Algorithmique, graphes et programmation dynamique

Notes de Cours

Rapport de Travaux Pratiques

Laurent Canet

Le 2 juillet 2003

Table des matières

Introduction

Ce texte n'a aucune valeur officielle de support pour les cours concernés. Il contient vraisemblablement beaucoup d'erreurs et d'inexactitudes, le lecteur se reportera à un ouvrage de réference pour les vérifications.

Le rapport de TP de programmation dynamique a été co-écrit avec mon binôme Thibaut Varene. Les sujets et les rapports de travaux pratiques de graphes ont été écrit par Gilles Bertrand et Michel Couprie.

Première partie

IN202 - Algorithmique

Introduction au cours d'algorithmique

Ce document fut à l'origine mes notes du cours d'*IN202* de l'ESIEE Paris. Cette unité a pour but d'initier les élèves ingénieurs à l'algorithmique grâce à des exemples détaillées. Le cours était divisé en 2 grandes parties, d'abord un apercu de l'algorithmique ainsi que de la notion de complexité en s'appuyant sur des exemples tels que le QUICKSORT, puis une description de l'approche dite de programmation dynamique. Des notions de programmation en C ainsi des connaissances basiques en analyses sont requises pour comprendre ce cours. D'autres unités du tronc commun ou de la majeure informatique de l'ESIEE Paris élargissent et approfondissent le sujet.

Mes notes de cours datent de mi-2002. Durant l'été de cette année j'ai enrichi le cours notament de la notion de classe de problème ainsi que des démonstrations, notament la complexité minimale d'un algorithme de tri ; finalement j'ai décidé de les publier sur ma page web.

Ces notes de cours ne constituent en rien un support officiel, il faut impérativement se réfèrer à un ou des ouvrages spécialisés qui seront cités dans une bibliographie (qui n'a pas encore été faite).

Je n'ai en aucun cas la prétention de faire un document interessant, exact ou précis. Néanmoins Si vous estimez que je racontre vraiment n'importe quoi, signalez le moi sur mon email : *canetl@esiee.fr*.

Chapitre 1

Système formel de preuve de programme de O'Hare

On peut formaliser un programme en une forme logique dite de *boîte noire* :

$$\underbrace{E}_{\text{Logique du 1er ordre}} \quad \underbrace{P}_{\text{Programme}} \quad \underbrace{S}_{\text{Logique du 1er ordre}}$$

De là, plusieurs règles agissent sur ces énoncés :

1.1 Règles et axiomes

Régle de la post-condition :

$$EPS' \quad ; \quad S' \Rightarrow S \quad \longmapsto EPS$$

Régle de la pré-condition :

$$E'PS \quad ; \quad E' \Rightarrow E \quad \longmapsto EPS$$

Régle du *ou* :

$$EPS \quad ; \quad E'PS \quad \longmapsto (E \vee E')PS$$

Régle du *et* :

$$EPS \quad ; \quad EPS' \quad \longmapsto EP(S \wedge S')$$

Règle du si-sinon :

$$(E \wedge B)PS \ ; \ (E \wedge \overline{B})QS \ \longmapsto E \ \{Si \ B \ alors \ P \ sinon \ Q\} \ S$$

Remarque : L'évaluation de B ne doit pas modifier E.

Règle du tant-que :

$$(E \wedge B)PE \ ; \longmapsto E\{tant \ que \ B \ faire \ P\}(E \wedge \overline{B})$$

Axiomes d'affectation :

$$EPA \ ; \ AQS \ \longmapsto E\{P;Q\}S$$

Exemple : Démontrer :

$$EPS \ ; \ E'PS' \ \longmapsto (E \wedge E')P(S \vee S')$$

1. EPS
2. $E'PS'$
3. $E \wedge E' \ \Rightarrow \ E$
4. $(E \wedge E')PS$ (Précondition sur 1 et 3)
5. $(E \wedge E')PS'$ (Précondition sur 2 et 3)
6. $(E \wedge E')P(S \wedge S')$ (Règle du ET sur 4 et 5)
7. $(S \wedge S') \ \Rightarrow \ (S \vee S')$
8. $(E \wedge E')P(S \vee S')$ (Postcondition sur 7 et 8)

1.2 Construction de programmes sur invariant

Construire un programme décrit par l'invariant, c'est utiliser une proposition booléenne $I(x_1, x_2, \cdots, x_n)$, fonction de plusieurs variables x_i, qui décrit le problème à un instant donné. Lorsque l'on veut faire un algorithme qui résoud EPS basé sur un invariant, on doit donner plusieurs propositions :

Invariant : Décrit la position du problème en fonction de variables x_i.

Initialisation : Ce sont des conditions sur les variables x_i, tels que la proposition E soit vérifié et $I(x_i)$ le soit également.

Condition d'arrêt : Ce sont des conditions sur les variables x_i, tels que la proposition S soit vérifiée. Lorsque la condition d'arrêt est vérifiée, le programme doit avoir fini son traitement, et S est verifiée.

Implications : Elles sont de la forme $I(x_1, x_2, \cdots, x_n) \bigwedge_i (P_i \ vraies) \Rightarrow I(x'_1, x'_2, \cdots, x'_3)$. Où P_i répresentent des propositions

On en déduit l'expression :

$$I(x_i)\ TANTQUE\ (\overline{\text{Condition d'arrêt}})FAIRE\ \ P_i\ \ I(x_i) \wedge (\text{Condition d'arrêt})$$

1.2.1 Exemple : somme des éléments d'un tableau

Soit un programme réalisant la somme de tous les éléments de 0 à $n-1$ d'un tableau T. On veut un programme P vérifiant l'énoncé suivant :

$$(n > 0)\ \ P\ \ \left(S = \sum_{i=0}^{n-1} T[i]\right)$$

On se propose de décrire ce programme par l'invariant :

Invariant : $I(s,k) \equiv s = \sum_{i=0}^{k-1} T[i]$. s est la somme des k premiers éléments de T.

Initialisation : $s = 0$ et $k = 0$. En effet la somme des 0 premiers éléments fait 0. Donc $I(0,0)$ est vérifié.

Condition d'arrêt : $k = n$. On s'arrête lorsque l'on a réalisée la somme des n premiers éléments, comme demandé.

Implication : $I(s,k) \wedge (s = s + T[k]) \Rightarrow I(s, k+1)$.

En sortie du programme, on aura $I(s,k) \wedge (k = n)$. La translation de cet algorithme en pseudo langage est très simple :

```
s <- 0
k <- 0
// Ici on a I(0,0)
tant que (k != n) faire :
    s <- s + T[k]
    // Ici on a I(s,k+1)
    k <- k + 1
    // Ici on a I(s,k)
fin tant que
// Ici on a I(s,n)
```

Chapitre 2

Problèmes de recherches

Le problème que nous nous posons est de rechercher un élément (nombre, caractère ou autre) dans un tableau homogène d'élements. La particularité de ce tableau est qu'il est dejà trié dans l'ordre croissant (ce qui impose d'avoir une relation d'ordre entre les éléments).

2.1 Remarques sur l'évaluation de complexité d'un programme

Dans ce paragraphe et ceux qui suivront, nous allons évaluer le temps d'éxecution des programmes en fonction de la taille n du problème. Comparer 2 programmes, c'est comparer les temps d'éxecution dans les pires des cas respectifs. Les constantes dépendent de la machine qui éxecutera l'algorithme.

Définition : Le programme P est de complexité $O(f(n))$ s'il existe α et n_0 tels que :

$$\forall n \geq n_0 \;\; T_p(n) \;\leq\; \alpha f(n)$$

Où $T_p(n)$ représente le temps d'éxecution du programme P dans le pire des cas.
Remarque : Si le programme P est en $O(n)$, alors il est aussi en $O(2n)$, $O(\frac{n}{2})$ ou $O(n^2)$.

2.2 Recherche dichotomique

Supposons que l'on cherche un élément noté e dans un tableau T. Le programme de recherche dichotomique que nous allons mettre au point répond à l'énoncé suivant :

$$(T[0] \leq e \leq T[n-1])\ RD\ (T[k] \leq e \leq T[k+1])$$

Invariant : $I(i,j) \equiv\ T[i] \leq e \leq T[j-1]$

Condition d'arrêt : $j = i + 2$

Initialisation : $(i = 0) \wedge (j = n)$

Implications : $I(i,j) \wedge ((j-i) > 2) \wedge \left(e < T[\frac{i+j}{2}]\right) \Rightarrow I(i, \frac{i+j}{2})$
$I(i,j) \wedge ((j-i) > 2) \wedge (e \geq T[\frac{i+j}{2}]) \Rightarrow I(\frac{i+j}{2}, i)$

Ce programme se base sur le fait que le tableau est dejà trié. On compare l'élément à rechercher avec l'élément du milieu du tableau. Si l'élément que l'on recherche est inférieur, alors il se trouve dans le sous-tableau de gauche, sinon dans celui de droite.
On continue à appliquer le même raisonnement de manière itérative jusqu'à ce que l'on a pu assez encadrer l'élément.

2.2.1 Code Source

```
1
2   i=0;
3   j=n;
4
5   while( j-i > 2)
6   {
7       if (e < T[(i+j)/2])  // I(i,(i+j)/2)
8           j = (j+i)/2;     // I(i,j)
9       else                 // I((i+j)/2,j)
10          i = (j+i)/2;     // I(i,j)
11  }
12  // Ici on a I(i,j)&&(j-i = 2)
```

On peut éventuellement faire une vérification préalable sur les données (pour voir si $e \in T[0 \cdots n-1]$)

2.2.2 Complexité

On rappelle l'invariant :

$$I(i,j) \equiv T[i] \leq e \leq T[j-1]$$

Précondition : $T[0] \leq e \leq T[n-1]$
Postcondition : $T[i] \leq e \leq T[i+1]$

L'initialisation se fait en temps constant (on suppose a). L'évaluation de la condition de la boucle est elle aussi en temps constant, noté b. Le corps de la boucle est également en temps constant c.

À chaque éxecution du corps de boucle, la taille du problème restant à traiter est divisée par 2. On a la suite valeurs $n, \frac{n}{2}, \frac{n}{4}, ..., 1$.

Hypothèse : La taille du problème n est supposée être un multiple de 2. $n = 2^p$. Avec $n = 2^p$ le corps de boucle est éxecuté p fois.

$$\begin{aligned} T_{rd} &= a + (p+1)b + pc \\ T_{rd} &= (a+b) + p(b+c) \\ T_{rd} &= \alpha log_2(n) + \beta \end{aligned}$$

Pour un problème de taille $n = 2^p$, la complexité de la recherche dichotomique est de forme logarithmique, mais supposons que la taille du problème ne soit pas un multiple de 2.
Pour $2^p \leq n \leq 2^{p+1}$

$$\alpha log_2(n) + \beta \leq T_{rd} \leq a + b(p+2) + (p+1)c$$

$$\alpha log_2(n) + \beta \leq T_{rd} \leq \alpha' log_2(n) + \beta'$$

On peut en déduire que le facteur dominant du temps de calcul est en $log_2(n)$, la recherche dichotomique est $\theta(log_2(n))$
Remarque : Le logarithme base 2 et le logarithme néperien (base e) ne différent que d'une constante, par conséquent la recherche dichotomique est $O(log(n))$.

2.3 Recherche séquentielle

La recherche séquentielle est une méthode plus traditionnelle : on regarde tous les éléments du tableau séquentiellement jusqu'á ce que l'on tombe sur e, l'élément que l'on cherchait.
L'avantage de la recherche séquentielle est qu'elle est possible sur des tableaux non-triés.

Invariant : $I(k) \equiv e \notin T[0...k-1]$

Initialisation : $k = 0$

Condition d'arrêt : $T[k] = e$

Implications : $I(k) \wedge (T[k] \neq e) \Rightarrow I(k+1)$

2.3.1 Code

```
1   k = 0;
2
3   while( T[k] != e)
```

```
4   {
5       k++
6   }
```

2.3.2 Complexité

La condition du *while* : `T[k] != e` est évaluée $n+2$ fois dans le pire des cas. Le corps de boucle `k++` est éxecutée n+1 fois dans le pire des cas.
Si on appelle a le temps d'éxecution de l'initialisation (k=0), b le temps d'éxécution de la condition de boucle, et c le temps d'éxécution du corps de boucle. Dans le pire des cas où $T[n] = e$, le temps d'éxécution de la recherche séquentielle est :

$$T_{rs} = a + b(n+2) + c(n+1)$$

Dans le cas général :

$$T_{rs}(n) = (b+c)n + a + 2b + c$$
$$T_{rs}(n) = \alpha n + \beta$$

On en déduit une complexité en $\theta(n)$.

2.4 Recherche arrière

Le but de cette algorithme est de rechercher un élément e dans un tableau à 2 dimensions $T[0...m-1][0...n-1]$ dont les lignes **ET** les colonnes sont triès par ordre croissant.
Exemple :

6	9	10	12	13	20
7	11	13	14	16	22
9	12	14	16	17	23
10	14	15	17	18	24
15	16	17	22	24	26

Nous avons 3 possibilités pour rechercher un élement dans un tel type de tableau :
- 1er algorithme : m recherches séquentielles : $\theta(m * n)$.
- 2e algorithme : m recherche dichotomiques : $\theta(m * log(n))$ ou $\theta(n * log(m))$.
- 3e algorithme : recherche arrière : $\theta(m + n)$.

L'algorithme de recherche arrière est basée sur l'invariant suivant :

Invariant : $I(x,p,q) \equiv e(m,n) = x + e(p,q)$ où $e(m,n)$ est le nombre de e dans $T[p...m-1][0...q]$

Initialisation : $(p = 0) \wedge (q = n - 1) \wedge (x = 0)$

Condition d'arrêt : $(p = m) \vee (q = -1)$

Implications :

$I(x, p, q) \wedge (p \neq m) \wedge (q \neq -1) \wedge (T[p][q] = e) \Rightarrow I(x + 1, p + 1, q - 1)$
$I(x, p, q) \wedge (p \neq m) \wedge (q \neq -1) \wedge (T[p][q] < e) \Rightarrow I(x, p + 1, q)$
$I(x, p, q) \wedge (p \neq m) \wedge (q \neq -1) \wedge (T[p][q] > e) \Rightarrow I(x, p, q - 1)$

2.4.1 Code

```
1   int p = 0;
2   int x = 0;
3   int q = n-1;
4
5   while( (p != n) && (q != -1) )
6   {
7       if (T[p][q] == e)           // I(x+1,p+1,q-1)
8           { x++; p++; q--; }      // I(x,p,q)
9       else if (T[p][q] < e)       // I(x,p+1,q)
10          { p++; }                // I(x,p,q)
11      else                        // I(x,p,q-1)
12          { q--; }                // I(x,p,q)
13  }
```

2.4.2 Complexité

Le corps de boucle est en temps constants (structure *if*, incrementations). La condition du while est elle aussi en $O(1)$, le corps de boucle est évalué au plus $m + n$ fois, donc la complexité de la recherche arrière est en $\theta(m + n)$

Chapitre 3

Problemes de tris

On veut trier un tableau $T[0...n-1] = [t_0...t_{n-1}]$ d'éléments par ordre croissant. Pour cela, on étudiera plusieurs algorithmes : le SLOWSORT, puis le QUICKSORT.

3.1 Slowsort

Invariant : $I(k) \equiv T[0...k-1]\ trie\ \nearrow \wedge T[0...k-1] \leq T[k...n-1]$

Condition d'arrêt : $k = n$

Initialisation : $(k = 0)$ ($I(0)$ est vrai)

Implications : $I(k) \wedge (k \neq n) \wedge (t_m = min(T[k...n-1])) \wedge (T[k] = t_m) \wedge (T[m] = t_k) \Rightarrow I(k+1)$

3.1.1 Code

```
1   k=0;    // I(0)
2   while(k != n)
3   {
4       m = min(T,k,n);
5       permuter(T[k], T[m]); // I(k+1)
6       k++;                  // I(k)
7   } // I(k) & (k=n)
```

3.1.2 Complexité

On suppose que les opérations des lignes 1,5,6 sont en temps constants. La comparaison de la ligne 2 est aussi supposée en temps constant.
Il reste à évaluer le temps d'éxecution de la ligne 4. On peut supposer que la recherche du minimum est en temps linéaire. Le temps de calcul du recherche du

min sur un tableau $T[k...n-1]$ s'écrit :

$$a(n-k)+b \leq T_{min}(n-k) \leq a'(n-k)+b'$$

Le corps de la boucle *while* est éxecutée n fois pour $k = 0, 1, 2, ..., n-1$.

$$a\sum_{i=0}^{n-1} i + nb + nc \leq T_{while} \leq a'\sum_{i=0}^{n-1} i + nb' + nc$$

$$a\frac{n(n-1)}{2} + n(b+c) \leq T_{while} \leq a'\frac{n(n-1)}{2} + n(b'+c)$$

$$\alpha n^2 + \beta n \leq T_{while} \leq \alpha' n^2 + \beta' n$$

En résumant les constantes d'initialisation par γ, on peut évaluer la complexité du SLOWSORT par :

$$\alpha n^2 + \beta n + \gamma \leq T_{ss}(n) \leq \alpha' n^2 + \beta' n + \gamma'$$

Pour des grandes valeurs ainsi $T_{ss}(n) = O(n^2)$.

3.2 Quicksort

L'algorithme de tri rapide, ou *quicksort* a été mis au point par O'HARE dans les années 50. Il repose sur une approche dite "diviser pour règner".
Cette approche consiste à séparer de facon optimale le problème en 2 (ou plus) problèmes de taille inférieure.
Le quicksort fonctionne ainsi :

```
1   procedure QS(Tableau T, entier i, entier j)
2       entier k
3       si (j-i > 1)
4           segmenter(T, i, j, k)
5           QS(T,i,k)
6           QS(T,k+1,j)
7       finsi
```

On remarque que l'essentiel du travail est effectué à la ligne 4 par la procédure *segmenter*. C'est cette procédure que nous nous attacherons à décrire. La fonction *segmenter* va choisir un élément $T[k]$ du tableau, appelé le pivot et réorganiser le tableau tel que :

$$T[i...k-1] \leq T[k] \leq T[k+1...j]$$

3.2.1 Segmenter

Cette procédure *segmenter* est bâtie sur l'invariant suivant :

Invariant : $I(k,k') \equiv\ T[i..k-1] < T[k] < T[k...k'-1]$

Initialisation : $(k = i) \wedge (k' = k+1)$

Condition d'arrêt : $k' = j$

Implications :

$I(k,k') \wedge (T[k'] > T[k]) \Rightarrow I(k,k'+1)$
$I(k,k') \wedge (T[k'] < T[k]) \wedge (T[k] = t_{k'}) \wedge (T[k+1] = t_k) \wedge (T[k'] = t_{k+1}) \Rightarrow$
$I(k+1,k'+1)$

3.2.2 Code Source

```
1   void segmenter(T,i,j,k)
2   {
3       int k';
4       k = i;
5       k'= k+1;
6
7       while(k' != j)
8           if (T[k'] < T[k])
9           else {
10              permuter(T[k'],T[k+1]);
11              permuter(T[k+1],T[k]);
12              k++;
13          } k'++;
14  }
```

3.2.3 Complexité du Quicksort

On remarque aisèment que la complexité du corps de quicksort dépend de la complexité de *segmenter*.

Complexité de segmenter

$segmenter(T,i,j,k)$ est fonction linéaire de $j-i$: $\theta(j-i)$.

$$\alpha(j-i) + \beta \leq T_{seg} \leq \alpha'(j-i) + \beta'$$

Complexité de quicksort

Il y a 2 cas de figures :

$\underline{j - i > 1 :}$

$$T_{QS}(i,j) \leq A_1 + A_2 + T_{seg}(j-i) + T_{QS}(i,k) + T_{QS}(k+1,j)$$

$$T_{QS}(i,j) \leq A_1 + A_2 + \alpha'(j-i) + \beta + T_{QS}(i,k) + T_{QS}(k+1,j)$$

$\underline{j - i \leq 1 :}$

$$T_{QS}(i,j) = A_1 + A_2$$

On a une équation de récurrence d'un majorant du temps de calcul de $QS(T,i,j)$:
• $j - i \leq 1 \;\; T_{QS}(j-i) = B_0$
• $j - i > 1 \;\; T_{QS}(j-i) \leq T_{QS}(k-i) + T_{QS}(j-k) + A(j-i) + B$

Nous sommes en présence de 2 cas limites :

Hypothése 1 : À chaque segmentation $T[i...j]$ est séparé en 2 tableaux de taille identiques : $T[i...k-1]$ et $T[k...j-1]$ tels que $k - i = j - k$. Dans ce cas, puisque les 2 tableaux sont de taille identique, on peut affirmer que la taille du problème est $n = 2^p$

Hypothése 2 : À chaque segmentation $T[i...j]$ est séparé en 2 tableaux de taille quelconque :

Traitons chacune des hypothèses :

$\underline{\text{Hypothèse numéro 1 :}}$

$$\bullet \; j - i = 1 = 2^0 \;\; \Rightarrow T_{QS}(2^0) = B_0$$

$$\bullet \; j - i > 2^{p>0} \;\; \Rightarrow T_{QS}(2^p) \geq T_{QS}(2^{p-1}) + T_{QS}(2^{p-1}) + (A * 2^p + B)$$

Essayons pour $n = 2^0, 2^1, 2^2, 2^3...$:

$$\begin{aligned}
n = 2^0 &\Rightarrow T(1) = B_0 \\
n = 2^1 &\Rightarrow T(2^1) \geq T(2^0) + T(2^0) + (A * 2^1 + B) \\
T(2^1) &= 2T(2^0) + (A2^1 + B) = 2B_0 + 2A + B \\
n = 2^2 &\Rightarrow T(2^2) \geq 2T(2^1) + (A * 2^2 + B) \\
T(2^2) &= 2^2 B_0 + 2^2 B + 2^3 A
\end{aligned}$$

D'une manière générale, la complexité du quicksort pour un problème de taille $n = 2^p$ est :

$$\begin{aligned} T(2^p) &= \alpha p * 2^p + \beta 2^p + \gamma \\ T(n = 2^p) &= \alpha log_2(n) * n + \beta 2^p + \gamma \end{aligned} \tag{3.1}$$

Donc, sous l'hypothèse d'un tableau séparé en 2 parties égales la complexité du quicksort est : $T_{QS}(n = 2^p) = O(n * log_2(n))$.

Considérons le cas où le tableau est déjà trié par ordre croissant :

$$\begin{aligned} T_{QS}(1) &= A(\text{constante}) \\ T_{QS}(n > 1) &= B + T_{seg}(n) + T_{QS}(1) + T_{QS}(n-1) \\ T_{QS}(n > 1) &= B + (\alpha n + \beta) + A + T_{QS}(n-1) \\ T_{QS}(n > 1) &= \alpha n + C + T_{QS}(n-1) \end{aligned}$$

Si on dévellope :

$$\begin{aligned} T_{QS}(n) &= \alpha \sum_{i=2}^{n} i + (n-1)C + A \\ T_{QS}(n) &= \alpha \frac{n(n+1)}{2} + (n-1)C + (A - \alpha) \\ T_{QS}(n) &= \alpha \frac{n^2}{2} \frac{\alpha n + 2(n-1)}{2} + (A - \alpha - C) \end{aligned}$$

La complexité d'un quicksort pour un tableau déjà trié est donc en $O(n^2)$, ce qui est beaucoup moins bon que la complexité originale en $O(n * log(n))$ pour un tableau quelconque.

3.2.4 Amélioration du quicksort

Nous avons vu que la complexité du quicksort dans le cas général est en $O(n * log(n))$, tandis que celle d'un tableau déja trié est en $O(n^2)$. La raison de cette complexité en $O(n^2)$ (identique à un slowsort pour le même tableau) est que la fonction *segmenter* coupe le tableau initial de taille n en 2 tableaux, l'un de taille 1, l'autre de taille $n - 1$.
Il faut donc améliorer le quicksort pour effectuer une coupe plus "égale". En effet, si le tableau initiale est coupé en 2 tableaux de taille $\frac{n}{2}$ la complexité du quicksort baisse à $O(n * log(n))$.

Voici une amélioration possible, qui consiste à échanger le pivot ($T[i]$) par un autre élement du tableau, tiré au hasard.

```
1   procedure QS(Tableau T, entier i, entier j)
2       entier k
3       si (j-i > 1)
4           segmenter(T, i, j, k)
5           a = tirage_aleatoire(T,i,j)
6           permuter(T[i], T[a])
7           QS(T,i,k)
8           QS(T,k+1,j)
9       finsi
```

On peut aussi remplacer le tirage aléatoire par un tirage plus recherché, en prenant par exemple, la valeur médiane. Dans tous les cas, ce tirage doit être en $\theta(n * log(n))$ pour conserver la complexité du quicksort.

3.3 Complexité minimum d'un algorithme de tri

Nous avons vu que des algorithmes intuitifs (donc naïfs), tels le SLOWSORT ou le BUBBLE SORT avaient des complexités de l'ordre de n^2. Des algorithmes améliorés, comme le QUICKSORT ou le HEAPSORT ont des complexités inférieures, en $n * log(n)$.On est en droit de se demander si ces algorithmes sont les plus optimaux, c'est à dire s'il existe un algorithme de tri en $O(f(n))$ avec $f(n) < n * log(n)$ à partir d'un certain rang. La réponse est non.

3.3.1 Démonstration

Nous allons tacher de démontrer que tout algorithme de tri par comparaison est optimal si sa complexité est en $O(n * log(n))$. Par conséquent, il faut démontrer qu'en présence des ces hypothès es, il faut au minimum $n * log(n)$ opérations (comparaisons) pour trier un ensemble.
Ce sont ici les hypothèses qui sont importantes : en effet nous avons basé tous les algorithmes de tri sur le fait que l'on possède sur l'ensemble E, une relation d'ordre totale. On utilise le plus souvent "$\leq$" ou "$\geq$" sur $\mathbb{N}$ ou $\mathbb{R}$.
Cette relation de comparaison est utilisé pour trier un ensemble $a_0, \cdots, a_i, \cdots a_n$ d'éléments. Au delà des valeurs de chaque a_i, ce sont les valeurs des propositions logiques booléennes $a_i \leq a_j$ qui sont utilisées pour déterminer l'ordre de l'ensemble.
Ainsi on peut modéliser n'importe quel algorithme de tri par comparaison sous la forme d'un arbre décisionnel. Un arbre décisionnel se presente sous la forme d'un arbre binaire dont chaque noeud représente une question utilisant la relation d'ordre établie (*est-ce que que a_i est inférieur à a_j*), le fils droit d'un noeud représente l'instruction exécutée si la réponse "oui" à cette question tandis que

le fils gauche est la réponse négative. Chaque feuille (les noeuds sans aucun fils) de l'arbre représente une permutation de l'ensemble de départ, correspondant à l'ensemble trié.

La condition pour qu'un arbre de décision correspondent bien à un algorithme de tri porte sur le nombre de feuille. En effet, l'algorithme doit être capable de produire toutes les permutations possible de l'ensemble de départ. Il y a $n!$ permutations d'un ensemble de cardinal de n, donc il faut que l'arbre ait $n!$ feuilles. De plus, il faut que toutes ces feuilles soit accessibles par la racine.
La longueur du plus long chemin dans l'arbre correspond au pire cas de l'algorithme.

Il suffit de déterminer la longueur du plus long chemin de l'arbre de décision pour déterminer le nombre de comparaisons d'un algorithme, donc sa complexité. Cette longueur, que l'on appelle hauteur, est noté h.
Dans un arbre de hauteur h, on a au maximum 2^h feuilles, nous pouvons donc encadrer l le nombre de feuilles ainsi :

$$n! \ \leq \ l \ \leq \ 2^h$$

Ce qui implique $h \geq log_2(n!)$. En utilisant l'approximation de STIRLING ($log(n!) = \theta(n * log(n))$), on obtient le résultat suivant ;

$$h = \Omega(nlog(n))$$

3.3.2 Conclusion

Ainsi nous avons démontré que n'importe quel algorithme de tri par comparaison peut-être modélisé sous la forme d'un arbre de décision, et par conséquent qu'il lui faut au minimum $\Omega(nlog(n))$ comparaisons dans le pire des cas. Un tel algorithme est qualifié d'optimal. Le QUICKSORT étudié n'est pas optimal car il nécessite dans le pire des cas $\Omega(n^2)$, cas où l'ensemble est déjà trié.
Néanmoins ce résultat ne représente pas une barrière pour créer des algorithmes de tri performants, car on peut réaliser des algorithmes de tri se basant sur autre chose que des comparaisons. Ainsi le RADIX SORT est en temps linéaire.

Chapitre 4

Approche diviser pour regnér

4.1 Diviser pour régner

Exemple : Le quicksort.

Typiquement l'approche *diviser pour régner* résoud un problème de taille n de la manière suivante :

1. traitement en $\theta(n)$: séparation du problème de taille n en p problèmes de tailles n_i,
2. Traitement (séquentiel ou simultann'e) des chacun des p problèmes de taille n_i de la même manière.

Généralement on utilise une séparation en 2 sous-problèmes. Lorsque $p = 2$ et $n_i = n/2$, la complexité de l'algorithme est en $\theta(n * log(n))$.

4.1.1 Tri Fusion

Le tri dit *Tri fusion* fonctionne ainsi :

1. *split* : séparation de l'ensemble E en 2 : $\theta(n)$
2. Tri fusion de E_1. $T_f(\frac{n}{2})$
3. Tri fusion de E_2. $T_f(\frac{n}{2})$
4. Fusion dans E des 2 ensembles triés E_1 et E_2 : $\theta(n)$

On prend :

$$T_F(1) = A(\text{constante})$$

$$T_F(n > 1) = \theta(n) + 2T_F(\frac{n}{2})$$

Le tri fusion est donc bien en $\theta(n * log(n))$. Le problème est que ce tri va coûter très cher en mémoire.

4.2 Limites de l'approche diviser pour règner

L'approche diviser pour régner a ses limites : c'est le recouvrement de sous problèmes.
Si l'on prend $T(n)$ le temps de calcul du problème de taille n, on a :

$$T(n) = T(f(n)) + \sum_i T(n_i)$$

Si $\sum_i n_i \leq n$ l'algorithme est efficace, sinon le temps de calcul devient exponentielle ($\Omega(2^n)$). Il faudra alors utiliser la programmation dynamique

Chapitre 5

Complexité des algorithmes

Dans cette section, nous nous attacherons à évaluer la complexité des algorithmes.
Evaluer la complexité d'un algorithme consiste à déterminer la forme du temps de calcul asymptotique. En d'autres termes il s'agit de déterminer quelle sera, dans l'expression du temps de calcul d'un algorithme, le terme $A_i = f(n)$ tels que tous les autres termes $A_{j \neq i}$ soit négligeable devant A_i pour $n = +\infty$.
Cette recherche de la complexité d'un algorithme se fait dans le pire des cas, c'est à dire que l'arrangement des données du problème, bien qu'aléatoire, pousse l'algorithme dans ses 'limites'.

5.1 Expression du temps de calcul

Chercher l'expression du temps de calcul d'un algorithme consiste à évaluer une fonction $f(n)$ donnant le temps de calcul d'un algorithme en fonction de la taille n du problème et de constantes C_i.
La taille du problème, noté n, peut très bien le cardinal de l'ensemble à trier pour un algorithme de tri. Il peut aussi s'agir d'un couple, ou d'une liste de variables, par exemple lors des problèmes traitant de graphes, on exprime f en fonction de (V, E), où V et E sont le nombre de *vertices* (arcs) et E le nombre de places dans ce graphe.

Pour déterminer cette expression, on travaille habituellement sur le code source du programme, ou sur une version en pseudo-langage détaillée. On prends soin de ne pas prendre en compte les fonctions d'entrée/sorties (en C : PRINTF et SCANF).
On prends chaque ligne de ce programme et on lui affecte une constante de coût c_i. Ce coût représente le temps d'éxecution de l'instruction à la ligne i, et dépend des caractèristiques de l'ordinateur qui éxecute l'algorithme. On essaye d'évaluer

également pour chaque ligne, le nombre de fois que cette ligne va être éxecutée, en fonction de n, et que l'on va noter $g_i(n)$.
Ainsi on obtient l'expression du temps de calcul en réalisant la somme suivante :

$$f(n) = \sum_{i=1}^{p} c_i * g_i(n)$$

L'étape suivante consiste à developper les fonctions $g_i(n)$ (lorsque celles-ci peuvent être developpés algébriquement, ou sommées lorsqu'elles sont sous la forme de série numériques), puis à ordonner selon les puissances de n décroissantes :

$$f(n) = C_0 * n^k + C_1 * n^{k-1} + \cdots + C_{k-1} * n + C_k$$

Lorsque le devéloppement présente des termes qui ne sont pas sous la forme n^k, on essaye de les encadrer, ou des les approximer, en tenant compte que l'on travaille sur des valeurs de n assez grandes. Par exemple $n^2 > n * log(n) > n$, ou $n! > n^k \ \forall k$.

Ce que l'on appelle complexité de l'algorithme, se note sous la forme $\theta(F(n))$. Avec $F(n)$ tel que, $(\forall n > N \ fix\ ), (\exists \alpha \in \mathbb{R})\ t.q.\ F(n) > \alpha f(n)$.
Concrètement, il s'agit du premier terme du devéloppement de $f(n)$ débarassé de sa constante.

Exemples
- $f(n) = A.n^2 + B.n + C.n$, alors l'algorithme est en $\theta(n^2)$ donc également en $\theta(5n^2)$.
- $f(n) = 50.n^5 + n!$, alors $\theta(n!)$.

En conclusion on peut dire que le temps de calcul d'un programme ne dépend pas uniquement de la complexité de l'algorithme sur lequel il se base. D'autres facteurs interviennent, notamment les constantes que l'on néglige, et les caractèristiques de l'ordinateur. Néanmoins la complexité permet de se faire une idée de l'évolution du temps de calcul, indépendament de l'ordinateur utilisé, lorsque l'on multiplie la taille du problème, par 10 par exemple.
Ainsi la complexité, permet en quelque sorte de juger de la performance des algorithmes lorsque l'on s'attaque à des problèmes de grande taille.

5.2 Notations

En plus de la notation $\theta(f(n))$, on utilise également deux autres notations : la notation O (grand "o"), et la notation Ω.

La notation $O(F(n))$ désigne un majorant du temps de calcul.
De même, la notation $\Omega(F(n))$ désigne un minorant du temps de calcul.

En appellant $F_i(n)$ l'ensemble des fonctions tels que l'algorithme soit en $O(f_i \in F_i)$ et $G_i(n)$ la famille de fonction tels que l'algorithme soit en $\Omega(g_i \in G_i)$, on alors $F_i \cap G_i = \phi_i$ avec ϕ_i la famille de fonctions de n tels que la complexité de la l'algorithme soit en $\theta(\varphi_i \in \phi_i)$.

Exemple : On peut déduire d'un algorithme qui est en $O(3n^2)$, $O(0.5n^3)$, $\Omega(10n^2)$ et $\Omega(n)$, qu'il sera en $\theta(n^2)$.

5.3 Calculs courants dans le calcul des complexités

Certains calculs courants peuvent apparaître lors de l'évaluation du temps de calcul d'un programme. Je n'ai pas la prétention de donner toutes les astuces permettant de donner la complexité d'un programme très facilement, car cela se fait au feeling avec l'habitude.
Néanmoins, cerains résultats importants méritent d'être retenus :

–
$$\sum_{i=1}^{n} i \;=\; \frac{n(n+1)}{2} \;=\; \theta(n^2)$$

–
$$\forall x < 1 \sum_{k=0}^{+\infty} x^k \;=\; \frac{1}{1-x}$$

–
$$\sum_{i=1}^{n} 1/i \;=\; ln(n) + O(1) \;=\; \theta(ln(n))$$

5.4 Un peu de théorie de la complexité

Nota Bene : *Partie non traitée en cours*

Au delà des algorithmes, que l'on peut classer par complexité, il y a les problèmes, auxquels les algorithmes répondent. Il existent plusieurs classes de problème.

5.4.1 Solution ou pas

Il faut d'abord différencier la solution d'un problème des solutions d'un problème. En effet, pour un problème P auquel il y a une ou plusieurs solutions S. Chercher S est différent de vérifier si parmi toutes les propositions logiques s, $s \in S$.
Prenons par exemple le cas de la factorisation d'un entier N. Déterminer l'ensemble des diviseurs premiers de N est différent de vérifier si une suite fini de nombres premiers correspond bien à la factorisation de N.

5.4.2 Classes de problèmes

La classe P : C'est la classe des problèmes pour lesquels il existent une solution que l'on peut déterminer intégralement en temps polynomial, c'est à dire qu'il existe un algorithme pour ce problème dont la complexité est en $\theta(n^k)$.

La classe E : C'est la classe des problèmes intrinsèquement exponentiel, c'est à dire que tout algorithme sera de la forme $\theta(k^n)$ avec k une constante.

La classe NP : (Abbréviation de *Non-Déterministe Polynomiaux* C'est la classe des problèmes dont on peut vérifier les solutions en temps polynomial, mais dont on ne connait pas forcèment d'algorithme permettant de trouver les solutions. Typiquement ce sont des problèmes plus difficiles que ceux de P.

5.4.3 NP-Complet

On dit qu'un problème est NP-Complet quand tous les problèmes appartenant à NP lui sont réductibles. C'est à dire si l'on trouve une solution polynomial à un problème NP-Complet, alors tous les problèmes NP-Complet peuvent être résolus en temps polynomial aussi. C'est pourquoi les problèmes dit NP-Complet sont généralement ceux qui sont les plus 'durs'.
Lorsque l'on a affaire à un problème NP-Complet, on essaye généralement de développer un algorithme qui *s'approche* de la solution.

Chapitre 6

Programmation Dynamique

Nous avons vu l'approche "*diviser pour régner*", qui consiste à séparer un problème de taille n en 2 problèmes de tailles inférieures, et ainsi de suite jusqu'a ce que l'on arrive à des problèmes de tailles 1.
Cette approche trouve sa limite lorsque l'on divise le problème en 2 mais que l'on effectue les mêmes traitements sur les 2 sous-probèmes : c'est le recouvrement
D'une manière générale, la programmation dynamique est nécessaire lorsque le traitement d'un problème de taille n dépend du traitement des problèmes de taille n_i, avec la relation suivante :

$$T_{pb}(n) = f(n) + \sum_i T_{pb}(n_i) \quad \wedge \sum_i n_i > n$$

On a alors une équation de récurrence qui conduit à une complexité non polynomiale (en $O(2^n)$). La programmation dynamique est donc nécessaire.

6.1 Cas concret : fibonnaci

La suite de FIBONACCI est donnée par la relation de récurrence :

$$U_n = \begin{cases} U_0 = 0 \\ U_1 = 1 \\ \forall n > 1 \;\; U_n = U_{n-1} + U_{n-2} \end{cases}$$

Cette fonction se programme trivialement dans tous les langages fonctionnels, y compris le C :

```
1   int fibo(int n)
2   {
3       if (n == 0) return 0;
4       if (n == 1) return 1;
5       else return fibo(n-1) + fibo(n-2);
6   }
```

Un tel programme a une complexité exponentielle en $O(2^n)$. En pratique, les temps de calcul dépasse la seconde pour calculs d'ordre $n > 40$. Ceci est due à la relaive lourdeur de la recursivité : l'ordinateur va passer beaucoup de temps à empiler et dépiler les arguments de la fonction. La programmation dynamique propose un algorithme modifié :

```
T := Tableau d'entier T[0...n]
T[0] <= 0
T[1] <= 1
pour i allant de 2 a n
    T[i] <= T[i-1] + T[i-2]
```

Cette solution est en $\theta(n)$ pour le temps processeur, mais demande une mémoire additionnelle en $O(n)$.

6.2 Exemple : Multiplication de matrices

Le but de cet algorithme est multiplier une suite de matrice réelles, de façon optimale.
Soit $A_0 A_1 \dots A_{n-1}$ n matrices compatibles pour la multiplication, c'est à dire que pour une matrice A_i (avec $1 \leq i \leq n-2$ donnée dont les dimensions sont (m_i, n_i), alors $m_i = n_{i-1}$ et $n_i = m_{i+1}$.

On admet que la multiplication de 2 matrices de dimension (m, n) et (n, p) se fait en mnp opérations.

La multiplication de matrice nétant pas commutative, on ne peut pas échanger 2 matrices dans la suite.

6.2.1 Exemple de l'importance du parenthèsage

L'ordre dans lequel on va procéder aux multiplications peut influer l'optimalité du calcul, en effet :
Soit 3 matrices A, B, C, dont les dimensions respectices sont (p, q), (q, r) et (r, s). Ces dimensions bien compatibles pour la multiplication.
Nous devons évaluer $T = A*B*C$. La multiplication de matrices étant associative, on peut procéder soit de la manière $T = A * (B * C)$ ou $T = (A * B) * C$.

Première méthode ($T = (A * B) * C$) :

$$\underbrace{\underbrace{(A_{p,q} \ * \ B_{q,r})}_{p*q*r} * C_{r,s}}_{p*r*s}$$

Le cout total de cette méthode est $T = pqr \ + \ prs \ = \ pr(q+s)$.

Deuxième méthode ($T = A * (B * C)$) :

$$\underbrace{A_{p,q} \ * \ \underbrace{(B_{q,r} \ * \ C_{r,s})}_{q*r*s}}_{p*q*s}$$

Le cout total de cette méthode est $T = qrs \ + \ pqs \ = \ qs(p+r)$.

On a un cout total pour ces 2 méthodes différent. Prenons un exemple :

$$p = 10 \ ; \ \ q = 10^4 \ ; \ \ r = 1 \ ; \ \ s = 10^4$$

Première méthode : $T = 10(10^4 + 10^4) = 2 * 10^4$
Deuxième méthode : $T = 10^8(10+1) = 11 * 10^8$
Le rapport est donc de $5,5 * 10^3$, ce qui est très significatif, d'où l'importance du parenthèsage.

6.2.2 Algorithme de calcul.

Cet algorithme va procéder en 2 temps :

1 - Déterminer le coût minimum du produit $< A_0 \dots A_{n-1} >$.
2 - Calculer $< A_0 \dots A_{n-1} >$ selon le parenthèsage optimal.

6.2.3 Coût minimum

Notation : Soit m_{ij} le nombre minimum de multiplications scalaires pour multipliers les matrices $A_i \dots A_{j-1}$.

$m_{i\ i+1} = 0 \ \forall i$.

On note $m_{ij}^{(k)}$ le cout du produit parenthés'e $(A_i \dots A_{k-1})(A_k \dots A_{j-1})$. On a $m_{ij}^{(k)} = m_{ik} + m_{kj} + P_i P_k P_j$.

Pour tout i, j tels que $j > i+1$, on note m_{ij} le coût optimal pour la multiplication des matrices $A_i \cdots A_{j-1}$. Ce coût est le plus petit m_{ij}^k.

$$m_{ij} = \overset{min}{i < k < j} \ (m_{ik} + m_{kj} + P_i P_k P_j)$$

On peut en déduire le code source de cette fonction aisèment :

```
1   int m(int i, int j, tabDimMat P)
2   {
3       if (j == i+1) return 0;
4       else return min(m(i,k,P)+m(k,j,P)+P[i]*P[j]*P[k]);
5   }
```

Cette méthode de calcul des coûts minimum a un énorme problème : il y a un recouvrement des sous problèmes : certains coûts seront calculés plusieurs fois
Pour contrer cela, nous allons stocker dans un tableau les m_{ij} dans une matrice :

$$\begin{bmatrix} M_{m0} & M_{m1} & \cdots & \cdots & M_{mn} \\ \vdots & \vdots & \cdots & M_{ii} & M_{ii+1} \\ \vdots & \ddots & M_{ii} & M_{ii+1} & M_{ii+1} \\ \vdots & \ddots & & & \\ M_{00} & \cdots & \cdots & \cdots & M_{0n} \end{bmatrix}$$

Les éléments M_{ii} de la diagonale représentent les coûts des produits $m_{ii} = 0$. La diagonale est donc nulle De même, la diagonale inferieure ($M_{i\,i+1}$) est nulle. On utilise l'algorithme suivant pour remplir la partie inférieure du tableau :

```
// Calcul des problemes de taille 1
    pour i allant de 0 a n-1
    M[i][i+1] = 0
// Calcul par taille t croissant
    pour t allant de 2 a n
    // calcul pour chaque taille t de tous les problemes de taille t
    pour i allant de 0 a (n-t)
    j = i+t
    M[i][i+t] = min( M[i][k] + M[k][i+t] +P[i]*P[i+t]*P[k] )
```

Chaque diagonale t (éléments du tableau $m[i][i+t]$) représente le coût d'un problème de taille t.
On peut maintenant établir l'invariant de l'algorithme de remplissage de la matrice des coûts :

Invariant : $I(\alpha, n, k) \equiv (m = min(M[i][n] + M[n][\alpha] + P[I] * P[n] * P[\alpha]) \wedge (k = argM[i][j])$

Initialisation : $(\alpha = i + 2) \wedge (x = i + 1) \wedge (k = i + 1)$

Arrêt : $\alpha = j$

Implications : $I(\alpha, m, k) \wedge (m' = M[i][\alpha] + M[\alpha][j] + P[i]P[\alpha]P[j]) \wedge (m' \geq m) \Rightarrow I(\alpha + 1, m, k)$
$I(\alpha, m, k) \wedge (m' = M[i][\alpha] + M[\alpha][j] + P[i]P[\alpha]P[j]) \wedge (m' < m) \Rightarrow I(\alpha + 1, m', k)$

6.2.4 Calcul optimal du produit

Nous venons de calculer les coûts optimaux pour le produits de matrices. Ces coûts sont stockés dans la partie inférieure d'une matrice M
Nous allons faire en sorte que la partie supérieure de la matrice M contient les valeurs k_{ij} tels que l'ont ait le coût minimum m_{ij}. $M[i][j] = m_{ij}$; $M[j][i] = k_{ij}$.
Le coût m_{0n} de la multiplication des $n-1$ matrices se trouve donc en $M[0][n]$.
Il nous reste donc maintenant à calculer effectivement le produits de ces $n-1$ matrices en utilisant les informations sur le parenthésage optimal contenu dans M.

Calcul du produit $A_i * \cdots * A_j$

```
1   void multopt(matrice resultat, matrice_couts M, matrice *A, int i, int j)
2   {
3       matrice P1, P2;
4
5       if (j=i+1) resultat= A[i];
6       else {
7           multopt(P1,M,A,M[j][i],j);
8           multopt(P2,M,A,i,M[j][i]);
9           mult_matrice(resultat,P1,P2);
10      }
11  }
```

L'appel initial à cette fonction est $multopt(resultat, A, M, 0, n)$.

6.3 Exemple : Recherche du plus long sous-mot commun à 2 chaines

Soit 2 mots A et B. Ces mots sont composés de caractères a_i et b_j.
$A \equiv a_0 a_1 a_2 \cdots a_{m-1}$
$B \equiv b_0 b_1 b_2 \cdots b_{m-1}$

Le problème que nous nous posons ici est de calculer le plus long sous mot M commun à 2 mots A et B.
Soit $L(x, y)$ la longueur du plus long sous-mot commun à $A(x)$ et $B(y)$ Ainsi pour tout i, j on a les propositions suivantes :

$$\begin{aligned} i > 0 \; et \; j > 0 : a_{i-1} \neq b_{j-1} &\Rightarrow L(i,j) = max(L(i-1,j), L(i,j-1)) \\ i > 0 \; et \; j > 0 : i > a_{i-1} = b_{j-1} &\Rightarrow L(i,j) = 1 + L(i-1,j-1)) \\ i = 0 &: L(0,j) = 0 \\ j = 0 &: L(i,0) = 0 \end{aligned}$$

6.3.1 Taille du problème

Le but est de calculer $L(m,n)$ longueur du problème $(A(m), B(n))$. $\forall i,j$, $L(i,j)$ sera calculé, par taille de problème croissant.
Il faut que chaque valeur $L(i,j)$ ne soit calculé qu'une seule fois. C'est pourquoi nous stockerons ces valeurs dans une matrice $M[0\cdots m][0\cdots n]$.

6.3.2 Reconstruction de la solution

```
// Calcul des pbs de taille 0
    pour i allant de 0 a m  L[i][0] = 0
    pour j allant de 0 a n  L[0][j] = 0
// Calcul de tous les probl\‘emes par taille croissant
    pour t allant de 1 a m
pour i allant de t a m
    si A[i-1]=B[t-1], alors L[i][t] = L[i-1][t-1] + 1
    sinon L[i][t] = max( L[i-1][t], L[i][t-1] )
pour j allant de t a n
    si A[t-1]=B[j-1], alors L[t][j] = L[t-1][j-1] + 1
    sinon L[t][j] = max( L[t][j-1], L[t-1][j] )
```

La complexité de cet algorithme est $\theta(m * n)$.

6.3.3 Exemple

Voici la matrice générée pour les deux chaînes :

$A =$ "bca" $\quad B =$ "$cabd$"

$$L = \left[\begin{array}{c|ccccc} \mathbf{3} & 0 & 1 & 2 & 2 & 2 \\ \mathbf{2} & 0 & 1 & 1 & 1 & 1 \\ \mathbf{1} & 0 & 0 & 0 & 1 & 1 \\ \mathbf{0} & 0 & 0 & 0 & 0 & 0 \\ \hline L & \mathbf{0} & \mathbf{1} & \mathbf{2} & \mathbf{3} & \mathbf{4} \end{array}\right]$$

6.3.4 Affichage du plus long sous mot

Appelons $M(p)$ le plus long sous-mot commun a $A(m)$ et $B(n)$. La longueur de de $M(p)$ a déjà èté calculée et se trouve à l'emplacement (m,n) de la matrice L dont la construction vient d'être détaillée.

$$M(p) = m_0 m_1 \cdots m_{p-1}$$

. En appelant $M(k)$ le k-préfixe, et $\overline{M(k)}$ le k-suffixe : $M(k) = m_0 m_1 \cdots m_{k-1}$
$\overline{M(k)} = m_p m_{p-1} \cdots m_{p-k}$

$$M(p) = M(k).\overline{M(k)}$$

Nous allons construire le programme de reconstruction de la plus longue sous chaîne commune, basé sur l'invariant suivant :

Invariant : $I(k,i,j) \equiv (\overline{M(k)}$ affiché$) \wedge (M(k) = PLSM(A(i), B(j))$reste à afficher$) \wedge (M(k).\overline{M(k)} = M(p))$

Init : $(k = L[m][n]) \wedge (i = m) \wedge (j = n)$

Arrêt : $k = 0$

Implications : $I(k,i,j) \wedge (k \neq 0) \wedge (L(i,j) = 1 + L(i-1, j-1)) \wedge (A[i-1]$affiché$) \Rightarrow I(k-1, i-1, j-1)$
$I(k,i,j) \wedge (k \neq 0) \wedge (L(i-1,j) > L(i,j-1)) \Rightarrow I(k, i-1, j)$
$I(k,i,j) \wedge (k \neq 0) \wedge (L(i-1,j) \leq L(i,j-1)) \Rightarrow I(k, i, j-1)$

Le code source de ce programme s'écrit tout simplement grâce à cet invariant :

```
1   int k = L[m][n];
2   int i = m;
3   int j = n;
4
5   while (k != 0)
6   {
7       if ( L[i][j] == L[i-1][j-1] )
8       {
9           printf("%c ", A[i-1]);
10          i--; j--; k--;
11      } else if (L[i-1][j] > L[i][j-1]) i--;
12      else j--;
13  }
```

On peut remarquer que ce programme affiche la plus longe sous-chaîne commune dans l'ordre inverse. Toutefois, ce n'est pas si grave car on sait inverser des chaines en temps linéaire.

Deuxième partie

IN311 - Programmation dynamique

Introduction au cours de programmation dynamique

Extrait de la brochure "Programme des enseignements de 3º Année" :

La programmation dynamique est une méthode d'optimisation opérant par phases (ou séquences) dont l'efficacité repose sur le principe d'optimalité de Bellman : "toute politique optimale est composée de sous-politiques optimales"
Cette unité met l'accent sur la diversité des domaines et des problèmes d'apparence très éloignés qui lorsqu'ils sont correctement modélisés relèvent de la même technique. Elle permet de mettre en pratique sur des problèmes de traitement du signal, de contrôle et d'informatique les concepts abordés dans les cours d'informatique logicielle et matérielle des deux premières années du tronc commun.
Résoudre un problème d'optimalité par la programmation dynamique est, dans ce contexte précis, une mise en oeuvre significative de la diversité des compètences attendues d'un ingénieur : modélisation du problème, proposition d'une équation de récurrence, proposition d'un algorithme efficace de résolution du problème (modélisation informatique des données, évaluation de complexité) ; étude des différentes optimisations possibles de l'algorithme ; programmation effective ; possibilités d'implantation sur architectures spécifiques déduites de l'algorithme et des contraintes de programmation.

Chapitre 7

Applications de la programmation dynamique

7.1 Un problème d'assortiment

7.1.1 Sujet

On considère m objets, de format $1, 2, \cdots, m$ (tous les objets sont de formats différents). Ces objets doivent être empaquetés avec du papier d'emballage.
Le papier d'emballage est disponible sous forme de feuilles chez le fabricant, et dans tous les formats convenables. Pour des raisons de coût de fabrication, la commande des m feuilles doit être faite parmi n formats seulement. Le coût d'une feuille croît avec le format. Un format permet d'emballer tout objet de format inférieur ou égal.
Le problème posé est celui du choix des n formats permettant d'empaqueter les m objets à moindre coût. Les données du problème sont le nombre d'objets m, le nombre de formats n et, pour tout format f, $1 \leq f \leq n$, son coût $c(f)$.

On demande un algorithme calculant en temps polynomial le coût minimum de la commande et un assortiment de formats qui permette d'obtenir ce coût minimum.

7.1.2 Résolution

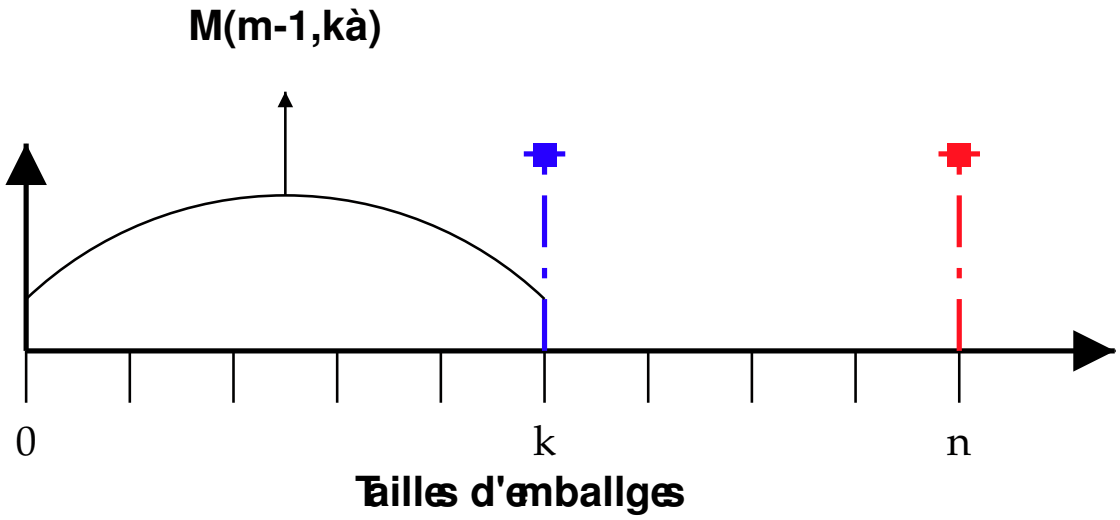
On part du principe d'optimalité de Bellman : *Une solution optimale est composée de sous-solutions elles-mêmes optimales.* On va raisonner ainsi par taille de problèmes croissants.
La taille du problème est ici m, le nombre de formats à choisir.
On note $M(m, n)$ le coût minimum pour emballer les objets de taille $1...n$ avec m formats. Ce coût implique que les formats de feuilles d'emballage sont réparties

au mieux pour emballer les objets les objets des n premières tailles.

On note $x(t) \mid \forall t \in [1, n]$, le nombre d'objets pour une taille t.
Si l'on ne dispose qu'une seule feuille de taille n pour emballer tous les objets, il est évident que celle-ci doit être de la taille du plus grand objet à emballer. D'une manière générale, la plus petite taille permettant d'emballer le plus grand objet doit être choisie. Quelque soit l'assortiment de feuilles choisies optimalement, celui-ci comporte la feuille de taille n.

7.1.3 Cas général

Nous savons dejà placé le dernier emballage, de taille n. Il nous reste donc à placer $m-1$ emballages optimalement. Pour cela on place un emballage en position k, ainsi on en revient a un problème de taille $m-1$, sur k tailles d'objets.



En notant C_t le coût pour emballer les n emballages :

$$C_t = M(m-1,k) + \sum_{t=k+1}^{n} (c(n) * x(t))$$

De là, on tire l'équation de récurrence :

$$\forall 1 < m \leq n \quad M(m,n) = \min^{m-1 \leq k \leq n-1} \left[M(m-1,k) + \sum_{t=k+1}^{n} c(n) * x(t) \right]$$

La borne inférieure du minimum vient du fait qu'il ne faut pas qu'il y ait plus de formats que de tailles. Si $k < m-1$, le problème ne serait pas optimal.

On doit donc calculer M pour les problèmes de taille 1 :

$$M(1,p) = \sum_{t=1}^{p} c(p) * x(t) \quad \forall p \in [1,n]$$

7.2 Compression d'image

7.2.1 Sujet

On considère une image en m niveaux de gris. Chaque pixel de l'image a un niveau de gris compris entre 0 et $m-1$. On veut "comprimer" cette image en restreignant à n le nombre de niveaux de gris de l'image. Ces n niveaux sont à choisir parmi les m valeurs de l'image d'origine.
Dans cette "compression", le niveau de gris de chaque pixel est remplacé par le niveau de gris le plus proche au sein des n niveaux choisis. La question posée est le choix de n niveaux de gris qui permettront cette compression avec le minimum d'erreur.
L'erreur est ainsi définie :

- pour chaque pixel l'erreur est la distance entre sa valeur de niveau de gris et le niveau de gris le plus proche parmi les n niveaux choisis.
- l'erreur de compression est la somme des erreurs des pixels de l'image.

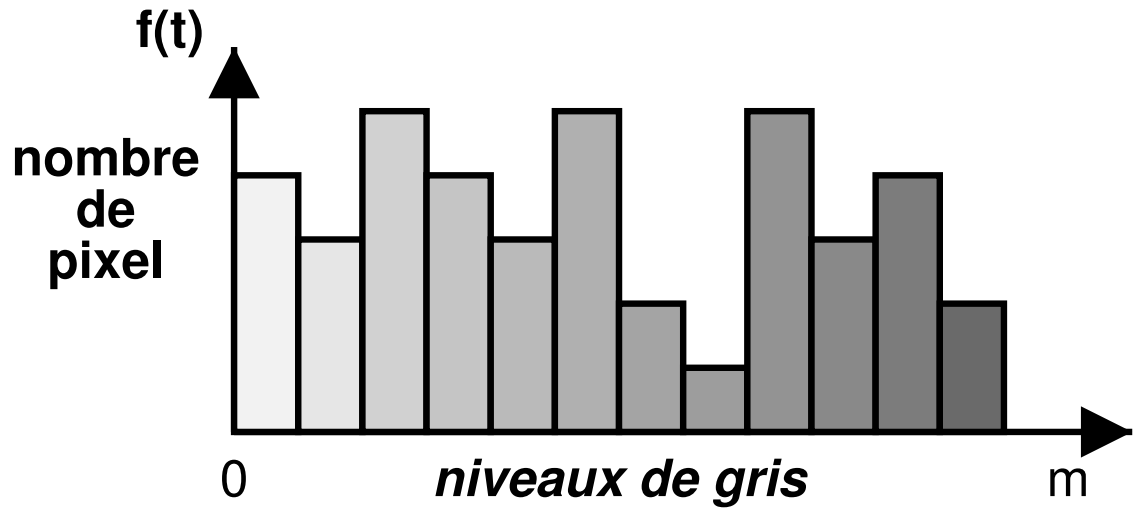
Propose un algorithme calculant en temps polynomial l'erreur de compression minimum et un choix de n niveaux de gris permettant d'atteindre cette erreur minimum.

7.2.2 Histogramme

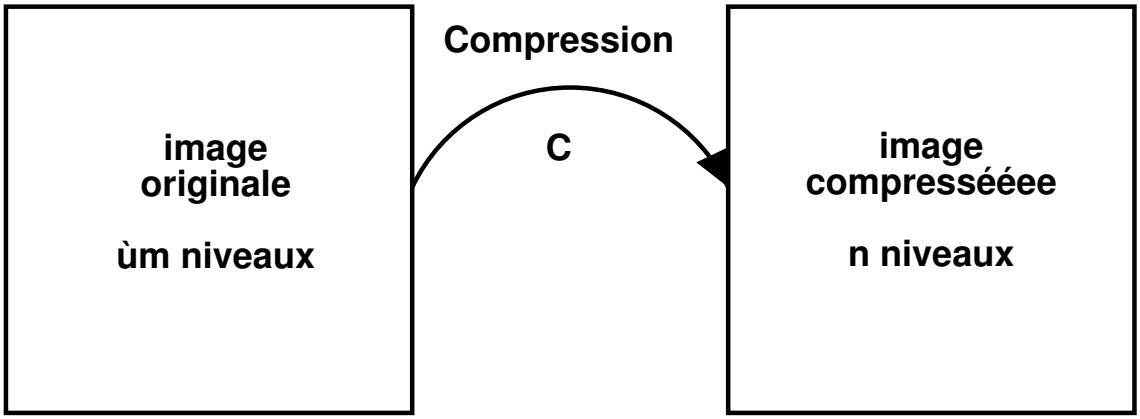
On dispose de l'histogramme de l'image. Cette histogramme représente la fonction discrète $\forall t \in [0, m-1] \longrightarrow f(t)$, où $f(t)$ représente le nombre de pixels de couleur t.

7.2.3 Résolution du problème

Le problème est de choisir au mieux les n niveaux de gris pour représenter l'image par les m niveaux déjà présent dans l'image. Pour cela il faut minimiser l'erreur.



On peut représenter la compression par une application entre les niveaux de gris de l'image originale et les niveaux de gris de l'image compressée.



En notant cette application $C(t))$ $\forall t \in [0, m-1]$, on peut exprimer l'erreur e_p commise pour chaque pixel p dont le niveau de gris est g_p :

$$e_p = |g_p - C(g_p)|$$

Ainsi la qualité de la compression s'exprime grâce à l'erreur totale E :

$$E = \sum_{p \in image} e_p = \sum_{p \in image} |g_p - C(g_p)|$$

Pour améliorer la qualité il faut minimiser l'erreur totale E.

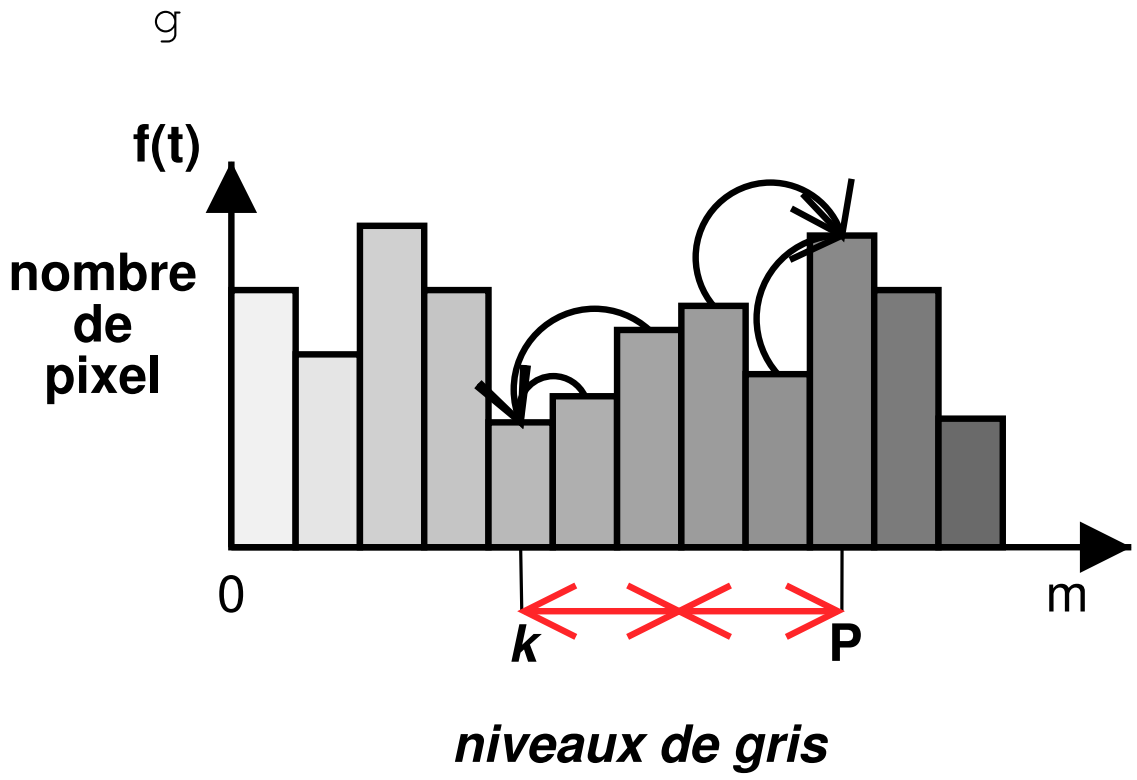
7.2.4 Equation de récurrence

On note l'erreur $X(N, M)$ l'erreur commise en représentant au mieux une compression de M niveaux de gris sur N. Ainsi $X(n, m) = E_{minimal}$.

Calculons $E(0, \gamma)$, c'est le cas où l'on n'utilise aucun niveau de gris pour représenter les γ premier niveaux de gris de l'image. Ainsi pour les niveaux de gris $0 \cdots \gamma - 1$ de l'image, il y a une erreur de commise. En admettant qu'ils soient tous representés par le niveau γ :

$$E(0, \gamma) = \sum_{g=0}^{\gamma-1} f(g) * |g - \gamma|$$

On se place dans le cas où l'on a déjà optimisé le choix des niveaux de gris pour minimiser la compression des $m - P$ derniers niveaux de gris de l'image. Nous allons calculer $X(t, P)$, l'erreur commise en compressant P niveaux de gris en t.
On admet que le niveau de gris k est choisi pour être un représentant. Les niveaux de gris entre k et P vont donc être representé soit par k, soit par P, selon leur position.



L'erreur $X(t, P)$ sera égale a l'erreur commise pour la représentation des pixels dont le niveau se situe entre k et P plus l'erreur $X(t-1, k)$ commise pour representer les k premiers niveaux de gris.

$$X(t, P) = e(k, P) + X(t-1, k)$$

Avec $e(k, P)$ l'erreur commise en représentant les pixels dont la couleur est dans la partie inférieur de l'intervalle $[k, P]$ par k et les pixels dont la couleur se trouve dans la partie supérieure par P.

$$e(k, P) = \sum_{g=k}^{\frac{k+P}{2}} (f(g) * |g-k|) + \sum_{g=\frac{k+P}{2}}^{P} (f(g) * |g-k|)$$

De là, on peut tirer l'équation de réccurence :

$$X(n, \gamma) = \overset{k\in[n-1,\gamma-1]}{min} [e(k, \gamma) + X(n-1, k)]$$

L'intervalle $[n-1, \gamma-1]$ s'explique par le fait que la compression est une application : on ne peut pas compresser un pixel en lui donnant 2 représentant, il faur donc $k \geq n$.

7.2.5 Cas particulier

L'équation de réccurence ne permet pas explicitement de placer le dernier représentant, à partir duquel s'effectue tout l'algorithme. Ce niveau se calcule ainsi :

$$E = X(n, m) = \overset{k\in[n-1,m-1]}{min} \left[X(n-1, k) + \sum_{g=k}^{m-1} f(g) * |g-k| \right]$$

7.2.6 Programmation

L'équation de récurrence correspond au calcul d'une erreur minimale. Pour chaque calcul de minima, on définit l'argument du min. comme étant la valeur de paramètre permettant d'obtenir ce coût minimum.
Ici le paramètre du minimum est k, chaque valeur de k que l'on obtiendra correspondra a un niveau de gris compressé

7.3 Justification de Parapgraphes

7.3.1 Sujet

On considère n mots de longueurs $l_0, \ldots l_{n-1}$, que l'on souhaire imprimer en une suite de lignes ayant chacune C caractères, sauf la dernière qui peut en

compter moins (le texte que vous lisez actuellement est un exemple du résultat escompté).
Le nombre c de caractères d'une ligne est la somme des longueurs des mots qui la composent, plus les blancs qui séparent les mots (un blanc entre tout couple de mots). Pour une telle ligne, il faut ajouter $C - c$ blancs supplémentaires entre les mots pour qu'elle ait exactement C caractères. Le coût de cet ajout est $(C - c)^3$.
Le coût de la justification du paragraphe est la somme de ces coûts sur toutes les lignes, sauf la dernière.
Proposer un algorithme calculant cette justification de paragraphes en un temps polynomial.

7.3.2 Algorithme

On a N mots de longueurs respectives $l_0, l_1, \cdots, l_{n-1}$.
On se place dans le cas d'une ligne (sauf la dernière, qui est un cas particulier). Cette ligne composée des mots $m_i m_{i+1} \cdots m_j$, a pour longueur L :

$$L = \sum_{k=i}^{j} (l_k + (j - i - 1))$$

Cette ligne amène un coût supplémentaire $(C - L)^3$. On va chercher à minimiser le coût total de formattage du document, c'est à dire minimiser $(C - L)^3$ pour chacune des lignes.

On note $e(k)$ le coût minimum de formattage des k premiers mots. Sur ces k premiers mots, on s'autorise à mettre **au plus** $k - 1$ retours chariots, soit k lignes. Pour la dernière ligne, on va chercher à minimiser le coût :

$$e(N) = min(e(k))$$

Pour tout k tel que la longueur des mots $k \cdots N$ soit inférieur à C (longueur maximale d'une ligne), soit

$$\sum_{i=k}^{N} (l_i + (N - k - 1)) \leq C$$

Pour la première ligne, le coût $e(1)$ est :

$$e(1) = (C - l_i)^3$$

d'une manière générale, on a :

$$e(m) = \overset{k}{min} \left[e(k-1) + \left(C - \sum_{i=k}^{m} l_i + (m - k - 1) \right)^3 \right]$$

Avec k tel que $\sum(l_i + (m - k - 1)) \leq C$

Troisième partie

IN302 - Graphes et algorithmes

Chapitre 8

Notions de base

8.1 Première définition

Soit E un ensemble fini. On appelle *graphe* sur E un couple $G = (E,\ \Gamma)$ où Γ est une application de $E \longrightarrow \mathcal{P}(E)$.

Exemple i:

Soit un graphe G tel que $E = \{1, 2, 3, 4\}$. Γ est définie par :
- $\Gamma(1) = 1, 2, 4$
- $\Gamma(2) = 3, 1$
- $\Gamma(3) = 4$
- $\Gamma(4) = \emptyset$

Cette définition de Γ est plutôt difficile à imaginer. La représentation sous formes d'arcs fléchés de la figure 8.1 est plus pratique.

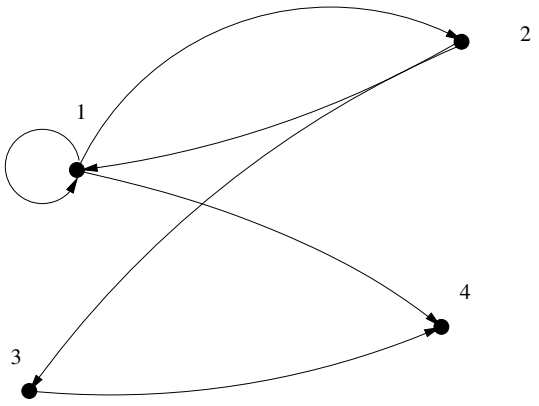


FIG. 8.1 – *Graphe de l'exemple 1*

• Tout élément de E est appelé sommet.
• Tout couple ordonné $(x, y) \in E \times E$ tel que $y \in \Gamma(x)$ est appelé arc du graphe.
• Si (x, y) est un arc, on dit que y est un successeur de x.
• Si (x, y) est un arc, on dit que x est un prédecesseur de y.

Le graphe G de l'exemple i comporte 4 sommets et 6 arcs.

On désigne par $\vec{\Gamma}$ l'ensemble des arcs. L'ensemble des arcs est une partie du produit cartèsien $E \times E$: $\vec{\Gamma} = \mathcal{P}(E \times E)$. On peut définir formellement $\vec{\Gamma}$ par :

$$\vec{\Gamma} = \{(x, y) \ \in \ E \times E \ / \ y \ \in \ \Gamma(x)\}$$

Remarque: On peut représenter G par (E, Γ) ou par $(E, \vec{\Gamma})$. On appellera également graphe *le couple $(E, \vec{\Gamma})$.*

8.2 Représentation en mémoire

Le problème qui se pose est celui de la représentation d'un graphe dans une mémoire informatique.

8.2.1 Représentation des ensembles finis

On peut voir que l'ensemble des arcs peut être représenté par l'ensemble des sucesseurs de chacun des sommets du graphe. Representer un graphe revient donc à représenter un ensemble fini. Soit $E = 1, \cdots, n$ et tel que $|E| = n$. Comment représenter un sous-ensemble X de E ?

Liste-tableau

Cette méthode de représentation d'un sous-ensemble consiste à aligner les éléments du sous ensemble à représenter dans un tableau. Ce tableau a une taille fixe égale au cardinal de E (cardinal de la plus grande partie de E). Pour savoir combien d'éléments comporte le sous-ensemble, on utilisera un caractère spécial que l'on définira comme étant la fin de la liste. On pourra également indiquer le nombre d'éléments du sous ensemble en début de liste.

Liste chaînée

On utilise une liste chaînée contenant les éléments du sous-ensemble X.

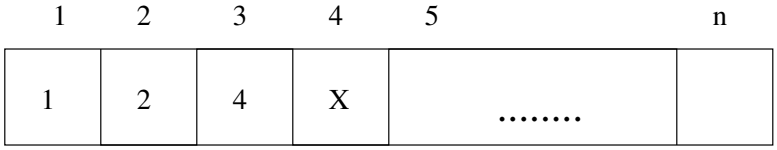
Tableau de booléens

On utilise un tableau de taille n (cardinal de E) et tel que la case i du tableau vaut VRAI (1) si l'élément i appartient au sous ensemble X. Si l'élément i de E n'appartient pas à X, alors la cause vaut FAUX (0).

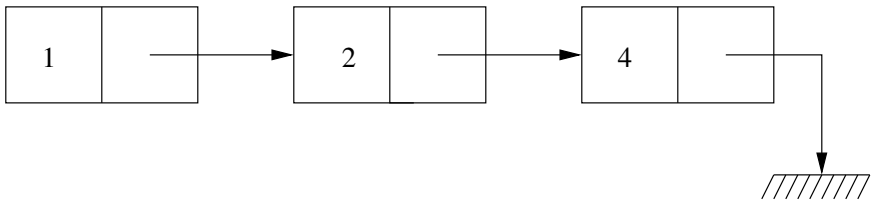
Exemple i:

Prenons $n = 9$ et $X = \{1, 2, 4\}$.

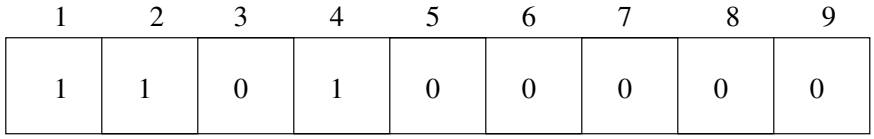
– Liste-tableau :



– Liste chaînée :



– Tableau de booléens :



8.2.2 Opérations sur un ensemble

Selon la représentation mémoire que l'on prend, les complexité des opérations élémentaires sur l'ensemble vont être changée. Le tableau suivant présente les différentes opérations élémentaires sur un ensemble ainsi que leur complexité selon le type de représentation mémoire choisie.

Opération		Liste chaînée	Tableau de booléens
Initialisation	$X = \emptyset$	$O(1)$	$O(n)$
Test d'appartenance	$x \in X$?	$O(n)$	$O(1)$
Test non-vide	$\exists ? x \in X$	$O(1)$	$O(n)$ [1]
Ajout d'un élément	$X = X \cup \{x\}$	$O(n)/O(1)$ [2]	$O(1)$

Le choix d'une représentation mémoire adaptée est un facteur à prendre en compte lors de la conception d'algorithmes.

8.2.3 Représentation d'un graphe

La représentation liée à $(E, \vec{\Gamma})$ est celle liée a l'intégralité du graphe. Plusieurs représentations sont possibles.

Deux tableaux

Cette représentation consiste à choisir deux tableaux I et T de taille $|\vec{\Gamma}|$, tels que $I(j)$ est le sommet initial de l'arc numéro j et $T(j)$ est le sommet terminal de l'arc numéro j.

Tableaux de listes de chaînées

Cela consiste à prendre un tableau M contenant $n = |E|$ listes chaînées et tel que $M(i)$ est la liste chaînée des arcs partant du sommet numéro i.

Matrice booléenne

C'est le type de données formé à partir des tableaux de booléens (8.2.1). La matrice M est telle que $M_{ij} = 1 \Leftrightarrow \exists$ arc entre i et j.

8.2.4 Complexité

Nous allons voir avec un exemple concret d'algorithme que le choix de la représentation peut influer sur la complexité des programmes et donc leur efficacité.

Algorithme de recherche des successeurs d'une partie

L'algorithme suivant a pour but de rechercher les successeurs d'une partie X d'un ensemble E.
Données : $G = (E, \vec{\Gamma})$, $X \subset E$.
Résultat : $Y =$ ensemble des sommets de G qui sont successeurs des sommets de X. On notera $Y = \alpha(X)$.

$$Y = \left\{ y \in E \ , \ \exists x \in X \ , \ (x, y) \in \vec{\Gamma} \right\}$$

1: $Y \leftarrow \emptyset$
2: **for all** $(x, y) \in \vec{\Gamma}$ **do**
3: **if** $x \in X$ **then**

4: $Y = Y \cup \{y\}$

5: **end if**

6: **end for**

Selon le choix du type de donnés pour X et Y, la complexité de l'algorithme précédent peut varier. Prenons par exemple Y sous forme d'un tableau de booléens et X sous forme de liste chaînée.
On note $n = |E|$ et $m = |\vec{\Gamma}|$.

L'instruction d'initialisation de Y à la ligne 1 est en $O(n)$. La boucle de la ligne 2 à 6 est executée m fois et son évaluation est en temps constant $O(1)$. Le test d'appartenance ligne 3 est en $O(n)$ car X est sous forme d'une liste chaînée, de plus il est executée m fois. L'ajout d'un élément à un tableau de booléens, ligne 4, est en temps constant. on peut conclure que la forme de la complexité de cet algorithme est en $O(mn)$.
Choisissons maintenant X sous forme d'un tableau de booléens. L'instruction de la ligne 3 devient en temps constant $O(1)$; le corps de boucle de la ligne 2-6 est en temps constant et est executé m fois. L'algorithme est alors $O(m+n)$ ce qui est bien meilleur que la complexité précédente qui est quadratique pour $m \approx n$.

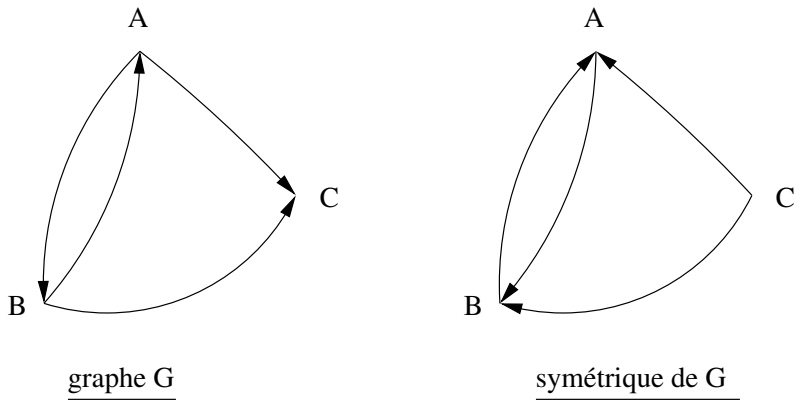
8.3 Définitions complémentaires

8.3.1 Symétrique

Soit $G = (E, \Gamma)$. On appelle **symétrique** de G le graphe $G^{-1} = (E, \Gamma^{-1})$ défini par :

$$\forall x \in E \ , \ \Gamma^{-1} = \{y \ , \ x \in \Gamma(y)\}$$

Exemple i:



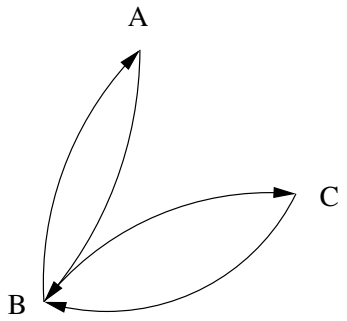
Graphe et son symétrique

Remarque: $y \in \Gamma^{-1}(x) \Leftrightarrow x \in \Gamma(x)$

Définition : Un graphe (E, Γ) est un graphe symétrique si $\Gamma^{-1} = \Gamma$.

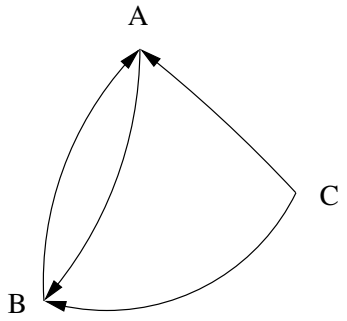
Exemple ii:

G symétrique



Exemple iii:

G non symétrique



Fermeture symétrique

Soit $G = (E, \Gamma)$. On appelle **fermeture symétrique** de G le graphe $G_s = (E, \Gamma_s)$ défini par :

$$\Gamma_s(x) = \Gamma(x) \cup \Gamma^{-1}(x) \quad \forall x \in E$$

La fermeture symétrique forme un graphe **non-orienté**.

Une **arête** est un ensemble $\{x, y\}$ ($x \in E$, $y \in E$) tel que $(x, y) \in \vec{\Gamma}$ ou $(y, x) \in \vec{\Gamma}$.

Algorithme de calcul du symétrique

Données : (E, Γ),
Résultat : Γ^{-1}.

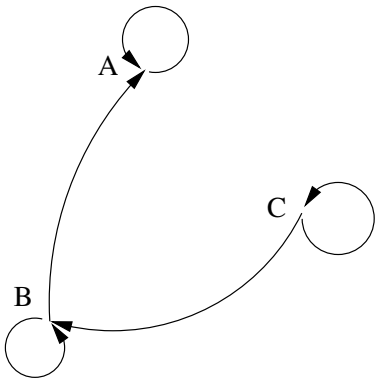
1: $\forall x \in E, \Gamma^{-1}(x) = \emptyset$
2: **for all** $y \in E$ **do**
3: **for all** $x \in \Gamma(y)$ **do**
4: $\Gamma^{-1}(x) = \Gamma^{-1}(x) \cup \{y\}$
5: **end for**
6: **end for**

Si l'on représente Γ^{-1} sous formes de liste chaînées, la ligne 1 est en $O(n)$. Intuitivement on pourrait dire que le corps des 2 boules imbriquées (ligne 4) est executé $n * n$ fois. Mais ces boucles reviennent à parcourir dans le pire des cas l'ensembles des arcs du graphe (m). Cet algorithme est donc en $O(n + m)$.

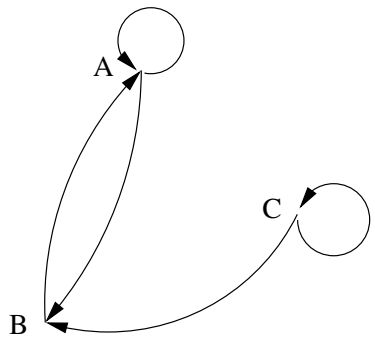
8.3.2 Reflexivité

Un graphe $G = (E, \Gamma)$ est **réflexif** si $\forall x \in E\ ,\ x \in \Gamma(x)$.
Exemple iv:

– G réflexif



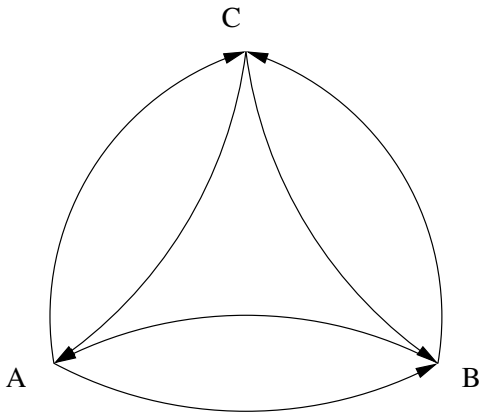
– G non réflexif



8.3.3 Graphe complet

Un graphe $G = (E, \Gamma)$ est dit **complet** si $\forall (x, y) \in E \times E\ ,\ x \neq y\ (x, y) \in \vec{\Gamma}$.
Exemple v:

G complet



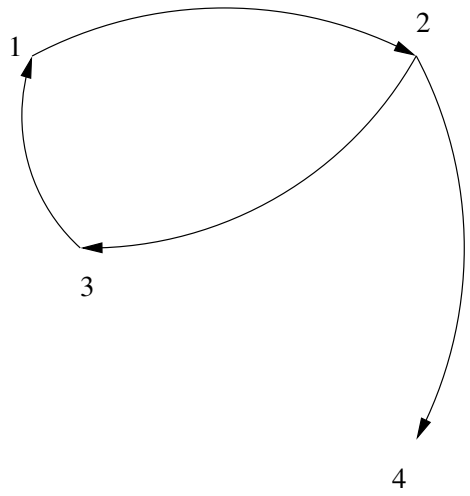
Les graphes complets à n sommets sont notés K_n.

8.4 Chemins, connexité

8.4.1 Chemins et chaînes

Soit $G = (E, \Gamma)$ et x_i une famille de points de E. Un chemin de x_0 à x_k est défini par $(x_0, x_1, x_2, \cdots, x_{k-1}, x_k)$ telle que $\forall i \in [1, k]$, $x_i \in \Gamma(x_{i-1})$. (Chaque point x_i du chemin est un sucesseur du point x_{i-1}).
On définit k comme étant la *longueur* du chemin : c'est le nombre d'arcs du chemin. Un chemin C est un *circuit* si $x_0 = x_k$.

Exemple i:



- $(1, 2, 1)$ n'est pas un chemin.
- $(1, 2, 3)$ est un chemin.
- $(1, 2, 3, 1, 2, 4)$ est un chemin.

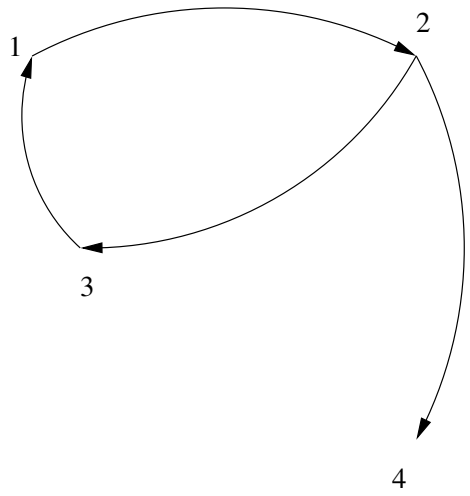
Chemin élémentaire

Un chemin C est dit *élémentaire* si les sommets x_i sont tous distincts (sauf éventuellement x_0 et x_k).

Proposition : Tout chemin C de x à y dans G contient (au sens suite extraite) un chemin élémentaire de x à y dans G.
Proposition : Soit $G = (E, \Gamma)$, avec $|E| = n$. La longueur d'un chemin élémentaire non-circuit dans G est inférieure à $n - 1$. De même la longueur d'un circuit élémentaire est inférieure à n.

Chaînes et cycles

Définition : Soit $G = (E, \Gamma)$, et Γ_s la fermeture symétrique de Γ. On appelle **chaîne** tout chemin dans (E, Γ_s).
On appelle **cycle** tout circuit dans (E, Γ_s) ne passant pas 2 fois par la même arête.
Exemple ii:



– $(4, 2, 1)$ n'est pas un chemin.
– $(4, 2, 1)$ est une chaîne.

Le tableau suivant résume les différentes définitions selon la nature du graphe :

Graphe orienté	Graphe non-orienté
arc	arête
chemin	chaîne
circuit	cycle

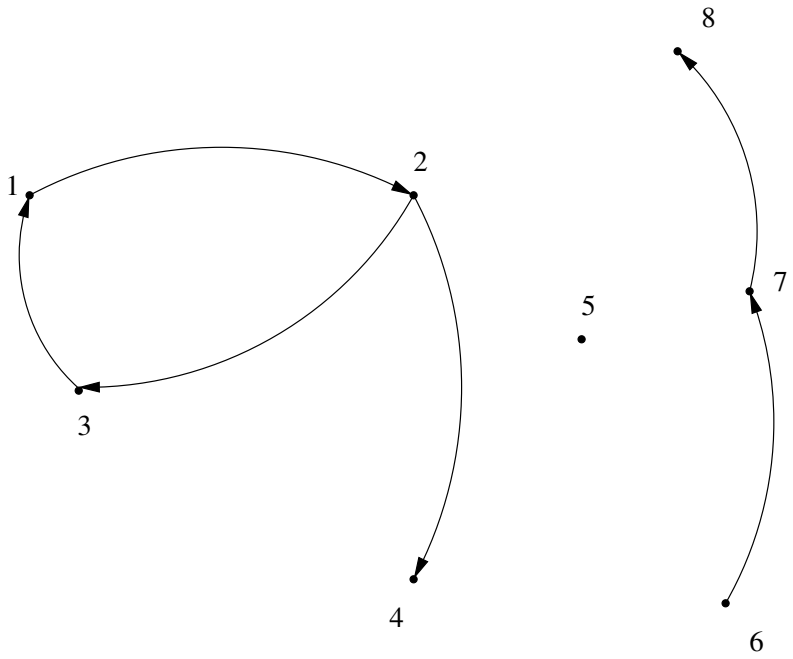
8.4.2 Composante connexe et fortement connexe

Composante connexe

Définition : Soit $G = (E, \Gamma)$ et $x \in E$. On définit C_x, la **composante connexe** de G contenant x par :

$$C_x = \{y \in E \ , \ \exists \text{chaîne entre } x \text{ et } y\} \cup \{x\}$$

Exemple iii:



- $C_1 = \{1, 2, 3, 4\} = C_2 = C_3 = C_4$
- $C_5 = \{5\}$
- $C_6 = C_7 = C_8$

La composante connexe contenant le sommet x est l'ensemble des points qu'il est possible d'attendre par une *chaîne* à partir de x.

Composante fortement connexe

Définition : Soit $G = (E, \Gamma)$ et $x \in E$. On définit C'_x, la **composante fortement connexe** de G contenant x par :

$$C'_x = \{y \in E \ , \ \exists \text{chemin entre } x \text{ et } y\} \cup \{x\}$$

Exemple iv:

Dans le graphe de l'exemple précédent :
- $C'_1 = \{1, 2, 3, 4\} = C'_2 = C'_3$
- $C'_4 = \{4\}$
- $C'_5 = \{5\}$

8.4.3 Calcul des composantes fortement connexes et des composantes connexes

Définitions préliminaires

Soit un graphe $G = (E, \Gamma)$, on définit plusieurs familles de graphes sur E :

- $\Gamma_0(x) = \{x\}$
- $\Gamma_1(x) = \Gamma(x)$
- $\Gamma_2(x) = \alpha(\Gamma_1)$
- $\cdots$
- $\Gamma_p(x) = \alpha(\Gamma_{p-1})$

(En utilisant la notation $\alpha(X)$ c'est à dire l'ensemble des sucesseurs de la partie X).
De même, on peut définir les prédecesseurs $\Gamma_p^{-1} = \alpha^{-1}(\Gamma_{p-1}^{-1})$.

Proposition : $\Gamma_p = \{y \in E$, $\exists$ un chemin de x à $y$dans G de longueur$p\}$
Proposition : $\Gamma_p^{-1} = \{y \in E$, $\exists$ un chemin de y à x dans G de longueur $p\}$
Définition :

$$\hat{\Gamma}_p = \bigcup_{i=0}^{p} \Gamma_i$$

$$\hat{\Gamma}_p^{-1} = \bigcup_{i=0}^{p} \Gamma_i^{-1}$$

Proposition : $\hat{\Gamma_p} = \{y \in E$, $\exists$ un chemin de x à y dans G de longueur $\leq p\} \cup \{x\}$. La même proposition s'applique aussi pour $\hat{\Gamma_p^{-1}}$. On définit aussi $\hat{\Gamma_\infty}(x)$ par l'ensemble des $y \in E$ tels qu'il existe un chemin de x à y dans G.
La composante fortement connexe C'_x de G contenant x se calcule par la formule :

$$C'_x = \hat{\Gamma}_\infty \cup \hat{\Gamma}_\infty^{-1}$$

La composante connexe de G contenant x s'écrit :

$$C_x = \hat{\Gamma}_{s\infty} = \hat{\Gamma}_{s\infty}^{-1}$$

Remarque: En général $\hat{\Gamma}_{s\infty} \neq \hat{\Gamma}_\infty \neq \hat{\Gamma}_{s\infty}^{-1}$.

Algorithme du calcul de la composante fortement connexe

Données : $(E, \Gamma, \Gamma^{-1}); a \in E$
Résultat : C'_a

1: $\hat{\Gamma}^\infty(a) \leftarrow \emptyset$; $T \leftarrow \{a\}$;
2: **while** $\exists x \in T$ **do**
3: $\quad T \leftarrow T \backslash \{x\}$

4: **for all** $y \in \Gamma(x)$ tel que $y \notin \hat{\Gamma}^{\infty}(a)$ **do**
5: $\hat{\Gamma}^{\infty}(a) \leftarrow \hat{\Gamma}^{\infty}(a) \cup \{y\}$
6: $T \leftarrow T \cup \{y\}$
7: **end for**
8: **end while**
9: $\hat{\Gamma}^{-\infty}(a) \leftarrow \emptyset$, $T \leftarrow \{a\}$;
10: **while** $\exists x \in T$ **do**
11: $T \leftarrow T \ \{x\}$
12: **for all** $y \in \Gamma^{-1}(x)$ **tel que** $y \notin \hat{\Gamma}^{-\infty}(a)$ **do**
13: $\hat{\Gamma}^{-\infty}(a) \leftarrow \hat{\Gamma}^{-\infty}(a) \cup \{y\}$
14: $T \leftarrow T \cup \{y\}$
15: **end for**
16: **end while**
17: $C'_a \leftarrow \left[\hat{\Gamma}^{\infty}(a) \cap \hat{\Gamma}^{-\infty}(a)\right] \cup \{a\}$

Algorithme du calcul de la composante connexe

Données : $(E, \Gamma, \Gamma^{-1}); a \in E$ *Résultat :* C_a

---

1: $C_a \leftarrow \{a\}$; $T \leftarrow \{a\}$
2: **while** $\exists x \in T$ **do**
3: $T \leftarrow T \backslash \{x\}$
4: **for all** $y \in \Gamma(x)$ tel que $y \notin C_a$ **do**
5: $C_a \leftarrow C_a \cup \{y\}$
6: $T \leftarrow T \cup \{y\}$
7: **end for**
8: **for all** $y \in \Gamma^{-1}(x)$ tel que $y \notin C_a$ **do**
9: $C_a \leftarrow C_a \cup \{y\}$
10: $T \leftarrow T \cup \{y\}$
11: **end for**
12: **end while**

8.4.4 Degrés

Soit un graphe $G = (E, \Gamma)$ et un sommet $x \in E$. On définit le **degré exterieur** de x, noté $d^+(x)$ par :

$$d^+(x) = |\Gamma(x)|$$

Le degré extérieur est le nombre de successeurs de x.
On définit le **degré intérieur** de G, noté $d^-(x)$ par :

$$d^-(x) = |\Gamma^{-1}(x)|$$

Le degré intérieur est le nombre de prédecesseurs de x.

Le degré du sommet x est défini par $d(x) = d^+(x) + d^-(x)$. La somme des degrés de tous les sommets d'un graphe G est paire.
Proposition : Dans tout graphe, le nombre de sommets de degré impair est pair.

$k-$**régularité**

Soit k un nombre entier positif, on dit que G est $k-$régulier si le degré de chacun de ses sommets est égal à k.

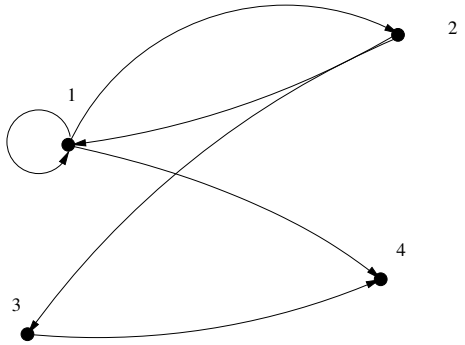
8.5 Représentation matricielle

Soit $G = (E, \Gamma)$. G peut se représenter par la matrice booléenne $\mathcal{M}_\Gamma = m_{ij}$ telle que :

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

Exemple i:

Soit le graphe $G = (E, \Gamma)$ suivant :



La matrice d'adjacence $\mathcal{M}_\Gamma$ associée à G est la suivante :

$$\left[\begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 1 & 1 & 0 & 1 \\ 2 & 1 & 0 & 1 & 0 \\ 3 & 0 & 0 & 0 & 1 \\ 4 & 0 & 0 & 0 & 0 \end{array}\right]$$

Considérons le produit booléen de matrices et les opérateurs $\vee$ (*ou* binaire) et $\wedge$ (*et* binaire).
Soit $A = a_{ij}$ et $B = b_{ij}$ deux matrices booléennes, alors $C = A * B = [c_{ij}]$ est défini par :

$$c_{ij} = \bigvee_{k=1}^{n} (a_{ik} \wedge b_{kj})$$

Soit $M^p = M * M * \cdots M$, p fois la matrice booléenne associée au graphe G, $M^p_{ij} = 1 \Leftrightarrow \exists$ un chemin de longueur p dans G de i à j.

Remarque: Pour tout graphe G, il n'existe pas de nombre $k \in \mathbb{N}$ tel que $M^k = M^{k+1}$. (Par exemple, prendre 2 points A et B et 2 arcs $\vec{AB}$ et $\vec{BA}$.

Soit $\hat{M}^p = M \vee M^2 \vee \cdots M^p$; $\hat{M}^p_{ij} = 1 \Leftrightarrow \exists$ un chemin de longueur $\leq p$ dans G de i à j.

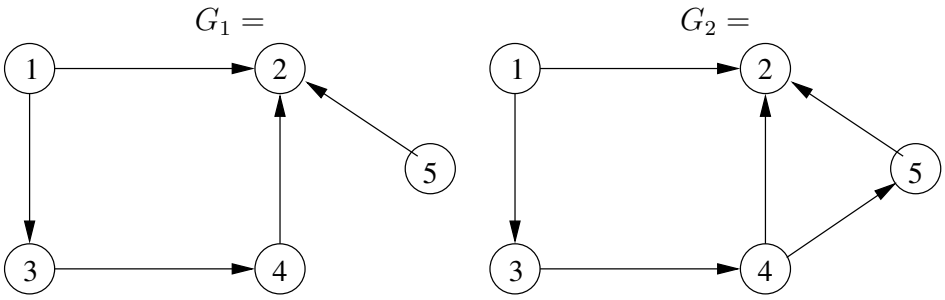
Remarque: Si $|E| = n$ alors $\forall p \leq n-1$, $\hat{M}^p = \hat{M}^{n-1} = \hat{M}^\infty$.

8.6 Graphe biparti

On dit qu'un graphe $G = (E, \Gamma)$ est *biparti* si l'ensemble E des sommets peut être partitionné (ou colorié) en deux sous-ensembles distincts E_1 et E_2 de telle sorte que :

$$\forall u = (x, y) \in \vec{\Gamma}, \left\{ \begin{array}{lcl} x \in E_1 & \Rightarrow & y \in E_2 \\ x \in E_2 & \Rightarrow & y \in E_1 \end{array} \right.$$

Exemple i:



On voit G_1 est biparti avec $E_1 = \{1, 4, 5\}$ et $E_2 = \{2, 3\}$. G_2 n'est pas biparti.

On peut montrer qu'un graphe est biparti si et seulement si il ne contient pas cycle impair.
Preuve : Soit un cycle $C = x_0, x_1, \cdots x_k$ avec $x_0 = x_k$ puisque C est un cycle. D'après la définition d'un graphe biparti, si $x_0 \in E_1$, alors $x_1 \in E_2$. Ainsi par réccurence, si on a $x_k \in E_1$, alors $x_{k+1} \in E_2$.
On peut poser que $x_k \in E_{k(2)+1}$ c'est à dire que x_k appartient à l'ensemble E_i avec i égal à k modulo 2 (plus 1 car on a posé que $x_0 \in E_1$). On a alors :

$$x_k \in \begin{cases} E_1 & \text{si k pair} \\ E_2 & \text{si k impair} \end{cases}$$

Puisque C est un cycle, $x_0 = x_k$ et donc si k impair on a $x_k = x_0 \in E_2$ ce qui est en contradiction avec l'hypothèse de départ $x_0 \in E_1$. On ne peut donc avoir de cycle de longueurs impaires dans un graphe biparti.

8.6.1 Algorithme

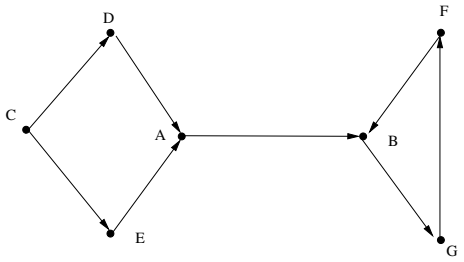
Chapitre 9

Arbre et arborescences

9.1 Définitions

9.1.1 Isthme

Soit $G = (E, \vec{\Gamma})$, $u \in \vec{\Gamma}$. On dit que u est un isthme si sa suppression de $\vec{\Gamma}$ augmente le nombre de composantes connexes de G.
Exemple i:



L'arc $u = (A, B)$ est un isthme.

Si u est un isthme, u n'appartient à aucun cycle de G.

Proposition : Soit $G = (E, \vec{\Gamma})$, tel que $|E| = n$ et $|\vec{\Gamma}| = m$. Si G est connexe, $m \geq n - 1$. Si G est sans cycle, $m \leq n - 1$.

9.1.2 Arbre

Un *arbre* est un graphe connexe et sans cycle. Une *forêt* est un graphe sans cycle.

Remarque: Un arbre ne comporte pas de boucles. Dans un arbre, chaque arc est un isthme.

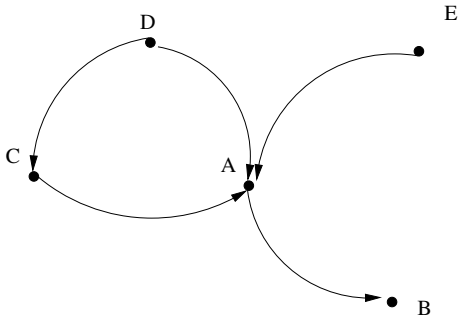
Proposition : Soit $G = (E, \vec{\Gamma})$, tel que $|E| = n \geq 2$ et $|\vec{\Gamma}| = m$. Les propriétés suivantes sont équivalentes :

1. G est connexe et sans cycles,
2. G est connexe et $m = n - 1$,
3. G est sans cycles et $m = n - 1$,
4. G est sans boucles et $\forall a \in E$, $\forall b \in E$, $\exists$ une chaîne élémentaire unique entre a et b.

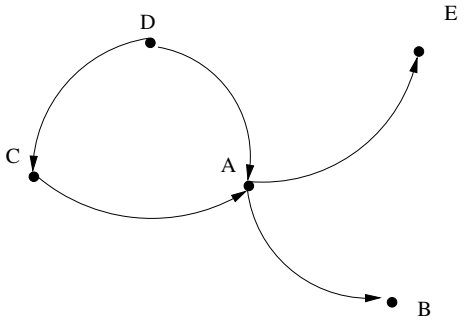
9.1.3 Racine

Soit $G = (E, \vec{\Gamma})$, $x \in E$. x est une *racine* de G (resp. *antiracine*) si $\forall y \in E$ $\{x\}$, $\exists$ un chemin de x à y. (resp. y à x)

Exemple ii:



Ce graphe n'a pas de racine, et a pour antiracine B.



Ce graphe a pour racine D et n'a pas d'antiracine.

9.1.4 Arborescences

G est une *arborescence* (resp. *anti-arborescence*) de racine $r \in E$ si :

1. G est un arbre
2. r est une racine (resp. anti-racine) de G

9.1.5 Découpage en niveau

Une arborescence peut être découpée en niveaux (cf figure 9.1).

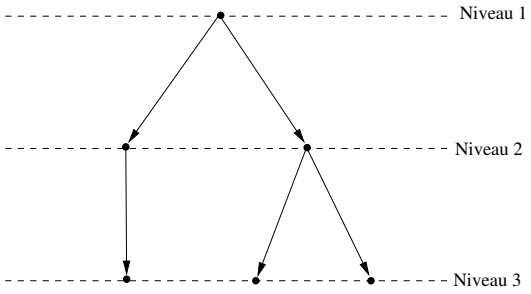


FIG. 9.1 – *Représentation en niveaux*

9.2 Exemples et applications

9.2.1 les pièces de monnaie

Soit 8 pièces de monnaies, dont une est fausse (elle est plus légère que les autres). On possède deux balances. La balance 1 permet de savoir quel plateau est le plus lourd ; la balance 2 possède 3 états ($>$, $<$, $=$). On cherche à trouver la fausse pièce en minimisant le nombre de pesées

Arborescence de décision

Soit $G = (E, \vec{\Gamma})$ une arborescence de racine $r \in E$. Soit X un ensemble fini de possibilités. G est une *arborescence de décision* sur X s'il existe une application $f : E \Rightarrow \mathcal{P}(X)$ telle que :

- $f(r) = X$
- $\forall a \in E \ / \ \Gamma(a) \neq \emptyset$, $f(a) = \bigcup_{b \in \Gamma(a)} f(b)$,
- $\forall a \in E \ / \ \Gamma(a) = \emptyset$, $|f(a)| = 1$.

Dans le cas de notre exemple de la balance, on pose 4 pièces sur chaque plateau. Un des 2 plateaux sera forcèment moins lourd car il y a une fausse pièce. On prend le lot de 4 pièces les moins lourdes et on le divise en 2 et ainsi de suite. Le nombre de pesées pour trouver la fausse pièce est égale au nombre de niveau du graphe 9.2 page suivante, soit ici 3.
On peut reproduire la figure 9.2 page suivante pour la balance 2. On aura alors un arbre à 2 niveaux.

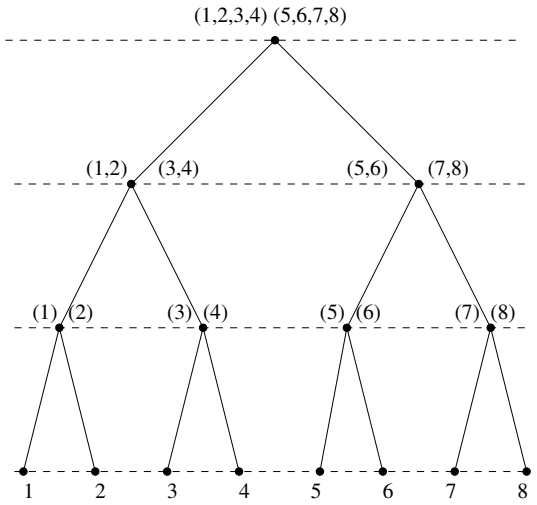


FIG. 9.2 – *Arbre de décision de la balance 1*

9.2.2 Arithmétique

On s'interesse à l'évaluation d'expressions arithmétiques, en respectant les priorités des différents opérateurs. Soit l'expression arithmètique (A) :

$$(A) = \frac{ae^x - b(y+1)}{2}$$

Le problème est l'ordre d'évaluation, car $a - b \neq b - a$. Il est alors nécessaire d'ordonner.

Sur l'arbre de la figure 9.3 page suivante correpondant à l'expression (A), chaque sommet du graphe est soit une donnée (litérale ou numérique), soit un opérateur. Les opérateur possédent au minimum 2 successeurs (sauf opérateur unaires). Les données n'ont aucun sucesseur ; ce sont les *feuilles* de l'arbre.

Arborescence ordonnée

Soit E un ensemble fini, on note $O(E)$ l'ensemble des $k-$uplets ordonnées d'éléments de E.
$\mathring{G} = (E, \mathring{\Gamma})$, où $\mathring{\Gamma}$ est une application de E dans $O(E)$ est un graphe ordonné sur E.
A tout graphe ordonné $(E, \mathring{\Gamma})$, on peut faire correspondre le graphe (E, Γ) tel que $\Gamma(x) = \{$ éléments du $k-$uplet $\mathring{\Gamma}(x)\}$. On a $\mathring{G} = (E, \mathring{\Gamma})$ une arborescence ordonnée (ou arborescence plane) si (E, Γ) est une arborescence.

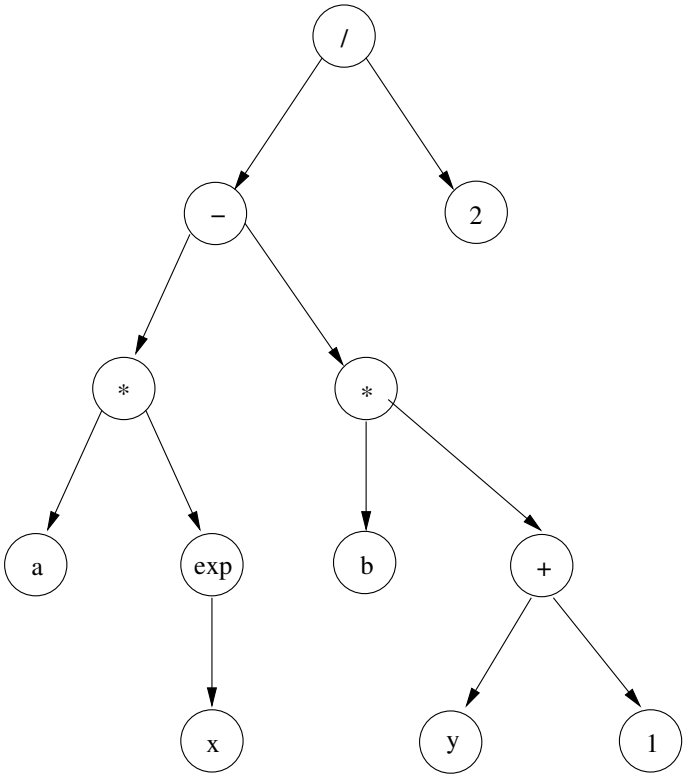


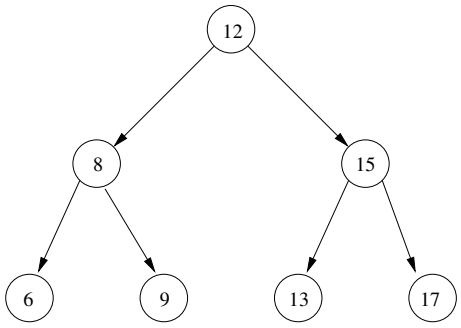
FIG. 9.3 – *Arbre d'évaluation arithmétique*

9.2.3 Arbre de recherche

Soit D un ensemble appelé *domaine*, muni d'une relation d'ordre totale (que l'on notera $\leq$). Soit $X \subset D$ et $n \in D$. La question que l'on se pose est de savoir si appartient à l'ensemble X.
On peut par exemple prendre $D =$ { l'ensemble des mots de la langue française }, X une partie de ces mots (par exemple l'ensemble des mots d'une page web), et l'on prend l'ordre lexicographique usuel.

Si X est représenté sous forme d'une liste, la résolution du problème $n \in X$? s'effectue en $O(|X|)$. Si X est représenté sous forme d'un tableau de booléen, la résolution s'effectue en $O(1)$ mais l'emcombrement mémoire est en $O(|D|)$. Si X est sous forme d'une arborescence binaire, la recherche s'effectue dans le pire des cas en $O(|X|)$, mais si l'arborescence est équilibrée, la recherche est en $O(log|X|)$. On a alors réalisé un *index*

Exemple i:



Arbre de recherche sur $\mathbb{N}$, pour la relation $\leq$. $X = \{6, 8, 9, 12, 15, 13, 17\}$

9.3 Arbre de poids minimum

9.3.1 Poids

Pour chaque arc $\in \vec{\Gamma}$, on associe une valeur, le *poids* de l'arc. Soit $G = (E, \vec{\Gamma})$, un graphe connexe et sans boucles. Soit P une application de $\vec{\Gamma} \rightarrow \mathbb{R}$, telle que $\forall u \in \vec{\Gamma}$, $P(u)$ est appelé *poids* de l'arc u.

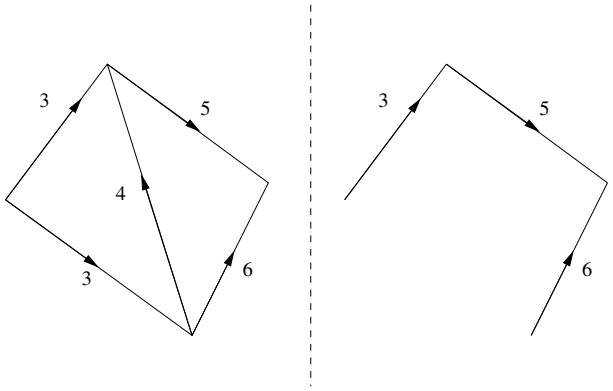
Le graphe $(E, \vec{\Gamma}, P)$ est appelé *graphe pondéré*. Soit $\vec{\Gamma}' \subset \vec{\Gamma}$ un graphe partiel de G, $P(\vec{\Gamma}') = \sum_{u \in \vec{\Gamma}'} P(u)$ définit le poids du graphe partiel $\vec{\Gamma}'$.
Soit $(E, \vec{\Gamma}')$ un graphe partiel de G qui soit un arbre. On dit que $(E, \vec{\Gamma}')$ est un arbre de poids maximum sur G, si $\forall \vec{\Gamma}'' \in \vec{\Gamma}$ tel que $(E, \vec{\Gamma}'')$ soit un arbre, on a la relation suivante :

$$P(\vec{\Gamma}'') \leq P(\vec{\Gamma}')$$

Soit $(E, \vec{\Gamma}')$ un graphe partiel de G qui soit un arbre. On dit que $(E, \vec{\Gamma}')$ est un arbre de poids minimum sur G, si $\forall \vec{\Gamma}'' \in \vec{\Gamma}$ tel que $(E, \vec{\Gamma}'')$ soit un arbre, on a la relation suivante :

$$P(\vec{\Gamma}'') \geq P(\vec{\Gamma}')$$

Exemple i:



Un graphe pondéré et son arbre de poids maximum.

Un arbre de poids maximum pour $G = (E, \vec{\Gamma}, P)$ est un arbre de poids minimum pour $G' = (E, \vec{\Gamma}, -P)$.

Remarque: On peut se demander combien d'arbres peut-on construire à partir d'un graphe. CAYLEY *a apporté la réponse en 1889. Soit* $G = (E, \vec{\Gamma})$ *avec* $|E| = n$*, on peut construire* n^{n-2} *arbres sur* G

9.3.2 Propriétés des arbres de poids maximum

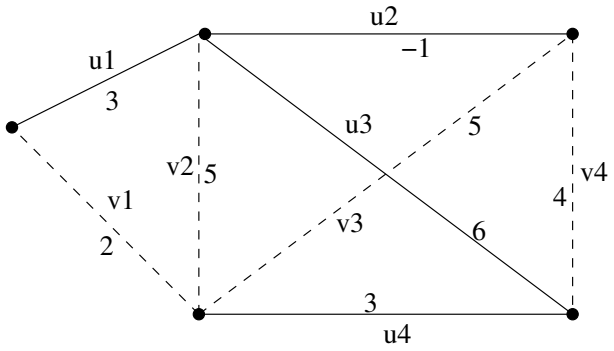
Soit $(E, \vec{\Gamma})$ un graphe connexe et sans boucle. Soit P l'application de pondération associé à ce graphe. Soit $\vec{A} \subset \vec{\Gamma}$ tel que $(E, \vec{A})$ soit un arbre.
On prend $\vec{u}$ un arc appartenant à $\vec{\Gamma}$ mais pas $\vec{A}$, C_u désigne l'ensemble des arcs de $\vec{A}$ formant une chaîne entre les extrémités de $\vec{u}$.
On prend $\vec{v}$ un arc appartenant à $\vec{A}$, Ω_v désigne l'ensemble des arcs de $\vec{\Gamma} \backslash \vec{A}$ ayant leur extrémités dans 2 composantes connexes différentes de $(E, \vec{A} \backslash \{\vec{v}\})$.

les 2 propriétés suivantes sont équivalentes et caractèrisent un arbre de poids maximum :

1. $\forall u \in \vec{\Gamma} \backslash \vec{A}$, $P(u) \leq \min_{\omega \in C_u} P(\omega)$.
2. $\forall v \in \vec{A}$, $P(v) \leq \max_{\omega \in \Omega_v} P(\omega)$.

Exemple ii:

On se propose d'étudier le graphe suivant :



On veut vérifier si l'arbre representé par les arcs en pointillés est bien un arbre de poids maximum.

$$\begin{array}{ll} C_{u1} = v_1, v_2 & \Omega_{v1} = u_1 \\ C_{u2} = v_2, v_3 & \Omega_{v2} = u_1, u_2, u_3 \\ C_{u3} = v_2, v_3, v_4 & \Omega_{v3} = u_2, u_3, u_4 \\ C_{u4} = v_3, v_4 & \Omega_{v4} = u_3, u_4 \end{array}$$

On a $P(u_1) = 3 > P(v_1) = 2$, la propriété 1 n'est donc pas vérifié. De plus $P(v_2) = 5 < P(u_3) = 6$, la propriété 2 n'est pas vérifiée. L'arbre representé n'est donc pas un arbre de poids maximum

9.4 Algorithmes de Kruskal

Nous allons voir des algorithmes qui permettent d'extraire les arbres de poids maximum d'un graphe.

9.4.1 Algorithme de Kruskal-1

Soit $G = (E, \vec{\Gamma})$ un graphe connexe et sans boucle, et P une application de pondération de $\vec{\Gamma}$ sur $\mathbb{R}$.
On note $\vec{\Gamma} = \{u_1, u_2, \cdots u_n\}$ avec $\forall i, P(u_i) \geq P(u_{i+1})$ (Les sommets sont triés par ordre décroissant).
Soit la famille $\vec{\Gamma}_i$ définie par :

$$\begin{array}{rcl} \vec{\Gamma}_0 & = & \emptyset \\ \vec{\Gamma}_i & = & \begin{cases} \vec{\Gamma}_{i-1} \cup \{u_i\} & \text{si } (E, \vec{\Gamma}_{i-1} \cup \{u_i\}) \text{ est sans cycles} \\ \vec{\Gamma}_{i-1} & \text{sinon} \end{cases} \end{array}$$

Alors $\vec{\Gamma}_n$ est un arbre de pois maximum.
Chaque $\vec{\Gamma}_i$ se construit en rajoutant l'arc u_i à un nouveau graphe G' si le graphe G' est sans cycle, c'est à dire s'il conserve ses propriétés d'arbre. Comme l'on rajoute les arcs les plus *valorisant* au début, on est sur d'avoir un arbre de poids maximum.
On n'a pas forcèment besoin de tout calculer, on peut s'arrêter quand $|\vec{\Gamma}_i| = n-1$.

complexité

Il faut d'abord constituer une liste triée par ordre décroissant des arcs du graphe. Cette opération s'effectue au mieux en $O(nlog(n))$. Le test pour savoir si un graphe est sans cycles et en $O(n+m)$. Comme ce test est réalisé m fois au pire, la complexité de l'algorithme de KRUSKAL-1 est en :

$$O(m(m+n) + n\log(n)) = O(m(m+n))$$

9.4.2 Algorithme de Kruskal-2

Cette algorithme procède de façon identique, mais au lieu de construire un nouveau graphe avec les arcs les plus coûteux, il procède en enlevant du graphe d'origine les arcs les moins valorisants. La condition de suppression d'un arc u_i ne s'écrit plus *tant que* $(E, \vec{\Gamma}_i)$ *est sans cycle* mais *tant que* $(E, \vec{\Gamma} \backslash \{u_i\})$ *est connexe.* Pour la même raison, les arcs doivent être triés par ordre croissant. La complexité de l'algorithme de KRUSKAL-2 est la même que celle de KRUSKAL-1.

9.4.3 Algorithme de Prim

L'algorithme de PRIM fonctionne ainsi :

1: $T = \emptyset$, $A = \{x \in E\}$
2: **while** $|T| \leq n - 1$ **do**
3: $\quad e = (u, v)$, tel que $u \in A$ et $v \notin A$ et e de coût minimal.
4: $\quad A = A \cup \{v\}$
5: $\quad T = T \cup e$
6: **end while**

Chapitre 10

Plus courts chemins

10.1 Définition

10.1.1 Réseau

Nous allons travailler sur un graphe sans boucle $G = (E, \vec{\Gamma}, l)$ avec l l'application poids ($\vec{\Gamma} \Rightarrow \mathbb{R}$). L'application p associe à chaque arc un réel appelé *longueur de l'arc.*
$(E, \vec{\Gamma}, l)$ est un **réseau**.

Soit $\pi = (x_0, x_1, \cdots x_n)$ un chemin dans G. On définit la longueur de π par :

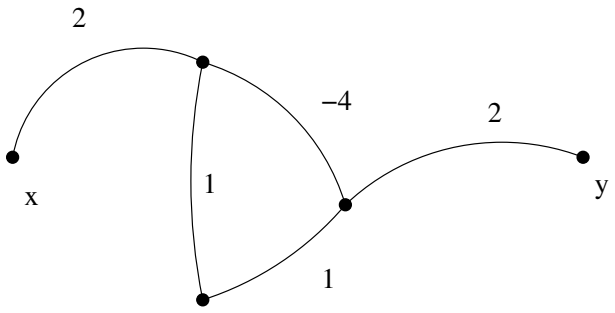
$$l(\pi) = \sum_{x_i, x_{i+1} \in \pi} l(\{x_i, x_{i+1}\})$$

10.1.2 Plus court chemin

Soient x et y deux sommets de E. Dans la recherche d'un plus court chemin de x à y, trois cas peuvent se présenter :

- Il n'existe aucun chemin de x à y (si x et y appartiennent à deux composantes connexes différentes de G).
- Il existe un chemin de x à y mais pas de plus court chemin.
- Il existe un plus court chemin de x à y.

Exemple i:



Il n'existe pas de plus court chemin de x à y. C'est à dire que pour tout chemin π de x à y, on peut trouver un chemin π' tel que $l(\pi') < l(\pi)$.

Remarque: Un plus court chemin dans $G = (E, \vec{\Gamma}, l)$ est un plus long chemin dans $G' = (E, \vec{\Gamma}, -l)$, et réciproquement.

Circuit absorbant

Un circuit de longueur négative est appelé *cycle absorbant*. Dans l'exemple précédent, il existe un cycle absorbant.
Si une composante connexe présente un cycle absorbant, alors il n'existe pas de plus court chemin entre deux points de cette composante.

Soit $R = (E, \vec{\Gamma}, l)$ un réseau. Soit $s \in E$, une condition nécessaire et suffisante pour qu'il existe un plus court chemin entre s et tout point de $E \backslash \{s\}$ est :

1. s est une racine de $(E, \vec{\Gamma})$ et,
2. Il n'y a pas de circuit absorbant dans R.

10.2 Problèmatique du plus court chemin

Il y a 3 problèmatiques à la recherche d'un plus court chemin :
– Soient x et y deux sommets de E, trouver un plus court chemin de x à y,
– Soit $i \in E$, trouver les longueurs $\pi(j)$ des plus courts chemin de i à j, $\forall j \in E \backslash \{i\}$.
– Trouver les longueurs des plus courts chemins de i à j, $\forall i, j \in E$.
Pour résoudre le problème (1) on doit d'abord résoudre (2) avec $i = x$.

10.2.1 Graphe des plus courts chemin

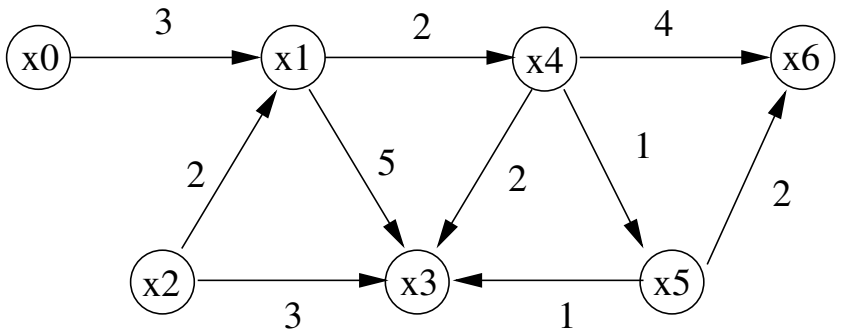
Soit $R = (E, \vec{\Gamma}, l)$ un réseau sans cycle absorbant, soit $i \in E$. On définit $\Pi_i : E \Rightarrow \mathbb{R} \cup \{\infty\}$ définie par :

$$\begin{cases} \Pi_i(x) & = & \text{la longueur d'un plus court chemin de i à x s'il existe} \\ \Pi_i(x) & = & \infty \text{ sinon} \end{cases}$$

Soit $(E, \vec{\Gamma}', l')$ le réseau défini par :

$$\vec{\Gamma}' = \left\{(x, y) \in \vec{\Gamma}, l'(\{x, y\}) = \pi_i(y) - \pi_i(x)\right\}$$

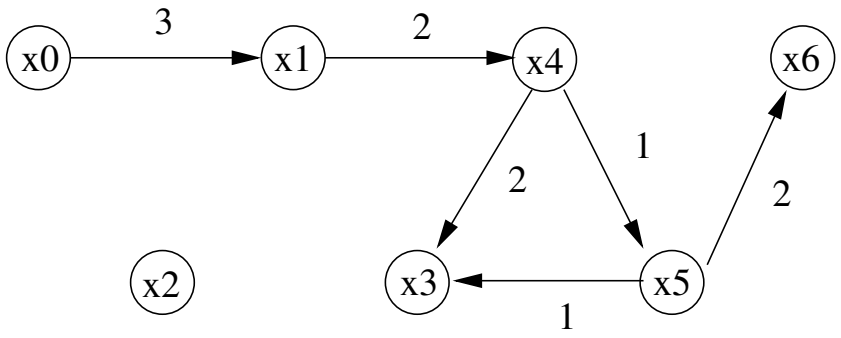
$(E, \vec{\Gamma}', l')$ est appelé *graphe des plus courts chemins de i*. $\vec{\Gamma}'$ est constitué des arcs faisant partie d'un plus courts chemin de i à un sommet de E.
Un plus court chemin de i à x dans R est un chemin de i à x dans $(E, \vec{\Gamma}')$.
Exemple i:



On peut construire l'application $\pi_{x_0}(x)$

x	$\pi_{x_0}(x)$
x_0	0
x_1	3
x_2	∞
x_3	7
x_4	5
x_5	6
x_6	8

Le graphe des plus court chemins est alors :



Arborescence de plus courts chemins

Soit $R = (E, \vec{\Gamma}, l)$ un réseau sans cycles absorbant. Soit $i \in E$ et $(E, \vec{\Gamma}')$ le graphe des plus courts chemins défini précédemment. On dit que (E', A) est une arborescence des plus courts chemins pour R si :

- $E' = \{x \in E, \pi_i(x) \neq \infty\}$
- (E', A) est une arborescence ayant pour racine le sommet i.
- $\vec{A} \subset \vec{\Gamma}$

Remarque: En général une arborescence des plus courts chemins n'est pas un arbre de poids minimum.

10.3 Algorithme de Floyd

Cet algorithme permet de calculer la longueur du plus court chemin dans un réseau $R = (E, \vec{\Gamma}, l)$, pour tout couple de sommets (x, y). On notera E_k l'ensemble constitué des k premiers sommets de E ($k \leq n$). On considère la matrice $L^0 = (l^0_{ij})$ telle que $l^0_{ii} = 0$ et si $i \neq j$:

$$l^0_{ij} = \begin{cases} l(i,j) & \text{si } (i,j) \in \vec{\Gamma} \\ 0 & \text{si } i = j \\ +\infty & \text{sinon} \end{cases}$$

On considère la matrice $L^k = (l^k_{ij})$ où l^k_{ij} représente la longueur minimum des chemins de i à j dont les sommets intermédiaires sont dans E_k.
Pour $k = 0$, on pose $E_k = \emptyset$, on voit alors que L^0 correspond bien à L^k avec $k = 0$. Ainsi L^n constitue la solution du problème, c'est à dire que l^n_{ij} est bien la longueur du plus court chemin de i à j dans l'ensemble $E_n = E$.
Pour calculer L_k on utilise l'équation de récurrence suivante (on supposera que le graphe est sans circuit absorbant) :

$$l^k_{ij} = \min\left(l^{k-1}_{ik} + l^{k-1}_{kj}, l^{k-1}_{ij}\right)$$

Ce problème peut se résoudre en utilisant la programmation dynamique. Sa complexité est en $O(n^3)$ (avec $n = |E|$) en temps de calcul, et $O(n^2)$ en mémoire.

En effet on n'a pas besoin de mémoriser chaque matrice L^k mais seulement la matrice courante et la précédente.

Pour détecter la présence de circuit absorbant, il suffit de regarder s'il existe un sommet x tel que $l(x,x) \leq 0$. En supposant qu'à l'état initial $L^0_{ii} = 0$, on applique la relation de récurrence sur toute la matrice et l'on regarde si des éléments diagonaux sont inférieurs à 0.
Lorsque l'équation de récurrence a été appliquée pour former les n matrices L^n, on a alors l^n_{ij} qui est égal à la longueur d'un plus court chemin du sommet i au sommet j (s'il existe, ∞ sinon).

10.4 Algorithme de Bellman

Nous allons étudier l'algorithme de Bellman pour calculer les plus courts chemins dans un réseau à longueurs quelconques. Cet algorithme ainsi que celui de Dijkstra ne calculent pas la longueur du plus court chemin pour tout couple de point $(x,y) \in G$, mais seulement la longueur des plus courts chemin d'un sommet inital i à tout autre sommet x du graphe.
But : soit $R = (E, \vec{\Gamma}, l)$, $i \in E$, calculer π_i.
L'idée est qu'un chemin de i à x dans R passe nécessairement par un $y^* \in \Gamma^{-1}(x)$.
La longueur d'un plus court chemin de i à x passant par y^* est :

$$\pi_i(x) = \pi_i(y^*) + l(y^*, x)$$

Et donc :

$$\pi_i(x) = \min_{y \in \Gamma^{-1}(x)} \{\pi_i(y) + l(y,x)\}$$

10.4.1 Algorithme en pseudo language

Données : $E, \Gamma, \Gamma^{-1}, l, i \in E$
Résultat : $\pi, CIRCABS$

1: $CIRCABS = FAUX$; $\pi^0(i) = 0$; $\pi^1(i) = 0$; $k = 1$
2: **for all** $x \in E \backslash \{i\}$ **do**
3: $\quad \pi^0(x) = \infty$
4: $\quad \pi^1(x) = \infty$
5: **end for**
6: **for all** $x \in E$ tel que $i \in \Gamma^{-1}(x)$ **do**
7: $\quad \pi^1(x) = l(i,x)$
8: **end for**

9: **while** $k < n+1$ et $\exists x \in E$ tel que $\pi^k(x) \neq \pi^{k-1}(x)$ **do**

10: $k = k+1$

11: **for all** $x \in E$ **do**

12: $\pi^k = \min\left[\pi^{k-1}(x), \pi^{k-1}(y) + l(y,x); y \in \Gamma^{-1}(x)\right]$

13: **end for**

14: **end while**

15: **if** $k = n+1$ **then**

16: $CIRCABS = VRAI$

17: **end if**

Cet algorithme renvoit également dans la variable $CIRCABS$ la présence d'un cycle absorbant.

Complexité

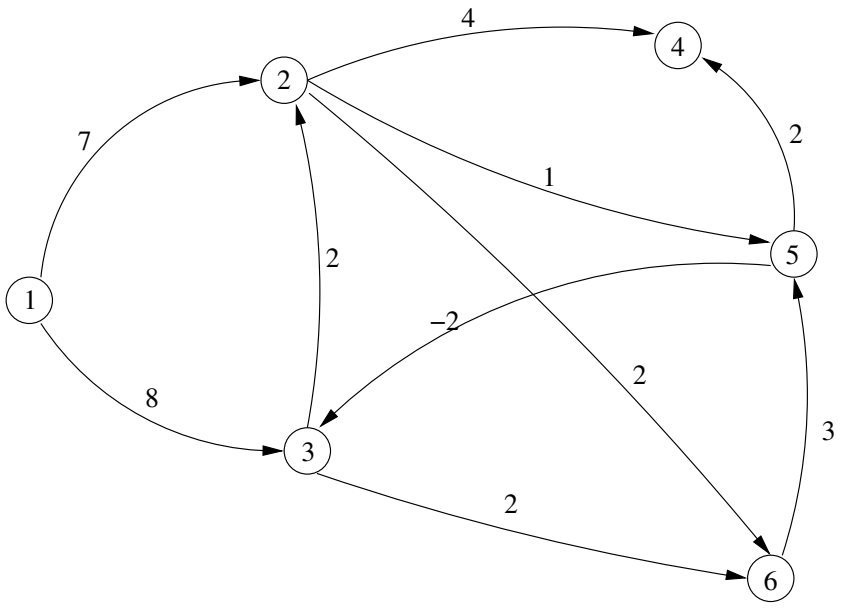
- L'initialisation de la ligne 1 est en temps constant $O(1)$
- La boucle d'initialisation des lignes 2 à 5 est en $O(n)$
- La boucle d'initialisation des lignes 6 à 8 est en $O(n+m)$
- Le corps de boucle principale est executé $n+1$ fois au maximum. La complexité de l'évaluation de la ligne 9 est en $O(n)$ et le corps de la boucle 9 à 14 est en $O(n+m)$.

Au total, la complexité de l'algorithme de Bellman est $O(n(n+m))$.

Remarque: La complexité de l'algorithme de Bellman *dans le cas d'un graphe dense ($m = n^2$) devient $O(n^3)$.*

10.4.2 Exemple d'éxécution

Exemple i:



Construisons π^k pas à pas pour l'algorithme de BELLMAN avec $i = 1$.

	1	2	3	4	5	6
π^0	0	∞	∞	∞	∞	∞
π^1	0	7	8	∞	∞	∞
π^2	0	7	8	11	8	9
π^3	0	7	6	10	8	9
π^4	0	7	6	10	8	8

10.4.3 Justification

L'idée est que la longueur d'un chemin du sommet i à n'importe quel sommet x est :

$$\pi_i(x) = \min_{y \in \Gamma^{-1}(x)} \{\pi_i(y) + l(y, x)\}$$

Cete formule locale marche pour tout le graphe, et il faut l'appliquer jusqu'a stabilité (ou jusqu'à n itérations, dans ce cas il y a présence de cycle absorbant).

On appelle $k-$chemin un chemin possèdant au plus k arcs. Dans l'algorithme de BELLMAN, $\pi_i^k(x)$ représente la longueur d'un au plus $k-$chemin de i à x (s'il existe, ∞ sinon).
Pour prouver cette proposition, nous allons procéder par récurrence.
La propriété est trivialement vérifiée pour $k = 0$ et $k = 1$. Supposons qu'elle est vérifiée jusqu'à l'étape $k - 1$ et plaçons-nous à l'étape k.
Soit $\mathcal{C}_k$ l'ensemble des sommets y tels qu'il existe un $k-$chemin de i à y.

– Si $x \notin \mathcal{C}_k$, $\forall y \in \Gamma^{-1}(x)$, $y \notin \mathcal{C}_{k-1}$. Donc d'après l'équation de récurrence à l'étape k, $\pi_k(x) = \pi_{k-1}(y) \forall y \in \Gamma^{-1}(x) = \infty$.
– Si $x \in \mathcal{C}_k$ et $x \neq i$ alors $\exists y \in \Gamma^{-1}(x)$ tel que $y \in \mathcal{C}_{k-1}$. Donc $\pi_{k-1}(y) < \infty$. Tout $k-$chemin de i à x passe par un $y \in \Gamma^{-1}(x)$. Un plus court $k-$chemin de i à x est composé un plus court $k-1$chemin de i à un sommet $y \in \Gamma^{-1}(x)$ et de l'arc (y, x) de longueur l. Il a pour longueur $\pi^{k-1}(y) + l(y, x)$. La longueur d'un plus court $k-$chemin de i à x est donc :

$$\min_{y \in \Gamma^{-1}(x)} \pi_i^{k-1}(y) + l(y, x)$$

Quand il existe, un plus court chemin de i à x possède au plus $n-1$ arcs (avec $n = |E|$). En effet un plus court chemin ne peut être qu'un chemin élémentaire (ne comportant pas de circuit).
Si à la fin de l'algo, $k \leq n$, alors $\forall x \in E$, il existe un plus court chemin de i à x. Il n'y pas de circuit absorbant. Si pour $k = n+1$, $\exists x \in E$ tel que $\pi^{n-1}(x) \neq \pi^n(x)$, alors il n'existe pas de plus court chemin entre i et x, donc le graphe présente un cycle absorbant.

Remarque: Pour implémenter l'algorithme de Bellman*, il suffit de 2 tableaux : π courant et π précédent. Les accès à ces tableaux se font par exemple grâce à une instruction du type $\pi^{k\%2}$ ou $k\%2$ signifie k modulo 2.*

10.5 Algorithme de Dikstra

L'algorithme de Dijkstra permet de trouver le plus court chemin dans un réseau (E, Γ, l) sans cycle absorbant. Dans un tel réseau on a l'équivalence suivante :

$$\forall i, x \in E^2 \quad \exists \quad \text{un chemin de } i \text{ a } x \Leftrightarrow \quad \exists \quad \text{un plus court chemin de } i \text{ a } x$$

L'algorithme de Dijkstra nécessite également que tous les arcs soient positifs ou nuls.

Le principe de l'algorithme est le suivant : Soit $R = (E, \vec{\Gamma}, l)$, pour tout $u \in \vec{\Gamma}$, $l(u) \geq 0$. Soit $i \in E$ sommet initial de la recherche. S désigne l'ensemble des sommets x de E pour lesquelles la longueur d'un plus court chemin de i à x a été calculé.
Initalement $S = \{i\}$. $\pi(i) = 0$.

On appelle $S-$chemin un chemin partant de i et ayant tous ses sommets dans S sauf le dernier.
Soit $Y = E \backslash S$, $\forall y \in Y$, $\pi'(y)$ est la longueur d'un plus court $S-$chemin de i à y. On sélectionne y^* dans Y tel que $\pi'(y^*) = \min_{y \in Y} \pi'(y)$.

10.5.1 Algorithme en pseudo language

Données : $E, \Gamma, l, i \in E$
Résultat : π, S

1: $S = \{i\}$, $\pi(i) = 0$, $k = 1$, $x_1 = i$
2: **for all** $x \in E \backslash \{i\}$ **do**
3: $\quad \pi(x) = \infty$
4: **end for**
5: **while** $k < n$ et $\pi(x_k) < \infty$ **do**
6: $\quad$ **for all** $y \in \Gamma(x_k)$ tel que $y \notin S$ **do**
7: $\quad\quad \pi(y) = \min[\pi(y), \pi(x_k) + l(x_k, y)]$
8: $\quad$ **end for**
9: $\quad$ Extraire un sommet $x \notin S$ tel que $\pi(x) = \min[\pi(y) \forall y \notin S]$
10: $\quad k = k + 1$, $x_k = x$, $S = S \cup \{x_k\}$.
11: **end while**

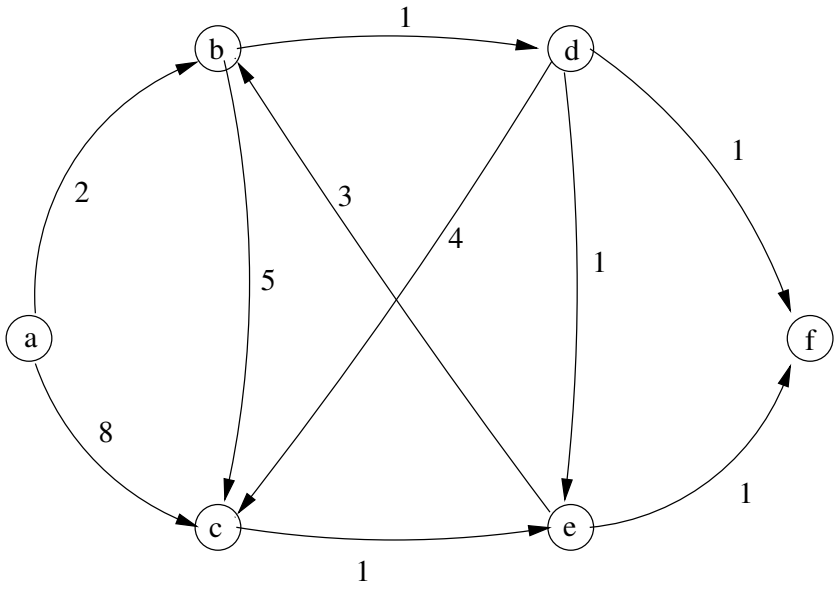
Complexité

Nous prendrons S sous la forme d'un tableau de booléen
- Les boucles d'initialisation des ligne 1 et 2 à 4 sont en $O(n)$
- Le corps de la boucle principale (ligne 5 à 11) est executé n fois.

La complexité de l'algorithme de DIJSKTRA est en $O(n^2)$.

10.5.2 Exemple d'éxecution

Exemple i:



Construisons S pas à pas pour l'algorithme de DIJKSTRA avec $i = a$.

S	k	x	y
$\{a\}$	1	a	b, c
$\{a, b\}$	2	b	c, d
$\{a, b, d\}$	3	d	c, e, f
$\{a, b, d, f\}$	4	f	/
$\{a, b, c, d, f\}$	5	c	e
$\{a, b, c, d, e, f\}$	6	e	/

A la fin de l'algo, les valeurs des coûts des plus courts chemins sont :

Sommet	a	b	c	d	e	f
Coût	0	2	5	3	7	4

10.6 Plus courts chemins (Exploration largeur)

Nous nous plaçons dans le cas d'un réseau $R = (E, \Gamma, l)$ tel que pour tout arc u dans $\vec{\Gamma}$, l'application de pondération soit égale à une constante strictement positive. On prendra $l(u) = c = 1$.

Pour trouver la longueur du plus court chemin d'un sommet i à tout autre sommet x (s'il existe), il existe un algorithme en temps linéaire $O(n+m)$ se basant sur la propriété de constance de l'application de pondération. C'est l'algorithme dit *d'exploration-largeur*.

Il consiste à parcourir le graphe "par couches" successives à partir de i et de proche en proche selon ses sucesseurs. La longueur du plus court chemin à x est alors le nombre minimal de *couches* parcourues pour aller de i à x (multiplié par la constante c).

10.6.1 Algorithme en pseudo langage

On prendra pour cet algorithme $l(u) = 1 \forall u \in \vec{\Gamma}$
Données : $E, \Gamma, i \in E$
Résultat : π, S

1: $\pi(i) = 0$, $S = \{i\}$, $T = 0$, $k = 1$
2: **for all** $x \in E$ **do**
3: $\quad \pi(x) = \infty$
4: **end for**
5: **while** $S \neq \emptyset$ **do**
6: $\quad$ **for all** $x \in S$ **do**
7: $\quad\quad$ **for all** $y \in \Gamma(x)$ tel que $\pi(y) = \infty$ **do**
8: $\quad\quad\quad \pi(y) = k$
9: $\quad\quad\quad T = T \cup \{y\}$
10: $\quad\quad$ **end for**
11: $\quad$ **end for**
12: $\quad S = T$, $T = 0$
13: $\quad k = k + 1$
14: **end while**

10.7 Graphes sans circuits : GSC

10.7.1 Définition

Un graphe sans circuit $(E, \vec{\Gamma})$ (GSC) est un graphe ne présentant aucun cycle. Si $G = (E, \vec{\Gamma})$ est un graphe sans circuit alors son symétrique (E, Γ^{-1}) l'est aussi. Un graphe $G = (E, \Gamma)$ est un graphe sans circuit équivaut à dire que toutes les composantes fortement connexes de G sont réduites à un seul point. C'est la caractèrisation des GSC.

10.7.2 Source et puit

Soit $G = (E, \vec{\Gamma})$ un graphe sans circuit et $x \in E$.
x est une *source* de G si $d^-(x) = |\Gamma^{-1}(x)| = 0$
x est un *puits* de G si $d^+(x) = |\Gamma^(x)| = 0$
Tout GSC possède au moins une source et un puits.

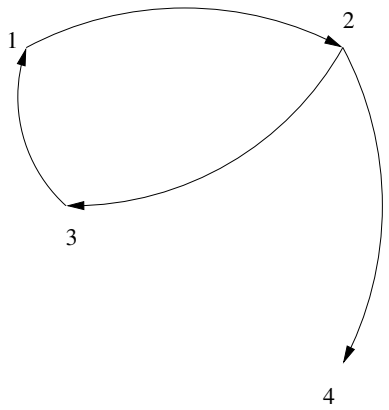
10.7.3 Rang et hauteur

Soit $G = (E, \Gamma)$ un graphe **quelconque**, et x un sommet de E. Le rang du sommet x, noté $r(x)$ est la longueur maximale d'un chemin **élémentaire** se terminant par x.

Soit $G = (E, \Gamma)$ un graphe **quelconque**, et x un sommet de E. Le hauteur du sommet x, noté $h(x)$ est la longueur maximale d'un chemin **élémentaire** débutant x.

Exemple i:

Soit un graphe G tel que $E = \{1, 2, 3, 4\}$ défini tel que :



Alors le rang du sommet 2 est 3, on peut trouver un chemin élémentaire de longueur 3. Par exemple $2, 3, 1, 2$.

La hauteur du sommet 3 est 3 aussi, il existe un chemin de longueur 3 débutant par $3 : 3, 1, 2, 4$.

$r(x) = 0$ équivaut à dire que x est un source. De même, si $h(x) = 0$, x est un puits.

10.7.4 Partition en niveaux

Soit $G = (E, \Gamma)$ un graphe quelconque. G admet une *partition en niveaux* s'il existe une partition de E telle que :

$$E = \bigcup_{i=0}^{p} E_i$$

Avec E_0 l'ensemble des sources de G, et $\forall i > 0$, E_i l'ensemble des sources de $(E \backslash E_{i-1}, \vec{\Gamma}_i)$ où $\vec{\Gamma}_i$ est la restriction de G à $E \backslash E_{i-1}$ c'est à dire l'ensemble des arcs de $\vec{\Gamma}$ ayant leurs 2 extremités dans $E \backslash E_{i-1}$.
On remarquera que $E_i = E \backslash E_0 \cup E_1 \cup \cdots \cup E_{i-1}$.

Un graphe sans circuit admet une partition en niveaux unique et telle que pour tout sommet x appartenant à E_i, $r(x) = i$.

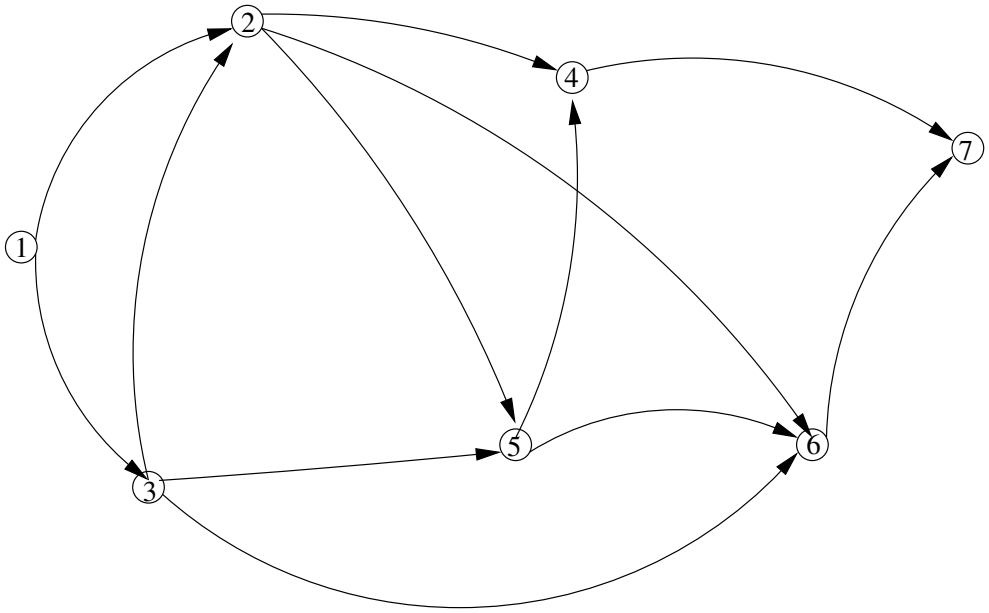
Algorithme circuit-niveaux

Voici un algorithme qui permet de déterminer la partition d'un graphe $G = (E, \Gamma)$.
Données : E, Γ, Γ^{-1}
Résultat : $E_i \subset E, CIRC$

1: $E_0 = \emptyset, N = 0, i = 0$
2: **for all** $x \in E$ **do**
3: $d^-(x) = |\Gamma^{-1}(x)|$
4: **end for**
5: **for all** $x \in E$ tel que $d^-(x) = 0$ **do**
6: $E_0 = E_0 \cup \{x\}$
7: $N = N + 1$.
8: **end for**
9: **while** $N < n$ et $E_i \neq \emptyset$ **do**
10: $E_{i+1} = \emptyset$.
11: **for all** $x \in E$ **do**
12: **for all** $y \in \Gamma(x)$ **do**
13: $d^-(x) = d^-(y) - 1$
14: **if** $d^-(y) = 0$ **then**
15: $E_{i+1} = E_{i+1} \cup \{y\}$
16: $N = N + 1$
17: **end if**
18: **end for**
19: **end for**
20: $i = i + 1$
21: **end while**
22: **if** $N < n$ **then**
23: $CIRC = VRAI$
24: **else**
25: $CIRC = FAUX$
26: **end if**

Exemple

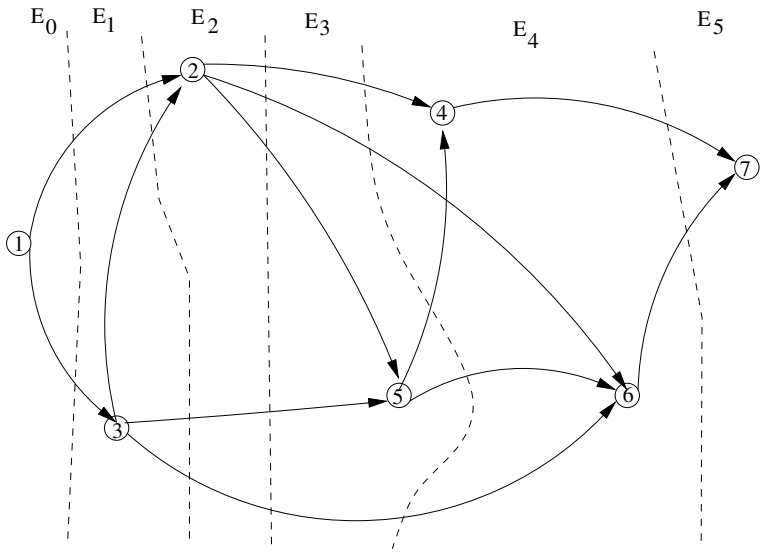
Exemple ii:



Pour chacun des sommets voici le degré intérieur $d^-(x)$:

x	1	2	3	4	5	6	7	Ensemble E
$d^-(x)$	0	2	1	2	2	3	2	
	1	0						$E_0 = \{1\}$
	0				1	2		$E_1 = \{3\}$
				1	0	1	1	$E_2 = \{2\}$
				0		0		$E_3 = \{5\}$
							0	$E_4 = \{4, 6\}$
								$E_5 = \{7\}$

Le graphe ainsi partitionné est :



10.7.5 Application à l'ordonnancement

Soit un **objectif** à atteindre dont la réalisation suppose l'éxécution de **tâches** soumises à des **contraintes** (notamment de précédence : la tâche N suppose que la tache M soit terminée pour pouvoir être réalisée).
Pour chaque tâche, on a une estimation de la durée. Le problème est de déterminer l'ordre et l'éxécution des tâches afin de minimiser le temps global du processus.

Chapitre 11

Cycles eulériens et hamiltoniens

11.1 Cycle eulérien

11.1.1 Définition

Soit $G = (E, \vec{\Gamma})$ un graphe non-orienté. Un *cycle eulérien* dans G est un cycle passant une et une seule fois par chaque arête de G. Un graphe est dit *eulérien* s'il possède un cycle eulérien.

11.1.2 Conditions pour qu'un graphe soit eulérien

Les conditions nécessaires pour qu'un graphe $G = (E, \vec{\Gamma})$ soit eulérien sont :
- $|\vec{\Gamma}| \geq 3$
- G doit être connexe
- Pour tout sommet x de E, le nombre $|\Gamma(x) \cup \Gamma^{-1}(x)| = d^+(x) + d^-(x)$ doit être pair

Ces conditions sont chacunes nécessaires, mais elle ne sont pas suffisantes individuellement.

11.1.3 Graphe semi-eulérien

Un chemin *semi-eulérien* est un chemin qui passe une et une seule fois par chacune des arêtes du graphe.

11.2 Cycle hamiltonien

11.2.1 Définition

Un cycle est dit *Hamiltonien* s'il passe une et une seule fois par tous les sommets du graphes. Un graphe est *Hamiltonien* s'il possède un cycle du même

nom.
Il n'y a pas de caractèrisation simple des graphes *Hamiltonien*, néanmoins on peut énoncer certaines propriétés :

- un graphe possèdant un sommet de degré 1 ne peut être hamiltonien
- si un sommet est de degré 2, alors les deux arêtes incidentes à ce sommet font partie du cycle hamiltonien
- les graphes complets K_n sont hamiltoniens.
- **Corrolaire de Dirac** Si $G = (E, \Gamma)$ est un graphe simple tel que $|E| \geq 3$ et si pour tout sommet $x \in E$ on a $d(x) \geq \frac{n}{2}$, alors G est Hamiltonien.

Chapitre 12

Flots et réseaux de transport

12.1 Modélisation d'un réseau de transport

Un réseau de transport est constitué par :
- Un graphe $(E, \vec{\Gamma})$,
- Un puits $p \in E$,
- Une source $s \in E \backslash \{p\}$,
- Une application c de $\vec{\Gamma}$ dans $\mathbb{R}^+$ et qui peut prendre comme valeur ∞. Pour tout u dans $\vec{\Gamma}$, $c(u)$ est appelé *capacité* de u. Typiquement $c(u)$ représente le débit maximal pouvant transiter par u.

12.1.1 Modélisation du traffic

Soit f une application $\vec{\Gamma} \to \mathbb{R}^+$. $f(u)$ représente le débit sur l'arc u.
On définit également les applications f^+ et f^- sur E définies par :

$$f^+(x) = \sum_{y \in \Gamma(x)} f\left(\{x, y\}\right), \quad \forall y \in \Gamma(x)$$

$$f^-(x) = \sum_{y \in \Gamma^{-1}(x)} f\left(\{y, x\}\right)$$

Les applications f^+ et f^- sont appelées respectivement *flot sortant* et *flot entrant* de x.

12.2 Propriétés d'un réseau de transport

12.2.1 Equilibre global

La propriété d'équilibre global énonce que la somme des flots entrants est égale à la somme des flots sortants, pour tous les sommets x de E, soit :

$$\sum_{x \in E} f^+(x) = \sum_{x \in E} f^-(x)$$

12.2.2 Equilibre local

La propriété d'équilibre global énonce que pour tout sommet x de E, le flot entrant dans x et égal au flot sortant de x, soit :

$$\forall x \in E \quad f^+(x) = f^-(x)$$

La fonction f est un **flot** sur $(E, \vec{\Gamma})$ si elle vérifie les deux propriétés d'équilibre.

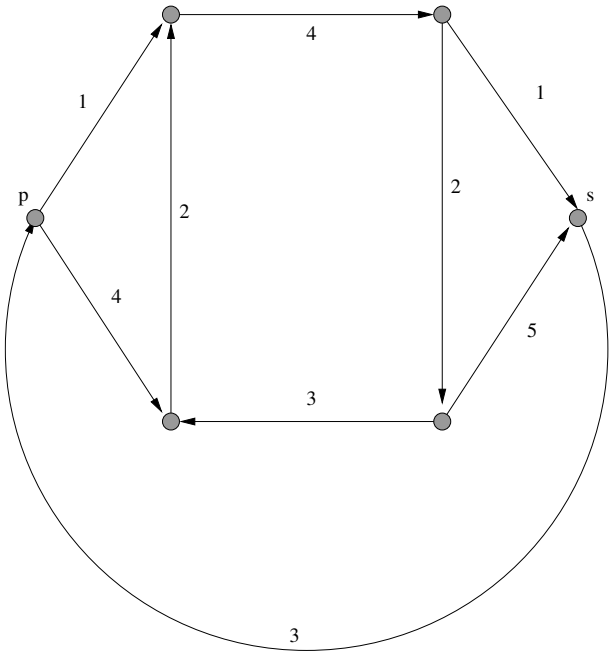
12.2.3 Arc de retour

Soir $R = (E, \vec{\Gamma}, p, s, c)$ un réseau de transport et f un flot sur $(E, \vec{\Gamma} \cup \{(p, s)\})$. L'arc (p, s) est appelé **arc de retour**. On appelle la quantité $f(\{p, s\})$ est appelé valeur du flot. La valeur du flot de retour est égale à la valeur du flot maximal que l'on peut passer dans le réseau du puits jusqu'à la source.

Le flot f est compatible avec le réseau R si pour chaque arc, le flot passant par cette arc est inférieure ou égal à la capacité de cet arc. :

$$\forall u \in \vec{\Gamma} \quad f(u) \leq c(u)$$

Exemple i:



Le flot maximal de p à s compatible avec R est égal à 3.

Soit R un réseau de transport. Le problème du flot maximum est de trouver un flot f de valeur maximale et qui soit compatible avec R.

12.3 Algorithme de Ford et Fulkerson

Cet algorithme a pour but de trouver le flot maximal dans un réseau R. L'algorithme de Ford et Fulkerson procède par améliorations succesives du flot f, en partant d'un flot compatible. Le flot nul par exemple est compatible et peut servir comme flot initial. Pour améliorer un flot compatible, on trouve un chemin de s à p.

Un arc $u \in \vec{\Gamma}$ est dit **saturé** si $f(u) = c(u)$. On ne peut plus augmenter le flot passant sur cet arc. Un flot est dit **complet** si tout chemin de s à p passe au moins par un arc saturé.

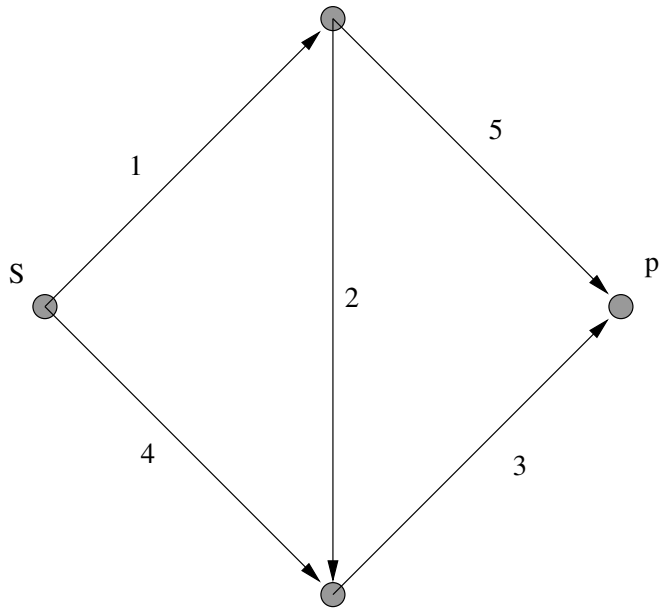
Supposons qu'il existe un chemin π de s à p tel qu'aucun arc de π n'est saturé. On appelle $r(\pi)$ la **valeur résiduelle** de π le nombre :

$$r(\pi) = \min_{u \in \pi} (c(u) - f(u))$$

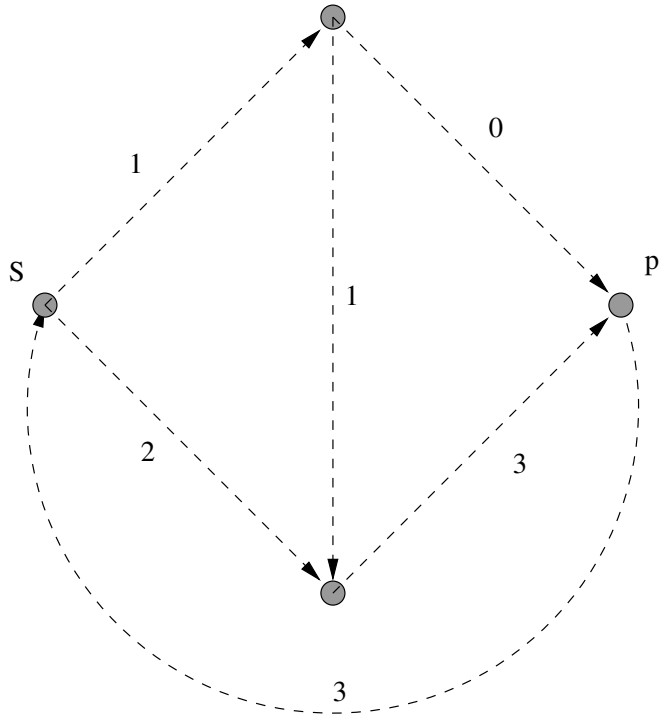
Un flot f sur un réseau R peut être amélioré de $r(\pi)$ (on augmente f de $r(\pi)$ sur tous les arcs de π et sur l'arc de retour $\{p, s\}$).

Exemple i:

Voici un réseau R et ses capacités :



Le flot associé à ce réseau est représenté par :



Ce flot est complet (tous les chemins de s à p ont une valeur résiduelle nulle) mais n'est pas maximal. Il existe un flot maximal tel que $f(\{p, s\})$ soit supérieur à 3.

12.3.1 Réseau d'écart

L'algorithme de FORD et FULKERSON introduit le *réseau d'écart* pour améliorer le flot courant. Le réseau d'écart associé à R est définit par :

$$\overline{R} = (E, \overline{\Gamma}, \overline{c})$$

Pour tout arc $u = (x, y) \in \vec{\Gamma}$, on associe au plus 2 arcs :

- $u^+ = (x, y)$ et tel que $\overline{c}(u^+) = c(u) - f(u)$. C'est **l'arc direct**. Sa capacité dans le flot d'écart est égale à sa capacité originale moins la valeur du flot passant par cet arc (c'est à dire la capacité résiduelle).
- $u^- = (y, x)$ **l'arc rétrograde**, si $f(u) > 0$. Dans ce cas $c(u^-) = f(u)$.

Soit $\overline{\pi}$ un chemin de s à p dans le réseau d'écart $\overline{R}$. On appelle capacité résiduelle de $\overline{\pi}$ le minimum des capacités résiduelles $\overline{c}(u)$ pour tous les arcs de u de $\overline{\pi}$. Tous chemin de s à p dans le réseau d'écart fournit un moyen d'augmenter le flot.

12.3.2 Algorithme

Initialisation

Soit $k = 0$, et soit f_0 un flot compatible avec R (par exemple le flot nul).

Recherche d'un moyen d'améliorer le flot courant f_k :

Soit f_k le flot courant. Soit $\overline{R_k} = (E, \overline{\Gamma}, \overline{c})$ le réseau d'écart associé à R et f_k. On recherche un chemin $\overline{\pi_k}$ de s à p dans le réseau d'écart $\overline{R}$. S'il n'existe pas de chemin, alors l'algorithme s'arrête et f_k est un flot maximal.

Amélioration du flot courant

Soit ϵ_k la capacité résiduelle de $\overline{\pi_k}$ $(= \min_{u \in \overline{\pi_k}}(c(u))$.
$\forall u \in \overline{\pi_k}$, on applique les règles suivantes

- Si u est un arc direct, alors $f_{k+1}(u) = f_k(u) + \epsilon_k$,
- Si u est un arc rétrograde, alors $f_{k+1}(u) = f_k(u) - \epsilon_k$,
- Si u est l'arc de retour direct $\{p, s\}$, alors $f_{k+1}(u) = f_k(u) + \epsilon_k$.

$k = k + 1$, et l'on recommence à l'étape 2 (recherche d'un chemin).

12.3.3 Preuve de l'algorithme de Ford et Fulkerson

Soit $G = (E, \vec{\Gamma})$ et $F \subset E$. On note $\vec{\sigma}^+(F)$ l'ensemble des arcs *sortants* de F.

$$\vec{\sigma}^+(F) = \{(x, y) \in E \times E \ / \ x \in F \ / \ y \notin F\}$$

$$\vec{\sigma}^-(F) = \{(x, y) \in E \times E \ / \ x \notin F \ / \ y \in F\}$$

Par conséquent $\vec{\sigma}^+(F) = \vec{\sigma}^-(E \backslash F)$.

Soit $R = (E, \vec{\Gamma}, p, s, c)$ un réseau de transport. On appelle **coupe** sur R un ensemble d'arcs $\vec{S} \subset \vec{\Gamma}$ de la forme $S = \vec{\sigma}(F)$ avec $F \subset E$, $s \in F$, $p \notin F$.

On appelle capacité de la coupe $\vec{S}$ la quantité $c(\vec{S}) = \sum_{u \in \vec{S}} c(u)$. Pour tout flot compatible avec R et pour toute coupe $\vec{S}$ sur R, on a :

$$f(\{p, s\}) \leq c(\vec{S})$$

A la fin de l'algorithme de Ford et Fulkerson, le flot courant est maximal. En effet quand l'algorithme se termine, il n'existe pas de chemin dans le réseau d'écart.

Chapitre 13

Résolutions de problèmes en Intelligence artificielle et optimisation

13.1 Exemple : le problème des 8 dames

On se place dans le cas d'un échiquier standart (de taille 8×8). Le problème est de placer 8 dames sur cet échiquier telle qu'aucune dame ne peut se faire prendre par une autre. (Une dame peut kidnapper toutes les dames se trouvant sur les lignes horizontales, verticales et diagonales).
La figure 13.1 page suivante présente le placement de 2 dames sur un échiquier. On voit que le nombre de cases disponibles pour le placement de 6 autre dames est très restreint. Deux approches sont possible pour le placement des huit dames :

1. Recherche aveugle : On place les dames sur des cases choisies aléatoirement parmis les cases libres. A un moment, il n'y aura plus de cases libres pour le placement des dames, il y aura alors remise en cause d'un ou des choix faits précédèmment et retour arrière.
2. Recherche heuristique : On place les dames sur des cases de telle manière que le nombre de cases "hachurées" (susceptibles d'être prises par cette dame) soit minimal.

13.1.1 Graphe de résolution de problème GRP

C'est un graphe $G = (E, \vec{\Gamma})$ tel que les sommets du graphe sont les états possible du problèmes. Il y a un arc $u = (i, j)$ dans le graphe si une règle permet de passer de l'état i à l'état j. On associe un coût $c(u)$ à cet arc. On doit distinguer le sommet initial et les sommets finaux, le but du problème.

Un graphe de résolution de problème (**GRP**) est un quintuplet $(S, \vec{\nabla}, i, B, c)$:
– S est l'ensemble des sommets du graphe (ou états du problème),

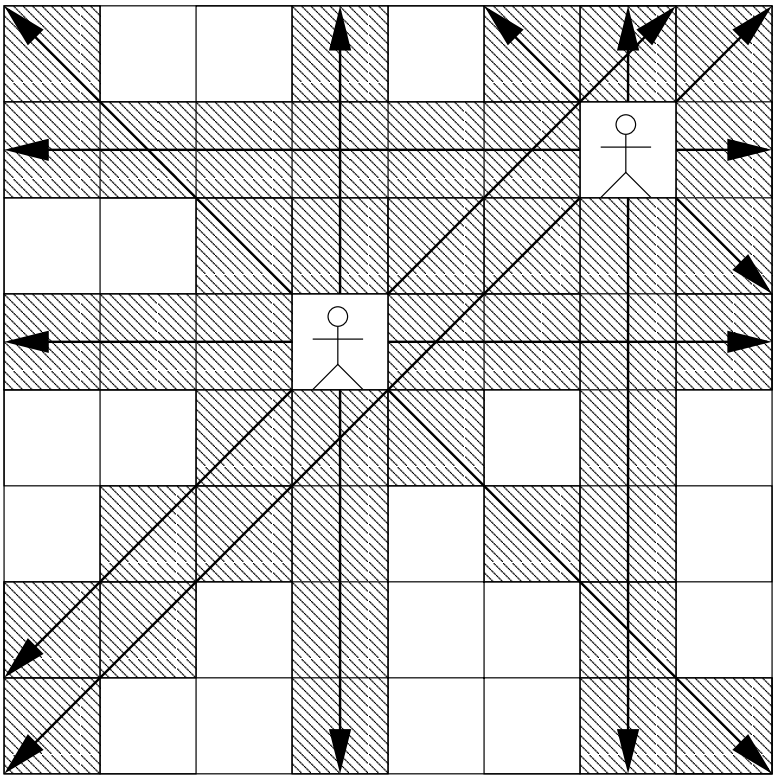


FIG. 13.1 – *Exemple de placement pour le jeu des 8 dames*

- $\vec{\nabla}$ l'ensemble des arcs,
- $i \in S$ le sommet initial (état inital du problème),
- $B \subset S$ l'ensemble des sommets buts,
- c le coût de l'application de chacune des règles.

Un graphe de résolution de problème est énorme et non-unique. Ainsi le but de la recherche heuristique est d'explorer partiellement le graphe pour aboutir à une solution (c'est à dire trouver le plus court chemin de i à un sommet $s \in B$).

13.2 Stratégies de recherche

13.2.1 Exemple du jeu de taquin

Le jeu du *taquin* est un jeu de plateau dans lequel il faut passer d'une configuration de départ à une configuration d'arrivée sachant que les cases ne peuvent se déplacer que sur une case vide immédiatement adjacente.
Les contraintes étant :

- Passer de la configuration de départ à la configuration d'arrivée
- Minimiser le nombre de déplacement

Configuration de départ Configuration d'arrivée

FIG. 13.2 – *Jeu de taquin*

Le graphe de résolution de problème associé à un tel jeu sera tel que :

- Les sommets du GRP seront les états possibles du jeu
- Le sommet initial sera la configuration de départ
- Le sommet but sera la configuration d'arrivée
- Il existe un arc de la configuration x à la configuration y si un mouvement autorisé permet de passer de x à y.
- Un tel GRP possède à la fois des **cycles simples** : il peut exister 2 chemins différents pour passer d'un sommet x à un sommet y ; et des **cycles circuits** : les règles permettent aussi de passer d'une configuration x à y puis de revenir à x.

13.2.2 Stratègie sans mémoire : la recherche arrière (backtrack)

Cette stratègie consiste à explorer le GRP via un chemin issu de d jusqu'à atteindre

- le but (c'est gagné)
- un puits (c'est perdu)
- une profondeur limite fixée d'avance

Si le but est non atteint, on revient au dernier choix effectué chronologiquement (en *oubliant* la partie venant d'être explorée). Pour eviter les circuits on mémorise les états du graphe se trouvant sur le chemin courant. Tout nouvel état atteint est comparé à la liste des états du chemin.
Cette stratègie pose toutefois des problèmes : si le but ne peut être atteint dans la limite de profondeur fixée, il faut augmenter celle-ci et tout recommencer.

13.2.3 Stratègie avec mémoire : la recherche avec graphe (RAG)

On a un GRP qui est défini implicitement, en général trop grand pour être representé en mémoire et être exploré en totalité.
On va développer (c'est à dire représenter explicitement) une partie du GRP suffisante pour trouver le sommet but. On éviter ainsi les explorations redondantes.

Algorithme

1. **Créer un graphe de recherche** G qui consiste uniquement en un noeud de départ d. Mettre d sur une liste appelée $OUVERT$
2. **Créer une liste appelée** $FERMEE$ qui est initialement vide.
3. **BOUCLE**, si $OUVERT$ est non-vide, sinon échec
4. **Sélectionner le premier noeud d'**$OUVERT$, l'enlever d'$OUVERT$, et le mettre dans $FERME$, Appeler ce noeud n.
5. **Si** n **est un noeud but, terminer la procédure** avec succès. Renvoyer la solution obtenue en retraçant le chemin le long des pointeurs de n jusqu'à d dans G. Ajoutons que les pointeurs seront construits en phase 7
6. **Développer le noeud** n, produisant l'ensemble M de ses successeurs et les mémoriser comme successeurs de n dans G.
7. **Mémoriser un pointeur vers** n **à partir des éléments de** M qui n'étaient pas déjà dans G (c'est à dire pas déjà dans $OUVERT$ ou $FERME$). Ajouter ces éléments de M à $OUVERT$. Pour chaque élément de M qui était déjà dans $OUVERT$ ou $FERME$, décider si l'on redirige ou non le pointeur vers n (voir ci-dessous). Pour chaque membre de M déjà dans $FERME$, décider pour chacun de ses descendants dans G si l'on redirige ou non le pointeur.
8. **Réordonner la liste** $OUVERT$, soit arbitrairement, soit selon des heuristiques.
9. **Aller à BOUCLE**

Les heuristiques utilisés à l'étape 8 peuvent être la profondeur (pour des parcours en profondeur ou en largeur), ou des stratègies adaptées au problème.

13.3 Algorithme A et A^*

Dans la recherche avec graphe (RAG) précèdemment vue, le choix de la fonction d'évaluation f pour le tri de $OUVERT$ s'avère être un point crucial. Dans le cas de l'algorithme A, on choisit f telle que pour tout sommet n du GRP, $f(n)$ estime le coût d'un chemin optimal de d à un but passant par n.
f peut se mettre sous la forme :

$$f(n) = g(n) + h(n)$$

- $g(n)$ estime le coût d'un chemin optimal de d à n.
- $h(n)$ estime le coût d'un chemin optimal de n à un but.

Posons $g^*(n)$ le coût d'un chemin optimal dans le GRP de d à n et $h^*(n)$ le coût d'un chemin optimal de n à un but. On a $f^*(n) = g^*(n) + h^*(n)$. On veut que f estime (au mieux) f^*.

Pour $g(n)$ il est naturel de prendre le coût de d à n dans le graphe de recherche (partie du GRP déjà developpé). Pour $h(n)$ on s'appuie sur l'information spécifique au problème. On nomme h fonction heuristique.

Si $\forall n, h(n) \leq h^*(n)$, alors l'algo A trouvera un chemin optimal de d à un but (s'il en existe). Dans ce cas, on parle **d'algorithme** A^*. Dans le cas particulier ou $h = 0$, on a bien un algorithme A^* mais qui sera plus long.

Quatrième partie

Annexes

Annexe A

TP1 - Tri par tas

A.1 Introduction

Ce premier TP du cours d'algorithmique nous a fait travailler sur un algorithme de tri : le tri par tas (ou HEAPSORT).
Nous avons vu en cours différents algorithmes de tri : le QUICKSORT, SLOWSORT et le BUBBLESORT en première année. Un des grands problèmes qui se posent dans l'étude des algorithmes est lévaluation de la complexité des programmes. Au delà des caractèristiques des ordinateurs, la complexité permet d'évaluer directement l'évolution de la durée déxecution des programmes.

Voici donc le rapport en LaTeX qui présentera le fonctionnement de l'algorithme HEAPSORT et le comparera à d'autres algorithmes de tri.

A.2 Equations Booléennes

A.2.1 feuille

Description

Une feuille est un nœud de l'arbre qui n'a pas de fils.

Code

```
1   int feuille(int i, int k)
2   {
3       return (2 * (i+1) > k);
4   }
```

A.2.2 interne1

Description

La fonction **interne1()** renvoie *true* lorsque le nœud n'a qu'un fils. Dans un arbre binaire presque parfait, un seul nœud au maximum répond à ce critère.

Code

```
1   int interne1(int i, int k)
2   {
3       return (2 * (i+1) == k);
4   }
```

A.2.3 interne2

Description

La fonction **interne2()** renvoie *true* lorsque le nœud a 2 fils non-vides. On peut alors définir la relation booléenne suivante :

$$interne1 \bigcup interne2 \bigcup feuille = \Omega$$

Où Ω designe l'ensemble des cas possibles.

Code

```
1   int interne2(int i, int k)
2   {
3       return (2 * (i+1) < k);
4   }
```

A.2.4 Dominance

A.2.5 Description

La fonction booléenne *d(T,i,k)* peut se construire grâce aux 3 fonctions que nous venons de définir. Elle répond à l'équation suivante

$$d(T,i,k) \equiv feuille(i,k)$$

$$\vee (interne1(i,k) \wedge T[i] \geq T[2*(i+1)-1])$$

$$\vee (interne2(i,k) \wedge T[i] \geq T[2*(i+1)] \wedge T[i] \geq T[2*(i+1)-1])$$

Code

```
1   bool d(int *T, int i, int k)
2   {
3       if (feuille(i,k)) return TRUE;
4       else if (interne1(i,k)) return (T[i] >= T[2*(i+1) - 1]);
5       else return ((T[i] >= T[2*(i+1)-1]) && (T[i] >= T[2*(i+1)]));
6   }
```

A.2.6 Temps de calcul

Les trois fonctions **interne1()**, **interne2()** et **feuille()** s'éxecutent en O(1) quelques soient les arguments. De plus ces fonctions peuvent se décomposer en primitive rapides :

- Multiplication par 2 (*Décalage binaire d'un bit vers la droite*) ;
- Addition d'un bit ;
- Comparaison.

Ainsi, la fonction **d(T,i,k)** s'éxecute en temps constant car elle est basée sur les mêmes primitives.

A.3 Tas

Dans cette section, nous nous attacherons à la réalisation des fonctions qui permettent de transformer un tableau d'entier en un tas (c'est à dire un arbre binaire presque parfait dont tout nœud est dominant). Pour cela, nous allons mettre au point une procédure **rétablirTas(T,j,k)** qui permettra l'installation d'un tas de racine j.

A.3.1 Fonction d'échange

Nous allons programmer une fonction **swap** qui permet d'échanger 2 élements d'un tablau.

```
1   void swap(int *T, int a, int b)
2   {
3       int tmp;
4
5       tmp = T[a];
6       T[a] = T[b]
7       T[b] = tmp;
8   }
```

A.3.2 rétablirTas

Description

Invariant : $I(j) \equiv (d(T,j,k) = 0) \wedge (\forall\ f\ \epsilon F(j)$ racineTas(i,j)

Initialisation : $j = i$

Condition d'arrêt : $d(T,j,k)$

Implications : $I(j) \wedge (\exists\ M\ \epsilon\ F(j)$ tel que $v(M) = max(v(m)\ \forall\ m\ \epsilon\ F(j))$
$\wedge\ (T[j] = t_M) \wedge (T[M] = t_j) \Rightarrow$ I(M)

Code

```
1   void retablirTas(int *T, int i, int k)
2   {
3       if (!d(T,i,k))
4           if ((T[2*(i+1)-1] > T[2*(i+1)]) || interne1(i,k))
5           {
6               swap(T,i, 2*(i+1)-1);
7               retablirTas(T,2*(i+1)-1,k);
8           } else
9           {
10              swap(T,i, 2*(i+1));
11              retablirTas(T,2*(i+1),k);
12          }
13  }
```

Complexité

Pour établir la complexité de la procédure **rétablirTas**, regardons pour chaque valeur de i le temps mis.
Appelons :

- A le temps déxecution de la procédure **swap** (temps constant $\forall$ les données)
- B le temps dévaluation des 2 expressions booléennes dans les *if* (constant lui aussi car **d(T,i,k)** est en temps constant).

Ainsi :
$T_{retablirTas}(1) \leq$ B
$T_{retablirTas}(2) \leq$ A + B + $T_{retablirTas}(0)$
...
D'une manière génerale, $T_{retablirTas}(2^n) =$ A + B + $T_{retablirTas}(2^{n-1})$
On peut donc dire que que **rétablirTas(n)** est en $\Theta(log_2(n))$

Exemples d'éxecution

Prenons un exemple de tableau à 6 valeurs :
T[0...10]=[17,2,10,4,1,5] **rétablirTas(T,1,5)** T[0...10]=[17,4,10,2,1,5]

A.3.3 FaireTas

Description

La procédure **FaireTas(T,n)** sert à établir un tas $T[0...n-1]$. Cette procédure se crée simplement en procédant à des appels à **rétablirTas** sur tous les élements du tableau T en partant de la fin.

Code

```
1   void FaireTas(int *T, int n)
2   {
3       int i;
4
5       for (i = n-1; i>= 0; i--) retablirTas(T,i,n);
6   }
```

Complexité

La complexité de la procédure **retablirTas** a déjà été établie en $\Theta(log_2(n))$. La procédure **FaireTas** faisant exactement n appels à **retablirTas**, on peut donc affirmer que la complexité de **FaireTas** est en $\underline{O(n * log_2(n))}$

A.3.4 Procédure triParTas

Description

Grâce aux fonctions précedemment définis **FaireTas()** et **retablirTas()**, nous pouvons crér la procédure **TriParTas** selon l'invariant suivant :

Invariant : $I(k) \equiv (T(0...k-1)]$ tas$) \wedge (T[0...k-1] \leq T[k...n-1]) \wedge (T[0...k-1] \nearrow)$

Initialisation : $k = n$

Condition d'arrêt : $k = 0$

Implications : $I(k) \wedge (T[k] = t_k) \wedge (T[0] = t_k) \wedge (T[0...k-1] \Rightarrow I(k-1)$

Code

```
1   void triParTas(int *T, int n)
2   {
3       int k = n;
4       FaireTas(T,n) // I(n)
5
6       while (k != 0) // I(k) && (k != 0)
7       {
8           swap(T, k-1, 0);
9           retablirTas(T, 0, k-1); // I(k-1);
10          k--; // I(k)
11      }
12  }
```

Complexité

Les instructions **swap** et **k−−;** étant en temps constant, la complexité du corps de boucle est la même que celle de la procédure **retablirTas**, donc en $O(log_2(n))$. L'initialisation de la procédure est quand à elle de complexité égale à **FaireTas**, donc $O(n * log_2(n))$ Par conséquent, la complexité de ce tri est en $O(n * log_2(n))$. Or tout tri d'éléments non-ordonnés dans un tableau est nécessairement en $\Omega(n * log_2(n))$. Ce qui nous amène à conclure que ce tri est en $\Theta(n * log_2(n))$.

Exemples d'éxecution

Voici quelques exemples déxécution pour des tableaux aléatoires :
[2 53 10 4 48 12 8 14 22 31] ⇒ [2 4 8 10 12 14 22 31 48 53]
[50 14 36 9 2 8 5 1 67 884] ⇒ [1 2 5 8 9 14 36 50 67 884]

A.3.5 Temps de calcul

Méthodologie de mesure

Pour mesurer le temps de calcul, j'ai utilisé une fonctionnalité particulière des processeurs Intel. En effet, tous les processeurs Pentium possèdent un compteur interne cadencé à la fréquence du processeur. Pour y accèder, une seule instruction assembleur suffit

```
1   inline unsigned long long int GetTick()
2   {
3       unsigned long long int x;
4       __asm__ volatile ("RDTSC" : "=A" (x));
5       return x;
6   }
```

Le format utilisé est un *long long int*, soit un entier codé sur 64 bits. Il n'y aucun risque de dépassement de capacité car un tel compteur peut prendre $2^{64} - 1$ valeurs, soit une durée de vie totale d'environ 584 années pour un processeur cadencé À 1 GHz. Pour mesurer le temps d'éxécution du tri, il suffit de faire un appel a la fonction **GetTick** juste avant et juste après le tri, puis de diviser la diffèrence par la fréquence annoncée du processeur
Les mesures sont effectuées sur un PC Pentium III à 850 Mhz sous linux. Pour minimiser l'impact des fonctions non-opératives (entrés/sorties, interruptions, scheduling), les mesures se font sur des procédures sans affichage.

Résultats

Les temps de tri sont en microsecondes (10^{-6} seconde). On a effectué 2 séries de mesures, une pour des suites de nombres aléatoires desordonnés, et une pour des séries de nombres déjà ordonnés.

Taille du problème	Tableau desordonné	Tableau ordonné
10^1	4 μs	2 μs
10^2	32 μs	29 μs
10^3	470 μs	461 μs
10^4	6766 μs	6281 μs
10^5	98981 μs	87360 μs
10^6	2057165 μs	1106381 μs

Pour des problèmes de tailles $n \leq 10^5$, on remarque que les résultats varient peu selon que le tableau soit déjà ordonné ou pas (contrairement à d'autres tri comme QUICKSORT). En revanche à partir de $n = 10^5$, les différences se creusent. Une étude plus approfondie de ce phénomène serait à envisager.

A.4 Comparaison avec d'autres algorithmes de tri

A.4.1 Slowsort

Description

Le tri SLOWSORT aussi appelé *tri par insertion* est un tri très simple basé sur l'invariant suivant :
$I(k, n) \equiv (T[0...k-1] \ \leq \ T[k...n-1]) \ \wedge \ (T[k...n-1] \ \nearrow)$
La complexité de cet algorithme est en $\Theta(n^2)$. Il devrait donc présenter des temps de calcul asymptotiques plus élevés que le HEAPSORT.

Code

```
1   void slowsort(int * T,int n)
2   {
3       int i,j;
4       int k = n;  // I(k,n)
5
6       while (k != 0)
7       {
8           i=0;
9           j=k-1;
10
11          while (i != k-1)
12          {
13              if (T[i] > T[j]) j = i;
14              i++;
15          }
16
17          if (j != k-1) swap (T[j],T[k-1]);       // I(k-1,n)
18          k--;                                    // I(k)
19      }
20  }
```

Comparaison

Pour comparer les tris, nous avons utilisé les mèmes méthodes de mesures de temps définies précedemment.

Taille	SLOWSORT (tableau desordonné)	SLOWSORT (tableau ordonné)	HEAPSORT (tableau desordonné)	HEAPSORT (tableau ordonné)
10^1	2 μs	$\leq 1\mu s$	4 μs	2 μs
10^2	35 μs	27 μs	32 μs	29 μs
10^3	2756 μs	2394 μs	471 μs	459 μs
10^4	239276 μs	235733 μs	6851 μs	6112 μs
10^5	53.469 s	53.755 s	100059 μs	88544 μs
10^6	$\geq$ 5000 s	$\geq$ 5000 s	2.103 s	1.107 s

On remarque aisèment le comportement extrèmement divergent du tri SLOWSORT pour des des valeurs de n très grandes. On peut estimer le temps de calcul du SLOWSORT pour un problème de taille 10^6 à environ 5000 secondes.

A.4.2 Quicksort Simple

Description

L'algorithme de tri QUICKSORT vu en cours est l'un des algorithmes de tri les plus performants. Sa meilleure complexité est en $O(n * log_2(n))$ et son pire cas en $O(n^2)$.

Comparaison

Taille	QUICKSORT (tableau desordonné)	QUICKSORT (tableau ordonné)	HEAPSORT (tableau desordonné)	HEAPSORT (tableau ordonné)
10^1	2 μs	1 μs	4 μs	2 μs
10^2	18 μs	21 μs	32 μs	29 μs
10^3	215 μs	1676 μs	471 μs	459 μs
10^4	2813 μs	125867 μs	6851 μs	6112 μs
10^5	33954 μs	34694231 μs	100059 μs	88544 μs
10^6	0.430 s	$\geq$ 10 s	2.103 s	1.107 s

L'algorithme QUICKSORT paraît bien meilleur que le HEAPSORT mais seulement pour des tableaux purement desordonnés. Nous pouvons désordonner les donnés mais cela augmenterait considérablement le temps de calcul, il faut donc améliorer le QUICKSORT.
Bien que ces 2 algorithmes aient la mème complexité en $\Theta(n * log(n))$, le QUICKSORT est plus rapide car ses constantes sont beaucoup plus basses.

A.4.3 Quicksort Avancé

Description

Cet algorithme est celui utilisé par la fonction **qsort()** de la librairie C standard gu GNU. Il s'agit d'un algorithme utilisant le QUICKSORT, ainsi que le SLOWSORT pour trier des petites parties du tableau. Il a été mis au point par Douglas C. SCHMIDT. Il a la particularité d'avoir une complexité en $\Theta(n * log(n))$ quelque soit l'état des données du problème.

Comparaison

Taille	QUICKSORT PRO (tableau desordonné)	QUICKSORT PRO (tableau ordonné)	HEAPSORT (tableau desordonné)	HEAPSORT (tableau ordonné)
10^1	12 μs	2 μs	4 μs	2 μs
10^2	33 μs	26 μs	32 μs	30 μs
10^3	804 μs	321 μs	482 μs	460 μs
10^4	4511 μs	3552 μs	6688 μs	6142 μs
10^5	47074 μs	43748 μs	95805 μs	86646 μs
10^6	0.559 s	0.559 s	2.09 s	1.114 s

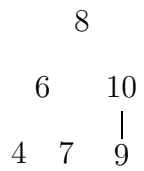
Contrairement au QUICKSORT simple, ce quicksort présente un meilleur comportement quelque soit les donnés, malgré des constantes plus élevées.

Annexe B

TP2 - Arbres de recherche optimaux

B.1 Introduction

Le premier T.P. d'IN202 nous avait presenté le HEAPSORT, une méthode de tri qui se basait sur des arbres. Ce deuxième T.P. traite lui aussi d'arbres binaires, mais cette fois-ci le problème qui se pose est celui de la recherche.
Un arbre binaire est une manière d'organiser des éléments d'un ensemble. Voici un exemple d'arbre binaire typique :



L'arbre présenté ci-dessus a pour racine 8. Chaque racine d'un arbre binaire a au plus 2 fils : 1 à gauche, 1 à droite, qui peuvent être nuls. La valeur du fils gauche est inférieure à la valeur de sa racine, et la valeur du fils droit est supérieure à la valeur de sa racine. On peut appliquer cette organisation sur n'importe quel ensemble de valeurs distinctes, muni d'une relation d'ordre. Par exemple on peut utiliser un ensemble de chaînes de caractères et l'ordre lexicographique.

L'avantage des arbres est de permettre une recherche facilitée dans les éléments de l'ensemble. Nous étudierons cet algorithme.

Ensuite, nous nous attacherons au cas où chaque élément de l'ensemble est affecté d'une probabilité d'apparition. Il faudra créer un algorithme pour créer l'arbre de recherche le plus optimal dans ce cas. Finalement nous essayerons

val

```
int val(arbre a)
{
    if (!vide(a)) return a->valeur;
    else return 0;
}
```

d'améliorer cet algorithme

B.2 Préliminaires

B.2.1 Types de données

Nous utiliserons une structure en langage C pour représenter les arbres. Chaque structure représente un arbre, et possède un pointeur vers le sous-arbre gauche et le sous-arbre droit.

```
struct typnoeud {
   int valeur;
   struct typnoeud *gauche;
   struct typnoeud *droit;
};
typedef struct typnoeud * arbre;
```

B.2.2 Fonctions d'analyse

Nous allons utiliser des fonctions pour analyser la structure d'un arbre (noté a) :

- **vide(a)** : si l'arbre désigné par a est vide, alors $vide(a) = vrai$ sinon $vide(a) = faux$.
- **val(a)** : si $vide(a)$ est faux, $val(a)$ est la valeur contenue dans le noeud a.
- **sag(a)** : si $vide(a)$ est faux, $sag(a)$ désigne la racine du sous-arbre gauche de a.
- **sad(a)** : si $vide(a)$ est faux, $sag(a)$ désigne la racine du sous-arbre droit de a.

vide

```
bool vide(arbre a)
{
    return (a == NULL);
}
```

sag

```
arbre sag(arbre a)
{
    if (!vide(a)) return a->gauche;
    else return NULL;
}
```

sad

```
arbre sad(arbre a)
{
    if (!vide(a)) return a->droit;
    else return NULL;
}
```

B.3 Recherche dans un arbre

On cherche à mettre au point un algorithme de recherche d'un élément x dans un arbre A, dont la racine est désignée par a.
Notre programme de recherche doit renvoyer un pointeur vers le sous-arbre dont la racine a' a pour valeur x. Si x n'appartient pas à l'ensemble (donc si x n'est pas dans l'arbre A), le programme doit renvoyer un sous-arbre vide.

B.3.1 Invariant

L'invariant de ce programme est donné et est :

$$I(a') \equiv x \notin a - a'$$

L'élément x n'est pas dans l'arbre désigné par a, sauf peut-être dans son sous-arbre désigné par a'.

B.3.2 Initialisation

L'initialisation de cet algorithme se fait grâce à $a = a'$. C'est à dire que l'élément x se trouve peut-être dans a. Ici $a - a' = \emptyset$ donc $x \notin \emptyset$.

B.3.3 Condition d'arrêt

Notre algorithme s'arrête lorsque les conditions de sortie du programme sont remplies, c'est à dire $vide(a') \vee (val(a') = x)$

B.3.4 Implications

On utilise le fait que pour tout arbre a, $val(a) > val(sag(a))$ et $val(a) < val(sad(a))$.

- $I(a') \wedge (vide(a') = faux) \wedge (x < val(a')) \Rightarrow I(sag(a'))$
- $I(a') \wedge (vide(a') = faux) \wedge (x > val(a')) \Rightarrow I(sad(a'))$

B.3.5 Algorithme

On utilise l'algorithme suivant :

```
procedure recherche(int x, arbre a)
    si x == val(a) renvoyer a
    si x < val(a) renvoyer recherche(x, sag(a))
    si x < val(a) renvoyer recherche(x, sad(a))
```

B.3.6 Code

```
1   arbre recherche(int x, arbre a)
2   {
3       arbre k;
4       k = a;      // I(a)
5
6       while( (!vide(k)) && (val(k) != x) )
7       {
8           if ((!vide(k)) && (x < val(k)))  // I(sag(k))
9               k = sag(k);                  // I(k)
10          else if ((!vide(k)) && (x > val(k)))
11              k = sad(k);
12      }
13      return k;
14  }
```

B.3.7 Complexité

T_i est le temps d'éxecution de l'instruction à la ligne i. On peut aisèment imaginer que $T_8 \approx T_{10}$ et $T_9 \approx T_{11}$.

$$\begin{aligned} T_{recherche}^{h(x)=0} &= T_3 + T_4 + T_6 \\ T_{recherche}^{h(x)=1} &= T_3 + T_4 + T_6 + T_8 + T_9 + T_6 + T_8 \end{aligned}$$

Ainsi d'une manière générale :

$$T_{recherche} = T_{init} + (h(x) + 1) * T_{condition\ while} + h(x) * T_{interieur\ while}$$

Plaçons nous dans le cas où x se trouve tout en bas de l'arbre, c'est à dire $h(x) = h(A)$. C'est un des cas extrèmes.

$$T_{recherche} \leq T_{init} + (h(A) + 1) * T_{while}$$

Sachant que T_{init} et T_{while} sont quasi-constants, on peut dire que cet algorithme est en $O(h(A))$.
On pourrait donc penser qu'il faut minimiser la hauteur de l'arbre pour y rendre optimales les recherches. On verra par la suite que cette propriété n'est plus vérifiée lorsque l'on affecte des probabilités aux éléments de l'arbre.

B.4 Arbre binaire de recherche optimal

Dans la suite du T.P., nous admettrons que chaque élément v_i de notre ensemble V est affecté d'une probabilité d'apparition $\pi(v_i)$ $(0 < \pi(v_i) \leq 1)$. On définit $C(A)$ le coût moyen de la recherche dans un arbre binaire A d'éléments de V.

$$C(A) \ = \ \sum_{i=0}^{n-1} (h(v_i) + 1) * \pi(v_i)$$

De plus on pose, $C(vide) = 0$.
On note également A_{ij}^k, l'arbre binaire contenant les éléments $v_i \cdots v_{j-1}$, dont la racine est est v_k.

Notre but est de déterminer l'arbre contenant les éléments $v_0 \cdots v_{n-1}$ le plus optimal, c'est à dire dont le coût est le plus faible. On pourrait déterminer tous les arbres possibles et calculer leurs coûts respectifs, mais cela donnerait un algorithme non-polynomial car le nombre d'arbres binaires contenant n éléments est minoré par $4^n/n^{3/2}$. On va donc utiliser la programmation dynamique.

B.4.1 Propriétés

On peut dégager un certain nombre de propriétés :
- D'une part, la hauteur h' dans A_{ij}^k d'un nœud, dont la hauteur dans un des sous arbres était h, est : $h' = h + 1$.
- Les coûts minimum c_{ii+1} valent $\pi(v_i)$. En effet, $c_{ii+1} = (h(v_i) + 1) * \pi(v_i) = \pi(v_i)$.
- On réunit deux sous arbres A_{ik} et A_{k+1j}, de coûts respectifs c_{ik} et c_{k+1j}, dans un arbre A_{ij}^k

$A_{ij}^k =$ v_k

A_{ik} A_{k+1j}

Le coût d'un tel arbre est :

$$C(A_{ij}^k) = C(A_{ik}) + C(A_{k+1j}) + \sum_{l=i}^{k-1} \pi(v_l) + \sum_{l=k+1}^{j-1} \pi(v_l) + \pi(v_k)$$

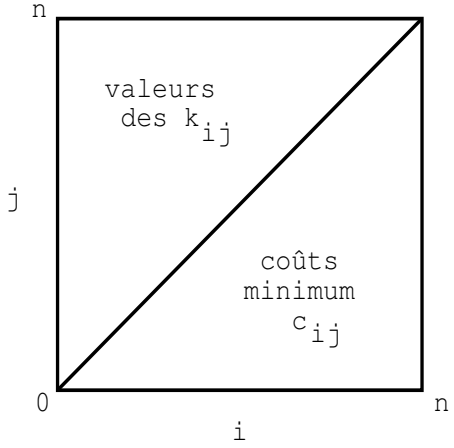
$$C(A_{ij}^k) = C(A_{ik}) + C(A_{k+1j}) + \sum_{l=i}^{j-1} \pi(v_l) \quad \text{(B.1)}$$

– La valeur c_{ij} du coût minimum pour un arbre binaire de recherche A_{ij} est le plus petit coût c_{ij}^k. Ainsi :

$$c_{ij} = \overset{i \leq k < j}{min}\ c_{ij}^k$$

B.4.2 Calcul de la matrice

Le but du problème est de calculer c_{0n} le coût minimum d'un arbre contenant tous les éléments $V[0 \cdots n-1]$. Pour éviter un recouvrement des sous problèmes, nous allons stocker les valeurs des coûts c_{ij} dans une matrice carré C. Ainsi $C[i][j] = c_{ij}$.
Il faudra également mémoriser la valeur k_{ij} qui permet le coût minimum c_{ij}.



Nous allons procéder par taille $j - i$ de problèmes croissant. Ainsi sur la diagonale centrale de notre matrice C, nous aurons les coûts $c_{ii} = 0$. Sur la t^{eme} diagonale, les coûts $C[i][i+t] = c_{i\,i+t}$. Dans la partie supérieure de la matrice, nous stockerons les k_{ij}

Invariant

Pour remplir notre matrice, nous allons utiliser l'algorithme basé sur l'invariant suivant :

Invariant :

$$I(a) \equiv (\forall r \in [0 \cdots n-a])\ (C[r][r+a] = \overset{k}{min}\ c^k_{r\,r+a})\ \wedge\ (C[r+a][r] = k)$$

Initialisation : $(\forall r \in [0 \cdots n])\ C[r][r] = 0$
$(\forall r \in [0 \cdots n-1])\ C[r][r+1] = \pi(v_r) \wedge C[r+1][r] = r$

Condition d'arrêt : $a = n$
Implications :

$$I(a) \wedge (a \neq n) \wedge ((\forall r \in [0 \cdots n-a-1])\ C[r][r+a+1] = \overset{k}{min}\ c^k_{r\,r+a+1} \wedge C[r+a+1][r] = k)$$

$$\Rightarrow I(a+1)$$

Code Source

Note : le code source de la fonction MINI est fourni en annexe.

```
1
2   double **faire_mat(int i, int j, double *proba)
3   {
4       double **C;
5       int k,a,r;
6       double *mint;
7
8       // allocation de la matrice
9       C = newmatrice(j,j);
10
11      // creation de la diagonale centrale : cout des problemes c_ii (=0)
12      for (k = 0; k < j; k++)
13          C[k][k] = 0; // I(0)
14
15      // creation de la 1ere diagonale : cout des problemes c_ii+1
16      for (k = 0; k < j-1; k++)
17          { C[k][k+1] = proba[k]; C[k+1][k] = k; } // I(1)
18
19      // creation des diagonales (cout des problemes par taille croissante)
20      a = 1;
21      while (a!= (j-i))
22      {
```

```
23          // remplissage de la a eme diagonale
24          for  (r = i; r < j-a-1; r++)
25          {
26              // on cherche le minimum
27              mint = mini(C,r,r+a+1,proba);
28              // mint[0] : cout minimum pour un probleme de taille a
29              // mint[1] : valeur du k pour lequel ce cout est minimum
30
31              C[r][r+a+1] = mint[0];
32              C[r+a+1][r] = mint[1];
33          }  // I(a+1)
34
35          a++;    // I(a)
36      }
37      return C;
38  }
```

Complexité

$$
\begin{aligned}
T_{opt}(n) &= T_9 + n * T_{13} + (n-1) * T_{17} + \sum_{a=1}^{n} T_{while}(a) \\
T_{opt}(n) &\leq \theta\left(\sum_{a=1}^{n} T_{while}(a)\right) \\
T_{while}(a) &\leq \theta\left(T_{min}(a) * (n-a)\right) \\
T_{min}(a) &\leq \theta\left(2a\right) \\
T_{opt}(n) &\leq \theta\left(\sum_{a=1}^{n} (n-a) * 2a\right) \\
T_{opt}(n) &\leq \theta\left(\sum_{a=1}^{n} 2an\right) \\
T_{opt}(n) &\leq \theta\left(2n \sum_{a=1}^{n} a\right)
\end{aligned}
$$

On en déduit que la complexité de FAIRE_MAT() est en $\theta(n^3)$ (car $\sum_{a=1}^{n} a = \theta(n^2)$).

Temps d'éxecution

Voici les temps d'éxecution de la fonction FAIRE_MAT() pour des ensembles générés aléatoirement. (Temps d'exécution sur un PC Pentium III 850 MHz)

Nombre d'éléments	10	100	200	500	1000
Temps	6 ms	12 ms	49 ms	1.3 s	10.5 s

B.4.3 Construction de l'arbre

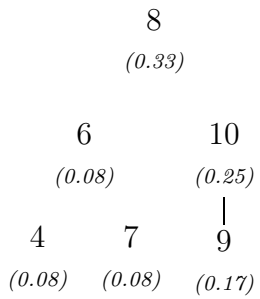
Nous devons construire l'arbre dont le coût est donné en $C[0][n]$. L'indice du nœud racine de cet arbre est donné en $C[n][0]$. Nous utiliserons une fonction récursive pour construire l'arbre.

Voici le code de la fonction ABO :

```
arbre abo(double **m, int *liste, int i, int j)
{
    int taille = j-i;
    int p = (int) m[j][i];

    if (taille <= 0) return faireArbreVide();
    else if (taille == 1) return faireArbre(liste[i], NULL, NULL);
    else return faireArbre(liste[p], abo(m,liste,i,p), abo(m,liste,p+1,j));
}
```

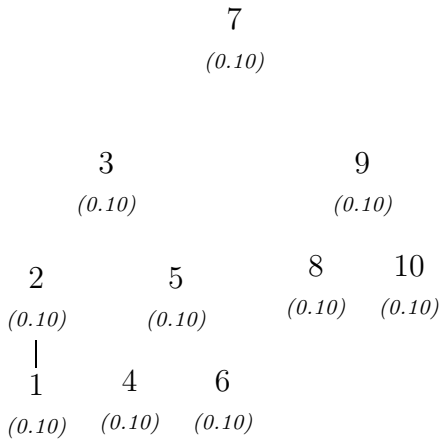
B.4.4 Exemples d'éxécution

(1.) $V = [4, 6, 7, 8, 9, 10]$
$\Pi = [1/12, 1/12, 1/12, 1/3, 1/6, 1/4]$



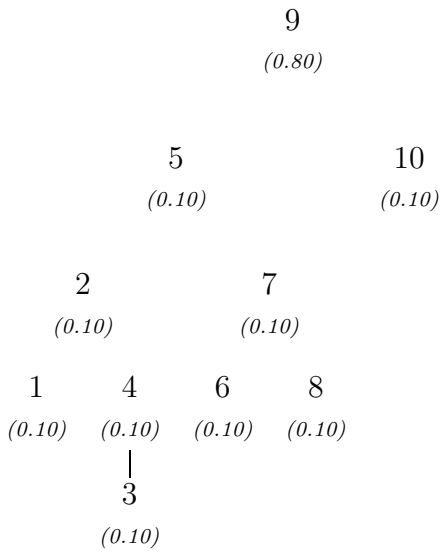
Coût de l'arbre : $C(A) = 2$.

(2.) $V = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]$
$\Pi = [0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1]$



Coût de l'arbre : $C(A) = 2.9$.
On pourrait penser, au vu de son aspect, que cet arbre est déséquilibré ou non-optimal. Or, cet arbre est basé sur des valeurs équiprobables. On se ramène donc au problème de recherche vu en II, et l'optimalité de l'arbre se juge en fonction de la hauteur de l'arbre.
La hauteur minimale h d'un arbre contenant n éléments est donné par $h = log_2(n+1)$. Avec 10 éléments, $h = 4$, c'est bien le cas ici.

(3.) $V = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]$
$\Pi = [0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.8, 0.1]$



Coût de l'arbre : $C(A) = 3.9$.

Sur cet arbre, on peut vérifier une autre propriété : un arbre binaire de recherche optimal n'est pas forcement de hauteur minimale. En effet, la hauteur est ici de 5, alors que la hauteur minimale d'un arbre binaire à 10 éléments est de 4.

B.5 Optimisation

Dans cette partie, nous allons nous attacher à essayer d'optimiser notre algorithme de construction de la matrice des coûts optimaux. Pour cela, nous travaillerons autour de l'expression :

$$c_{ij} = \overset{i \leq k < j}{min}\ c_{ik} + c_{k+1\,j} + \sum_{l=i}^{j-1} \pi(v_l)$$

Dans un premier temps, nous optimiserons le calcul de la somme des probabilités, puis nous essayerons d'utiliser des propriétés de la matrice des coûts pour optimiser le temps de calcul de notre algorithme.

B.5.1 Calcul des probabilités

Il apparaît dans l'expression des c_{ij} qu'une somme de probas est faite. Appelons $SP(i,j)$ la somme des probabilités des éléments $v_i, v_{i+1} \cdots v_j$.

$$SP(i,j) = \sum_{k=i}^{j} \Pi(k)$$

Notre fonction MINI (fournie en annexe) refait le calcul $\sum_{i \leq l < j} \pi(v_l) = SP(i, j-1)$ pour chaque calcul de c_{ij}. On peut aisèment imaginer que certains calcul seront fait plusieurs fois, puisque le calcul de c_{0n} fait intervenir $SP(0, n-1)$. Or $SP(0, n-1) - \Pi(n-2) = SP(0, n-2)$ et $SP(0, n-2)$ intervient dans le calcul de $c_{0\,n-1}$. Pour éviter un recouvrement lors des calcul des $SP(i,j)$, nous allons stocker ces sommes dans une matrice $S[0 \cdots n][0 \cdots n]$. Ainsi $SP(i,j) = S[i][j]$. Seule la partie infèrieure de cette matrice sera utilisée.
Sur la diagonale de cette matrice apparaîtront les probabilités élementaires $\Pi[i] = S[i][i]$. Nous remplirons cette matrice diagonale par diagonale, selon la formule suivante :

$$SP(i,j) = SP(i, j-1) + SP(i+1, j) - SP(i+1, j-1)$$

Code source

Le code de la fonction de calcul de cette matrice est donc :

```
1   double **probasum(double *proba, int n)
2   {
3       double **SP;
4       int k,a,r;
5
6       SP = newmatrice(n,n);
7
8       // remplissage de la premiere diagonale
9       for (k = 0; k < n; k++) SP[k][k] = proba[k];
10
11      // remplissage des diagonales
12      for (a = 1; a < n; a ++)
13      {
14          for (r = 0; r < n-a ; r++)
15              SP[r][r+a]=SP[r][r+a-1] + SP[r+1][r+a] - SP[r+a][r+a-1];
16      }
17      return SP;
18  }
```

Comparaison du temps de calcul

Malgré le fait que l'on évite un recouvrement de sous problèmes, la complexité globale de l'algorithme de création de la matrice des coûts minimaux ne va pas être affectée. En effet, si l'on reprend la formule de calcul des c_{ij} :

$$c_{ij} = \overset{i \leq k < j}{min}\ c_{ik} + c_{k+1\,j} + \sum_{l=i}^{j-1} \pi(v_l)$$

Le temps de calcul d'un c_{ij} est fonction de la taille $k = j - i - 1$. En admettant que dans la version non-optimisé, le calcul de la sommes de i à $j-1$ des probabilités n'est fait qu'une seule fois au début de la fonction MINI, on a le temps d'éxecution suivant :

$$\begin{aligned} T_{calcul\ min} &= T_{min} + T_{somme\ probas} + \varphi \\ T_{calcul\ min} &= \alpha k + \beta k + \varphi \\ T_{calcul\ min} &= \gamma k + \varphi \end{aligned}$$

Si on utilise l'optimisation que l'on vient de décrire, et que l'on calcule la matrice SP de somme des probabilités au début du programme, $T_{somme\ probas}$ n'est plus fonction de k, mais est en temps constant. Toutefois $T_{calcul\ min}$ ne change que d'un coefficient.

On en déduit que la complexité de notre algorithme reste en $O(n^3)$. Toutefois, comme on peut le remarquer sur le tableau suivant, la différence due aux coefficients est nettement visible, et compense le temps perdu à la création de la matrice SP (en $\theta(\frac{n^2}{2})$).

Nombre d'éléments	10	100	200	500	1000
Temps (Non-optimisé)	6 ms	12 ms	49 ms	1.3 s	10.5 s
Temps (Optimisé)	6 ms	12 ms	44 ms	1.16 s	9.5 s

Il faudra néanmoins tenir compte du fait que la taille mémoire allouée pour les matrices des coûts et des somme de probabilités est trés importante : de l'ordre de $2n^2$ pour un problème de taille n. De plus, on travaille sur des types de donnés flottants, dont l'occupation en mémoire est assez importante.
On peut également remarquer que l'algorithme de calcul de la matrice des sommes de probabilités n'occupe que la moitié de l'espace mémoire qui lui est alloué, ce qui représente la perte d'environ un quart de l'espace mémoire total utilisé.

B.5.2 Utilisation des propriétés de la matrice

Nous allons utiliser la proprieté de monotonie de la matrice des coûts. Cette propriété démontrée par KNUTH énonce que $\forall i, j \;\; M[i][j-1] \leq M[i][j] \leq M[i+1][j]$.
Notre matrice des coûts c_{ij} vérifie cette propriété sur chacune de ses parties triangulaires inférieures et supérieures :

$$\forall i, j \; c_{i\,j-1} \leq c_{ij} \leq c_{i+1\,j}$$

En appellant k_{ij} l'indice de l'élément racine du sous arbre de coût c_{ij}, on a :

$$\forall i, j \; k_{i\,j-1} \leq k_{ij} \leq k_{i+1\,j}$$

Dans l'expression du calcul des c_{ij}, on peut changer l'expression $i \leq k < j$ par la formule précedente :

$$c_{ij} = \min^{C[i][j-1] \leq k \leq C[j][i+1]} c_{ik} + c_{k+1\,j} + \sum_{l=i}^{j-1} \pi(v_l)$$

Complexité

Appelons α la taille du problème traité par la fonction MINI_OPTIM. D'après la relation de monotonie, on a $\forall i, j \; t.q. \; 0 \leq i < j \leq n \;\; k_{i\,j-1} \leq k_{ij} \leq k_{i+1\,j}$, on a $\alpha = k_{i+1\,j} - k_{i\,j-1}$. On peut donc encadrer α :

$$
\begin{array}{rccll}
k_{i\,i} \leq & & k_{i\,i+1} & \leq k_{i+1\,i+1} & \\
i \leq & & k_{i\,i+1} & \leq i+1 & \text{(B.2)} \\
k_{i+1\,i+1} \leq \cdots \leq k_{i+1\,j-1} \leq & & k_{i+1\,j} & \leq k_{i+2\,j} \leq \cdots \leq k_{j-1\,j} & \\
(4.1) \Rightarrow \quad i+1 \leq & & k_{i+1\,j} & \leq j & \text{(B.3)} \\
k_{i\,i+1} \leq \cdots \leq k_{i\,j-2} \leq & & k_{i\,j-1} & \leq k_{i+1\,j-1} \leq \cdots \leq k_{j-2\,j-1} & \\
(4.1) \Rightarrow \quad i \leq & & k_{i\,j-1} & \leq j-1 & \text{(B.4)} \\
(4.2) \wedge (4.3) \Rightarrow \quad i+1-i \leq & \alpha = k_{i+1\,j} - k_{i\,j-1} & & \leq j-j+1 & \\
1 \leq & & \alpha & \leq 1 & \text{(B.5)}
\end{array}
$$

Par conséquent, $\alpha = 1$. On va maintenant regarder le temps de calcul global de la fonction MINI_OPTIM :

$$
\begin{aligned}
T_{mini\_optim}(\alpha) &= T_{somme\,probas} + T_{min} \\
T_{mini\_optim}(\alpha) &= O(1) + O(\alpha) \\
T_{mini\_optim}(\alpha) &= O(1) + O(1) \\
T_{mini\_optim}(\alpha) &= O(1)
\end{aligned}
$$

La nouvelle fonction MINI_OPTIM est en temps constant ($O(1)$). En effet, grâce aux optimisations du calcul de la somme des probabilités, l'initialisation se fait en temps en constant.
Ceci implique donc que la complexité de la création de la matrice des coûts optimums devient $O(n^2)$.

Code

Nous allons juste modifier le code de notre fonction MINI :

```
double *mini_optim(double **C, int i, int j,  double **probasum)
{
    double somme = 0;
    double *mint;

    int a = (int) C[j-1][i];
    int b = (int) C[j][i+1];

    // mint est un petit tableau de 2 doubles qui nous permettra de renvoyer
    // 2 valeurs
    mint = (double *)malloc(2 * sizeof(double));

    somme = probasum[i][j-1];
```

```
    if ( (C[i][a]+C[a+1][j]) < (C[i][b]+C[b+1][j]) )
    {
mint[0] = C[i][a] + C[a+1][j] + somme;
mint[1] = a;
    } else {
mint[0] = C[i][b] + C[b+1][j] + somme;
mint[1] = b;
    }

    return mint;
}
```

Comparaison

La nouvelle complexité de notre algorithme, bien meilleure que la précédente, se vérifie en pratique : (ces temps ne concernent uniquement que le calcul de la matrice des coûts minimaux)

Nombre d'éléments	10	100	200	500	1000
Temps (Non-optimisé)	6 ms	12 ms	49 ms	1.3 s	10.5 s
Temps (Optimisé)	5 ms	9 ms	22 ms	112 ms	481 ms

Code source complet

Ce code source presente quelques différences avec les blocs de code des différentes fonctions présentées auparavant. Ces modifications ne concernent que la représentation des types de donnés, mais ne changent en rien les algorithmes ou les performances cités.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

typedef int bool;

#define TRUE 1
#define FALSE 0

// structure d'un arbre
// NB: on ne stocke pas la valeur du noeud directement, mais son indice.
// cela permet de faire des structures 'legere' et de pouvoir garder une reference vers la probabilite
// d'apparition de tel noeud
// (Une telle structure comportant a la fois la valeur du noeud et sa proba est plus lourde
// en memoire et a provoque des problemes d'alignement)
struct typnoeud {
    int indice;
    struct typnoeud *gauche;
    struct typnoeud *droit;
};
```

```
typedef struct typnoeud * arbre;

// cree une liste d'elements aleatoires
int *randomlist(int n)
{
    int *bla;
    int i;

    bla = malloc(n * sizeof(int));
    for (i = 0; i<n; i++) bla[i] = random()/200000;
    return bla;
}

// cree une liste de probabilites aleatoires
double *randomprobas(int n)
{
    double *bla;
    int i;

    bla = malloc(n * sizeof(double));
    for (i = 0; i<n; i++)  bla[i] = ( (double) random()) /RAND_MAX;
    return bla;
}

// retourne TRUE si a est vide
bool vide(arbre a)
{
    return (a == NULL);
}

// retourne la valeur de l'arbre, 0 si arbre est vide
int val(arbre a, int *ensemble)
{
    if (!vide(a)) return ensemble[a->indice];
    else return 0;
}

// retourne le sous-arbre gauche
arbre sag(arbre a)
{
    if (!vide(a)) return a->gauche;
    else return NULL;
}

// retourne le sous-arbre droit
arbre sad(arbre a)
{
    if (!vide(a)) return a->droit;
    else return NULL;
}

// recherche dans un arbre binaire de recherche.
arbre recherche(int x, arbre a, int *ensemble)
{
    arbre k;
    k = a; // I(x,a,a)

    while( (!vide(k)) && (val(k,ensemble) != x) )
    {
if ((!vide(k)) && (x < val(k,ensemble))) k = sag(k);
else if ((!vide(k)) && (x > val(k,ensemble))) k = sad(k);
    }
    return k;
}
```

```
// affiche une matrice
void print_2dtab(double **tab, int m, int n)
{
    int i = 0;
    int j = 0;

    for (i = 0; i<m; i++)
    {
for (j = 0; j<n; j++) printf("%f ", tab[i][j]);
printf("\n");
    }
}

// alloue une matrice vide
double **newmatrice(int i, int j)
{
    double **C;
    int k;

    C = (double **) malloc(i * sizeof(double *));
    for (k = 0; k<j; k++) C[k] = (double *)malloc(j * sizeof(double));

    return C;
}

// calcul d'un c_ij minimal (algorithme optimise)
double *mini_optim(double **C, int i, int j,  double **probasum)
{
    int k = 0;
    double cmin;
    double kmin;
    double somme = 0;
    double *mint;

    int a = (int) C[j-1][i];
    int b = (int) C[j][i+1];

    // mint est un petit tableau de 2 doubles qui nous permettra de renvoyer
    mint = (double *)malloc(2 * sizeof(double));
    somme = probasum[i][j-1];

    if ( (C[i][a]+C[a+1][j]) < (C[i][b]+C[b+1][j]) )
    {
mint[0] = C[i][a] + C[a+1][j] + somme;
mint[1] = a;
    } else {
mint[0] = C[i][b] + C[b+1][j] + somme;
mint[1] = b;
    }

    return mint;
}

// calcul d'un c_ij minimal
double *mini(double **C, int i, int j, double **probasum)
{
    int k = 0;
    double cmin;
    double kmin;
    double somme = 0;
    double *mint;

    // mint est un petit tableau de 2 doubles qui nous permettra de renvoyer
```

```
    // 2 valeurs
    mint = (double *)malloc(2 * sizeof(double));

    // on calcule la somme des probas une bonne fois pour toutes
    for (k = i; k< j; k++) somme += probasum[k][k];
    // (ou somme = probasum[i][j-1]

    // on admet que k=i convient pour un cout minimal (pour l'instant :))
    k = i;
    kmin = k;
    cmin = somme + C[i][k] + C[k+1][j];

    for (k = i; k< j; k++)
    {
// regarde si cette valeur de k fait un cout plus petit que ce que
// l'on avait avant
if (somme + C[i][k] + C[k+1][j] < cmin)
{
    // on a trouve un nouveau min
    cmin = somme + C[i][k] + C[k+1][j];
    kmin = (double) k;
}
    }

    // renvoie du minimum definitif
    mint[0] = cmin;
    mint[1] = kmin;
    return mint;
}


// creation de la matrice des couts min
double **faire_mat(int i, int j, double **probasum)
{
    double **C;
    int k,a,r;
    double *mint;

    // allocation de la matrice
    C = newmatrice(j,j);

    // creation de la diagonale centrale : cout des problemes c_ii (=0)
    for (k = 0; k < j; k++) C[k][k] = 0; // I(0)
    // creation des 1eres diagonales : cout des problemes c_ii+1
    for (k = 0; k < j-1; k++) { C[k][k+1] = probasum[k][k]; C[k+1][k] = k; }

    // creation des diagonales (cout des problemes par taille croissante)
    a = 1;
    while (a!= (j-i))
    {
// remplissage de la a eme diagonale
for  (r = i; r< j-a-1; r++)
{
    mint = mini(C,r,r+a+1,probasum);
//     mint = mini_optim(C,r,r+a+1,probasum);
    C[r][r+a+1] = mint[0];
    C[r+a+1][r] = mint[1];
}  // I(a+1)
a++; // I(a)
    }
    return C;
}

// creation de la matrice de somme des probabilites
double **probasum(double *proba, int n)
```

```
{
    double **C;
    int k,a,r;

    // allocation de la matrice
    C = newmatrice(n,n);

    for (k = 0; k < n; k++) C[k][k] = proba[k];

    for (a = 1; a < n; a++)
    {
for  (r = 0; r< n-a; r++)
    C[r][r+a] = C[r][r+a-1] + C[r+1][r+a] - C[r+1][r+a-1];
    }
    return C;
}


arbre faireArbreVide(void)
{
    return NULL;
}

// cree un nouvel arbre de valeur v avec 2 sous arbres a1 et a2
arbre faireArbre(int v, arbre a1, arbre a2)
{
    arbre nouveau;

    nouveau = (arbre) malloc(sizeof(arbre));
    nouveau->indice = v;
    if (a1 != NULL) nouveau->gauche = a1;
    if (a1 != NULL) nouveau->droit = a2;
}

// cree l'arbre binaire optimal correspondant a la matrice des valeurs
arbre abo(double **m, int i, int j)
{
    int taille = j-i;
    int p = (int) m[j][i];

    // taille nulle -> arbre nul
    if (taille <= 0) return faireArbreVide();
    // taille 1 : une seule valeur
    else if (taille == 1) return faireArbre(i, NULL, NULL);
    // taille >1
    else  return faireArbre(p, abo(m, i, p), abo(m, p+1, j) );
}

// affiche un arbre
void DrawTree(arbre truc, int etage, int *liste, double *probas)
{
    int i;
    if (truc != NULL) {
        for(i = 0; i < etage-1; i++) printf("    ");

        if (etage !=0) printf("\\---");
        printf("%d (%0.2f)\n", liste[truc->indice], probas[truc->indice]);

DrawTree(truc->gauche, etage+1, liste, probas);
        DrawTree(truc->droit, etage+1, liste, probas);
    }
}

// affiche un arbre en qtree
void DrawTree_Latex(arbre truc, int etage, int *liste, double *probas)
```

```
{
    if (etage == 0) printf("\\Tree");
    if (truc != NULL)
    {
if ((truc->droit != NULL) || (truc->gauche != NULL))
{
    printf("[.%d\\arb{%0.2f} ", liste[truc->indice], probas[truc->indice]);
    DrawTree(truc->gauche, etage+1, liste, probas);
    DrawTree(truc->droit, etage+1, liste, probas);
    printf("] ");
} else printf("%d\\arb{%0.2f} ", liste[truc->indice], probas[truc->indice]);
    }
    if (etage == 0) printf("\n");
}


int main(int argc, char **argv)
{
    arbre a;
    double **m;
    double **p;

    int nbelem = 6;
    int liste[] = {4,6,7,8,9,10};
    double probas[] = {0.083,0.083,0.083,0.333,0.1667,0.25 };

    // Creation de la matrice somme des probas
    p = probasum(probas,nbelem+1);

    // creation de la matrice des couts minimum
    m = faire_mat(0,nbelem+1,p);
    print_2dtab(m,nbelem+1,nbelem+1);

    printf("Cout total de l'arbre: %f\n", m[0][nbelem]);
    // creation de l'arbre a partir de la matrice
    a = abo(m,0,nbelem);
    DrawTree(a,0, liste, probas);

    return EXIT_SUCCESS;
}
```

Annexe C

TP3 - Sommes de sous-ensembles

C.1 Premier algorithme

Soit $E = e_0, e_1, \cdots, e_{n-1}$ un ensemble de naturels ($E \subset N$). On cherche à calculer un sous-ensemble E' de E tel que la somme des éléments de E' fasse s.
Une première solution à ce problème consiste à tester tous les sous-ensembles de E jusqu'à ce que l'on trouve un ensemble de somme cherchée.
Cet algorithme est impraticable, car il existe $2^n - 1$ sous-ensembles pour un ensemble de cardinal n. Ceci nous donne un algorithme dont la complexité est de $\Omega(2^n)$ dans le pire des cas.
Un tel algorithme est impraticable car sa complexité est non-polynomiale.

C.1.1 Petit programme de dḿonstration

Voici un petit programme qui permet de calculer tous les sous-ensembles d'un ensemble E :

```
1   void calcul_ssens(int *ens, int cardinal_e)
2   {
3       int i,j;
4       float max_ens;
5
6       max_ens = pow(2,cardinal_e) - 1;
7
8       for (i = 1; i<=max_ens; i++)
9       {
10          int s = 0;
11          for (j = cardinal_e; j >= 0; j--)
12          {
13              if( (i >> j)%2 ) s += ens[j];
14          }
```

```
15            // Ici on a dans s la somme d'un sous-ensemble de E
16        }
17    }
```

NB : on peut modifier cet algorithme pour s'arrêter lorsque l'on a trouvé une somme précise.

Soit T_i le temps d'éxecution de l'instruction à la ligne i.
Essayons de calculer la complexité de la fonction CALCUL_SSENS pour un ensemble E de cardinal n.

$$\begin{aligned}
T_{calcul\ ssens}(n) &= T_{init} + T_{for\ ligne\ 8} \\
T_{calcul\ ssens}(n) &= T_6 + max_ens * (T_{for\ ligne\ 11}) \\
T_{calcul\ ssens}(n) &= log_2(n) + (2^n - 1) * (n) \\
T_{calcul\ ssens}(n) &= \theta\left(log_2(n) + n * (2^n - 1)\right) \\
T_{calcul\ ssens}(n) &= \theta\left(n * (2^n - 1)\right)
\end{aligned}$$

Ce programme n'est peut-être pas le plus optimal pour calculer tous les sous-ensembles mais il montre que cet algorithme ne peut pas atteindre des temps de calcul polynomiaux.

C.1.2 Résultats

Voici quelques temps de calcul pour des ensembles aléatoires de cardinal :

Cardinal de E :	5	10	15	20	25
Temps :	4 ms	10 ms	24 ms	822 ms	32 s

On constate que cet algorithme est très inadapté, il faut donc en trouver un plus performant.

C.2 Deuxième algorithme

Nous allons mettre au point un deuxième algorithme qui se basera sur la programmation dynamique pour calculer les sous-ensembles.

C.2.1 Définitions

On appelle E_m le sous ensemble de E contenant les éléments $e_0, e_1, \cdots e_{m-1}$. Soit $v(m, x)$ la valeur de vérité de la proposition :

$$\exists E' \ ; \ E' \subset E_m \ \ tel\ que \ \ x = \sum_{e' \in E'} e'$$

Nous avons plusieurs formules pour calculer $v(m, x)$:
- $v(0, 0) = vrai$ et $v(0, x \neq 0) = faux$ (Un ensemble vide est de somme nulle).
- $v(m, x < 0)$ (Il n'existe pas d'ensemble de sommes négatives).
- $v(m, x) = v(m-1, x') \vee v(m-1, x'')$ pour $x' = x$ et $x'' = x - e_{m-1}$.

C.2.2 Calcul de la matrice

Nous allons créer une matrice $V[0 \cdots n][0 \cdots s]$ telle que $V[m][x] = v(m, x)$. Les dimensions de cette matrice sont (n, s) avec n le cardinal de l'ensemble E et s la somme de tous les éléments de E.
Pour cela nous allons nous servir des équations décrites précèdemment. Nous allons mener le calcul des $v(m, x)$ par valeurs de m croissantes.

Code

```
1   int maxsum(int *E, int n)
2   {
3       int somme = 0;
4
5       for (k = 0; k<n; k++) somme += E[k];
6       return somme;
7   }
8
9   int **faire_mat(int *E, int carde, int sommetot)
10  {
11      int **V;
12
13      V = newmatrice(carde+1, sommetot+1);
14
15      V[0][0] = true;
16
17      for (int k = 1; k<=carde; k++)
18      {
19          for (int i = 1; i<=sommetot; i++)
20          {
```

```
21                if (V[k-1][i] == true) V[k][i] = true;
22                if (V[k-1][i - E[k-1]] == true) V[k][i] = true;
23            }
24        }
25        return m;
26    }
```

Complexité

On touche chaque case de la matrice V une fois. Cette matrice étant de dimension $n * s$, la complexité de ce programme est donc en $\theta(n * s)$.

Exemples d'éxecution

- *1^er exemple :*
  $E = 1, 3, 7, 4, 8$, la matrice V obtenue grâce à l'éxecution du programme est :
  s : 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
  0 : 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
  1 : 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
  2 : 0 1 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
  3 : 0 1 0 0 1 0 0 0 1 0 0 1 0 0 0 0 0 0 0 0 0 0 0
  4 : 0 1 0 0 1 1 0 0 1 0 0 1 1 0 0 1 0 0 0 0 0 0 0
  5 : 0 1 0 0 1 1 0 0 1 1 0 1 1 1 0 1 1 0 0 1 1 0 0 1

C.2.3 Calcul du sous ensemble de somme s

On se propose de calculer le sous-ensemble E' de somme s grâce à la matrice V que nous venons de calculer. Pour cela nous allons utiliser l'invariant suivant :

$$I(m,x) \equiv (k \in [m+1; n-1] \Rightarrow E'[k] = (e_k \in E')) \wedge (s = x + \sum_{k \in [m+1][n-1]} e_k)$$

L'initialisation d'un tel invariant se fait avec :

$$(x = s) \wedge (m = n - 1)$$

La condition d'arrêt est $x = 0$ (Tous les éléments de la somme ont été trouvés). Les implications sont :

$$\begin{aligned} I(m,x) \wedge V[m-1][x] &\Rightarrow I(m-1,x) \\ I(m,x) \wedge V[m-1][x - E_{m-1}] \wedge (E'[m-1] = true) &\Rightarrow I(m-1, x - E_{m-1}) \end{aligned}$$

Cet algorithme est très rapide, en $\theta(m)$.

Programme

```
1   int *trouve_somme(matrice V, tableau E, tableau Ep, int carde, int somme)
2   {
3       int x = somme;
4       int m = carde;
5
6       if (V[carde][somme] == false)
7       {
8           printf("Somme pas possible\n");
9           return NULL;
10      }
11
12      while (x != 0)
13      {
14          if (V[m-1][x] == true) m--;
15          else if (V[m-1][x-E[m-1]] == true)
16          {
17              x = x - E[m-1];
18              m--;
19              Ep[m] = TRUE;
20          }
21      }
22  }
```

Le tableau Ep résultant est de la forme binaire ($TRUE$ si $Ep[i]$ est nécessaire pour la somme, $FALSE$ s'il ne l'est pas).

Exemple d'éxecution

Voici quelques exemples d'éxecution avec comme ensemble $E = 1, 3, 7, 4, 8$.

- 1^{er} exemple : $s = 23$

```
lc:~/work/algo/tp3$ ./tp3 23
Somme totale 23
1 1 1 1 1
```

- 2^{e} exemple : $s = 18$

```
lc:~/work/algo/tp3$ ./tp3 18
Somme totale 23
Somme pas possible
```

- 3^{e} exemple : $s = 5$

```
lc:~/work/algo/tp3$ ./tp3 5
Somme totale 23
1 0 0 1 0
```

C.3 Optimisation & parallélisation

C.3.1 Parallélisation du calcul de la matrice

Nous allons essayer d'améliorer le temps de calcul de la matrice V grâce à la parallélisation. Nous allons pour cela linéariser une des boucles de notre algorithme.
En effet, pour calculer chaque ligne $V[m][0 \cdots s]$, on a besoin de la ligne précédente $V[m-1][0 \cdots s]$, mais chaque élément $V[m][s]$ peut être calculé indépendamment des autres $V[m][s']$.
Nous pouvons donc calculer chaque case de la ligne $V[m][0 \cdots s]$ sur un processeur indépendant. Pour atteindre ce parallèlisme optimal, il nous faut $s+1$ processeur. La complexité en temps descend alors à $\theta(n)$.

C.3.2 Application

Pour programmer une version parallèle de notre algorithme, nous allons utiliser les directives OpenMP. Ces directives sont interprétés directement par le compilateur C qui va les traduire en un programme multi-tâche dont l'éxecution se fera en parallèle sur plusieurs processeurs. L'ensemble de la documentation sur le sujet est disponible sur *http ://www.openmp.org*.
Nous allons utiliser l'implémentation OpenMP du compilateur C/C++ intel *icc*. Le programme sera compilé et testé sur un bi-processeur Pentium III à 850MHz sous linux.

Code source

Voici le code source de la fonction de calcul de la matrice parallelisé

```
1   int **faire_mat(int *E, int carde, int sommetot)
2   {
3       int **V;
4
5       V = newmatrice(carde+1, sommetot+1);
6
7       V[0][0] = true;
8
9       for (int k = 1; k<=carde; k++)
10      {
11  #pragma omp parallel
12  {
13  int i;
14  #pragma omp for schedule(dynamic,1) nowait
```

```
15          for (int i = 1; i<=sommetot; i++)
16          {
17              if (V[k-1][i] == true) V[k][i] = true;
18              if (V[k-1][i - E[k-1]] == true) V[k][i] = true;
19          }
20  }
21  #pragma omp parallel end
22      }
23      return m;
24  }
```

La compilation d'un tel programme se fait ainsi : (Compilateur intel 6.0 architecture IA32 sous linux) :

```
lc:~/work/algo/tp3$ icc tp3.c -openmp -openmp_report2
tp3.c
tp3.c(77) : (col. 1) remark: OpenMP DEFINED LOOP WAS PARALLELIZED.
tp3.c(74) : (col. 1) remark: OpenMP DEFINED REGION WAS PARALLELIZED.
```

Nombre de processeurs

Le nombre de *threads* utilisé par OpenMP est controlé par l'intermédiaire d'une variable d'environnement du shell (*OMP_NUM_THREADS*). Nous avons vu que pour une parallélisation optimale, il nous faut s processeurs. Or pour des ensembles très grands, cette valeur de s est importante. En pratique, il faudrait donc mettre la valeur de s dans cette variable d'environnement.

Le problème est que OpenMP se base sur le principe des threads Unix. Chaque thread apparaît comme un processus indépendant vis-à-vis du scheduler. Dans une situation idéale, le nombre de threads active est inférieure au nombre de processeurs libres, donc chaque threads s'effectue sur un processeur indépendant. Si l'on fixe un nombre de threads important, plus important que le nombre de processeur, le scheduler va devoir choisir quel processeur va executer quelle thread. Sur les systèmes Unix modernes, le scheduler est généralement en $\theta(n)$ ou $\theta(log(n))$. On en déduit donc qu'il faut réduire le nombre de threads pour ne pas perdre trop de temps dans les interruptions scheduler.
Le bon sens conduit à fixer un nombre de threads égal ou légèrement supérieur au nombre de processeurs.
Ainsi pour un nombre de threads p, le temps de calcul T_s est divisé par p, mais cela ne change pas la forme de la complexité en temps de notre algorithme. Pour $p = s$, on se trouve en situation optimale.

Comparaison des temps de calcul

Nous choisirons ici de faire les calculs à 2 threads (c'est à dire égal au nombre de processeurs). Cette parallèlisation n'est pas optimale, mais on peut s'attendre à ce que le temps de calcul soit divisé par 2.
Les ensembles E choisis seront générés aléatoirement. La valeur de s dépend de la valeur des éléments de cet ensemble. La plage de valeurs de notre générateur de nombre aléatoires variant entre 1 et 1000, on peut donc déduire que $s \leq 1000 * n$ avec n le cardinal de E.
De plus nous avons inséré des constantes (grâces à des instruction *sleep*) pour pouvoir garder un temps de calcul mesurable sans pour autant allouer trop de mémoire.

Cardinal de E	10	50	100	200
Temps de calcul (1 Processeur)	0.53 s	0.63 s	2.28 s	7.4s
Temps de calcul (2 Processeurs)	0.53 s	1.13 s	2.78 s	10.43 s

Ces temps de calcul peuvent paraître surprenant car notre programme est plus lent lorsqu'il est parallélisé !

Après analyse, tout ceci apparaît très normal. En effet, l'implémentation OpenMP d'Intel se base sur le principe des threads. Or la déclaration d'une thread nécessite beaucoup de temps processeur, car il faut entre autre initialiser un processus, initialiser l'espace mémoire, copier les ressources mémoires locales, et fixer les différents systèmes de partages des ressources mémoire. De même, à la fin de la boucle, le processus maître attend la fin de l'éxecution des p processus fils. Ces initialisations de threads sont refaites pour chaque ligne de notre matrice car nous avons restreint la région parallèle à l'intérieur de la première boucle *for*. Ces constantes sont négligeables lorsque la région parallélisé nécessite un temps de calcul important, mais ici seules 2 comparaisons et 1 assignation doivent être exécutés en paralléle.

C.3.3 Conclusion

Cette parallélisation peut être très bénéfique pour la complexité de l'algorithme (réduction à $\theta(n)$). Mais le nombre de processeurs nécessaires pour obtenir une parallélisation optimale et le faible temps d'éxecution des boucles parallèles nécessite une technique de programmation parallèle adaptée.

C.4 Code source commenté

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

// affiche une matrice
void print_2dtab(int **tab, int m, int n)
{
    int i = 0;
    int j = 0;

    printf("    ");
    for (j = 0; j<n; j++) printf("%2.1d ", j);
    printf("\n");

    for (i = 0; i<m; i++)
    {
printf("%2.1d: ", i);
for (j = 0; j<n; j++) printf("%2.1d ", tab[i][j]);
printf("\n");
    }
}

// alloue une matrice vide de taille i*j
int **newmatrice(int i, int j)
{
    int **C;
    int k;

    C = (int **) malloc(i * sizeof(int *));
    for (k = 0; k<j; k++) C[k] = (int *)malloc(j * sizeof(int));

    return C;
}

// calcul somme de 0 a n des E_i
int maxsum(int *E, int n)
{
    int somme = 0;

    for (int k = 0; k<n; k++) somme += E[k];
    return somme;
}

// Calcul de la matrice V(m,x)
bool **faire_mat(int *E, int carde, int sommetot)
{
    bool **m;

    m = newmatrice(carde+1, sommetot+1);

    m[0][0] = true;
```

```
    for (int k = 1; k<=carde; k++)
    {
for (int i = 1; i<=sommetot; i++)
{
    if (m[k-1][i] == 1) m[k][i] = true;
    if (m[k-1][i - E[k-1]] == 1) m[k][i] = true;
}
    }
    return m;
}

/* Invariant: I(m,x) = k e [m+1, n+1] => E'[k]=(e_k e E') ET s = x + sum e_k
 * Init: m =  n-1  ET  x = s
 * Arret: x = 0
 * Implic: I(m,x) & T[m-1][x] => I(m-1,x)
 * Implic: I(m,x) & T[m-1][x-E[m-1] ] & E'[m-1] VRAI => I(m-1,x-E[m-1])
 */
int *trouve_somme(bool **V, tableau E, tableau Ep, int carde, int somme)
{
    int x = somme;
    int m = carde;

    if (V[carde][somme] == false)
    {
printf("Somme pas possible\n");
return NULL;
    }

    while (x != 0)
    {
if (V[m-1][x] == true) m--;
else if (V[m-1][x-E[m-1]] == true)
{
    x = x - E[m-1];
    m--;
    Ep[m] = TRUE;
}
    }
    return Ep;
}

int main()
{
    bool **m;
    int *Ep;
    int sommetot;
    int i;

    int e[] = {1,3,7,4,8};
    int carde = 5;
```

```
    sommetot = maxsum(e, carde);
    printf("Somme totale %d\n", sommetot);

    m = faire_mat(e, carde, sommetot);
    print_2dtab(m, carde+1, sommetot+1);

    Ep = trouve_somme(m,e,carde,atol(argv[1]));
    for (i = 0; i<carde; i++) printf("%d ", Ep[i]);
    printf("\n");

    return 0;
}
```

Annexe D

TP IN311 : le sac à ados

D.1 Position du problème

On souhaite remplir un sac à dos de capacité C en maximisant la valeur de son chargement. Plus précisément, on consideère un ensemble de N objets indicés de 0 à $N-1$. Tout objet d'indice $i \in [0; N-1]$ est de volume vol_i et de valeur val_i. On demande de déterminer un sous-ensemble $S \subseteq [0; N-1]$ d'indices vérifiant les deux conditions :

1. $\sum_{i \in S} vol_i \leq C$;
2. $\sum_{i \in S} val_i maximum$.

D.2 Equation du récurrence

Soit $V(n, C)$ la valeur maximum d'un chargement de volume total C composé d'objets choisis parmis les n premiers objets.

On cherche à déterminer la valeur $V(N, C)$ qui est la valeur maximum du chargement du sac à dos de capacité C. Pour trouver l'équation de réccurence de $V(n, c)$, on suppose tous les problèmes résolus à l'étape $V(n-1, k), \quad \forall k$. Les solutions de ce problèmes sont bien sur optimales.

Lorsque l'on se trouve en $V(n, k)$, on a 2 possibilités :

- Soit l'objet $n-1$ rentre dans le cas (son volume est inferieur à k) et est interessant à prendre. Dans ce cas, on le prend, et l'on retire son volume du volume disponible dans le sac, et l'on augmente la valeur du sac,

$$V(n, k) = V(n-1, c - Vol_{n-1}) + Val_{n-1}$$

- Soit l'objet $n-1$ n'est pas interessant à prendre. Dans ce cas, la composition du sac ne change pas :

$$V(n, k) = V(n-1, k)$$

Puisque l'on cherche à maximiser la valeur du sac, l'équation de récurrence cherchée est donc la suivante :

$$V(n > 0, k) = \max\left(V(n-1, k), V(n-1, k - Vol_{n-1}) + Val_{n-1}\right)$$

La base de cette équation est :

$$V(0, k) = 0, \forall 0 \leq k \leq K$$

(La valeur du sac comportant 0 objets est nul)

D.3 Complexité

Le programme demandé s'écrit en pseudo-langage comme suit :

```
1  procedure remplit_tab
2    pour j allant de 0 a C
3      tab[0][j] <- 0
4
5    pour i allant de 1 a N
6      pour j allant de 1 à C
7            tab[i][j] <- max(tab[i-1][j], tab[i-1][j-volume(i-1)] + valeur(i-1))
8
9  fin procedure
```

L'instruction de la première boucle (ligne 3) s'execute en temps constant $O(1)$, et elle est executée $C+1$ fois.
Le corps de la 2 boucle est lui aussi en temps constant (maximum de 2 termes), et est executé $N*(C+1)$ fois
la complexité de la procèdure *remplit_tab* est donc en $\theta(\theta(C+1)+\theta(N*(C+1)))$, donc $\theta(C*N)$).

D.4 Reconstruction de la solution

On a la fin de la procèdure *remplit_tab*, la valeur optimal du sac est $V(N,C)$. Il faut maintenant rechercher quels sont les objets permettant d'atteindre cette valeur maximale. Pour cela, il suffit de prendre l'équation de récurrence *a l'envers*.
On commence sur la dernière ligne du tableau, pour $n=N$ et $k=C$.
Si $V(n-1,k)=V(n,k)$, alors on peut dire grâce à l'équation de récurrence, que l'objet $n-1$ n'a pas participé à la solution. Si $V(n-1,k)\neq V(n,k)$ alors l'objet $(n-1)$ est dans l'arrangement du sac de valeur maximal. En quelque sorte, on le retire du sac, en faisant $k=k-volume(n-1)$.
L'algorithme de cette procèdure est le suivant :

```
1  procedure affiche_solution
2    j = C
3    pour i allant de N a 0
4      si tab[i][j] est different de tab[i-1][j]
5        alors l'objet participe a l'arrangement du sac
6            afficher l'objet i-1
7            j <- j - volume(i-1)
```

La complexité de cette procédure est en $\theta(N)$. On peut améliorer le programme en changeant la condition d'arrêt : à la ligne 3 on change l'instruction par *pour i allant de N à 0* ***ET*** *tant que* $j>0$. Ainsi on ne regarde plus chaque objet, et l'on s'arrete une fois qu'on a vidé tout le sac. En revanche cette modification ne change pas la complexité de l'algorithme.

D.5 Exemples d'execution

Exemple simple

Prenons 10 objets (dont les volumes varient entre 1 et 20. Le volume du sac est 15 :

Couts	4	9	1	1	7	6	2	4	6	8
Volumes	3	2	19	14	19	1	18	5	12	18

Voici la matrice V :

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	4	4	4	4	4	4	4	4	4	4	4	4
0	0	9	9	9	13	13	13	13	13	13	13	13	13	13
0	0	9	9	9	13	13	13	13	13	13	13	13	13	13
0	0	9	9	9	13	13	13	13	13	13	13	13	13	13
0	0	9	9	9	13	13	13	13	13	13	13	13	13	13
0	6	9	15	15	15	19	19	19	19	19	19	19	19	19
0	6	9	15	15	15	19	19	19	19	19	19	19	19	19
0	6	9	15	15	15	19	19	19	19	19	23	23	23	23
0	6	9	15	15	15	19	19	19	19	19	23	23	23	23
0	6	9	15	15	15	19	19	19	19	19	23	23	23	23

Le valeur optimal du sac est donc de 23. La reconstruction du sac donne l'arrangement suivant :

1. 8e objet, cout = 4, volume = 5
2. 6e objet, cout = 6, volume = 1
3. 2e objet, cout = 9, volume = 2
4. 1e objet, cout = 4, volume = 3

On a bien $4 + 6 + 9 + 4 = 23$ et $5 + 1 + 2 + 3 = 11 \leq 15$.

Sac vide

Prenons un exemple où la capacité du sac est 2 :

```
Couts: 2 4 5 7 2 9 6 2 4 2
Vols:  6 4 7 12 7 3 17 18 6 16
  0   0
  0   0
  0   0
  0   0
  0   0
  0   0
  0   0
  0   0
  0   0
  0   0
  0   0

Cout du sac: 0
```

Le programme a bien trouvé qu'aucun objet ne rentrait dans le sac, et donc pas de solutions.

Sac plein

Prenons un grand sac, de taille 200. Nous n'afficherons pas la matrice V pour cause de taille excessive !

```
Couts: 4 8 1 3 7 5 0 5 6 2
Vols:  12 11 18 13 2 19 6 7 2 13

Cout du sac: 41
On prend le 10eme objet, cout=2 volume=13
On prend le 9eme objet, cout=6 volume=2
On prend le 8eme objet, cout=5 volume=7
On prend le 6eme objet, cout=5 volume=19
On prend le 5eme objet, cout=7 volume=2
On prend le 4eme objet, cout=3 volume=13
On prend le 3eme objet, cout=1 volume=18
On prend le 2eme objet, cout=8 volume=11
On prend le 1er objet, cout=4 volume=12
```

L'algorithme a bien choisi tous les objets, sauf celui de coût 0, qui ne sert à rien si on veut maximiser la valeur du sac.

D.6 Programme complet et commenté

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include <math.h>
#include <sys/types.h>
#include <unistd.h>
#include <string.h>

#define N       10      /* nb d'objet */
#define Cmax    10      /* cout max d'1 objet */
#define Vmax    20      /* volume max d'1 objet */
#define C       200     /* capacite du sac */

#define max(x,y) ((x) < (y)) ? (y) : (x) /* routine pour calculer le max */
/* On utilise un tableau uni-dimensionnel,
 * on d~Ncale donc du nombre de colonnes par lignes */
#define V(x,y)   resultat[(x)*(C)+(y)]

/*
 * Cr■ation d'un tableau de couts al■atoires
 */
int *couts(int nb)
{
        int i;
        int *tab = malloc(sizeof(*tab)*nb);

        for (i=0; i<nb; i++)
                tab[i] = abs(((float)rand()/RAND_MAX)*Cmax);
```

```
30          return tab;
    }

    /*
     * Cr■ation d'un tableau de volumes al■atoires
35   */
    int *volumes(int nb)
    {
            int i;
            int *tab = malloc(sizeof(*tab)*nb);
40
            for (i=0; i<nb; i++)
                    tab[i] = abs(((float)rand()/RAND_MAX)*Vmax);

            return tab;
45  }

    /*
     *  Routine d'affichage d'un tableau bi-dimensionnel
     */
50  void print_2dtab(int *resultat, int m, int n)
    {
        int i = 0;
        int j = 0;

55      for (i = 0; i<m; i++)
        {
            for (j = 0; j<n; j++)
                    printf("%3d ", V(i,j));
            printf("\n");
60      }
    }

    int main(void)
    {
65          int *cout;
            int *volume;
            int *resultat = malloc(sizeof(*resultat)*(N+1)*C);
            int i,j;

70          srand(time(NULL)^getpid());     /* Cr■ation d'entropie */

            cout = couts(N);        /* g■n■ration et affectation des couts */
            volume = volumes(N);    /* g■n■ration et affectation des volumes */

75          /* Affichage du tableau des couts initiaux */
            printf("Couts: ");
            for (i=0; i<N; i++)
                    printf("%u ", cout[i]);
```

```
80          printf("\n");

            /* Affichage du tableau des volumes initiaux */
            printf("Vols:  ");
            for (i=0; i<N; i++)
85                  printf("%u ", volume[i]);

            printf("\n");

            /* Base de la r■curence */
90          for (i=0; i<C; i++)
                    resultat[i] = 0;

            /* Equation de r■currence */
            for (i=1; i<=N; i++) {
95                  for (j = 1; j < C; j++) {
                            /* On v~Nrifie que le r~Nsultat potentiel ne soit pas
                             * sup~Nrieur ~H la capacit~N totale du sac. */
                            if (j - volume[i-1] < 0)
                                    V(i,j) = V(i-1,j);
100                         else
                                    V(i,j) = max(V(i-1,j), V(i-1,j-volume[i-1]) + cout[i-1]);
                    }
            }
            print_2dtab(resultat,(N+1),C);  /* On affiche le r■sultat demand■ */
105         printf("\n");

            j = C-1;
            printf("Cout du sac, %d\n", V(N,C-1));
            for (i=N; i>0; i--) {
110                 if (V(i,j) != V(i-1,j))
                    {
                            printf("On prend le %d%s objet, cout=%d volume=%d\n",
                                            i, (i==1)?"er":"eme", cout[i-1],volume[i-1]);
                            j -= volume[i-1];
115                 }
            }

            return 0;
    }
```

Annexe E

Corrections de TD IN302

E.1 TD 1 : Recherche de circuits

E.1.1 Utilisation de l'algorithme de composantes fortement connexes

1. Comment utiliser l'algorithme de recherche de composantes fortement connexes vu en cours pour calculer toutes les composantes fortes d'un graphes. Appliquer la méthode sur le graphe ci-dessous.
2. Comment modifier cet algo pour disposer d'un algorithme **CIRCUIT** qui, étant donné un graphe (E, Γ) et un sommet $a \in E$, permet de savoir s'il existe un circuit passant par le sommet a. Complexité de calcul de cet algo ?
3. Comment utiliser **CIRCUIT** pour disposer d'un algorithme qui, étant donné un graphe (E, Γ) et $a \in E$, indique s'il existe un circuit passant par a et qui dans ce cas retourne un tel circuit. Complexité de calcul de cet algorithme ?

Algorithme CIRCUIT

Données : $(E, \Gamma), a$,
Résultats : $\exists CIRC$, $CIRC$

1: $X = \{a\}$, $T = \{a\}$, $\exists CIRC = F$, $CIRC = (\epsilon)$.
2: **while** $\exists x \in T$ et $\exists CIRC = F$ **do**
3: $T = T \backslash \{x\}$
4: **for all** $y \in \Gamma(x)$ **do**
5: **if** $y = a$ **then**
6: $\exists CIRC = V$
7: **end if**
8: **if** $y \notin X$ **then**
9: $X = X \cup \{y\}$
10: $T = T \cup \{y\}$
11: $P(y) = x$

12: **end if**
13: **end for**
14: **end while**
15: **if** $\exists CIRC = V$ **then**
16: **while** $x \neq a$ **do**
17: $CIRC = (x, CIRC)$
18: $x = P(x)$
19: **end while**
20: $CIRC = (a, CIRC, a)$
21: **end if**

E.1.2 Recherche de nouveaux algorithmes

On cherche à disposer d'algorithmes linéaires en temps de calcul pour résoudre le problème précédent. Pour cela on va faire une exploration en profondeur. On appelle profondeur d'un sommet x le plus petit nombre d'arcs qu'il faut pour aller de a à x. Une exploration en profondeurd d'abord consiste à explorer en priorité les sommets les plus profonds. Avec un telle technique il est possible de résoudre le problème en temps linéaire.

Algorithme Exploration-Profondeur

Données : $(E, \Gamma), a$,
Résultats : Aucun
Si L est une liste, on appelle $PREM(L)$ le premier élément de L, $RESTE(L)$ désigne la liste L amputée de son premier élément.

1: **for all** $x \in E$ **do**
2: $VISITE(x) = FAUX$
3: **end for**
4: $L = (a, \epsilon)$, $VISITE(a) = VRAI$
5: **while** $L \neq \epsilon$ **do**
6: $x = PREM(L)$, $y = PREM(\Gamma(x))$
7: **if** $y = \epsilon$ **then**
8: $L = RESTE(L)$
9: **end if**
10: **if** $y \neq \epsilon$ et $VISITE(y) = VRAI$ **then**
11: $\Gamma(x) = RESTE(\Gamma(x))$
12: **end if**
13: **if** $y \neq \epsilon$ et $VISITE(y) = FAUX$ **then**
14: $VISITE(y) = VRAI$, $L = (y, L)$
15: $\Gamma(x) = RESTE(\Gamma(x))$
16: **end if**
17: **end while**

E.2 Le problème de l'affectation

On désire affecter n personnes à n travaux. Chaque personne devant effectuer un travail et un seul. Le coût d'affectation de la personne i au travail j est d_{ij} (il peut s'agir par exemple d'un coût de formation). On cherche à affecter les personnes aux travaux de sorte que la somme des coûts soit minimale.
Proposer un graphe de résolution pour le problème de l'affectation. Quelle est la complexité de calcul de l'algorithme qui analyse toutes les solutions possibles.
Proposer un algorithme A^* pour résoudre le problème de l'affectation. Appliquer cet algorithme au cas où la matrice d_{ij} est donnée par :

$$\begin{bmatrix} 8 & 3 & 1 & 5 \\ 11 & 7 & 1 & 6 \\ 7 & 8 & 6 & 8 \\ 11 & 6 & 4 & 9 \end{bmatrix}$$

E.2.1 Correction

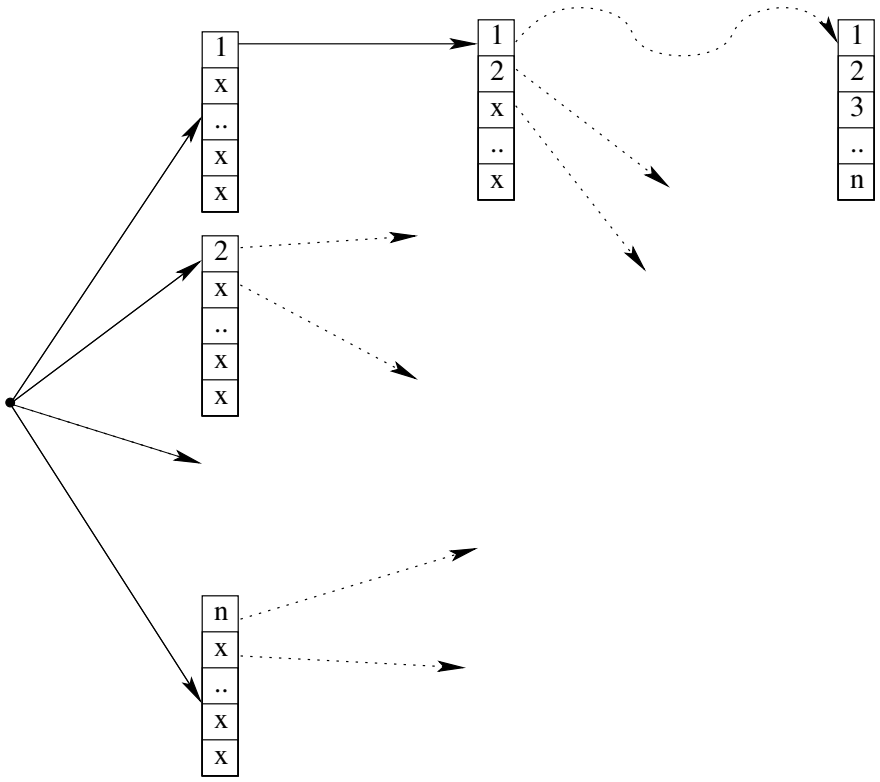


FIG. E.1 – *Problème d'affectation*

Sur la figure E.1, on peut voir que les noeuds *but* du GRP sont les différentes combinaisons des n personnes aux n travaux. Soit $n!$ combinaisons. Le parcours de toutes ces combinaisons individuellement est en $O(n!)$.

Pour l'algorithme A^* il faut appliquer la procédure Recherche-Avec-Graphe (**RAG**) en triant les sommets dans $OUVERT$ selon la fonction suivante :

$$f(x) = g(x) + h(x)$$

En prenant $g(x)$ coût optimal pour arriver au sommet x et $h(x)$ une estimation du coût restant. En pratique on peut prendre h heurisitique pour la personne i telle que :

$$h = \sum_{\text{j = taches restantes}} \min d_{ij}$$

Sur l'exemple donné, le meilleur arrangement est $(4, 3, 1, 2)$, son coût total est 19.

E.3 Le problème de la carte routière

On cherche à résoudre de façon automatique le problème suivant : étant données une carte routière, deux villes A et B, trouver la route la plus courte (en nombre de kilomètres) entre A et B. Trouver une bonne heuristique pour résoudre ce problème en utilisant un algorithme A^*. Spécifier de façons précise les données nécessaire au problème.

E.3.1 Correction

Pour tout sommet x du graphe (du réseau routier), on note $f(x) = g(x) + h(x)$ avec $g(x)$ la distance d'un chemin de A à x. On peut par exemple prendre comme heuristique $h(x)$ la distance euclidienne séparant le sommet x courant du sommet but B. Dans ce cas, on a besoin des coordonnées (latitudes et longitudes par exemple) des villes.

E.4 Le problème du voyageur de commerce

On cherche à résoudre le problème du voyageur de commerce, c'est à dire trouver un circuit de coût minimum passant une seule fois par chaque sommet. Pour cela on s'appuyera sur l'algorithme A^* pour lequel on donnera plusieurs heuristiques.

Annexe F

Graphes TP 2 - Analyse de trajectoires dans une chambre à bulle

Sujet

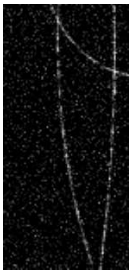


Image originale

Image seuillée

L'image ci-dessus à gauche est extraite d'une photographie de chambre à bulles (source CERN). A droite, l'image a été seuillée pour détecter les règions les plus claires (image *bs0*). Le fichier bs0.list contient les coordonnées des barycentres de ces régions (la première ligne du

fichier indique le nombre de points). Deux autres images sont à votre disposition : *bs1* et *bs2*, dont les fichiers de coordonnées sont respectivement *bs1.list* et *bs2.list*. Notre but dans ce TP est de reconstituer au mieux les trajectoires des particules (ensembles de points plus ou moins alignés) en négligeant les points isolés.

Question 1

Ecrire un programme qui lit un fichier de points, et construit un graphe antisymétrique sans boucle tel que :

- à chaque point correspond un sommet du graphe,
- deux sommets sont reliés par un arc si la distance entre les points correspondants est inférieure à un seuil d.

Correction

```
fd = fopen(argv[1], "r"); // ouverture fichier
if (!fd)
{
    fprintf(stderr, "cannot open file: %s\n", argv[1]);
    exit(0);
}
fscanf(fd, "%d", &ns);    // lit le nombre de sommets dans le fichier
na = (ns * (ns - 1)) / 2; // calcule le nombre d'arcs
g = InitGraphe(ns, na);
for (i = 0; i < ns; i++)
fscanf(fd, "%lf %lf", &(g->x[i]), &(g->y[i]));

fclose(fd);

// construit le graphe des distances sur les
// sommets de g (antisymetrique et sans boucle)
narc = 0;
for (i = 0; i < ns; i++)
    for (j = i+1; j < ns; j++)
    {
       AjouteArc(g, i, j);
       g->tete[narc] = i;
       g->queue[narc] = j;
       g->v_arcs[narc] =
      -DistanceEuclidienne(g->x[i], g->y[i], g->x[j], g->y[j]);
       narc++;
    }
```

Question 2

Visualiser les graphes obtenus pour différentes valeurs de d. Peut-on trouver une valeur qui permette d'isoler les points de bruit des trajectoires de particules, dans les trois images *bs0*,

bs1, *bs2* ?

La réponse est **Non**.

Question 3

Quelle est (approximativement) la valeur minimale d_{min} de d telle que le graphe obtenu à partir de *bs0.list* soit connexe ?
Proposez (sans l'implémenter) une méthode pour trouver la valeur exacte de d_{min} pour un ensemble quelconque de points.

Correction

Partir du graphe vide sur N sommets, ajouter les arcs par poids croissant jusqu'à ce que le graphe obtenu soit connexe.
Sur le graphe *bs0*, cette valeur est d'environ 55. On obtient alors le graphe de la figure F.1 page suivante.

Complexité de la méthode ?

Tri en $O(M \log M)$, avec M le nombre d'arcs ($M = N*(N-1)/2 \approx N^2$). Test de connexité en $O(n+m)$ pour n sommets et m arcs, d'où au pire

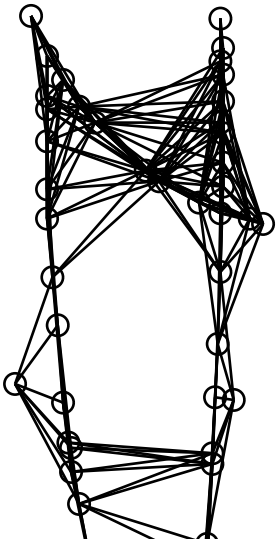
$$O(M \log M + N + 1 + N + 2 + \cdots + N + M) = O(M \log M + NM + M^2) = O(M^2)$$

Question 4

On suppose que les trajectoires se coupent entre elles et ne forment pas de cycle. Calculez sur le graphe valué par les distances un arbre de poids minimum. Pour cela, il faut implémenter un des algorithmes étudiés en cours (Kruskal 1, 2). Les fonctions de tri, test d'existence d'un cycle et test de connexité sont disponibles, ainsi que les fonctions de base pour ajouter ou retirer un arc, initialiser un nouveau graphe, etc.

Corrigé - Code de Kruskal 1

```
/* ================================================================== */
graphe * Kruskal1(graphe * g, graphe *g_1)
/* ================================================================== */
{
   int n = g->nsom;
   int m = g->narc;
   graphe * apm;    /* pour le resultat */
   graphe * apm_1;  /* pour la detection de cycles */
   int *A;          /* tableau pour ranger les index des arcs */
   int i, j, t, q;
   boolean *Ct = EnsembleVide(n);

   /* tri des arcs par poids croissant */
   A = (int *)malloc(m * sizeof(int)); /* allocation index */
```

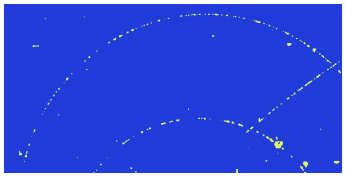


```
   if (A == NULL) {
      fprintf(stderr, "Kruskal1 : malloc failed\n");
      exit(0);
   }

   for (i = 0; i < m; i++) A[i] = i; /* indexation initiale */
   TriRapideStochastique(A, g->v_arcs, 0, m-1);

   /* construit le graphe resultat, initialement vide */
   apm = InitGraphe(n, n-1);
   apm_1 = InitGraphe(n, n-1);

   /* Boucle sur les arcs pris par ordre decroissant:
      on ajoute un arc dans apm, si cela ne cree pas
      un cycle dans apm.
      Arret lorsque nb arcs ajoutes = n-1. */

   j = 0;
   i = m - 1;
   while (j < n-1)
   {
      t = g->tete[A[i]]; q = g->queue[A[i]];
      CompConnexe(apm, apm_1, t, Ct);
      if (!Ct[q])
      {
         AjouteArc(apm, t, q);
         AjouteArc(apm_1, q, t);
         j++;
      }
      i--;
      if (i < 0) {
         fprintf(stderr, "Kruskal1 : graphe d'origine non connexe\n");
         exit(0);
      }
   }

   // recopie les coordonnees des sommets pour l'affichage
   if (g->x != NULL)
      for (i = 0; i < n; i++) {
         apm->x[i] = g->x[i];  apm->y[i] = g->y[i]; }

   free(A);
   free(Ct);
   TermineGraphe(apm_1);
   return apm;
} /* Kruskal1() */
```

Commentez les résultats obtenus sur les trois images *bs0*, *bs1*, *bs2*, en particulier : quels sont les défauts de la méthode ? Illustrez ces défauts par des exemples précis.

Deux types de défauts fréquents : les points de bruit qui créent des "branches", et ceux qui

génèrent des déviations dans les trajectoires. Egalement, des déviations peuvent être constatées près des croisements de trajectoires, et de mauvaises connexions peuvent être effectuées entre deux trajectoires proches l'une de l'autre.



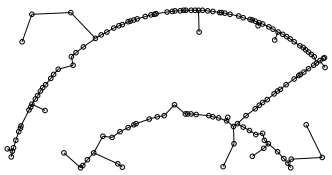


FIG. F.2 – Algorithme de KRUSKAL appliqué au graphe *bs1*

Question 5

On voit sur certaines configurations qu'il est nécessaire de tenir compte du fait que des points appartenant à une même trajectoire sont approximativement alignés.

On se propose de redéfinir la nature des poids associés aux arcs, de façon à favoriser la sélection des arcs qui non seulement sont courts, mais sont également alignés avec des arcs adjacents. De plus, cette définition devra être relativement simple à implémenter.

Une idée qui marche :

On considère le segment (i, j), avec i et j dans E, l'ensemble des points à relier. Soit i' le symétrique de i par rapport à j, et j' le symétrique de j par rapport à i. On peut prendre :

$$p = d(i, j) + \alpha * \left[\min_{k \in E \setminus \{i,j\}} d(i', k) + \min_{k \in E \setminus \{i,j\}} d(j', k) \right]$$

Le point i' (idem j') peut être interprété comme une "prédiction" de l'emplacement d'un prochain point de trajectoire, si le segment (i, j) fait effectivement partie d'une trajectoire.

Le paramètre α permet de donner plus ou moins d'importance à la distance ou à la linéarité des trajectoires. Typiquement, $\alpha = 1$.

Question 6

Proposez et implémentez une stratégie pour éliminer les points de bruit, c'est-à-dire les points qui ne font clairement partie d'aucune trajectoire.

Idée

Une branche est une partie de l'arbre située entre un point extrémité (un seul voisin) et un point multiple (ayant 3 voisins ou plus). On pourra éliminer les branches consituées de moins de K points, avec K entre 2 et 5 typiquement.

Annexe G

Graphes TP 3 - Ballades dans le métro

Position du problème

La RATP propose à ses usagers un service interactif, nommé Citéfut ?é, qui trouve le plus court trajet entre deux stations du métro au choix de l'utilisateur. Notre but dans ce TP sera de réaliser une version simplifiée d'un tel logiciel, et de réfléchir aux stratégies les plus efficaces pour résoudre des problèmes du même type.
Vous trouverez dans le fichier *metro_complet.graph* un graphe valué construit de la façon suivante :

- Chaque sommet correspond à une station pour une ligne donnée (par exemple, **République [ligne 3]** et **République [ligne 5]** sont deux sommets diff ?rents).
- A chaque sommet est associé la position de la station sur une carte (échelle $1cm \approx 25.7m$).
- A chaque sommet est associé le nom de la station (chaîne de caractères).
- Deux sommets forment un arc orienté si le métro relie directement les stations correspondantes (le graphe n'est pas symétrique à cause de quelques "sens uniques", par exemple du côté de la porte d'Auteuil). Cet arc est valué par le temps estimé du trajet en secondes (en prenant pour base une vitesse moyenne de $10m/s$, soit $36km/h$).
- Deux sommets forment forment deux arcs symétriques l'un de l'autre s'il est possible de passer à pied sans changer de billet entre les stations correspondantes. Ces arcs sont alors valués par une estimation du temps moyen de trajet et d'attente (120s).

Visualisez le graphe du métro parisien grâce au programme **graphe2ps.exe**

Dijkstra dans le métro

Voici le code d'une fonction realisant l'algorithme de Dijkstra dans le cas présent.

```
#define INFINITY LONG_MAX
/* ================================================================== */
/*! \fn void Dijkstra(graphe * g, int i)
    \param g (entrée) : un graphe pondéré. La pondération de chaque arc doit
                        se trouver dans le champ \b v_arc de la
                        structure \b cell .

    \param i (entrée) : un sommet de \b g.
```

```
    \brief calcule, pour chaque sommet x de g, la longueur d'un plus court
           chemin de i vers x. Cette longueur est stockée dans le champ
           \b v_sommets de \b g .
*/

void Dijkstra(graphe * g, int i)
/* ================================================================== */
{
   int n = g->nsom;
   boolean * S = EnsembleVide(n);
   int k, x, y;
   pcell p;
   TYP_VSOM vmin;

   S[i] = TRUE;
   for (k = 0; k < n; k++) g->v_sommets[k] = INFINITY;
   g->v_sommets[i] = 0;
   k = 1;
   x = i;
   while ((k < n) && (g->v_sommets[x] < INFINITY))
   {
      for (p = g->gamma[x]; p != NULL; p = p->next)
      { /* pour tout y successeur de x */
          y = p->som;
          if (!S[y])
          {
              g->v_sommets[y] =
                 min((g->v_sommets[y]),(g->v_sommets[x]+p->v_arc));
          } // if (!S[y])
      } // for p
      // extraire un sommet x hors de S de valeur g->v_sommets minimale
      vmin = INFINITY;
      for (y = 0; y < n; y++)
      if (!S[y])
         if (g->v_sommets[y] <= vmin) { x = y; vmin = g->v_sommets[y]; }
      k++;
      S[x] = TRUE;
   } // while ((k < n) && (g->v_sommets[x] < INFINITY))

} /* Dijkstra() */
```

Note : on peut ajouter un paramètre à cette procédure pour arrêter le travail dès qu'un sommet donné est atteint. Cela rendra plus efficace la recherche d'un plus court chemin d'un sommet d vers un sommet a.

Plus court chemin

```
/* ================================================================== */
/*! \fn graphe * PCC(graphe * g, int i, int a)
    \param g (entrée) : un graphe pondéré. La pondération de chaque arc
```

```
                        doit se trouver dans le champ \b v_arc de
                        la structure \b cell .

    \param i (entrée) : un sommet de \b g.
    \param a (entrée) : un sommet de \b g.
    \return un plus court chemin de d vers a sous forme d'un graphe.
    \brief trouve un plus court chemin de d vers a, et retourne ce
           chemin sous forme d'un graphe.
*/

graphe *PCC(graphe * g, int i, int a)
/* ================================================================== */
{
   int n = g->nsom;
   boolean * S = EnsembleVide(n);
   int k, s, t;
   pcell p;
   TYP_VARC vmin, v;
   graphe * pcc = InitGraphe(n, n-1); /* pour le resultat */
   graphe * g_1 = Symetrique(g);
   int n_som_explore = 0;

   // DIJKSTRA (avec arret des que le sommet a est atteint)

   S[i] = TRUE;
   for (k = 0; k < n; k++) g->v_sommets[k] = INFINITY;
   g->v_sommets[i] = 0;
   s = i;
   while ((s != a) && (g->v_sommets[s] < INFINITY))
   {
      n_som_explore++;
      pcc->v_sommets[s] = 1; // pour le coloriage des sommets explores
      for (p = g->gamma[s]; p != NULL; p = p->next)
      { /* pour tout t successeur de s */
          t = p->som;
          if (!S[t])
          {
             v = g->v_sommets[s]+p->v_arc;
             if (g->v_sommets[t] > v) g->v_sommets[t] = v;
          } // if (!S[t])
      } // for p

      // extraire un sommet s hors de S de valeur g->v_sommets minimale
      vmin = INFINITY;
      for (t = 0; t < n; t++)
      if (!S[t])
         if (g->v_sommets[t] <= vmin) { s = t; vmin = g->v_sommets[t]; }
      S[s] = TRUE;
    } // while ((s != a) && (g->v_sommets[s] < INFINITY))
    if (s != a)
    {
```

```
      printf("WARNING: pas de pcc trouve (s = %d , a = %d)\n", s, a);
      return NULL;
   }
   printf("pi(a) = %d\n", g->v_sommets[a]);
   printf("nb. sommets explores = %d\n", n_som_explore);

   // CONSTRUCTION DU PCC

   while (s != i)
   {
       for (p = g_1->gamma[s]; p != NULL; p = p->next)
       { /* pour tout t predecesseur de s */
          t = p->som;
          if ((g->v_sommets[s]-g->v_sommets[t]) == p->v_arc)
          {
              AjouteArcValue(pcc, t, s, p->v_arc);
              s = t;
              break;
          }
       } // for p
   }

   TermineGraphe(g_1);
   for (k = 0; k < n; k++) { pcc->x[k] = g->x[k]; pcc->y[k] = g->y[k]; }
   return pcc;

} /* PCC() */
```

Programme principal avec visualisation :

```
/* ========================================================================= */
void main(int argc, char **argv)
/* ========================================================================= */
{
   graphe *g, *pcc;
   int na, ns;     /* nombre d'arcs, nombre de sommets */
   int d, a;

   if (argc != 4)
   {
      fprintf(stderr, "usage: %s graphe_value d a\n", argv[0]);
      exit(0);
   }

   g = ReadGraphe(argv[1]);             /* lit le graphe a partir du fichier */
   d = atoi(argv[2]);
   a = atoi(argv[3]);
   pcc = PCC(g, d, a);

   // genere une figure en PostScript
```

```
   EPSGraphe(pcc, "pcc.ps", 4, 2, 25, 0, 0, 1, 0);

   TermineGraphe(g);
   TermineGraphe(pcc);

} /* main() */
```

Variante

Le plus simple est de modifier l'algorithme, mais attention, la stratégie ci-dessus pour construire le PCC ne marchera plus. Il faudra au cours de l'execution de Dijkstra mémoriser des pointeurs inverses (graphe auxiliaire).

Approche A^*

On désire maintenant résoudre le même problème par une approche A^*. Définissez un graphe de résolution de problème et proposez une ou plusieurs heuristiques A^*, sans tenir compte des temps d'arrêt aux stations.

Il suffit maintenant de modifier légèrement l'algorithme de Dijkstra pour implémenter la stratégie A^* pour ce problème. Re'alisez cette implémentation, et visualisez (voir EPSGraphe) l'ensemble des sommets explorés pour quelques exemples de trajets. Comparez le nombre de ces sommets avec le nombre de sommets explorés par Dijkstra.

```
#define INFINITY LONG_MAX
#define ECHELLE 25.7
#define VITESSE 10.0
/* ================================================================== */
/*! \fn graphe * PCCheuristique(graphe * g, int d, int a)
    \param g (entrée) : un graphe pondéré. La pondération de chaque arc
                        doit se trouver dans le champ \b v_arc de la
                        structure \b cell .
                        les coordonnées cartesiennes de chaque sommet
                        doivent se trouver dans les champs \b x et \b y
                        de \b g.

    \param d (entrée) : un sommet de \b g.
    \param a (entrée) : un sommet de \b g.
    \return un plus court chemin de d vers a sous forme d'un graphe.
    \brief trouve un plus court chemin de d vers a, et retourne ce
           chemin sous forme d'un graphe.
*/
graphe * PCCheuristique(graphe * g, int d, int a)
/* ================================================================== */
{
   int n = g->nsom;
   boolean * S = EnsembleVide(n);
   int k, s, t, x;
   pcell p;
   TYP_VARC vmin, v, h;
```

```
graphe * pcc = InitGraphe(n, n-1); /* pour le resultat */
graphe * g_1 = Symetrique(g);
int n_som_explore = 0;

// DIJKSTRA AVEC HEURISTIQUE

S[d] = TRUE;
for (k = 0; k < n; k++) g->v_sommets[k] = INFINITY;
g->v_sommets[d] = 0;
k = 1;
s = d;
while ((s != a) && (g->v_sommets[s] < INFINITY))
{
   n_som_explore++;
   pcc->v_sommets[s] = 1; // pour le coloriage des sommets explores
   for (p = g->gamma[s]; p != NULL; p = p->next)
   { /* pour tout t successeur de s */
      t = p->som;
      if (!S[t])
      {
          v = g->v_sommets[s]+p->v_arc;
          if (g->v_sommets[t] > v) g->v_sommets[t] = v;
      } // if (!S[t])
   } // for p

   // extraire un sommet s hors de S de valeur
   // g->v_sommets[s] + h(s)  minimale,
   // avec h(s) = dist(s,a)*ECHELLE/VITESSE
   vmin = INFINITY;
   for (t = 0; t < n; t++)
       if (!S[t])
       {
           h = (TYP_VARC)(dist(g->x[t],g->y[t],g->x[a],g->y[a])
                  * ECHELLE / VITESSE);
           v = g->v_sommets[t];
           if (v < INFINITY) v += h;
           if (v <= vmin) { s = t; vmin = v; }
        }
   k++;
   S[s] = TRUE;
} // while ((s != a) && (g->v_sommets[s] < INFINITY))

if (s != a)
{
   printf("WARNING: pas de pcc trouve \n");
   return pcc;
}
printf("pi(a) = %d\n", g->v_sommets[a]);
printf("nb. sommets explores = %d\n", n_som_explore);

// CONSTRUCTION DU PCC
```

```
   while (s != d)
   {
      for (p = g_1->gamma[s]; p != NULL; p = p->next)
      { /* pour tout t predecesseur de s */
         t = p->som;
         if ((g->v_sommets[s]-g->v_sommets[t]) == p->v_arc)
         {
             AjouteArcValue(pcc, t, s, p->v_arc);
             s = t;
             break;
         }
      } // for p
      if (p == NULL)
      {
          printf("WARNING: pas de pcc trouve \n");
          TermineGraphe(g_1);
          return pcc;
      }
   }
   TermineGraphe(g_1);
   for (k = 0; k < n; k++) { pcc->x[k] = g->x[k]; pcc->y[k] = g->y[k]; }
   return pcc;

} /* PCCheuristique() */
```

Programme principal avec visualisation :

```
/* ========================================================================= */
void main(int argc, char **argv)
/* ========================================================================= */
{
   graphe *g, *pcc;
   int na, ns;     /* nombre d'arcs, nombre de sommets */
   int d, a;

   if (argc != 4)
   {
      fprintf(stderr, "usage: %s graphe_value d a\n", argv[0]);
      exit(0);
   }

   g = ReadGraphe(argv[1]);             /* lit le graphe a partir du fichier */
   d = atoi(argv[2]);
   a = atoi(argv[3]);
   pcc = PCCheuristique(g, d, a);

   // genere une figure en PostScript
   EPSGraphe(pcc, "pcc.ps", 4, 2, 25, 0, 0, 1, 0);

   TermineGraphe(g);
```

```
   TermineGraphe(pcc);

} /* main() */
```

Exemple d'éxecution

Trajet : Eglise de Pantin - Mairie d'ivry. Le trajet est indique en trait noir sur la figure, les sommets explorées par l'algorithme A^* sont plein, les autres sont juste entourés.

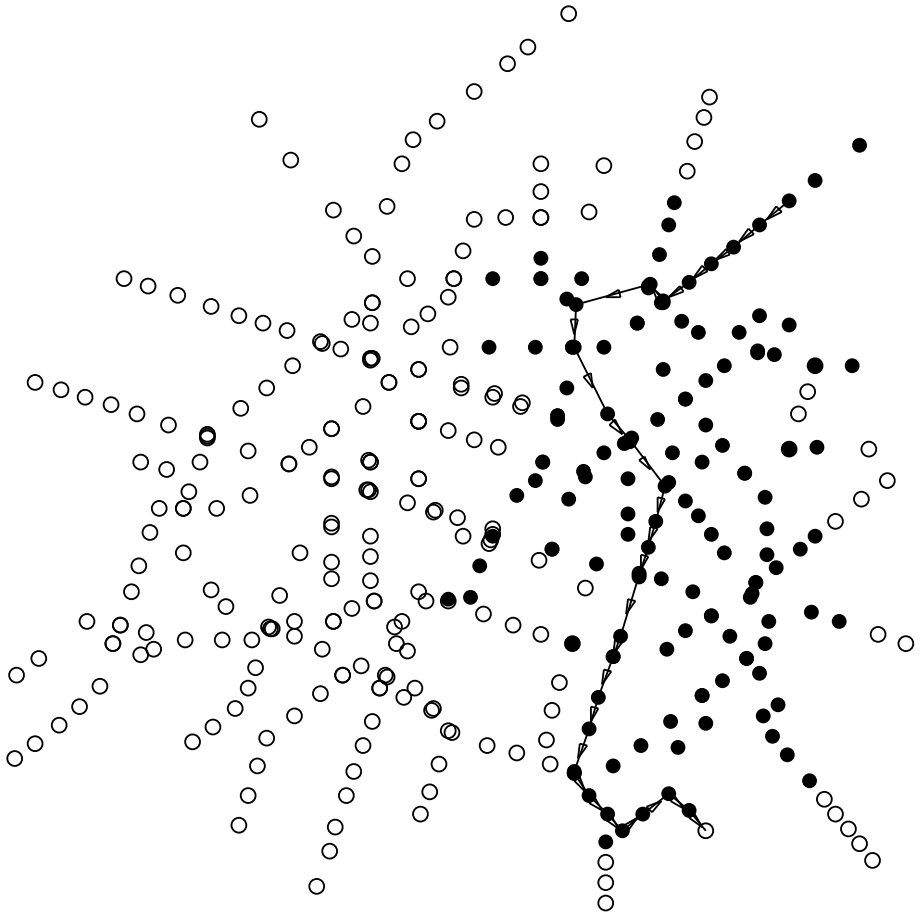


FIG. G.2 – Exemple d'application de l'algorithme A^*