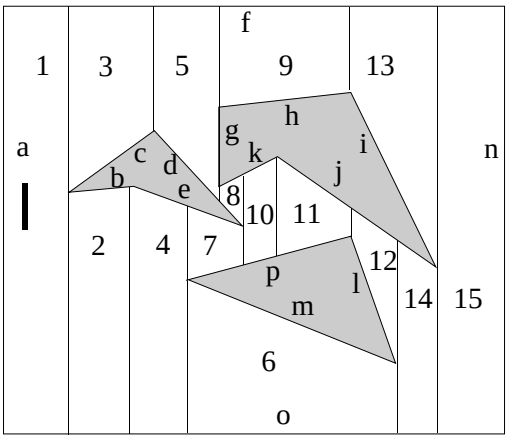


Figure 1.2: *Partitionnement de l'espace libre en cellules.*

Deux cellules sont *voisines* si elles partagent un segment de frontière de longueur non nulle. Nous définissons ainsi un graphe $G_{\pi/2}$ appelé *graphe des cellules* dont les sommets sont les cellules que l'on vient de définir, deux cellules étant reliées par une arête si elles sont voisines. La figure 1.3 donne le graphe $G_{\pi/2}$ associé aux données de la figure 1.2.

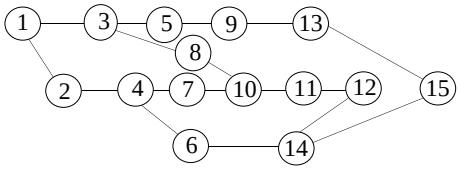


Figure 1.3: *Le graphe $G_{\pi/2}$ associé à l'exemple de la figure précédente.*

Le graphe $G_{\pi/2}$ ainsi calculé est indépendant de la longueur l du segment $\mathcal{X}$. Dans une deuxième étape, on émonde chaque cellule C en retranchant une «bande» de hauteur l à sa partie supérieure, on obtient ainsi une nouvelle collection de cellules C'. La figure 1.4 montre que la cellule 8 de la figure 1.2 a disparu, les cellules 11 et 12 ne sont plus voisines, les cellules 7, 11, 12 sont devenues des triangles.

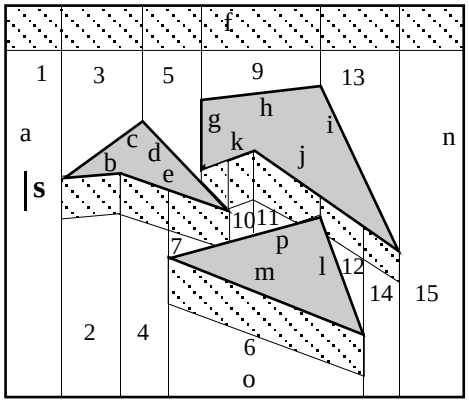


Figure 1.4: *Emondage des cellules.*

Le graphe obtenu en mettant à jour les relations de voisinage du graphe $G_{\pi/2}$ est noté $G_{\pi/2}(l)$ (figure 1.5).

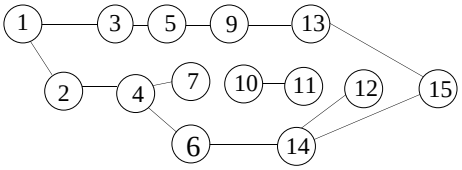


Figure 1.5: *Le graphe $G_{\pi/2}(l)$ obtenu après émondage.*

Nous démontrerons qu'il existe un mouvement libre d'une position p_0 à une position p_1 si et seulement si les cellules contenant respectivement p_0 et p_1 sont dans la même composante connexe du graphe $G_{\pi/2}(l)$. Ainsi, un problème topologique se trouve ramené à un problème combinatoire simple.

Nous donnons maintenant un certain nombre de résultats qui justifieront la validité de l'algorithme utilisé. Ensuite, nous exposerons l'algorithme détaillé ainsi que l'évaluation de sa complexité.

Nous commençons par donner la définition précise de la décomposition en cellules, et du graphe $G_{\pi/2}(l)$, et prouvons que l'existence d'un mouvement de translation se ramène à l'existence d'une chaîne dans ce graphe.

Soit $p \in \mathcal{L}_{\pi/2}$ une position libre de $\mathcal{X}$, (rappelons que la position p est représentée par un point qui est son extrémité inférieure) et soit Δ la droite verticale passant par p. En faisant glisser le segment $\mathcal{X}$ le long de Δ à partir de p tout en restant dans $\mathcal{L}_{\pi/2}$, $\mathcal{X}$ atteint deux positions extrémales p_{inf} et p_{sup} qui sont des positions

semi-libres. Plus précisément, la composante connexe du **point** p dans $\Delta \cap \mathcal{L}$ est un intervalle ouvert $]p_{inf}, p_{sup}[$ tel que chaque position p_{inf} et p_{sup} «touche» un ou deux segments de $\mathcal{S}$.

Si le point p_{sup} (resp. p_{inf}) appartient à un seul segment de $\mathcal{S}$, on note ce segment *Sup*(p) (resp. *Inf*(p)). Sinon, p_{sup} (resp. p_{inf}) est un sommet commun à deux segments de $\mathcal{S}$. Si ces deux segments sont d'un même côté (au sens large) par rapport à Δ, on note *Sup*(p) (resp. *Inf*(p)) celui qui est le plus bas (resp. le plus haut); dans le cas contraire, on note *Sup*(p) (resp. *Inf*(p)) celui qui est à gauche de Δ. La figure 1.6 indique comment est déterminé *Sup*(p) en cas d'ambiguïté, selon les cas de figure (l'intérieur des obstacles est en gris).

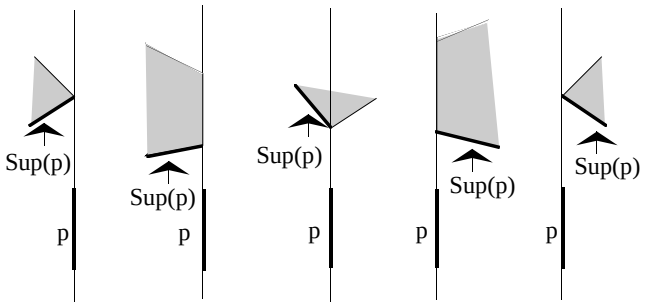


Figure 1.6: *Choix du segment Sup(p) selon la position de p.*

Ainsi à chaque position libre p est associé un couple (*Inf*(p), *Sup*(p)) de deux segments de $\mathcal{S}$.

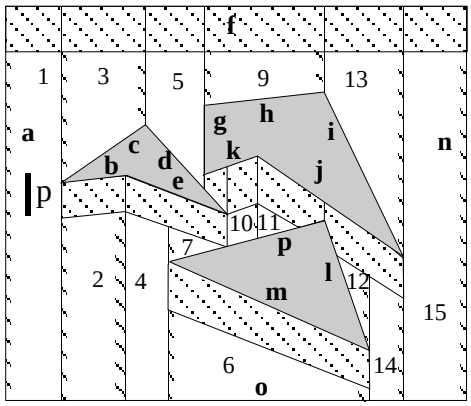
Deux positions libres p et p' sont *équivalentes* si *Inf*(p) =*Inf*(p') et *Sup*(p) =*Sup* (p'). Ainsi $\mathcal{L}_{\pi/2}$ (considéré comme une partie de $\mathbb{R}^2$) est partitionné en *cellules* qui sont *les composantes connexes* des classes de la relation d'équivalence que nous venons de définir. La figure 1.7 donne cette partition sur l'exemple utilisé avec, pour chaque cellule, le couple de segments (*Sup,Inf*) associé. Les hachures précisent à quelle cellule la frontière appartient, la convention étant que la frontière appartient à la cellule du côté hachuré.

Lemme 1.1. *Soit $\vec{u}$ le vecteur $(0, l)$. Une cellule C est un trapèze $(d_-(C), d_+(C), g_+(C), g_-(C))$ (éventuellement dégénéré en triangle), dont les côtés parallèles $[d_-(C), d_+(C)]$ à droite et $[g_-(C), g_+(C)]$ à gauche sont verticaux, et vérifient :*

- *$(d_-(C)$ ou $t_{\vec{u}}(d_+(C)) \in S)$ et $(g_-(C)$ ou $t_{\vec{u}}(g_+(C)) \in S)$*
- *$[g_-(C), d_-(C)]$ est une partie d'un segment de $\mathcal{S}$, de même pour $t_{\vec{u}}([g_+(C), d_+(C)])$ $(g, d, -, +$ sont mis respectivement pour gauche, droit, inférieur, supérieur).*

Preuve. Cela découle immédiatement de la définition d'une cellule . ∎

Notons que les cellules de $\mathcal{L}_{\pi/2}$ ne sont en général ni ouvertes ni fermées. En effet ce sont exactement les cellules C' décrites plus haut et qui constituent les sommets



1:(o, f) 9:(h, f)
2:(o, b) 10:(p, k)
3:(c, f) 11:(p, j)
4:(o, e) 12:(l, j)
5:(d, f) 13:(i, f)
6:(o, m) 14:(o, j)
7:(p, e) 15:(o, f)

Figure 1.7: *Les cellules de* $\mathcal{L}_{\pi/2}$.

du graphe $G_{\pi/2}(l)$. Par définition, les côtés non verticaux n'appartiennent pas aux cellules car ils correspondent à des positions semi-libres, et en ce qui concerne les côtés verticaux cela varie selon que les points $d_-(C), d_+(C), g_+(C), g_-(C)$ appartiennent ou non à S, et la réponse se déduit simplement selon les cas (la figure 1.8 donne un exemple).

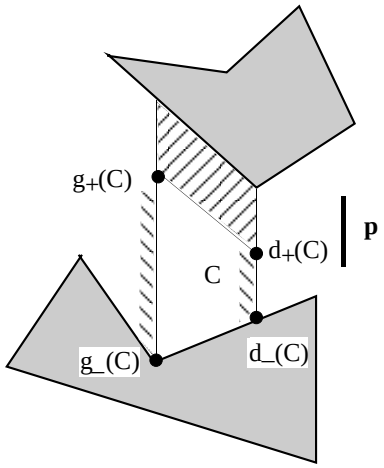


Figure 1.8: *Une cellule.*

Deux cellules sont dites *voisines* si leurs frontières partagent un segment de longueur non nulle. Notons que ce segment est nécessairement vertical.

Lemme 1.2. *Si deux cellules sont voisines, il existe un intervalle ouvert (vertical) appartenant à leur frontière commune et à* $\mathcal{L}_{\pi/2}$.

Preuve. Cela découle du fait que les côtés verticaux (extrémités exclues) d'une cellule sont inclus dans $\mathcal{L}_{\pi/2}$. ∎

Le graphe $G_{\pi/2}(l)$ est donc défini ainsi : ses sommets sont les cellules précédemment définies, et deux sommets sont adjacents si et seulement si les cellules correspondantes sont voisines. Le calcul direct de ce graphe est difficile, nous verrons qu'il est plus simple de calculer d'abord le graphe $G_{\pi/2}$, puis d'en déduire $G_{\pi/2}(l)$.

La proposition suivante traduit la condition topologique d'existence d'un mouvement de translation en une propriété combinatoire du graphe $G_{\pi/2}(l)$.

Proposition 1.3. *Soient p et p' deux positions libres et $C(p), C(p')$ les cellules auxquelles elles appartiennent respectivement. Il existe un mouvement de translation de p vers p' si et seulement si $C(p)$ et $C(p')$ appartiennent à la même composante connexe du graphe $G_{\pi/2}(l)$.*

Preuve.

La condition est suffisante.

Soient p et p' deux positions libres, et $C(p)$ et $C(p')$ leurs cellules appartenant à la même composante connexe de $G_{\pi/2}(l)$. On examine successivement trois cas.

- Supposons d'abord que $C(p) = C(p')$.

Posons $p = (x, y)$ et $p' = (x', y')$. Le segment $[p, p']$ est contenu dans $C(p)$. Alors l'application $[0, 1] \to \mathbb{R}^2$ qui à tout réel α de $[0, 1]$ fait correspondre $(x + \alpha(x' - x), y + \alpha(y' - y))$ est un mouvement libre de translation de $\mathcal{X}$ de la position p vers la position p'. En effet cela provient du fait qu'un trapèze est convexe et donc que le segment $[p, p']$ est tout entier contenu dans la cellule $C(p)$ (ici, p et p' sont considérés en tant que points).

On montre de même que si p' appartient à la frontière de $C(p)$ ou si p appartient à la frontière de $C(p')$ alors la même formule définit un mouvement libre de translation de p vers p'.

- Supposons que $C(p)$ et $C(p')$ sont des cellules voisines.

Utilisant le lemme 1.2, soit t une position libre appartenant à la frontière commune de $C(p)$ et $C(p')$. Alors, grâce à l'étude des cas précédents, il existe un mouvement libre de p vers t et un mouvement libre de t vers p'. Il suffit de composer les deux mouvements pour obtenir le résultat.

- Il reste à examiner le cas général. Puisque les sommets de $G_{\pi/2}(l)$ correspondant à $C(p)$ et $C(p')$ appartiennent à la même composante connexe de $G_{\pi/2}(l)$, il existe dans $G_{\pi/2}(l)$ une chaîne $(C_0 = C(p), C_1, \ldots, C_k = C(p'))$. Soit t_i une position libre appartenant à la frontière commune de C_i et C_{i+1} pour $i = 0, \ldots, k-1$. Il existe des mouvements libres de p à t_0, de t_i à t_{i+1} pour $i = 0, \ldots, k-1$ et de t_{k-1} à p', et il suffit de composer ces mouvements dans l'ordre pour obtenir un mouvement libre de translation de p vers p'.

Cela termine la preuve dans un sens.

Réciproquement, supposons qu'il existe un mouvement libre m de $\mathcal{X}$ de la position p vers p'. Soit Γ la «trajectoire» du point le plus bas dans le mouvement m, cette trajectoire a pour extrémités p et p'. La trace des cellules sur Γ décompose Γ en courbes adjacentes dont la suite des extrémités est notée $(p_0 = p, p_1, \ldots, p_k = p')$, de telle sorte que dans le mouvement m, lorsque l'objet passe par la position p_i, sa position change de cellule et passe à une cellule voisine. Ainsi la suite des cellules par lesquelles passe la position de l'objet dans le mouvement m forme une chaîne de $G_{\pi/2}(l)$, et p et p' sont dans la même composante connexe. ∎

Nous sommes maintenant en mesure de décrire l'algorithme de prétraitement, i.e. la construction des graphes $G_{\pi/2}$ et $G_{\pi/2}(l)$ et de résoudre le problème posé.

12.1.3 Prétraitement des données

Construction du graphe des cellules $G_{\pi/2}$

L'algorithme utilisé est un algorithme de balayage. On appelle *événement d'abscisse* α, que l'on note $e(\alpha)$, l'ensemble des segments de $\mathcal{S}$ ayant une extrémité au moins d'abscisse α. Grâce aux hypothèses faites (voir 12.1.1), chaque événement contient au moins 2 et au plus 4 segments. Les événements sont *simples* (2 ou 3 segments) (fig.1.9 (a)) ou *doubles* (4 segments) (figure 1.9 (b)).

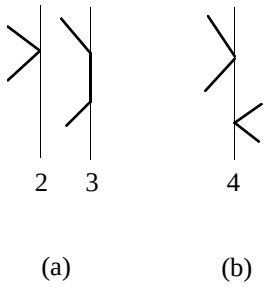


Figure 1.9: *Evénements (a) simples ou (b) double.*

Un événement simple $e(\alpha)$ est *un début, un milieu ou une fin* selon que les segments de $e(\alpha)$ sont à droite, de part et d'autre, ou à gauche de la droite d'équation $x = \alpha$ (figure 1.10).

Les événements sont triés par ordre croissant d'abscisse. Le plan est balayé de gauche à droite par une droite verticale Δ. Dans une position donnée de Δ, on peut classer les segments de $\mathcal{S}$ en trois classes : un segment est *traité* s'il est situé à gauche au sens large de Δ, et s'il a au moins un point strictement à gauche de Δ, *non traité* s'il est à droite strictement de Δ, *actif* sinon (figure 1.11).

Les notions analogues sont définies pour les cellules. Un événement est traité si son abscisse est inférieure ou égale à celle de Δ. Au cours du balayage, trois structures de données sont maintenues :

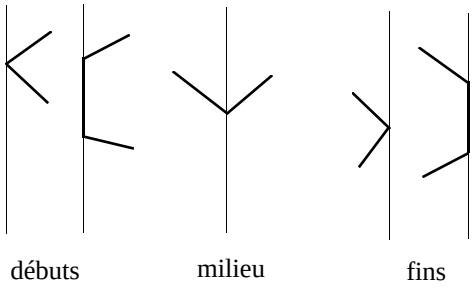


Figure 1.10: *Débuts, milieu et fins.*

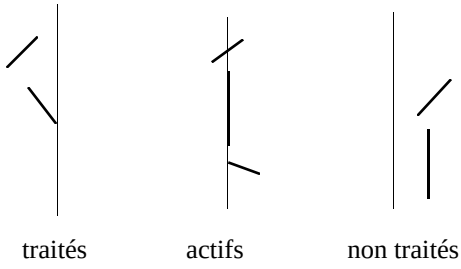


Figure 1.11: *Segments traités, actifs, non traités.*

- la liste L des événements non traités;
- la liste $\mathcal{V}$ des segments actifs triée par ordre croissant de l'ordonnée du point d'intersection avec Δ;
- le graphe partiel de $G_{\pi/2}$ des cellules traitées ou actives.

Etant donnés deux segments consécutifs dans $\mathcal{V}$ (sans prendre en compte les segments verticaux), la région située entre ces deux segments au voisinage droit de Δ est soit intérieure à un obstacle, soit intérieure à une cellule C située entre ces deux segments. Dans ce dernier cas, les deux sommets gauches de C, $g_-(C)$ et $g_+(C)$ ont déjà été calculés lors d'une position antérieure de Δ, ainsi que les cellules voisines de C situées à gauche de C. Ces informations sont ajoutées respectivement dans la structure $\mathcal{V}$ et dans $G_{\pi/2}$. Nous préciserons plus tard comment réaliser l'implémentation de ces structures. Lorsque la droite Δ passe par la position verticale correspondant au côté droit de la cellule c, ses deux extrémités droites $d_-(c), d_+(c)$ sont calculées, ainsi que les cellules voisines situées à droite de c, ainsi la cellule c est traitée.

L'algorithme ci-dessous calcule le graphe $G_{\pi/2}$.

procédure GRAPHE-DES-CELLULES($G_{\pi/2}$);
(1) trier les événements par abscisses croissantes : $L = (e_1, \ldots, e_p)$;
(2) initialiser $\mathcal{V}$ et $G_{\pi/2}$;
(3) pour i de 1 à p faire
(4) supprimer e_i dans L et mettre à jour $\mathcal{V}$ et $G_{\pi/2}$
finpour.

Précisons ce que sont l'initialisation et la mise à jour en fonction du type de l'événement considéré :

Initialisation de $\mathcal{V}$: Soient p_0 et p_1 les segments horizontaux respectivement inférieur et supérieur du rectangle R et p_0 et p_1 leur extrémité gauche. Alors G est initialisé avec un seul sommet C_0 tel que $g_-(C_0) = p_0$ et $g_+(C_0) = p_1$, et $\mathcal{V} = (p_0, p_1)$.

Mise à jour de $\mathcal{V}$ et $G_{\pi/2}$:

Nous étudions seulement les événements simples, le cas des événements doubles s'en déduisant simplement.

• L'événement e_i est une *fin* , il y a quatre types de fins notés f_1, f_2, f_3, f_4 (figure 1.12).

$_*$ de type f_1 : On a $e_i = \{s, s'\}$. Soit s_- le prédécesseur de s et s'_+ le successeur de s' dans L. Soient C_i la cellule située entre s_- et s, et C_j celle située entre s' et s'_+. On achève le traitement de C_i en posant $d_-(C_i) = \Delta \cap s_-,\ d_+(C_i) = P$, et celui de C_j en posant $d_-(C_j) = P,\ d_+(C_j) = \Delta \cap s'_+$. On ajoute au graphe $G_{\pi/2}$ une nouvelle cellule C_k adjacente à C_i et C_j et on calcule $g_-(C_k) = d_-(C_i),\ d_+(C_k) = d_+(C_j)$. Enfin, on supprime s et s' dans L.

$_*$ de type f_2 : On a $e_i = \{s, s'\}$. Le traitement de la cellule C_i est achevé en posant $d_-(C_i) = d_+(C_i) = P$. On supprime s et s' dans L.

$_*$ de type f_3 : On a $e_i = \{s, t, s'\}$. Soit s_- le prédécesseur de s et s'_+ le successeur de s' dans L. Soient C_i la cellule située entre s_- et s, et C_j celle située entre s' et s'_+. On achève le traitement de C_i en posant $d_-(C_i) = \Delta \cap s_-,\ d_+(C_i) = P$, et celui de C_j en posant $d_-(C_j) = Q,\ d_+(C_j) = \Delta \cap s'_+$. On ajoute au graphe $G_{\pi/2}$ une nouvelle cellule C_k adjacente à C_i et C_j et on calcule $g_-(C_k) = d_-(C_i),\ d_+(C_k) = d_+(C_j)$. On supprime s, t et s' dans L.

On laisse le soin au lecteur de traiter le cas f_4 qui est voisin du cas f_2.

• L'événement e_i est un *milieu*, il y a quatre types de milieux, notés m_1, m_2, m_3, m_4 (figure 1.13).

$_*$ de type m_1 : Soit s_+ le successeur de s dans L, et C_i la cellule située entre s et s_+. On remplace s par s' dans L, et on termine le traitement de C_i par

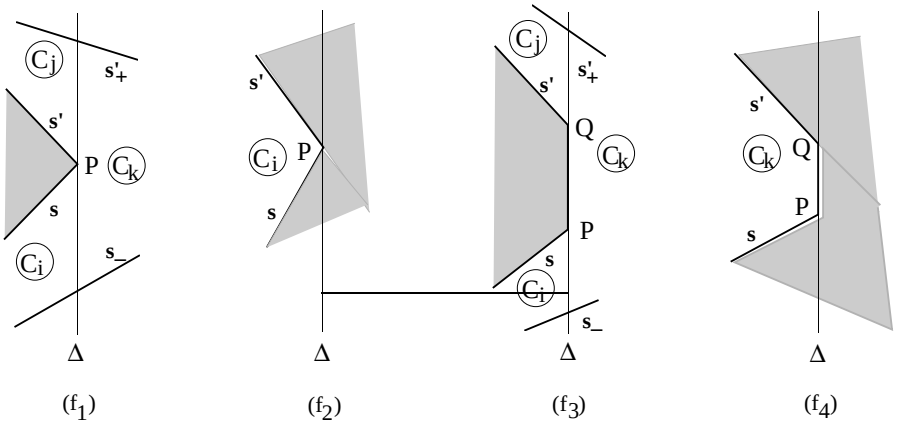


Figure 1.12: *L'événement en cours est une fin.*

$d_-(C_i) = P,\ d_+(C_i) = \Delta \cap S_+$, enfin on crée une nouvelle cellule C_j adjacente à C_i en posant $g_-(C_j) = P,\ g_+(C_j) = d_+(C_i)$.

On laisse au lecteur le soin d'examiner les autres cas $m_2,\ m_3,\ m_4$ qui se traitent de manière similaire.

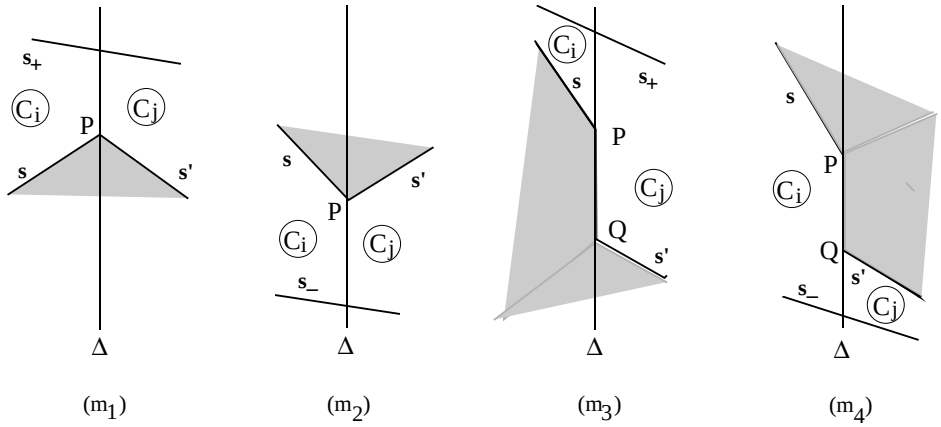


Figure 1.13: *L'événement en cours est un milieu.*

• L'événement e_i est un *début*, il y a quatre types de débuts, notés d_1, d_2, d_3, d_4 (figure 1.14). On laisse le soin au lecteur de traiter ce cas en s'inspirant de ce qui a été fait précédemment et de la figure 1.14.

En vue d'évaluer l'algorithme proposé, précisons tout d'abord quelles structures de données sont utilisées pour implémenter l'algorithme. Les opérations effectuées sur la liste L justifient l'emploi d'une structure de tas. Les régions sont numérotées par ordre croissant d'apparition dans le balayage. Le graphe $G_{\pi/2}$ est représenté par une liste d'adjacence, chaque cellule étant représentée par un enregistrement contenant ses quatre sommets. La liste $\mathcal{V}$ est gérée par une structure d'arbre binaire équilibré (AVL par exemple), chaque nœud représentant un segment s contient un pointeur sur son successeur s_+ et son prédécesseur s_- ainsi que (dans

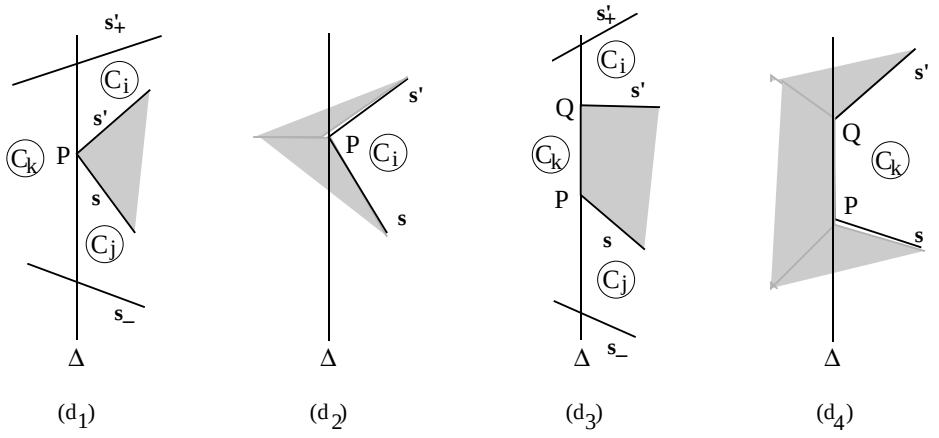


Figure 1.14: *L'événement en cours est un début.*

le cas où il n'est pas vertical) un pointeur dans la liste d'adjacence de $G_{\pi/2}$ vers la cellule incidente à s.

Grâce à ce choix de structures de données, on déduit :

Proposition 1.4. *Si* $\mathrm{Card}(\mathcal{S}) = n$*, le calcul du graphe* $G_{\pi/2}$ *par la procédure* GRAPHE-DES-CELLULES($G_{\pi/2}$) *s'effectue en temps* $O(n \log n)$ *et en espace* $O(n)$.

Preuve. La ligne (1) prend un temps $O(n \log n)$, la ligne (2) un temps $O(1)$. Dans la ligne (4), la mise à jour de $\mathcal{V}$ prend un temps $O(\log n)$ et celle de $G_{\pi/2}$ un temps $O(1)$. Le résultat est donc démontré en ce qui concerne le temps. Quant à l'espace, le graphe $G_{\pi/2}$ étant planaire, l'espace utilisé est bien $O(n)$ car chaque cellule contient au moins un sommet sur sa frontière et chaque sommet appartient à la frontière d'au plus trois cellules. ∎

Emondage des cellules - Calcul du graphe $G_{\pi/2}(l)$

Il reste à modifier le graphe $G_{\pi/2}$ obtenu en fonction de la longueur du segment à déplacer pour obtenir le graphe $G_{\pi/2}(l)$. Voici l'algorithme :

```
procédure EMONDAGE;
(1)  pour chaque cellule C = (d_-(C), d_+(C), g_+(C), g_-(C)) faire
(2)     remplacer C par la cellule C' obtenue
        en translatant [g_+(C), d_+(C)] d'une longueur l vers le bas;
(3)     si C' est vide alors supprimer C dans G_{π/2}
(4)     sinon pour chaque cellule C'' adjacente à C faire
(5)        si C'' n'est pas voisin de C' alors
           supprimer dans G_{π/2} l'arc (C, C'') finsi
        finsi
     finpour.
```

Proposition 1.5. *Si* $\mathrm{Card}(\mathcal{S}) = n$*, le calcul de* $G_{\pi/2}(l)$ *à partir de* $G_{\pi/2}$ *s'effectue en temps* $O(n)$ *par la procédure* EMONDAGE .

Preuve. Les lignes (2) et (3) s'exécutent en temps $O(1)$ ainsi que la ligne (5) car chaque cellule, compte tenu des hypothèses faites (pas plus de 2 sommets alignés verticalement) admet au plus six cellules voisines . Les lignes (2), (3) et (5) sont exécutées $O(n)$ fois, d'où le résultat. ∎

Les algorithmes GRAPHE-DES-CELLULES($G_{\pi/2}$) et EMONDAGE constituent donc le prétraitement des données pour $\theta = \pi/2$. Pour calculer le graphe G_θ pour une valeur de θ quelconque on transforme au préalable l'enesemble $\mathcal{S}$ par une rotation d'angle de mesure $\pi/2 - \theta$ ce qui prend un temps $O(n)$ et ne change donc pas la complexité de l'algorithme.

12.1.4 Résolution du problème de translation

Revenons aux questions posées au début de cette section, à savoir :

(1) : Déterminer si une position p appartient à $\mathcal{L}_{\pi/2}$.

Pour résoudre ce problème, remarquons que dans la liste d'adjacence du graphe $G_{\pi/2}(l)$, les cellules sont rangées par ordre croissant d'abscisse du côté gauche, et en cas d'égalité par ordre croissant d'ordonnée. Donc, en procédant par dichotomie, on détermine en temps $O(\log n)$ si p appartient à une cellule du graphe $G_{\pi/2}(l)$ et donc à $\mathcal{L}_{\pi/2}$.

(2) : Etant données deux positions libres p_0 et p_1, déterminer s'il existe un mouvement m de $\mathcal{X}$ de la position p_0 à la position p_1, et si oui en calculer un.

Ce deuxième problème se résout comme suit. On calcule comme indiqué ci-dessus les cellules $C(p_0)$ et $C(p_1)$ auxquelles p_0 et p_1 appartiennent respectivement. Il ne reste plus qu'à calculer si $C(p_0)$ et $C(p_1)$ appartiennent à la même composante connexe de $G_{\pi/2}(l)$ et à donner en cas de réponse positive un mouvement possible ce qui se fait en temps $O(n)$ (proposition 1.3).

Il nous reste à démontrer que le résultat reste valable dans le cas général c'est-à-dire sans restriction sur le nombre de sommets alignés, ni sur les obstacles qui peuvent avoir un morceau de frontière commune (leur intersection restant d'intérieur vide).

Si les obstacles ne sont pas disjoints, les définitions de *Inf*(p), *Sup*(p) pour une position libre p s'adaptent sans problème, il y a seulement plus de cas à traiter en gardant le même principe.

Par contre, le fait que le nombre possible de sommets alignés soit quelconque nécessite une adaptation des algorithmes, car un événement peut contenir plus de quatre segments. Cela ne pose pas de problème majeur pour l'algorithme : on traite les segments dans l'ordre croissant d'ordonnée, simplement lors de la

création d'une cellule, le calcul de son sommet gauche supérieur reste en attente un certain temps (figure 1.15).

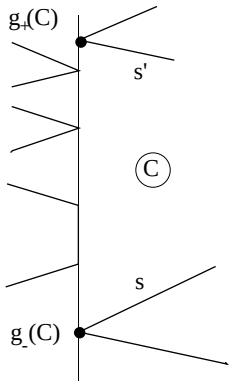


Figure 1.15: *$g_+(C)$ est calculé au cours de l'insertion de s'.*

Quant à l'algorithme EMONDAGE, il faut modifier l'implémentation du graphe $G_{\pi/2}$, si l'on veut conserver la même complexité en temps. En effet le nombre de cellules adjacentes à une cellule donnée est maintenant $O(n)$. Donc pour la mise à jour des relations d'adjacence la ligne (5) ne prend plus un temps $O(1)$. On modifie l'implémentation de la façon suivante : les listes de voisins sont des listes doublement chaînées, et pour chaque arête (C, C'), on établit un pointage aller retour entre l'enregistrement de la cellule C' dans la liste des voisins de C et l'enregistrement de la cellule C dans la liste des voisins de C'. Ainsi, la boucle 4 prend un temps $O(k)$ où k est le nombre de voisins de C. L'algorithme prend alors un temps $O(m)$ où m est le nombre d'arêtes du graphe, donc un temps $O(n)$ puisque le graphe est planaire.

Théorème 1.6. *Si* $\mathrm{Card}(\mathcal{S}) = n$*, le problème de translation d'un segment se résout par un prétraitement en temps* $O(n \log n)$ *et en espace* $O(n)$ *; on détermine en temps* $O(\log n)$ *si une position donnée est libre, et en temps* $O(n)$ *s'il existe un mouvement libre d'une position* p_0 *à une position* p_1*.*

12.2 Déplacement d'un disque

12.2.1 Introduction

Le cadre que nous nous fixons ici est le suivant : les obstacles sont des polygones simples non chevauchants, mais pouvant partager des *sommets* communs.

L'objet à déplacer est un disque de rayon r. Dans toute cette section n désigne le nombre de côtés des obstacles. Etant données les propriétés de symétrie d'un disque, les définitions vues dans l'introduction se simplifient. Une *position du*

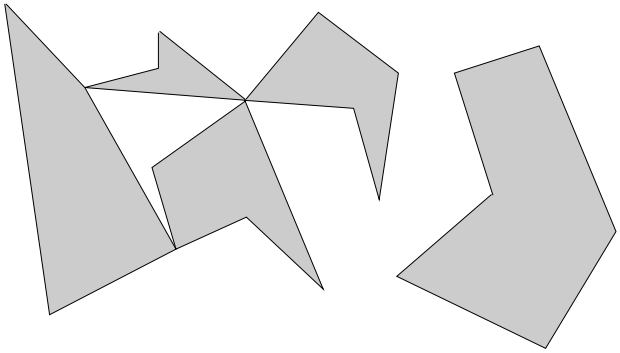


Figure 2.1: *Exemple d'obstacles.*

disque est simplement définie par la position de son centre, donc par un point. Et un *mouvement du disque de la position* $p_0 \in \mathbb{R}^2$ *à la position* $p_1 \in \mathbb{R}^2$ est une application continue m de $[0,1]$ dans R^2 telle que $m(0) = p_0$ et $m(1) = p_1$. L'ensemble $m[0,1]$ est la *trajectoire* du disque.

Le problème de la planification de trajectoire d'un disque se formule ainsi :

Etant donné un ensemble $\mathcal{S}$ des segments constituant les côtés des obstacles, avec pour chaque segment la donnée de son successeur et de son prédécesseur dans le contour direct de l'obstacle auquel il appartient, et le rayon r du disque à déplacer :

(**1**) Déterminer si une position $p \in \mathbb{R}^2$ est libre.

(**2**) Etant données deux positions libres p_0 et p_1, déterminer s'il existe un mouvement m de $\mathcal{X}$ de la position p_0 à la position p_1, et si oui en calculer un.

12.2.2 Rétraction — Diagramme de Voronoï

On suppose comme dans 12.1 que l'espace libre $\mathcal{E}$ est à l'intérieur d'un rectangle.

L'espace $\mathcal{E}$ est un ouvert du plan, et sa frontière $F(\mathcal{E})$ est constituée de lignes polygonales. On divise chaque segment constituant un côté de $F(\mathcal{E})$ en trois éléments disjoints : ses extrémités et l'intervalle ouvert restant. Soit $\mathcal{S}$ l'ensemble de ces éléments constitué de points et d'intervalles ouverts deux à deux disjoints. On notera $Vor'(\mathcal{S})$ le diagramme de Voronoï extérieur de $\mathcal{S}$ i.e. **la restriction de** $Vor(\mathcal{S})$ **à l'espace libre** $\mathcal{E}$ **et à sa frontière** $F(\mathcal{E})$.

Conformément aux résultats vus dans la section 11.4, $Vor(\mathcal{S})$ est un graphe planaire qui partitionne le plan en régions $R(s)$ pour $s \in \mathcal{S}$ (les régions sont fermées, il s'agit donc d'un partitionnement au sens où deux régions ne se chevauchent pas). Par ailleurs, pour $s \in \mathcal{S}$, $R(s)$ jouit des propriétés suivantes :

- $R(s)$ est homéomorphe à un disque fermé (éventuellement de rayon nul ou infini).

- La frontière de $R(s)$ est constituée de segments de droite et d'arcs de parabole.
- $R(s)$ est étoilée relativement à s.
- Si s est un point alors s appartient à la frontière de $R(s)$ (cela est dû au fait qu'il existe dans $\mathcal{S}$ un intervalle d'extrémité s).
- Si s est un intervalle, alors s est contenu dans l'intérieur de $R(s)$, et la frontière de $R(s)$ passe par les extrémités de s (figure 2.2).

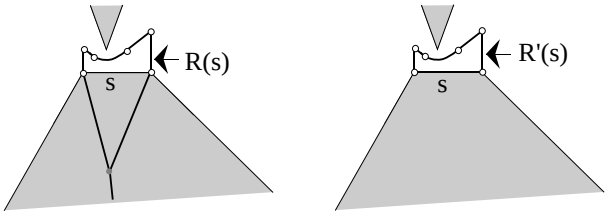


Figure 2.2: *Une région $R(s)$, et la région $R'(s)$ associée (en gras).*

Il s'ensuit que $Vor'(\mathcal{S})$ partitionne l'espace $\bar{\mathcal{E}} = \mathcal{E} \cup F(\mathcal{E})$ en régions $R'(s) = R(s) \cap \overline{\mathcal{E}}$ pour $s \in \mathcal{S}$ qui vérifient des propriétés déduites de celles de $R(s)$:

- $R'(s)$ est homéomorphe à un disque fermé (éventuellement de rayon nul).
- La frontière de $R'(s)$ est constituée de segments de droite et d'arcs de parabole.
- $R'(s)$ est étoilée relativement à s.
- Si s est un point alors s appartient à la frontière de $R'(s)$.
- Si s est un intervalle, alors s est contenu dans la frontière de $R'(s)$ ainsi que ses extrémités.

Amplitude et voisins d'un point de $\mathcal{E}$

Pour tout point $p \in \mathcal{E}$ on définit *l'amplitude de p*, que l'on note $\delta(p)$, comme étant la distance de p au complément de $\mathcal{E}$ dans le plan. Comme $\mathcal{E}$ est ouvert, cette distance est «atteinte», ou encore il existe au moins un point $q \notin \mathcal{E}$ tel que $\delta(p) = d(p, q)$. Naturellement, q est sur la frontière de $\mathcal{E}$ c'est-à-dire appartient à l'un des côtés des obstacles. En fait, $\delta(p)$ représente tout simplement la valeur maximale du rayon d'un disque ouvert de centre p contenu dans $\mathcal{E}$. On a donc :

$$\mathcal{L} = \{p \in \mathcal{E} \mid \ \delta(p) > r\}$$

Notons que l'ensemble des points $q \notin \mathcal{E}$ tels que $\delta(p) = d(p, q)$ est fini puisque c'est l'ensemble des points d'intersection d'un cercle avec un nombre fini de segments. Ce sont les points du complémentaire de $\mathcal{E}$ les plus près de p, on les appellera *voisins de p*. On notera *voisins*(p) cet ensemble :

$$\mathit{voisins}(p) = \{q \notin \mathcal{E} \mid \delta(p) = d(p, q)\}$$

Avec ces notations, il est clair qu'un point p appartient à $Vor(\mathcal{S})$ si et seulement si $voisins(p)$ possède plus d'un point.

$$Vor(\mathcal{S}) = \{p \mid \text{Card}(voisins(p)) \geq 2\}$$

Rétraction de $\mathcal{E}$ sur $Vor'(\mathcal{S})$

On définit une application $\rho : \mathcal{E} \longrightarrow Vor'(\mathcal{S})$ appelée *rétraction*, de la façon suivante :

Soit p un point de $\mathcal{E}$, et q un point de $voisins(p)$.

Si q est le seul voisin de p, alors q est soit un point élément de $\mathcal{S}$, soit un point d'un intervalle s de $\mathcal{S}$, et dans les deux cas, p est intérieur à une région $R'(s)$, avec $s \in \mathcal{S}$ et $d(p,s) = d(p,q) = \delta(p)$. Comme $R'(s)$ est étoilée, d'intérieur non vide puisqu'il contient p, homéomorphe à un disque, on en déduit que la demi-droite $[q, \overrightarrow{qp}($ coupe la frontière de $R'(s)$ en un point **unique** $t \in Vor'(\mathcal{S})$; on pose alors $\rho(p) = t$.

Si $voisins(p)$ admet plusieurs points, alors p appartient à $Vor'(\mathcal{S})$ et les demi-droites $[q, \overrightarrow{qp}($ définies précédemment définissent toutes le même point t à savoir précisément le point p lui-même. Donc, la définition de ρ donnée dans le cas où $voisins(p)$ n'a qu'un point reste consistante dans les autres cas et implique que ρ est l'identité pour les points de $\mathcal{E}$ qui sont sur $Vor'(\mathcal{S})$.

Proposition 2.1. *a) La rétraction ρ est une application continue de $\mathcal{E}$ sur $Vor'(\mathcal{S}) \cap \mathcal{E}$.*

b) Si $\rho(p) \neq p$ alors la fonction δ est strictement croissante sur le segment allant de p à $\rho(p)$.

Preuve. a) L'application ρ est clairement continue en tout point de $\mathcal{E} - Vor'(\mathcal{S})$. Si $p \in Vor'(\mathcal{S})$ alors p appartient à la frontière de plusieurs régions $R(s_1), \ldots, R(s_k)$, et pour toute suite de points p_i tendant vers p et restant dans R_{s_j}, la suite $\rho(p_i)$ tend vers $\rho(p) = p$. Donc pour toute suite de points p_i tendant vers p, $\rho(p_i)$ tend vers $\rho(p)$.

b) Si $\rho(p) \neq p$ alors p appartient à l'intérieur d'une région $R'(s)$ et pour tout point $p' \in]p, \rho(p)]$ on a :

$$\delta(p) = d(p,s) = d(q,p) < d(q,p') = d(s,p') = \delta(p')$$

■

Corollaire 2.2. *Si $p \in \mathcal{L}$ alors il existe un mouvement libre (rectiligne) du disque de la position p vers la position $\rho(p)$.*

Preuve. C'est une conséquence immédiate de la partie b) de la proposition. ■

Proposition 2.3. *Il existe un mouvement libre du disque de la position $p_0 \in \mathcal{L}$ vers la position $p_1 \in \mathcal{PL}$ si et seulement s'il existe un mouvement libre du disque de la position $\rho(p_0)$ vers la position $\rho(p_1)$ dans $Vor'(\mathcal{S}) \cap \mathcal{L}$.*

Preuve.

La condition est suffisante :

Par le corollaire 2.2, il existe un mouvement libre de p_0 vers $\rho(p_0)$ et un mouvement libre de $\rho(p_1)$ vers p_1, donc en composant les trois mouvements on obtient le résultat.

Réciproquement, soit μ un mouvement libre de p_0 vers p_1, et $\mathcal{C} = \mu[0,1]$. Alors, par la Proposition 2.1, $\rho(\mathcal{C}) \subset Vor'(\mathcal{S}) \cap \mathcal{L}$ et le composé $\rho \circ \mu$ est un mouvement libre du disque de la position $\rho(p_0)$ vers la position $\rho(p_1)$ dans $Vor'(\mathcal{S}) \cap \mathcal{L}$. On peut dire que $\rho \circ \mu$ est le «rétracté sur $Vor'(S)$» du mouvement μ, comme on le voit sur la figure 2.5. ∎

12.2.3 Exposé de l'algorithme

Question Q_1

Pour savoir si une position p est libre pour le disque donné, c'est-à-dire si le disque, centré au point p, est entièrement contenu dans l'espace libre $\mathcal{E}$, il existe une méthode «naïve» qui résout le problème en temps $O(n)$. Il suffit pour cela de calculer $\delta(p)$ en calculant la distance de p à chaque élément de $\mathcal{S}$. Cela permet, en considérant un point q appartenant à la frontière de $\mathcal{E}$ et qui réalise $d(p,q) = \delta(p)$, et en comparant la position du segment $[p,q]$ aux côtés des obstacles passant par q, de tester si $p \in \mathcal{E}$. Puis la comparaison de $\delta(p)$ avec le rayon r du disque détermine si p est une position libre.

Nous allons voir qu'un prétraitement en temps $O(n \log n)$ permet de résoudre la question en temps $O(\log n)$.

Prétraitement

Le graphe $Vor(\mathcal{S})$ partitionne le plan en régions dont les côtés sont des segments ou des arcs de parabole. On peut appliquer l'algorithme vu en 11.3.3, moyennant les modifications suivantes :

On détermine pour chaque arc de parabole ses points extrémaux (horizontalement parlant, c'est-à-dire ceux d'ordonnée extrémale), et on scinde l'arc en deux si ses points extrémaux ne coïncident pas avec ses extrémités (figure 2.3). Cette modification se fait en temps $O(n)$. Elle est nécessaire pour que la deuxième recherche dichotomique dans une bande verticale soit possible, et il faut pour cela qu'à l'intérieur de chaque bande, les «traces» des régions puissent être ordonnées verticalement.

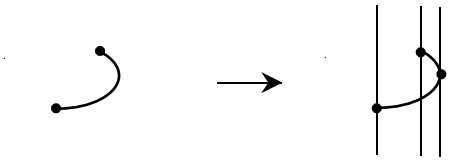


Figure 2.3: *Scission d'un arc de parabole.*

Ainsi utilisant les résultats des chapitres précédents, on en déduit le prétraitement suivant en temps $O(n \log n)$ et en espace $O(n)$:

procédure PRÉTRAITEMENT (Q_1);
(1) pour chaque point $q \in \mathcal{S}$ faire
établir une liste circulaire des côtés des obstacles incidents à q relativement à l'ordre polaire relatif à q
finpour;
(2) calculer $Vor(\mathcal{S})$;
(3) scinder les arcs de parabole de $Vor(\mathcal{S})$ si nécessaire;
(4) calculer l'arbre persistant $\mathcal{A}$ de la subdivision planaire associée.

Traitement de (Q_1)

Pour tester si p est une position libre, on peut alors appliquer l'algorithme de la section 11.3.3 qui détermine en temps $O(\log n)$ à quelle région de Voronoï $R(s)$ le point p appartient. Il reste à déterminer si cette position est libre. Or la position de p relativement à s détermine si $p \in \mathcal{E}$ en temps $O(1)$ si s est un intervalle, et en temps $O(\log n)$ grâce à la ligne (1) du prétraitement si s est un point. En effet, chaque intervalle s a une orientation liée au contour direct du polygone auquel il appartient; on détermine donc en temps constant si p est à gauche ou à droite de l'intervalle orienté, ce qui permet d'en déduire s'il est dans $\mathcal{E}$ ou non. Par contre, si s est un point, le nombre d'obstacles incidents à s est $O(n)$ et déterminer si p est dans $\mathcal{E}$ se calcule en temps $O(\log n)$.

procédure POSITION-LIBRE(p);
(1) déterminer s tel que $p \in R(s)$;
(2) déterminer selon la nature de s (point ou intervalle) et la place de p relativement à s si $p \in \mathcal{E}$;
(3) $l := \delta(p)$;
(4) si $l > r$ alors $p \in \mathcal{L}$ sinon $p \notin \mathcal{L}$.

D'où le résultat :

Théorème 2.4. *Etant donné un espace libre polygonal borné à n sommets et un disque de rayon donné, un prétraitement (indépendant du disque) en temps $O(n \log n)$ et en espace $O(\log n)$ permet de déterminer en temps $O(\log n)$ si une position donnée du disque est une position libre.*

Question Q_2

Etant données deux positions libres p_0 et p_1, il s'agit de déterminer s'il existe un mouvement m de $\mathcal{X}$ de la position p_0 à la position p_1, et si oui d'en calculer un.

On utilise pour ce faire les résultats obtenus en 12.2.2. Conformément à la proposition 2.3, il suffit de déterminer s'il existe un mouvement libre du disque de la position $\rho(p_0)$ vers la position $\rho(p_1)$ dans $Vor'(\mathcal{S}) \cap \mathcal{L}$. Pour résoudre ce problème efficacement, il suffit d'ajouter dans le graphe $Vor'(\mathcal{S})$ une information supplémentaire relative à chaque arête.

Soit e une arête de $Vor'(\mathcal{S})$. Rappelons que cette arête est un segment ou un arc de parabole. On définit sa *largeur* $\lambda(e)$ comme étant $max\{\delta(p) \mid p \in e\}$. Or pour chaque arête e, il existe deux éléments s et s' de $\mathcal{S}$ tels que pour tout point p de e on ait :

$$\delta(p) = d(p, s) = d(p, s')$$

On peut donc calculer en temps $O(1)$ le nombre $\lambda(e)$. La figure 2.4 donne un exemple.

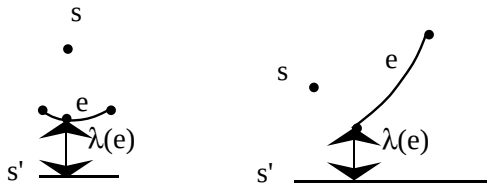


Figure 2.4: $\lambda(e)$ *lorsque e est un arc de parabole.*

Ce prétraitement étant réalisé, la résolution du problème devient : soient e_0 et e_1 les arêtes respectives auxquelles $\rho(e_0)$ et $\rho(e_1)$ appartiennent. Si $e_0 = e_1$, on détermine en temps constant si le mouvement de $\rho(p_0)$ vers $\rho(p_1)$ le long de e_0 est libre. Sinon $\rho(p_0)$ scinde l'arête e_0 en deux nouvelles arêtes e'_0 et e''_0 (l'une pouvant être de longueur nulle) dont on calcule la largeur, de même pour $\rho(p_1)$. On doit scinder les arêtes e_0 et e_1, car il se peut que l'on ait $\lambda(e_0) < r$ et que néanmoins l'une des deux arêtes e'_0 ou e''_0, par exemple e'_0, ait une largeur $> r$, auquel cas le disque peut se déplacer à partir de la position $\rho(p_0)$ le long de l'arête e_0 dans un sens (à savoir sur e'_0), mais pas dans l'autre (c'est-à-dire sur e''_0). Il reste à chercher dans le graphe $Vor'(\mathcal{S})$ ainsi transformé s'il existe un chemin de $\rho(p_0)$ vers $\rho(p_1)$ empruntant uniquement des arêtes f tels que $\lambda(f) > r$. Cette recherche se fait en temps $O(n)$.

Prétraitement

procédure PRÉTRAITEMENT (Q_2);
(1) calculer le diagramme de Voronoï $Vor(\mathcal{S})$;
(2) construire le sous-graphe $Vor'(\mathcal{S})$ de $Vor(\mathcal{S})$ obtenu en supprimant les arêtes de $Vor(\mathcal{S})$ intérieures aux obstacles
(3) pour chaque arête $e \in Vor'(\mathcal{S})$ calculer $\lambda(e)$.

Complexité du prétraitement

- L'étape (1) prend un temps $O(n \log n)$ (théorème 4.11 du chapitre 11);

- Chaque arête e de $Vor(\mathcal{S})$ est (aux extrémités près) contenue dans un obstacle ou dans l'espace libre $\mathcal{E}$, et on en décide en temps constant en examinant sa situation par rapport aux deux éléments s et s' tels que e est une arête de la frontière entre les deux régions $R(s)$ et $R(s')$. Ainsi l'étape (2) prend un temps $O(n)$;

- L'étape (3), comme on l'a vu, se réalise en temps $O(n)$.

Traitement de (Q_2)

procédure MOUVEMENT LIBRE DISQUE(p_0, p_1);
(1) $p'_0 := \rho(p_0)$; $p'_1 := \rho(p_1)$; {Calcul des rétractés}
(2) soient e_0 et e_1 les arêtes respectives de $Vor'(\mathcal{S})$ auxquels p'_0 et p'_1 appartiennent;
(3) pour $i = 0, 1$ faire
  remplacer dans $Vor'(\mathcal{S})$, l'arête e_i par les deux arêtes e'_i et e''_i obtenues par scission de e_i par p'_i et calculer leur largeur;
  finpour
(4) chercher un chemin de p'_0 à p'_1 dans le nouveau graphe utilisant uniquement des arêtes de largeur $> r$ joignant p'_0 à p'_1.

Exemple.

La figure 2.5 donne un exemple de résolution du problème.

Le mouvement du disque calculé par l'algorithme se décompose donc en trois mouvements : le premier, qui est rectiligne, amène le disque de la position p_0 à la position rétractée $\rho(p_0)$ qui est sur une arête de Voronoï; le deuxième mouvement fait glisser le centre du disque le long d'un chemin formé d'arêtes de Voronoï (qui «guident» le mouvement) vers la position rétractée $\rho(p_1)$ de la position p_1; puis le disque quitte le diagramme de Voronoï et, par un mouvement rectiligne arrive en position p_1.

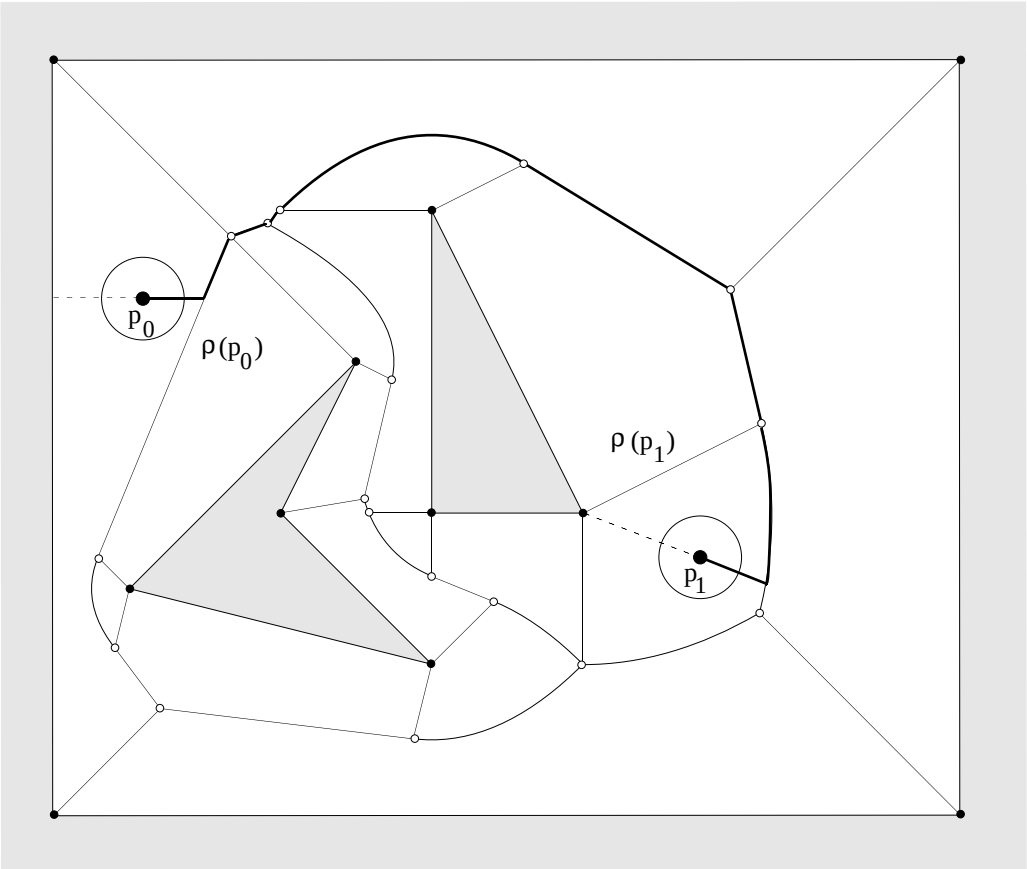


Figure 2.5: *Un exemple de mouvement utilisant cet algorithme.*

Théorème 2.5. *Etant donné un espace libre polygonal borné à n sommets, un disque de rayon donné, et deux positions libres p_0 et p_1 de ce disque, un prétraitement (indépendant du disque) en temps $O(n \log n)$ et en espace $O(n)$ permet de déterminer en temps $O(n)$ un mouvement libre de ce disque, s'il existe, de la position p_0 vers la position p_1.*

Preuve.

- On détermine en temps $O(\log n)$ à quelle région $R(s_i)$ chaque position p_i appartient. Comme $R(s_i)$ est étoilée relativement à s_i, il s'ensuit que les extrémités des arêtes de la frontière de $R(s_i)$ dans un parcours positif constituent le circuit polaire de ces points relativement à un point quelconque de s_i, et donc on peut déterminer en temps $O(\log m)$ l'image $\rho(p_i)$ où m est le nombre d'arêtes de la frontière de $R(s_i)$. Ainsi (1) prend un temps $O(\log n)$.
- (3) prend un temps $O(n)$, de même que (4). ∎

12.2.4 Remarques

Il existe une autre approche, très naturelle, de ce problème et analogue à celle que nous avons développée pour le déplacement d'un segment en translation, qui consiste, d'une certaine manière à chercher des déplacements qui « longent » les obstacles. Ces déplacements sont semi-libres et constituent les « positions » limites des déplacements libres. On peut comme dans 12.1.2 diviser l'espace libre en cellules qu'on émonde ensuite en fonction du rayon du disque, ce qui donne lieu à une division de l'ensemble des positions libres en composantes connexes. Un mouvement libre est possible d'un position à une autre si ces deux positions appartiennent à la même composante connexe (figure 2.6).

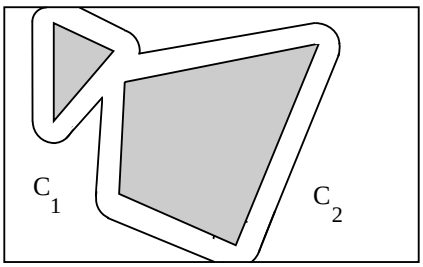


Figure 2.6: *Composantes connexes des déplacements libres.*

L'approche qui est exposée ici est en quelque sorte la stratégie opposée, à savoir que l'on cherche des déplacements le plus loin possible des obstacles puisqu'on déplace essentiellement le disque sur le diagramme de Voronoï des obstacles. Cette approche présente un avantage par rapport à l'autre : le prétraitement est indépendant de la taille du disque à déplacer, ce qui n'est pas le cas dans l'autre méthode où l'émondage est fonction du disque choisi. Par ailleurs l'algorithme peut s'étendre à des objets convexes de forme polygonale si l'on se restreint à des mouvements de translation, ainsi qu'à des objets réduits à un segment pour des mouvements quelconques.

Il faut signaler par contre aussi les inconvénients de cette méthode. Pour les raisons que nous avons évoquées plus haut, elle est loin de fournir le « plus court » déplacement, comme le montre l'exemple de la figure 2.7 : si l'on considère deux positions très proches et très près d'un des côtés de l'espace libre constitué d'un rectangle simple, l'algorithme fournit un déplacement passant par le « centre » du rectangle et donc très loin du plus court chemin.

Notes

L'étude du problème de planification de trajectoires sous son aspect algorithmique est très récente puisque tous les travaux réalisés sur ce sujet datent de moins de

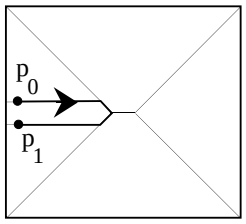


Figure 2.7: *Un «mauvais déplacement».*

dix ans. Il existe plusieurs approches qui font appel à des domaines variés des mathématiques allant de la géométrie traditionnelle à la géométrie algébrique en passant par la topologie, l'algèbre et la combinatoire.

La théorie des ensembles semi-algébriques est une première approche qui permet de résoudre le problème sous des hypothèses très générales, mais dont la complexité exponentielle est un inconvénient majeur d'un point de vue algorithmique; c'est pourquoi nous n'en avons pas parlé ici. Néanmoins ces travaux dus en particulier à J. Canny sont très intéressants d'un point de vue théorique et permettent de mieux cerner la nature et les difficultés du problème.

Les exemples que nous avons traités illustrent deux autres méthodes.

L'algorithme de déplacement d'un segment par translation dans un environnement polygonal dû à D. Leven et M. Shamir :

D. Leven et M. Shamir, An Efficient and Simple Motion Planning Algorithm for a Ladder Amiddst Polygonal Barriers, *J. Algorithms* **8** (1987), 192–215

correspond à une méthode de projection qui est utilisée dans un cadre plus général que celui que l'on s'est fixé, en particulier pour des objets de forme polygonale quelconque dans le plan.

L'algorithme de déplacement d'un disque dans un environnement polygonal dû à C. Ó'Dunlaing et C. Yap :

C. Ó'Dunlaing, C. Yap, A "retraction" method for planning the motion of a disk, *J. Algorithms* **6** (1985), 104–111

est obtenu par un principe de rétraction. Cette approche a été étendue au cas d'un segment, et à la translation d'un polygone convexe.

Les références à ces travaux se trouvent dans :

J.T. Schwartz et M. Shamir, Algorithmic Motion Planning in Robotics, *Handbook of Theoretical Computer Science Vol. A*, Elsevier, (1990), 391–430.

Index

Notes et Compléments

Ces notes et compléments sont écrits depuis la mise sur le réseau du manuscrit. Ils contiennent les corrections d'erreurs, et aussi quelques indications sur des développements plus récents.

Chapitre 3. Structures de données

Gestion des partitions

Note 1 (page 65)

Une preuve plus simple est donnée dans la leçon 11, pages 52–57, du livre : D. Kozen, *The Design and Analysis of Algorithms*, Springer-Verlag, 1991.

Chapitre 9. Automates

Algorithme de Hopcoft

Note 2 (page 321)

Il faut lire $L_p = \{w \in A^* \mid p \cdot w \in T\}$.

Note 3 (page 323)

Dans la procédure, dernière ligne, lire $\mathcal{P} := \mathcal{P} \lhd \mathcal{L}$

Note 4 (page 324)

Il manque une ligne : la scission par $(238, b)$:

$\mathcal{P} : 46 - 123578$	
Scission relative à $(46, a)$	$\mathcal{P} : 46 - 12378 - 5$
Scission relative à $(5, b)$	$\mathcal{P} : 46 - 17 - 238 - 5$
Scission relative à $(238, b)$	$\mathcal{P} : 46 - 17 - 28 - 3 - 5$
Scission relative à $(28, a)$	$\mathcal{P} : 46 - 1 - 7 - 2 - 8 - 3 - 5$
Terminé	

Une présentation détaillée de l'algorithme de Hopcoft est parue dans :

T. Knuutila, Re-describing an algorithm by Hopcroft, *Theoretical Computer Science* **250** (2001), 333–363.