Complexité des algorithmes
- *notes de cours* -

Jérôme Galtier et Alexandre Laugier

12 mars 2010

Table des matières

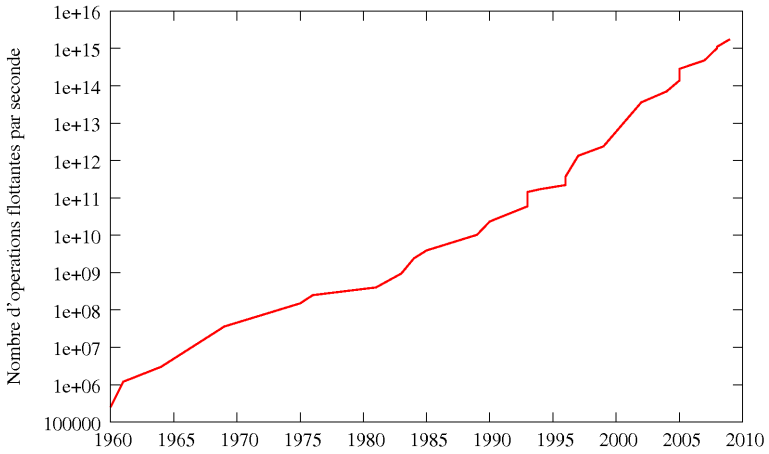

Figure 1: Puissance de calcul du meilleur ordinateur mondial au fil des ans.

L'algorithmique est une discipline sous-jacente à une partie immense de l'économie aujourd'hui. Le monde du travail a été radicalement transformé pour laisser une large place à l'activité assistée par ordinateur.

la complexité est en quelque sorte la maîtrise de l'algorithmique. Cette science vise à déterminer, sous un certain nombre de conditions, quel sont les temps de calcul et espace mémoire requis pour obtenir un résultat donné. Une bonne maîtrise de la complexité se traduit par des applications qui tournent en un temps prévisible et sur un espace mémoire controlé. A l'inverse, une mauvaise compréhension de la complexité débouchent sur des latences importantes dans les temps de calculs, ou encore des débordements mémoire conséquents, qui au mieux, "gêlent" les ordinateurs ("swap" sur le disque dur) et au pire font carrément "planter" la machine.

Sur la Fig. 1 on constate avec une échelle logarithmique la puissance considérable qu'ont acquis les ordinateurs modernes. On voit que les machines en l'espace de cinquante ans, ont amélioré leur capacités de calcul de moins d'un million d'opérations flottantes par seconde, à plus d'*un million de milliards* d'opérations flottantes par seconde. Cette révolution oblige à un changement profond de compréhension des algorithmes qui précisémment commandent ces machines.

En effet, un raisonnement simpliste pourrait consister à penser que la puissance de calcul est pratiquement infinie. De fait, beaucoup de programmes peuvent s'exécuter rapidement sur ces supercalculateurs, voire même sur les ordinateurs que nous cotoyons tous les jours. Cependant, la plupart des calculs que l'ont peut imaginer ne sont pas, loin s'en faut, simples. Par exemple, beaucoup d'opérations sur une liste font intervenir l'ordre de la liste en question. Or le nombre d'ordres possibles pour une liste de n éléments, noté $n!$, croît excessivement vite. Par exemple, supposons que nous voulions faire sur le meilleur calculateur existant aujourd'hui, au cours d'une seconde, une opération flottante sur toutes les listes ordonnées possibles à n éléments. Alors malheureusement, malgré toute notre puissance de calcul, il nous faudrait nous limiter aux listes à 17 éléments ou moins ($17! = 3,55\ 10^{14}$)!

Par ailleurs même si la puissance de calcul va croissant, le coût requis par opération

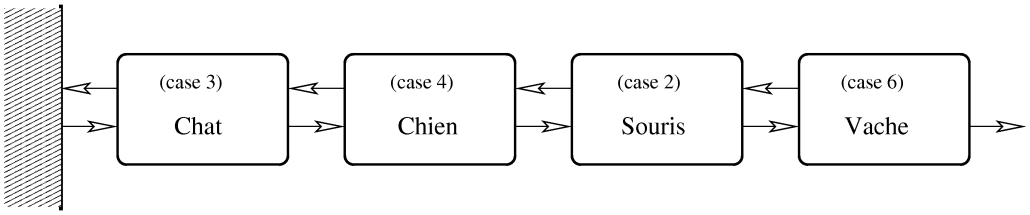

Figure 2: Une liste doublement chaînée.

flottante a tendance à stagner. Cela fait que la consommation énergétique due aux calculs devient phénoménale. On a estimé que la puissance énergétique totale consommée par les ordinateurs dans le monde était en 2005 de 200 TWh, ce qui correspond à la production d'une trentaine de tranches de centrales nucléaires à 800 MWatt chacune, ou encore, en termes d'empreinte carbone, de 30 million d'automobiles. L'ordinateur le plus puissant du monde en 2009, le Cray Jaguar, consomme 7 MWatt pour une puissance de calcul mesurée de 1,759 TFlops. Ce coût demandé par l'organisation Top500 n'est qu'un coût partiel de calcul. Les spécialistes du calcul haute performance estiment que très bientôt les coûts énergétiques totaux d'un supercalculateur seraient de l'ordre de 100 MWatt. Tout cela a un coût phénoménal, tant en termes de ressources financières que planétaires et humaines.

En résumé, les ressources de calcul d'aujourd'hui sont loin d'être infinies, et il convient de les estimer avec précision. Ce cours donne les bases d'une bonne analyse des temps de calcul et donne plusieurs exemples marquants de calcul complexes réalisés dans un temps maîtrisé.

1 Quelques notions d'algorithmique et compléxité élémentaire

1.1 Strutures de calcul

Un calcul bien mené utilise les structures appropriées. Nous examinons ici plusieurs structures simples qui permettent une analyse relativement facile.

1.1.1 La liste

La liste peut se voir comme une suite finie d'objets $a_1, \ldots, a_n$. Elle permet notamment:

- l'insertion, la supression d'éléments à toute place (en $O(1)$),
- la recherche d'un élément donné (en $O(n)$),
- la création, le test si la liste est vide ou non (en $O(1)$).

Une mise en œuvre simple de ce type de structure consiste à utiliser une liste doublement chaînée, c'est-à-dire une liste où on garde la mémoire de l'élément précédent et suivant de la liste. Dans l'exemple de la Fig. 2, cela donnera:

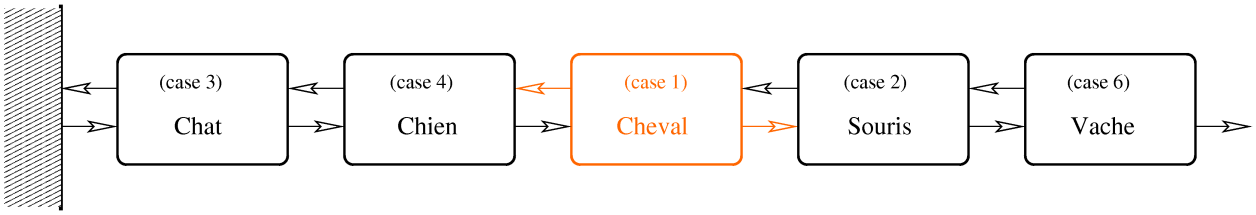

Figure 3: Liste de la Fig. 2 modifiée.

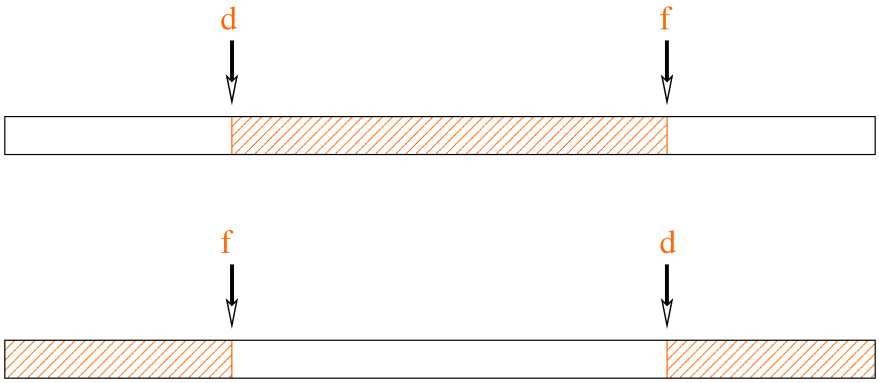

Figure 4: Une file placée dans un tableau.

Case	0	1	2	3	4	5	6
Valeur	(racine)		Souris	Chat	Chien		Vache
Suivant	3		6	4	2		
Précédent			4	0	3		2

Si on ajoute une valeur, par exemple comme indiqué sur la Fig. 3, alors le tableau de la liste deviendra:

Case	0	1	2	3	4	5	6
Valeur	(racine)	Cheval	Souris	Chat	Chien		Vache
Suivant	3	2	6	4	1		
Précédent		4	1	0	3		2

Cette manière de procéder a l'avantage de pouvoir insérer un élément avant ou aorès un autre (avant "souris" ou après "chien" dans notre exemple) sans se préoccuper de savoir précisémment dans quelle liste ces éléments figurent.

1.1.2 La pile

La pile est une structure où on insère les éléments d'un seul et même coté. On parle souvent de pile LIFO (pour *last-in first-out*): le dernier entré est le premier à ressortir.

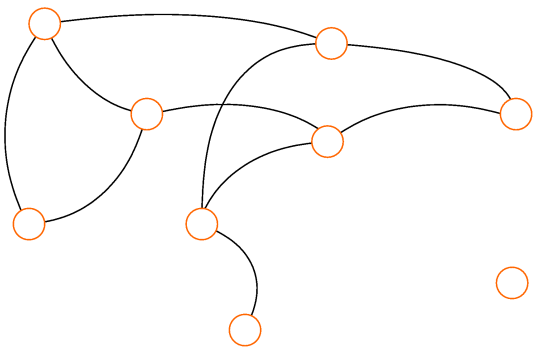

Figure 5: Un graphe non-orienté.

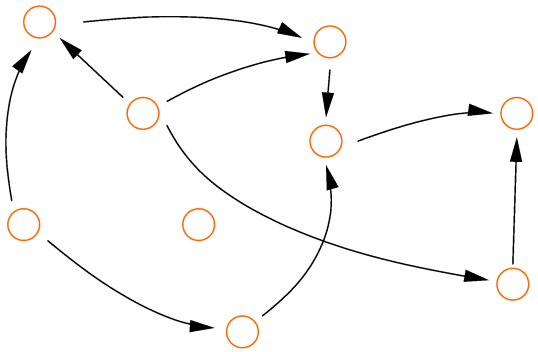

Figure 6: Un graphe orienté.

1.1.3 La file

La file est une structure où l'on peut insérer et supprimer de cotés opposés. Une application courante est la file FIFO (pour *first-in first-out*). On parle aussi de file *bilatère* pour mentionner cette propriété d'insertion et de suppression des deux cotés. Ces structures servent souvent en tant que "tampon" (ou *buffer* en anglais). Sur la Fig. 4 on voit deux états classiques d'une file rangée dans un tableau. Sur le haut de la figure le pointeur de début d se trouve avant le pointeur de fin f, ce qui signifie que la liste (en hachuré) se trouve au milieu. Sur le bas de la figure, c'est la situation inverse qui apparaît, avec une position des éléments de la file aux extrémités du tableau.

1.1.4 Graphes et arbres

Les graphes sont des structures très courantes en algorithmique. Il sont formés d'entités élémentaires que l'on appelle nœuds ou sommets. On rajoute entre ces sommets des connexions qui lient deux sommets, soit de manière non-orientée (on parle alors d'arête, et le graphe est dit non-orienté, voir Fig. 5), soit spécifiquement d'un sommet vers un autre (dans ce cas la connexion est appelée arc, et le graphe est un graphe orienté, encore appelé digraphe, voir Fig. 6).

Un graphe est dit connexe s'il existe toujours un chemin d'un sommet à un autre (on s'autorise, dans le cas des digraphes, à prendre des arcs à rebrousse-poil). Un digraphe est fortement connexe si il existe toujours un chemin d'un sommet vers un autre en respectant le sens des arcs. Un cycle est un chemin qui, partant d'un certain sommet, parcours un certain nombre d'autres sommets et revient au point de départ.

Un arbre est un graphe connexe sans cycle. On peut distiguer dans un graphe une racine qui est un sommet unique. L'existence d'une racine permet d'orienter toutes les arêtes du graphes de manière naturelle: on oriente l'arête depuis la racine vers les extrêmités de l'arbre.

1.2 Tri

Un algorithme de base pour le calcul de la compléxité est le tri. Supposins qu'une liste à trier se trouve sous la forme d'une liste doublement chaînée, comme expliqué au paragraphe 1.1.1. Une première méthode consiste à parcourir toute la liste, sélectionner lélément maximal, le supprimer de la liste courante, puis de l'insérer en tête d'une liste à trier. Cette méthode demande n opérations de recherche sur la liste courante de taille n, puis une opération de suppression, et enfin une opération d'insertion. Soit $n+2$ opérations pour passer d'une liste de taille nà une liste de taille $n-1$. Le temps total de calcul est alors, en nombre d'opérations, de:

$$(n+2)+(n+1)+(n)+(n-1)+..+3=\sum_{i=1}^{i=n} i+2=\frac{n^2+5n}{2}.$$

Une méthode alternative consiste à utiliser un *tas*. Le tas se forme sur une structure d'arbre particulière appelée arbre binaire quasi-complet.

Soit G un arbre non-orienté et muni d'une racine. Appelons père le voisin d'un sommet qui se trouve du coté de la racine, et fils ses autres voisins. Désignons par feuille un sommet qui n'a pas de fils. Appelons niveau d'un sommet la distance, en termes de longueur de chemin, qui le sépare de la racine.

Un *arbre binaire complet* est un arbre doté d'une racine, dont toutes les feuilles sont au même niveau, et dont tous les autres sommets ont exactement deux fils. Un arbre binaire quasi-complet est un arbre complet auquel on a éventuellement supprimé un certain nombre de feuilles, mais uniquement "en bas à droite" (voir la Fig. 7). Alors qu'un arbre binaire complet possède $2^{h+1}-1$ sommets, où h représente le niveau des feuilles, un arbre binaire quasi-complet peut avoir un nombre quelconque de feuilles. On ajoute un sommet en placant une feuille à la position inoccupée la plus à gauche sur le dernier niveau incomplet (et éventuellement en commaçant un nouveau niveau si nécessaire. On retire un sommet en supprimant le sommet le plus à droite du dernier niveau. La Fig. 8 donne toutes les topologies d'arbres binaires quasi-complets de 1 à 16 sommets.

Munissons maintenant chaque sommet de l'arbre d'une valeur numérique. On dira que l'arbre est bien ordonné si la valeur d'un père est toujours inférieure à celle de ses fils. Un *tas* est un arbre binaire quasi-complet bien ordonné (voir Fig. 9). Le tas est une structure qui permet pour un ensemble de n éléments

(1) d'ajouter un nouvel élément en $O(\log(n)$ opérations,

(2) d'extraire le minimum du tas en $O(\log(n)$ opérations.

Pour insérer un nouvel élément dans un tas, on s'autorise dans un premier temps à violer la règle de bien-ordonnancement pour obtenir un un arbre binaire quasi-complet ayant le bon nombre de sommets. Par exemple, si on introduit un nouveau sommet ayant la valeur 7 à l'arbre de la Fig. 9, on obtient dans un premier temps l'arbre de la Fig. 10, qui est quasi-complet. Ensuite, on rétablit le bon-ordonnancement en partant du sommet introduit et en l'échangeant avec son père si la valeur du père est supérieure. Cette opération rétablit

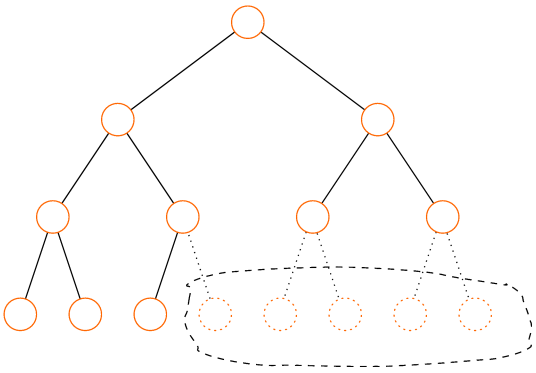

Figure 7: Un arbre binaire quasi-complet: on part d'un arbre binaire complet, et on supprime des feuilles en bas à droite (ici, 5 feuilles).

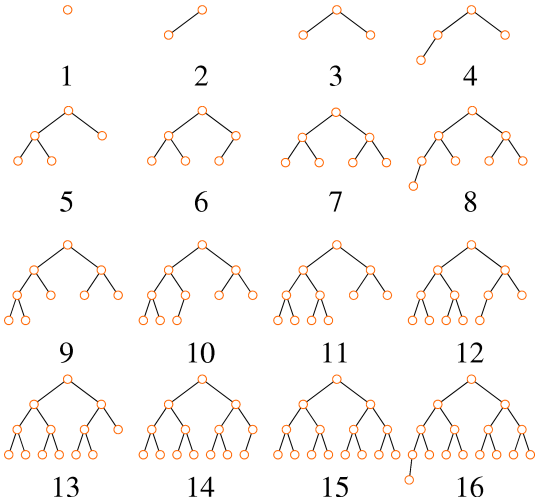

Figure 8: Topologies des arbres binaires quasi-complets de 1 à 16 sommets.

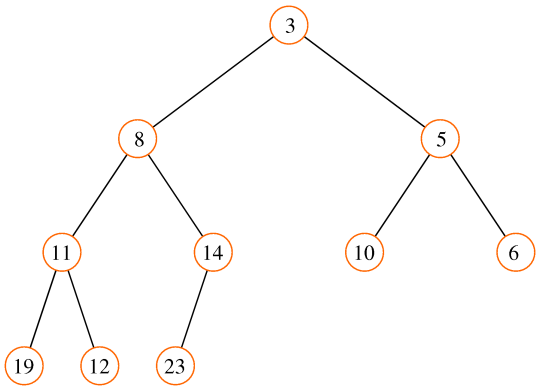

Figure 9: Un tas.

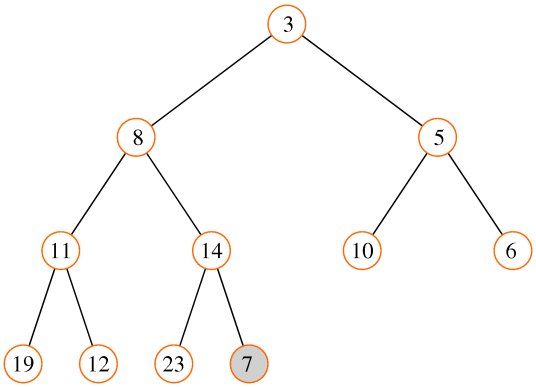

Figure 10: Arbre binaire quasi-complet résultant de l'addition de la valeur "7".

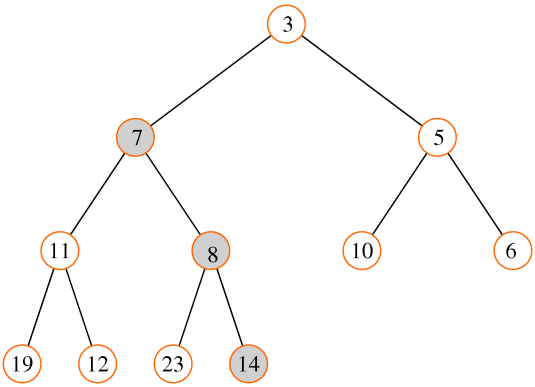

Figure 11: Tas obtenu après l'addition de la valeur "7".

le bon-ordre sur le lien père/fils mais peut éventuellement introduire une violation de la règle de bien-ordonnancement sur le lien grand-père/père. On échange dans ce cas les sommets père/grand-père, et ainsi de suite éventuellement jusqu'à la racine. Dans le cas le pire, on remonte le chemin jusqu'à la racine, soit $O(\log(n))$ opérations. Dans notre exemple, les sommets des valeurs 7 et 14 sont d'abord échangés, puis les sommets des valeurs 8 et 7, pour donner le tas de la Fig. 11.

L'autre opération majeure est l'extraction de la valeur minimale. La valeur minimale se trouve à la racine. Dans un premier temps, comme le montre la Fig. 12, on maintient simplement la structure d'arbre binaire quasi-complet en déplaçant le contenu de la dernière feuille (en bas à droite) sur la racine. Cette dernière feuille est donc supprimée. On rétablit ensuite le bon-ordonnancement de la manière suivante: le contenu du sommet à mettre à jour est comparé à ses deux fils, et est échangé avec son fils de valeur minimale. On déplace alors le problème d'ordonnancement au fils choisi pour l'échange et sa descendance. On continue à échanger les valeur de ce sommet avec le plus petit de ses fils jusqu'à ce que l'arbre soit ordonné, comme le montre la Fig. 13. L'opération se fait en $O(\log(n))$.

Au final, le tri par tas sur une liste de n élément commence par n insertions, suivies de n extractions de la valeur minimale, pour un total de $O(n \log(n))$ opérations.

1.3 Chemins Eulériens et Hamiltoniens

Le problème du chemin dit Eulérien est apparu en rapport avec la ville de Königsberg (aujourd'hui Kaliningrad), dont une carte sommaire est donné en Fig. 14. La question était de savoir s'il existait une promenade passant une et une seule fois par tous les ponts (représentés en brun).

Leonhard Euler a résolu cette question en 1735 en démontrant qu'une telle promenade d'existait pas. Sa démonstration repose sur une représentation en graphe du problème (Fig. 15). Il s'agit alors de trouver un chemin ou un cycle qui passe une et une seule fois par toutes les arêtes, moyennant un passage éventuel plusieurs fois par le même sommet. La démonstration d'Euler repose sur la constatation que (1) si un tel chemin ou cycle

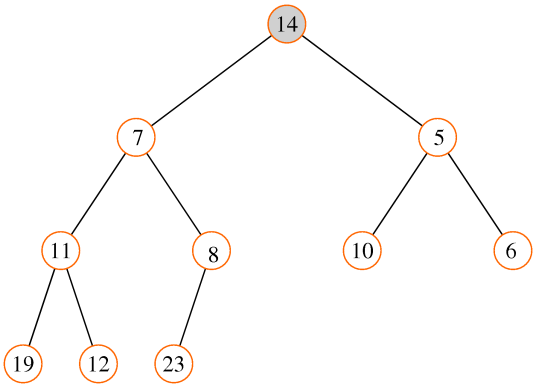

Figure 12: Arbre binaire quasi-complet résultant de l'extraction de la valeur "3".

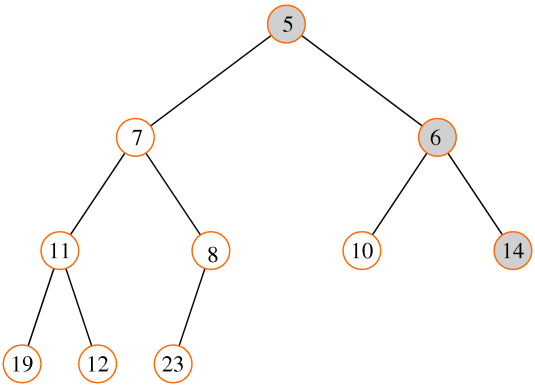

Figure 13: Tas obtenu après l'extraction de la valeur "3".

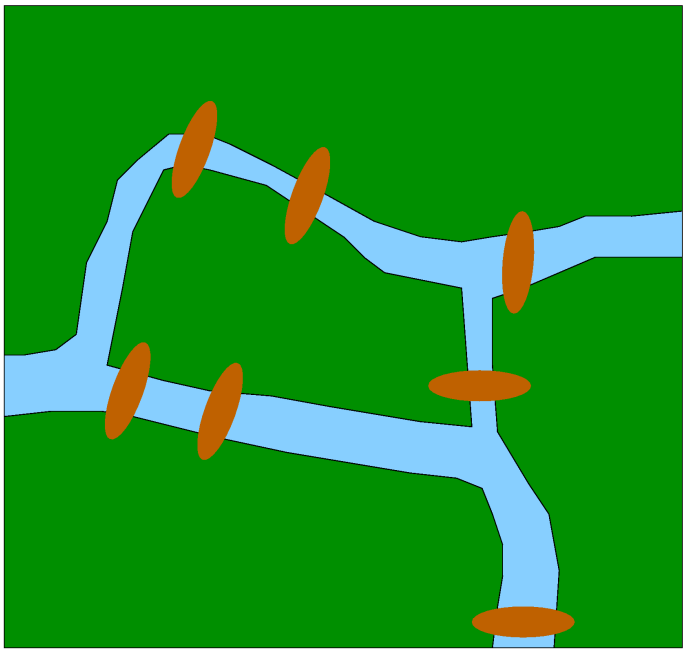

Figure 14: Topographie de la ville de Königsberg (aujourd'hui Kaliningrad).

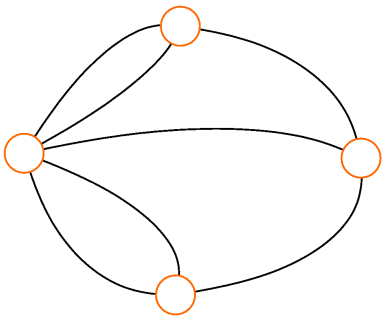

Figure 15: Graphe correspondant au problème d'Euler.

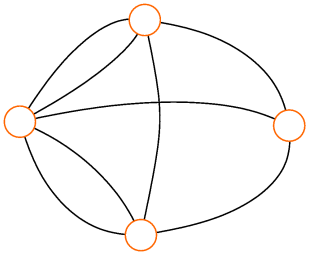

Figure 16: Un graphe légèrement modifié.

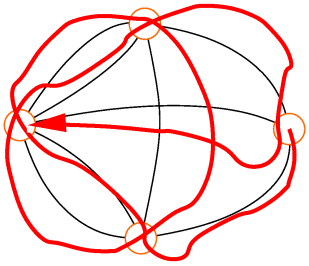

Figure 17: Un chemin eulérien.

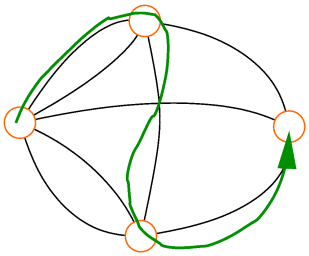

Figure 18: Un chemin hamiltonien.

existe, alors tous les sommets sauf éventuellement les sommets de départ et d'arrivée ont un nombre pair d'arêtes adjacentes. De plus, (2) tous les sommets non-isolés sont alors dans la même composante connexe du graphe. Euler a démontré que les conditions (1) et (2) étaient également suffisantes pour s'existence d'un tel chemin. Ainsi, la Fig.15 n'a pas de chemin eulérien puisque quatre sommets ont un nombre impair d'arêtes adjacentes.

Une variante de ce problème concerne l'existence d'un chemin ou d'un cycle hamiltonien: il s'agit d'un cycle ou chemin sur le graphe qui passe une et une seule fois par tous les sommets. Nous montrons sur un graphe légèrement modifié (Fig. 16) un chemin eulérien (Fig. 17) et un chemin hamiltonien (Fig. 18). Nous allons voir dans la suite qu'en dépit des apparences, le problème qui consiste à trouver un chemin eulérien est considérablement plus simple que celui du chemin hamiltonien.

2 Modélisation de la complexité

Cette partie du cours s'inspire essentiellement de [4].

2.1 Machines de Turing

Alan Turing
(1912-1954)

Une machine de Turing est un modèle simplifié d'ordinateur qui permet d'évaluer ce qui est faisable en matière de calcul. Bien que très élémentaire, elle représente une première approche fondamentale pour la théorie de la compléxité.

La machine, comme indiqué sur la Fig. 19, est un ordinateur qui lit et écrit des symboles sur une bande enregistreuse de longueur infinie, et le fait à la position du curseur où elle se trouve. La machine lit et écrit un seul symbole à la fois en suivant des règles précises. Ces règles dépendent d'un "état" dans lequel se trouve la machine, ainsi que du symbole qui est lu. La machine possède un nombre fini d'états.

Formellement, on se donne donc, pour définir la machine de Turing:

- un ensemble Q d'états,
- un ensemble Σ de symbole,

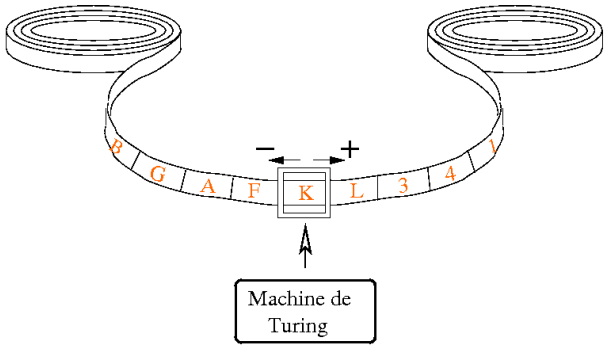

Figure 19: Une machine de Turing.

- un état initial ι,
- un symbole distingué $\sqcup \in \Sigma$ (appelé le symbole "blanc"),
- un ensemble d'états finaux $\mathcal{A} \subseteq Q$,
- une fonction de transition δ, de $Q \times \Sigma$ dans $Q \times \Sigma \times \{-1, +1\}$. Son sens est le suivant: si la machine est dans l'état q_0 et qu'elle lit le symbole σ_0 à la position courante du curseur, et en écrivant $\delta(q_0, \sigma_0) = (q_1, \sigma_1, \varepsilon_1)$, alors la machine va écrire à l'emplacement du curseur le symbole σ_1, passe à l'état q_1. Par ailleurs, elle avance le curseur sur la bande d'une case si $\varepsilon_0 = +1$ et le recule d'une case si $\varepsilon_0 = -1$.

Dans le cas d'une **machine de Turing déterministe**, le fonctionnement est le suivant. On fournit à la machine le problème initial, par exemple le graphe sur lequel un chemin eulérien ou hamiltonien doit être calculé, ce qui est exprimé avec un certain encodage sur les cases 0 à n de la bande de la machine de Turing, et la machine effectue son calcul, et on se demande si on peut garantir que la machine va au bout d'un certain temps répondre par VRAI ou FAUX sur la nature de problème, par exemple à la question existe t-il un chemin eulérien ou hamiltonien, va t-elle répondre VRAI ou FAUX dans un temps déterminé.

Si par contre on s'intéresse à une **machine de Turing non-déterministe**, alors on s'autorise, une fois que le problème a été écrit, à rajouter des cases magiques par exemple avant la case 0, qui encode une solution devinée du problème. Dans le cas d'une question sur l'existence d'un chemin, on pourra fournir l'encodage complet d'un chemin donnant la solution du problème. La machine vérifie que les cases magiques fournissent bien une solution au problème, et répond par VRAI ou FAUX. La question est de savoir si, étant donné un problème ayant une solution, il existe au moins une affectation des cases magiques qui atteste de l'existence de cette solution en un temps déterminé. Par exemple, si un chemin eulérien ou hamiltonien existe dans le graphe, si on fournit à la machine sur sa bande l'encodage du graphe et d'un chemin valide, la machine peut-elle en un temps déterminé attester la validité de ce chemin?

Dans les deux cas, lorsque la machine est dans un état final ($q \in \mathcal{A}$), elle s'arrête.

2.2 Les classes P et NP

La **classe** P est l'ensemble des problèmes pour lesquels une machine de Turing déterministe peut trouver une solution en un temps polynomial. La **classe NP** est l'ensemble des problèmes pour lesquels une machine de Turing non-déterministe peut trouver une solution en temps polynomial.

2.3 Le problème SAT et le théorème de Levin-Cook

Le problème SAT est un problème de décision. On se donne une liste de variables booléennes (i.e. prenant la valeur "VRAI" ou "FAUX") $X_1, \ldots, X_p$, et une certaine expression $Exp(X_1, \ldots, X_d)$ à satisfaire, composée uniquement des variables $X_1, \ldots, X_d$, des opérations binaires ET (noté $\wedge$), et OU (noté $\vee$), et de l'opérateur unitaire NON (noté $\neg$). Pour la clarté des expressions, on notera aussi les parenthèses. Le problème est le suivant:

SAT: Existe t-il une assignation des variables $(X_1, \ldots, X_d)$ *dans* $(VRAI, FAUX)^d$ *telle que* $Exp(X_1, \ldots, X_d)$ *soit vraie?*

Par exemple la formule $(\neg X) \vee Y$ demande que soit X =FAUX, soit Y =VRAI. On désigne souvent cette relation par "X implique Y" (notée $X \Longrightarrow Y$): si X est VRAI, alors nécessairement Y est VRAI. Un certain nombre de relations élémentaires de logique méritent d'être rappelées. Si A, B et C sont des expressions booléennes, alors les relations suivantes sont vraies.

(i) $\neg(A \vee B) \quad = \quad (\neg A) \wedge (\neg B)$,

(ii) $\neg(A \wedge B) \quad = \quad (\neg A) \vee (\neg B)$,

(iii) $(A \wedge B) \vee C \quad = \quad (A \vee C) \wedge (B \vee C)$,

(iv) $(A \vee B) \wedge C \quad = \quad (A \wedge C) \vee (B \wedge C)$.

Usant de ces relations, il est possible de mettre l'expression $Exp(X_1, \ldots, X_p)$ dans une forme canonique, où l'opérateur dominant est ET, et l'opérateur intermédiaire est OU, et l'opérateur dominé est NON. En effet, il suffit dans un premier temps, si une forme $\neg(A \vee B)$ ou $\neg(A \wedge B)$ apparaît dans la formule, d'appliquer respectivement (i) ou (ii). Puis, dans un deuxième temps, si une forme $(A \wedge B) \vee C$ apparaît dans l'expression, on peut la remplacer en utilisant (iii).

On obtient aussi une formule sous la forme de conjonction (AND) de clauses. Chaque clause est formée de disjonctions de variables ou de négation de variables. *C'est cette forme canonique qui sert de référence pour calculer la taille de l'expression* $Exp(X_1, \ldots, X_d)$.

Définition 2.1 *Une instance de SAT* $Exp(X_1, \ldots, X_d)$ *est de taille* n *si la somme du nombre de variables qui apparaît par clause est égal à* n.

Définition 2.2 *On dit qu'un problème* $\mathcal{P}$ *est NP-complet s'il est lui-même NP et que tout problème NP se réduit à* $\mathcal{P}$ *en temps polynomial.*

Théorème 2.1 (Levin-Cook) *SAT est NP-complet*

PREUVE. On se donne la liste de variables suivante:

Variable	Signification	Nombre de variables
$S_{i,j,k}$	Vrai si la case i de la bande contient le symbole j à l'étape k du calcul	$O(p(n)^2)$
$H_{i,k}$	Vrai si la tête de lecture/écriture de la machine est sur la case i de la bande à l'étape k	$O(p(n)^2)$
$Q_{q,k}$	Vrai si la machine est dans l'état q à l'étape k du calcul	O(n(p))

L'expression du problème booléen de SAT comprendra alors une première partie de clauses qui garantiront les conditions générales de fonctionnement de la machine:

Clauses	Conditions	Interprétation	Nombre de Clauses
$S_{i,j,0}$	La case i des données d'entrée contient le symbole j	Contenu initial de la bande	$O(p(n))$
$Q_{\iota,0}$		Etat initial de la machine	1
$H_{0,0}$		Position initiale de la tête de lecture/écriture	1
$\bigvee_{j\in\Sigma} S_{i,j,k}$		Au moins un symbole existe sur chaque case de la bande à l'étape k	$O(p(n)^2)$
$\neg S_{i,j,k} \vee \neg S_{i,j',k}$	$j \neq j'$	Un seul symbole par case de la bande à l'étape k	$O(p(n)^2)$
$\bigvee_i H_{i,k}$		La tête de lecture est au moins à un emplacement à l'étape k	$O(p(n)^2)$
$\neg H_{i,k} \vee \neg H_{i',k}$	$i \neq i'$	La tête de lecture ne peut être qu'à un emplacement au plus à l'étape k	$O(p(n))$
$\bigvee_{q\in\mathcal{Q}} Q_{q,k}$		La machine est au moins dans un état à l'étape k	$O(p(n)^2)$
$\neg Q_{q,k} \vee \neg Q_{q',k}$	$q \neq q'$	La machine est au plus dans un état à l'étape k	$O(p(n))$
$\neg S_{i,j,k} \vee H_{i,k} \vee S_{i,j,k+1}$		Une case de la bande qui n'est pas sous la tête de lecture/écriture reste inchangée	$O(p(n)^2)$
$\bigvee_{f\in\mathcal{A}} Q_{f,p(n)}$		La machine doit finir dans un état terminal	$O(1)$

Il reste ensuite à garantir le fonctionnement de la machine elle-même sur la bande. Cela se fait par les conditions suivantes, qui parcourent toutes les valeurs possibles de i, j, k, q

pour $i \in \{-p(n), \ldots, 0, \ldots, p(n)\}$, $j \in \Sigma$, $k \in \{0, \ldots, p(n)\}$, et $q \in \mathcal{Q}$. On pose par ailleurs $\delta(i,j) = (q', j', \Delta)$. les clauses suivantes apparaissent alors, devenant actives si et seulement si à l'étape k, la machine est dans l'état q en position i et lit le symbole j.

Clauses	Effet si à l'étape k, la machine est dans l'état q en position i et lit le symbole j	Nombre de Clauses
$\neg H_{j,k} \vee \neg Q_{q,k} \vee \neg S_{i,j,k} \vee H_{j+\Delta,k+1}$	la tête se déplace de Δ à l'étape $k+1$	$O(p(n)^2)$
$\neg H_{j,k} \vee \neg Q_{q,k} \vee \neg S_{i,j,k} \vee Q_{q',k+1}$	la machine passe à l'état q' à l'étape $k+1$	$O(p(n)^2)$
$\neg H_{j,k} \vee \neg Q_{q,k} \vee \neg S_{i,j,k} \vee S_{i,j',k+1}$	la case i de la bande contient le symbole j' à l'étape $k+1$	$O(p(n)^2)$

En conclusion, si il est possible de valider une solution d'un problème sans connaître les cases magiques, alors le prblème SAT décrit ci-dessus nous donnera les valeurs de ces cases pour trouver une solution. Donc tout problème de NP (vérifiable par une machine de Turing non-déterministe) peut se ramener à un problème SAT en temps polynomial.

Si par ailleurs il existait un algorithme polynomial pour résoudre SAT, alors on pourrait, en connaissant la machine de Turing qui vérifie la solution, retrouver une solution valide au problème initial. ∎

2.4 Autres problèmes NP-Complets

Nous donnons dans ce qui suit une série de réductions fondamentales vers des problèmes NP-complets majeurs.

2.4.1 3-SAT

Un problème 3-SAT est un problème SAT où toutes les clauses ont exactement trois termes.

Théorème 2.2 *3-SAT est NP-complet.*

Preuve. Il suffit de montrer qu'on peut ransformer un problème SAT en un problème 3-SAT en temps polynomial. Soient $C_1, \ldots, C_k$ les clauses du problème SAT. On ajoute pour chaque clause C_i, $||C_i| - 3|$ nouvelles variables, notées $Y_{i,j}$ pour $j \in \{1, \ldots, ||C_i| - 3|\}$.

Si $|C_i| = 1$, alors on crée pour le problème 3-SAT les clauses

$$\begin{aligned} K_i^1 &= C_i \vee Y_{i,1} \vee Y_{i,2} \\ K_i^2 &= C_i \vee Y_{i,1} \vee \neg Y_{i,2} \\ K_i^3 &= C_i \vee \neg Y_{i,1} \vee Y_{i,2} \\ K_i^4 &= C_i \vee \neg Y_{i,1} \vee \neg Y_{i,2}. \end{aligned}$$

Si $|C_i| = 2$, alors on crée pour le problème 3-SAT les clauses

$$\begin{aligned} K_i^1 &= C_i \vee Y_{i,1} \\ K_i^2 &= C_i \vee \neg Y_{i,1}. \end{aligned}$$

Si $|C_i| = 3$, alors $K_i^1 = C_i$.

Si $|C_i| > 3$, notons $C_i = Z_1 \vee \dots Z_p$ où Z_j est un littéral ou une inversion de littéral, et on forme:

$$\begin{aligned} K_i^1 &= Z_1 \vee Z_2 \vee Y_{i,1} \\ K_i^2 &= Y_{i,1} \vee Z_3 \vee Y_{i,2} \\ &\vdots \\ K_i^{j+1} &= Y_{i,j} \vee Z_{j+2} \vee Y_{i,j+1} \\ &\vdots \\ K_i^{p-2} &= Y_{i,p-3} \vee Z_{p-1} \vee Z_p. \end{aligned}$$

Au final, la conjonction des clauses K_i^j définit notre problème 3-SAT. ∎

2.4.2 VERTEX COVER

Définition 2.3 *Soit $G = (V, E)$ un graphe non-orienté. Un vertex cover $S \subseteq V$ est un ensemble intersectant toutes les arêtes, c'est-à-dire*

$$\forall e \in E \quad S \cap e \neq \emptyset.$$

Le problème VERTEX COVER est le suivant: étant donnés un graphe G et un entier k, existe t-il un VERTEX COVER S tel que $|S| \leq k$?

Théorème 2.3 *VERTEX COVER est un problème NP-complet.*

Preuve. Le problème VERTEX COVER est évidemment NP, car si on nous donne un ensemble $S \subseteq V$ de taille k réputé être la solution du problème, il est facile de le vérifier en temps polynomial.

On se donne un problème 3-SAT, composé des littéraux $X_1, \dots, X_n$, et des clauses $C_1, \dots, C_m$ et on montre qu'on peut construire à partir de cette instance un graphe G et un entier k tels que la résolution du problème VERTEX COVER entraîne la résolution de notre problème initial en temps polynomial.

Le graphe G que nous construisons pour cela, illustré sur la Fig. 20 aura les $2n + 3m$ sommets suivants: u_i^0, u_i^1, pour $i \in \{1, \dots, n\}$ et v_j^1, ,v_j^2, v_j^3, pour $j \in \{1, \dots, m\}$. Par ailleurs, son ensemble d'arêtes sera

$$E = \bigcup_{i\{1,\dots,n\}} \{\{u_i^0, u_i^1\}\} \cup \bigcup_{i\{1,\dots,n\}} \{\{v_j^1, v_j^2\}, \{v_j^1, v_j^3\}, \{v_j^2, v_j^3\}\} \cup \bigcup_{j\{1,\dots,m\}} W_j.$$

et nous précisons W_j dans ce qui suit.

Notons par x_j^1, x_j^2, x_j^3 les indices des littéraux présents dans la clause C_j et notons $z_j^1 = 1$ si le premier terme est une négation du littéral, $z_j^1 = 0$ sinon. De même pour z_j^2 et z_j^3 pour les littéraux d'indice x_j^2 et x_j^3. On pose alors:

$$W_j = \{\{u_{x_j^\alpha}^{z_j^\alpha}, v_j^\alpha\}\}_{\alpha=1,2,3}.$$

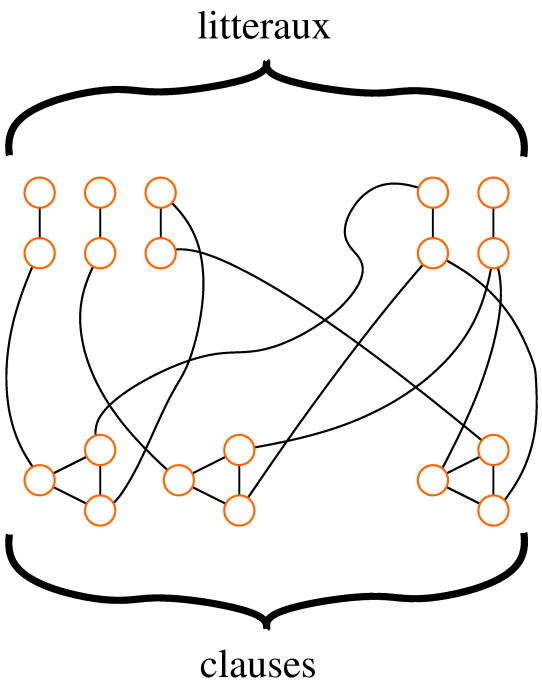

Figure 20: Construction d'un VERTEX COVER à partir de 3-SAT.

Fixons maintenant $k = n + 2m$. Comme pour chaque i, l'arête $\{u_i^0, u_i^1\}$, nécessairement ou u_i^0 ou u_i^1 est dans S, soit au minimum n sommets. De même, on peut voir que pour chaque j, deux des sommets $\{v_j^1, v_j^2, v_j^3\}$ sont dans S, soit au minimum $2m$ sommets. Donc si $|S| = n + 2m$, il contient nécessairement pour chaque $i \in \{1, \ldots, n\}$ un et un seul sommet parmi $\{u_i^0, u_i^1\}$ et pour chaque $j \in \{1, \ldots, m\}$, deux et seulement deux sommets dans $\{v_j^1, v_j^2, v_j^3\}$.

Le sommet qui n'est pas pris dans $\{v_j^1, v_j^2, v_j^3\}$ représente le littéral qui est vérifié dans la clause. L'affectation des sommets $\{u_i^0, u_i^1\}$ garantit l'unicité de la valeur VRAI ou FAUX pour un littéral donné. Ainsi, un VERTEX COVER de taille $n+2m$ fournit bien une solution à 3-SAT. ∎

2.4.3 CYCLE HAMILTONIEN

Le problème est le suivant: étant donné un graphe non-orienté G existe t-il un cycle hamiltonien parcourant G?

Théorème 2.4 *CYCLE HAMILTONIEN est un problème NP-complet.*

PREUVE. Il est une fois de plus évident que si on nous propose un certain chemin sur un graphe, on sait vérifier en temps polynomial que c'est un cycle et qu'il est bien hamiltonien. Donc CYCLE HAMILTONIEN est NP.

Pour établir la complétude du problème, on se donne un certain graphe $G = (V, E)$ et un entier k correspondants à un problème de VERTEX COVER, et on montre que ce problème

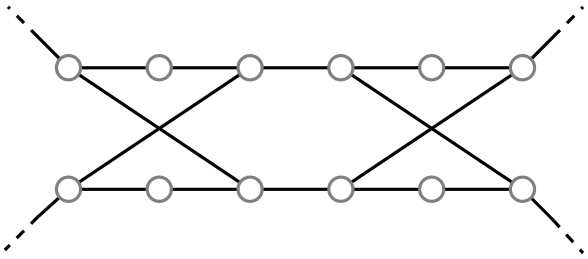

Figure 21: Un gadget pour le cycle hamiltonien.

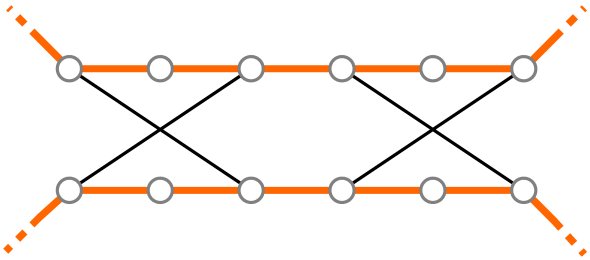

Figure 22: Un parcours possible du gadget.

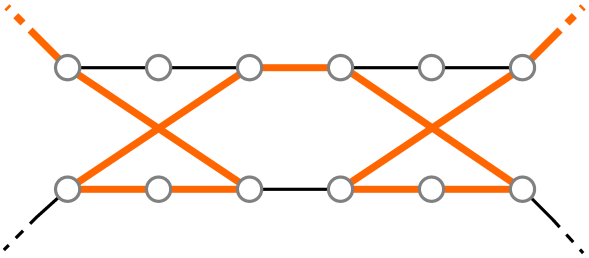

Figure 23: Un autre parcours possible du gadget.

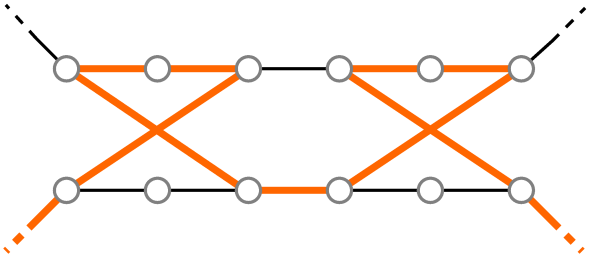

Figure 24: Un dernier parcours possible du gadget.

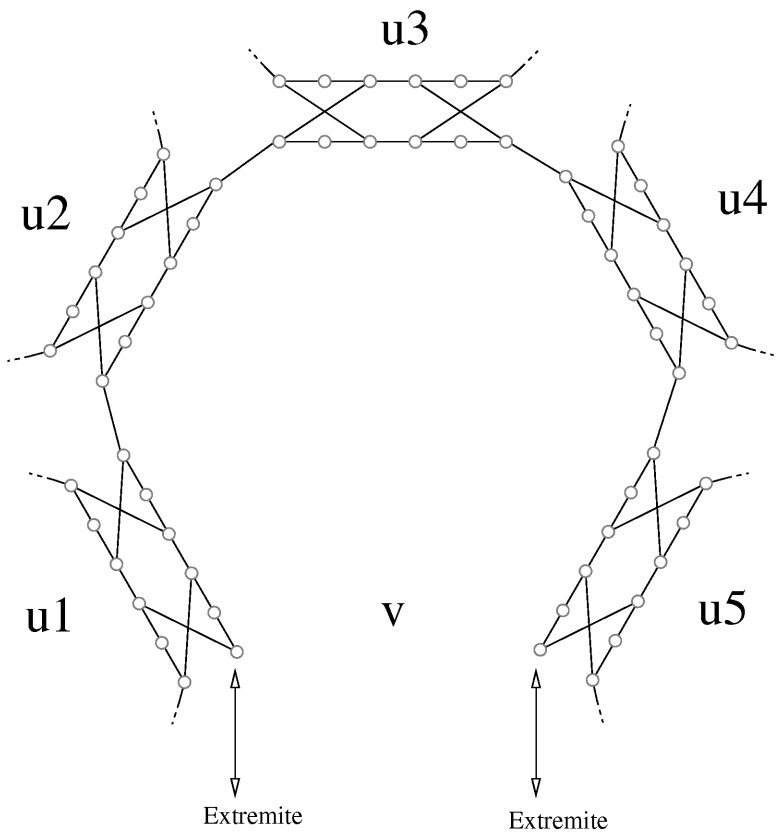

Figure 25: Construction pour un sommet $v \in V$. Les sommets adjacents de v sont ici $u1$, $u2$, $u3$, $u4$ et $u5$.

peut se ramener en temps polynomial à un problème de cycle hamiltonien. Pour comprendre cela, on étudie un certain gadget, dans ce cas un sous-graphe, donné sur la Fig. 21. On suppose que ce gadget fait partie du graphe à construire, et les seuls sommets ayant une arête avec le reste du graphe sont les quatre sommets des extrémités. Par une étude de cas, on comprend que les seuls chemins internes qui permettent une couverture par un chemin hamiltonien sont ceux donnés par les Fig 22, 23, et 24.

Le nouveau graphe $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, construit pour le problème du cycle hamiltonien, est construit comme suit. Le nombre de sommets $|\mathcal{V}|$ est égal à $k + 12|E|$. Pour chaque arête de E, 12 sommets de $\mathcal{G}$ sont introduits formant un gadget. Ensuite, pour un sommet $v \in E$, on relie tous les gadgets correspondant aux arêtes adjacentes à V en série comme indiqué sur la Fig. 25. On peut ainsi associer à chaque sommet v non-isolé de G deux extrémités: entrée du premier gadget en série et la sortie du dernier, comme indiqué sur la figure.

On crée ensuite k sommets $f_1, \ldots, f_k$ de $\mathcal{G}$ que l'on relie chacun à toutes les extrémités de $v \in V$.

Si un cycle hamiltonien existe dans ce graphe, il va nécessairement passer par chacun de ses k sommets. Pour passer d'un sommet f_i à un autre, noté f_j, il doit nécessairement choisir une extrémité d'un certain $v \in V$ et sortir par l'autre. Au total un ensemble S de k sommets de V sera choisi pour parcourir tous les $f_1, \ldots, f_k$. Pour que tous les gadgets soient parcourus, il faut nécessairement que les sommets de S couvrent toutes les arêtes de E. ∎

3 Un algorithme polynomial pour la programmation linéaire

Nous avons vu au cours précédent comment déterminer si un système d'inégalités linéaires a une solution mais nous avons négligé deux questions : peut-on déterminer si un polyèdre est vide en temps polynomial de la dimension de l'espace ? Comment trouver l'optimum d'une fonction objective sur un polyèdre, peut-on le faire en temps polynomial ? La question de la complexité des problèmes a été soulevée par J. Edmonds dans [3], article dans lequel il parle de problèmes pour lesquels il existe des algorithmes mieux que finis.
Dès 1910, C. de la Vallée Poussin,[2], s'intéressant au problème d'approximation de Chebyshev qui consiste à trouver le minimum de $\| Ax - b \|_\infty$, décrivit une méthode qui peut être considérée comme précurseur de l'algorithme du simplexe proposé en 1947 par G.B. Dantzig, [1]. Cet algorithme qui procède par changement de base, pivotage, n'est pas polynomial au moins pour les règles de pivotage connues jusqu'ores ; nous ne décrivons pas cet algorithme. Le lecteur peut se reporter à [9] ou à [10]. Jusqu'en 1979 aucun algorithme polynomial n'était connu pour résoudre un programme linéaire qui est un problème de la forme $Max\{c^t x | Ax \leq b, x \geq 0\}$. L. Khachiyan, s'inspirant des travaux de N.Z. Shor [11] et [12], proposa le premier algorithme, connu sous le nom de méthode des ellipsoïdes, prouvant que la programmation linéaire est un problème polynomial. C'est cet algorithme que nous allons exposer succintement ci-dessous, puis nous en verrons une conséquence importante

due à M. Grötschel, L. Lovász et A. Schrijver, [5].

3.1 La méthode de Khachiyan

Nous commençons cette étude en rappelant la définition d'ellipsoïde, en donnant quelques propriétés et en introduisant des notions sur la taille des sommets d'un polyèdre.

Définition 3.1 *Une matrice symétriqe M est dite définie positive si $x^tMx > 0\ \forall x \neq 0$*

Définition 3.2 *Une région $\mathcal{R} \in \mathbb{R}^n$ est un ellipsoïde s'il existe une matrice définie positive D et un vecteur $z \in \mathbb{R}^n$ tels que $\mathcal{R} = \{x \in \mathbb{R}^n |\ (x-z)^tD(x-z) \leq 1\}$, z est le centre de l'ellipsoïde .*

Cette définition montre qu'un ellipsoïde est une transformation affine de la boule unité $B = \{x \in \mathbb{R}^n |\ x^tx \leq 1\}$. Le théorème suivant précise de quelle manière on peut déterminer un ellipsoïde contenant un demi-ellipsoïde.

Théorème 3.1 *Soit E un ellipsoïde de paramètres (z, D) et soit $a \neq 0$ un vecteur de $\mathbb{R}^n$. Soit E' un ellipsoïde contenant $E \cap \{x |\ a^tx \leq a^tz\}$ et tel que le volume de E' soit le plus petit possible. Alors E' est unique, il a pour paramètres (z', D') avec :*

$$\begin{aligned} z' &= z - \frac{1}{n-1}\frac{Da}{\sqrt{a^tDa}} \\ D' &= \frac{n-1}{n^2-1}\left[\ D - \frac{2}{n-1}\frac{Daa^tD}{a^tDa}\ \right] \end{aligned}$$

de plus $\frac{\text{vol}(E')}{\text{vol}(E)} < e^{-\frac{1}{2n+2}}$.

Etant donné un polyèdre $P \subseteq \mathbb{R}^n$ on définit la facette-complexité de P, ψ, comme étant le plus petit entier $\psi \geq n$ tel qu'il existe un système d'inégalités linéaires à coefficients rationels $Ax \leq b$ décrivant P et dont chacune des lignes a une taille bornée par ψ. La taille d'une ligne est la longueur du plus grand numérateur ou dénominateur des coefficients qui la composent. On définit un paramètre semblable pour les sommets : la complexité sommets ν comme étant le plus petit entier $\nu \geq n$ tel qu'il existe des vecteurs à composantes rationelles $x_1, \ldots, x_k, y_1, \ldots, y_l$ chacun de taille au plus ν et tels que $P = conv(\{x_1, \ldots, x_k\}) + cône\{y_1, \ldots, y_l\}$. De plus on a $\nu \leq 4n^2\psi$ et $\psi \leq 4n^2\nu$.
La méthode des ellipsoïdes travaille en plusieurs temps : tout d'abord on s'assure qu'un polyèdre est non vide et si tel est le cas elle donne un point intérieur de ce polyèdre. Ensuite on calcule un point $x_<$ optimal à ϵ près. Enfin on transforme le point intérieur $x_<$ en un sommet x^* optimal. Pour la clarté de l'exposé nous supposons dans ce qui suit que le polyèdre P est borné et de pleine dimension et que nous pouvons effectuer tout calcul aussi précisément que requis. Dans la première étape qui consiste à déterminer un point intérieur du polyèdre la méthode va construire une suite d'ellipsoïdes de volume décroissant jusqu'à ce qu'un d'entre eux ait son centre dans le polyèdre. Pour initialiser la méthode le premier ellipsoïde est la boule de rayon 2^ν, $P \subseteq \{x |\ \parallel x \parallel \leq 2^\nu\}$. Donc le centre de

ce premier ellipsoïde est $z_0 = 0$ et la matrice associée est $D_0 = 2^{2\nu}Id$, Id désignant la matrice identité. Supposons maintenant que nous soyons rendus à l'étape i, alors $P \subseteq E_i$ qui est un ellipsoïde de paramètres (z_i, D_i). Si $z_i \in P$ alors nous avons trouvé un point de P, sinon supposons que z_i viole l'inégalité $a_{j_0}x \leq b_{j_0}$ appartenant au système qui défini P. Si E_{i+1} est le plus petit ellipsoïde contenant $E_i \cap \{x|\ a_{j_0}^t x \leq a_{j_0}^t z_i\}$. Alors étant donné que P est contenu tant dans E_i que dans le demi-espace $\{x|\ a_{j_0}^t x \leq a_{j_0}^t z_i\}$ on peut dire que $P \subseteq E_{i+1}$. Nous avons vu au théorème 3.1 comment déterminer (z_{i+1}, D_{i+1}) les paramètres de E_{i+1}. Nous avons vu que les ellipsoïdes E_i et E_{i+1} ont des volumes dont le rapport est inférieur à $e^{-\frac{1}{2n+2}}$, par ailleurs nous savons que le volume de l'ellipsoïde initial est inférieur à $4^{n\nu}$ aussi par récurrence nous pouvons dire que $vol(E_{i+1}) \leq 4^{n\nu}e^{-\frac{i+1}{2n+2}}$. Etant donné que nous avons supposé que P est borné et qu'il est de pleine dimension nous pouvons dire qu'il existe $n+1$ vecteurs affinement indépendant $x_0, \ldots, x_n$ tels que $P = conv(\{x_0, \ldots, x_n\})$ aussi $vol(P) \geq vol(conv(\{x_0, \ldots, x_n\})) \geq 2^{-2n\nu}$. Supposons que nous ayons exécuté un nombre I d'itérations de la méthode, on a alors :

$$2^{-2n\nu} \leq vol(P) \leq vol(E_I) < 4^{n\nu}e^{-\frac{I}{2n+2}}$$

Nous voyons bien que la proposition ci-dessus ne tient plus si on prend un nombre d'itérations suffisamment grand, $I \geq 16^{n^2\nu}$ par exemple. Aussi nous pouvons affirmer que nous aurons trouver un point de P avant d'atteindre l'itération I, celle-ci est totalement déterminée par un polynôme de la taille des descripteurs de P.
La méthode de Khachiyan nous permet donc de trouver en un temps polynomial un point d'un polyèdre. Elle permet aussi de déterminer de manière approchée l'optimum d'une fonction objective c. Considérons x_0 un point appartenant à l'intérieur du polyèdre P et deux sphères centrées en x_0, une de rayon r contenue dans P et l'autre de rayon R contenant P. L'algorithme démarre donc avec une matrice définie positive $D_0 = R^2Id$. Considérons à l'itération j le point courant x_j. On note $\delta_j = c^t x_j$ la valeur de la fonction objective au point x_j. Si x_j appartient à l'intérieur de P on calcule alors un ellipsoïde E_{j+1} de paramètre x_{j+1}, D_{j+1} contenant $P \cap \{x|\ -c^t x < -\delta_j\}$ comme vu précédemment. Si x_j n'appartient pas à l'intérieur de P alors il existe une inégalité $a_{i_j}^t x \leq b_{i_j}$ parmi celles définissant P telle que $a_{i_j}^t x_j \geq b_{i_j}$ on calcule un ellipsoïde E_{j+1} contenant $E_j \cap \{a_{i_j}^t x \leq a_{i_j}^t x_j\}$ comme vu lors de la recherche de points admissibles. Grâce à un argument sur le rapport de volume des ellipsoïdes on peut montrer qu'une solution approchée à ϵ prés est obtenue après un nombre d'itérations égal à $2n(n+1)\lceil log\frac{2R^2\|c\|}{r\epsilon}\rceil$.
A ce stade nous possédons un algorithme polynomial pour résoudre de manière approchée un programme linéaire, nous allons voir que via une transformation polynomiale du programme linéaire, l'algorithme en fournit une solution optimale. Pour cela modifions la fonction objective c du programme linéaire comme suit. On pose $\xi = 2\nu^{2n}$ et on considère la fonction objective $c_0 = c\xi + (1, \xi, \xi^2, \ldots, \xi^{n-1})^t$. Sans perte de généralité on peut supposer que c est un vecteur dont les coordonnées sont des entiers.

Proposition 3.1 *Soit $P = \{x \in \mathbb{R}^n | Ax \leq b\}$ un polyèdre rationnel. Le programme linéaire $Max\{c_0^t x,\ x \in P\}$ a un optimum unique x^* qui est une solution optimale du programme linéaire $Max\{c^t x,\ x \in P\}$.*

Preuve :
Considérons $\bar{x}$ un sommet de P distinct de x^*, ces deux vecteurs étant supposés à coordonnées rationnelles on peut poser $\bar{x} = (\frac{\bar{p}_1}{\bar{q}_1}, \ldots, \frac{\bar{p}_n}{\bar{q}_n})^t$ et $x^* = (\frac{p_1^*}{q_1^*}, \ldots, \frac{p_n^*}{q_n^*})^t$. La différence de ces deux vecteurs est alors donnée par le terme générique :

$$x_i^* - \bar{x}_i = (p_i^* \bar{q}_i - q_i^* \bar{p}_i) \left(\prod_{j \neq i} q_j^* \bar{q}_j \right) \left(\prod_{j=1}^{n} q_j^* \bar{q}_j \right)^{-1}$$

Ceci permet de dire qu'il existe $\alpha < \frac{1}{2}\xi$ et un vecteur u entier dont chaque composante d'indice i vérifie $|u_i| < \xi$ tel que $x^* - \bar{x} = \frac{u}{\alpha}$. Par hypothèse x^* est l'optimum unique du programme $Max\{c_0^t x,\ x \in P\}$ aussi pouvons-nous écrire :

$$c_0^t(x^* - \bar{x}) = c_0^t \frac{u}{\alpha} = \frac{1}{\alpha}(c^t \xi^n u + \sum_{i=1}^{n} \xi^{i-1} u_i) \geq 0$$

Etant donné que $|U_i| < \xi$ on a : $\left| \sum_{i=1}^{n} \xi^{i-1} u_i \right| \leq \left| (\xi - 1) \sum_{i=1}^{n} \xi^{i-1} \right| = \left| \xi^n - 1 \right|$. Comme $\xi^n c^t u \geq \sum_{i=1}^{n} \xi^{i-1} u_i$ et que $c^t u$ est un entier relatif il vient que $c^t u \geq 0$. Aussi a-t-on établi que quelque soit le sommet $\bar{x}$ on a $c^t x^* \geq c^t \bar{x}$, ce qui montre que x^* est une solution optimale du programme $Max\{c^t x,\ x \in P\}$. Par ailleurs on peut remarquer que $c_0^t(x^* - \bar{x}) \neq 0$, ce qui montre que x^* est l'optimum unique du programme $Max\{c_0^t x,\ x \in P\}$.

∎

On peut donc dire que nous disposons d'un algorithme polynomial pour résoudre un programme linéaire. Remarquons que dans l'étape de recherche d'un point de l'espace appartenant au polyèdre P nous avons supposé que celui-ci était donné explicitement par le système d'inégalités linéaires qui le définit. L'algorithme que nous venons de voir est polynomial mais le nombre de contraintes m intervient dans ce polynôme. Que se passe-t-il si le nombre de contraintes est grand, de taille exponentielle par rapport au nombre de variables, ou si le polyèdre nous est donné de manière implicite ?

4 Le problème de séparation

Considérons un polyèdre P défini par un ensemble d'inégalités linéaires dont la taille maximale est ψ. De plus supposons que nous ne connaissions pas explicitement cet ensemble d'inégalités, mais que nous soyons capables de résoudre, en un temps polynomial de n et de ψ, le problème suivant :
Séparation : étant donné un vecteur à composantes rationnelles y, appartient-il à P ? Sinon retourner un hyperplan $H = \{x|\ c^t x = b\}$ tel que $c^t y < b$ et $c^t u \geq b\ \forall u \in P$.
Remarquons que cette définition du problème de séparation nous renvoie au théorème de Farkas, Minkowsky et Weyl. On peut remarquer que si le polyèdre P est de pleine dimension la méthode des ellipsoïdes permet de trouver un vecteur de P en un temps polynomial

de n et ψ, pour peu qu'on puisse traiter le problème de séparation sur le polyèdre considéré en temps polynomial. Ce fait fut remarqué par M. Grötschel, L. Lovász et A. Schrijver en 1981, [5], ainsi que par R. Karp et C. Papadimitriou, [6]. En effet la résolution en temps polynomial du problème de séparation nous permet de tester en temps polynomial à chacune des étapes si le centre de l'ellipsoïde appartient au polyèdre P et si ce n'est pas le cas de déterminer une contrainte qui est violée.
Le théorème suivant, dont le lecteur pourra trouver la preuve dans [10], précise ce qui vient d'être dit sur la relation existant entre réalisabilité et séparation.

Théorème 4.1 *Il existe un algorithme $\mathcal{R}$ permettant de tester la réalisabilité d'un système d'inégalités linéaires, de taille maximum ψ, définissant un polyèdre $P \in \mathbb{R}^n$ et un polynôme $\chi(\psi, n)$ de sorte que si les entrées de l'algorithme de réalisabilité sont ψ, n et un algorithme de séparation sur P, $\mathcal{S}$, alors $\mathcal{R}$ calcule un vecteur $x \in P$ ou conclut que ce polyèdre est vide en un temps borné par $\alpha\chi(\psi, n)$, α étant le temps maximum nécessaire à $\mathcal{S}$ pour s'exécuter sur des entrées de taille maximum $\chi(\psi, n)$.*

Ce théorème a des conséquences très importantes, notamment en ce qui concerne l'optimisation. Nous allons voir maintenant comment cela conduit à l'équivalence des problèmes de séparation et d'optimisation. On définit le problème d'optimisation sur un polyèdre rationnel P par :
Optimisation : étant donné un vecteur à composantes rationnelles c, fournir une des réponses suivantes :

1. $x_0 = Argmin\{c^t x |\ x \in P\}$
2. $y_0 \in recône(P)$ tel que $c^t y > 0$, $recône(P) = \{y \in \mathbb{R}^n |\ x + y \in P,\ \forall x \in P\}$ est appelé cône de récession ou cône caractéristique de P. Ceci correspond au cas où l'optimum sur P de la fonction objective c n'est pas borné.
3. montrer que P est vide.

Corollaire 4.1 *Il existe un algorithme $\mathcal{O}$ tel que si les entrées de $\mathcal{O}$ sont $(n, \psi, \mathcal{S})$; n et ψ étant des entiers naturels, $\mathcal{S}$ étant un algorithme de séparation sur le polyèdre rationnel P défini par des inégalités linéaires de taille bornée par ψ, alors $\mathcal{O}$ résoud pour la fonction objective c à composantes rationnelles le problème d'optimisation sur P en un temps borné par un polynôme de n, ψ, la taille de c et le temps d'exécution de l'algorithme $\mathcal{S}$.*

Nous ne donnons ici que des éléments de preuve pour ce corollaire ainsi que pour le suivant concernant le cas où l'optimum de la fonction objective c sur le polyèdre P est fini, nous invitons le lecteur à se reporter à l'article originel [5], à [10] page 178, à [8] pages 201 à 208, ou bien encore à [9] pages 162 à 163 pour une preuve exhaustive.
Squelette de preuve :
Supposons que les paramètres $(n, \psi, \mathcal{S})$ et c satisfont les hypothèses du corollaire et que ξ soit la taille du vecteur c. Etant donné que l'optimum,δ, du programme $Max\{c^t x |\ x \in P\}$ est borné on peut affirmer qu'il est atteint par un vecteur de taille au plus $4n^2\psi$. Aussi pouvons-nous dire que la taille de δ est inférieure ou égale à $2\xi + 8n^2\psi$. Posons $\beta = 3\xi + 12n^2\psi$

et prenons $m_0 = -2^\beta$ et $M_0 = 2^\beta$. Procédons alors par dichotomie en testant grâce à l'algorithme $\mathcal{R}$ si le polyèdre $P \cap \{x|\ c^t x \geq \frac{m_i+M_i}{2}\}$ est vide ou non. S'il est vide on met à jour le majorant : $M_{i+1} = \frac{m_i+M_i}{2}$, sinon on met à jour le minorant : $m_{i+1} = \frac{m_i+M_i}{2}$. Le nombre d'étapes nécessaires à ce parcours dichotomique de l'intervalle $[m_0, M_0]$ nous assure alors d'avoir obtenu l'optimum du programme $Max\{c^t x|\ x \in P\}$ en un temps borné par un polynôme de n, ψ et ξ.

∎

Nous allons voir maintenant la forme polaire du corollaire 4.1 qui montre que le problème de séparation se réduit polynomialement au problème d'optimisation.

Corollaire 4.2 *Il existe un algorithme $\mathcal{O}^*$ tel que si les entrées de $\mathcal{O}^*$ sont $(n, \psi, \mathcal{O})$; n et ψ étant des entiers naturels, $\mathcal{O}$ étant un algorithme d'optimisation sur le polyèdre rationnel P défini par des inégalités linéaires de taille bornée par ψ, alors $\mathcal{O}^*$ résoud pour tout vecteur y à composantes rationnelles le problème d'optimisation sur P en un temps borné par un polynôme de n, ψ, la taille de y et le temps d'exécution de l'algorithme $\mathcal{O}$.*

De même que pour le corollaire précédent nous ne donnons qu'un canevas de la preuve que le lecteur pourra trouver dans les références précédemment indiquées.
Squelette de preuve :
Considérons P^* le polaire de $P = \{x \in \mathbb{R}^n|\ Ax \geq 0\}$, alors par définition $P^* = \{y \in \mathbb{R}^n|\ y^t A \geq 0\}$; ainsi peut-on dire que si P est décrit par un ensemble d'inégalités de taille maximum ψ il en va de même pour P^*. Par ailleurs nous savons que $x_0 \in P$ si et seulement si $Min\{x_0^t y|\ y \in P^*\} \geq 0$ et réciproquement $y_0 \in P^*$ si et seulement si $Min\{y_0^t x|x \in P\} \geq 0$. Ainsi peut-on voir que si on sait optimiser en temps polynomial sur le polyèdre P alors on sait séparer en temps polynomial sur son polaire P^* et réciproquement. Par hypothèse on possède un algorithme d'optimisation sur P donc de séparation sur P^*. Alors par le corollaire 4.1 on peut dire qu'il existe un algorithme s'exécutant en un temps polynomial de n, ψ et le temps d'exécution de l'algorithme $\mathcal{O}$ permettant de résoudre le problème d'optimisation sur P^* ; l'existence de l'algorithme $\mathcal{O}^*$ suit alors.

∎

Les corollaires 4.1 et 4.2 nous permettent d'énoncer le résultat suivant qui établit l'équivalence entre les problèmes de séparation et d'optimisation.

Corollaire 4.3 *(**Grötschel, Lovász, Schrijver**) Pour tout polyèdre rationnel P le problème de séparation peut être résolu en temps polynomial si et seulement si le problème d'optimisation peut être résolu en temps polynomial.*

Corollaire 4.4 *Si P est un polyèdre rationnel et P^* son polaire, alors les quatre propositions suivantes sont équivalentes :*

1. *Il existe un algorithme polynomial pour résoudre le problème d'optimisation sur P;*
2. *il existe un algorithme polynomial pour résoudre le problème d'optimisation sur P^*;*
3. *il existe un algorithme polynomial pour résoudre le problème de séparation sur P;*
4. *il existe un algorithme polynomial pour résoudre le problème de séparation sur P^*.*

5 Approximation des problèmes

Nous donnons dans cette section quelques idées marquantes du concept d'approximation des problèmes.

5.1 Classification des algorithmes d'approximation

Le fait qu'un problème est NP-complet ne signifie pas que les seules solutions à ce problèmes sont purement heuristiques. D'ordinaire on classe les réponses possibles en algorithmique par trois grandes dénominations:

APX c'est une *classe* qui contient les problèmes approximables à un facteur multiplicatif constant près en temps polynomial. Autrement dit, il exite un programme qui calcule en temps polynomial une solution qui est au plus à un facteur K de la solution mathématiquement optimale.

PTAS Le terme PTAS (pour *polynomial-time approximation scheme*) désigne une famille de programmes d'approximation qui obtiennent une solution à un facteur multiplicatif constant près aussi proche que l'on veut de 1. En d'autres termes, pour tout $\varepsilon > 0$, il existe un programme qui calcule en temps polynomial une approximation à $1 + \varepsilon$ près de la valeur optimale.

FPTAS Le terme PTAS (pour *fully polynomial-time approximation scheme*) désigne un programme PTAS dont la complexité est non seulement polynomiale en la taille du problème, mais aussi en $1/\varepsilon$.

Par extension, le terme "APX-complet" est utilisé pour parler de la classe des problèmes APX pour lesquels il existe un PTAS qui les réduit à n'importe quel autre problème d'APX. Par "APX-difficile" on entend qu'une instance du problème en question est "APX-complète".

Pour les problèmes mentionnés dans le document, on sait[1] faire la classification suivante:

Problème	Approximation sur	Statut
Tri		P
Programmation linéaire		P
Chemin Eulérien		P
SAT	Nombre de clauses maximum satisfaites	APX-complet
3-SAT	Nombre de clauses maximum satisfaites	APX-complet
CYCLE HAMILTONIEN	Trouver le plus long chemin	APX-complet
VERTEX COVER	Nombre de sommets nécessaires	APX mais pas de PTAS

5.2 La programmation dynamique

La programmation dynamique est une méthode tabulaire où l'on cherche à obtenir les meilleurs résultats possibles par un tableau T. Le problème classique pour la programmation dynamique est le sac-à-dos. Soit un sac dans lequel on peut mettre n objets d'encombrement

[1] http://www.csc.kth.se/ viggo/problemlist/

respectifs $c_1, \ldots, c_n$ et de valeur $v_1, \ldots, v_n$. Etant donnée une capacité de sac K, on cherche l'ensemble $S \subseteq \{1, \ldots, n\}$ tel que

$$\sum_{j \in S} c_j \leq K,$$

et qui maximise $\sum_{j \in S} v_j$.

On initialise donc le tableau T à 0 pour toutes des valeurs de 1 à K; puis pour chaque indice i de 1 à n on effectue les opérations suivantes:

Pour j décrémentant de $K - c_i$ à 0, si $T[j + c_i] < T[j] + v_i$ alors $T[j + c_i] := T[j] + v_i$.

Cette méthode utilise Kn pas de calcul et obtient, pour chaque $k \in \{1, \ldots, K\}$, la meilleure valeur possible obtenue avec un sac de capacité k.

Cette méthode est assez déconcertante car le problème du sac-à-dos est NP-complet. L'explication de ce paradoxe vient du fait que les cas difficiles du problème du sac-à-dos sont ceux où les valeurs c_i et v_i sont exponentielles en fonction de n, et donc K est une exponentielle de n. Cette méthode montre que la précision avec laquelle on veut répondre à une question est un point déterminant pour l'évaluation d'un algorithme. Le paragraphe suivant tient compte de cette perspective pour améliorer la complexité d'un problème de la classe P.

5.3 Approximation de multi-flot

Pour plus d'informations sur ce thème nous renvoyons le lecteur à [7, chapitre 19]. Le problème du flot multi-commodités peut s'écrire comme suit:

Problème 5.1 *Etant donné $G = (V, E)$ un digraphe, une fonction de capacité $u : E \to \mathbb{N}$, et un ensemble de requêtes $H \subseteq V \times V$, on note $n = |V|$, $m = |E|$ et $k = |H|$.*

Trouver $(x^f)_{f \in H}$ où

- *x_f est un $s - t$ flot pour chaque $f = (s, t) \in H$,*
- *$\forall e \in E \qquad \sum_{f \in H} x^f(e) \leq u(e)$,*
- *la valeur $\sum_{f \in H} value(x^f)$, notée $OPT(G, H, u)$, est maximum.*

Traduit en programme linéaire, ce problème admet une formulation très élégante. Soit $\mathcal{P}$ l'ensemble des $s - t$ chemins, pour $(s, t) \in H$. On cherche

$$OPT(G, H, u) = \max \left\{ \sum_{P \in \mathcal{P}} y(P) \ : \ y \geq 0 \text{ et } \forall e \in E \quad \sum_{P \in \mathcal{P},\ e \in P} y(P) \leq u(e) \right\}. \qquad (1)$$

Ce programme mathématique admet un dual qui s'écrit de la manière suivante:

$$OPT(G,H,u) = \min\left\{ z\cdot u \ : \ z \geq 0 \text{ et } \forall P \in \mathcal{P} \quad \sum_{e\in E} z(e) \geq 1 \right\}. \tag{2}$$

On se donne un facteur d'approximation $\varepsilon > 0$ petit. On propose pour ce problème l'algorithme $\mathcal{A}$ suivant:

(1) $\forall P \in \mathcal{P} \quad y(P) := 0$
$\delta := (n(1+\varepsilon))^{-\lceil \frac{5}{\varepsilon} \rceil}(1+\varepsilon)$
$\forall e \in E \quad z(e) := \delta$

(2) Soit P tel que $z(P) = \sum_{e\in P} z(e)$ soit minimum. Si $z(P) \geq 1$, aller en (4).

(3) $\gamma := \min_{e\in P} u(e)$
$y(P) := y(P) + \gamma$

$\forall e \in P \quad z(e) := z(e)(1 + \frac{\varepsilon\gamma}{u(e)})$

(4) $\xi := \max_{e\in E} \frac{1}{u(e)} \sum_{P\in\mathcal{P} \ : \ e\in P} y(P)$
$\forall P \in \mathcal{P} \quad y(P) := \frac{y(P)}{\xi}$

Nous avons alors le résultat suivant:

Théorème 5.1 *Soit $\varepsilon < \frac{1}{2}$, l'algorithme $\mathcal{A}$ donne $\frac{1}{1+\varepsilon}OPT(G,H,u)$ en $O(\frac{1}{\varepsilon^2}km(m+n\log(n))\log(n))$.*

Preuve. Tout d'abord remarquons que la complexité de l'algorithme est bien celle annoncée. La condition d'arrêt (en (2)) fait qu'un chemin ne peut pas emprunter un arc de poids supérieur à 1 sauf dans la dernière étape de l'algorithme. Donc pour tout arc e, on a $z(e) \leq 1+\varepsilon$. Soit t le nombre d'itérations, c'est-à-dire le nombre d'executions de l'étape (3). Une itération demande k calculs de plus court chemin (Dijkstra), soit $O(k(m+n\log(n))$ étapes de calcul. Au cours de chaque itération, au moins un arc voit augmenter son poids d'un facteur $1+\varepsilon$. Comme la valeur de l'arc croît de δ à $1+\varepsilon$, cela limite le nombre de telles augmentations possibles pour un arc à $\lceil \log_{1+\varepsilon}(1/\delta)\rceil + 1$. Etant donné qu'il n'y a que m arcs, on a nécessairement $t \leq m(\lceil \log_{1+\varepsilon}(1/\delta)\rceil + 1)$, d'où la complexité de l'algorithme.

Ensuite, notons que le calcul en (4) garantit bien la faisabilité de la solution en termes de respect des capacités.

Désignons dans ce qui suit $z^{(i)}$ la valeur du vecteur z après la $i^{\text{ème}}$ itération, ainsi que P_i et γ_i les valeurs respectives du chemin P et du scalaire γ choisis au cours de l'itération i. $z^{(0)}$ désigne par prolongement naturel le vecteur de $\mathbb{R}^E$ uniformément égal à δ.

Par ailleurs, soit e l'arc qui détermine la valeur de ξ, et désignons par $i_1, \ldots, i_p$ les indices des itérations où cet arc est augmenté. Nous avons alors $\xi = \sum_{P\ni e} y(P) = \gamma_{i_1} + \cdots + \gamma_{i_p}$. Les

multiplications sur la valeur de $z(e)$ de cet arc sont de $1+\frac{\varepsilon\gamma_{i_j}}{u(e)}$ et l'inégalité $1+\varepsilon x \geq (1+\varepsilon)^x$ pour $0 \leq x \leq 1$ permet d'écrire:

$$1+\varepsilon \geq z(e) \geq \delta(1+\varepsilon)^{\gamma_{i_1}+\cdots+\gamma_{i_p}}.$$

En somme, nous obtenons

$$\xi \leq \log_{1+\varepsilon}\left(\frac{1+\varepsilon}{\delta}\right). \tag{3}$$

Regardons maintenant comment le vecteur z évolue au cours des itérations. En observant les calculs effectués, on peut facilement établir la relation suivante:

$$z^{(i)} \cdot u = z^{(i-1)} \cdot u + \varepsilon\gamma_i \sum_{e\in P_i} z^{(i-1)}(e).$$

Si on note maintenant $\alpha(x)$ la fonction qui au vecteur positif $x \in \mathbb{R}_+^E$ associe $\min_{P\in\mathcal{P}} \sum_{e\in P} x(e)$, alors:

$$(z^{(i)} - z^{(0)}) \cdot u = \varepsilon \sum_{j=1}^{j=i} \gamma_i \alpha(z^{(j-1)}). \tag{4}$$

Remarquons maintenant d'après (2) que

$$OPT(G,H,u) = \min\left\{\frac{x\cdot u}{\alpha(x)} \ : \ x \in \mathbb{R}_+^E\right\}.$$

En combinant cette équation avec (4) on obtient:

$$(\alpha(z^{(i)}) - \delta n)OPT(G,H,u) \leq OPT(G,H,u)\alpha(z^{(i)} - z^{(0)}) \leq (z^{(i)} - z^{(0)}) \cdot u.$$

Il vient

$$\alpha(z^{(i)}) \leq \delta n + \frac{\varepsilon}{n} \sum_{j=1}^{j=i} \gamma_j \alpha(z^{(j-1)}). \tag{5}$$

Montrons maintenant par récurrence que

$$\delta n + \frac{\varepsilon}{n} \sum_{j=1}^{j=i} \gamma_j \alpha(z^{(j-1)}) \leq \delta n e^{\frac{\varepsilon}{OPT(G,H,u)} \sum_{j=1}^{j=i} \gamma_j}. \tag{6}$$

où e désigne l'exponentielle de 1. Le résultat est évident pour $i = 0$. Considérons maintenant $i \geq 1$. Nous avons

$\delta n + \frac{\varepsilon}{n} \sum_{j=1}^{j=i} \gamma_j \alpha(z^{(j-1)})$	
$= \delta n + \frac{\varepsilon}{n} \sum_{j=1}^{j=i-1} \gamma_j \alpha(z^{(j-1)}) + \frac{\varepsilon}{n}\gamma_i \alpha(z^{(i-1)})$	en séparant le dernier terme,
$\leq \delta n + \frac{\varepsilon}{n} \sum_{j=1}^{j=i-1} \gamma_j \alpha(z^{(j-1)}) + \frac{\varepsilon}{n}\gamma_i \left(\delta n + \frac{\varepsilon}{n} \sum_{j=1}^{j=i-1} \gamma_j \alpha(z^{(j-1)})\right)$	par (5),
$\leq (1 + \frac{\varepsilon}{n}\gamma_i) \left(\delta n + \frac{\varepsilon}{n} \sum_{j=1}^{j=i-1} \gamma_j \alpha(z^{(j-1)})\right)$	par factorisation,
$\leq (1 + \frac{\varepsilon}{n}\gamma_i) \delta n e^{\frac{\varepsilon}{OPT(G,H,u)} \sum_{j=1}^{j=i-1} \gamma_j}$	par récurrence,
$\leq \delta n e^{\frac{\varepsilon}{OPT(G,H,u)} \sum_{j=1}^{j=i} \gamma_j}$	en utilisant $1+\varepsilon x \leq (1+\varepsilon)^x$ pour $x \in [0,1]$.

Le critère d'arrêt de l'algorithme combiné à (5) et (6) implique que

$$1 \leq \alpha(z^{(t)}) \leq \delta n e^{\frac{\varepsilon}{OPT(G,H,u)} \sum_{j=1}^{j=t} \gamma_j},$$

donc $\sum_{i=1}^{i=t} \gamma_i \geq \frac{OPT(G,H,u)}{\varepsilon} \ln(1/(\delta n))$.

Finalement, l'algorithme retourne une solution de valeur $\frac{1}{\xi}(\sum_{i=1}^{i=t} \gamma_i)$ et nous avons, en utilisant (3),

$$\frac{1}{\xi} \sum_{i=1}^{i=t} \gamma_i \geq \frac{OPT(G,H,u) \ln(1/(\delta n))}{\varepsilon \log_{1+\varepsilon}\left(\frac{1+\varepsilon}{\delta}\right)} \geq \frac{OPT(G,H,u)}{1+\varepsilon}.$$

∎

References

[1] G.B. Dantzig. Maximization of a linear function of variables subject to linear inequalities. *Proceedings of Linear Programming Conference, June 20-24, 1949*, pages 359–373, 1951.

[2] C de la Vallée Poussin. Sur la méthode de l'approximation minimum. *Annales de la société scientifique de Bruxelles*, 1910.

[3] J. Edmonds. Paths,Trees, and Flowers. *Canadian Journal of Mathematics*, 17:449–467, 1965.

[4] M.R. Garey and D.S. Johnson. *Computers and Intractability.* W.H. Freeman and Company, 1979.

[5] M. Grötschel, L. Lovász, and A. Schrijver. The ellipsoid method and its consequences in combinatorial optimization. *Combinatorica*, 1:169–197, 1981.

[6] R.M. Karp and C.H. Papadimitriou. On linear characterizations of combinatorial optimization problems. *SIAM Journal on Computing*, 11:620–632, 1982.

[7] B. Korte and J. Vygen. *Combinatorial Optimization.* Springer, 2000.

[8] J.F Maurras. *Complexité et LP-Réduction.* Manuscript, 1997.

[9] G.L. Nemhauser and L.A. Wolsey. *Integer and Combinatorial Optimization.* Wiley Interscience, Series in Discrete Mathematics and Optimization, 1988.

[10] A. Schrijver. *Theory of linear and integer programming.* Wiley Interscience, Series in Discrete Mathematics and Optimization, 1986.

[11] N.Z. Shor. Utilization of the operation of space dilation in the minimization of convex functions. *Cybernetics*, vol 6:7–15, 1970.

[12] N.Z. Shor. Cut-off method with space extension in convex programming problems. *Cybernetics*, vol 13:94–96, 1977.