Introduction à l'algorithmique

Claude Jard

2009

Ce document contient les notes du cours commun d'algorithmique donné à l'ENS Cachan en première année des magistères de Mathématiques, et d'Informatique et Télécommunication.

Contents

Chapter 1

Introduction

1.1 L'informatique est-elle une science ?

1.1.1 Le monde devient numérique

Le monde devient numérique et on l'observe tous les jours à travers de grands bouleversements. Les plus visibles sont probablement dans les domaines de la communication (Internet, téléphone) et de l'audio-visuel (MP3, photo/vidéo, télé). Mais il y aussi le commerce en ligne, la cartographie interactive, les transports (GPS, pilotage, sécurité), l'industrie (gestion, CAO, travail à distance), les sciences (modélisation et expérimentation numérique), la médecine (imagerie, chirurgie), etc.

L'informatique n'est plus la science (ou la technique) des ordinateurs, elle devient la science de l'information numérique. C'est aussi une grande industrie qui irrigue toutes les autres. Certains pensent que nous vivons aujourd'hui une révolution du même ordre d'importance en terme d'impact sur la société toute entière que celle de l'imprimerie.

Elle est surtout un formidable espace d'innovation. Bien que toutes ces innovations techniques peuvent apparaître comme disparates, on découvre petit à petit qu'elles reposent sur un socle commun de concepts, de modèles, de méthodes qui forment aujourd'hui la science informatique en pleine construction. L'informatique a créé ses objets et ses méthodes. Elle n'est pas une science de la nature comme la physique, l'astronomie ou l'archéologie. Mais désormais elle a trouvé des sujets d'observation et des modèles avec le langage, la génétique, la cognition, la communication..., en plus de ses propres créations.

1.1.2 Les grands jalons

Les pionniers (1930-1945) sont des mathématiciens (von Neumann, Turing, Eckert, Gödel). Il s'agit de réfléchir aux modèles de calcul automatique et imaginer les premières architectures électroniques. Suit la période des grands centres de calcul (1955-1975), grands équipements pour le calcul scientifique,

mais aussi la gestion. Puis on assiste à la déconcentration avec l'invention des mini-ordinateurs (1975-1980) qui peuvent équiper les établissements et les laboratoires. L'accent est mis sur le développement des systèmes d'exploitation, la programmation et les premières mises en réseaux dans le monde des télécommunications. La déconcentration se poursuit avec l'apparition des ordinateurs personnels (le "PC") (1980-1995). La miniaturisation permet aussi de spécialiser des calculateurs pour "embarquer" l'informatique dans les systèmes. On assiste à l'explosion de l'industrie informatique et des applications. Parallèlement, l'industrie des télécommunications se développe et la convergence informatique-télécom (1995-2005) produit de très grands réseaux d'ordinateurs. C'est l'explosion de l'Internet et du téléphone portable. On assiste actuellement (2005-2015) à une ouverture considérable de l'informatique vers de grands domaines d'application: sciences, audio-visuel, médecine, transports, ... Le sans-fil est une aide considérable. Et ensuite ? On prépare un monde informatique ubiquitaire où les objets informatiques sont dans le réseau, possèdent une certaine autonomie, communiquent et interagissent avec les humains.

1.1.3 Des défis

L'informatisation de notre société (à laquelle nous participons, ne serait-ce qu'en tant que consommateur) n'est pas sans danger. Les questions de sécurité, d'éthique, de droit, prennent de plus en plus d'importance. Il y aussi la question fondamentale des "bugs". La facilité d'écriture de logiciels, d'applications informatiques, est trompeuse. Il est difficile d'exprimer précisément ce que l'on veut faire (la "spécification"), difficile de prouver qu'un programme se comporte conformément à sa spécification, difficile de le faire évoluer dans la durée. Les décideurs n'en ont pas forcément conscience et la liste des pannes logicielles, voire des catastrophes dans les applications critiques s'allonge. La conception de logiciels sàrs reste un vrai défi pour la recherche en informatique.

Un autre aspect dont il faut bien prendre conscience est l'indépendance aujourd'hui entre l'information et son support. Le monde devient numérique. Cela veut dire que l'information peut être manipulée, transformée automatiquement, transportée, copiée à l'infini.

1.2 La machine à information

On peut la voir à trois niveaux : le niveau matériel (circuits, disques, réseaux, capteurs), le niveau logiciel (système, applications, "middleware") et l'interface personne-machine (souvent un point faible). La logique et l'intrication de ces différents niveaux est généralement invisible. Par exemple, une application donnée à été écrite dans un langage de haut-niveau spécialisé, puis compilée dans un langage de plus bas niveau pour enfin aboutir à une suite de bits interprétée par des circuits électroniques spécialisés. La distance entre l'expression du problème et de l'algorithme associé et ce qu'exécute en final la machine peut être énorme. L'expansion en terme du nombre d'instructions à effectuer est très

grande, mais les processeurs modernes misent sur la rapidité d'exécution.

En ce qui concerne le matériel, il faut être conscient de la "loi de Moore". Il s'agit d'une observation empirique sur le fait que la taille des transistors (électronique élémentaire de commutation) diminue exponentiellement au cours des années. Cela est vrai pour leur coàt et augmente exponentiellement la quantité intégrable sur une puce de silicium. En 1970, un processeur contenait quelques milliers de transistor avec un transistor à 1 dollar. Aujourd'hui, un dual core Itanium Intel contient plus d'un milliard de transistors pour un coàt unitaire d'un cent millionnième de cents. Produire de tels circuits est le fait d'une industrie électronique lourde (et l'erreur est lourde de conséquence). Concevoir un circuit moderne est aussi complexe que concevoir un avion. Un des freins possibles pour la poursuite de la loi de Moore est la miniaturisation et les problèmes de puissance dissipée. On s'interroge sur la technologie "post-Moore".

La situation du logiciel est différente. Il s'agit d'un très long texte technique (plusieurs millions de lignes de code dans un téléphone portable). On construit de gros textes par assemblage et réutilisation de textes plus élémentaires, mais on ne maîtrise pas complètement l'augmentation de la complexité (le coàt du test et de la maintenant est déjà bien supérieur au coàt de la conception). On n'observe pas de loi de Moore... Il existe toute une faune de langages de programmation. On se permet même de concevoir des langages spécialisés avec leur compilateur associée pour écrire des applications spécifiques. La hantise est le bug. Force est pourtant de constater qu'en ce qui concerne les langages généraux de programmation, on privilégie souvent le plus souple au plus sàr.

Les bugs commencent à coàter cher (téléphone américain, Ariane 5, Pentium, ...). La recherche académique investit dans des méthodes formelles (mathématiques) permettant de détecter les bugs avant exécution ou déploiement. L'industrie, sauf dans des secteurs particuliers comme la conception de circuits, reste sceptique et privilégie souvent le "quick and dirty". A quand la notion de garantie pour le logiciel ?

1.3 L'algorithmique

Selon l'encyclopédie, un algorithme est "la spécification d'un schéma de calcul, sous forme d'une suite finie d'opérations élémentaires obéissant à un enchainement déterminé", ou encore "une méthode ou procédé permettant de résoudre un problème à l'aide d'un calculateur – c'est à dire, une séquence finie d'instructions exécutables dans un temps fini et avec une mémoire finie".

Il existe une notion formelle d'algorithme, liée aux fonctions calculables en arithmétique et à la théorie de la récursivité. De grands noms jalonnent cet aspect de la discipline informatique.

- Edsger Dijkstra : (ex. plus court chemin)
- Michael Rabin : (ex. automates probabilistes, infinis, ...)

- Donald Knuth : (ex. analyseur LR(k), Tex)
- Leslie Lamport : (ex. horloges logiques, Latex)
- Robert Tarjan : (ex. DFS, graphes)

Il est assez extraordinaire de penser que la plupart nous sont contemporains et que l'on a pu les cotoyer...

1.3.1 Les critères algorithmiques

Pour concevoir un algorithme, il faut d'abord se placer dans ce cadre d'un modèle de machine. Il peut s'agir du modèle séquentiel classique (la machine de von Neumann), ou d'un modèle parallèle, d'un modèle réparti sur un réseau... Ensuite, il faut avoir à l'esprit plusieurs objectifs. Le premier est sans nul doute celui de la correction : il faut prouver que l'algorithme calcule bien ce que l'on attend (en pratique, cela veut dire pour un calcul de type transformationnel, qu'il termine et rend les valeurs attendues en fonction des entrées considérées). Le deuxième est celui de la performance : quel est le temps de calcul prévisible en fonction de la taille des entrées (notion de complexité en temps) à Quel est la mémoire consommée jusqu'à la terminaison (notion de complexité en mémoire). En général, on arrive à calculer une complexité asymptotique dans le cas pire. Le cas moyen est beaucoup plus complexe. De plus en plus, on pose aussi la question de la consommation en énergie. Enfin, il faut avoir à l'esprit que pour un certain nombre de questions complexes, on peut être intéressé par une réponse algorithmique approchée.

1.3.2 Les structures algorithmiques

Le premier aspect concerne les données manipulées (les "objets"). On peut citer les bits, les entiers, les flottants, les mots, les graphes, les matrices, les images, ... comme objets de bases. Ensuite, il est commode de pouvoir les organiser (les structurer). Les structures de données usuelles sont les tableaux, les listes, les piles, les arbres, ...

Il s'agit ensuite d'organiser les traitements sur les données. Les traitements sont effectués par des instructions pilotées par des structures de contrôle. On utilisera la séquence, la boucle, la récursion, et (éventuellement) le parallélisme.

1.3.3 Premier exemple

Un cours d'algorithmique se doit de citer l'algorithme d'Euclide. Conçu en -400 avant J.C., il s'impose comme le premier procédé algorithmique connu. Il s'agit de calculer le PGCD de deux entiers positifs. L'algorithme proposé est très simple à exprimer en utilisant la récursion :

```
Euclide (a,b) :
1. si b = 0
2.     alors retourner a
```

```
3.     sinon retourner Euclide (b, a mod b)
```

ou encore de façon itérative :

```
Euclide (a,b) :
1. tantque b > 0
2.   r := a mod b ;
3.   a := b ;
4.   b := r
5. retourner a
```

La correction de l'algorithme est d'abord fondée sur un théorème disant que le PGCD(a,b) est égal au PGCD(b, a mod b). Cela justifie que dans le cas où on ne peut pas conclure, le problème est transformé en un autre problème (plus simple, c'est-à-dire plus petit). La démonstration du théorème est classique et repose sur le fait que $(a \ mod \ b) = a -$ partie entière$(a/b) * b)$ et l'identité de Bezout. Il faut ensuite montrer que l'algorithme termine. Ceci est assuré par le fait que le second argument de la fonction Euclide décroit strictement en restant positif.

L'analyse du temps d'exécution est comme habituel plus délicate. On va évaluer le temps d'exécution comme étant proportionnel au nombre d'appels de la fonction Euclide. Celui-ci va dépendre évidemment des valeurs de a et b. Dans ce type de situation, on se contente souvent d'une borne. Ici, on peut affirmer que tous les deux appels le nombre a diminue au moins de moitié. En effet, si $a > b$ (dans le cas contraire, le premier appel s'y ramène), alors $(a \, mod \, b) < a/2$, puisque si $b \leq a/2$, alors $(a \ mod \ b) < b \leq a/2$, et si $b > a/2$, alors le quotient est égal à 0 et le reste égal à $a - b$ qui est donc strictement inférieur à $a/2$. Le nombre d'appels de la fonction est donc majoré par les termes de la récurrence : $A_0 = 0$, $A_a = A_{a/2} + 2$, dont la solution est $A_a = 2log_2(a)$. On dira que la complexité de l'algorithme d'Euclide est au plus de l'ordre de $log \, a$.

Est-ce que l'on peut affiner ce résultat ? Oui, en trouvant le pire cas pour atteindre la borne. Le pire cas est donné par le théorème de Lamé (1844) disant que cela se produit lorsque a et b sont deux nombres de Fibonacci successifs F_n et F_{n+1} (on rappelle que $F_k = f_{k-1} + F_{k-2}$) et que le nombre d'appels est exactement n. On en déduit que dans le pire des cas le nombre d'appels est au maximum de $1,44 * log_2(n)$.

L'évaluation en moyenne est malheureusement très compliquée comme en général. Dans le cas d'Euclide, on connaît le résultat suivant donnant un nombre d'appels de $(12 * ln(2) * ln(n))/\pi^2 + 1,47$.

1.3.4 De la spécification à l'algorithme

En résumé de la démarche de conception d'un algorithme, il faut retenir plusieurs étapes :

- La spécification. Il s'agit de caractériser toutes les entrées légales et les sorties attendues, fonction des entrées.

- La solution algorithmique. Il s'agit d'exprimer la façon de transformer une entrée légale quelconque en une sortie attendue.

- La preuve de correction. On peut distinguer la correction partielle qui consiste à s'assurer que l'algorithme ne calcule que des bonnes valeurs (propriété de sàreté), et la correction totale qui assure que l'algorithme va terminer (propriété de vivacité).

- L'évaluation générique a priori de l'efficacité. Il faut bien se mettre d'accord sur ce qui coàté. Cet aspect est en général défini par un modèle abstrait de machine et renvoit à la théorie de la complexité. C'est un travail difficile et on se contente souvent d'une borne à une complexité asymptotique en fonction de la taille des entrées.

- L'optimalité. C'est une plus value importante. Elle consiste à montrer que pour la classe de machine considérée, il n'existe pas d'autres algorithmes améliorant la complexité de l'algorithme proposé. C'est en général difficile à obtenir et le caractère optimal de nombreux algorithmes célèbres est ouvert.

Il faut noter que cette vision classique de l'algorithme vu comme une fonction transformant des entrées pour produire des sorties au bout d'un temps fini est un peu réductrice. Il y a en effet de nombreuses fonctions informatiques qui ne sont pas transformationnelles, mais réactives et restent actives en continu (les fonctions systèmes par exemple). On peut éventuellement se ramener au cadre fonctionnel en considérant la transformation de séquences d'entrées en séquences de sorties.

1.3.5 Deuxième exemple : le calcul de x^n

Soit x et n des entiers positifs. On pose $y_0 = x$ et on utilise la règle du jeu suivante : on calcule successivement des nombres y_1, y_2, ... en permettant de calculer le prochain y_i comme étant le produit de deus y déjà calculés ($y_i = y_j y_k$, pour $j, k \in [0..i-1]$.

Le but est d'atteindre x^n le plus vite possible, i.e. trouver

$$Opt(n) = min\{i \mid y_i = x^n\}$$

Un algorithme naïf est de calculer les $y_i = y_0 y_{i-1}$. Comme $x^n = y_{n-1}$, le coàt de cet algorithme est de $n-1$ multiplications.

On peut évidemment faire mieux. Par exemple, considérons la méthode binaire. On écrit n en binaire. On remplace chaque 1 par SX et chaque 0 par S. On enlève le premier SX. Le mot obtenu donne une façon de calculer x^n en interprétant S par la mise au carré ($y_i = y_{i-1} y_{i-1}$) et X par la multiplication par x ($y_i = y_{i-1} y_0$).

Il est clair que les règles du jeu sont respectées. D'autre part, à chaque itération, on calcule un nouveau x^m, m devenant de plus en plus grand. Il reste

à montrer que l'on tombe bien sur x^n à un moment donné. Cela est lié aux propriétés de la représentation binaire.

Par exemple, $x^{23} = y_7$.

Le nombre de multiplications nécessaires est le nombre de chiffres de la représentation binaire de n (le nombre de S), plus le nombre de X qui est donnée par le nombre de 1 (moins un) de la représentation binaire : soit

$$\lceil log_2 n \rceil + nb(1) - 1$$

Cet algorithme est toujours pas optimal puisque la méthode binaire donne 6 itérations pour $n = 15$, alors que x^{15} peut se calculer en 5 étapes par la progression suivante : $y_0 = x, y_1 = y_0 y_0 = x^2, y_2 = y_1 y_0 = x^3, y_3 = y_2 y_1 = x^9, y_4 = y_3 y_3 = x^{10}, y_5 = y_4 y_3 = x^{15}$.

La recherche d'un algorithme optimal a fait émerger l'arbre de Knuth. Cet arbre sur les entiers est construit incrémentalement. Le (k+1)-ième niveau de l'arbre est construit en prenant chaque nœeud n du k-ième niveau (ces nœeud définissent un chemin $a_0 a_1 .. a_{k-1}$ depuis la racine de valeur 1), de gauche à droite, et en créant les successeurs de n avec les valeurs $n + a_0, n + a_1, ..., n + a_{k-1}$ si ils n'existent pas déjà dans l'arbre.

Le calcul de x^n s'effectue en considérant le chemin menant à n dans cet arbre. Les valeurs traversées donnent les puissances successives à construire en respectant la règle du jeu.

Il se trouve que cette méthode perd très rarement contre la méthode binaire. Elle est très souvent optimale, mais pas toujours malheureusement. Les plus petits nombres pour lesquels la méthode n'est pas optimale sont 77, 154, 233, ... La recherche de $Opt(n)$ reste ouverte... Aujourd'hui, on essaie de l'encadrer, sachant qu'évidemment $Opt(n) \geq \lceil log_2 n \rceil$ puisqu'on ne peut pas faire mieux que doubler l'exposant à chaque itération.

1.3.6 A vous de concevoir un algorithme

On considère une rue de longueur arbitrairement grande dans les deux directions. Les maisons qui la bordent sont numérotées séquentiellement (la rue peut être modélisée par l'ensemble des entiers relatifs par exemple. Partant d'un endroit donné (par exemple 0), l'objectif du promeneur est de se rendre à une maison précise qu'il saura reconnaître mais dont il ne connaît pas le numéro.

Il s'agit de trouver un algorithme de coàt linéaire $C.n$. n est le numéro de la maison à chercher. Considérant que passer devant une maison coàte 1, calculer C et montrez que votre algorithme est optimal.

Chapter 2

Les limites des algorithmes

2.1 Calculabilité

Il n'est pas difficile d'écrire des programmes dont le comportement reste incompris... Prenons la célèbre conjecture de Collatz qui dit que la suite $u_{n+1} = \begin{cases} u_n/2 & si \quad u_n \; pair \\ 3u_n + 1 & sinon \end{cases}$ converge (1990).

Cela revient à dire que le programme suivant se termine toujours :

```
fonction collatz (n : entier) : entier;
debut
 si n=1 alors retourne 1
 sinon si pair(n) alors retourne collatz(n/2)
       sinon retourne collatz(3*n + 1)
fin
```

Cela reste un problème ouvert, même si il a été expérimenté sur de très grandes plages de données. Les valeurs de n oscillent, grandissent parfois bien au delà de la valeur initiale, mais semblent toujours converger vers 1. On obtient une suite de nombres qui est appelée le vol du nombre de départ. Les nombres de la suite sont appelés les étapes du vol, le plus grand nombre obtenu dans la suite est appelé l'altitude maximale du vol et le nombre d'étapes avant d'obtenir 1 est appelé la durée du vol. Par exemple, l'altitude maximale du nombre 30000 est 250504. Le vol du nombre 30000 a 178 étapes.

On imagine bien alors que la plupart des conjectures de ce type peut se ramener à un problème de décision de l'arrêt d'un programme. Il ne faudra pas être surpris d'apprendre que la question de la terminaison d'un programme est en général difficile. On va même voir que l'on ne pourra pas écrire de programme permettant de décider de la terminaison d'un autre programme en général.

Puisque l'on a défini un algorithme séquentiel comme une fonction se terminant, il est clair que tous les "programmes" ne sont pas des algorithmes. On est donc face à la situation suivante :

- Il y a une limite intrinsèque à la mécanisation : même en supposant une mémoire et un temps aussi grands que l'on veut certains problèmes n'ont pas de solution algorithmique.
- Ces problèmes requièrent évidemment des domaines infinis (correspondent à une question générale).
- Malheureusement la plupart des questions intéressantes sur les programmes ne sont pas calculables.

On comprend donc qu'une bonne partie des travaux consacrés à l'amélioration de la qualité des logiciels à travers des méthodes de preuve et de vérification doivent se situer à la frontière de la calculabilité : plus le modèle de programme est simple, plus on saura raisonner automatiquement dessus, mais moins il capture d'aspects. Par exemple, on peut faire beaucoup de choses sur les automates finis, mais la plupart des problèmes ne peuvent pas être décrits par des automates finis.

2.2 Le modèle de la machine de Turing

Il s'agit d'imaginer le modèle de calcul le plus simple possible, mais aussi le plus expressif possible d'un point de vue théorique afin de capturer ce qu'est "l'essence du calcul". Ce modèle universel est du à Turing (1936) et est toujours aujourd'hui le plus puissant que l'on a pu imaginer.

Une machine de Turing $M = (Q, \Sigma \cup \{\_\}, \delta, q_0, q_f)$ est la donnée :

- d'une mémoire infinie sous forme d'un ruban divisé en cases, remplie avec des lettres d'un alphabet (fini) de ruban $\Sigma \cup \{\_\}$;
- D'une tête de lecture ;
- D'un ensemble fini d'états Q, incluant un état initial q_0 et un état final q_f ;
- D'une fonction de transition $\delta \subseteq Q \times \Sigma \cup \{\_\} \times Q \times \Sigma \cup \{\_\} \times \{L, R\}$ qui, pour chaque état de la machine et symbole se trouvant sous la tête de lecture précise :
  - l'état suivant,
  - un symbole qui sera écrit sur le ruban,
  - un sens de déplacement de la tête de lecture (Gauche ou Droit).

Il reste à donner la façon dont une telle machine s'exécute (notion de *sémantique opérationnelle*).

- Initialement, une donnée (finie) d'entrée sur le ruban est présente, les autres cases sont blanches, la tête se trouve au début du ruban.

- A chaque étape, la machine :
  - lit le symbole se trouvant sous la tête de lecture,
  - remplace ce symbole suivant la fonction de transition,
  - déplace sa tête de lecture d'une case à droite ou à gauche suivant la fonction de transition,
  - change d'état comme indiqué dans la fonction de transition.

Considérons l'exemple de la détection de palindromes. Graphiquement, on utilise la façon habituelle de représenter les automates finis. Les transitions sont étiquetées (condition/action). La condition est la lecture d'un symbole, l'action est une paire désignant l'écriture d'un symbole et l'action de déplacement de la tête de lecture. Dans l'exemple de la figure 2.1 suivante, on note x une lettre quelconque de l'alphabet différente de # et $\_$. On fait aussi l'hypothèse que si aucune transition n'est tirable, la machine stoppe dans un état d'erreur.

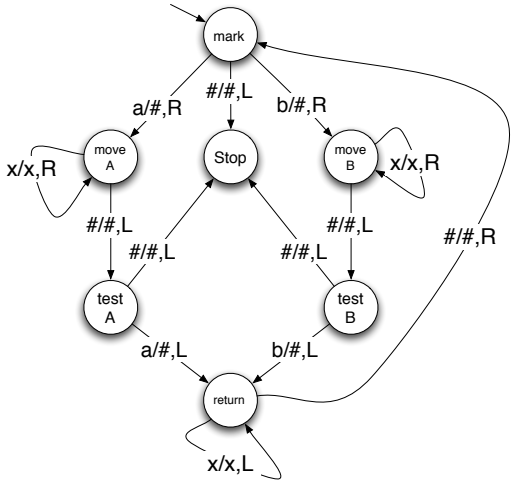

Figure 2.1: Détection de palindromes par machine de Turing

On utilise la lettre # comme marqueur. Le palindrome initial est entré sur le ruban. On peut regarder par exemple le fonctionnement de la machine pour l'entrée *abba*## (le reste du ruban étant rempli de blancs ($\_$)).

2.3 Les fonctions calculables

De façon plus abstraite, on peut considérer les machines de Turing comme des machines calculant des fonctions. Une machine de Turing calcule une fonction $f : \Sigma^* \rightarrow \Sigma^*$, si pour toute entrée w, elle s'arrête toujours dans une configuration

où $f(w)$ se trouve sur le ruban. Dans la mesure où l'on n'a pas réussi à trouver des machines plus puissantes, la thèse (Church) est que les fonctions calculables par les machines de Turing sont les fonctions calculables par un algorithme.

Il s'avère que partant du modèle de base des machines de Turing, on peut construire des modèles beaucoup plus élaborés, mais pas plus puissants du point de vue des fonctions qu'ils calculent. Les machines de Turing peuvent simuler :

- plusieurs bandes,
- une bande bi-infinie,
- un contrôle non déterministe,
- des automates à plus de 2 piles,
- des automates à file,
- des machines avec mémoire à accès direct (RAM),
- ...
- les programmes séquentiels (avec abstraction de mémoire infinie).

Les machines parallèles, sous leurs différentes formes, n'ont pas changé l'universalité du modèle des machines de Turing. La thèse de Church-Post reste ouverte...

On peut s'amuser à reconstruire toutes les fonctions de base des calculateurs avec les machines de Turing, même si en pratique les processeurs ne fonctionnent pas comme les machines de Turing (en effet, les opérations arithmétiques et logiques sont cablées et non interprétées par machine ; de même la mémoire matérielle est structurée). Les figures suivantes 2.2 et 2.3 donnent des exemples de conversion d'entiers dans l'écriture "batons" vers une représentation binaire inversée. Avec cette représentation, on peut réaliser facilement un additionneur binaire. Réaliser des fonctions plus compliquées peut se faire en connectant des machines de Turing entre elles, la sortie de l'une étant l'entrée de l'autre.

2.4 Les fonctions non calculables

Maintenant que l'on sait décrire des fonctions calculables par machines de Turing, il est intéressant de comprendre à quoi peuvent ressembler des fonctions non calculables. Un exemple fort utile, car beaucoup de problèmes non calculables peuvent s'y ramener, est celui du problème de la correspondance de Post (PCP).

Il se définit ainsi :

- Soit une collection de dominos $P = \{[t_/b_1], ..., [t_k/b_k]\}$, où les t_i et b_i sont des mots dans un alphabet Σ.
- Une correspondance est une suite $i_1, ..., i_l$, où $t_{i1}...t_{il} = b_{i1}...b_{il}$.

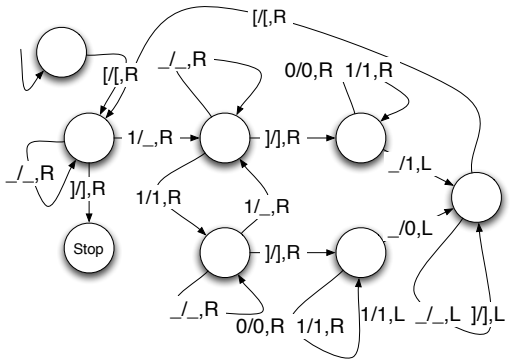

Figure 2.2: Conversion bàtons - binaire inversé. L'entrée est fournie sous la forme [11111].

- Le problème est de déterminer si P possède une correspondance.

Exemple 1 : $\{[b/ca], [a/ab], [ca/a], [abc/c]\}$ a la correspondance $[a/ab][b/ca][ca/a][a/ab][abc/c]$.

Exemple 2 : $\{[abc/ab], [ca/a], [acc, ba]\}$ n'a pas de correspondance.

Le nombre n de dominos constitue la taille du problème PCP(n) considéré. La longueur (en nombre de dominos) du PCP est la longueur de la correspondance la plus courte que l'on peut construire avec le jeu de dominos donné. Par exemple, avec le jeu PCP(3) $\{[100/1], [0, 100], [1, 00]\}$, on obtient la longueur 7 en considérant la séquence de dominos suivante : 1311322 (1001100100100). Un autre exemple de taille 3 ressemblant est $\{[100/1], [0, 100], [1, 0]\}$. Sa longueur augmente pourtant considérablement et est de 75 avec 2 plus petites correspondances. L'exemple suivant de taille 4 $\{[000/0], [0/111], [11, 0], [10, 100]\}$ est de longueur 204 et n'admet qu'une plus petite correspondance.

Ce problème a été présenté par Emil L. Post en 1946 comme un problème typique indécidable, même pour des tailles données a priori. Le PCP de taille 2 est décidable (l'algorithme et sa preuve restent complexes). Le PCP de taille supérieure à 7 est indécidable. La décidabilité de PCP de taille entre 3 et 6 reste un problème ouvert. Le PCP est souvent utilisé pour prouver l'indécidabilité de nombreux problèmes par une technique de réduction. La réduction consiste à montrer que si le problème considéré était décidable, alors il en serait de même pour un PCP déjà prouvé indécidable.

De façon plus poétique, je rappelle l'holorime de Victor Hugo : "galamandelareinealatourmagnanime" avec la suite (4,2,3,6,1,7,5) du jeu suivant : $\{[ala/tour], [aman/dela], [dela/reine], [gal/galaman], [magnanime/anime], [reine/ala], [tour/magn]\}$.

```
Gal, amant de la reine, alla, tour magnanime,
Galamment de l'arène, à la tour Magne, à Nîmes.
```

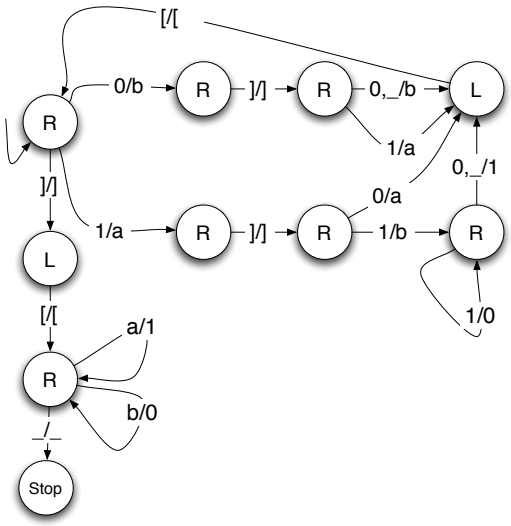

Figure 2.3: Additionneur d'entiers binaires inversés. L'entrée est donnée sous la forme [1011]100111 pour 13+57 par exemple. La sortie sera [1011]0110001, soit 70 en binaire inversé.

L'indécidabilité de PCP de taille supérieure à 7 est prouvée en se ramenant à un calcul par machine de Turing. L'idée est la suivante. On montre que quelque soit une machine de Turing M et une entrée w, on peut construire une instance P de PCP dans laquelle une correspondance est calculable pour M sur w.

Pour simplifier, on considère le problème P', instance de PCP obligeant de commencer avec le premier domino : $P' = \{[\#/\#q_0w\#]\} \cup \{[qa/br] \mid \delta(q,a) = (r,b,R)\} \cup \{[cqa/rcb] \mid \delta(q,a) = (r,b,L), c \in \Sigma\} \cup \{[a/a]\} \cup \{[\#/\#], [\#/\_\#]\} \cup \{[aq_f/q_f], [q_fa/q_f]\} \cup \{[q_f\#\#/\#]\}$. Le PCP général est construit en modifiant $[\#/\#q_0w\#]$ par $[*\#/*\#q_0w\#*]$, et chaque $[t/b]$ par $[*t/b*]$. On ajoute enfin $[* <> / <>]$. Donc, si on pouvait déterminer si une instance de PCP possède une correspondance, on serait capable de savoir une machine M quelconque accepte w ou non. L'indécidabilité de PCP est donc impliquée par l'indécidabilité de "l'arrêt de la machine de Turing".

On en vient donc au problème de l'arrêt d'une machine de Turing quelconque. La question est, étant donnée une machine de Turing M et une configuration initiale du ruban w, est-ce que M atteint son état final q_f ? La question universelle revient donc à essayer de construire une machine de Turing capable de décider si une machine de Turing s'arrête ou non. Pour cela, il faut représenter la machine à analyser et son entrée sur le ruban de la machine qui doit décider l'arrêt. On peut prendre un codage fini quelconque dans lequel la machine est transformée en une suite finie de symboles.

Par exemple, la figure 2.4 suivante donne un codage possible d'une machine

reconnaissant le langage $\{a^n b^n \mid n \geq 0\}$ en utilisant les symboles $\{0, 1, , , (,)\}$.

```
(                 // Début du codage de M
101,                // Nombre d'états : 5 en binaire
10,                 // Nombre de symboles de Σ: 2 en binaire
101,                // Nombre de symboles de Γ: 5 en binaire
(                 // Début du codage des transitions
(0,1,1,11,1),         // Transition (0,a) --> (1,A,R)
(0,100,11,100,1),     // Transition (0,B) --> (3,B,R)
(1,1,1,1,1),          // Transition (1,a) --> (1,a,R)
(1,10,10,100,0),      // Transition (1,b) --> (2,B,L)
(1,100,1,100,1),      // Transition (1,B) --> (1,B,R)
(10,1,10,1,0),        // Transition (2,a) --> (2,a,L)
(10,11,0,11,1),       // Transition (2,A) --> (0,A,R)
(10,100,10,100,0),    // Transition (2,B) --> (2,B,L)
(11,100,11,100,1),    // Transition (3,B) --> (3,B,R)
(11,0,100,0,1),       // Transition (3,#) --> (4,#,R)
),                // Fin du codage des transitions
0,                                          // Codage de l'état initial
(                 // Début du codage de l'ensemble F des états finaux
100                 // Le seul état final est l'état 4
)                 // Fin du codage de l'ensemble F des états finaux
)                 // Fin du codage de M
```

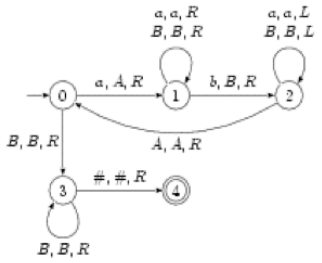

Figure 2.4: Codage d'un machine de Turing

2.5 L'argument de diagonalisation

On va utiliser un argument de comparaison de cardinalité sur les ensembles infinis. On rappelle la hiérarchie des infinis de Cantor, permettant de dire que $\mid A \mid \leq \mid B \mid$ si il existe une fonction injective de A vers B, et $\mid A \mid = \mid B \mid$ si il existe une fonction bijective de A vers B. Soit $\mathbb{N}$ les entiers naturels, $\mathbb{N}_P$ les entiers pairs, $\mathbb{R}$ les nombres réels, $\mathcal{P}(\mathbb{N})$ les parties de $\mathbb{N}$ et $\mathcal{P}(\mathbb{R})$ les parties de $\mathbb{R}$. On a $\mid \mathbb{N} \mid = \mid \mathbb{N}_P \mid$, $\mid \mathbb{N} \mid < \mid \mathbb{R} \mid = \mid \mathcal{P}(\mathbb{N}) \mid < \mid \mathcal{P}(\mathbb{R}) \mid$.

Le point important ici est la preuve que $\mid \mathbb{N} \mid < \mid \mathcal{P}(\mathbb{N}) \mid$. Considérons une relation binaire R sur un ensemble A. Notons $R_x = \{y \mid y \in A \wedge (x, y) \in R\}$ les images par R d'un élément x. Considérons aussi l'ensemble diagonal $D = \{y \mid y \in A \wedge (y, y) \notin R\}$. Par définition $\forall x \in A, R_x \neq D$. Supposons au contraire que $\mid \mathbb{N} \mid = \mid \mathcal{P}(\mathbb{N}) \mid$, alors on peut énumérer les sous-ensembles $B_0, B_1, ...$ de $\mathbb{N}$ avec la bijection $\mathcal{P}(\mathbb{N}) \to \mathbb{N}$. Mais alors l'ensemble diagonal $\{y \mid y \notin B_y\}$ de la relation $\{(x, y) \mid x \in B_y\}$ manque à la liste, d'où la contradiction.

Cette propriété simple suffit pour affirmer qu'il y a plein de langages qui ne correspondent à aucune machine de Turing. En effet, l'ensemble des machines de Turing sur un alphabet Σ (MT_Σ) est dénombrable : $\mid MT_\Sigma \mid = \mid \mathbb{N} \mid$, puisque l'on peut encoder une machine de Turing par une chaîne finie de symboles et que les chaînes sont dénombrables (prendre par exemple un ordre lexicographique sur les chaînes valides codant les machines de Turing). L'ensemble des langages

sur Σ est $\mathcal{P}(\Sigma^*)$. On a $\mid \Sigma^* \mid = \mid \mathbb{N} \mid$, donc $\mid \mathcal{P}(\Sigma^*) \mid > \mid \mathbb{N} \mid = \mid MT_\Sigma \mid$.

Revenons maintenant au problème de l'arrêt. Supposons une machine de Turing H qui décide cette propriété :
H((M,w)) = *accept* si M(w) s'arrête
= *reject* sinon

Considérons alors la machine D(w) qui :

- Simule H((w,w)) ("bootstrap"),
- boucle indéfiniment si H retourne *accept*,
- retourne *accept* si H retourne *reject*.

Considérons maintenant la machine D en entrée d'elle-même (D(D)). Si D(D) s'arrête, H((D,D))=*reject* et donc D(D) boucle. Si D(D)) ne s'arrête pas, H((D,D)) = *accept* et donc D(D) s'arrête. Il y a contradiction : D (et donc H) ne peuvent exister.

2.6 Conséquences

Par une analyse statique d'un programme arbitraire il est impossible de déterminer, en général :

- si le programme possède une boucle infinie,
- si il bloque,
- si un test dans le programme est passant ou pas,
- si une exécution arbitraire (suite d'instructions) peut être exécutée,
- si une variable sera utilisée,
- etc., etc.

Les analyses fournies par les compilateurs sont des approximations (qui s'améliorent au cours du temps et en fonction de l'augmentation des puissances).

Mais il ne faut pas être trop pessimiste. Les questions mentionnées ne peuvent pas être intégralement résolues de manière générale. Ceci ne veut pas dire qu'elles ne puissent pas être résolues dans un grand nombre de cas particuliers, par des algorithmes spécifiques à ces cas particuliers. C'est un des sujets de l'informatique et du génie informatique...

2.7 Décidabilité et complexité

Il est important de ne pas confondre les problèmes indécidables et les problèmes à haut niveau de complexité. Un problème indécidable est un problème logiquement impossible à résoudre, ni avec un ordinateur présent ni futur (à moins de trouver une façon différente de le formuler). Aucun algorithme ne peut exister.

Un problème à haute complexité (par exemple, exponentiel) est un problème qui demande des calculs laborieux Des algorithmes existent. Le calcul peut être effectué sur des petits ensembles. Le calcul pourrait devenir pratique avec des ordinateurs futurs. Ces questions de complexité seront abordées dans la section suivante.

2.8 A vous de voir

Sauriez-vous démontrer qu'un automate fini à file à la puissance en fait d'une machine de Turing ? Les actions d'un tel automate sont :

- lire un symbole en tête de file,
- éventuellement écrire un symbole en queue de file,
- éventuellement changer d'état.

Comparez avec les automates finis à pile.

Chapter 3

Aperçu sur la théorie de la complexité

3.1 Complexité en temps

Dans la conception d'un algorithme, il est nécessaire d'aller plus loin que juste savoir si le problème est calculable ou pas et évaluer son coàt. La question est est-ce que ce sera "calculable" en pratique ? Ce n'est pas toujours le cas et dépend en général du modèle de machine utilisé et éventuellement de la représentation des données.

Par exemple, si on s'intéresse à la complexité en temps et si on considère le modèle de la machine de Turing $M(n)$. On suppose que M termine sur toutes les entrées n. La *complexité temporelle* peut être définie comme la fonction $f(n)$ qui donne le nombre de transitions exécutées par la machine.

Il est très difficile en général de calculer $f(n)$ de façon exacte. On se contente souvent d'une *analyse asymptotique*. La méthode est la suivante : on considère une entrée de grande taille n et on ne garde que le terme dominant. On peut même oublier le coefficient multiplicateur pour ne garder que l'ordre de grandeur, que l'on note avec la notation "O".

Par exemple, si $f(n) = 5n^3 + 2n^2 + 22n + 6 = O(n^3)$.

Cette notion se formalise. On dit que deux fonctions f et g sont du même ordre ($f(n) = O(g(n))$ si et seulement si $\exists c, n_0 : \forall n \geq n_0,\ f(n) \leq c.g(n)$. Dans l'exemple précédent, on peut prendre $n_0 = 10$ et $c = 6$ puisque $\forall n \geq 10, 5n^3 + 2n^2 + 22n + 6 \leq 6n^3$.

On remarque que l'ordre de grandeur n'est pas sensible à la base du logarithme puisque $log_b n = log_2 n / log_2 b$. C'est pourquoi on notera simplement des complexités en $O(logn)$ sans plus de précision. Le calcul des ordres de grandeur peut être facilité par l'utilisation de propriétés arithmétiques comme $O(n^2) + O(n) = O(n^2)$.

Prenons par exemple le langage formé des chaînes binaires $A = \{0^k 1^k\}_{k \geq 0}$. Ce langage peut être reconnu par une machine de Turing qui effectue les

opérations suivantes :

1. Parcours du ruban et rejette si un 0 est trouvé à droite d'un 1.
2. Tant qu'il existe au moins un 0 et un 1, parcours du ruban et effacement d'un 0 et d'un 1.
3. Rejette si il reste un 0 ou un 1, accepte sinon.

Soit n la taille de la chaîne d'entrée. L'étape 1 coûte $2n$ (parcours et retour au début du ruban). Dans la deuxième étape, on peut parcourir au pire n fois la boucle sur des chaînes dont la longueur diminue de 2 à chaque fois. Ce qui donne un coût de $2\sum_{i=0}^{n-1} = n(n-1)$. La troisième coûte n. On obtient donc une complexité asymptotique de $O(n^2)$. En disposant d'un langage de programmation standard, quelle complexité peut-on atteindre ?

3.1.1 La classe $TIME(f(n))$

La notion de classe de complexité permet de regrouper les problèmes de difficulté comparable et de bien distinguer les niveaux de difficulté. Intuitivement, on sent bien qu'il y a une différence fondamentale entre les questions que l'on pourra résoudre en temps exponentiel et celles que l'on pourra résoudre en temps polynomial. En pratique, la barrière entre ce qui est calculable dans un temps raisonnable et ce qui ne le sera pas passe souvent par cette frontière.

Considérons une fonction f donnant la complexité en fonction de la taille du problème n donnée en par un entier ($f : \mathbb{N} \rightarrow \mathbb{R}^+$). Pour avoir une définition simple, on ramène le problème du calcul au problème de la décision de l'appartenance d'une chaîne à un langage formel. $TIME(f(n))$ est l'ensemble de tous les langages décidables en $O(f(n))$ par une machine de Turing. On a vu par exemple que $A \in TIME(n^2)$. Peut-on mettre le problème A dans une classe de complexité moindre ?

La réponse est évidemment positive. On peut commencer par améliorer la machine du paragraphe précédent en permettant à la machine de tester la parité du nombre de 0 et 1. Cela s'effectue aisément par un comptage modulo 2, mobilisant deux états de l'automate fini de contrôle de la machine de Turing. La machine devient :

1. Parcours du ruban et rejette si un 0 est trouvé à droite d'un 1.
2. Tant qu'il existe au moins un 0 et un 1 :
   - Rejette si le nombre de bits de la chaîne est impair,
   - Parcours du ruban et effacement d'un 0 sur deux et d'un 1 sur deux.
3. Rejette si il reste un 0 ou un 1, accepte sinon.

La machine est correcte puisque vérifier la parité à chaque tour de boucle revient à tester l'égalité des longueurs de la chaîne de 0 et de la chaîne de 1. Par exemple la chaîne 000000000000001111111111111 est "épluchée" ainsi à chaque

tour : 000000111111, 000111 et 01. Il y a au plus $1 + log_2(n)$ itérations. Le coàt global est donc de $O(n) + (1 + log_2(n))(O(n) + O(n)) + O(n) = O(nlogn)$.

Pour améliorer la complexité et obtenir un coût linéaire, qui sera évidemment optimal, il faut changer le type de la machine ("l'algorithme"). Intuitivement, le problème revient à savoir compter les 0 et les 1 jusqu'à des valeurs arbitrairement grandes. Cela ne peut donc pas se faire à l'intérieur du contrôle fini de la machine et on va devoir utiliser le ruban. Pour simplifier, on va utiliser une machine à deux rubans (on sait par ailleurs qu'une telle machine est simulable par une machine de Turing à un seul ruban). La machine proposée est :

1. Parcours du ruban et rejette si un 0 est trouvé à droite d'un 1.

2. Parcours des 0 du ruban 1 jusqu'à trouver un 1, tout en copiant les 0 sur le ruban 2.

3. Parcours des 1 restants jusqu'à la fin du ruban 1. Pour chaque 1 lu, effacer un 0 sur le ruban 2. Rejette si le ruban 2 devient vide.

4. Accepte si le ruban 2 est vide, sinon rejette.

Cet exemple de décision du langage A illustre bien le fait que la classe de complexité obtenue dépend du modèle de calcul considéré. Cela raffine bien la notion de décidabilité puisque les machines à 1 ou 2 rubans décident la même classe de langages.

La méthode de simulation d'une machine de Turing à 2 rubans par une machine de Turing à 1 ruban fait que chaque étape de la première nécessite n étapes de la deuxième. Ceci permet d'énoncer le théorème suivant.

Théorème 1 *Pour toute fonction $f(n) \geq n$, si un problème est résolu en $O(f(n))$ sur une machine de Turing à 2 rubans, il est soluble en $O(f^2(n))$ sur une machine de Turing à 1 ruban.*

3.2 La puissance du non-déterminisme

Dans notre définition des machines de Turing, on a permis à l'automate de contrôle d'être non-déterministe au sens classiques de automates finis (dans un état donné de la machine, le symbole lu sous la tête de lecture peut déclencher des transitions différentes). Les machines de Turing déterministes et non-déterministes reconnaissent la même classe de langages. Il n'en est pas de même pour la complexité. Intuitivement, pour fonctionner, une machine de Turing non-déterministe va être obligée d'explorer toutes les possibilités de mouvement en construisant un arbre dont les feuilles vont être étiquetées par le rejet ou l'acceptation. Dans le cas déterministe, on a juste la construction d'une suite séquentielle d'actions, la suite se terminant par un rejet ou une acceptation. Il en découle une puissance de discrimination pour les machines non-déterministes, donnée par le théorème suivant.

Théorème 2 *Pour toute fonction $f(n) \geq n$, si un problème est résolu en $O(f(n))$ sur une machine de Turing non-déterministe, il est soluble en $2^{O(f(n))}$ sur une machine de Turing déterministe.*

La preuve réside dans la méthode de simulation d'une machine de Turing non-déterministe par une machine de Turing déterministe. La technique consiste à construire l'arbre d'exécution. Pour assurer la terminaison, cet arbre est construit en largeur d'abord. Le simulateur a 3 rubans. Sur une entrée de taille n, chaque branche dans l'arbre d'exécution a une longueur au pire de $f(n)$. Le nombre total de feuilles est au pire de $b^{f(n)}$ (b étant le degré sortant). Le nombre de nœuds est du même ordre (par exemple dans l'arbre binaire complet de hauteur n, il y a 2^n feuilles et $2^n - 1$ nœuds internes). D'oà la complexité de l'ordre de $2^{O(f(n))}$. La simulation des 3 rubans par un ruban coàte en fait un carré, qui est absorbé dans la complexité exponentielle de la simulation.

On peut regarder comment fonctionne le simulateur. Il possède 3 rubans : le premier est le ruban d'entrée contenant la donnée, le deuxième est le ruban de simulation qui va contenir la machine de Turing à exécuter, et le troisième sert de mémoire d'adresses pour permettre l'exploration de l'arbre d'exécution.

Les nœuds de l'arbre sont désignés par une chaîne sur $\{1, 2, ..., b\}$ donnant la liste des choix successifs de la machine. Le principe de fonctionnement est le suivant :

1. Initialement, le ruban 1 contient la donnée d'entrée w, le ruban 2 la machine de Turing à simuler et le ruban 3 est vide.

2. Copie du ruban 1 sur le ruban 2.

3. Utilisation du ruban 2 pour simuler la machine avec entrée w sur une branche du calcul non-déterministe. Avant chaque étape, consulte le symbole sur le ruban 3 pour déterminer le choix à faire. Si le ruban 3 est vide ou si le nœud atteint est rejet, aller à l'étape 4. Si accepte, arrêt.

4. Remplace la chaîne du ruban 3 avec la chaîne suivante dans l'ordre lexicographique. Retourner à l'étape 2.

3.3 La classe P

Les deux théorèmes précédents montrent des correspondances de complexités différentes entre machines de Turing ($O(n)$ se transforme en $O(n^2)$, dans le cas des machines à 1 ou 2 rubans ; $O(n)$ se transforme en $O(2^n)$, dans le cas des machines de Turing déterministes et non-déterministes). Les différences polynomiales peuvent être considérées faibles tandis que les différences exponentielles sont considérées très significatives. Par exemple, pour $n = 1000$, n^3 vaut 1 milliard (c'est grand mais accessible par un ordinateur), 2^n est un nombre considérable, inaccessible en pratique. La “force brute” consistant à explorer l'ensemble des solutions d'un problème (par exemple pour la factorisation en nombres premiers, problème utilisé en cryptographie) est souvent exponentielle.

Définition 1 *P est la classe des langages qui sont décidables en temps polynomial sur une machine de Turing déterministe.* $P = \bigcup_k TIME(n^k)$.

P est invariant pour tous les modèles de calcul qui sont polynômialement équivalents à une machine de Turing déterministe. P correspond grossièrement à la classe de problèmes qui peuvent être résolus "raisonnablement" sur un ordinateur. Nous verrons dans la suite du cours de nombreux exemples de problèmes qui peuvent être résolus en temps polynomial. Par exemple le problème du chemin dans un graphe est dans la classe P ($Chemin = \{(G, s, t) \mid$ G est un graphe orienté qui possède un chemin de s vers t$\}$). En effet, étant donné (G, s, t), on connaît un algorithme polynomial qui décide si il existe un chemin de s à t dans le graphe G.

Il faut quand même ne pas utiliser directement la force brute. Celle-ci ne produit pas un algorithme polynomial. Si m est le nombre de sommets du graphe, un chemin peut être de longueur m. Le nombre de chemins potentiels est de m^m.

On propose l'algorithme suivant pour (G, s, t) :

1. Marquer le sommet s.
2. Répéter jusqu'à qu'il n'y ait plus de sommet supplémentaire marqué :
   - Parcourir tous les arcs du graphe. Si un arc (a, b) est trouvé avec a marqué et b non marqué, marquer le sommet b.
3. Si t est marqué, accepte. Sinon rejet.

Cet algorithme possède $m + 2$ étapes. Le parcours est lui aussi polynomial. L'ensemble reste donc polynomial.

3.4 La classe NP

Cette classe va utiliser la puissance de calcul des machines de Turing non-déterministes.

Définition 2 $NTIME(f(n)) =$
$\{L | L$ *est un langage décidable en temps* $O(f(n))$
par une machine de Turing non-déterministe$\}$. $NP = \bigcup_k NTIME(n^k)$.

Comme exemple de problème NP, on peut citer le problème de la clique. Une clique dans un graphe non orienté est un sous-graphe dans lequel chaque paire de sommets est connectée par un arc. Une $k - clique$ est une clique contenant k sommets. $Clique = \{(G, k) \mid G$ possède une k-clique$\}$.

On dit qu'un problème L est $Co - NP$ si et seulement si le complémentaire de L est dans NP. Mais on ne sait toujours pas si $Co - NP = NP$!

D'ailleurs, une des plus fameuses questions ouvertes en informatique théorique et en mathématiques est de savoir si $P = NP$. Beaucoup pensent que non et de nombreux problèmes se réduisent à cette question. C'est le cas

par exemple du problème SAT, pour lequel Cook et Levin ont montré en 1970 que $SAT \in P$ si et seulement si $P = NP$.

Le problème SAT est le suivant. On considère l'ensemble des formules booléennes (par exemple, $f = (\neg x \wedge y) \vee (x \wedge \neg z)$). On dit que f est satisfiable si il existe des valeurs pour x, y et z telles que f est vraie (010 dans notre exemple). $SAT = \{(f) \mid f \text{ est satisfiable}\}$.

Etant donnée une formule booléenne, on peut toujours la mettre sous une forme normale conjonctive (une conjonction de disjonctions) en utilisant les lois de De Morgan). On peut considérer le problème simplifié $3SAT$ comme étant SAT réduit aux formules normales conjonctives avec des disjonctions de taille 3.

Théorème 3 *Si Clique est soluble en temps polynomial, $3SAT$ l'est aussi et réciproquement.*

Pour prouver ce théorème, on considère la formule f suivante : $f = (a_1 \vee b_1 \vee c_1) \wedge (a_2 \vee b_2 \vee c_2) \wedge ... \wedge (a_k \vee b_k \vee c_k)$. Les a_i, b_i et c_i sont appelés les littéraux et sont soit une variable, soit la négation d'une variable. On considère alors un graphe G correspondant en identifiant un nœud par littéral et en reliant tous les sommets entre eux, sauf si ils appartiennent à la même disjonction ou si l'un est la négation de l'autre. On va montrer que f est satisfiable si et seulement si G possède une $k - clique$.

Supposons f satisfiable. Au moins un littéral est vrai dans chaque disjonction. On en prend un de ce type dans chaque disjonction. Le sous-graphe correspondant est une $k - clique$ (ses sommets n'appartiennent pas à la même disjonction et ils ne peuvent pas être contradictoires).

Réciproquement, supposons que G a une $k - clique$. Deux sommets d'une telle clique ne peuvent appartenir à la même disjonction. Donc chaque disjonction contient exactement un sommet de la clique. On peut trouver des valeurs pour les variables booléennes de f de telle sorte que les variables des sommets de la clique soient vraies. Ceci est toujours possible puisqu'il ne peut pas y avoir de contradiction dans la clique. Le résultat est que f est vrai, donc f est satisfiable.

3.5 La complexité en espace

La complexité spatiale d'une machine de Turing qui s'arrête sur toutes les entrées est le nombre maximum de cases du ruban qu'elle lit. On peut définir la classe $SPACE(f(n))$ et la classe $NSPACE(f(n))$ pour les machines de Turing non-déterministes.

Par exemple, SAT est de complexité spatiale linéaire $SPACE(O(n))$. Ceci se prouve en remarquant que l'on peut évaluer la valeur de vérité de la formule pour chaque affectation possible des variables. Si on trouve vrai une fois, la machine accepte et rejette sinon. La machine n'a besoin de stocker que l'affectation courante des variables.

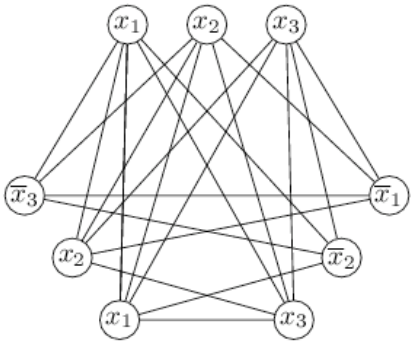

Figure 3.1: Graphe associé à la formule
$(x_1 \vee x_2 \vee x_3) \wedge (\neg x_1 \vee \neg x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee \neg x_3)$

Théorème 4 $NSPACE(f(n)) \subseteq SPACE(f^2(n))$

Ce théorème signifie que le comportement non-déterministe est spacialement plus puissant, mais dans une moindre proportion que ce qui ce passe pour la complexité temporelle. La simulation du non-déterminisme par une machine déterministe utilise la récursion, mise en œuvre par une pile d'appel (dont la place prise dépend de la profondeur de la récursion et permet donc une réutilisation de la mémoire sur "back-track").

On peut définir d'autres classes de complexité comme $PSPACE = \bigcup_k SPACE(n^k)$ et $EXPTIME = \bigcup_k TIME(2^k)$.

On a la hiérarchie suivante :

$$P \subseteq PSPACE = NPSPACE \subseteq EXPTIME$$

.

3.6 Exercice en liaison avec les grammaires formelles

Sauriez-vous montrer que tout langage défini par une grammaire "context-free" est dans la classe P ?

Chapter 4

Les grands théorèmes du tri et de la recherche

4.1 Introduction : le tri par insertion

La première idée pour trier un tableau $A[1..n]$ d'éléments ordonnables (ici on prend pour simplifier des entiers) est de pratiquer la technique du joueur de cartes. On parcourt séquentiellement le tableau et on place l'entier courant au milieu des entiers déjà traités et triés. Le problème est évidemment le coût de cette insertion qui peut obliger à décaler un ensemble important de valeurs. Le programme de cet algorithme est le suivant :

```
  pour j de 2 à n
1: faire clé := A[j] ;
2:       i := j - 1 ;
         tant que i > 0 et A[i] > clé
3:       faire A[i+1] := A[i] ;
4:            i := i - 1 ;
5:       A[i+1] := clé
```

La correction de cet algorithme est assez évidente. Une façon standard de formaliser la preuve et de s'affranchir de la boucle sur j est d'intuiter un *invariant* de boucle. Ici, l'idée est de montrer qu'à l'étape j, $A[1, j-1]$ est trié. On le prouve par récurrence sur j. C'est vrai à l'étape initiale pour $j = 2$. Supposons maintenant que $A[1, j-1]$ est trié. Le corps de la boucle déplace $A[j-1]$, $A[j-2]$, ... d'une case vers la droite jusqu'à trouver la bonne position pour $A[j]$ qui est inséré. Donc, à la fin de cette étape, $A[1, j]$ est trié. La boucle sur j termine avec $j = n+1$. L'invariant prouve la correction de l'algorithme à ce moment.

Pour évaluer la complexité temporelle de l'algorithme, on va compter le nombre d'affectations (les instructions numérotées du programme). Soit x_j le nombre de fois que la boucle "tant que" est passante pour la valeur de j. Les

affectations 1,2 et 5 sont exécutées $n-1$ fois. Les affectations 3 et 4 sont exécutées $\sum_{j=2}^{n} x_j$ fois. Ce qui fait un coàt total $T(n) = 3(n-1) + 2\sum_{j=2}^{n} x_j$.

Pour continuer le calcul, il faut faire des hypothèses sur les données. Le cas le plus favorable est celui du tableau déjà trié. On a alors pour tous les j, $x_j = 0$, donc $T(n) = 3(n-1)$. On obtient une complexité en temps linéaire, mais qui évidemment ne sera pas vraiment effective. Le cas le plus défavorable est celui du tableau trié à l'envers, puisqu'il déclenche un insertion systématique. $x_j = j-1$, donc $T(n) = 3(n-1) + n(n-1) = (n-1)(n+3)$. On obtient une complexité en temps quadratique. On peut aussi s'intéresser à une situation moyenne. Par exemple, si les données sont tirées aléatoirement de façon uniforme, en moyenne $x_j = (j-1)/2$, donc $T(n) = (n-1)(3+n/2)$. Cela ne modifie pas la complexité asymptotique en $O(n^2)$.

Le paragraphe suivant va montrer que cette complexité peut être améliorée en appliquant un paradigme algorithmique important : la stratégie dite "diviser pour régner".

4.2 Diviser pour régner

Cette stratégie conduit à un algorithme de tri célèbre et habituellement employé dans les applications informatiques : le "tri rapide" ("quick sort" en anglais).

Diviser le problème, c'est trouver arbitrairement un "pivot" à l'index q permettant de diviser le tableau A en 2 sous-tableaux $A[1..q-1]$ et $A[q+1..n]$ tels que chaque élément du premier est inférieur ou égal au pivot et chaque élément du deuxième est supérieur ou égal au pivot. Le pivot est donc à sa place finale.

Régner consiste à trier les 2 sous-tableaux par des appels récursifs à la procédure de tri (celle-ci va donc de nouveau décider d'un pivot dans chaque sous-tableau et se rappeler). L'arrêt de la récursion a lieu pour des tableaux de taille 1.

L'appel initial à la procédure est *Tri-Rapide (A, 1, n)*. Le code est le suivant :

```
Tri-Rapide (A, p, r) :
 si p < r
   alors q := Partition (A, p, r) ;
         Tri-Rapide (A, p, q-1) ;
         Tri-Rapide (A, q+1, r)
```

La correction est évidente puisque chaque pivot est correctement placé si la fonction *Partition* est correcte. Pour la complexité temporelle, le point clé est aussi la partition : si on arrive à faire le partage en temps linéaire, le coût sera $TR(n) = c.n + TR(u) + TR(v)$, avec $u + v = n$ et $TR(1) = 0$.

Dans le cas le plus favorable, le partage va être équilibré, la partition divisant le tableau en deux parties "égales". C'est-à-dire $u, v \leq \lceil n/2 \rceil$. Donc $TR(n) \leq 2TR(\lceil n/2 \rceil) + c.n$. On en déduit $TR(n) \leq log_2 n.c.n$. On obtient donc une complexité en $O(nlogn)$.

Pour obtenir une partition en temps linéaire, l'idée est la suivante. On balaye le tableau en maintenant deux positions, l'une à gauche et l'autre à droite, telles que les éléments qui précèdent la position de gauche soient inférieurs ou égaux au pivot et ceux qui se trouvent entre la position de gauche et la position de droite soient supérieurs au pivot.

La position de droite est avancée à droite tant que la condition est respectée. Quand ce n'est plus possible, l'élément de la position droite est inférieur ou égal au pivot et celui qui suit la position gauche est strictement plus grand que le pivot. Les deux sont alors échangés, les positions peuvent se décaler d'un coup vers la droite, et le processus continue. Pour simplifier, on choisit comme pivot le dernier élément du tableau.

L'algorithme se décrit ainsi :

```
Partition (A, p, r) :
    x := A[r] ;
    i := p - 1 ;
    pour j de p à r-1
       faire si A[j] <= x
             alors i := i+1 ;
                   permuter A[i] et A[j]
    permuter A[i+1] et A[r]
    retourner i+1
```

$A[r]$ est le pivot autour duquel se fera le partitionnement. La correction du partitionnement s'effectue en considérant l'invariant de boucle $P(i,j)$, pour des indices i et j donnés. On affirme que $P(i,j) = (\forall k, p \leq k \leq i, A[k] \leq x) \wedge (\forall k, i < k < j, A[k] > x)$ est vrai. $P(p-1,p)$ est vrai. Si $A[j] > x$, alors $P(i,j+1)$ est vrai puisque l'algorithme ne fait qu'incrémenter j de 1 ($P(i,j+1) = P(i,j) \wedge (A[j] > x)$. Sinon, si $A[j] \leq x$, $P(i+1,j+1)$ est vrai puisque l'algorithme échange $A[i]$ et $A[j]$ avant d'incrémenter i et j de 1 ($P(i+1,j+1) = P(i,j) \wedge (A[i] \leq x \wedge (A[j] > x))$.

Le coût du partitionnement est évidemment linéaire (il n'y a qu'une seule boucle de p à $r-1$).

Revenons sur l'évaluation du coût global du tri. Dans le cas le plus défavorable, le partitionnement est systématiquement déséquilibré et produit un sous-problème de taille $n-1$ et un de taille 0. Donc, $TR(n) = TR(n-1) + c.n$. Ce qui donne $TR(n) = c\sum_{k=2}^{n} k$, soit une complexité en $O(n^2)$.

Mais c'est une situation extrême (qui désavantage tout de même les tableaux déjà presque triés...). En pratique, on s'attend à des situations de déséquilibre éventuellement important, mais partageant tout de même le travail. Par exemple, considérons un déséquilibre systématique de 9 contre 1 : $TR(n) = TR(9n/10) + T(n/10) + c.n$, ce qui produit une complexité en $O(nlogn)$. On voit que le *log* est obtenu du moment qu'il y a partage, même si il n'est pas équitable. Un découpage ayant un facteur de proportionnalité constant engendre toujours un arbre récursif de profondeur de l'ordre de *logn*. Prenons l'exemple du découpage du nombre 900. Au premier niveau, on a 90, 810. Au deuxième,

9,81,81,729. Au troisième, 1,8,8,73,8,73,73,656. La branche la plus courte est de hauteur 3 ($log_{10}900$). La branche la plus longue a une hauteur de $log_{10/9}900$.

On peut penser a priori améliorer l'algorithme de partitionnement en choisissant le pivot aléatoirement plutôt que de prendre systématiquement et arbitrairement le dernier élément du tableau. L'algorithme se modifie ainsi :

```
Partition-aléatoire (A, p, r) :
        i := random(p,r) ;
        permuter A[r] et A[i] ;
        retourner Partition (A,p,r)
```

On attend que cela produise des partitions équilibrées en moyenne. Notons $D(n)$ le nombre moyen de comparaisons entre éléments de $A[1..n]$ durant l'algorithme de tri. Supposons que le premier pivot soit le i-ème élément de A. Le pivot est comparé à $n-1$ autres éléments durant l'algorithme de partitionnement qui partage le tableau en 2 sous-tableaux de tailles $i-1$ et $n-i$. Dans le cas d'un tirage aléatoire uniforme des valeurs de A, i prend les valeurs $1, 2, ..., n$ avec la même probabilité. En considérant $D(0) = D(1) = 0$, on a donc

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

Cette dernière équation permet d'écrire que :
$nD(n) - (n-1)D(n-1) =$
$= (n(n-1) + 2\sum_{i=1}^{n} D(i-1)) - ((n-1)(n-2) + 2\sum_{i=1}^{n-1} D(i-1))$
$= 2(n-1) + 2D(n-1)$
On en déduit

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

Ou encore

$$\frac{D(n)}{n+1} = \frac{4}{n+1} - \frac{2}{n} + \frac{D(n-1)}{n}$$

En posant $E(n) = \frac{D(n)-2}{n+1}$, on obtient la récurrence suivante :

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

On en déduit que $E(n) = E(0)+2(H_{n+1}-1) = 2H_{n+1}-4$ avec $H_n = \sum_{i=1}^{n} \frac{1}{i}$ le n-ème nombre harmonique. En retournant à la fonction D, on obtient

$$D(n) = 2(n+1)H_{n+1} - 4n - 2$$

.

Pour n grand, la formule d'Euler dit que H_n se comporte comme $O(logn)$. On peut donc affirmer que $D(n)$ est en $O(nlogn)$.

4.3 Les théorèmes du tri

Théorème 5 *Il faut* $\lceil log_2 n! \rceil$ *comparaisons au pire pour trier n éléments. La complexité correspondante est* $O(nlogn)$.

Théorème 6 *Il faut* $\lceil log_2 n! \rceil$ *comparaisons en moyenne pour trier n éléments. La complexité correspondante est* $O(nlogn)$.

La conséquence est qu'il n' y a pas d'espoir d'avoir un algorithme de tri en temps linéaire si il utilise des comparaisons.

Pour prouver le premier théorème, il faut trouver un minorant en nombre de comparaisons requises. Dans un tri par comparaison, on se sert uniquement de comparaisons pour obtenir des informations sur l'ordre de la suite d'entrée $a_1, ...a_n$ (pour simplifier on peut supposer qu'on n'utilise que des tests $a_i \leq a_j$). La suite des comparaisons peut être abstraite en un arbre de décision binaire (les nœuds sont les tests et les feuilles les permutations obtenues). Par exemple, dans le tri de 3 valeurs par insertion, l'arbre de décision est donné par la figure 4.1 suivante. Un nœud étiqueté $i : j$ signifie que l'on effectue la comparaison des éléments $A[i]$ et $A[j]$. Les feuilles indiquent l'ordre dans lequel les valeurs doivent être triées, c'est-à-dire la permutation utilisée pour y arriver.

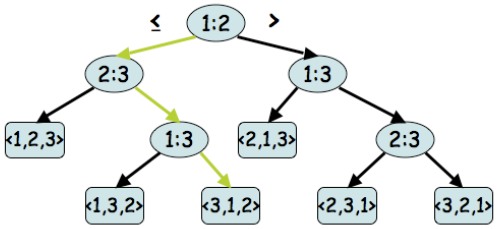

Figure 4.1: Arbre de décision pour le tri de 3 valeurs. La branche colorée est celle employée pour trier par exemple l'entrée $6, 8, 5$

Tout algorithme de tri correct (et général pour pouvoir traiter toutes les entrées possibles) doit être capable de produire toutes les permutations possibles de son entrée : une condition nécessaire est que chacune des n! permutations figure comme feuille de l'arbre de décision. La longueur du plus court chemin de la racine à une feuille donne le nombre de comparaisons effectuées par l'algorithme (on ne considère que les feuilles accessibles par l'algorithme) dans le cas le plus défavorable. La hauteur de l'arbre donne donc un minorant du nombre de comparaisons effectuées par un algorithme de tri. Considérons

un arbre de hauteur h avec l feuilles accessibles correspondant à un tri par comparaison de n éléments. On a $n! \leq l$. Puisqu'un arbre binaire de hauteur h n'a pas plus de 2^h feuilles, on a $n! \leq l \leq 2^h$, soit $log_2(n!) \leq h$. Par la formule de Stirling : $n!$ est approximativement égal à $\sqrt{2\pi n}\frac{n}{e}^n$. Le passage au logarithme donne donc une complexité en $O(nlogn)$.

4.4 Un tri linéaire

Pour concevoir un algorithme de tri linéaire, il va falloir renoncer à effectuer des comparaisons, et utiliser en fait des propriétés sur les nombres à trier. Considérons l'algorithme de tri par dénombrement, proposé par Seward en 1954.

On suppose que les valeurs sont des entiers dans l'intervalle $[0..k]$. Lorsque k est de l'ordre de n, le tri va s'effectuer en $O(n)$.

L'algorithme est le suivant :

```
Tri-denombrement (A,B,k) :
/* B donne le résultat, C[0..k] est intermédiaire */
 pour i de 0 à k
        faire C[i] := 0 ;
 pour j de 1 à n
        faire C[A[j]] := C[A[j]] + 1 ;
        /* C[i] contient le nb d'éléments égaux à i */
 pour i de 1 à k
        faire C[i] := C[i] + C[i-1] ;
        /* C[i] contient le nb d'éléments <= i */
 pour  j de n à 1
        faire B[C[A[j]]] := A[j] ;
              C[A[j]] := C[A[j]] - 1
```

La correction repose sur le fait que les éléments sont directement placés au bon endroit dans le tableau B, grace au calcul fait dans le tableau C permettant de réserver la place nécessaire. Un avantage de cet algorithme est sa stabilité définie comme étant le fait que les éléments égaux gardent leur ordre initial (certaines applications le demandent).

Son coût en nombre d'affectations est clairement linéaire : il y a des boucles simples sur k et sur n. Sa complexité temporelle est en $O(n + k)$. Si k reste de l'ordre de n, on obtient un tri linéaire en la taille du tableau à trier.

Comme exemple, si $A = [2, 5, 3, 0, 2, 3, 0, 3]$, la deuxième boucle produit $C = [2, 0, 2, 3, 0, 1]$. La troisième boucle produit $C = [2, 2, 4, 7, 7, 8]$. Lors de la dernière boucle, les éléments dans le tableau B sont placés dans l'ordre des cases $7, 2, 6, 4, 1, 5, 8, 3$.

4.5 Arbres binaires

Les tableaux sont des structures qui permettent un accès direct à un élément à partir de son indice. Par contre, l'insertion ou la suppression dans de telles

structures d'un élément à une position donnée sont des opérations coàteuses.

Les arbres binaires présentés ici sont en fait un bon compromis. Ils permettent un accès "relativement" rapide à un élément à partir de son identifiant, appelé ici une "clé". Dans les arbres binaires, l'ajout et la suppression sont également des opérations "relativement" rapides.

Par relativement rapide, on entend que le temps d'exécution d'une opération n'est pas proportionnel au nombre d'éléments dans la structure. En effet, dans un tableau non ordonné de 256 éléments par exemple, il faut parcourir au plus les 256 éléments pour trouver celui que l'on cherche. Comme on le verra par la suite, dans un arbre binaire de 256 éléments, il suffit de parcourir au plus 8 éléments pour trouver celui que l'on cherche.

Un arbre est constitué de nœuds. Un nœud a deux successeurs. Ceux-ci sont désignés comme étant le "fils gauche" et le "fils droit" du nœud. Un nœud est donc une structure qui contient un élément à stocker et deux pointeurs sur les noeuds fils.

```
structure NOEUD:
   ELEMENT elt;
   NOEUD * fg;
   NOEUD * fd;
fin structure;
```

On supposera que l'identifiant d'un élément, que l'on a appelé la clé, est un objet de type CLE. On disposera aussi de la fonction *cle*, $cle(e)$ retournant la clé de l'élément e. On supposera également qu'un arbre ne peut pas contenir deux éléments ayant la même clé. La clé est donc un moyen unique d'identifier et de localiser un élément dans un arbre. Un arbre sera toujours fait de telle sorte que les éléments dans le sous-arbre gauche d'un nœud ont une clé inférieure à celle du nœud. De même, les éléments dans le sous-arbre droit d'un nœud ont une clé supérieure à celle du nœud. Cela va faciliter la recherche d'un élément par rapport à sa clé. Le noeud au sommet de l'arbre est appelé "racine" et un arbre sera référencé simplement par un pointeur sur cette racine.

```
type ARBRE: NOEUD *;
```

4.5.1 Opérations sur la structure

Nous allons présenter ici quelques opérations classiques que l'on peut effectuer sur un arbre. On a choisi de les écrire de manière récursive et d'utiliser une notation proche du langage C en utilisant des pointeurs.

Initialiser un arbre : cette fonction initialise les valeurs de la structure représentant l'arbre pointé par a, afin que celui-ci soit vide. Il suffit de mettre le pointeur sur la racine égal à NIL.

```
fonction initialiser(ARBRE * a):
 *a :=  NIL;
fin fonction;
```

Préparer un noeud : cette fonction alloue un nouveau nœud et place l'élément e à l'intérieur. Ses deux fils sont initialisés à la valeur NIL. La fonction retourne l'adresse de ce nouveau nœud. Au cas oà l'allocation de mémoire échoue, NIL est renvoyé.

```
fonction NOEUD *  preparerNoeud(ELEMENT e):
 n  ALLOUER(NOEUD,1);
 si (n != NIL) alors
  n->elt :=  e;
  n->fg := NIL;
  n->fd := NIL;
 fin si;
 rendre n;
fin fonction;
```

Ajouter un noeud : cette fonction ajoute le nœud pointé par n dans l'arbre pointé par a. Pour cela, l'arbre est parcouru à partir de la racine pour descendre jusqu'à l'endroit où sera inséré le nœud. A chaque nœud, deux possibilités sont offertes pour descendre. On prendra à gauche si la clé du nœud visité est supérieure à la clé du nœud à insérer. A l'opposé, on prendra à droite si la clé du nœud visité est inférieure. En cas d'égalité, l'insertion ne peut pas se faire et la fonction retourne FAUX. On arrête la descente quand le fils gauche ou droit choisi pour descendre vaut NIL. Le noeud pointé par n est alors inséré à ce niveau.

```
fonction BOOLEEN  ajouterNoeud(ARBRE * a, NOEUD * n):
 si (*a = NIL) alors
  *a := n;
  rendre VRAI;
 fin si;
 si (cle(n->elt) = cle((*a)->elt)) alors rendre FAUX;
 si (cle(n->elt) < cle((*a)->elt)) alors
  rendre ajouterNoeud(&((*a)->fg),n);
 fin si;
 rendre ajouterNoeud(&((*a)->fd),n);
fin fonction;
```

Ajouter un élément : cette fonction ajoute l'élément e dans l'arbre pointé par a. Pour cela, un nœud est préparé puis ajouté dans l'arbre. Si le nœud ne peut pas être alloué ou si la clé de l'élément est déjà présente dans l'arbre, alors la valeur FAUX est retournée.

```
fonction BOOLEEN  ajouterElement(ARBRE * a, ELEMENT e):
 n := preparerNoeud(e);
 si (n = NIL) alors rendre FAUX;
```

```
 si (non ajouterNoeud(a,n)) alors
  LIBERER(n);
  rendre FAUX;
 fin si;
 rendre VRAI;
fin fonction;
```

Clé existe à : cette fonction cherche si un élément possède la clé c dans l'arbre a. Pour cela, l'arbre est parcouru à partir de la racine. Comme pour l'ajout d'un nœud, deux possibilités sont offertes quand on visite un nœud. Si on trouve l'élément qui a la clé c, alors VRAI est retourné. Sinon, on aboutit sur un fils (gauche ou droit) qui vaut NIL, ce qui signifie que la recherche est terminée et qu'aucun élément n'a la clé recherchée. FAUX est alors retourné.

```
BOOLEEN  existeCle(ARBRE a, CLE c):
 si (a = NIL) alors rendre FAUX;
 si (c = cle(a->elt)) alors rendre VRAI;
 si (c < cle(a->elt)) alors rendre existeCle(a->fg,c);
 rendre existeCle(a->fd,c);
fin fonction;
```

Extraire la clé maximum : cette fonction extrait le nœud qui a la plus grande clé dans l'arbre pointé par a. Pour cela, l'arbre est parcouru en descendant sur la droite tant que cela est possible. Le dernier nœud visité est celui recherché. Il est supprimé de l'arbre et son adresse est retournée. Au cas oà l'arbre est vide, la valeur NIL est retournée.

```
fonction NOEUD *  extraireMaximum(ARBRE * a):
 si (*a = NIL) alors rendre NIL;
 si ((*a)->fd = NIL) alors
  n := *a;
  *a := (*a)->fg;
  rendre n;
 fin si;
 rendre extraireMaximum(&((*a)->fd));
fin fonction;
```

Supprimer la racine : cette fonction supprime le nœud à la racine de l'arbre pointé par a. Quatre cas se présentent. Si l'arbre est vide, FAUX est retourné. Si la racine n'a pas de fils gauche, alors la racine est supprimée et le fils droit prend sa place. Si la racine n'a pas de fils droit, alors la racine est supprimée et le fils gauche prend sa place. Enfin, si la racine a deux fils, alors la racine est supprimée et le nœud ayant la plus grande clé dans le fils gauche prend sa place. Dans les trois derniers cas, VRAI est retourné.

```
fonction BOOLEEN  supprimerRacine(ARBRE * a):
```

```
 si (*a = NIL) alors rendre FAUX;
 n := *a;
 si (n->fg = NIL) alors *a := n->fd;
 sinon si (n->fd = NIL) alors *a := n->fg;
 sinon
  *a := extraireMaximum(&(nfg));
  (*a)->fg := n->fg;
  (*a)->fd := n->fd;
 fin si;
 LIBERER(n);
 rendre VRAI;
fin fonction;
```

Extraire un élément : cette fonction extrait l'élément ayant la clé c de l'arbre pointé par a. L'élément extrait est stocké à l'adresse e. A partir de la racine, on parcours l'arbre jusqu'à trouver la clé c. A ce moment, l'élément du nœud trouvé est stocké à l'adresse e et le nœud est supprimé en appliquant la fonction *supprimerRacine* sur le sous-arbre dont la racine est le nœud en question.

```
fonction BOOLEEN  extraireElement(  ARBRE * a, CLE c,
ELEMENT * e):
 si (*a = NIL) alors rendre FAUX;
 si (c < cle((*a)->elt)) alors
  rendre extraireElement(&((*a)->fg),c,e);
 fin si;
 si (c > cle((*a)->elt)) alors
  rendre extraireElement(&((*a)->fd),c,e);
 fin si;
 *e := (*a)->elt;
 rendre supprimerRacine(a);
fin fonction;
```

4.5.2 Equilibrage des arbres

Très facilement, on peut se rendre compte que les arbres binaires tels qu'ils ont été présentés jusqu'à présent ont un inconvénient majeur. En effet, les opérations d'ajout et de suppression ne garantissent pas un certain équilibre de l'arbre, c'est-à-dire qu'il se peut qu'il y ait beaucoup plus de nœuds d'un côté que de l'autre. Cela signifie que la recherche d'une clé d'un côté sera plus lente qu'une recherche de l'autre côté. On considérera par la suite qu'un arbre est équilibré si, pour chacun de ses noeuds, la différence entre la hauteur du sous-arbre gauche et la hauteur du sous-arbre droit est d'au plus une unité. Dans un premier temps, nous allons modifier la structure d'un nœud afin d'y intégrer des informations concernant l'équilibrage de l'arbre. Ensuite, nous présenterons des opérations appelées "rotations" qui permettent de rééquilibrer un arbre le cas

échéant. Enfin, nous verrons à quelle occasion employer ces rotations et comment modifier les opérations présentées précedemment pour qu'elles garantissent l'équilibre d'un arbre à tout moment.

Pour chaque nœud n, on rajoute un champ h qui indique la hauteur du sous-arbre de racine n, un arbre sans nœud ayant une hauteur égale à 0. Voici la nouvelle structure.

```
structure NOEUD:
 ELEMENT elt;
 NOEUD * fg;
 NOEUD * fd;
 ENTIER  h;
fin structure;
```

Concernant cette nouvelle information, nous présentons maintenant les deux fonctions suivantes de mise à jour de la hauteur et de la mesure du déséquilibre.

Mise à jour de la hauteur d'un noeud : cette fonction met à jour le champ h du nœud pointé par n. L'actualisation est faite en se basant sur les champs h des fils du nœud. Si un fils vaut NIL, alors on considérera sa hauteur comme étant nulle. La hauteur du nœud pointé par n est donc la plus grande des deux hauteurs des fils augmentée d'une unité. On utilise l'instruction MAX qui retourne la plus grande des deux valeurs passées en paramètre.

```
fonction majHauteur(NOEUD * n):
 si (n->fg = NIL) alors hg := 0;
 sinon hg := n->fg->h;
 si (n->fd = NIL) alors hd := 0;
 sinon hd := n->fd->h;
 n->h := MAX(hd,hg) + 1;
fin fonction;
```

Mesure du déséquilibre d'un arbre : cette fonction retourne une valeur qui mesure le déséquilibre de l'arbre a. En fait, il s'agit de la différence entre la hauteur du sous-arbre gauche et celle du sous-arbre droit. Si un fils vaut NIL, alors on considérera sa hauteur comme étant nulle.

```
fonction ENTIER  desequilibre(ARBRE a):
 si (a = NIL) alors rendre 0;
 si (a->fg = NIL) alors hg := 0;
 sinon hg := a->fg->h;
 si (a->fd = NIL) alors hd := 0;
 sinon hd  afdh;
 rendre (hg - hd);
fin fonction;
```

Les rotations : en supposant que tous les nœuds impliqués dans la rotation existent, voici la fonction qui effectue la rotation droite sans oublier de mettre à jour la hauteur des noeuds déplacés.

```
fonction rotationRD(ARBRE * a):
 n := *a;
 si (n->fg = NIL) alors /* Erreur */
 *a := n->fg;
 n->fg := (*a)->fd;
 (*a)->fd := n;
 majHauteur(n);
 majHauteur(*a);
fin fonction;
```

La figure 4.2 illustre son fonctionnement.

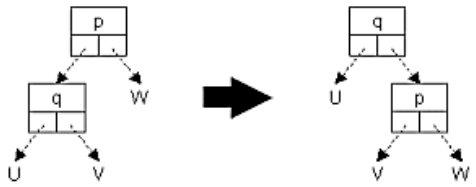

Figure 4.2: Rotation droite d'un arbre binaire.

La rotation à gauche est symétrique à la rotation à droite. Les notions de gauche et de droite sont simplement inversées.

On considère aussi une rotation double gauche-droite (ainsi que son symétrique droite-gauche). Il s'agit d'appliquer une rotation RG sur le fils gauche de la racine, puis d'appliquer une rotation RD sur la racine elle-même.

Opérations d'ajout et de suppression avec rééquilibrage : les opérations d'ajout et de suppression d'un élément présentées ici garantissent qu'un arbre reste toujours équilibré. L'idée principale est la suivante. On effectue tout d'abord l'opération d'ajout ou de suppression comme elle a été présentée précédemment. Seulement, on mémorise le chemin qui nous a permis de trouver la position de l'élément inséré ou supprimé. Ensuite, on remonte ce chemin jusqu'à la racine. A chaque nœud visité, on regarde s'il y a un déséquilibre de plus d'une unité. Si c'est le cas, un rééquilibrage s'impose en effectuant une rotation présentée précédemment.

Dans un premier temps, on présente la fonction de rééquilibrage qui effectue une rotation sur la racine d'un arbre dans le but de le rééquilibrer. Ensuite, on

détaille les modifications apportées aux opérations *, ajouterNoeud, extraireMaximum, extraireElement* pour que l'ajout et la suppression garantissent l'équilibre de l'arbre à tout moment.

Rééquilibrer un arbre : cette fonction rééquilibre l'arbre pointé par a en effectuant une rotation. Cette opération suppose que les sous-arbres de la racine sont équilibrés. Elle est exécutée après l'ajout ou la suppression d'un élément sur un arbre équilibré. Un rééquilibrage sera effectué si le déséquilibre vaut +2 ou -2. On détaille ici le cas oà le déséquilibre vaut +2, le cas -2 étant symétrique. Dans ce cas, il y a un déséquilibre à gauche. Trois possibilités se présentent alors : le déséquilibre du fils gauche vaut +1, 0 ou -1. Dans le premier cas, une rotation à droite rééquilibre l'arbre. La figure 4.3 illustre ce cas.

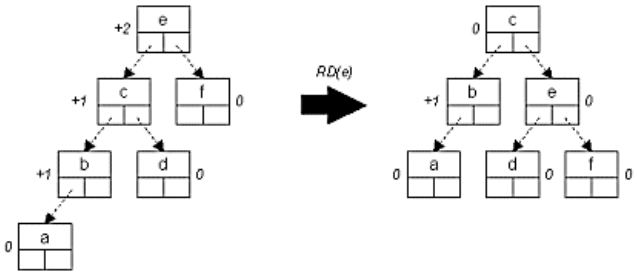

Figure 4.3: Rééquilibrage par rotation droite d'un arbre binaire.

Le deuxième cas ne peut pas se produire, car cela signifierait que l'arbre était déjà déséquilibré avant la suppression ou l'ajout d'un élément.

Dans le dernier cas, une rotation double gauche-droite rééquilibre l'arbre. La figure 4.4 illustre ce cas.

Voici donc le code de la fonction de rééquilibrage.

```
fonction reequilibrer(ARBRE * a):
 d := desequilibre(*a);
 si (d = +2) alors
  si (desequilibre((*a)->fg) = -1) alors
   rotationRGD(a);
  sinon
   rotationRD(a);
  fin si;
 sinon si (d = -2) alors
```

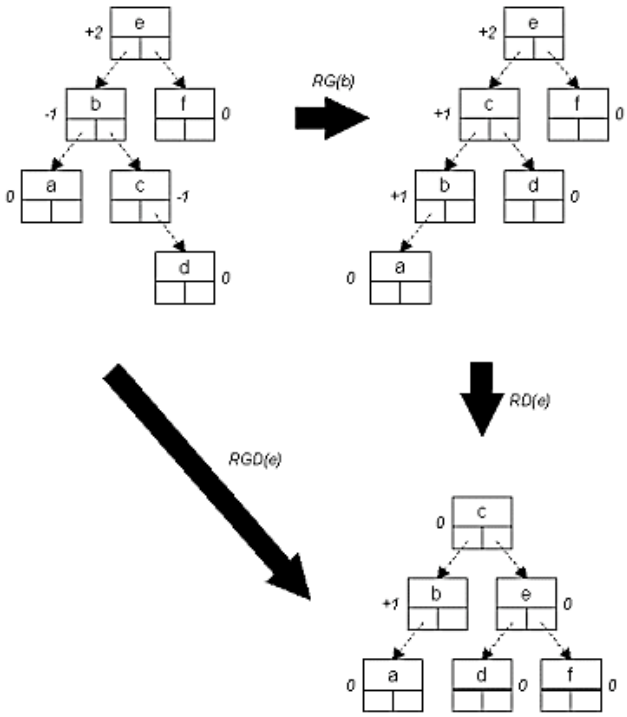

Figure 4.4: Rééquilibrage par rotation double gauche-droite d'un arbre binaire.

```
  si (desequilibre((*a)->fd) = +1) alors
   rotationRDG(a);
  sinon
   rotationRG(a);
  fin si;
 fin si;
fin fonction;
```

Ajouter un noeud (avec rééquilibrage) : les modifications apportées à cette fonction sont les suivantes. Tout d'abord, le nœud inséré se voit attribué une hauteur de 1. Ensuite, après chaque appel récursif à la fonction, la hauteur du nœud racine est mise à jour et une action de rééquilibrage est engagée sur ce même nœud. En effet, après chaque appel récursif, l'arbre est susceptible d'être déséquilibré puisqu'un élément y a été ajouté.

```
fonction BOOLEEN  ajouterNoeud(ARBRE * a, NOEUD * n):
 si (*a = NIL) alors
  *a := n;
  n->h := 1;
  rendre VRAI;
 fin si;
 si (cle(n->elt) = cle((*a)->elt)) alors rendre FAUX;
```

```
 si (cle(n->elt) < cle((*a)->elt)) alors
  ok := ajouterNoeud(&((*a)->fg),n);
 sinon
  ok := ajouterNoeud(&((*a)->fd),n);
 fin si;
 si (ok) alors
  majHauteur(*a);
  reequilibrer(a);
 fin si;
 rendre ok;
fin fonction;
```

Extraire la clé maximum (avec rééquilibrage) : les modifications apportées à cette fonction sont les suivantes. Après chaque appel récursif à la fonction, la hauteur du nœud racine est mise à jour et une action de rééquilibrage est engagée sur ce même nœud. En effet, après chaque appel récursif, l'arbre est susceptible d'être déséquilibré puisqu'un élément y a été supprimé.

```
fonction NOEUD *  extraireMaximum(ARBRE * a):
 si (*a = NIL) alors rendre NIL;
 si ((*a)->fd = NIL) alors
  n := *a;
  *a := (*a)->fg;
  rendre n;
 fin si;
 n  extraireMaximum(&((*a)fd));
 si (n != NIL) alors
  majHauteur(*a);
  reequilibrer(a);
 fin si;
 rendre n;
fin fonction;
```

Extraire un élément (avec rééquilibrage) : les modifications apportées à cette fonction sont les suivantes. Après chaque appel récursif à la fonction, la hauteur du nœud racine est mise à jour et une action de rééquilibrage est engagée sur ce même nœud. En effet, après chaque appel récursif, l'arbre est susceptible d'être déséquilibré puisqu'un élément y a été supprimé.

```
fonction BOOLEEN  extraireElement(  ARBRE * a, CLE c,
                                    ELEMENT * e):
 si (*a = NIL) alors rendre FAUX;
 si (c < cle((*a)->elt)) alors
  ok := extraireElement(&((*a)->fg),c,e);
 sinon si (c > cle((*a)->elt)) alors
  ok := extraireElement(&((*a)->fd),c,e);
```

```
 sinon
  *e := (*a)->elt;
  ok := supprimerRacine(a);
 fin si;
 si (ok et *a != NIL) alors
  majHauteur(*a);
  reequilibrer(a);
 fin si;
 rendre ok;
fin fonction;
```

Chapter 5

Graphes et fermeture transitive

5.1 Graphes et relations : les définitions de base

Définition 3 *Un graphe est constitué de sommets et d'arcs orientés (ou arêtes) : $G = (X, \Gamma)$ avec X un ensemble fini de sommets et $\Gamma \subseteq X^2$ la relation de liaison. On note n le cardinal de X.*

Lemme 1 *Si un graphe ne contient pas de sommets isolés,*

$$\lceil \frac{n}{2} \rceil \leq \mid \Gamma \mid \leq n^2$$

On peut considérer un graphe comme une relation binaire. Soit R une relation binaire sur X, rappelons que R est dite :

- Réflexive ssi $\forall x \in X, R(x, x)$,
- Symétrique ssi $\forall x, y \in X, R(x, y) \Rightarrow R(y, x)$,
- Transitive ssi $\forall x, y, z \in X, R(x, y)$ et $R(y, z) \Rightarrow R(x, z)$.

C'est une relation d'équivalence si elle vérifie les 3 points précédents.

La figure 5.1 à gauche montre un exemple de graphe.

Graphiquement, la réflexivité se traduit par des "bouclettes" sur les sommets. La symétrie veut dire qu'il n'y a pas d'orientation en fait. Une classe d'équivalence est un graphe complet (ou clique).

Définition 4 *On appelle* chemin *une suite $u = x_1..x_m$ de sommets tels que $\forall i, (x_i, x_{i+1}) \in \Gamma$.*

On peut concaténer les chemins : soit $u = x_1..x_m$ et $v = y_1..y_p$, $u.v = x_1..x_m y_1..y_p$. La *longueur* d'un chemin est son nombre d'arêtes : $\mid x_1..x_m \mid = m - 1$. Un chemin est *élémentaire* si les x_i sont distincts. Il contient un circuit sinon.

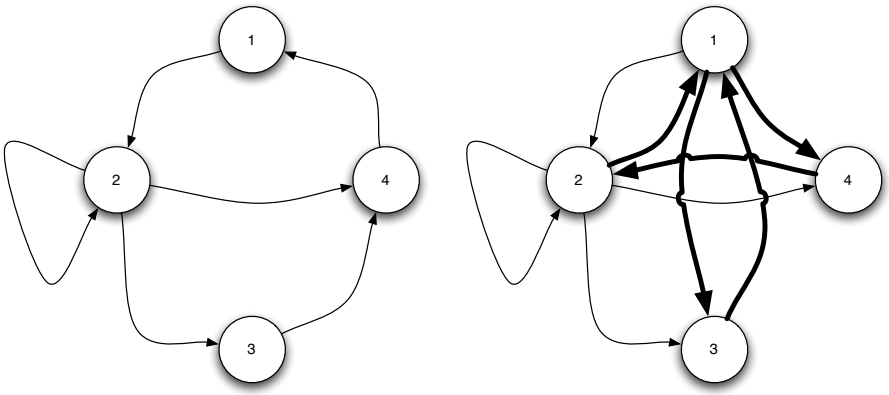

Figure 5.1: Un exemple de graphe G et son carré G^2.

Lemme 2 *La longueur d'un chemin élémentaire est au plus de* $|\Gamma|$ *et aussi au plus de* $n-1$.

5.2 Graphe des chemins

On peut définir des opérations sur les graphes ayant les mêmes sommets, comme l'union (on la notera aussi $+$) et l'itération (puissance). Soient $G_1 = (X, \Gamma_1)$ et $G_2 = (X, \Gamma_2)$, l'union est définie comme $G_1 \cup G_2 = (X, \Gamma_1 \cup \Gamma_2)$. Soit $G = (X, \Gamma)$, le graphe G^2 est le graphe (X, Γ') avec $\Gamma' = \{(x, z) \mid \exists y \in X, (x, y) \in \Gamma \wedge (y, z) \in \Gamma\}$.

Un graphe G peut être représenté par une matrice booléenne carrée de taille $n \times n$, avec $G_{ij} = 1$ ssi $(x_i, x_j) \in \Gamma$. L'union des graphes correspond à l'addition logique (ou). Sa complexité est en $O(n^2)$. La multiplication des graphes correspond à la multiplication logique (et) des matrices : $G_{ij} = \bigvee_{k=1}^{n} (G_{1ik} \wedge G_{2kj})$. La complexité du calcul du carré d'un graphe est en $O(n^3)$.

Un algorithme de calcul du carré est le suivant :

```
Gamma' = {} ;
pour chaque sommet x
  pour chaque sommet z
    pour chaque sommet y
      si Gamma(x,y) et Gamma(y,z)
         alors Gamma' := Gamma' + {(x,z)} ; (break)
```

L'instruction *break* permet de sortir brutalement de la boucle sur y. Cela ne modifie pas la complexité, mais évite de traiter de l'information redondante.

Un graphe G est transitif si sa relation Γ est transitive. C'est-à-dire si $\Gamma' \subseteq \Gamma$. Comme G et G^2 ont les mêmes sommets, on écrira $G^2 \subseteq G$.

La figure 5.1 à gauche montre le graphe carré G^2, du graphe de droite G, prouvant que G n'est pas transitif.

Le *graphe des chemins* est défini comme la fermeture du graphe.

Définition 5 *Soit G un graphe, on définit sa fermeture comme le graphe $G^+ = \{(x,y) \mid \exists$ un chemin $u_1u_2..u_m$ dans G avec $m > 1, u_1 = x, u_m = y\}$.*

G^+ est transitif.

Lemme 3 $G^+ = \bigcup_{p=1}^{n} G^p$

Preuve : $G^p = \{(x,y) \mid \exists u_1u_2..u_{p+1}, u_1 = x, u_{p+1} = y$ chemin de $G\}$. $\bigcup_{p=1}^{n} G^p = \{(x,y) \mid \exists m \in [2, n+1], u_1u_2..u_m, u_1 = x, u_m = y$ chemin de $G\}$. Pour n plus grand, il existe un circuit et le sommet sur lequel on reboucle a déjà été considéré dans un chemin plus court.

Un algorithme naïf pour calculer G^+ est le suivant :

```
G+ := {} ;
G' := G ;
tant que G+ <> G' faire
         G+ := G' ;
         G' := G' + G'.G
```

Sa complexité est en $O(n^4)$.

On peut faire plus astucieux. Soit B une matrice obtenue à partir de la matrice booléenne du graphe G en mettant 1 sur la diagonale (on rajoute systématiquement les bouclettes). Quelle informations nous donnent les puissances booléennes B^2, B^3, ..., B^k sur les chemins dans Gà Réponse : $B^k_{st} = 1$ ssi il existe dans G un chemin de s à t de longueur inférieure ou égale à k. La démonstration se fait aisément par induction sur k. On peut donc calculer la fermeture transitive en calculant les puissances B^2, B^4, B^8,... jusqu'à la plus petite puissance avec l'exposant strictement plus grand que n. Comme un chemin de longueur m ne peut pas contenir plus de $m+1$ sommets, on aura pas oublié des chemins. Comme on progresse de façon géométrique, le nombre de multiplications de matrices est de l'ordre $O(logn)$. La complexité du calcul de la fermeture devient alors $O(n^3logn)$.

5.3 Fermeture et multiplication : même combat

Le théorème suivant montre que la complexité du calcul de fermeture est dominée par la complexité du produit de matrices.

Théorème 7 *Le temps $T(n)$ nécessaire pour calculer la fermeture transitive d'une matrice booléenne X est du même ordre que le temps $M(n)$ nécessaire pour calculer le produit de deux matrices booléennes.*

Pour en faire la preuve, si n n'est pas une puissance de 2, on étend par X par la matrice $\overline{X} = \begin{pmatrix} A & 0 \\ 0 & 1 \end{pmatrix}$, où 1 est la matrice identité de la taille adéquate pour

obtenir un côté en puissance de 2. On divise alors la matrice en 4 parties égales $\begin{pmatrix} A & B \\ C & D \end{pmatrix}$. Cela divise le graphe en deux parties A et D, les sous-matrices B et C codant respectivement pour les arcs reliant des sommets de A vers des sommets de D et des sommets de D vers des sommets de A. Considérant la fermeture réflexo-transitive, on peut une forme générale de $\overline{X}^* = \begin{pmatrix} E & F \\ G & H \end{pmatrix}$ avec $E = (A + BD^*C)^*$, $F = EBD^*$, $G = D^*CE$, $H = D^* + D^*CEBD^*$.

Cette matrice est calculable par la séquence d'opérations suivante : $T_1 = D^*$; $T_2 = BT_1$; $E = (A + T_2C)^*$; $F = ET_2$; $T_3 = T_1C$; $G = T_3E$; $H = T_1 + GT_2$.

On a donc calculé 2 fermetures transitives, 6 multiplications et 2 additions sur des matrices de tailles moitié. Avec $n = 2^k$, et $M(m)$ le coàt de la multiplication de 2 matrices de taille $m \times m$, on a un coàt de calcul de la fermeture de $T(n) = 2T(\frac{n}{2}) + 6M(\frac{n}{2}) + \frac{n^2}{2}$. En sommant, on obtient $T(n) = 6\sum_{i=0}^{k-1} 2^i M(\frac{n}{2^{i+1}}) + \sum_{i=1} \frac{n^2}{2^{i+1}}$. Ou encore $T(n) = 6\sum_{i=0}^{k-1} 2^i M(\frac{n}{2^{i+1}}) + n - 1$. Le dernier pas est de constater que $\forall u, M(u) \geq 4M(\frac{u}{2})$ puisque l'on divise la matrice en 4. On peut en déduire que $\forall i, M(n) \geq 4^{i+1}M(\frac{n}{2^{i+1}})$ ou encore que $\forall i, 2^i M(\frac{n}{2^{i+1}}) \leq \frac{M(n)}{2^{i+2}}$. On en conclut que $T(n) \leq 6\sum_{i=0}^{k-1} \frac{M(n)}{2^{i+2}} + n - 1$. Or $\sum_{i=0}^{k-1} \frac{1}{2^{i+2}} = \frac{n-2}{2n}$. Donc $T(n) \leq \frac{3(n-2)}{n} M(n) + n - 1$. Puisque $O(M(n)) > O(n)$, $O(T(n)) = O(M(n))$. D'oà le théorème.

On en déduit donc que la complexité du calcul de la fermeture peut être ramenée en $O(n^3)$. La complexité du produit de deux matrices carrées à valeurs dans un corps est un sujet très étudié, avec de très nombreuses applications. Pour aller plus loin, on peut considérer l'approche de Strassen.

5.4 Algorithme de Strassen

En 1969, Volker Strassen imagine un algorithme permettant de descendre le nombre de multiplications en dessous de n^3. Pour se fixer les idées, prenons le produit de matrices 2x2 C=AB :

$$\begin{pmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix}$$

Cela nécessite 8 multiplications et 4 additions. L'idée de Volker Strassen est de trouver un moyen de multiplier les matrices A et B en utilisant seulement 7 multiplications au lieu de 8 par des combinaisons d'opérations astucieuses entre les éléments de A et B.

Posons:

$$\begin{aligned} E_1 &= (a_{12} - a_{22})(b_{21} + b_{22}) \\ E_2 &= (a_{11} - a_{22})(b_{11} + b_{22}) \\ E_3 &= (a_{11} - a_{21})(b_{11} + b_{12}) \\ E_4 &= (a_{11} + a_{21})b_{22} \\ E_5 &= a_{11}(b_{12} - b_{22}) \\ E_6 &= a_{22}(b_{21} - b_{11}) \\ E_7 &= (a_{21} + a_{22})b_{11} \end{aligned}$$

Les éléments de C s'obtiennent ainsi :

$$c_{11} = E_1 + E_2 - E_4 + E_6$$
$$c_{12} = E_4 + E_5$$
$$c_{21} = E_6 + E_7$$
$$c_{22} = E_2 - E_3 + E_5 - E_7$$

Cela fait bien 7 multiplications. L'économie d'une multiplication a toujours des conséquences. Dans notre cas, on se retrouve avec 18 additions. Pour appliquer la méthode aux matrices $n \times n$, on va utiliser la technique de la preuve du théorème précédent. On se ramène au cas d'une matrice avec n égal à une puissance de 2. On partitionne notre matrice en 4 sous-matrices afin de pouvoir appliquer les formules vues plus haut de manière récursive.

Cela donne le code suivant :

```
Fonction Strassen(A,B: matrices; N: entier): matrices;

si N n'est pas une puissance de 2
  Alors on ajoute des lignes et des colonnes de 0 afin d'accéder
        à la puissance de 2 la plus proche superieurement.
si N=1 Alors Strassen=A*B;
sinon
       On partitionne A et B en blocs de taille N/2;

       E1=Strassen(A11 - A22,B21 +  B22,N/2);
       E2=Strassen(A11 + A22,B11 +  B22,N/2);
       E3=Strassen(A11 - A21,B11 +  B12,N/2);
       E4=Strassen(A11 + A12,B22,N/2);
       E5=Strassen(A11,B12 - B22,N/2);
       E6=Strassen(A22,B21 - B11,N/2);
       E7=Strassen(A21 + A22,B11,N/2);

       C11=E1 + E2 - E4 + E6;
       C12=E4 + E5;
       C21=E6 + E7;
       C22=E2 - E3 + E5 - E7;
```

Finalement, la méthode de Strassen a un coàt en calcul de l'ordre de $O(n^{log_2 7}) = O(n^{2.81})$. Peut-on mieux faireà En 1970, S. Winograd a apporté une légère amélioration à l'algorithme de Strassen en réduisant le nombre d'additions à 15.

Donc 7 multiplications et 15 additions. Ce serait très intéressant de passer à 6 multiplications. Malheureusement, Winograd, Hopcroft et Kerr en 1971, ont achevé de montrer que l'on ne pouvait pas multiplier des matrices 2x2 avec moins de 7 multiplications. Dans le même registre technique, certains spécialistes explorent des pistes telles que l'amélioration des produits de matrices 3x3, 5x5, etc .... A titre indicatif on parvient à multiplier des matrices 3x3

avec 22 mutiplications. En 1980, S. Winograd et D. Coppersmith ont réussi à concevoir une méthode dont le coàt est de l'ordre de $O(n^{2.5})$. En 1987, ils parvinrent à descendre jusqu'à $O(n^{2.376})$. C'est à l'heure actuelle l'algorithme le plus rapide que l'on connaisse.

5.5 Algorithme de Roy-Warshall

Il s'agit de l'algorithme de fermeture le plus utilisé en pratique et de complexité cubique. L'idée est d'optimiser en exploitant la redondance induite par les transitivités qui apparaissent dans la construction des puissances (il peut y avoir plusieurs chemins de longueurs différentes entre deux sommets). Pour cela, on traite les sommets dans un ordre prédéfini donné par une numérotation arbitraire : $X = \{x_1, ..., x_n\}$. Soit $\Theta_p = \{(x, y) \mid \exists$ un chemin de x à y ne passant que par des sommets d'indice $\leq p\}$. Evidemment, $\Theta_n = G^+$ et $\Theta_0 = G$.

Lemme 4 $\Theta_{p+1} = \Theta_p \cup \{(x, y) \mid (x, x_{p+1}) \in \Theta_p \wedge (x_{p+1}, y) \in \Theta_p\}$

La preuve se fait par récurrence. Pour $p = 0$, le lemme est satisfait. Supposons qu'il soit vrai pour $p-1$. La seule façon de construire un chemin entre x et y incorporant un nouveau sommet x_{p+1} est d'avoir un arc entre un sommet x_k et x_{p+1} avec $k \leq p$, ainsi qu'un arc de retour entre x_{p+1} et un sommet x_l avec $l \leq p$. Donc (x, x_{p+1}) et (x_{p+1}, y) sont dans Θ_p.

On en déduit l'algorithme suivant :

```
Theta[0] = Gamma ;
pour p de 0 à n-1 faire
  Theta[p+1] = {} ;
  pour chaque sommet x
    pour chaque sommet y
      si (x,y) dans Theta[p] ou
         ((x,x[p+1]) dans Theta[p] et (x[p+1],y) dans Theta[p])
                        alors Theta[p+1] = Theta[p+1] + {(x,y)}
```

Pour économiser la mémoire, on peut calculer "sur place" pour obtenir l'algorithme final :

```
Algorithme (Roy-Warshall) :
  pour p de 0 à n-1 faire
    pour chaque sommet x
      pour chaque sommet y
         si (x,y) non dans Gamma et
            (x,x[p+1]) dans Gamma  et (x[p+1],y) dans Gamma
                alors Gamma = Gamma + {(x,y)}
```

Sa complexité est évidemment en $O(n^3)$.

La figure 5.2 montre un exemple de construction avec les arcs ajoutés à chaque valeur successive de p.

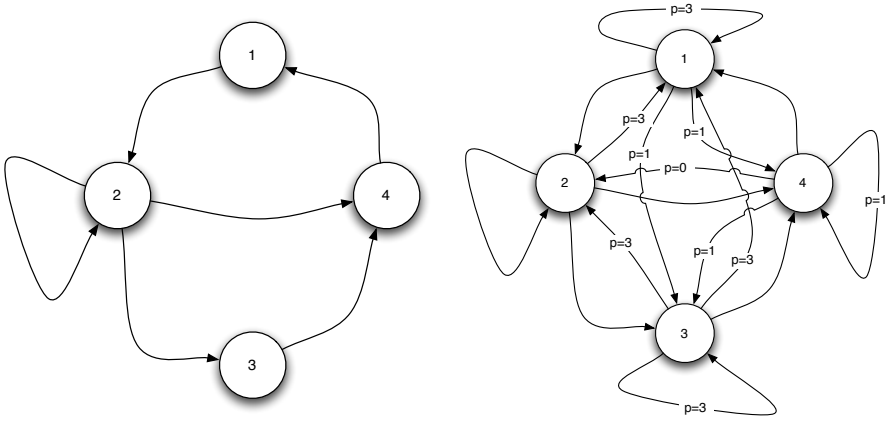

Figure 5.2: La construction de la fermeture par l'algorithme de Roy-Warshall.

5.6 Routage global

Le calcul de fermeture peut être aisément étendu pour calculer une fonction de routage permettant par exemple d'acheminer des messages d'un point à un autre lorsque le graphe est vu comme le modèle de la topologie d'une réseau de télécommunication. Le résultat du calcul est la suite des sommets à traverser pour aller d'un point à un autre.

```
Routage global :
 pour p de 0 à n-1 faire
    pour chaque sommet x
       pour chaque sommet y
         si (x,y) non dans Gamma et
            (x,x[p+1]) dans Gamma  et (x[p+1],y) dans Gamma
                alors Gamma := Gamma + {(x,y)} ;
                      u(x,y) := u(x,x[p+1]).u(x[p+1],y)
```

La table de routage obtenue pour le graphe G de la figure 5.1 est

$$\begin{pmatrix} u & 1 & 2 & 3 & 4 \\ 1 & 1241 & 12 & 123 & 124 \\ 2 & 241 & 22 & 23 & 24 \\ 3 & 341 & 3412 & 34123 & 34 \\ 4 & 41 & 412 & 4123 & 4124 \end{pmatrix}$$

Il faut bien remarquer que cet algorithme ne produit pas forcément le plus court chemin. Le résultat dépend en fait de la numérotation. Par exemple, en cas d'existence de chemins 1234 et 154, l'algorithme produit $u(1,4) = 1234$.

5.7 Routage local

Plutôt que le calcul des chemins globaux, on est souvent intéressé en pratique par le calcul de tables de routage locales, donnant pour chaque sommet (un routeur de l'Internet par exemple) le voisin sur lequel acheminer le message pour atteindre le but final. Celà répond à la préoccupation de "passage à l'échelle" du routage pour des grands systèmes.

Il s'agit là encore d'une modification simple de Roy-Warshall.

```
Algorithme (Routage local) :
  pour chaque sommet x
     pour chaque sommet y
        si (x,y) dans Gamma alors next(x,y) := y ;
  pour p de 0 à n-1 faire
     pour chaque sommet x
        pour chaque sommet y
           si (x,y) non dans Gamma et
              (x,x[p+1]) dans Gamma  et (x[p+1],y) dans Gamma
               alors Gamma := Gamma + {(x,y)} ;
                     next(x,y) := next(x,x[p+1])
```

La table de routage locale obtenue pour le graphe G de la figure 5.1 est

$$\begin{pmatrix} next & 1 & 2 & 3 & 4 \\ 1 & 2 & 2 & 2 & 2 \\ 2 & 4 & 2 & 3 & 4 \\ 3 & 4 & 4 & 4 & 4 \\ 4 & 1 & 1 & 1 & 1 \end{pmatrix}$$

Chaque ligne s constitue la table à implanter sur le routeur s.

Chapter 6

Parcours de graphes

6.1 Schéma générique d'exploration

Soit $G = (X, \Gamma)$ un graphe. On veut parcourir le plus vite possible les sommets de G qui sont atteignables à partir d'un sommet donné x_0.

Définition 6 *Le* cône *des sommets accessibles depuis x_0 est le sous-graphe de G défini par $G_0 = \{x \in X \mid \exists$ un chemin de x_0 à $x\}$, avec $x_0 \in G_0$.*

Un cône est fermé à droite : $\forall x \in X, (x \in G_0 \wedge (x, y) \in \Gamma) \Rightarrow y \in G_0$.
La figure 6.1 montre un exemple de cône.

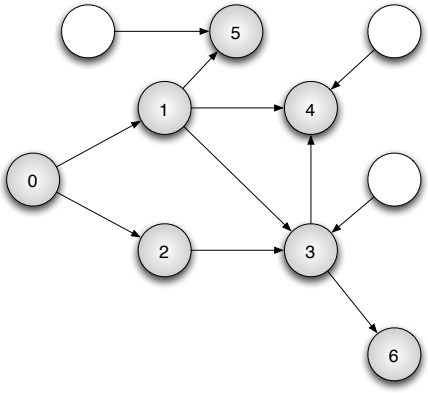

Figure 6.1: Le cône issu du sommet 0 pour un exemple de graphe (en grisé).

L'algorithme de calcul d'un cône est très simple :

```
Algorithme (générique) :
```

```
Initialement marquer x0
Règle : si x marqué et (x,y) dans Gamma et y non marqué
           alors marquer y
```

Il s'agit d'un schéma algorithmique non déterministe. L'ordre des applications de la règle est indifférent, l'important est que cela converge vers le résultat. On a donc un schéma algorithmique générique abstrait dont on peut quand même prouver la correction.

Le principe de la preuve est le suivant :

- Preuve de sécurité (il ne passe rien de mal) : si x est marqué alors $x \in G_0$. C'est vrai à l'étape initiale ($x_0 \in G_0$). Par récurrence, supposons qu'à l'étape $N, \forall x, x$ marqué $\Rightarrow x \in G_0$. L'étape $N+1$ est définie par une application de la règle. Puisque la règle s'est appliquée, elle a traité un y tel que $(x, y) \in \Gamma$. y devient alors marqué. Comme par hypothèse $x \in G_0$, alors $y \in G_0$.

- Preuve de vivacité (un jour, il se passe quelque chose de bien) : tous les sommets de G_0 finiront par être marqués par l'application successive de la règle. Sous l'hypothèse qu'une règle activable sera activée au bout d'un temps fini, l'algorithme se termine (au pire quand tous les sommets seront marqués, la règle n'est plus activable). Par contradiction, supposons $\exists x$ non marqué dans l'état final et $x \in G_0$. Il existe donc un chemin u de x_0 à x. Soit y le premier sommet non marqué de u. $y \neq x_0$. On peut donc considérer le sommet z prédécesseur de y dans u. On a alors $x \in G_0$ et $(z, y) \in \Gamma$ et y non marqué. La règle peut donc s'appliquer, ce qui contredit l'hypothèse d'état final.

6.2 Spécialisation par utilisation d'une structure de donnée

Il s'agit maintenant d'expliciter le choix du x, sommet à considérer dans la règle de l'algorithme générique. Pour celà, on va utiliser une structure de donnée qui va stocker une sorte de "frontière" définissant les prochains sommets à considérer dans le parcours.

Cette structured de donnée peut être aussi définie de façon abstraite par un "tas" accédé et modifié par les opérations suivantes :

- En entrée :
  - *Init* : crée le tas en mémoire et l'initialise vide ;
  - *Ajouter(x)*: met l'élément x dans le tas.
- En sortie :
  - *Observer* : rend un élément pris dans le tas (choisi de façon interne à la structure de donnée) ;

– *Extraire(x)* : enlève l'élément x du tas ;
– *Estvide* : rend vrai si le tas est vide.

On spécialise ainsi l'algorithme générique, dont la correction n'est pas mise en cause (il s'agit d'un raffinement) :

```
Algorithme (piloté par un tas) :
   Init ;
   Ajouter(x0) ; marquer x0 ;
   tantque non estvide faire
     x := Observer ;
     si il existe y tel que (x,y) dans Gamma et y non marqué
        alors ajouter(y) ; marquer y
        sinon extraire(x)
```

6.3 Représentation des structures de donnée

Regardons d'abord la façon de représenter le graphe G à parcourir en mémoire. On a vu la technique matricielle (dite celle de la matrice d'adjacence). Il s'agit d'une matrice booléenne avec $G[x, y] = 1$ si $(x, y) \in \Gamma$. L'espace mémoire est en $O(n^2)$, par contre la décision d'adjacence ($(x, y) \in \Gamma$) est en coàt temporel constant ($O(1)$). Si il y a peu d'arcs, les matrices sont "creuses".

Une deuxième technique possible est le codage par listes de successeurs. On peut le faire avec des tableaux : $(x, y) \in \Gamma$ ssi $\exists k \mid G[x].S[k] = y$. Le tableau S peut être représenté par une liste chaînée $y_1 \rightarrow y_2 \rightarrow ... \rightarrow y_n$ pour éviter des tableaux creux dans le cas oà le nombre de successeurs dans le graphe est très variable. L'espace mémoire utilise est en $O(n+ \mid \Gamma \mid)$. Du point de vue de la complexité globale, il s'agit d'un compromis souvent bon entre l'utilisation de la mémoire et l'utilisation du temps.

Etudions maintenant la question de la représentation du tas, en distinguant deux structures : la *file* et la *pile*.

Une file est un espace mémoire repéré par deux pointeurs : un début et une fin. Les éléments sont insérés par le pointeur de fin et extraits par le pointeur du début. Une mise en œuvre possible est l'utilisation d'un tableau circulaire $F[0..N-1]$ si on connaît une borne N à la taille de la file. Le principe de fonctionnement est "premier-entré", "premier sorti" (FIFO en anglais).

```
Init : début := 0 ; fin := 0 ;
Observer : F[début]
Ajouter(x) : F[fin] := x ;
             fin := fin+1 mod N
Extraire(x) : début := début+1 mod N
Estvide : fin=début
```

Toutes ces opérations se font en $O(1)$. Si on ne connaît pas une borne N, il y a la possibilité d'utiliser des listes dynamiques (avec une perte de temps liée à l'accès).

Une pile est une structure de donnée dont le principe de fonctionnement est inverse : le "dernier entré" sera le "premier sorti" (LIFO en anglais), comme une pile d'assiettes. Comme précédemment, on peut considérer un tableau circulaire. Un seul pointeur d'accès est suffisant en maintenant une variable donnant la hauteur de la pile.

```
Init : hauteur := 0 ;
Observer : P[hauteur-1]
Ajouter(x) : P[hauteur] := x ;
             hauteur := hauteur+1
Extraire(x) : hauteur := hauteur-1
Estvide : hauteur=0
```

Les opérations s'effectuent toujours en 0(1) et l'utilisation de listes dynamiques est aussi possible bien sàr.

6.4 Parcours en largeur d'abord

Il s'agit du parcours piloté par une file. La numérotation des sommets du cône de la figure 6.1 est obtenue par son parcours en largeur. La file contient successivement les sommets 0, 01, 012, 12, 123, 123, 123, 234, 345, 345, 456, 56, 6.

L'algorithme donne en prime le prédécesseur de chaque sommet sur un plus court chemin issu de x_0 (l'information est dans la file). La taille de la file est de l'ordre de la largeur du graphe (son degré maximum).

La performance de l'algorithme peut être optimisée. On limite le nombre de boucles dynamiques "tantque" en indiquant "en dur" le traitement de tous les successeurs de x (qui seront bien ceux sélectionnés par la structure en file). On obtient l'algorithme suivant :

```
Algorithme (parcours en largeur d'abord) :
  Init ;
  Ajouter(x0) ; marquer x0 ;
  tantque non vide faire
     x := Observer ;
     Extraire(x) ;
     pour chaque sommet y tel que (x,y) dans Gamma et y non marqué
        Ajouter(y) ; marquer y
```

6.5 Parcours en profondeur d'abord

Il s'agit du parcours piloté par une pile. Ajouter un sommet est l'empiler ; l'extraire est dépiler ; l'observer consiste à regarder le sommet de pile. Dans la figure 6.2, l'ordre de parcours des sommets (leur numérotation par un parcours en profondeur) est indiqué entre parenthèses.

La taille de la file est de l'ordre de la profondeur du graphe (son diamètre). C'est intéressant pour les graphes denses, et permet surtot de détecter les cycles.

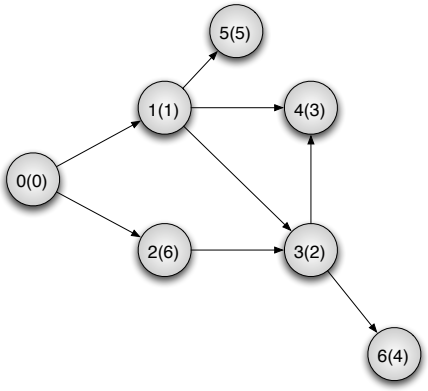

Figure 6.2: Numérotation par un parcours en profondeur.

L'évolution de la pile sur cet exemple est : 0, 01, 013, 0124, 013, 0135, 013, 01, 0, 02, 0.

La notion de pile apparaît aussi pour exécuter des fonctions récursives. Il n'est donc pas étonnant d'avoir une version récursive très simple du parcours en profondeur. L'empilement est mis en œuvre par l'appel de la fonction, le dépilement est son retour, le sommet de pile est passé en paramètre. La pile utilisée est donc implicite : il s'agit de la pile de récursion générée à la compilation.

```
Algorithme (parcours en profondeur d'abord) : explorer (x0)

  Explorer (x) :
        marquer x ;
        tantque il existe y tel que (x,y) dans Gamma et y non marqué
            Explorer (y)
```

Une utilisation fréquente du parcours en profondeur est le *tri topologique* d'un graphe sans circuit.

Un graphe est dit sans circuit si sa fermeture transitive est une relation d'ordre partielle sur l'ensemble des sommets. Toute relation d'ordre partielle admet une extension linéaire, c'est-à-dire une relation d'ordre total qui préserve l'ordre partiel. Effectuer un tri topologique sur un graphe sans circuit consiste à trouver une extension linéaire.

Par exemple, le graphe de la figure 6.3 possède 1234567 comme extension linéaire.

Considérons le graphe dual obtenu en renversant les arcs. On peut réaliser un tri topologique en effectuant un parcours en profondeur partant successivement

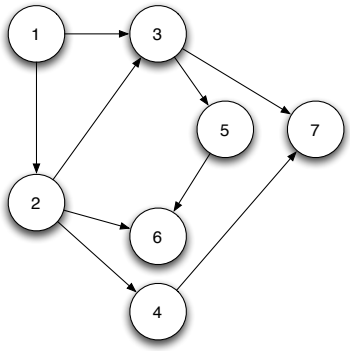

Figure 6.3: Un graphe sans circuit ayant 1234567 comme extension linéaire.

des sommets sur lequel aucun arc n'aboutit. On numérote les sommets au moment oà ils sont dépilés. Il suffit alors de concaténer les séquences obtenues. Dans l'exemple de la figure 6.3, on peut partir du sommet 7 pour obtenir 12347 puis 56 qui produit 1234756 comme extension linéaire.

6.6 Composantes fortement connexes

Définissons d'abord la co-accessibilité comme étant la possibilité de passer d'un sommet à un autre dans le graphe. Comment construire à partir d'un graphe donné, un squelette acyclique en regroupant dans des macro-sommets les sommets co-accessibles à La construction d'un tel graphe a l'avantage de séparer les questions dues aux cycles et celles liées à la structure acyclique.

Définition 7 *$x \, co \, y$ ssi $(x = y)$ ou $((x, y) \in \Gamma^+ \wedge (y, x) \in \Gamma^+)$.*

co est une relation d'équivalence. Les *composantes fortement connexes* (CFC) sont les classes d'équivalence de *co* (graphe quotient). Un exemple est donnée dans la figure 6.4.

Le graphe réduit $G_{co} = (X_{co}, \Gamma_{co})$ est donc défini par $(u, v) \in \Gamma_{co}$ ssi $u \neq v \wedge \exists x \in u, \exists y \in v, (x, y) \in \Gamma \wedge \neg(x \, co \, y)$.

Lemme 5 *G_{co} est acyclique.*

Preuve : Supposons qu'il soit cyclique. C'est-à-dire que $\exists\{u_1, ..., u_n\} \in X_{co}$ avec $u_n = u_1$ et $\forall i \in [1..n-1], (u_i, u_{i+1}) \in \Gamma_{co}$. Donc $\forall i \in [1..n-1], \exists x_i, y_i \in u_i, (y_i, x_{i+1}) \in \Gamma \wedge (y_{n-1}, x_1) \in \Gamma$. Mais $(x_i, y_i) \in \Gamma^+$ ou $x_i = y_i$, donc il existe un chemin de G passant successivement par $x_1 y_1 x_2 ... y_{n-1} x_1$. Donc $u_1 = u_2 = ... = u_n$ et il n'y a pas de cycle.

On en déduit l'algorithme de Foulk, qui pour chaque sommet x calcule sa CFC :

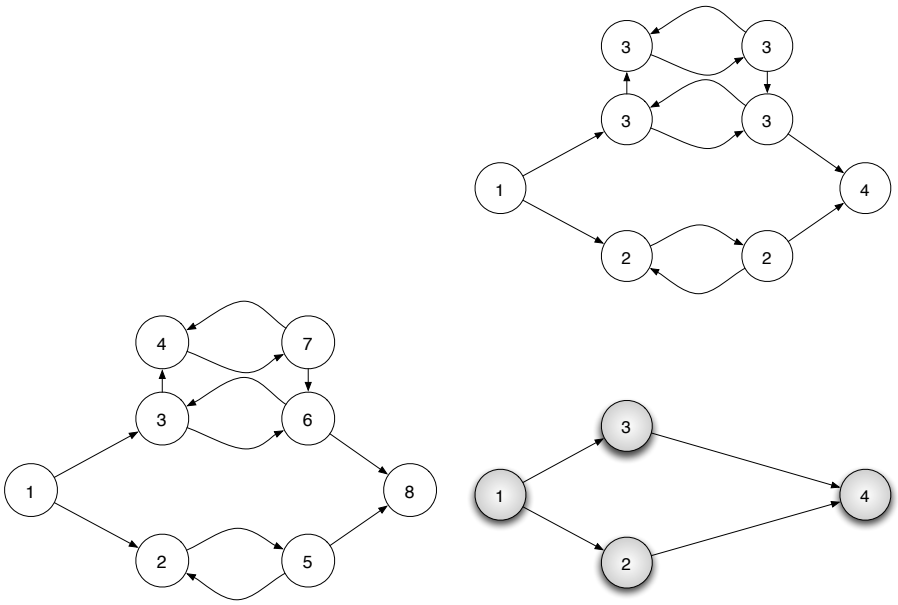

Figure 6.4: Un graphe avec des circuits et son graphe des CFC.

```
Algorithme de Foulk (calcul naïf des CFC) :
construire G+ ;
pour chaque sommet x
   C(x) = {x} ;
   pour chaque sommet y <> x
      si (x,y) et (y,x) dans Gamma+
         alors C(x) := C(x) + {y} ;
```

Cet algorithme a la complexité du calcul de la fermeture.

On peut améliorer la situation avec l'idée suivante : partant d'un sommet x, trouver l'ensemble des sommets accessibles, puis l'ensemble des sommets à partir desquels x est accessible. La CFC sera l'intersection des ensembles obtenus.

On obtient alors l'algorithme suivant :

```
Algorithme (de montée-descente) :
procedure descente(x) = marquer x avec D ;
          pour tout y tel que (x,y) dans Gamma,
                    y non marqué D et C(y) = {} faire
                           descente(y) ;
procedure montée(x) = marquer x avec M ;
          pour tout y tel que (y,x) dans Gamma,
                    y non marqué M et C(y) = {} faire
                         montée(y) ;
```

```
pour chaque sommet x
  si C(x) = {} faire
     C(x) = {x} ;
     effacer les marques ;
     montée(x) ;
     descente(x) ;
     pour chaque sommet y marqué D et M faire
        C(x) := C(x) + {y} ;
     pour tout sommet y dans C(x) faire
        C(y) := C(x) ;
```

Cet algorithme reste en $O(n^3)$ dans le cas pire oà la montée ou la descente sont en $O(n^2)$. Il se comporte en pratique en $O(n^2)$. C'est le gain obtenu en se passant du calcul préliminaire de la fermeture.

Pour aller plus loin, on va montrer comment détecter les cycles dans un parcours en profondeur d'abord.

6.7 Détection de cycles par un parcours : algorithme de Tarjan

On reprend l'algorithme de parcours en profondeur d'un graphe orienté en marquant aussi les arcs pour être sàr de les emprunter tous :

```
Algorithme (de détection de circuits accessibles à partir de z) :
   empiler z ;
   marquer z ;
   tantque pile non vide faire
       x := sommetpile ;
       si il existe y tel que (x,y) dans Gamma et (x,y) non marqué alors
          marquer (x,y) ;
          si y dans la pile alors circuit
             sinon si y non marqué alors
                      marquer y ;
                      empiler y ;
       sinon dépiler x ;
```

La terminaison est assurée par les faits suivants :

- Si x est dépilé, tous les y successeurs sont marqués ainsi que tous les arcs sortants.
- Chaque arc est marqué au plus une fois.
- Chaque sommet est dépilé au plus une fois.
- A chaque itération, on marque un arc ou on dépile un sommet.

Il n'y a donc qu'un nombre fini d'itérations.

La détection de cycle repose sur le fait que la pile contient le chemin courant depuis z.

Sur notre exemple de graphe de la figure 6.4, l'algorithme, exécuté à partir du sommet 1 produit les cycles 25, 47, 3476 et 36. Il produit les cycles mais pas encore les CFC qui seront des unions de cycles.

Le parcours en profondeur peut être étendu en mettant dans la pile, non plus les sommets, mais les CFC, afin d'obtenir le célèbre algorithme de Tarjan.

```
Algorithme (calcul des CFC par Tarjan) :
  pour tout sommet x, C(x) := {x} ;
  pour tout sommet x
     si C(x) non marqué alors
        init ;
        empiler C(x) ;
        marquer C(x) ;
        tantque pile non vide faire
            A := sommetpile ;
            si il existe x dans A, y dans B /* B est un C(z) ou
                                              est dans la pile */
               et (x,y) dans Gamma non marqué alors
               marquer (x,y) ;
                si B dans la pile alors
                  fusionner A et B dans la pile ; /* y compris
                                          les intermédiaires */
                sinon si B non marqué alors
                           marquer B ;
                           empiler B ;
            sinon dépiler A ;
                  numéroter A;
```

Sur l'exemple de la figure 6.4, cet algorithme produit les contenus de pile suivants :

```
{1}
{1}{2}
{1}{2}{5} (fusion)
{1}{2,5}
{1}{2,5}{8}
CFC(1) = {8}
CFC(2) = {2,5}
{1}
{1}{3}
{1}{3}{4}
{1}{3}{4}{7} (fusion)
{1}{3}{4,7}
{1}{3}{4,7}{6}
```

```
{1}{3}{4,7}{6} (fusion)
{1}{3,4,7,6}
{1}{3,4,7,6}
CFC(3) = {3,4,7,6}
CFC(4) = {1}
```

Les macro-sommets A dépilés lors de l'algorithme forment une partition. En effet tous les sommets seront considérés ($C(x)$ non marqué initialement). Cette partition initiale est seulement modifiée par les opérations de fusion qui préservent le fait que l'ensemble des sommets est partitionné.

Deuxièmement, considérons un ensemble A dépilé et deux sommets x et y dans A. Alors il existe un chemin de x à y puisqu'ils se sont retrouvés dans A suite une fusion d'ensembles se trouvant de façon contig¸e dans la pile. Si A et B sont dans la pile et A sous B, $\exists x, y \in A \times B, (x, y) \in \Gamma$.

La terminaison est assurée par le fait qu'à chaque itération, on marque un arc ou on dépile un macro-sommet. On ne marque pas deux fois un arc et un macro-sommet dépilé n'est pas réempilé.

Chapter 7

Algorithmes gloutons

On appelle algorithme glouton un algorithme qui suit le principe de faire, étape par étape, un choix optimum local, dans l'espoir d'obtenir un résultat optimum global. Par exemple, dans le problème du rendu de monnaie (donner une somme avec le moins possible de pièces), l'algorithme consistant à répéter le choix de la pièce de plus grande valeur qui ne dépasse pas la somme restante est un algorithme glouton. Dans les cas oà l'algorithme ne fournit pas systématiquement la solution optimale, il est appelé une heuristique gloutonne.

7.1 Algorithme de Dijkstra

On considère des graphes dont les arcs sont valués. La valeur d'un chemin est la somme des valuations rencontrées. La question est de trouver un chemin de valeur minimale (on dira optimale) entre deux sommets. Dans le cas oà les valuations sont toutes strictement positives, il s'agira de trouver les chemins les plus courts entre deux sommets (dans l'objectif de routage par exemple).

Prenons par exemple le graphe de la figure 7.1.

La question est de produire des chemins les plus courts pour aller du sommet 0 à un autre sommet du graphe. Pour ce problème simple, l'algorithme de Dijkstra est le suivant :

```
Algorithme (de Dijkstra) :

  l(0) := 0 ;
  pour chaque sommet x <> 0  l(x) := infini ;
  tantque il existe x non marqué et l(x) <> infini faire
      choisir un tel x avec l(x) minimum ;
      marquer x ;
      pour tout arc (x,y)
         si l(y) > l(x) + val(x,y) alors
            l(y) := l(x) + val(x,y) ;
            pred(y) := x ;
```

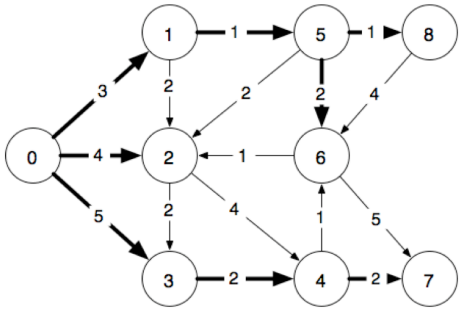

Figure 7.1: Un graphe valué.

Sa complexité est évidemment en $O(nd)$ où d est le degré maximum du graphe. La preuve de correction de l'algorithme repose sur les faits suivants :

- Tout sommet x tel que $l(x) \neq \infty$ est accessible depuis le sommet 0. En effet, si x est marqué, alors $l(x) \neq \infty$. Et si x est marqué et $(x, y) \in \Gamma$ alors $l(y) \neq \infty$.
- A chaque itération, au moins un sommet de plus est marqué. La conséquence est que l'algorithme termine.
- A la terminaison, x est marqué si et seulement si $l(x) \neq \infty$. La conséquence est qu'à la terminaison, x est accessible depuis le sommet 0 ssi x est marqué.

Il reste à montrer que les chemins calculés sont minimaux. L'algorithme est "glouton" car on avance en prenant un prochain sommet x avec $l(x)$ minimum. Il est optimal car au moment du marquage du sommet x, $l(x) \leq val(u)$, où u est un chemin quelconque de 0 à x et $val(u)$ sa valeur.

Cette propriété se montre par récurrence. Elle est vraie au premier marquage (le marquage du sommet 0). On suppose qu'elle est vraie pour les k premières itérations. On regarde le sommet x marqué à l'itération $k+1$. Montrons que $l(x)$ est minimal. Soit u un chemin de 0 à x, soit z' le premier sommet non marqué de u et z son prédécesseur sur u. Appelons u_1 le chemin de 0 à z et u_2 le chemin de z' à x. z est marqué par définition et a été marqué à une itération précédente, donc par récurrence $l(z) \leq val(u_1)$. Quand z a été marqué, $l(z')$ a été ajusté si nécessaire : $l(z') \leq l(z) + val(z, z')$. Donc $l(z') \leq val(u_1) + val(z, z')$. Comme $v(u_2) \geq 0$, $l(z') \leq val(u_1) + val(z, z') + val(u_2) = val(u)$. Mais juste avant d'être marqué, $l(x)$ était minimal parmi les sommets non marqués à distance finie. Donc $l(x) \leq l(z')$. D'où le résultat. $l(x)$ n'est plus modifié après marquage.

7.2 Exemple du gymnase

On considère un gymnase dans lequel se déroulent de nombreuses épreuves : on souhaite en "caser" le plus possible, sachant que deux événements ne peuvent avoir lieu en même temps (il n'y a qu'un gymnase). Un événement i est caractérisé par une date de début d_i et une date de fin f_i. On dit que deux événements sont compatibles si leurs intervalles de temps sont disjoints. On veut résoudre ce problème (optimiser le remplissage) à l'aide d'un programme glouton.

Un premier essai consiste à trier les événements par durée puis on gloutonne (i.e. on met les plus courts en premier s'ils sont compatibles avec ceux déjà placés). Cela ne conduit pas forcément optimal : imaginer les événements $j = [1, 5], i = [4, 7], k = [6, 10]$. La stratégie gloutonne placera i en premier, empêchant alors de placer j et k qui sont pourtant compatibles entre eux.

On aurait pu aussi classer les événements par date de début. Ce n'est pas non plus optimal puisqu'un événement urgent long peut empêcher une multitude d'événements courts à suivre.

La bonne intuition est de trier les événements par date de fin croissante. On peut alors prouver l'optimalité de gloutonnerie.

Soit f_1 l'élément finissant le plus tôt. On va montrer qu'il existe une solution optimale contenant cet événement. Soit donc une solution optimale arbitraire $O = f_{i1}..f_{ik}$, avec k le nombre maximum d'événements pouvant avoir lieu dans le gymnase. Si $f_{i1} = f_1$, c'est démontré. Sinon on remplace f_{i1} par f_1. Comme f_1 finit avant tout autre événement et que f_{i2} était compatible avec f_{i1}, alors f_{i2} est compatible avec f_1. La solution avec f_1 au début est donc aussi optimale. Après avoir plaçé f_1, on ne considère que les événements compatibles avec f_1 et on réitère la procédure.

7.3 Coloriage d'un graphe

Soit un graphe $G = (X, \Gamma)$ non orienté. On veut colorier les sommets, mais deux sommets reliés par une arête ne doivent pas avoir la même couleur. Le but est de minimiser le nombre de couleurs utilisé.

Théorème 8 *Un graphe est coloriable avec 2 couleurs si et seulement ses cycles sont de longueur paire.*

Montrons d'abord que si il est coloriable, ses cycles sont de longueur paire. Supposons que G possède un cycle $s_1..s_{2k+1}$ et qu'il soit 2-coloriable. Si s_1 est bleu, alors s_2 est rouge. Par récurrence, les sommets impairs sont bleus et les sommets pairs rouges. Mais s_1 est à côté de s_{2k+1}, d'oà la contradiction.

Dans l'autre sens, supposons que tous les cycles soient de longueur paire. On suppose que G est connexe (sinon on résoud indépendamment les composantes). On parcours G en largeur. Soit $x_0 \in X, X_0 = \{x_0\}$ et $X_{n+1} = \bigcup_{y \in X_n} \{$ fils de y dans l'ordre du parcours $\}$. Si il existe k, m et z tel que $z \in X_{2k} \cap X_{2m+1}$, alors z est à la distance $2k$ de x_0 et à la distance $2m + 1$ de x_0. Le cycle

correspondant est de longueur $2(m + k) + 1$ impaire, d'oà la contradiction. Donc $\forall k, m \quad X_{2k} \cap X_{2m+1} = \emptyset$. On colorie en bleu les éléments de X_i avec i impair et en rouge les éléments de X_i avec i pair. Il ne peut pas y avoir deux sommets reliés, l'un dans X_i, l'autre dans X_j avec i et j de même parité, sinon, on remontant à x_0, on formerait un cycle de longueur $i + j + 1$ impaire. On a donc colorié le graphe avec deux couleurs seulement.

Un graphe 2-coloriable est dit "biparti" car on peut partitionné le graphe en deux parties avec toutes les arêtes allant d'un ensemble à l'autre.

Pour colorier un graphe général, on peut gloutonner de la façon suivante :

- on considère les sommets dans un ordre arbitraire,
- on leur attribue la plus petite valeur possible, i.e. la plus petite valeur qui n'a pas déjà été attribuée à un voisin.

Le nombre de couleurs utilisées est bornée par le degré maximum du graphe (+1). La borne est atteinte pour le cas pire : la clique.

7.4 Algorithme de Brelaz

On en arrive à l'algorithme glouton de Brelaz qui colorie en priorité les sommets ayant beaucoup de voisins déjà coloriés. On définit le degré-couleur d'un sommet comme le nombre de ses voisins déjà coloriés. On initialise à 0 le degré-couleur des sommets du graphe puis on gloutonne en prenant parmi les sommets de degré-couleur maximal, un sommet de degré maximal et en lui attribuant la plus petite valeur possible. Il s'agit donc d'un glouton sur deux caractéristiques, le degré-couleur et le degré.

Prenons le graphe de la figure 7.2.

Théorème 9 *L'algorithme de Brelaz est optimal sur les graphes bipartis.*

Ce théorème signifie que l'algorithme de Brelaz réussit toujours à colorier les graphes bipartis en deux couleurs. Remarquons d'abord que le glouton ordinaire peut se tromper. Prenons la couronne de taille 3 $\{(b1, r1), (b2, r2), (b3, r3), (b1, r2), (b2, r3), (b3, r1)\}$. Si on commence par colorier $b1$ et $r3$ avec la même couleur, on sera obligé d'introduire une troisième couleur. L'algorithme de Brelaz, lui, ne se trompe pas. En effet, il ne coloriera que des sommets de degré-couleur égal à 1.

Pour terminer, il faut savoir que le problème général du coloriage des graphes est NP-complet. C'est-à-dire qu'il est dans la classe de complexité NP, mais aussi qu'il est difficile, dans le sens où il est au moins aussi difficile que tous les problèmes de la classe NP. D'où l'intérêt de se limiter à des heuristiques, gloutonnes par exemple...

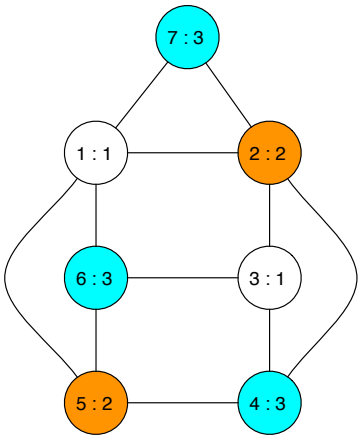

Figure 7.2: Un graphe non orienté 3-colorié par l'algorithme de Brelaz, en partant du sommet 1.

Chapter 8

Programmation dynamique

Dans de nombreux problèmes d'optimisation (ce que l'on appelle la "recherche opérationnelle"), la seule solution algorithmique connue consiste à énumérer l'espace des solutions. Cela se fait en général de façon incrémentale, la question essentielle étant de respecter les dépendances causales entre les solutions.

8.1 Pièces de monnaie

Il s'agit d'un exemple introductif. On dispose de pièces en nombre illimité, de valeurs $\{v_i\}$. Etant donné une somme S, on veut la réaliser avec un nombre minimum de pièces. Il faut bien sûr qu'il existe un indice i tel que $v_i = 1$ pour pouvoir réaliser n'importe quelle somme.

8.1.1 Essai glouton

La première idée est de "gloutonner". On trie les tas de pièces par ordre décroissant de valeur. Pour chaque valeur (dans cet ordre), on prend le maximum de pièces possibles. Plus formellement, soit R la somme restante, initialisée à S. Pour chaque valeur v_i, prendre $c_i = \lfloor R/v_i \rfloor$ pièces et calculer $R := R - c_i.v_i$ jusqu'à $R = 0$.

Théorème 10 *Cet algorithme est optimal pour le jeu de pièces* $\{10, 5, 2, 1\}$.

Preuve : considérons une somme S quelconque. On a au plus une pièce de 5, sinon il suffit de prendre une pièce de 10 pour obtenir une meilleure solution. On a aussi au plus une pièce de 1, sinon une pièce de 2 améliore la solution. On a au plus deux pièces de 2, sinon une pièce de 5 et une pièce de 1 suffisent. Puisqu'on ne prendra pas une solution avec une pièce de 5, une pièce de 1 et deux pièces de 2 (dans ce cas il vaut mieux prendre une pièce de 10), on a au plus trois pièces qui ne sont pas des pièces de 10 dans une solution optimale. Leur somme est forcément inférieure ou égale à 9. Le reste de la somme S est donc formé avec des pièces de 10, soit $\lfloor S/10 \rfloor$ pièces de 10, ce que calcule bien l'algorithme

glouton. R vaut alors S mod 10. Pour conclure, il suffit de voir les coefficients optimaux pour les sommes inférieures à 9 et de constater que ce sont bien ceux calculés par l'algorithme. En effet $1 = 1*1, 2 = 1*2, 3 = 1*2+1*1, 4 = 2*2, 5 = 1*5, 6 = 1*5+1*1, 7 = 1*5+1*2, 8 = 1*5+1*2+1*1, 9 = 1*5+2*2$.

Malheureusement, l'optimalité n'est pas garantie en général par l'algorithme glouton pour n'importe quel jeu de pièces. On peut le voir avec le jeu de pièces $\{6, 4, 1\}$, puisque par exemple, $8 = 1*6+4*1$ en gloutonnant, bien qu'on ait aussi $8 = 2*4$ donnant une meilleure solution. .

Caractériser les jeux de pièces pour lesquels l'algorithme glouton est optimal est un problème ouvert ! Il existe des résultats partiels. Par exemple, considérons la famille de jeux de pièces $\{d^i\}_{0 \leq i \leq n-1}$, avec $d > 1$.

Théorème 11 *L'algorithme glouton est optimal pour les jeux de pièces* $\{d^i\}_{0 \leq i \leq n-1}$, *avec* $d > 1$.

Preuve : soit $g = g_0..g_{n-1}$ la solution gloutonne (les g_i donnant le nombre de pièces de valeur d^i) et $o = o_0..o_{n-1}$ une solution optimale. Si $g = o$ alors le théorème est démontré. Sinon, soit j le plus grand indice tel que $g_j \neq o_j$. $g_j > o_j$ par définition de l'heuristique gloutonne. De plus, les deux sont solutions, donc $\sum_{i=0}^{n-1} g_i d^i = \sum_{i=0}^{n-1} o_i d^i$. Comme $g_i = o_i$ pour $i > j$, $\sum_{i=j+1}^{n-1} g_i d^i = \sum_{i=j+1}^{n-1} o_i d^i$. Donc $\sum_{i=0}^{j} g_i d^i = \sum_{i=0}^{j} o_i d^i$. Ou encore $\sum_{i=0}^{j-1} o_i d^i = \sum_{i=0}^{j-1} g_i d^i + (g_j - o_j) d^j$. Cela implique que $\sum_{i=0}^{j-1} o_i d^i \geq (g_j - o_j) d^j \geq d^j$, puisque $g_j - o_j \geq 1$. Montrons que cela implique qu'au moins un des o_i $(0 \leq i \leq j-1)$ est supérieur ou égal à d. En effet, sinon $\sum_{i=0}^{j-1} o_i d^i \leq \sum_{i=0}^{j-1} d d^i$. Or $\sum_{i=0}^{j-1} d^{i+1} = \frac{d^j - 1}{d-1}$. Donc $\sum_{i=0}^{j-1} o_i d^i \leq d^{j-1}$ ce qui contredit le résultat précédent. Donc $\exists i, 0 \leq i \leq j, \mid o_i \geq d$. Mais alors o' définie par : $o'_l = o_l$ pour l différent de i et $i+1$, $o'_i = o_i - d$, $o'_{i+1} = o_{i+1} + 1$ est aussi une solution puisque $\sum_{k=1}^{n} o'_k d^{k-1} = \sum_{k=1}^{n} o_k d^{k-1} - o_i d^{i-1} - o_{i+1} d^i + (o_i - d) d^{i-1} + (o_{i+1} + 1) d^i = \sum_{k=1}^{n} o_k d^{k-1}$. Et elle contient moins de pièces puisque $d > 1$. Donc o n'était pas optimale. Il ne reste plus que la première alternative possible.

8.1.2 Le cas général

Retournons au cas général. On cherche $m(S)$ le minimum de pièces avec les valeurs $(v_1, ..., v_n)$, triées par valeurs croissantes $(v1 = 1)$. On cherche une formulation récursive du problème. Soit $z(S, i)$ le nombre minimum de pièces choisies parmi les i premières $(v_1, ..., v_i)$ pour faire S. Si on sait calculer les $z(S, i)$, on résoudra le problème puisque $z(S, n) = m(S)$. Sur les bords, on a évidemment $z(0, i) = 0$ et $z(S, 1) = S$. Si $S < v_i$, la solution optimale ne comporte pas de pièces v_i, donc $z(S, i) = z(S, i-1)$. Sinon, on peut utiliser une pièce de valeur v_i ou non. La solution optimale sera de prendre le minimum de pièces pour chacune des alternatives. La récurrence est donc la suivante :

$$z(S, i) = \begin{cases} z(S, i-1) & \text{si } S < v_i \\ min(z(S, i-1), z(S - v_i, i) + 1) & \text{sinon} \end{cases}$$

Un programme récursif s'en déduit immédiatement. La pile d'exécution contiendra toutes les valeurs intermédiaires nécessaires au calcul de $m(S)$. On ne les connait pas a priori et elles sont découvertes "dynamiquement". Par exemple, pour le jeu de pièces $(v_1, v_2, v_3) = (1, 4, 6)$, $m(8) = z(8, 3) = min(z(8, 2), z(2, 3) + 1)$, $z(8, 2) = min(z(8, 1), z(4, 2) + 1)$, $z(4, 2) = min(z(4, 1), z(0, 1) + 1)$, $z(2, 3) = z(2, 2)$, $z(2, 2) = z(2, 1)$. Ce qui donne au retour $z(2, 2) = 2$, $z(2, 3) = 2$, $z(4, 2) = 1$, $z(8, 2) = 2$, et en final $m(8) = 2$ (et non 3 comme le donne l'algorithme glouton).

Une programmation non récursive consiste à remplir un tableau à deux dimensions $z[0.. \mid S \mid, 1..n]$. On initialise la première colonne à 0 et la première ligne avec l'identité. Puis on le remplit dans un ordre compatible avec les dépendances : le calcul de la case (x, i) demande que les cases $(x, i - 1)$ et $(x - v_i, i)$ soient déjà calculées. Ce qui sera assuré par exemple en balayant sur les valeurs puis sur la somme. La complexité est en $O(nS)$.

8.2 Sac à dos

On considère maintenant un problème plus général, classique de l'optimisation : le problème du "sac à dos", permettant d'aborder une classe de problèmes très réalistes dans lesquels on doit optimiser une quantité sous contrainte.

On se donne n objets ayant pour valeurs (entières) $c_1, ..., c_n$ et pour poids entier (ou volume) $w_1, ..., w_n$. Le but est de remplir le sac à dos en maximisant la valeur totale empaquetée ($\sum_{i=1}^{n} c_i$) sous la contrainte de poids ou de contenance maximale ($\sum_{i=1}^{n} w_i \leq W$ où W est la contenance maximale du sac.

Avec une méthode gloutonne, on trie en fonction du rapport "qualité/prix" c_i/w_i, puis on gloutonne en remplissant le sac avec le meilleure élément possible à chaque tour. Est-ce optimal ? Le problème ici est que l'on travaille avec des entiers (on ne peut pas couper les objets).

Avec la même idée que le problème du gymnase, il n'est pas difficile d'imaginer un contre-exemple. Soit 3 objets. Le premier ($c_i/w_i max$) remplit le sac à lui tout seul et les deuxième et troisième sont tels que $c_1 > c_2$ et $c_1 > c_3$, mais $c_2 + c_3 > c_1$ et tels qu'ils puissent rentrer ensemble dans le sac : $W = 10$, $(w_1, w_2, w_3) = (6, 5, 5)$, $(c_1, c_2, c_3) = (7, 5, 5)$.

En programmation dynamique, comme tout à l'heure, on décide de calculer une fonction un peu plus complexe pour avoir une expression récursive de la solution. $C(v, i)$ est le meilleur coût pour remplir un sac de taille v avec les i premiers objets (triés en ordre décroissant de leur qualité/prix. $C(W, n)$ est la solution recherchée. La récurrence est assez simple : soit on a pris le dernier objet (auquel cas le volume diminue mais le coût augmente), soit on ne l'a pas pris (et ni le volume, ni le coût n'a changé). Donc

$$C(v, i) = \max\left(C(v, i - 1), C(v - w_i, i - 1) + c_i\right)$$

8.3 Multiplications enchaînées

Dans ce problème voisin, on considère une suite de matrices dont il faut calculer le produit global. La clé est de trouver une bonne façon de factoriser le calcul pour avoir le moins possible de multiplications élémentaires à effectuer. Soit donc une chaîne $A_1..A_n$ de matrices $A_i(p_{i-1}, p_i)$.

Etudions d'abord le nombre $P(n)$ de parenthèsages possible. Pour $n = 1$, il y a une seule façon de faire : $P(1) = 1$. Pour $n \geq 2$, le produit peut être vu comme le produit de deux sous produits bien parenthèsés avec une démarcation qui peut avoir lieu entre les k et $(k+1)^{eme}$ matrices, pour tout $k \leq n-1$. On obtient donc la récurrence :

$$P(n) = \sum_{k=1}^{n-1} P(k)P(n-k)$$

Il s'agit en fait des nombres de Catalan. Asymptotiquement, ils se comportent en $O(4^n/n^{\frac{3}{2}})$. Cette croissance exponentielle indique que la force brute doit être abandonnée. On va essayer la programmation dynamique.

Cherchons le parenthèsage optimal pour la chaîne $A_i..A_j$. Ce parenthèsage sépare $A_i..A_j$ en deux parties $A_i..A_kA_{k+1}..A_j$ pour un k tel que $i \leq k < j$. Pour que le parenthèsage $A_i..A_j$ soit optimal, il faut que ceux de $A_i..A_k$ et $A_{k+1}..A_j$ le soient aussi. Soit $m(i,j)$ le nombre minimum de multiplications pour le calcul de $A_i..A_j$. Si $j = i$, le problème est trivial : $m(i,i) = 0$. Sinon $m(i,j) = m(i,k) + m(k+1,j) + p_{i-1}p_kp_j$, puisque $A_i..A_k$ est de dimension (p_{i-1}, p_k), et $A_{k+1}..A_j$ est de dimension (p_k, p_j). Mais on ne connait pas k. On explore toutes les solutions, en prenant la meilleure, en programmation dynamique donc :

$$m(i,j) = min_{i \leq k < j}(m(i,k) + m(k+1,j) + p_{i-1}p_kp_j)$$

Le calcul s'effectue en respectant les dépendances : le calcul de $m(i,j)$ demande le calcul de tous les $m(i,k)$ et tous les $m(k+1,j)$. L'initialisation se fait sur $m(i,i+1) = p_{i-1}p_ip_{i+1}$ et $m(i,i) = 0$. Le coût est en $O(n^3)$.

8.4 Plus longue sous-suite

Le dernier exemple concerne la recherche de similarité dans les mots. Considérons deux mots $A = a_1..a_n$, $B = b_1..b_m$. Une sous-chaîne de A est un mot $a_{i1}..a_{ik}$ où chaque lettre est extraite dans l'ordre du mot A. On recherche les sous-chaînes communes à A et B et plus particulièrement la plus longue sous-chaîne commune (PLSCC). Par exemple : $A = aabaababaa$, $B = ababaaabb$, $PLSCC = ababaaa$.

La encore, la solution exhaustive avec l'essai des 2^n sous-suites dans chaque chaîne est peu efficace. En programmation dynamique, notons $p(i,j)$ la longueur de la PLSCC entre les i et j premières lettres de A et de B respectivement.

$$p(i,j) = max(p(i,j-1), p(i-1,j), p(i-1,j-1) + [a_i = b_j])$$

avec $[a_i = b_j] = 1$ si $a_i = b_j$, 0 sinon.

Cette expression récursive demande une preuve. On constate d'abord que ce max est inférieur à $p(i,j)$, car $p(i,j)$ est croissant en i et en j. Preuve de l'égalité : si $a_i \neq b_j$, a_i et b_j ne peuvent pas appartenir tous deux à la même sous-suite commune. On a donc trois cas, soit a_i appartient à la sous-suite et $p(i,j) = p(i,j-1)$, soit c'est b_j qui y appartient, et on a $p(i,j) = p(i-1,j)$, soit aucun des deux n'appartient et $p(i,j) = p(i-1,j-1)$. Si $a_i = b_j$, la PLSCC contient $a_i = b_j$ et alors $p(i,j) = p(i-1,j-1)+1$.

On en dérive le programme suivant :

```
PLSCC(A,B) :
   n := longueur(A) ; m := longueur(B) ;
   pour i de 1 à n faire p[i,0] := (0,|) ;
   pour j de 0 à m faire p[0,j] := (0,-) ;
   pour i de 1 à n faire
      pour j de 1 à m faire
         si A[i] = B[j] alors
            p[i,j] := (p[i-1,j-1]+1,/)
         sinon si p[i-1,j] >= p[i,j-1] alors
                  p[i,j] := (p[i-1,j],|)
               sinon p[i,j] := (p[i,j-1],-)
```

Les éléments du tableau "-", "/" et "|" forment un chemin de $p[0,0]$ à $p[n,m]$. Les lettres de la PLSCC sont les coordonnées des cases contenant "/".

Le coût est en $O(nm)$. Il existe de meilleurs algorithmes dans la littérature pour les cas où les séquences sont ressemblantes ou les cas où les séquences sont très différentes. On trouve alors des algorithmes en $O((n-r)m)$ ou en $O(rm)$ avec r la longueur de la PLSCC de deux mots de longueurs n et m.

Chapter 9

Programmation linéaire

Il s'agit d'un contexte assez général d'optimisation sous contraintes linéaires.

9.1 Exemple introductif d'une usine de production

Il s'agit d'optimiser la production d'une usine qui, partant de deux types de matière première (par exemple, du pétrole de deux qualités différentes), $brut1$ et $brut2$, produit de l'essence et du résidu, avec les paramètres suivants:

- le prix d'achat d'un baril de $brut1$ est 24, celui d'un baril de $brut2$ est 15.
- le prix de vente d'un baril d'essence est 36, celui d'un baril de résidu est 10.
- avec le $brut1$, le rendement de production d'essence est de 80%. Il est de 44% avec le $brut2$.
- le coàt de production d'un baril d'essence est de 0.5 avec le $brut1$ et de 1 avec le $brut2$.
- la capacité maximum de production de l'usine est de 24000 barils d'essence par jour.

Les variables du problème que l'on cherche à décider sont le nombre de barils de $brut1$ et $brut2$ raffinés par jour. On les note p_1 et p_2. L'objectif est de maximiser le profit. Ce profit, qui est égal au montant de la vente, moins celui des achats, moins le coàt de production est de $36 * (0.80p_1 + 0.44p_2) + 10 * (0.20p_1 + 0.56p_2) - 24p_1 - 15p_2 - 0.5p_1 - p_2$, soit $6.3p_1 + 5.44p_2$. On cherche donc à maximiser cette quantité, mais en respectant bien sàr la contrainte imposée par l'usine, à savoir $0.80p_1 + 0.44p_2 \leq 24000$.

Ceci introduit la forme générale à laquelle on peut toujours se ramener pour un problème d'optimisation linéaire.

Définition 8 *Le problème d'optimisation linéaire est un système de la forme :*

$$max(\sum_{j=1}^{n} c_j x_j) \text{ sous contraintes } \bigwedge_{i=1}^{m} \sum_{j=1}^{n} a_{ij} x_j \leq b_i \wedge \bigwedge_{j=1}^{n} x_j \geq 0$$

Une *solution*àest un ensemble $(x_1, .., x_n)$ de valeurs satisfaisant les contraintes. Une solution est *optimale* si elle maximise l'objectif (elle n'est pas forcément unique).

Il n'existe pas forcément de solution optimale, soit parce qu'il n'y a pas de solution, soit parce que il n'y a pas d'optimum fini. Par exemple le problème :

$$max(x_1 - x_2) \text{ s.c.}$$
$$(-2x_1 + x_2 \leq -1) \wedge (-x_1 - 2x_2 \leq -2) \wedge (x_1 \geq 0) \wedge (x_2 \geq 0)$$

possède des solutions comme $(1, 1)$ ou $(5, 0)$ mais on peut toujours trouver une solution meilleure.

9.2 La méthode du simplexe

Elle a été introduite par Dantzig en 1955 et est connue aussi sous le nom de la méthode des points intérieurs.

La première idée est de se passer des inégalités inférieures en rajoutant des variables d'écarts pour obtenir des équations linéaires.

Par exemple :

$$max\ z = 5x_1 + 4x_2 + 3x_3 \text{ s.c.}$$
$$\begin{cases} 2x_1 + 3x_2 + x_3 \leq 5 \\ 4x_1 + x_2 + 2x_3 \leq 11 \\ 3x_1 + 4x_2 + 2x_3 \leq 8 \\ x_1, x_2, x_3 \geq 0 \end{cases}$$

est équivalent à :

$$max\ z = 5x_1 + 4x_2 + 3x_3 \text{ s.c.}$$
$$\begin{cases} x_4 = 5 - 2x_1 - 3x_2 - x_3 \\ x_5 = 11 - 4x_1 - x_2 - 2x_3 \\ x_6 = 8 - 3x_1 - 4x_2 - 2x_3 \\ x_1, x_2, x_3, x_4, x_5, x_6 \geq 0 \end{cases}$$

La deuxième idée est de partir d'une solution $(x_1, .., x_n)$ est d'essayer d'en trouver une meilleure $(x'_1, .., x'_n)$, c'est-à-dire ici $5x_1+4x_2+3x_3 < 5x'_1+4x'_2+3x'_3$. Par exemple, on peut partir de $(0, 0, 0, 5, 11, 8)$ donnant $z = 0$. On essaie alors d'augmenter z en jouant sur une de ses variables. Prenons par exemple x_1. Puisque $x_2 = x_3 = 0$, on a $z = 5x_1$. Les contraintes se réécrivent en :

$$\begin{cases} x_2 = 0 \\ x_3 = 0 \\ x_4 = 5 - 2x_1 \\ x_5 = 11 - 4x_1 \\ x_6 = 8 - 3x_1 \\ x_1, x_4, x_5, x_6 \geq 0 \end{cases}$$

Les contraintes de positivité de x_4, x_5 et x_6 impliquent respectivement $x_1 \leq \frac{5}{2}$, $x_1 \leq \frac{11}{4}$ et $x_1 \leq \frac{8}{3}$. On considère la contrainte la plus dure et la valeur extrême de x_1 soit $\frac{5}{2}$. Toutes les valeurs des variables de l'objectif étant donc fixées, on en déduit une nouvelle solution $(\frac{5}{2}, 0, 0, 0, 1, \frac{1}{2})$ et un meilleur objectif atteint $z = \frac{25}{2}$. La prochaine étape est de réécrire le système en exprimant les variables positives (x_1, x_5 ,x_6) en fonction des variables nulles (x_2, x_3, x_4). On obtient :

$$max\ z = \tfrac{25}{2} - \tfrac{7}{2}x_2 + \tfrac{1}{2}x_3 - \tfrac{5}{2}x_4 \text{ s.c.}$$
$$\begin{cases} x_1 = \frac{5}{2} - \frac{3}{2}x_1 - \frac{1}{2}x_3 - \frac{1}{2}x_4 \\ x_5 = 1 + 5x_2 + 2x_4 \\ x_6 = \frac{1}{2} + \frac{1}{2}x_2 - \frac{1}{2}x_3 + \frac{3}{2}x_4 \\ x_1, x_2, x_3, x_4, x_5, x_6 \geq 0 \end{cases}$$

De l'expression de z, on voit que l'amélioration de l'objectif ne peut être obtenue que par l'augmentation de x_3, puisque les autres coefficients sont négatifs. Gardons donc l'idée que x_2 et x_4 sont nulles mais pas x_3. Des contraintes $x_1 \geq 0$ et $x_6 \geq 0$, on déduit des contraintes sur x_3 : $x_3 \leq 5$ et $x_3 \leq 1$. Comme précédemment, on choisit la plus contraignante qui donne $x_3 = 1$. $x_6 = 0$, donc les nouvelles variables nulles sont x_2, x_4, x_6 et on réitère le procédé consistant à exprimer x_1, x_3, x_5 en fonction de x_2, x_4, x_6. On obtient :

$$max\ z = 13 - 3x_2 - x_4 - x_6 \text{ s.c.}$$
$$\begin{cases} x_1 = 2 - 2x_2 - 2x_4 + x_6 \\ x_5 = 1 + 5x_2 + 2x_4 \\ x_3 = 1 + x_2 + 3x_4 - x_6 \\ x_1, x_2, x_3, x_4, x_5, x_6 \geq 0 \end{cases}$$

La solution courante est $(2, 0, 1, 0, 1, 0)$ donnant $z = 13$. Tous les coefficients de z étant négatifs, on n'a pas intérêt à augmenter une variable de l'objectif. Cette solution est donc optimale.

Résumons la méthode. Soit un problème linéaire, donné sous forme standard

$$max(\sum_{j=1}^{n} c_j x_j) \text{ sous contraintes } \bigwedge_{i=1}^{m} \sum_{j=1}^{n} a_{ij} x_j \leq b_i \wedge \bigwedge_{j=1}^{n} x_j \geq 0$$

On introduit les variables d'écarts $x_{n+1}, .., x_{n+m}$ et on réécrit le problème en

$$max(\sum_{j=1}^{n} c_j x_j) \text{ sous contraintes } \bigwedge_{i=1}^{m} x_{n+i} = b_i - \sum_{j=1}^{n} a_{ij} x_j \wedge \bigwedge_{j=1}^{n+m} x_j \geq 0$$

Chaque itération du simplexe produit un nouveau système appelé "dictionnaire" dans lequel les équations expriment m variables et l'objectif z en fonction des n autres variables. Un dictionnaire est dit *réalisable* si en posant à 0 les variables droites, on obtient une solution. Mais attention, il existe des solutions qui ne correspondent pas à un dictionnaire réalisable. C'est le cas par exemple pour la solution $(1, 0, 1, 2, 5, 3)$ de l'exemple précédent. Ce n'est pas grave, l'important est bien sàr de ne pas laisser des solutions optimales. Les variables gauches d'un

dictionnaire sont appelées les *variables de base*, les autres sont les *variables hors base*. A chaque itération de l'algorithme, une variable entre dans la base et une variable en sort. Le choix de la variable à entrer est dicté par l'amélioration de l'objectif z. La variable à sortir est décidée par la contrainte de conserver toutes les variables positives. Le changement de base ainsi réalisé s'appelle un *pivotage*. Si tous les coefficients de l'objectif sont négatifs, un optimum est atteint et l'algorithme peut s'arrêter. Si tous les coefficients sont positifs, on est sàr qu'il n'y a pas d'optimum fini et l'algorithme peut aussi s'arrêter.

IL y a une interprétation géométrique à l'algorithme du simplexe. Le système d'égalités linéaires définit un polytope des solutions possibles. L'algorithme démarre d'un noœud du polytope et se déplace le long des arcs du polytope jusqu'à trouver le noœud d'une solution optimale.

9.2.1 Le problème de la convergence

Le problème est de garantir que l'algorithme va terminer au bout donc d'un nombre fini d'itérations. Ce n'est pas évident a priori puisque l'algorithme tel qu'il a été exprimé pourrait boucler en regénérant un dictionnaire déjà construit sans pouvoir pour autant conclure (le nombre de dictionnaires possibles pour un problème est fini).

En 1977, Bland a trouvé une règle qui assure la terminaison. C'est en fait assez simple, il suffit lorsqu'il y a un choix possible sur la variable entrante dans la base ou sur la variable sortante de la base, de choisir toujours la variable de plus petit indice, considérant les variables totalement ordonnées. La preuve de ce résultat n'est pas tout à fait triviale.

9.2.2 Le problème de l'initialisation

L'algorithme du simplexe demande pour démarrer que le dictionnaire initial soit réalisable, c'est-à-dire que la mise à 0 des variables droites donne une solution, ou encore que le vecteur 0 sur les variables de décision du problème soit une solution. Ce n'est évidemment pas le cas dès qu'un b_i est strictement négatif.

Par exemple, considérons le problème suivant :

$$max\ z = x_1 - x_2 + x_3 \text{ s.c.}$$
$$\begin{cases} 2x_1 - x_2 + 2x_3 \leq 4 \\ 2x_1 - 3x_2 + x_3 \leq -5 \\ -x_1 + x_2 - 2x_3 \leq -1 \\ x_1, x_2, x_3 \geq 0 \end{cases}$$

Son dictionnaire initial est :

$$\begin{cases} x_4 = 4 - 2x_1 + x_2 - 2x_3 \\ x_5 = -5 - 2x_1 + 3x_2 - x_3 \\ x_6 = -1 + x_1 - x_2 + 2x_3 \\ z = x_1 - x_2 + x_3 \end{cases}$$

Il s'agit maintenant de transformer ce dictionnaire non réalisable en un dictionnaire réalisable à partir duquel le simplexe va pouvoir se dérouler. Pour cela, on va aussi utiliser l'algorithme du simplexe !

On va considérer le problème auxilliaire suivant qui va pour objectif de trouver $x_0 = 0$:

$$max\ z = -x_0 \text{ s.c.}$$
$$\begin{cases} 2x_1 - x_2 + 2x_3 - x_0 \leq 4 \\ 2x_1 - 3x_2 + x_3 - x_0 \leq -5 \\ -x_1 + x_2 - 2x_3 - x_0 \leq -1 \\ x_0, x_1, x_2, x_3 \geq 0 \end{cases}$$

Son dictionnaire initial est :

$$\begin{cases} x_4 = 4 - 2x_1 + x_2 - 2x_3 + x_0 \\ x_5 = -5 - 2x_1 + 3x_2 - x_3 + x_0 \\ x_6 = -1 + x_1 - x_2 + 2x_3 + x_0 \\ z = -x_0 \end{cases}$$

On le transforme immédiatement en rentrant x_0 dans la base et en sortant la variable x_i correspondante au b_i le plus négatif (x_5 ici). On obtient alors :

$$\begin{cases} x_0 = 5 + 2x_1 - 3x_2 + x_3 + x_5 \\ x_4 = 9 - 2x_2 - x_3 + x_5 \\ x_6 = 4 + 3x_1 - 4x_2 + 3x_3 + x_5 \\ z = -5 - 2x_1 + 3x_2 - x_3 - x_5 \end{cases}$$

qui par construction est réalisable et peut donc enclencher le simplexe. La prochaine variable à entrer est x_2. Ce qui fait sortir x_6. Le nouveau dictionnaire résultant est :

$$\begin{cases} x_0 = 2 - \frac{1}{4}x_1 - \frac{5}{4}x_3 + \frac{1}{4}x_5 + \frac{3}{4}x_6 \\ x_2 = 1 + \frac{3}{4}x_1 + \frac{3}{4}x_3 + \frac{1}{4}x_5 - \frac{1}{4}x_6 \\ x_4 = 7 - \frac{3}{2}x_1 - \frac{5}{2}x_3 + \frac{1}{2}x_5 + \frac{1}{2}x_6 \\ z = -2 + \frac{1}{4}x_1 + \frac{5}{4}x_3 - \frac{1}{4}x_5 - \frac{3}{4}x_6 \end{cases}$$

Entre alors x_1 et sort x_4. Ce qui donne :

$$\begin{cases} x_0 = \frac{5}{6} - \frac{5}{6}x_3 + \frac{1}{6}x_4 + \frac{1}{6}x_5 + \frac{2}{3}x_6 \\ x_1 = \frac{14}{3} - \frac{5}{3}x_3 - \frac{2}{3}x_4 + \frac{1}{3}x_5 + \frac{1}{3}x_6 \\ x_2 = \frac{9}{2} - \frac{1}{2}x_3 - \frac{1}{2}x_4 + \frac{1}{2}x_5 \\ z = -\frac{5}{6} + \frac{5}{6}x_3 - \frac{1}{6}x_4 - \frac{1}{6}x_5 - \frac{2}{3}x_6 \end{cases}$$

On rentre alros x_3 et on sort x_0. Le fait de sortir x_0 va permettre de conclure puisque l'on obtiendra forcément l'objectif $z = -x_0$ et on pourra conclure. Ce qui donne :

$$\begin{cases} x_1 = 3 + 2x_0 - x_4 - x_6 \\ x_2 = 4 + \frac{3}{5}x_0 - \frac{3}{5}x_4 + \frac{2}{5}x_5 - \frac{2}{5}x_6 \\ x_3 = 1 - \frac{6}{5}x_0 + \frac{1}{5}x_4 + \frac{1}{5}x_5 + \frac{4}{5}x_6 \\ z = -x_0 \end{cases}$$

L'algorithme se termine donc et produit l'optimum $x_0 = 0$. Donc il existe une solution au problème initial. Son dictionnaire initial réalisable s'obtient en éliminant les x_0 et en écrivant l'objectif $z = x_1 - x_2 + x_3$ en fonction des variables hors-base :

$$\begin{cases} x_1 = 3 - x_4 - x_6 \\ x_2 = 4 - \frac{3}{5}x_4 + \frac{2}{5}x_5 - \frac{2}{5}x_6 \\ x_3 = 1 + \frac{1}{5}x_4 + \frac{1}{5}x_5 + \frac{4}{5}x_6 \\ z = -\frac{1}{5}x_4 - \frac{1}{5}x_5 + \frac{1}{5}x_6 \end{cases}$$

Ce dictionnaire est cette fois-ci réalisable puisque $(3, 4, 1, 0, 0, 0)$ est une solution. Elle permet d'obtenir $z = 0$. On peut alors dérouler le simplexe pour améliorer z...

9.2.3 Complexité

En pratique, on observe que le nombre d'itérations requis est de l'ordre de $\frac{3m}{2}$ où m est le nombre de contraintes (non sensible donc au nombre de variables). Cet algorithme est mis en oœuvre dans de nombreux logiciels utilisés comme CPLEX, OSL, XPRESS, MPSX, ...

On peut néanmoins construire des exemples pathologiques dans lesquels le nombre d'itérations est exponentiel. Par exemple, on peut montrer que :

$$max\ z = \sum_{j=1}^{n} 10^{n-j} x_j \text{ s.c. } \bigwedge_{i=1}^{n} (2 \sum_{j=1}^{i-1} 10^{i-j} x_j + x_i \leq 100^{i-1}) \wedge \bigwedge_{j=1}^{n} (x_j \geq 0)$$

va provoquer 2^{n-1} itérations.

Chapter 10

Notions de géométrie algorithmique

De nombreux domaines d'application demandent de savoir construire de manière efficace des objets de nature géométrique : robotique, vision par ordinateur, informatique graphique, imagerie médicale, réalité virtuelle, conception assistée, ... Cette sous-discipline de l'algorithmique a pris un vrai essor à partir de 1975 avec les aspects suivants :

- l'identification de structures géométriques fondamentales : polytopes, triangulations, arrangements, diagrammes de VoronoÔ ;
- l'utilisation de la géométrie combinatoire (relations faces-sommets par exemple) ;
- l'invention de nouvelles techniques algorithmiques comme le balayage par exemple ;
- l'intérêt croissant pour les algorithmes probabilistes.

Les difficultés de programmation résident dans la précision des calculs et dans le traitement des cas particuliers. Dans la suite, on va se limiter à deux dimensions pour des raisons pédagogiques.

10.1 Segments de droites

Les points du plan sont définis par un couple de coordonnées : $p = (x, y)$. Un segment de droite est défini par deux points $p_1 = (x_1, y_1)$ et $p_2 = (x_2, y_2)$. Il s'agit en fait de l'ensemble des points $p_3 \in [p_1 p_2]$ qui sont une combinaison linéaire convexe de p_1 et p_2 :

$$p_3 = \alpha p_1 + (1 - \alpha) p_2 \text{ avec } \alpha \in [0..1]$$

De nombreuses questions sur la manipulation d'objets géométriques utilisent des fonctions élémentaires comme :

1. Donnés $[p_0p_1]$ et $[p_0p_2]$, est-ce que $[p_0p_1]$ est situé dans le sens des aiguilles d'une montre par rapport à $[p_0p_2]$ à
2. Donnés $[p_0p_1]$ et $[p_1p_2]$, si l'on parcours $[p_0p_1]$ puis $[p_1p_2]$, la bifurcation au point p_1 se fait-elle vers la gauche à
3. $[p_1p_2]$ et $[p_3p_4]$ sont-ils sécants à

On va montrer que ces questions peuvent être résolues en $O(1)$ sans division ni fonction trigonométrique.

Etant donné un point p, il définit aussi un segment $[Op]$ avec O étant l'origine du plan. On considérera aussi le vecteur associé, noté p. Un outil intéressant sur les vecteurs est le produit p_1p_2 qui définit l'aire signée du parallélogramme, formé des points O, p_1, p_2 et $p_1 + p_2$. La figure 10.1 donne un moyen simple de calculer l'aire (signée) en fonction des coordonnées des points $p_1(a, b)$ et $p_2(c, d)$. L'aire cherchée est l'aire du grand rectangle moins l'aire des petits moins l'aire des triangles. Ce qui donne $p_1p_2 = (a + c)(b + d) - 2bc - dc - ab = ad - bc$ (ou encore le déterminant de la matrice (p_1p_2)). On a $p_1p_2 = -p_2p_1$.

Si p_1p_2 est strictement positif, alors p_1 est placé dans le sens des aiguilles d'une montre par rapport à p_2 (en tournant autour de l'origine). Si il est nul, les vecteurs sont colinéaires et les points O, p_1 et p_2 sont alignés.

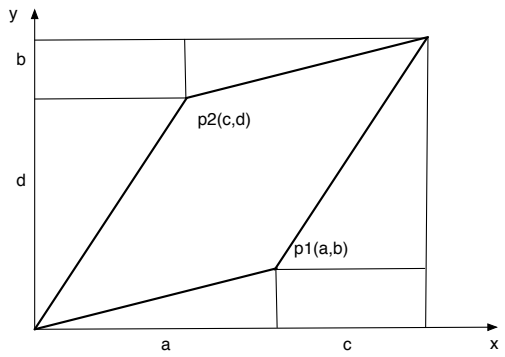

Figure 10.1: Calcul de la formule de l'aire d'un parallélogramme.

La première question revient à savoir si le produit $(p_1-p_0)(p_2-p_0)$ est positif. La deuxième question revient à calculer l'orientation de l'angle $< p_0p_1p_2 >$, ce que l'on peut faire sans calculer l'angle. En effet, cela revient à savoir si $[p_0p_2]$ se situe dans le sens des aiguilles d'une montre par rapport à $[p_0p_1]$.

La question de l'intersection est un peu plus compliquée. Deux segments se coupent si et seulement si chaque segment coupe la droite contenant l'autre ou si une extrémité d'un segment appartient à l'autre segment (c'est le cas limite).

Notons $d_{ijk} = (p_k - p_i)(p_j - p_i)$. $d_{ijk} > 0$ signifie que le point p_k se trouve à gauche du segment $[p_i p_j]$. $d_{ijk} \leq 0$ signifie que le point p_k se trouve à droite du segment $[p_i p_j]$. Le cas $d_{ijk} = 0$ indique que les trois points p_i, p_j et p_k sont alignés.

On va calculer $d_{341}, d_{342}, d_{123}$ et d_{124} pour avoir l'orientation relative de chaque extrémité d'un segment par rapport à l'autre segment. $[p_1 p_2]$ traverse la droite support de $[p_3 p_4]$ si $[p_3 p_1]$ et $[p_3 p_2]$ ont des orientations opposées relativement à $[p_3 p_4]$. En pareil cas, les signes de d_{341} et d_{342} diffèrent. De même, $[p_3 p_4]$ traverse la droite support de $[p_1 p_2]$ si les signes de d_{123} et d_{124} diffèrent. Dans le cas où d_{ijk} vaut 0, le point p_k est sur la droite support du segment $[p_i p_j]$. La procédure `Sur-segment` indique si p_k est entre les extrémités de $[p_i p_j]$. Le programme décidant si les segments se coupent est donc le suivant :

```
Intersection (p1,p2,p3,p4) :
    d1 := Direction (p3,p4,p1) ;
    d2 := Direction (p3,p4,p2) ;
    d3 := Direction (p1,p2,p3) ;
    d4 := Direction (p1,p2,p4) ;
    si ((d1>0 et d2<0) ou (d1<0 et d2>0)) et
       ((d3>0 et d4<0) ou (d3<0 et d4>0)) alors retour vrai
       sinon si d1=0 et Sur-segment (p3,p4,p1) alors retour vrai
         sinon si d2=0 et Sur-segment (p3,p4,p2) alors retour vrai
           sinon si d3=0 et Sur-segment (p1,p2,p3) alors retour vrai
             sinon si d4=0 et Sur-segment (p1,p2,p4) alors retour vrai
               sinon retour faux

Direction (pi,pj,pk) = (pk-pi)(pj-pi)
Sur-segment (pi,pj,pk) = (min(xi,xj)<=xk<=max(xi,xj) et
                          (min(yi,yj)<=yk<=max(yi,yj))
```

10.2 Technique de balayage

On cherche par exemple à généraliser la question de l'intersection. Soit S un ensemble de n segments du plan. Le problème consiste à détecter toutes les paires de segments de S qui s'intersectent et à calculer les coordonnées de leurs points d'intersection (soit a le nombre d'intersections). Une méthode naÔve consiste à tester les $n(n-1)/2$ paires de segments, ce qui donne une complexité en $O(n^2)$. Ce qui est néanmoins optimal puisqu'il peut y avoir de l'ordre de n^2 intersections (il suffit de constater que le nombre maximal d'intersections avec n segments est celui que l'on peut obtenir avec $n-1$ segments auquel on ajoute au plus $n-1$ intersections. On obtient donc la somme des premiers entiers qui est quadratique). En pratique a est très petit devant n^2. On va montrer que l'algorithme par balayage sera de complexité $O((n+a).log\,(n+a))$.

Pour simplifier et éviter le traitement fastidieux des cas particuliers, on suppose que toutes les extrémités et intersections ont des abcisses différentes et que trois segments ne peuvent pas se couper en un seul point.

L'idée est que si deux segments S et S' s'intersectent, toute droite verticale Δ dont l'abscisse est suffisamment proche de celle de l'intersection intersecte les deux segments. De plus S et S' sont consécutifs dans la suite des segments intersectés par Δ, triés par ordre croissant des ordonnées de leurs points d'intersection avec Δ. On maintient l'état de la droite de balayage comme étant la liste triée (par ordre croissant des ordonnées des points d'intersection avec Δ) des segments qu'elle coupe. Cette liste (appelée dans la suite Y) ne se trouve modifiée que lorsque Δ rencontre l'une des extrémités d'un segment ou un point d'intersection.

L'algorithme va gérer deux structures dynamiques X et Y. X va contenir la liste triée par ordre croissant des abcisses des extrémités des segments, ainsi que les abcisses des points d'intersection (non connus initialement). Initialement, X ne contient que les abcisses des extrémités de chaque segment. Y va contenir la liste des segments intersectés par la droite de balayage. Y est vide initialement. L'algorithme va balayer la structure X (d'oà le nom de "balayage vertical"). Pour un segment S de Y, on notera $pre(S)$ et $succ(S)$ son prédécesseur et son successeur dans la structure si il existe. X contient trois types d'événements : une abcisse d'extrémité gauche, une abcisse d'extrémité droite ou une abcisse d'une intersection. Les traitements pour chaque cas sont les suivants :

- (S, x) est l'extrémité gauche d'un segment S : S est inséré dans la structure Y. Si $pre(S)$ et S (ou S et $succ(S)$) s'intersectent, leur point d'intersection est inséré dans X ;

- (S, x) est l'extrémité droite d'un segment S : on retire S de Y. Si $pred(S)$ et $succ(S)$ s'intersectent en un point d'abscisse supérieure à x, leur point d'intersection est inséré dans X si il ne s'y trouve pas déjà ;

- (SS', x) est l'abcisse du point d'intersection des segments S et S' : on échange S et S' dans Y ;

La figure 10.2 illustre le fonctionnement de l'algorithme de balayage, montrant le moment oà une intersection est détectée puis insérée.

La structure Y compte au maximum n segments et doit supporter les opérations de recherche, d'insertion, de suppression. Elles peuvent être obtenues en $O(log\, n)$ avec des techniques de type arbre équilibré. La structure X compte au maximum $2n + a$ événements. De la même façon, on peut se ramener en une complexité en $O(log(n + a)$. Il ne faut pas oublier le coàt de l'initialisation en $O(n.log\ n)$, puisqu'un tri préalable est nécessaire. $2n + a$ événements sont traités et chaque événement implique un nombre borné d'opérations, d'oà le coàt global annoncé en $O((n + a).log\ (n + a))$.

Pour terminer l'algorithme, il faut aussi s'intéresser aux cas limites. Il s'agit souvent de la difficulté principale et source de "bugs". Dans le tri préalable, si il y a égalité entre deux extrémités, on peut placer les extrémités gauches avant les droites, puis dans chacun de ces ensembles, trier dans l'ordre des ordonnées. Dans la preuve, la condition de non croisement de trois segments au même point n'est pas nécessaire. De même, l'algorithme reste correct même si des segments

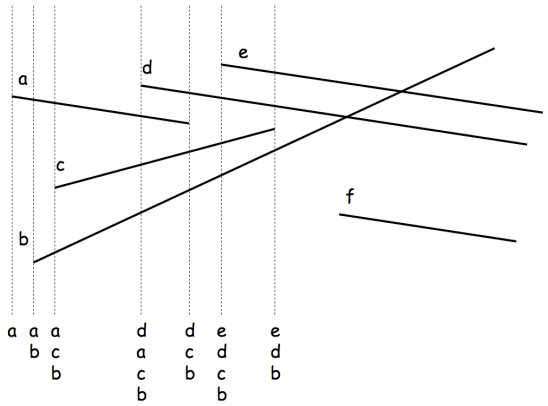

Figure 10.2: La technique de balayage.

sont verticaux, à condition que l'extrémité inférieure d'un segment vertical soit traitée comme si c'était une extrémité gauche et que l'extrémité supérieure soit traitée comme une extrémité droite.

10.3 Enveloppe convexe

Dans un grand nombre d'applications graphiques, il est nécessaire de sélectionner des ensembles de points en ne retenant que leur contour. Formellement, l'enveloppe convexe $EC(Q)$ d'un ensemble Q de n points est le plus petit polygone convexe P tel que chaque point de Q est soit sur le contour de P, soit à l'intérieur. Chaque sommet de $EC(Q)$ est un point de Q. Les algorithmes affichent les sommets de $EC(Q)$ dans l'ordre contraire des aiguilles d'une montre, par exemple dans le balayage de Graham avec une complexité en $O(nlog\, n)$ ou dans le parcours de Jarvis avec une complexité en $O(nh)$, oà h est le nombre de sommets de $EC(Q)$.

Regardons le balayage circulaire de Graham, défini ainsi :

```
Balayage-Graham (Q) :
  Soit p0 le point de Q d'ordonnée minimale
   (si plusieurs prendre le plus à gauche)
  Soient (p1,..,pm) les autres points de Q triés par angles polaires
   (mesurés relativement à p0 dans le sens trigonométrique)
   (si plusieurs de même angle, on ne garde que le plus éloigné de p0)
  Empiler (p0,S) ; Empiler (p1,S) ; Empiler (p2,S) ;
  pour i de 3 à m faire
    Tant que pi n'est pas à gauche de [sous-sommet(S), sommet(S)]
       faire Dépiler (S) ;
    Empiler (pi,S)
  retour S
```

La preuve s'effectue en montrant l'invariant de boucle suivant : à l'indice i, S contient $EC(Q_{i-1})$ dans l'ordre trigonométique ($Q_i = \{p_0, .., p_i\}$). La encore, c'est la complexité du tri initial qui domine, pour donner une complexité en $O(nlog\, n)$.

La figure 10.3àillustre le fonctionnement de l'algorithme.

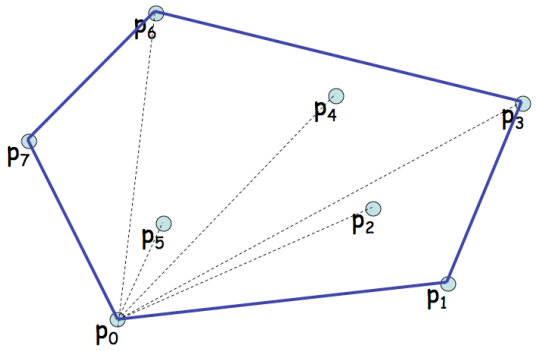

Figure 10.3: Balayage circulaire de Graham pour calculer une enveloppe convexe.

10.4 Diagrammes de VoronoÔ

Soit S un ensemble fini de n points du plan, appelés "germes". On appelle cellule de Voronoï associée au germe p de S, l'ensemble des points qui sont plus proches de p que tout autre germe de S.

$$V(p) = \{x | \forall q \in S, d(x,p) \leq d(x,q)\}$$

La frontière entre les cellules de deux germes distincts se situe forcément sur la médiatrice qui sépare les deux germes. En effet, les points de cette médiatrice sont équidistants des deux germes donc on ne peut pas affirmer qu'ils se situent dans l'une ou l'autre cellule de Voronoi. Pour un ensemble de germes, le diagramme de Voronoi se construit donc en déterminant les médiatrices de chaque couple de germes. Un point d'une médiatrice appartient alors à une frontière de Voronoi s'il est équidistant d'au moins deux germes et qu'il n'existe pas de distance plus faible entre ce point et un autre germe de l'ensemble.

Les diagrammes de Voronoï sont utilisés, ou réinventés sous de nombreux noms, dans différents domaines. Ils interviennent souvent lorsque l'on cherche à partitionner l'espace en sphères d'influence. La figure 10.4 donne un exemple de diagramme de Voronoï.

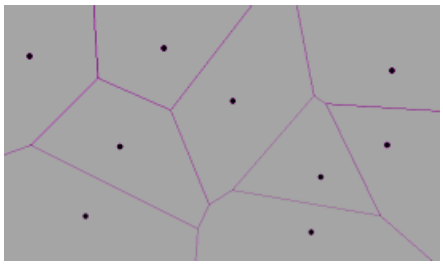

Figure 10.4: Un diagramme de Voronoï.

10.4.1 Construction incrémentale

L'idée consiste à procéder de manière incrémentale : supposons que le diagramme de Voronoï d'un ensemble $P_{n-1} = \{p_1, .., p_{n-1}\}$ de $n-1$ points du plan a été construit ; que faut-il modifier à ce diagramme pour construire le diagramme d'un ensemble $P = P_{n-1} \cap \{p_n\}$ à Intuitivement, on comprend aisément que l'insertion d'un point p_n ne va modifier le diagramme de Voronoï que localement. Quels sont les sommets de Voronoï qui vont disparaître lors de cette insertion à Ce sont ceux dont le cercle associé (circonscrit aux germes voisins) contient p_n, car on sait que l'intérieur des cercles centrés aux sommets

de Voronoï et circonscrits aux germes voisins doit toujours être vide. Ces sommets sont dans une zone limitée du diagramme, ce qui conduit à un algorithme rapide (en $O(n)$) pour insérer p_n et mettre à jour le diagramme de Voronoï. Cet algorithme, proposé par Green et Sibson en 1978, fonctionne de la manière suivante :

1. trouver le germe p_i tel que $p_n \in V(p_i)$;

2. tracer la médiatrice du segment $[p_i p_n]$ et calculer ses intersections x_1 et x_2 avec la frontière de $V(p_i)$ (il n'y en a que deux car la cellule est convexe) ;

3. $[x_1 x_2]$ est, par construction, une arête de Voronoï du diagramme de P ; c'est même l'arête séparant $V(p_i)$ de $V(p_n)$. x_2 est lui sur une arête de Voronoï du diagramme de P_{n-1}, séparant $V(p_i)$ d'une autre cellule $V(p_j)$;

4. recommencer le processus en remplaçant p_i par p_j ;

5. itérer jusqu'à retomber sur x_1 ;

6. on a ainsi construit la frontière de $V(p_n)$; mettre à jour les arêtes de Voronoï qu'elle intersecte, en supprimant les morceaux à l'intérieur de $V(p_n)$.

La figure 10.5 illustre le fonctionnement de l'algorithme.

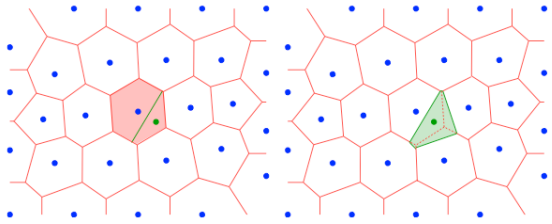

Figure 10.5: Evolution incrémentale d'un diagramme de Voronoï.

La cellule $V(p_n)$ peut avoir au pire $n-1$ arêtes ; le temps de calcul de cette cellule est donc en $O(n)$. Puisqu'il y a n diagrammes de Voronoï successifs à calculer, la complexité totale de cet algorithme est en $O(n^2)$.

10.4.2 VoronoÔ par balayage

L'algorithme le plus utilisé actuellement pour calculer un diagramme de Voronoï est l'algorithme de Steven Fortune (1987, Laboratoires Bell AT&T), démontré comme asymptotiquement optimal, permettant de calculer le diagramme de VoronoÔ d'un ensemble de n points du plan avec une complexité de $O(nlog\,n)$. du à Steven Fortune (1987). Il est facilement implémentable. Cet algorithme est dit par balayage car il "balaie" progressivement le plan par une ligne et selon une certaine direction, de telle manière qu'à tout moment, dans la zone déjà balayée, le diagramme de VoronoÔ est construit de manière définitive. Cela peut sembler impossible, car l'apparition d'un nouveau point de P lors du balayage va modifier certaines cellules avant ce point. L'idée géniale de Fortune consiste à utiliser une troisième dimension, afin d'"anticiper" les modifications du diagramme (en quelque sorte, de "voir l'avenir").

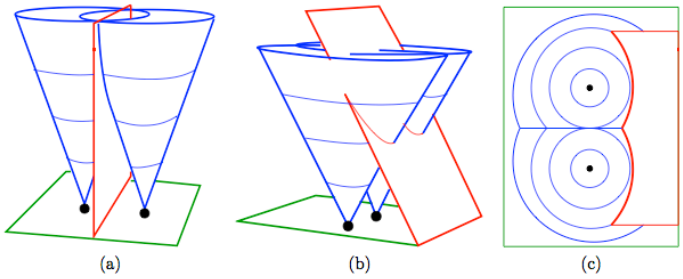

Figure 10.6: Algorithme de Fortune. (a) L'intersection de deux cônes se projette sur la médiatrice du segment reliant les deux germes correspondants. (b) Les cônes sont balayés par un plan incliné d'un angle identique à l'angle des cônes. (c) Vue du dessous : l'intersection du plan incliné avec l'ensemble des cônes correspond à un front parabolique.

Chapter 11

Algorithmique probabiliste et parallèle

Une façon pragmatique de contourner des complexités algorithmiques rédhibitoires pour des problèmes de grande taille est d'avoir recours est de renoncer au déterminisme et d'avoir recours aux probabilités. Il faudra évidemment relàcher des exigences en admettant un résultat approché.

Lors de l'exécution, l'algorithme fait des choix probabilistes guidés par des tirages aléatoires : par exemple des lancers de pièces pile ou face. On peut regrouper les méthodes en deux catégories :

- les méthodes de type Monte-Carlo : le résultat du calcul est aléatoire. Le calcul est toujours rapide, mais probablement correct (c'est-à-dire correct avec une certaine probablité).
- les méthode de type Las-Vegas : le temps d'exécution est aléatoire. Le résultat est toujours correct, mais la rapidité est aléatoire. Un exemple d'algorithme de Las Vegas est l'algorithme de tri rapide aléatoire que l'on a vu dans le cours oà les pivots sont choisis aléatoirement mais le résultat est toujours trié. La définition usuelle d'un algorithme de Las Vegas est que seulement l'espérance du temps de calcul est finie.

11.1 Générateurs aléatoires

On suppose l'existence d'un générateur de nombres aléatoires dont l'utilisation se fait à coàt unitaire.

Définition 9 *Soit a, b, deux nombres réels. La fonction $uniforme(a, b)$ retourne une valeur x choisie de façon aléatoire et uniforme dans l'intervalle $[a, b]$.*

On utilise souvent sa version entière :

Définition 10 *Si a et b sont deux entiers, alors la fonction $uniforme(a, b)$ retourne la valeur entière $v \in [a, b]$ avec la probabilité $\frac{1}{b-a+1}$.*

Pour certaines applications le vrai hasard est important : loteries, cryptographie, ... On peut pour celà utiliser des données issues du matériel de l'ordinateur comme le bit le moins significatif de l'horloge, des prédictions dans les caches, ...

Mais il devient impossible de répéter l'exécution d'un calcul. Les programmes sont difficiles à déboguer ou à comparer. En pratique, on utilise des générateurs de nombres *pseudo-aléatoires*.

Définition 11 *Une séquence de nombres est dite pseudo-aléatoire si elle est générée de façon déterministe mais semble avoir été produite de façon purement aléatoire (passe avec succès certains tests statistiques).*

Un exemple est la méthode linéaire congruentielle. On choisit minutieusement 4 nombres :

- m : le modulo ($m > 0$) ;
- a : le multiplicateur ($0 \leq a < m$) ;
- c : le saut ($0 \leq c < m$) ;
- x_0 : la valeur de départ ($0 \leq x_0 < m$).

La séquence de nombres pseudo-aléatoire est :

$$x_{n+1} = (ax_n + c) \bmod m$$

Certains auteurs recommendent (entre autres) : $m = 2^{31} - 1$, $a = 16807$, $c = 0$.

11.2 Algorithmes numériques

On peut calculer avec le hasard. Prenons l'exemple célèbre du calcul de π.

Considérons un carré de 1 cm de côté et le cercle circonscrit de rayon 0,5 cm. On choisit alors un point p de façon aléatoire et uniforme dans le carré. Les probabilités étant proportionnelles aux surfaces, la probabilité pour que p soit dans le cercle est de $\frac{\pi}{4}$.

L'idée de l'algorithme est donc de choisir de façon aléatoire, uniforme et indépendante n points dans le carré. Soit X, le nombre de points dans le cercle. L'espérance $E(X) = \frac{n\pi}{4}$. La loi faible des grands nombres dit que la moyenne empirique va converger vers l'espérance mathématique : pour tout $\epsilon > 0$, $\lim_{n\to\infty} P(|X - E(X)| \geq \epsilon) = 0$. On en en déduit le code suivant :

```
fonction piMC(n)
  k=0 ;
  pour i=1 à n faire
    x=uniforme(0,1) ;
    y=uniforme(0,1) ;
   si x^2 + y^2 <= 1 alors k=k+1
  retourner 4*k/n
```

On peut généraliser l'idée précédente pour calculer $\int_0^1 f(x)dx$.

```
fonction viser_juste(f,n)
  k=0 ;
  pour i=1 à n faire
    x=uniforme(0,1) ;
    y=uniforme(0,1) ;
    si y<=f(x) alors k=k+1
  retourner k/n
```

En pratique on utilise des algorithmes déterministes dont la convergence est plus rapide.

11.3 Comptage probabiliste

On veut compter le nombre de truites dans un lac. On suppose qu'il est possible de pêcher des truites (avec remise) de façon aléatoire, uniforme et indépendante. On procède de la façon suivante:

```
répéter
  capturer une truite
  la peindre en rouge
  la remettre dans le lac
jusqu'à ce qu'on recapture une truite rouge.
```

Soit k, le nombre de truites capturées avant de prendre une truite rouge. On estime le nombre de truites dans le lac à $\frac{k^2}{2}$. Pour le montrer, il faut calculer la probabilité de répétitions. Par exemple, il est connu que pour une salle de 25 personnes par exemple, la probabilité que l'anniversaire de 2 d'entre elles tombe le même jour est presque de 50%.

En effet, le nombre de façons de choisir k objets différents parmi n est $n(n-1)...(n-k-1)$ et le nombre de façons de choisir k objets parmi n est n^k. Le ratio donne $\frac{n!}{(n-k)!n^k}$ et la probabilité d'avoir deux objets identiques est donc de $1-\frac{n!}{(n-k)!n^k}$. Pour l'anniversaire, on a $n=365$ et $k=25$, ce qui donne une valeur supérieure à 56%.

On a $\frac{n!}{(n-k)!n^k} \approx e^{-\frac{k^2}{2n}}$ qui se prouve en utilisant la formule de Stirling : $n! \approx \sqrt{2\pi n}(\frac{n}{e})^n$. Lorsqu'on fait tendre n vers l'infini, la probabilité d'une répétition est donc de l'ordre de $\frac{k^2}{2n}$.

On en déduit un algorithme permettant d'estimer le nombre de truites dans l'ensemble S :

```
fonction compter(S)
  k = 0 ; t = vide ;
  a = uniforme(S) ;
  tantque a n'est pas dans T faire
    k = k+1 ;
    T = T + {a} ;
    a = uniforme(S)
  retourner k*k/2
```

En pratique, il sera nécessaire d'itérer suffisamment pour que plusieurs éléments aient été comptés 2 fois.

11.4 Le problème des philosophes

Des philospohes sont assis autour d'une table, une assiette de spaghetti devant eux. Chacun partage une fourchette avec son voisin de droite et une autre avec son voisin de gauche. Il peut, vérifier si une fourchette est libre et s'en saisir, ou en relacher une. Il mange s'il détient une fourchette dans chaque main.

Le problème est de trouver un algorithme pour chaque philosophe tel que deux philosophes ne détiennent pas ensemble la même fourchette (correction) et tel que chaque philosophe mange au bout d'un temps raisonnable (vivacité).

Il a été montré qu'il n'existe pas de solution symétrique (les philosophes appliquent le même algorithme) à ce problème !

Par exemple, une solution symétrique incorrecte est :

```
1. Attendre que la fourchette de gauche soit libre, la saisir
2. Attendre que la fourchette de droite soit libre, la saisir
3. Manger
4. Reposer les deux fourchettes
```

Le principe de l'algorithme probabiliste est le suivant :

- chaque philosophe tire au sort la fourchette dont il va se saisir en premier ;
- si elle est libre il la prend, sinon il attend qu'elle soit libre ;
- une fois qu'il obtient une fourchette, il examine si l'autre est libre ;
- si tel est le cas il la prend sinon il repose la première.

On imagine une solution informatique par variables partagées. Les fourchettes $f[i]$ sont partagées entre les philosophes P_{i-1} et P_i. Ce sont des variables booléennes initialisées à $vrai$. L'algorithme (symétrique) pour le philosophe P_i est défini ainsi :

```
Répéter infiniment
(1)  prem = uniforme(0,1) ;
(2)  attendre jusqu'à f[i+prem] ;
(3)  f[i+prem] = faux ;
     si f[i+1-prem] alors
     f[i+1-prem] = faux ;
(4)     manger
(5)  sinon f[i+prem] = vrai
```

Il existe des situations de blocage mais qui interviennent avec une très faible probabilité. Si l est le temps maximum d'une étape, on peut montrer que la probabilité pour que l'on arrive dans un temps $10l$ à un état dans lequel au moins un philosophe mange est supérieure ou égale à $\frac{1}{16}$.

11.5 Macro-communication sur un anneau

On dispose de p machines arrangées en un anneau orienté $(0..p-1)$. Chaque machine peut connaître son numéro logique ($my\_num()$) et le nombre total de machines ($tot\_proc\_num()$). La règle du jeu est : toutes les machines exécutent le même programme (mais sur des données locales différentes : fonctionnement SPMD, "Single Program Multiple Data").

Chaque machine peut envoyer un message à son successeur sur l'anneau : $send(@, L)$. @ est l'adresse chez l'émetteur de la donnée. L est sa longueur. De même, chaque machine peut décider de recevoir une message en provenance de son prédécesseur : $receive(@, L)$. @ est l'adresse de rangement de la donnée. L est sa longueur. Comme toutes les machines exécutent le même programme, il doit y avoir un "receive" en face de chaque "send". Sinon, il y aura un blocage. Les adresses sont locales (et peuvent donc être différentes entre le send et le receive).

On va faire l'hypothèse réaliste de coàt linéaire de la communication : $\beta + L\tau$.

11.5.1 Diffusion

Le problème est de faire en sorte qu'une machine P_k (ex. maître) diffuse un message à toutes les autres machines. L'algorithme est le suivant :

```
Broadcast(k,@,L) :
        q := my_num()
        p := tot_proc_num()
        si q=k alors send(@,L)
        sinon
              si q=k-1 mod p alors receive(@,L)
              sinon
                        receive(@,L)
                        send(@,L)
```

Le temps de la diffusion sera de $(p-1)(\beta + L\tau)$.

Le problème peut être immédiatement généralisé en la question de l'échange total : chaque machine P_k envoie un message à toutes les autres machines.

L'algorithme est le suivant :

```
All_to_all(my@,@,L) :
        q := my_num()
        p := tot_proc_num()
        @[q] := my@
        pour i=1 à p-1
           send(@[q-i+1],L) // receive(@[q-i],L
```

On fait tourner les messages en $p-1$ étapes de communication. Les liens de communication sont utilisés à plein régime. Le temps de la diffusion totale est le même que précédemment.

On peut aussi considéré la diffusion pipelinée : pour des longs messages, la diffusion peut être optimisée en utilisant l'idée du pipeline en divisant le messages en r morceaux. L'algorithme devient :

```
Broadcast(k,@,L) :
        q := my_num()
        p := tot_proc_num()
        si q=k alors
           pour i=0 à r-1 send(@[i],L/r)
        sinon
           si q=k-1 mod p alors
              pour i=0 à r-1 receive(@[i],L/r)
           sinon
                 receive(@[0],L/r)
                 pour i=0 à r-2
                   send(@[i],L/r) // receive(@[i+1],L/r
                 send(@[r],L/r)
```

Le temps de la diffusion est

$$(p-1)(\beta + \frac{L\tau}{r}) + (r-1)(\beta + \frac{L\tau}{r}) = (p-2+R)(\beta + \frac{L\tau}{r})$$

On peut vouloir trouver la découpe r qui minimise ce temps. En annulant la dérivée de la fonction, on obtient $r = \sqrt{\frac{(p-2)L\tau}{\beta}}$. Pour ce r optimal, on obtient un temps de diffusion total de

$$(\sqrt{\frac{p-2}{\beta}} + \sqrt{L\tau})^2$$

Pour de longs messages, on récupère un temps de diffusion de l'ordre de $L\tau$ qui est évidemment optimal, et qui ne dépend pas du nombre de machines !