Algorithmique I - Cours et Travaux Dirigés
L3, Ecole Normale Supérieure de Lyon

Anne Benoit

Septembre 2007

Table des matières

Préface

Les traditions changent et le cours d'algo n'est plus toujours le mercredi à la même heure, les enseignants rajeunissent, et le poly se débogue grâce aux gentils étudiants, TD-men et enseignants.

Donc voici la nouvelle version du poly remis à neuf, toujours pour satisfaire votre envie de savoir. Bien sûr il reste sans nul doute de nombreuses erreurs glissées dans ces pages, merci de me faire part de vos trouvailles par courrier électronique à *Anne.Benoit@ens-lyon.fr*.

Lyon, Juillet 2007
Anne Benoit

Préface d'Yves Robert

Ce polycopié rassemble les cours et travaux dirigés (avec corrigés) du module *Algorithmique* de l'ENS Lyon. A l'origine prévu pour la première année du Magistère d'Informatique, le module s'intègre désormais dans la troisième année de la Licence d'Informatique. Et dire que personne ne s'est rendu compte du changement !

Cela fait à peine dix ans que j'enseigne ce cours. A défaut de changer le contenu (pour faire quoi d'autre ?) ou d'utiliser autre chose que le tableau et la craie (idem ?), je change les irrésistibles traits d'humour qui font tout le charme de ces séances du Mercredi (l'horaire ne change pas non plus). Et j'use toute une batterie de TD-men and women, lesquels ont apporté leur contribution au fil des ans, construisant ou améliorant des séances de travaux dirigés. Je les remercie tous sincèrement, par ordre d'apparition : Odile Millet-Botta, Tanguy Risset, Alain Darte, Bruno Durand, Frédéric Vivien, Jean-Christophe Dubacq, Olivier Bodini, Daniel Hirschkoff, Matthieu Exbrayat, Natacha Portier, Emmanuel Hyon, Eric Thierry, Michel Morvan et Yves Caniou.

Sans aucune pression ou presque, Yves Caniou et Eric Thierry ont réussi à se motiver pour rassembler les TD. L'année précédente, j'avais rassemblé les cours. Enfin, quand on dit *rassembler*, c'est surtout les gentils étudiants-scribes qui rassemblent, en tapotant de leurs doigts agiles la quintessence de notre enseignement inégalable.

Ce polycopié est le concurrent le plus sérieux du Cormen dans le monde, ou du moins dans le septième arrondissement de Lyon ! Mais renonçant à de fabuleux droits d'auteur, l'équipe pédagogique (c'est nous) a décidé de mettre cet ouvrage à la libre disposition des nombreux étudiants assoiffés de savoir (c'est vous). *Enjoy !* Et merci de signaler erreurs et omissions par courrier électronique à *Yves.Robert@ens-lyon.fr*.

Lyon, Mai 2005
Yves Robert

Biblio

Voici quelques pointeurs bibliographiques (voir aussi les références données à la fin de chaque chapitre) :

Introduction to Algorithms de T. H. Cormen, C. E. Leiserson et R. L. Rivest [2]. L'ouvrage de référence par excellence, à acheter et conserver toute sa vie d'informaticien. Il se murmure qu'une deuxième édition est parue, avec un auteur de plus. Et une traduction française.

Computers and Intractability, a Guide to the Theory of NP-Completeness de M. R. Garey et D. S. Johnson [5]. Essentiellement pour son introduction à la NP-complétude au sens fort, et quelques jolies réductions. On revient toujours à son catalogue de problèmes NP-complets.

The art of Computer Programming , les trois tomes de Knuth [6], et bientôt quatre, pour leurs exercices incroyables

Sinon, j'aime bien :

- *Types de Données et Algorithmes*, le livre de Froidevaux, Gaudel et Soria [4], pour l'analyse fine des problèmes de tri
- le NP-compendium, maintenu sur le Web (`http://www.nada.kth.se/~viggo/problemlist/compendium.html`), pour les résultats d'approximation. Le livre qui correspond, *Complexity and Approximation*, de Ausiello et al. [1] est vraiment très complet, avec une bonne introduction aux schémas d'approximation.
- *Algorithms and Complexity*, le livre de Wilf [12], dont la première édition, épuisée, est disponible librement à l'url `http://www.cis.upenn.edu/~wilf/`. Une jolie introduction à la NP-complétude, avec une preuve concise du théorème de Cook, plein d'algorithmes, de l'humour, dans un fichier .pdf à télécharger absolument
- *Compared to what ? : an introduction to the analysis of algorithms*, le livre de Rawlins [8], qui contient une mine d'exercices originaux
- *Introduction to Graph Theory*, de West [11], mon livre préféré de graphes

Enfin, deux livres plus difficiles, à réserver aux plus aventureux : celui de Kozen [7], *The design and analysis of algorithms*, contient les notes de cours et exercices (certains corrigés) d'un cours de niveau avancé donné à Cornell, et celui de Vazirani [10], *Approximation algorithms*, dont le titre résume bien le contenu.

Chapitre 1

Introduction : calcul de x^n

Ce chapitre se base sur un petit exemple facile pour définir l'algorithmique et la notion de complexité d'un problème.

1.1 Énoncé du problème

On étudie le problème du calcul de x^n, étant donnés x et n (n étant un entier positif). Soulignons que x n'est pas nécessairement un nombre, il peut s'agir d'une matrice ou d'un polynôme à plusieurs indéterminées : si la multiplication a un sens, la division n'en a pas !

On pose $y_0 = x$, et on utilise la "règle du jeu" suivante : si j'ai déjà calculé $y_1, y_2, \ldots, y_{i-1}$, je peux calculer y_i comme produit de deux résultats précédents arbitraires :

$$y_i = y_j \cdot y_k, \text{ avec } 0 \leq j, k \leq i-1$$

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

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

1.2 Algorithme naïf

Considérons l'algorithme *naïf* suivant :

$$y_i = y_0 \cdot y_{i-1}$$

On a $y_{n-1} = x^n$, le coût est donc de $n-1$.

1.3 Méthode binaire

On trouve facilement un algorithme plus efficace :

$$x^n = \begin{cases} x^{n/2} \cdot x^{n/2} & \text{si } n \text{ est pair,} \\ x^{\lfloor n/2 \rfloor} \cdot x^{\lfloor n/2 \rfloor} \cdot x & \text{si } n \text{ est impair.} \end{cases}$$

On peut aussi formuler l'algorithme de la façon suivante. On écrit n en écriture binaire. Puis on remplace chaque "1" par SX et chaque "0" par S, et on enlève le premier SX (celui qui est à gauche). Le mot obtenu donne une façon de calculer x^n, en traduisant S par l'opération *mettre au carré* (squaring), et X par l'opération *multiplier par* x. Par exemple, pour $n = 23$ (n=10111),

la chaîne obtenue est SX S SX SX SX, en enlevant le premier SX, on obtient SSXSXSX. On calcule donc dans l'ordre x^2, x^4, x^5, x^{10}, x^{11}, x^{22}, et x^{23}.

La correction de l'algorithme se justifie facilement à partir des propriétés du système binaire. Le coût est de :

$$\lfloor \log n \rfloor + \nu(n) - 1,$$

où $\nu(n)$ représente le nombre de 1 dans l'écriture binaire de n. Bien sûr, comme dans tout ouvrage d'informatique qui se respecte, les logarithmes sont en base 2.

Cette méthode binaire n'est pas optimale : par exemple avec $n = 15$, on obtient la chaîne SXSXSX, d'où 6 multiplication alors que en remarquant que $15 = 3 \cdot 5$, on a besoin de 2 multiplications pour trouver $y = x^3$ ($y = (x \cdot x) \cdot x$) puis de 3 autres pour calculer $x^{15} = y^5$ (on applique la méthode binaire : y^2, y^4, y^5).

1.4 Méthode des facteurs

La méthode des facteurs est basée sur la factorisation de n :

$$x^n = \begin{cases} (x^p)^q & \text{si } p \text{ est le plus petit facteur premier de } n \ (n = p \times q), \\ x^{n-1} \cdot x & \text{si } n \text{ est premier.} \end{cases}$$

Exemple : $x^{15} = (x^3)^5 = x^3.(x^3)^4 = ...$ (5 multiplications).

Remarque : Avec les puissances de 2, cette méthode est identique à la méthode binaire.

Remarque : Cette méthode n'est pas optimale, par exemple pour $n = 33$ on a 7 multiplications avec la méthode des facteurs et seulement 6 avec la méthode binaire.
$x^{33} = (x^3)^{11} = x^3.(x^3)^{10} = x^3.((x^3)^2)^5 = x^3.y.y^4$ avec $y = (x^3)^2$.
$x^{33} = x.x^{2^5}$.

Remarque : Il existe une infinité de nombres pour lesquels la méthode des facteurs est meilleure que la méthode binaire (prendre $n = 15 \cdot 2^k$), et réciproquement (prendre $n = 33 \cdot 2^k$).

Arnaque : Il faut souligner que le coût de la recherche de la décomposition de n en facteurs premiers n'est pas pris en compte dans notre formulation. C'est pourtant nécessaire pour quantifier correctement le coût de la méthode des facteurs. Le problème est qu'on ne sait pas, à ce jour, trouver la décomposition en temps polynomial en n. Ce problème est NP-complet (notions de NP-complétude dans la suite du cours).

1.5 Arbre de Knuth

Une autre méthode consiste à utiliser *l'arbre de Knuth*, représenté Figure 1.1. Le chemin menant de la racine de l'arbre à n indique une séquence d'exposants permettant de calculer x^n de façon efficace.

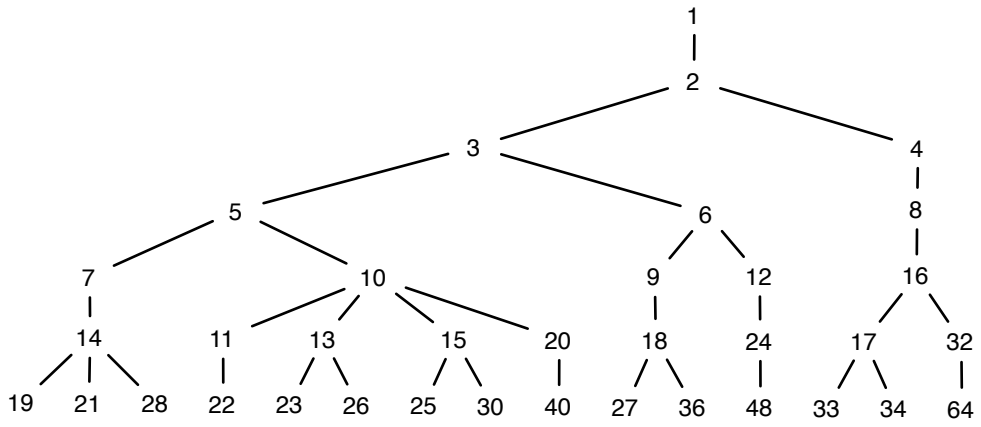


FIG. 1.1 – Les sept premiers niveaux de l'arbre de Knuth.

Construction de l'arbre Le $(k+1)$-ème niveau de l'arbre est défini à partir des k premiers niveaux de la façon suivante : prendre chaque nœud n du k-ème niveau, de gauche à droite, et les relier avec les nœuds

$$n+1, n+a_1, n+a_2, \ldots, n+a_{k-1} = 2n$$

(dans cet ordre), où $1, a_1, \ldots, a_{k-1} = n$ représente le chemin de la racine à n. On n'ajoutera pas un nœud si celui ci est déjà présent dans l'arbre.

Voici quelques statistiques dues à Knuth : les plus petits nombres pour lesquels la méthode n'est pas optimale sont $n = 77, 154, 233$. Le plus petit n pour lequel la méthode de l'arbre est supérieure à la fois à la méthode binaire et à celle des facteurs est $n = 23$. Le plus petit n pour lequel la méthode de l'arbre est moins bonne que celle des facteurs est $n = 19879 = 103 \cdot 193$; de tels cas sont rares : pour $n \leq 100000$, l'arbre bat les facteurs 88803 fois, fait match nul 11191 fois et perd seulement 6 fois (cf livre de Knuth).

1.6 Résultats sur la complexité

Théorème 1. $Opt(n) \geq \lceil \log n \rceil$

Preuve. Soit un algorithme permettant de calculer les puissances de x, avec pour tout i, $y_i = x^{\alpha(i)}$. Montrons par récurrence que $\alpha(i) \leq 2^i$. Pour la base, on a bien $y_0 = x$ d'où $1 \leq 2^0 = 1$. Soit i un entier, il existe j et k $(j, k < i)$ tels que $y_i = y_j \cdot y_k$. On a $\alpha(i) = \alpha(j) + \alpha(k)$, par l'hypothèse d'induction, $\alpha(j) \leq 2^j \leq 2^{i-1}$ (car $j \leq i-1$), et de même $\alpha(k) \leq 2^{i-1}$. On en déduit donc que $\alpha(i) \leq 2^{i-1} + 2^{i-1} = 2^i$.

Intuitivement, la preuve exprime qu'on ne peut pas faire mieux que doubler l'exposant obtenu à chaque étape de l'algorithme. □

Grâce au théorème précédent, et à l'étude de la méthode binaire, dont le nombre d'étapes est inférieur à $2 \log n$, on a le résultat suivant :

$$1 \leq \lim_{n \to \infty} \frac{Opt(n)}{\log n} \leq 2.$$

Théorème 2. $\lim_{n\to\infty} \frac{Opt(n)}{\log n} = 1$

Preuve. L'idée est d'améliorer la méthode binaire en appliquant cette méthode en base m. Posons $m = 2^k$, où k sera déterminé plus tard, et écrivons n en base m :

$$n = \alpha_0 m^t + \alpha_1 m^{t-1} + \cdots + \alpha_t,$$

où chaque α_i est un "chiffre" en base m, donc compris entre 0 et $m-1$. Puis on calcule tous les x^d, $1 \leq d \leq m-1$) par la méthode naïve, ce qui requiert $m-2$ multiplications. En fait, on n'a pas forcément besoin de toutes ces valeurs, seulement des x^{α_i}, mais comme on les calcule "au vol" on peut les calculer tous sans surcoût.

Ensuite on calcule successivement :

$$\begin{array}{rcl} y_1 & = & (x^{\alpha_0})^m \cdot x^{\alpha_1} \\ y_2 & = & (y_1)^m \cdot x^{\alpha_2} = x^{(\alpha_0 m+\alpha_1)m+\alpha_2} \\ \vdots & & \\ y_t & = & (y_{t-1})^m \cdot x^{\alpha_t} = x^n \end{array}$$

On effectue pour chaque ligne $k+1$ opérations (k élévations au carré pour calculer la puissance m-ème, et une multiplication), d'où le coût total du calcul de x^n :

$$t \cdot (k+1) + (m-2).$$

En remplaçant m par 2^k, et t par $\lfloor \log_m n \rfloor$, on trouve un coût total de

$$\lfloor \log_m n \rfloor (k+1) + 2^k - 2 \leq \frac{\log_2 n}{k}(k+1) + 2^k$$

(on rappelle que $\log_a b = \log_x b / \log_x a$). Il suffit donc de prendre k tendant vers l'infini, pour que $(k+1)/k$ tende vers 1, et tel que $2^k = o(\log n)$, par exemple $k = \lfloor 1/2 \log(\log n) \rfloor$ (on aura alors $2^k \leq \sqrt{\log n}$). □

Remarquons que cette technique est tout de même assez compliquée, tout cela uniquement pour gagner un facteur 2 par rapport à la méthode binaire.

1.7 Exercices

Exercice 1.7.1. *Le grand saut*

Le problème est de déterminer à partir de quel étage d'un immeuble, sauter par une fenêtre est fatal. Vous êtes dans un immeuble à n étages (numérotés de 1 à n) et vous disposez de k étudiants. Il n'y a qu'une opération possible pour tester si la hauteur d'un étage est fatale : faire sauter un étudiant par la fenêtre. S'il survit, vous pouvez le réutiliser ensuite, sinon vous ne pouvez plus.

Vous devez proposer un algorithme pour trouver la hauteur à partir de laquelle un saut est fatal (renvoyer $n+1$ si on survit encore en sautant du n-ème étage) en faisant le minimum de sauts.

1 - Si $k \geq \lceil \log_2(n) \rceil$, proposer un algorithme en $\mathcal{O}(\log_2(n))$ sauts.

2 - Si $k < \lceil \log_2(n) \rceil$, proposer un algorithme en $\mathcal{O}(k + \frac{n}{2^{k-1}})$ sauts.

3 - Si $k = 2$, proposer un algorithme en $\mathcal{O}(\sqrt{n})$ sauts.

Correction.

1 - La complexité en $O(\log_2(n))$ qui est indiquée nous aiguille vers une dichotomie. En effet, en supposant que l'on a $k \geq \lceil \log_2(n) \rceil$, on obtient le résultat sur les étages de i à j en jetant un étudiant depuis le $n^{\text{ème}}$ étage, où $n = j - i/2$, puis en itérant le procédé sur les étages de i à $n-1$ si la chute à été fatale, et sur les étages de n à j dans le cas contraire. La méthode dichotomique nous garantit alors que l'on obtient le bon résultat (lorsqu'il ne reste plus qu'un seul étage, c'est-à-dire lorsque l'on calcule pour les étages de $i = j$), et ce avec une complexité logarithmique dans le pire des cas.

2 - Puisqu'ici on ne dispose que de $k < \lceil \log_2(n) \rceil$ étudiants, on ne peut pas appliquer directement la méthode dichotomique proposée précédemment. Cependant, on va remédier au problème de manière simple en appliquant une recherche dichotomique avec $k-1$ étudiants, de manière à délimiter un intervalle d'étages dans lequel se trouve l'étage recherché. On se sert alors du dernier étudiant restant pour parcourir l'intervalle de façon linéaire, donc en le jetant de chaque étage en partant du plus bas de l'intervalle, jusqu'au plus haut. Après avoir jeté les $k-1$ premiers étudiants, si l'on n'a pas encore trouvé le bon étage, il reste exactement $n/2^{k-1}$ étages dans l'intervalle de recherche, d'où une complexité dans le pire des cas en $O(k + n/2^{k-1})$ sauts.

3 - Dans le cas particulier où l'on a $k = 2$, on ne veut pas avoir à tester chaque étage de façon linéaire, c'est pourquoi on va reprendre à notre compte les idées précédentes, et notamment celle qui consiste à délimiter un intervalle de recherche. Nous découpons donc l'ensemble des étages en "tranches" de $\sqrt{n}$ étages, avant de jeter le premier étudiant de chacun des étages de début de tranche. Lorsque l'étudiant y laisse sa peau, on se ramène au dernier étage n testé qui ne soit pas fatal, et on n'a plus qu'à parcourir de manière linéaire l'intervalle allant de l'étage $n+1$ à l'étage fatal trouvé précédemment. On a ainsi deux séries d'essais en $O(\sqrt{n})$, et donc une complexité finale dans le pire des cas également en $O(\sqrt{n})$ sauts.

Exercice 1.7.2. *Cherchez la star*

Dans un groupe de n personnes (numérotées de 1 à n pour les distinguer), une *star* est quelqu'un qui ne connait personne mais que tous les autres connaissent. Pour démasquer une star, s'il en existe une, vous avez juste le droit de poser des questions, à n'importe quel individu i du groupe, du type "est-ce que vous connaissez j ?" (noté "$i \to j$?"), on suppose que les individus répondent la vérité. On veut un algorithme qui trouve une star, s'il en existe, ou sinon qui garantit qu'il n'y a pas de star dans le groupe, en posant le moins de questions possibles.

1 - Combien peut-il y avoir de stars dans le groupe ?

2 - Ecrire le meilleur algorithme que vous pouvez et donner sa complexité en nombre de questions (on peut y arriver en $\mathcal{O}(n)$ questions).

3 - Donner une borne inférieure sur la complexité (en nombre de questions) de tout algorithme résolvant le problème. ((*Difficile*) prouver que la meilleure borne inférieure pour ce problème est $3n - \lfloor \log_2(n) \rfloor - 3$).

Correction.

1 - Il ne peut y avoir qu'une seule star dans le groupe, puisque s'il y en a une, elle ne connait personne, et donc tous les autres sont inconnus d'au moins une personne, et ne peuvent donc être eux aussi des stars.

2 - Lorsqu'on effectue le test $''i \to j\,?\,''$, c'est-à-dire lorsque l'on cherche à savoir si la personne i connaît la personne j, on obtient le résultat suivant :

– si oui, alors i n'est pas une star, mais j en est potentiellement une.
– si non, alors j n'est pas une star, mais i en est potentiellement une.

L'algorithme consiste alors a parcourir le tableau des personnes une fois, en gardant en memoire à chaque instant la personne i qui est jusqu'ici reconnue par tous, tandis qu'au $j^{\text{ème}}$ test, toutes les autres personnes, dont les indices sont les $k < j$ (avec $k \neq i$, bien sûr) ne peuvent pas des stars. Cet algorithme s'écrit donc de la manière suivante :

$i \leftarrow 1$ **et** $j \leftarrow 2$
tant que $j \leq n$ **faire**
  ▶ **si** $''i \rightarrow j''$ **alors faire** $j \leftarrow j + 1$
  sinon faire $i \leftarrow j$ **et** $j \leftarrow j + 1$
$i_{star} \leftarrow$ *vrai* **et** $k \leftarrow 1$
tant que ($k \leq n$ **et** i_{star}) **faire**
  ▶ **si** $''k \rightarrow i''$ **alors faire** $k \leftarrow k + 1$
  sinon faire $i_{star} \leftarrow$ *faux*
si i_{star} **alors faire retourner** *" i est la star "*
sinon faire retourner *" il n'y a pas de star "*

3 - En observant notre algorithme, on constate que l'on peut utiliser comme borne inférieure pour le nombre de questions la valeur $3n-2$, puisque l'on fait ici deux boucles sur $n-1$ personnes et une boucle sur l'ensemble des n personnes. Pour ce qui est de la borne inférieure optimale, il s'agit d'une question difficile, dont on n'explicitera pas la solution ici.

Exercice 1.7.3. *Bricolage*

Dans une boite à outils, vous disposez de n écrous de diamètres tous différents et des n boulons correspondants. Mais tout est mélangé et vous voulez appareiller chaque écrou avec le boulon qui lui correspond. Les différences de diamètre entre les écrous sont tellement minimes qu'il n'est pas possible de déterminer à l'œil nu si un écrou est plus grand qu'un autre. Il en va de même avec les boulons. Par conséquent, le seul type d'opération autorisé consiste à essayer un écrou avec un boulon, ce qui peut amener trois réponses possibles : soit l'écrou est strictement plus large que le boulon, soit il est strictement moins large, soit ils ont exactement le même diamètre.

1 - Ecrire un algorithme simple en $\mathcal{O}(n^2)$ essais qui appareille chaque écrou avec son boulon.

2 - Supposons qu'au lieu de vouloir appareiller tous les boulons et écrous, vous voulez juste trouver le plus petit écrou et le boulon correspondant. Montrer que vous pouvez résoudre ce problème en moins de $2n - 2$ essais.

3 - Prouver que tout algorithme qui appareille tous les écrous avec tous les boulons doit effectuer $\Omega(n \log n)$ essais dans le pire des cas.

Problème ouvert : proposer un algorithme en $o(n^2)$ essais pour résoudre ce problème.

Correction.

1 - Algorithme

Pour appareiller les boulons et les écrous, il suffit de prendre un boulon arbitrairement, de le tester avec tous les écrous.On trouve alors le bon en au plus n tests et il suffit alors de recommencer avec tous les autres boulons.On obtient donc le résultat en au plus $\frac{n(n-1)}{2} = O(n^2)$ tests.

2 - Le plus petit écrou et son boulon

Le principe est de numéroter les boulons et les écrous de 1 à n, de manière arbitraire, et de faire progresser des compteurs (par exemple i et j) pour marquer le minimun courant dans l'une des catégories, et la structure en cours de test dans l'autre.

début
  while *(i ≤ n) && (j ≤ n)* **do**
    si *i=j=n* **alors** sortir de la boucle ;
    si *ecrou.i = boulon.j* **alors** s'en souvenir et faire $i := i + 1$;
    si *ecrou.i < boulon.j* **alors** $j := j + 1$; $min = ecrou$;
    si *ecrou.i > boulon.j* **alors** $i := i + 1$; $min = boulon$;
fin

A la fin de cette boucle,

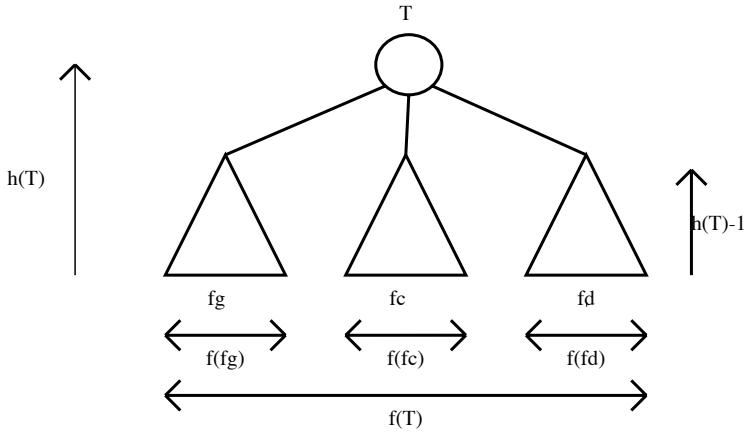
- si min=écrou, l'écrou i est le plus petit.
- si min=boulon, le boulon j est le plus petit. Et dans les deux cas, on sait déjà quel est l'élément qui correspond : en effet, le compteur de l'autre catégorie vaut n+1 (ou n dans un cas spécial),et on a donc déjà rencontré le minimum. la seule raison possible pour laquelle il n'a pas été conservé est donc qu'il ait été testé avec l'autre minimum. Pour le cas spécial ou i=j=n, le dernier test est inutile : en effet, on sait que l'un des deux, est minimum, et que une fois le boulon minimum atteind, j reste fixe : d'où le boulon n est minimum, et son homologue a soit déjà été trouvé, soit est l'écrou restant.

Cette boucle effectue donc au plus $2 * n - 2$ tests.

3 - Algorithme d'appareillage des écrous et de leur boulon

On note $f(T)$ le nombre de feuilles de l'arbre et $h(T)$ sa hauteur. Si T est un arbre ternaire $h(T) \geq \lceil log_3 f(T) \rceil$. Par induction sur la hauteur de T :

- Pour $h(T) = 0$ l'arbre à une feuille donc on a bien $h(T) \geq \lceil log_3 f(T) \lceil$.
- Pour $h(T) > 0$ un arbre ternaire a trois fils, le fils gauche fg, le fils central fc, et le fils droit fd de hauteur inférieur à $h(T) - 1$.On suppose que $h(T) \geq \lceil log_3 f(T) \rceil$ est vrai pour $h(T) \leq k$ k étant fixé on veut démontrer que cette propriété est vrai aussi pour $h(T) = k + 1$. Les feuilles d'un arbre sont aussi celles de ses fils. Donc



$$f(T) = f(fd) + f(fg) + f(fd)$$

de plus :

$$h(T) \leq h(fd) + 1$$

$$h(T) \leq h(fg) + 1$$

$$h(T) \leq h(fc) + 1$$

or par induction comme $h(T) = k + 1$, $h(fg) \leq k$, $h(fg) \leq k$,et $h(fg) \leq k$ on a :

$$k = h(fc) \leq \lceil log_3 f(fc) \rceil$$

$$h(fd) \leq \lceil log_3 f(fd) \rceil$$

$$h(fg) \leq \lceil log_3 f(fg) \rceil$$

Donc $f(fc), f(fg), f(fd) \leq 3^k$ or :

$$f(T) = f(fc) + f(fg) + f(fd)$$

donc :

$$f(T) \leq 3 \times 3^k = 3^{k+1}$$

D' où on en déduit que :

$$h(T) \geq log_3 f(T)$$

Il y a $!n$ agencements boulons-écrous possibles donc l'arbre de décision, qui est ternaire a pour hauteur : $log_3 !n \sim n log_3 n$, donc la complexité dans le pire cas de l'algo est de : $\Omega(n \times log n)$.

1.8 Références bibliographiques

La présentation du cours s'inspire de Knuth [6]. Les exercices *Le grand saut* et *Bricolage* sont tirés de Rawlins [8].

Chapitre 2

Diviser pour régner

2.1 Algorithme de Strassen

Calculons un produit de matrices :

$$\begin{pmatrix} r & s \\ t & u \end{pmatrix} = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \cdot \begin{pmatrix} e & f \\ g & h \end{pmatrix}$$

L'algorithme classique calcule en $Add(n) = n^2(n-1)$ additions et $Mult(n) = n^3$ multiplications. En effet, il y a n^2 coefficients à calculer, chacun correspondant un produit scalaire de taille n, donc avec n multiplications, $n-1$ additions, et une affectation. Peut-on faire mieux ?[1] V. Strassen répond que oui : soient

$$\begin{aligned} p_1 &= a(g-h) \\ p_2 &= (a+b)h \\ p_3 &= (c+d)e \\ p_4 &= d(f-e) \\ p_5 &= (a+d)(e+h) \\ p_6 &= (b-d)(f+h) \\ p_7 &= (a-c)(e+g) \end{aligned}$$

alors on peut écrire

$$\begin{aligned} r &= p_5 + p_4 - p_2 + p_6 \\ s &= p_1 + p_2 \\ t &= p_3 + p_4 \\ u &= p_5 + p_1 - p_3 - p_7 \end{aligned}$$

Comptons maintenant les opérations :

Classique	*Strassen*
$Mult(2) = 8$	$Mult(2) = 7$
$Add(2) = 4$	$Add(2) = 18$

On a gagné une multiplication, mais en perdant 14 additions ; donc, pour des matrices $2x2$, c'est un bilan négatif. Par contre, il est remarquable que l'opération ne nécessite pas la commutativité de la multiplication. On peut donc l'utiliser avec des matrices de taille n paire.

[1] Dans les temps anciens de l'informatique, il était surtout intéressant de diminuer le nombre de multiplications, quitte à augmenter le nombre d'additions. L'architecture pipelinée des processeurs actuels permet de réaliser, en moyenne, une addition ou une multiplication par temps de cycle.

Supposons donc $n = 2m$ pair, et utilisons la méthode précédente une seule fois, en partitionnant chaque matrice de départ en quatre sous-blocs de taille $m \times m$. On effectuera m^3 multiplications pour chaque p_i, d'où un total de $Mult(n) = 7m^3 = 7n^3/8$ multiplications. Pour les additions, il y a deux sources : les additions effectuées dans chaque produit p_i, donc au nombre de $7m^2(m-1)$, et les additions pour former les 18 matrices auxiliaires, au nombre de $18m^2$. D'où $Add(n) = 7m^3 + 11m^2 = 7n^3/8 + 11n^2/4$. Asymptotiquement, le terme dominant est en $7n^3/8$ pour $Mult(n)$ comme pour $Add(n)$, et la nouvelle méthode est gagnante pour n assez grand. La raison profonde est la suivante : une multiplication de deux matrices de taille n requiert $O(n^3)$ opérations, alors que leur addition n'en demande que $O(n^2)$. Pour n grand, les additions sont "gratuites" en face des multiplications. Ce qui n'était pas le cas pour les nombres réels.

L'algorithme de Strassen est l'application récursive de la décomposition précédente. On considère le cas où n est une puissance de 2, i.e. $n = 2^s$. Si n n'est pas une puissance de 2, on étend les matrices avec des 0 à la puissance de 2 supérieure :

$$\begin{pmatrix} X & 0 \\ 0 & 0 \end{pmatrix}$$

et on remplacera dans les formules qui suivent $\log n$ par $\lceil \log n \rceil$.

Prenons donc $n = 2^s$. On fonctionne de manière récursive : on refait le même découpage pour chacun des produits de matrice p_i. On s'arrêtera quand on arrive à des matrices de taille 1, ou mieux, à la taille pour laquelle la méthode est plus coûteuse que la méthode classique ($n \approx 32$).

Soient :

- $M(n)$ = nombre de multiplications réalisées par l'algorithme de Strassen pour multiplier 2 matrices de taille n
- $A(n)$ = nombre d'additions réalisées par l'algorithme de Strassen pour multiplier 2 matrices de taille n

On a :

$$\begin{cases} M(1) = 1 \\ M(n) = 7 \times M(n/2) \end{cases} \Longrightarrow M(n) = 7^s = 7^{log_2(n)} = n^{log_2(7)}$$

Comme précédemment, les additions viennent de 2 sources : les additions liées aux additions de matrices pour la construction des termes p_i et les additions effectuées pour les multiplications de matrices (appels récursifs). On a donc :

$$\begin{cases} A(1) = 0 \\ A(n) = 7 \times A(n/2) + 18 \times (n/2)^2 \end{cases} \Longrightarrow A(n) = 6 \times (n^{log_2(7)} - n^2)$$

(on verra Section 2.4 comment résoudre cette récurrence). On voit que l'ordre de grandeur du coût des calculs n'est plus en n^3, mais seulement en $n^{log_2 7} \approx n^{2.8}$.

L'algorithme de Strassen n'est pas souvent utilisé car il introduit des problèmes d'instabilité numérique. Notons enfin qu'il existe des algorithmes de complexité meilleure que celle de Strassen, et que le problème de déterminer la complexité du produit de deux matrices est encore ouvert. La seule borne inférieure connue est en $O(n^2)$: il faut bien toucher chaque coefficient au moins une fois. Le meilleur algorithme connu à ce jour est en $O(n^{2.376})$.

2.2 Produit de deux polynômes

Le but est de multiplier deux polynômes. On notera n-polynômes les polynômes de degré strictement inférieur à n, donc avec n coefficients. Soient :

- P : n-polynôme : $\sum_{i=0}^{n-1} a_i X^i$
- Q : n-polynôme : $\sum_{i=0}^{n-1} b_i X^i$
- $R = P \times Q =?$: $(2n-1)$-polynôme

Soient :

- $M(n)$ = nombre de multiplications réalisées par l'algorithme pour multiplier deux n-polynômes
- $A(n)$ = nombre d'additions réalisées par l'algorithme pour multiplier deux n-polynômes

Avec l'algorithme usuel de multiplication de n-polynômes :

$$M(n) = n^2 \text{ et } A(n) = n^2 - \underbrace{(2n-1)}_{affectations} = (n-1)^2$$

Pour compter les affectations, on note que l'on doit faire une affectation pour chaque coefficient du polynôme R, ce sont des additions en moins.

On suppose n pair, $n = 2m$

$$\left.\begin{array}{l} P = P_1 + X^m \times P_2 \\ Q = Q_1 + X^m \times Q_2 \end{array}\right) \text{ avec } P_1,\ P_2,\ Q_1,\ Q_2 \ m\text{-polynômes}$$

Soient :

$$\begin{array}{l} R_1 = P_1 \times Q_1 \\ R_2 = P_2 \times Q_2 \\ R_3 = (P_1 + P_2) \times (Q_1 + Q_2) \end{array}$$

On a alors : $R = R_1 + (R_3 - R_2 - R_1) \times X^m + R_2 \times X^{2m}$, et R_1, R_2, R_3 sont des $(n-1)$-polynômes.

Quel est l'intérêt ? On ne réalise que 3 multiplications de polynômes de taille moitié. Les additions sont de deux types : les additions de polynômes, et les additions liées aux multiplications de m-polynômes.

On suppose maintenant que $n = 2^s$ et on applique l'algorithme de manière récursive.

$$\left\{\begin{array}{l} M(1) = 1 \\ M(n) = 3 \times M(n/2) \end{array}\right. \Longrightarrow M(n) = 3^s = n^{log_2(3)}$$

$$\left\{\begin{array}{l} A(1) = 0 \\ A(n) = \underbrace{3 \times A(n/2)}_{appels\ recursifs} + \underbrace{2 \times n/2}_{pour\ avoir\ R_3} + \underbrace{2 \times (n-1)}_{pour\ avoir\ R_3 - R_2 - R_1} + \underbrace{(n-2)}_{construction\ de\ R} \end{array}\right.$$

En effet, pour la construction de R :

$$R = R_1 + (R_3 - R_1 - R_2) \times X^m + R_2 \times X^{2m} = \underbrace{\sum_{i=0}^{n-2} r_i^1 \times X^i}_{X^0 \to X^{n-2}} + \sum_{i=0}^{n-2} z_i \times X^{i+n/2} + \underbrace{\sum_{i=0}^{n-2} r_i^2 \times X^{i+n}}_{X^n \to X^{2n-2}}$$

Tous les termes du terme du milieu (sauf x^{n-1}) s'additionnent aux termes de gauche et de droite, il faut donc faire $n-2$ additions. D'où :

$$\left\{\begin{array}{l} A(1) = 0 \\ A(n) = 3 \times A(n/2) + 4 \times n - 4 \end{array}\right. \Longrightarrow A(n) = 6n^{log_2(3)} - 8n + 2$$

(on verra Section 2.4 comment résoudre cette récurrence)

Remarque Le meilleur algorithme de multiplication de polynômes est en $O(n \times log(n))$, il est obtenu par transformée de Fourrier rapide (FFT).

$$\left[P, Q \underbrace{\longrightarrow}_{evaluation} \underbrace{P(x_i), Q(x_i)}_{en\ 2n\ points} \longrightarrow P(x_i) \times Q(x_i) \underbrace{\longrightarrow}_{interpolation} P \times Q \right]$$

L'évaluation et l'interpolation sont réalisées en n^2 par la méthode classique, en utilisant Newton et Lagrange. Pour la FFT, on évalue les polynômes en les racines complexes de l'unité.

2.3 Master theorem

Diviser pour règner : On considère un problème de taille n, qu'on découpe en a sous-problèmes de taille n/b permettant de résoudre le problème. Le coût de l'algorithme est alors :

$$\begin{cases} S(1) = 1 \\ S(n) = a \times S(n/b) + \underbrace{Reconstruction(n)}_{c \times n^\alpha\ en\ general} \end{cases}$$

	a	b	α	c
Strassen	7	2	2	18
Polynomes	3	2	1	4

$$S(n) = a \times S(n/b) + R(n) = a^2 \times S(n/b^2) + a \times R(n/b) + R(n) = \ldots$$

On pose $n = b^k$ $(k = \log_b(n))$, on a alors :

$$S(n) = \underbrace{a^k}_{n^{log_b(a)}} \times S(1) + \underbrace{\sum_{i=0}^{k-1} a^i \times R(n/b^i)}_{\sum} \quad avec\ R(n) = c \times n^\alpha$$

et :

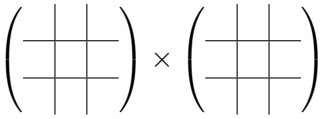
$$\sum = c \times n^\alpha \sum_{i=0}^{k-1} (a/b^\alpha)^i$$

On distingue alors plusieurs cas :

1. $(a > b^\alpha) : \sum \sim n^\alpha \times (\frac{a}{b^\alpha})^k \sim a^k \Longrightarrow S(n) = O(n^{log_b(a)})$
2. $(a = b^\alpha) : \sum \sim k \times n^\alpha \Longrightarrow S(n) = O(n^\alpha \times log(n))$
3. $(a < b^\alpha) : \sum \sim c \times n^\alpha \times \frac{1}{1 - \frac{a}{b^\alpha}} \Longrightarrow S(n) = O(n^\alpha)$

La preuve complète est dans le Cormen.

Retour sur Strassen :
A la lumière de ce qui précède, et si on découpait les matrices en 9 blocs de taille $n/3$ au lieu de 4 blocs de taille $n/2$?



On a :

a	b	α	c
a	3	2	

Pour que l'algorithme soit plus intéressant que Strassen, il faut que :

$$\begin{array}{rrcl} & log_3(a) & < & log_2(7) \\ \Rightarrow & a & < & e^{log_2(7)\times log_2(3)} \\ \Rightarrow & a & < & 7^{log_2(3)} \approx 21,8 \end{array}$$

C'est un problème ouvert : on connaît une méthode avec $a = 23$ mais pas avec $a = 21$!

2.4 Résolution des récurrences

2.4.1 Résolution des récurrences homogènes

$$\begin{cases} p_0 \times s_n + p_1 \times s_{n-1} + \cdots + p_k \times s_{n-k} = 0 \\ p_i \ constantes \end{cases}$$

Soit $P = \sum_{i=0}^{k} p_i \times X^{k-i}$. On cherche les racines de P. Si les racines de P sont distinctes :

$$s_n = \sum_{i=0}^{k} c_i \times {r_i}^n$$

Sinon, si q_i est l'ordre de multiplicité de r_i

$$s_n = \sum_{i=0}^{l} \underbrace{P_i(n)}_{polynome\ de\ degre\ q_i - 1} \times {r_i}^n$$

2.4.2 Résolution des récurrences avec second membre

On note E l'opérateur de décalage : $E\{s_n\} = \{s_{n+1}\}$.
On définit les opérations suivantes sur les suites :

$$c.\{s_n\} = \{c.s_n\} \quad (2.1)$$

$$(E_1 + E_2)\{s_n\} = E_1\{s_n\} + E_2\{s_n\} \quad (2.2)$$

$$(E_1 E_2)\{s_n\} = E_1(E_2\{s_n\}) \quad (2.3)$$

$$\left(ex : \begin{array}{l} (E-3)\{s_n\} = \{s_{n+1} - 3s_n\} \\ (2+E^2)\{s_n\} = \{2s_n + s_{n+2}\} \end{array} \right)$$

$P(E)$ annule $\{s_n\}$ si $P(E)\{s_n\} = \overline{0}$ ex :

suite	*annulateur*
$\{c\}$	$E-1$
$\{Q_k(n)\}$	$(E-1)^{k+1}$
$\{c^n\}$	$E-c$
$\{c^n \times Q_k(n)\}$	$(E-c)^{k+1}$

où $Q_k(n)$ est un polynôme de degré k. En effet, il suffit de montrer la dernière relation, par récurrence sur k :

$$\begin{array}{rcl} (E-c)^{k+1} \underbrace{\{c^n \times Q_k(n)\}}_{\{c^n \times (a_0 n^k + Q_{k-1}(n))\}} & = & (E-c)^k \underbrace{[(E-c)\{c^n(a_0 n^k + Q_{k-1}(n))\}]}_{\{c^{n+1}(a_0(n+1)^k + Q_{k-1}(n+1)) - c^{n+1}(a_0 n^k + Q_{k-1}(n))\}} \\ & = & (E-c)^k[c^{n+1} \times R_{k-1}(n)] \\ & = & \overline{0} \end{array}$$

(par hypothèse de récurrence)

Résolution pour les additions dans l'algorithme de Strassen :

$$A(n) = 7 \times A(n/2) + 18 \times \frac{n^2}{4}$$

On a $n = 2^s$, on pose $A_s = A(2^s)$

$$A_{s+1} = 7 \times A_s + 18 \times \frac{(2^{s+1})^2}{4} = 7 \times A_s + 18 \times 4^s$$

$$(E-4)\ \underbrace{(E-7)\{A_s\}}_{A_{s+1}-7A_s=18\times 4^s} = \overline{0}$$

$$\Rightarrow A_s = k_1 \times 7^s + k_2 \times 4^s$$

avec :

$$\begin{aligned} A_0 &= 0 \\ A_1 &= 18 \end{aligned}$$

On en déduit les valeurs de k_1 et k_2 données plus haut.

Résolution pour les additions dans l'algorithme de multiplication de polynômes :

$$\begin{aligned} A(n) &= 3 \times A(n/2) + 4n - 4 \\ A_s &= 3 \times A_{s-1} + 4 \times 2^s - 4 \end{aligned}$$

D'où : $(E-1)(E-2)(E-3)\{A_s\} = \overline{0}$

$$\Rightarrow A_s = k_1 \times 3^s + k_2 \times 2^s + k_3$$

avec :

$$\begin{aligned} A_0 &= 0 \\ A_1 &= 4 \\ A_2 &= 24 \end{aligned}$$

On en déduit les valeurs de $k_1 = 6$, $k_2 = -8$ et $k_3 = 2$ données plus haut.

2.5 Multiplication et inversion de matrices

Soient :
- $M(n)$ = coût de la multiplication de 2 matrices d'ordre n
- $I(n)$ = coût de l'inversion d'une matrice d'ordre n
- Hypothèses :
  - $O(n^2) \leq \begin{matrix} M(n) \\ I(n) \end{matrix} \leq O(n^3)$
  - $M(n)$ *et* $I(n)$ *croissants*

Théorème 3. *$M(n)$ et $I(n)$ ont le même ordre de grandeur.*

Preuve. On montre que chaque opération est au moins aussi "difficile" que l'autre :

La multiplication est au moins aussi complexe que l'inversion

On veut multiplier les matrices A et B de taille n. Soit Z de taille $3n$ suivante :

$$Z = \begin{pmatrix} I & A & 0 \\ 0 & I & B \\ 0 & 0 & I \end{pmatrix}$$

$$\Rightarrow Z^{-1} = \begin{pmatrix} I & -A & A.B \\ 0 & I & -B \\ 0 & 0 & I \end{pmatrix}$$

D'où : $M(n) \leq I(3n)$

L'inversion est au moins aussi complexe que la multiplication

On procède en deux étapes :

Si A est symétrique définie positive

$$A = \begin{pmatrix} B & {}^tC \\ C & D \end{pmatrix}$$

Soit $S = D - C.B^{-1}.\ {}^tC$ (Shur complement).

$$A^{-1} = \begin{pmatrix} B^{-1} + B^{-1}.\ {}^tC.S^{-1}.C.B^{-1} & -B^{-1}.\ {}^tC.S^{-1} \\ -S^{-1}.C.B^{-1} & S^{-1} \end{pmatrix}$$

Il suffit de construire :

$$B^{-1}, C.B^{-1}, (C.B^{-1}).\ {}^tC, S^{-1}, S^{-1}.(C.B^{-1}),\ {}^t(C.B^{-1}).(S^{-1}.C.B^{-1})$$

d'où : $I(n) = 2 \times I(n/2) + 4 \times M(n) + O(n^2) = O(M(n))$

Cas général On pose $B = {}^tA \times A$. On veut calculer A^{-1}, et on sait calculer B^{-1} car B est symétrique définie positive.

$$I = B^{-1}.B = B^{-1}.({}^tA.A) = (B^{-1}.\ {}^tA).A \Rightarrow A^{-1} = B^{-1}.\ {}^tA$$

D'où : $I(n) = 2 \times M(n) + O(M(n)) = O(M(n))$

□

2.6 Exercices

Exercice 2.6.1. *Matrices de Tœplitz*

Une *matrice de Tœplitz* est une matrice $n \times n$ $(a_{i,j})$ telle que $a_{i,j} = a_{i-1,j-1}$ pour $2 \leq i, j \leq n$.

1 - La somme de deux matrices de Tœplitz est-elle une matrice de Tœplitz ? Et le produit ?

2 - Trouver un moyen d'additionner deux matrices de Tœplitz en $\mathcal{O}(n)$.

3 - Comment calculer le produit d'une matrice de Tœplitz $n \times n$ par un vecteur de longueur n ? Quelle est la complexité de l'algorithme ?

Correction.

1 – Par linéarité, la somme de deux Tœplitz reste une Tœplitz. Ce n'est pas vrai pour le produit. Par exemple,

$$\begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix} \times \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 2 \end{pmatrix}$$

2 – On n'additionne que les premières lignes et les premières colonnes, ce qui fait $2n-1$ opérations.

3 – On ne considère que les matrices de taille $2^k \times 2^k$.

On décompose la matrice en blocs de taille $n = 2^{k-1}$

$$\mathbf{M} \times \mathbf{T} = \begin{pmatrix} A & B \\ C & A \end{pmatrix} \times \begin{pmatrix} X \\ Y \end{pmatrix}$$

On pose :

$$\begin{aligned} U &= (C+A)X \\ V &= A(Y-X) \\ W &= (B+A)Y \end{aligned}$$

et on calcule :

$$\mathbf{M} \times \mathbf{T} = \begin{pmatrix} W-V \\ U+V \end{pmatrix}$$

Calcul de la complexité : On note respectivement $M(n)$ et $A(n)$ le nombre de multiplications et d'additions pour un produit de matrices $n \times n$.

$$\begin{aligned} M(2^k) &= 3M(2^{k-1}) \\ M(1) &= 1 \end{aligned}$$

$$\begin{aligned} A(2^k) &= 3A(2^{k-1}) + 2(2^k - 1) + 3.2^{k-1} \\ A(1) &= 0 \end{aligned}$$

On résoud les récurrences :
on pose $M_s = M(2^s)$, et on pose $A_s = A(2^s)$.

le calcul pour les multiplications :

$$\begin{cases} M_s = 3M_{s-1} \\ M_0 = 1 \end{cases}$$

$$(E-3)M_s = \bar{0}; M_s = 3^{\log n} = n^{\log 3}$$

puis le calcul pour les additions :

$$\begin{cases} A_s = 3A_{s-1} + 7.2^{s-1} - 2 \\ A_0 = 0 \end{cases}$$

$$\begin{aligned} &(E-3)(E-2)(E-1)\{A_s\} = \bar{0} \\ &A_s = i3^s + j2^s + l \\ &A_0 = A(1) = 0 \longrightarrow i + j + l = 0 \\ &A_1 = A(2) = 5 \longrightarrow 3i + 2j + l = 5 \\ &A_2 = A(4) = 27 \longrightarrow 9i + 4j + l = 27 \\ &\text{D'où } i = 6, j = -7, l = 1 \end{aligned}$$

Exercice 2.6.2. *Recherche d'un élément majoritaire*

Soit E une liste de n éléments rangés dans un tableau numéroté de 1 à n. On suppose que la seule opération qu'on sait effectuer sur les éléments est de vérifier si deux éléments sont égaux ou non. On dit qu'un élément $x \in E$ est *majoritaire* si l'ensemble $E_x = \{y \in E | y = x\}$ a strictement plus de $n/2$ éléments. Sauf avis contraire, on supposera que n est une puissance de 2. On s'intéressera à la complexité dans le pire des cas.

1 - Algorithme naïf

Écrire un algorithme calculant le cardinal de c_x de E_x pour un x donné. En déduire un algorithme pour vérifier si E possède un élément majoritaire. Quelle est la complexité de cet algorithme ?

2 - Diviser pour règner

2.1- Donner un autre algorithme récursif basé sur un découpage de E en deux listes de même taille. Quelle est sa complexité ?

2.2 - Donner un algorithme similaire (en précisant sa complexité) dans le cas général où n n'est pas une puissance de 2.

3 - Encore mieux

Pour améliorer l'algorithme précédent, on va se contenter dans un premier temps de mettre au point un algorithme possédant la propriété suivante :

- soit l'algorithme garantit que E ne possède pas d'élément majoritaire,
- soit l'algorithme fournit un entier $p > n/2$ et un élément x tels que x apparaisse au plus p fois dans E et tout élément autre que x apparaît au plus $n - p$ fois dans E.

3.1 - Donner un algorithme récursif possédant cette propriété. Quelle est sa complexité ?

3.2 - Même question quand n n'est pas une puissance de 2.

3.3 - En déduire un algorithme efficace vérifiant si E possède un élément majoritaire.

4 - Encore encore mieux

On change la manière de voir les choses. On a un ensemble de n balles et on cherche le cas échéant s'il y a une couleur majoritaire parmi les balles.

4.1 - Supposons que les balles soient rangées en file sur une étagère, de manière à n'avoir jamais deux balles de la même couleur à côté. Que peut-on en déduire sur le nombre maximal de balles de la même couleur ?

On a un ensemble de n balles, une étagère vide où on peut les ranger en file et une corbeille vide. Considérons l'algorithme suivant :

- Phase 1 - Prendre les balles une par une pour les ranger sur l'étagère ou dans la corbeille. SI la balle n'est pas de la même couleur que la dernière balle sur l'étagère, la ranger à côté, et si de plus la corbeille n'est pas vide, prendre une balle dans la corbeille et la ranger à côté sur l'étagère. SINON, c'est à dire si la balle est de la même couleur que la dernière balle sur l'étagère, la mettre dans la corbeille.

- Phase 2 - Soit C la couleur de la dernière balle sur l'étagère à la fin de la phase 1. On compare successivement la couleur de la dernière balle sur l'étagère avec C. SI la couleur est la même on jette les deux dernières balles sur l'étagère, sauf s'il n'en reste qu'une, auquel cas on la met dans la corbeille. SINON on la jette et on jette une des balles de la corbeille, sauf si la

corbeille est déjà vide auquel cas on s'arrête en décrétant qu'il n'y a pas de couleur majoritaire. Quand on a épuisé toutes les balles sur l'étagère, on regarde le contenu de la corbeille. Si elle est vide alors il n'y a pas de couleur majoritaire, et si elle contient au moins une balle alors C est la couleur majoritaire.

4.2 - (*Correction de l'algorithme*) Montrer qu'à tout moment de la phase 1, toutes les balles éventuellement présentes dans la corbeille ont la couleur de la dernière balle de l'étagère. En déduire que s'il y a une couleur dominante alors c'est C.

Prouver la correction de l'algorithme.

4.3 - (*Complexité*) Donner la complexité dans le pire des cas en nombre de comparaisons de couleurs des balles.

5 - Optimalité

On considère un algorithme de majorité pour les couleurs de n balles. Le but est de regarder faire l'algorithme, en choisissant au fur et à mesure les couleurs des balles pour que l'algorithme ait le maximum de travail (tout en restant cohérent dans le choix des couleurs). On obtiendra ainsi une borne inférieure de complexité pour un algorithme de majorité. C'est la technique de l'*adversaire*.

À tout moment de l'algorithme, on aura une partition des balles en deux ensembles : l'*arène* et les *gradins*. L'arène contient un certain nombre de composantes connexes de deux sortes : les *binômes* et les *troupeaux*. Un binôme est un ensemble de deux balles pour lesquelles l'algorithme a déjà testé si elles étaient de la même couleur et a répondu non. Un troupeau est un ensemble non vide de balles de la même couleur, connectées par des tests de l'algorithme. Ainsi, un troupeau avec k éléments a subi au moins $k-1$ comparaisons de couleurs entre ses membres. Soient B le nombre de binômes et T le nombre de troupeaux. Soient g le nombre d'éléments dans les gradins et t le nombre total d'éléments dans tous les troupeaux. Enfin, soit $m = \lfloor n/2 \rfloor + 1$ le "*seuil de majorité*".

Au début de l'algorithme toutes les balles sont des troupeaux à un élément. La stratégie de l'adversaire est la suivante. L'algorithme effectue un test $couleur(x) = couleur(y)$?

1. Si x ou y sont dans les gradins, la réponse est non.
2. Si x (resp. y) est dans un binôme, la réponse est non et x (resp. y) est envoyé dans les gradins alors que l'élément restant devient un troupeau singleton.
3. Si x et y sont dans le même troupeau la réponse est oui.
4. Si x et y sont dans des troupeaux différents alors cela dépend de $d = B + t$.
   (a) $d > m$: cela signifie que les balles sont dans des troupeaux singletons. La réponse est non et les balles deviennent un nouveau binôme.
   (b) $d = m$: la réponse est oui et les troupeaux de x et y fusionnent.

5.1 - Vérifier que les quatre cas précédents traitent tous les cas possibles.

Montrer qu'à tout moment $d \geq m$ et que si $d > m$ alors tous les troupeaux sont des singletons.

5.2 - Montrer qu'à tout moment les deux coloriages suivants sont consistants avec les réponses de l'adversaire :

(i) Toutes les balles sont de couleurs différentes sauf celles qui sont dans un même troupeau.

(ii) Une même couleur est attribuée à toutes les balles de tous les troupeaux et à une balle de chaque binôme. Les balles restantes ont chacune une couleur distincte.

5.3 - Montrer que si un algorithme correct s'arrête alors l'arène ne contient qu'une seule composante connexe qui est un troupeau de taille m.

5.4 - À tout moment le nombre de comparaisons inégales effectuées par l'algorithme est au moins $2g + B$ et le nombre de comparaisons égales est au moins $t - T$.

5.5 - Considérons un algorithme qui résout la majorité. Montrer qu'il existe une donnée pour laquelle l'algorithme effectue au moins $2(n - m) = 2\lceil n/2 \rceil - 1$ comparaisons d'inégalité et au moins $\lfloor n/2 \rfloor$ comparaisons d'égalité, et donc au moins au total $3\lceil n/2 \rceil - 2$ comparaisons.

Correction.

On se restreint ici aux algorithmes où la seule opération qu'on sait effectuer sur les éléments est de tester si deux éléments sont égaux. Si la réponse est `OUI`, on dira qu'il s'agit d'une *comparaison égale*, et si la réponse est `NON` d'une *comparaison inégale*.

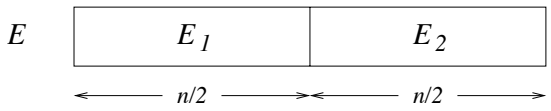
Remarque : s'il existe un élément majoritaire, il est nécessairement unique.

1 - Algorithme naïf

début
  pour i **de** *1* **à** n **faire**
    $c \leftarrow 0$;
    pour j **de** *1* **à** n **faire**
      si $E[j] = E[i]$ **alors** $c \leftarrow c + 1$;
    si $c > n/2$ **alors** retourner "$E[i]$ est majoritaire"
  Retourner "Pas d'élément majoritaire".
fin

Complexité : nombre total de comparaisons = n^2.

2 - Diviser pour règner



2.1 - **Principe :**

- Couper E en deux tableaux E_1 et E_2 de tailles $n/2$ (on suppose n pair).
- S'il existe un élément majoritaire x dans E, alors x est majoritaire dans au moins une des deux listes E_1 et E_2 (en effet si x non majoritaire dans E_1, ni dans E_2, alors dans E, $c_x \leq n/4 + n/4 = n/2$).
- Algorithme récursif : calculer les éléments majoritaires de E_1 et de E_2 (s'ils existent) avec le nombre total d'occurences de chacun et en déduire si l'un des deux est majoritaire dans E.

L'algorithme suivant $Majoritaire(i, j)$ renvoie un couple qui vaut (x, c_x) si x est majoritaire dans le sous-tableau $E[i..j]$ avec c_x occurences et qui vaut $(-, 0)$ s'il n'y a pas de majoritaire dans $E[i..j]$, l'appel initial étant $Majoritaire(1, n)$. La fonction $Occurence(x, i, j)$ calcule le nombre d'occurences de x dans le sous-tableau $E[i..j]$.

Algorithm: $Majoritaire(i, j)$

---

début
  si $i = j$ **alors** retourner $(E[i], 1)$;
  sinon
    $(x, c_x) = Majoritaire(i, \lfloor (i+j)/2 \rfloor)$;
    $(y, c_y) = Majoritaire(\lfloor (i+j)/2 \rfloor + 1, j)$;
    si $c_x \neq 0$ **alors** $c_x \leftarrow c_x + Occurence(x, \lfloor (i+j)/2 \rfloor + 1, j)$;
    si $c_y \neq 0$ **alors** $c_y \leftarrow c_y + Occurence(x, i, \lfloor (i+j)/2 \rfloor)$;
    si $c_x > \lfloor (j-i+1)/2 \rfloor$ **alors** retourner (x, c_x) ;
    sinon si $c_y > \lfloor (j-i+1)/2 \rfloor$ **alors** retourner (y, c_y) ;
    sinon retourner $(-, 0)$.
fin

Complexité : le nombre total de comparaisons dans le pire des cas $C(n)$ vérifie la relation (on suppose que n est une puissance de 2) :

$$C(n) = 2(C(\frac{n}{2}) + \frac{n}{2}) \text{ avec } C(1) = 0$$

On en déduit $C(n) = n \log_2(n)$.

2.2 - Si n n'est pas une puissance de 2, les trois points du principe précédent restent vrais en coupant E en un tableau de taille $\lfloor n/2 \rfloor$ et l'autre de taille $\lceil n/2 \rceil$, et l'algorithme décrit ci-dessus fonctionne sans aucune modification. La récurrence pour la complexité est alors $C(n) = C(\lfloor \frac{n}{2} \rfloor) + C(\lceil \frac{n}{2} \rceil) + n$ avec $C(1) = 0$, dont la résolution donne $C(n) \sim n \log_2(n)$.

3 - Encore mieux

Remarque 1 : soit x vérifiant la propriété, alors aucun autre élément $z \neq x$ ne peut être majoritaire (car $c_z \leq n - p < n/2$), donc, dans ce cas, x est un *candidat-majoritaire*. Autrement dit un algorithme vérifiant les propriétés de l'exercice nous fournit :

- soit la garantie qu'il n'existe aucun élément majoritaire,
- soit un *candidat-majoritaire* x, dont on peut vérifier ensuite s'il est vraiment majoritaire en reparcourant une fois le tableau (soit $n - 1$ comparaisons).

Remarque 2 : un élément majoritaire x est bien un *candidat-majoritaire* avec $p = c_x$ (donc s'il existe un majoritaire, il sera bien proposé comme *candidat-majoritaire* par l'algorithme).

3.1 - Algorithme récursif $Candidat - majoritaire(E)$ renvoyant `AUCUN` s'il n'y a pas d'élément majoritaire dans E et sinon renvoyant (x, p) avec les bonnes propriétés.

Algorithm:$Candidat - majoritaire(E)$

---

début
  si E *n'a qu'un élément* x **alors** retourner $(x, 1)$;
  sinon
    couper E en deux tableaux E_1 et E_2 de taille $n/2$;
    appeler $Candidat - majoritaire(E_1)$ qui renvoie `AUCUN` ou (x, p) ;
    appeler $Candidat - majoritaire(E_2)$ qui renvoie `AUCUN` ou (y, q) ;
    suivant la résponse pour E_1 et E_2, faire
    si `AUCUN` *et* `AUCUN` **alors** retourner `AUCUN` ;
    si `AUCUN` *et* (y, q) **alors** retourner $(y, q + \frac{n}{4})$;
    si (x, p) *et* `AUCUN` **alors** retourner $(x, p + \frac{n}{4})$;
    si (x, p) *et* (y, q) **alors**
      si $x \neq y$ *et* $p > q$ **alors** retourner $(x, p + \frac{n}{2} - q)$;
      si $x \neq y$ *et* $p < q$ **alors** retourner $(y, q + \frac{n}{2} - p)$;
      si $x \neq y$ *et* $p = q$ **alors** retourner `AUCUN`
      (sinon ce serait x ou y mais $c_x \leq \frac{n}{2}$ et $c_y \leq \frac{n}{2}$) ;
      si $x = y$ **alors** retourner $(x, p + q)$
fin

Complexité : on a supposé n un puissance de 2, le nombre de comparaisons $C(n)$ pour un tableau à n éléments vérifie : $C(n) = 2C(\frac{n}{2}) + 1$ avec $C(1) = 0$, ce qui donne $C(n) = n - 1$.

3.2 - Si on ne suppose pas que n est une puissance de 2, couper E en deux tableaux de tailles $\lfloor \frac{n}{2} \rfloor$ et $\lceil \frac{n}{2} \rceil$, et affiner l'analyse.

Si n est pair, faire comme à la question précédente en remplaçant juste $\frac{n}{4}$ par $\lceil \frac{n}{4} \rceil$. La complexité vérifie alors $C(n) = C(\lfloor \frac{n}{2} \rfloor) + C(\lceil \frac{n}{2} \rceil) + 1$.

Si n est impair, $n = 2m + 1$, adapter la procédure précédente de la manière suivante :

début
  si $m = 0$, *E n'a qu'un élément* x **alors** retourner $(x, 1)$;
  sinon
    couper E en un tableau E_1 de taille m et un tableau E_2 de taille $m + 1$;
    appeler $Candidat - majoritaire(E_1)$ qui renvoie `AUCUN` ou (x, p) ;
    appeler $Candidat - majoritaire(E_2)$ qui renvoie `AUCUN` ou (y, q) ;
    suivant la résponse pour E_1 et E_2, faire
    si `AUCUN` *et* `AUCUN` **alors** retourner `AUCUN` ;
    si `AUCUN` *et* (y, q) **alors** retourner $(y, q + \lceil \frac{m}{2} \rceil)$;
    si (x, p) *et* `AUCUN` **alors** retourner $(x, p + \lceil \frac{m+1}{2} \rceil)$;
    si (x, p) *et* (y, q) **alors**
      si $x \neq y$ *et* $p \geq q$ **alors** retourner $(x, p + m + 1 - q)$;
      si $x \neq y$ *et* $q \geq p + 1$ **alors** retourner $(y, q + m - p)$;
      si $x = y$ **alors** retourner $(x, p + q)$
fin

La complexité vérifie toujours $C(n) = C(\lfloor \frac{n}{2} \rfloor) + C(\lceil \frac{n}{2} \rceil) + 1$ avec $C(1) = 0$. La résolution de cette formule de récurrence donne $C(n) = n - 1$.

3.3 - Algorithme complet pour rechercher un majoritaire :
- Recherche d'un *candidat-majoritaire* avec l'algorithme précédent, soit $n-1$ comparaisons.
- Si l'algorithme renvoie un candidat, vérification que le candidat est bien majoritaire en reparcourant le tableau, soit $n - 1$ comparaisons.

Complexité au total $= 2n - 2$ comparaisons.

4 - Encore encore mieux

4.1 - Soit k le nombre de balles rangées en file sur l'étagère telles que deux balles consécutives n'aient jamais la même couleur, alors le nombre de balles d'un même couleur est toujours $\leq \lceil \frac{k}{2} \rceil$.

Preuve précise : utiliser le *lemme des tiroirs*, c'est à dire "si M chaussettes sont rangées parmi m tiroirs avec $M > m$, alors il existe un tiroir avec au moins deux chaussettes". Ici découper l'étagère en $\lceil \frac{k}{2} \rceil$ tiroirs disjoints qui sont des paires d'emplacements consécutifs sur l'étagère. S'il y a plus de $\lceil \frac{k}{2} \rceil$ balles d'une même couleur, il en existe deux dans un même tiroir, c'est à dire ici adjacentes sur l'étagère.

4.2 - *Correction de l'algorithme :* on montre des invariants (propriétés restants vraies pendant une phase de l'algorithme).

- Phase 1 -
Invariant 1 : si la corbeille n'est pas vide, toutes les balles dans la corbeille ont la même couleur que la dernière ballle sur l'étagère.
Preuve : montrer que si la prop. est vraie à un instant donné, elle reste vraie à l'étape suivante. Plusieurs cas peuvent se produire ici, vérifier la conservation de la propriété dans chaque cas (à faire) :

- ajout d'une balle différente de la dernière et corbeille pleine ...
- ajout d'une balle différente de la dernière et corbeille vide ...
- ajout d'une balle identique à la dernière ...

Invariant 2 : sur l'étagère, il n'y a jamais deux balles de même couleur côte à côte.
Preuve : idem au cas par cas.

Fin de Phase 1 : soit C la couleur de la dernière balle sur l'étagère. Les balles d'autres couleurs sont dans les $k - 1$ premières balles de l'étagère, où k est le nombre de balles sur l'étagère et $k \leq n$. Comme il n'y a pas deux balles adjacentes de même couleur, d'après la question 1, le nombre de balles d'une couleur différente de C est $\leq \lceil \frac{n-1}{2} \rceil = \lfloor \frac{n}{2} \rfloor$. Donc la seule couleur candidate pour être majoritaire est C.

- Phase 2 - vérifie si C est bien majoritaire :

- Quand on retire une paire de balles, on en a toujours une de couleur C et l'autre différente, donc C est majoritaire si et seulement si une majorité de balles à la fin (étagère + corbeille) est de couleur C.
- Deux cas de fin se présentent :
  - La phase 2 s'arrête car on a besoin d'une balle de la corbeille mais la corbeille est vide. Dans ce cas, la dernière balle sur l'étagère n'est pas de couleur C et d'après la question 1, au plus la moitié des balles restant sur l'étagère sont de couleur C, il n'y a pas de majoritaire. Ok, c'est bien ce que répond l'algorithme.
  - L'algorithme va jusqu'au bout de l'étagère, les balles restantes (s'il en reste) sont dans la corbeille, elles sont toutes de couleur C. Donc C est majoritaire si et seulement si la corbeille est non vide, c'est bien le test effectué par l'algorithme.

4.3 - **Complexité :**
Phase 1 : $(n - 1)$ comparaisons (-1 car la première balle est directement posée sur l'étagère).
Phase 2 : une comparaison par paire de balles éliminées, plus éventuellement une comparaison avec C pour la dernière balle de l'étagère s'il en reste une unique, soit $\leq \lceil \frac{n}{2} \rceil$ comparaisons (-1 car au début de la phase 2, on sait que la dernière balle sur l'étagère est de couleur C).
Complexité totale $\leq \lceil \frac{3n}{2} \rceil$ comparaisons.

5 - Optimalité

5.1 - Tous les cas possibles sont bien représentés. Au départ $d = n$ et tous les troupeaux sont des singletons. Quels cas modifient d ?
Cas 1 : non.
Cas 2 : non (car -1 binôme, +1 dans les troupeaux).
Cas 3 : non.
Cas 4(a) : oui, d diminue de 1 (mais les troupeaux restent des singletons).
Cas 4(b) : non.
Conclusion :

- Invariant $d \geq m$ (si d change, c'est que $d > m$ avec le cas 4(a) et alors $d - 1 \geq m$).
- $d > m$ implique que tous les troupeaux sont des singletons.

5.2 - Consistence des réponses de l'adversaire à tout moment.
Coloration (i) : correspond bien aux réponses données, car respecte les binômes (*comparaisons inégales*) et les troupeaux (*comparaisons égales*). De plus les balles envoyées dans les gradins ont toujours engendré la réponse NON aux comparaisons, ce qui fonctionne si leurs couleurs sont toutes différentes et différentes des couleurs des balles dans l'arène.
Coloration (ii) : idem avec de plus le fait qu'une balle dans un binôme n'a jamais été comparée à quelqu'un d'autre que son binôme (donc pas de risque de contradiction avec les choix des couleurs ailleurs).

5.3 - Un algorithme correct ne peut s'arrêter que si l'arène ne contient qu'une composante connexe qui sera un troupeau de taille m.

Preuve par l'absurde : supposons que l'arène contienne au moins deux composantes (ce qui implique $n \geq 2$ et $m \geq 2$). Par définition de d, chaque troupeau contient $< d$ balles. Chaque troupeau contient donc $< m$ balles, car soit $d = m$, soit $d > m$ et d'après la question 1 chaque troupeau est un singleton. La coloration (i) n'a pas d'élément majoritaire, mais la coloration (ii) a un élément majoritaire (car $d \geq m$). Donc aucun algorithme correct ne peut s'arrêter à cet instant.

Conclusion : à la fin, il y a nécessairement une unique composante, nécessairement de taille $d = m$ (par définition de d et d'après la question 1 qui implique que $d \not> m$).

5.4 - A tout moment, on a eu jusqu'à présent :

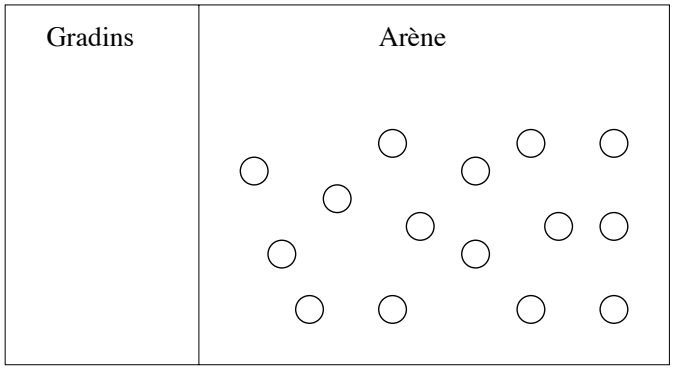
- Nombre de *comparaisons inégales* $\geq 2g + B$, car il faut 2 comparaisons inégales pour mettre une balle dans les gradins (entrée dans un binôme puis sortie du binôme) et il y a B comparaisons inégales qui ont construit les binômes en place.
- Nombre de *comparaisons égales* $\geq t - T$, car un troupeau i avec t_i balles est connexe donc il a $\geq t_i - 1$ arêtes, soit ici $t_i - 1$ comparaisons égales.

5.5 - En fin d'algorithme, d'après la question 3, $g = n - m$, $B = 0$, $t = m$ et $T = 1$. Donc, avec la question 4, il y a eu $\geq 2(n - m) = 2\lceil \frac{n}{2} \rceil - 2$ comparaisons inégales et $\geq m - 1 = \lfloor \frac{n}{2} \rfloor$ comparaisons égales, soit au total $\geq \lceil \frac{3n}{2} \rceil$ comparaisons.

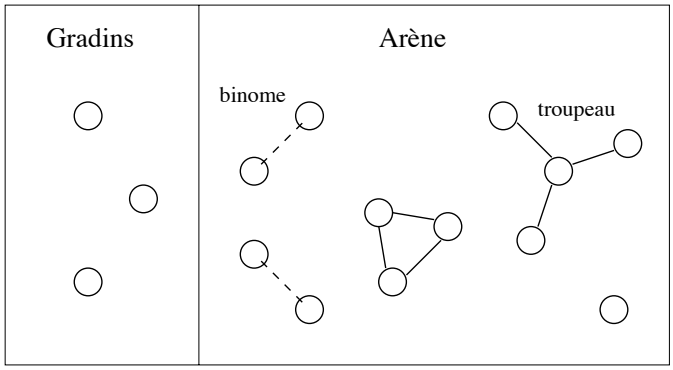
2.7 Références bibliographiques

La présentation du cours, et l'exercice *Matrices de Toeplitz*, s'inspirent du Cormen [2]. L'exercice *Recherche d'un élément majoritaire* est tiré de Darte et Vaudenay [3].

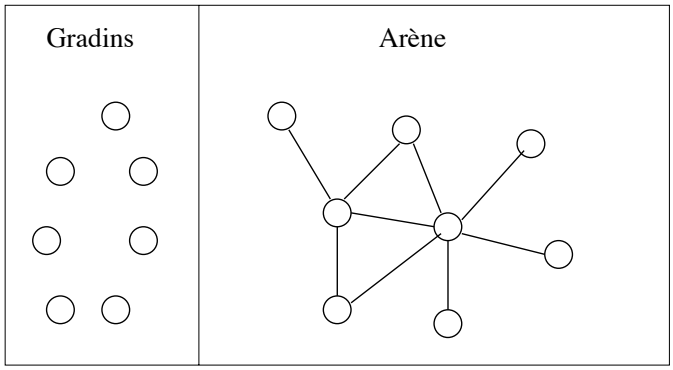
Départ



Pendant



Fin



Chapitre 3

Programmation dynamique

3.1 Pièces de Monnaies

On dispose de pièces en nombre illimité, de valeurs 10, 5, 2 et 1. On veut arriver a une somme S avec le minimum de pièces.

Algorithme Glouton : Trier les types de pièces par valeurs décroissantes. Pour chaque valeur de pièce, maximiser le nombre de pièces choisies. Plus formellement, soit R la somme restante, initialisée à S. Pour chaque valeur v_i, prendre $c_i = \lfloor \frac{R}{v_i} \rfloor$ pièces, et poser $R \leftarrow R - c_i \cdot v_i$.

Pour prouver l'optimalité de l'algorithme glouton avec les valeurs 10, 5, 2 et 1 :
- Au plus une pièce de 5 (sinon une de 10)
- Au plus une pièce de 1 (sinon une de 2)
- Au plus deux pièces de 2 (sinon une de 5 et une de 1)
- Au plus quatre pièces qui ne sont pas des pièces de 10 ($1 * 5 + 2 * 2 + 1 = 1 * 10$), pour un total inférieur ou égal à 9
- Le nombre de pièces de 10 est donc $\lfloor \frac{S}{10} \rfloor$
- Conclure avec ce qui précède

Remarque : l'algorithme glouton n'est pas optimal pour le jeu de pièces de valeurs $(6, 4, 1)$: pour $S = 8$, le glouton renvoie $8 = 6 + 1 + 1$ alors qu'on a $8 = 4 + 4$.

Dans le cas général, on cherche $m(S)$, le nombre minimum de pièces pour faire S avec des pièces de valeur $(a_1, a_2, \ldots, a_n)$. Pour cela on introduit la fonction z telle que :
- $z(T, i)$ = nombre minimum de pièces choisies parmi les i premières $(a_1, ..., a_i)$ pour faire T
- on remarque que $z(S, n) = m(S)$, on résout un problème apparemment plus compliqué que le problème de départ

Mais on a une relation de récurrence :

$$z(T, i) = \min \begin{cases} z(T, i-1) & i\text{-ème pièce non utilisée} \\ z(T - v_i, i) + 1 & \text{on a utilisé une fois (au moins) la } i\text{-ème pièce} \end{cases}$$

Il faut initialiser la récurrence correctement, par exemple en posant $z(T, i) = 0$ pour $T \leq 0$ ou $i = 0$.

Comment calculer toutes les $S \cdot n$ valeurs de z dont on a besoin ? Un algorithme qui calcule par colonnes (boucle sur i externe, boucle sur T interne) permet de respecter les dépendances, i.e. de toujours avoir en membre droit des valeurs précédemment obtenues. On obtient une complexité en $O(n * S)$, alors que l'algorithme glouton avait une complexité en $O(n \log n)$ (l'exécution est linéaire, mais il faut auparavant trier les valeurs).

pour *i=1..n* **faire**
  pour *T=1..S* **faire**
    Calculer $z(T, i)$: il faut
    - $z(T, i-1)$, calculé à l'itération précédente ou $i = 0$
    - $z(T - v_i, i)$, calculé précédemment dans cette boucle ou $T \leq 0$

  fin
fin

Remarque Caractériser les jeux de pièces pour lesquels l'algorithme glouton est optimal est un problème ouvert. Il est facile de trouver des catégories qui marchent (par exemple des pièces $1, B, B^2, B^3, \ldots$ pour $B \geq 2$) mais le cas général résiste !

3.2 Le problème du sac à dos

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

3.2.1 En glouton

On va travailler sur le rapport "qualité/prix". On commence par trier les objets selon les $\frac{c_i}{w_i}$ décroissants, puis on gloutonne en remplissant le sac par le plus grand élément possible à chaque tour.

Question : Est-ce optimal ? Non.

Le problème qui se pose est que l'on travaille avec des éléments discrets non divisibles. Prenons un contre-exemple : si l'on considère 3 objets, le 1er (c_i/w_i max) remplissant le sac à lui seul (aucun autre objet ne rentre) et les 2ème et 3ème tels que $c_1 > c_2$, $c_1 > c_3$, mais que $c_2 + c_3 > c_1$ et que les deux objets puissent rentrer ensemble dans le sac. Alors la solution donnée par glouton (remplissage par le premier objet) est sous-optimale, contrairement au remplissage par les objets 2 et 3.

$$(W = 10) \qquad (w_1 = 6; w_2 = 5; w_3 = 5) \qquad (c_1 = 7; c_2 = 5; c_3 = 5)$$

est un exemple d'un tel cas de figure.

3.2.2 Par programmation dynamique

Afin de résoudre le problème par une récurrence, on va le compliquer. Un problème de remplissage plus complexe que celui de départ est celui où la taille du sac et le nombre d'objets à employer sont arbitraires. Posons alors $C(v, i)$ comme l'expression du meilleur coût pour remplir un sac de taille v avec les i premiers objets. Résoudre le problème de remplissage du sac de taille W avec les n objets revient à donner $C(W, n)$.

La récurrence : soit on a pris le dernier objet, soit on ne l'a pas pris, d'où

$$C(v, i) = \max \begin{cases} C(v, i-1) & \text{dernier objet non considéré} \\ C(v - w_i, i-1) + c_i & \text{dernier objet pris} \end{cases}$$

La solution optimale des sous-problèmes va servir à retrouver la solution optimale du problème global, sans que l'on calcule jamais 2 fois la même valeur. Ceci implique par contre de respecter les dépendances du calcul.

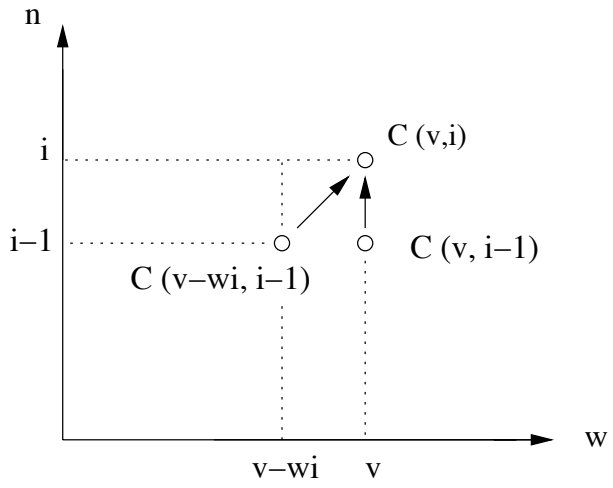


FIG. 3.1 – Attention au respect des dépendances !

Remarque Le coût de glouton est en $O(n \log n)$ (tri de n nombres) tandis que le coût du traitement en programmation dynamique est en $O(nW)$. Ce n'est donc pas polynomial en la taille des données, car les données sont en $\sum_{i=1}^{n} \log v_i + \sum_{i=1}^{n} \log c_i \leq n(\log W + \log C)$, donc W est exponentiel en la taille des données.

3.3 Quelques exemples de programmation dynamique

Les exemples que nous allons voir ici concernent le traitement de chaînes de matrices d'une part, et la détermination de la plus longue sous-suite de deux chaînes de caractères d'autre part.

Credo du programmeur dynamique : "Demain est le premier jour du reste de ta vie."

3.3.1 Chaînes de matrices

Considérons n matrices A_i, de taille $P_{i-1} \times P_i$ On veut calculer $A_1 \times A_2 \times \ldots \times A_n$. Le problème que l'on se pose est de trouver dans quel ordre effectuer les calculs (et donc comment parenthéser l'expression).

Le nombre de façons différentes d'effectuer le produit peut se calculer par récurrence. Soit $C(i)$ le nombre de façons d'effectuer un produit de i matrices. On coupe le produit en deux juste après la matrice d'indice k, et on obtient donc pour $n \geq 2$:

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

On comprend intuitivement que le nombre de façons de calculer dépend de la façon dont les deux parties de l'expression ont été elles-mêmes calculées. La condition initiale est $C(1) = 1$. Knuth nous explique que l'on peut calculer $C(n)$ par l'intermédiaire des nombres de Catalan (obtenus par séries génératrices [1]). On a $C(n) = \frac{1}{n} C_{2n-2}^{n-1} = \Omega(4^n/n^{1.5})$. Cette quantité effarante nous fait bien vite déchanter, et l'on se résout à abandonner la stratégie de la force brute.

[1] En cas de besoin urgent de compulser une référence sur les séries, se référer à Flajeolet et Sedgewick

Pourquoi faire simple quand on peut faire compliqué ? Tâchons de nous ramener à de la programmation dynamique (après tout c'est le titre de ce paragraphe). Nous cherchions à calculer $A_1 \times \ldots \times A_n$, cherchons donc le coût optimal du calcul de $A_i \times \ldots \times A_j$ (noté $C(i,j)$). La solution de notre problème s'obtiendra pour $i = 1$ et $j = n$.
La récurrence :
Supposons que la meilleure façon de couper $(A_i, \ldots, A_j)$ est de couper à un endroit k :

$$\underbrace{(A_i \ldots A_k)}_{\text{coût optimal pour l'obtenir : } C(i,k)} \quad * \quad \underbrace{(A_{k+1} \ldots A_j)}_{\text{coût optimal pour l'obtenir : } C(k+1,j)}$$

Le coût total d'un parenthésage en k est donc de

$$C(i,k) + C(k+1,j) + \text{ coût du produit des 2 membres } (P_{i-1} * P_k * P_j)$$

et le coût optimal s'obtient par

$$\min_{k=i}^{j-1} \{C(i,k) + C(k+1,j) + (P_{i-1} * P_k * P_j)\}$$

Le calcul se fait en respectant les dépendances : le calcul de $C(i,j)$ demande tous les $C(i,k)$ et tous les $C(k+1,j)$

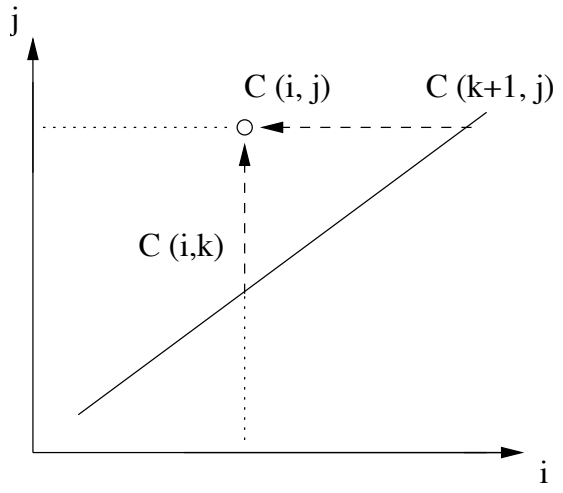


FIG. 3.2 – L'initialisation se fait sur $C(i, i+1) = P_{i-1} * P_i * P_{i+1}$ et $C(i,i) = 0$ sur la diagonale

L'algorithme réalisant le calcul en respectant les dépendances sera en $O(n^3)$ (for diag = ... for élément dans diag calcul for k = ...)

Pour reconstruire la solution, il faut mémoriser pour chaque couple (i,j) l'indice $s(i,j)$ où il faut couper.

Question :
Cet algorithme peut-il s'écrire en récursif ? Oui.
Cela demande t-il plus de calcul ? Non, à condition de s'arrêter à l'entrée d'un calcul $C(i,j)$ déjà effectué. On initialise toutes les valeurs de $C(i,j)$ à $+\infty$, et on effectue un test à l'entrée de l'appel récursif. Sinon c'est exponentiel.

3.3.2 Plus longue sous-suite

Exemple : Votre petit frère apprend l'anglais avec un logiciel qui demande "What is the capital of the USA ?". Il serait intéressant que le logiciel soit capable de différencier "New-

York" de "Washington" mal écrit, pour donner une réponse appropriée (mauvaise réponse ou mauvaise orthographe).
On peut alors définir une mesure de *distance d'édition*, c'est à dire le nombre de transformations nécessaires pour passer d'une chaîne à une autre.
Soient

$$A = a_1...a_n \; B = b_1...b_m$$

On définit une sous-chaîne de A comme étant $a_{i_1}a_{i_2}...a_{i_k}$.
On recherche les sous-chaînes communes à A et B, et plus particulièrement la Plus Longue Sous-Chaîne Commmune ou $PLSCC$.
Exemple :
$A = aabaababaa, \; B = ababaaabb$
alors $PLSCC = ababaaa$, non consécutifs.

Solution exhaustive

On essaie toutes les sous-suites : on énumère les 2^n sous-suites dans chaque chaîne, puis on les compare. Sans doute peu efficace.

Programmation dynamique

On cherche la plus longue sous-chaîne commune entre les mots A et B, de longueurs respectives n et m. On recherche :

$$PLSCC(n,m) : \begin{cases} A[1 \dots n] \\ B[1 \dots m] \end{cases}$$

Pour cela, on utilise la plus longue sous-chaîne commune entre les i et j premières lettres de A et B respectivement, soit :

$$PLSCC(i,j) = p(i,j) : \begin{cases} A[1 \dots i] \\ B[1 \dots j] \end{cases}$$

On peut alors enclencher une récurrence pour résoudre le problème de la recherche de $p(i,j)$.

$$p(i,j) = max \begin{cases} p(i,j-1) \\ p(i-1,j) \\ p(i-1,j-1) + [a_i = b_j] \quad \text{avec } [a_i = b_j] \text{ vaut 1 si } a_i = b_j, \text{ 0 sinon} \end{cases}$$

Preuve
On constate d'abord que :

$$max \begin{cases} p(i,j-1) \\ p(i-1,j) \\ p(i-1,j-1) + [a_i = b_j] \end{cases} \leq p(i,j)$$

Trivial en effet car $p(i,j)$ est croissant à valeurs entières en i et en j.
On peut diviser la preuve de l'égalité en deux cas : soit (1) $a_i \neq b_j$, soit (2) $a_i = b_j$.

1. 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 deux 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 $p(i,j) = p(i-1,j)$.

2. Si $a_i = b_j$, deux cas se présentent à nouveau : soit la PLSCC contient $a_i = b_j$, et alors $p(i,j) = p(i-1,j-1)+1$, soit elle ne contient pas $a_i = b_j$ et donc $p(i,j) = p(i-1,j-1)+0$.

On a donc prouvé la relation de récurrence.

Exemple
$A = a\ b\ c\ b\ d\ a\ b$
$B = b\ d\ c\ a\ b\ a$

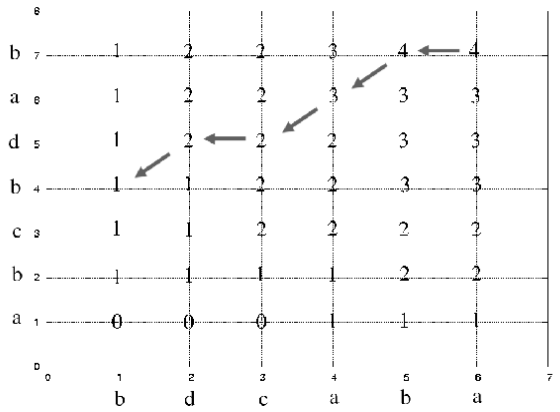


FIG. 3.3 – La $PLSCC$ est *bdab*.

La $PLSCC$ trouvée est *bdab* : dans la Figure 3.3, on trouve une lettre commune lorsque les flèches sont diagonales. En effet, l'algorithme consiste à prendre l'option qui maximise le score, c'est à dire le maximum de la pente.

Le coût de la recherche de la plus longue sous-chaîne commune est de $O(nm)$. Il existe de meilleurs algorithmes dans la littérature, de deux types :

- les algorithmes qui sont efficaces lorsque les séquences sont ressemblantes
- les algorithmes qui sont efficaces lorsque 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.

3.3.3 Location de skis

Voici un exercice très joli, pour lequel il est conseillé de chercher la solution avant de lire la réponse !

Allocation de skis aux skieurs Spécifier un algorithme efficace pour une attribution optimale de m paires de skis de longueur $s_1, \ldots, s_m$, respectivement, à n skieurs ($m \geq n$) de taille $h_1, \ldots, h_n$, respectivement, via une fonction (injective) $f : \{1, \ldots, n\} \to \{1, \ldots, m\}$, f étant optimale lorsqu'elle minimise $\sum_{k=1}^{n} |s_{f(k)} - h_k|$. On donne l'indication suivante : soit $A[n,m]$ ce minimum. Définir $A[i,j]$ pour des valeurs plus petites $i \leq n$ et $j \leq m$ (lesquelles ?), par une équation de récurrence (i et j font référence aux i premiers skieurs et aux j premières paires de skis, respectivement).

Complexité Analyser la complexité (en veillant à affiner l'analyse de sorte à garantir que l'algorithme soit en $O(n \log n)$ si $m = n$).

Grand choix de skis Montrer qu'on peut avoir une meilleure complexité lorsque $n^2 = o(m)$. (Indication : se restreindre à $O(n^2)$ paires de skis).

Allocation de skis aux skieurs On considère le même problème, mais en ne prenant que les i premiers skieurs et les j premières paires de skis (avec évidemment les mêmes longueurs et tailles). Pour écrire la récurrence définissant $A[i,j]$, on est naturellement amené à s'intéresser à la j-ème paire de skis et au i-ème skieur. Il y a deux cas. Si la solution optimale pour i,j n'utilise pas la j-ème paire de skis, alors $A[i,j] = A[i,j-1]$. Si la j-ème paire de skis est utilisée, il est tentant de l'affecter au i-ème joueur. On fait désormais l'hypothèse que les suites $s_1,\ldots,s_m$ et $h_1,\ldots,h_n$ sont rangées dans l'ordre croissant, ou plus précisément l'ont été dans une première phase de l'algorithme. On a la propriété suivante :

Lemme 1. (P) *Si $f : \{1,\ldots,i\} \to \{1,\ldots,j\}$ (injective) est telle qu'il existe $i_1 \leq i$ avec $f(i_1) = j$, alors $\sum_{k=1}^{i} |s_{f(k)} - h_k| \geq \sum_{k=1}^{i} |s_{f'(k)} - h_k|$, avec f' obtenue à partir de f en échangeant les images de i_1 et i (donc $f'(i_1) = f(i)$, $f'(i) = j$ et $f' = f$ ailleurs).*

Preuve. Intuitivement, la propriété (P) dit que si on a une attribution avec deux paires de ski pour deux skieurs donnés, mieux vaut (plus exactement on ne perd rien à) attribuer la plus petite paire au plus petit skieur et la plus grande paire au plus grand skieur. On pose

$$j_1 = f(i) \quad A = |s_j - h_{i_1}| + |s_{j_i} - h_i| \quad B = |s_{j_1} - h_{i_1}| + |s_j - h_i| \; .$$

Il faut montrer $A - B \geq 0$. On examine les différentes positions relatives de s_{j_1}, s_j, h_{i_1} et h_i.

$$\begin{aligned} s_{j_1} \leq h_{i_1} \leq h_i \leq s_j : A - B &= (s_j - h_{i_1} + h_i - s_{j_1}) - (h_{i_1} - s_{j_1} + s_j - h_i) \\ &= 2(h_i - h_{i_1}) \;\geq\; 0 \end{aligned}$$

$$\begin{aligned} s_{j_1} \leq h_{i_1} \leq s_j \leq h_i : A - B &= (s_j - h_{i_1} + h_i - s_{j_1}) - (h_{i_1} - s_{j_1} - s_j + h_i) \\ &= 2(s_j - h_{i_1}) \;\geq\; 0 \end{aligned}$$

$$\begin{aligned} s_{j_1} \leq s_j \leq h_{i_1} \leq h_i : A - B &= (h_{i_1} - s_j + h_i - s_{j_1}) - (h_{i_1} - s_{j_1} + h_i - s_j) \\ &= 0 \end{aligned}$$

(On a omis les cas $h_{i_1} \leq s_{j_1} \leq h_i \leq s_j$ et $h_{i_1} \leq h_i \leq s_{j_1} \leq s_j$ qui sont similaires.) □

Par la propriété (P), on peut supposer que le i-ème skieur se voit attribuer la j-ème paire de skis. En effet, si ce n'est pas le cas pour f, alors f' tel que défini plus haut est meilleure que f, donc est optimale aussi, et attribue la j-ème paire de skis au i-ème skieur. Et on a $A[i,j] = A[i-1,j-1] + |s_j - h_i|$. Au total, on a prouvé :

$$A[i,j] = \min(A[i,j-1], (A[i-1,j-1] + |s_j - h_i|)) \; .$$

L'algorithme consiste donc à calculer ces $A[i,j]$, et si l'on veut calculer l'attribution f en même temps, on note $f(i) = j$ au passage si on est dans le deuxième cas.

Complexité Combien d'entrées $A[i,j]$ faut-il calculer ? Il est facile de voir que les appels récursifs depuis $A[n,m]$ laissent de côté les deux triangles gauche inférieur et droite supérieur de la matrice (n,m) qui ensemble ont une dimension (n,n). Donc, ce ne sont pas nm, mais $(m-n)n$ entrées qui sont à calculer. Si l'on compte le temps du tri des deux suites, on arrive donc à une complexité

$$O((m \log m) + (n \log n) + (m-n)n) \; .$$

Ceci est satisfaisant dans le cas particulier où $m = n$: on obtient alors $O(m \log m)$, et en effet, si $m = n$, il suffit de trier les deux suites et de poser $f = id$ (ce qui se voit par récurrence en utilisant la propriété (P)). (Noter que la complexité mal affinée $O(m \log m) + O(n \log n) + mn$ aurait donné $O(n^2)$.)

Grand choix de skis On remarque que si les listes sont triées, on peut se contenter, pour chaque skieur, de trouver la meilleure paire de skis et les $n-1$ meilleures paires de skis après la meilleure, de part et d'autre de la meilleure longueur. Cette fenêtre S_i de taille $2n-1$ permet de prévoir les conflits avec les autres skieurs. Plus précisément, les paires de skis ainsi déterminées représentent un ensemble $S = \bigcup_{i=1}^{n} S_i$ de $n(2n-1)$ paires de skis avec de possibles répétitions tel que l'on peut définir une injection $f : \{1, \ldots, n\} \to S$ avec $f(i) \in S_i$ pour tout i (choisir successivement une paire de skis différente pour chaque skieur, la marge de manoeuvre laissée d'au moins $n-1$ autres paires de skis pour chaque skieur garantit que c'est possible). On peut donc appliquer l'étude du début à n et à cet ensemble S dont le cardinal est au plus $n(2n-1)$. Le temps pris pour déterminer le sous-ensemble S est $O(n((\log m) + n))$: le premier n correspond à la recherche des S_i successifs, $\log m$ est le temps pour déterminer la meilleure paire de skis (par dichotomie dans la liste triée des s_i), et le dernier n correspond à la taille de la fenêtre autour de la meilleure hauteur.

Il reste à voir qu'une solution optimale pour ce sous-ensemble de paires de skis est optimale pour l'ensemble de toutes les paires de skis. On remarque que pour tout $j \notin S$, on a par construction que pour tout i il existe au moins n éléments $s_{j_1}, \ldots, s_{j_n}$ de S_i tels que $|s_j - h_i| \geq |s_{j_k} - h_i|$ $(1 \leq k \leq n)$. (Noter ici l'importance d'avoir pris des paires *de part et d'autre* de la meilleure paire de skis.) Si donc il existait une fonction d'attribution optimale qui atteint j (ainsi possiblement que d'autres valeurs hors de S), on pourrait remplacer sans conflit (cf. ci-dessus) ces valeurs par des valeurs de S, et donc obtenir une autre fonction optimale à valeurs dans S.

La complexité relative à n et S s'exprime comme suit (en y incorporant le temps de recherche des éléments de S) :

$$O(n^2 \log n) + O(n \log n) + O((n^2 - n)n) + O(n((\log m) + n)) = O(n^3 + n \log m) \ .$$

Ce temps est meilleur que le temps obtenu en (A2) si $n^2 = o(m)$, car alors $O(n^3 + n \log m) = o((m \log m) + (n \log n) + (m - n)n)$. En effet, $(n \log m)/(m \log m)$ et $n^3/(m-n)n$ tendent vers 0 (noter qu'on a fortiori $n = o(m)$).

3.4 Exercices

Exercice 3.4.1. *Triangulation de polygones*

On considère les polygones convexes du plan. Une triangulation d'un polygone est un ensemble de cordes qui ne se coupent pas à l'intérieur du polygone et qui le divisent en triangles.

1 - Montrer qu'une triangulation d'un polygone à n côtés a $(n-3)$ cordes et $(n-2)$ triangles.

2 - Le problème est celui de la triangulation optimale de polygones. On part d'un polygone convexe $P = \langle v_0, ..., v_n \rangle$, où $v_0, \ldots, v_n$ sont les sommets du polygone donnés dans l'ordre direct, définie sur les triangles formés par les côtés et les cordes de P (par exemple $w(i,j,k) = \|v_i v_j\| + \|v_j v_k\| + \|v_k v_i\|$ est le périmètre du triangle $v_i v_j v_k$). Le problème est de trouver une triangulation qui minimise la somme des poids des triangles de la triangulation.

On définit pour $1 \leq i < j \leq n$, $t[i,j]$ comme la pondération d'une triangulation optimale du polygone $\langle v_{i-1}, ..., v_j \rangle$, avec $t[i,i] = 0$ pour tout $1 \leq i \leq n$.

Définir t récursivement, en déduire un algorithme et sa complexité.

3 - Si la fonction de poids est quelconque, combien faut-il de valeurs pour la définir sur tout triangle du polygone ? Comparez avec la complexité obtenue.

4 - Si le poids d'un triangle est égal à son aire, que pensez-vous de l'algorithme que vous avez proposé ?

Correction.

1- On procède par récurrence sur $n \geq 3$.
Pour $n = 3$, le polygône est un triangle, qui a bien $n - 2 = 1$ triangle et $n - 3 = 0$ corde.
Soit $n \in \mathbb{N}, n \geq 3$. On suppose le résultat démontré pour tout polygône convexe possédant un nombre de côtés strictement inférieur à n. Soit alors un polygône à n côtés et une triangulation de ce polygône. Considérons une corde qui divise ce polygône en deux polygônes, respectivement à $(i+1)$ et $(j+1)$ côtés avec $i + j = n, i \geq 2, j \geq 2$ (comme le montre la figure 3.4).

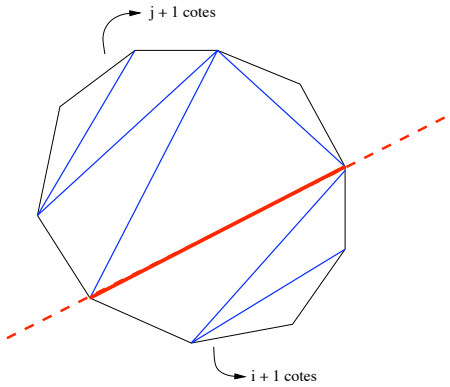


FIG. 3.4 – Une corde (en bleu) qui divise un polygone en deux

Par hypothèse de récurrence, on a $(i-2)$ cordes dans le premier polygône et $j-2$ dans le second, ce qui fait pour le polygône entier $(i-2)+(j-2)+1 = n-3$ cordes en tout (le $+1$ représente la corde qui séparait le polygône en deux).

De même, on obtient un total de $(i-1)+(j-1) = n-2$ triangles dans le polygône en tout, ce qui achève la démonstration.

2- Pour trouver une formule de récurrence sur t, il suffit de s'aperçevoir que le $v_{i-1}v_j$ fait partie d'un triangle dont le troisième sommet est v_k. On en déduit la formule : $t[i,j] = min_{i \leq k \leq j-1}(t[i,k] + t[k+1,j] + w(i-1,k,j))$ (*On remarque que la convention* $t[i,i]$ *est bien adaptée puisque la formule ci-dessus reste vraie quand* $j = i+1$ *: on a bien* $t[i,i+1] = w(i-1,i,i+1)$.).

Nous pouvons maintenant écrire un algorithme qui calcule $t[1,n]$, et ce à partir des $t[k,j], i+1 \leq k \leq j$ et des $t[i,k], i \leq k \leq j-1$, c'est à dire, si l'on représente les valeurs à calculer dans un diagramme (i,j) comme celui donné dans la figure 3.5, à partir des valeurs situées "en-dessous" et "à droite" du point $(1,n)$. On s'aperçoit alors qu'il suffit de calculer les $t[i,j]$ pour i de 1 à n et pour $j \geq i$, et ce par $(j-i)$ croissant.

Ceci donne l'algorithme :

début
  pour i ***de*** 0 ***à*** n **faire**
    $t[i,i] \leftarrow 0$
  pour d ***de*** 1 ***à*** $n-1$ **faire**
    pour i ***de*** 1 ***à*** $n-d$ **faire**
      $t[i,i+d] \leftarrow min_{i \leq k \leq j-1}(t[i,k] + t[k+1,j] + w(i-1,k,j))$
  Retourner $t[1,n]$
fin

Cet algorithme calcule $t[i,j]$ en $2(j-i)$ additions et en $j-i-1$ comparaisons pour calculer le min.

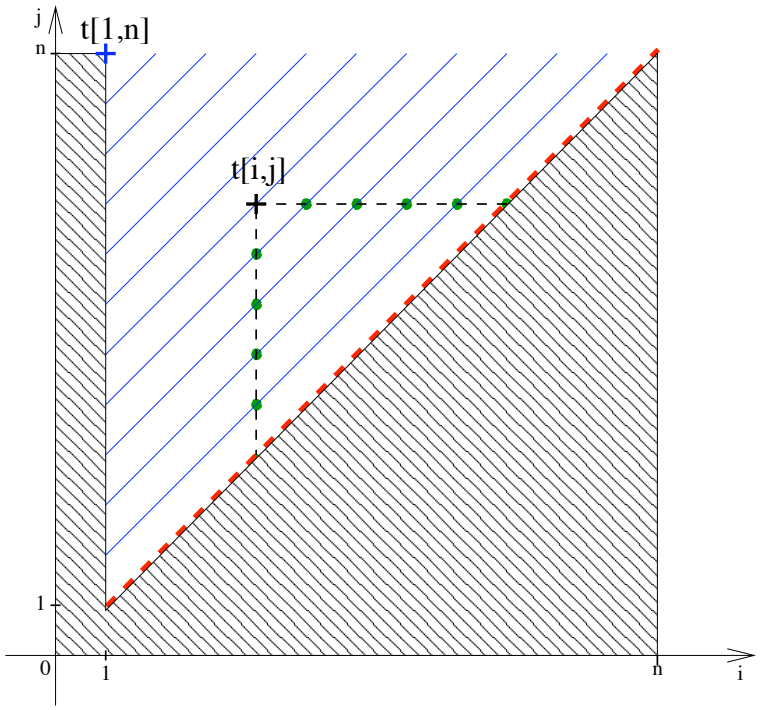


FIG. 3.5 – Diagramme (i, j) des valeurs à calculer par l'algorithme de résolution

Le nombre total d'additions est donc : $A_n = \sum_{d=1}^{n-1}((n-d)\cdot(2d))$ où $(n-d)$ correspond au nombre de $t[i,j]$ à calculer sur la diagonale $j-i=d$ et $2d$ au nombre d'additions dans le calcul de chaque $t[i,j]$ où $j-i=d$, ce qui nous donne :

$$\begin{aligned} A_n &= 2n\cdot\sum_{d=1}^{n-1} d - 2\sum_{d=1}^{n-1} d^2 \\ &= 2n\cdot\frac{n\cdot(n-1)}{2} - 2\frac{(n-1)\cdot n\cdot(2n-1)}{6} \\ &= \frac{(n-1)\cdot n\cdot(n+1)}{3} \\ &\sim n^3/3 = \Theta(n^3) \end{aligned}$$

Pour calculer le nombre T_n de tests à effectuer, il suffit de remarquer qu'on fait deux fois moins de tests que d'additions dans le calcul du minimum, ce qui nous donne $T_n = A_n/2 - C_n$ où C_n représente le nombre total de $t[i,j]$ calculés (ie le nombre d'affectations) On en déduit :

$$\begin{aligned} T_n &= A_n/2 - \sum_{d=1}^{n-1}(n-p) \\ &= \Theta(n^3) \end{aligned}$$

La complexité globale de l'algorithme est donc en $\Theta(n^3)$.

3- Dans le cas général, il faut autant de valeurs pour définir la fonction w qu'il n'y a de triangles possibles, soit $C_{n+1}^3 = \Theta(n^3)$. Comme il faudra bien lire chacune de ces valeurs

pour construire une triangulation optimale, n'importe quel algorithme doit fonctionner en une complexité d'au moins $\Theta(n^3)$. Notre algorithme est donc optimal en complexité du point de vue de l'ordre de grandeur.

4- Dans le cas où le poids d'un triangle est égal à son aire, toutes les triangulations donnent le même poids qui est l'aire du polygône. L'algorithme précédent est donc inadapté, et on peut à la place procéder par exemple comme suit :

Algorithme

- Prendre une triangulation quelconque (par exemple avec les cordes $(v_0v_i)_{2\leq i\leq n-1}$).
- Faire la somme des poids des triangles.

Cet algorithme n'effectue aucun test et $n-3$ additions (bien en dessous du $\Theta(n^3)$!).

Exercice 3.4.2. *Jeu de construction*

On veut construire une tour la plus haute possible à partir de différentes briques. On dispose de n types de briques et d'un nombre illimité de briques de chaque type. Chaque brique de type i est un parallélépipède de taille (x_i, y_i, z_i) et peut être orientée dans tous les sens, deux dimensions formant la base et la troisième dimension formant la hauteur.

Dans la construction de la tour, une brique ne peut être placée au dessus d'une autre que si les deux dimensions de la base de la brique du dessus sont *strictement inférieures* aux dimensions de la base de la brique du dessous.

Proposer un algorithme efficace pour construire une tour de hauteur maximale.

Correction.

Si on prend un parallélépipède quelconque $(x_i,\ y_i,\ z_i)$, on s'aperçoit qu'on peut le poser sur la tour de 6 manières différentes.

On peut commencer par remarquer que si on peut poser une brique sur une autre de façon à ce que la longueur de la 1re brique soit parallèle à la largeur de la 2e, alors on peut aussi poser la 1re brique sur la 2e de façon à ce que les largeurs (et donc aussi les longueurs) des deux briques soient parallèles (comme le montre la figure 3.6). On peut donc faire l'hypothèse que les briques seront posées les unes sur les autres de telle sorte que les largeurs des briques soient toutes parallèles entre elles. (*Le nombre de configurations par parallélépipède est ainsi réduit à 3.*)

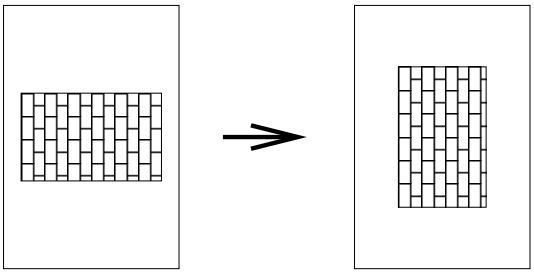


FIG. 3.6 – Si une brique posée sa largeur parallèle à la longueur de celle en dessous, alors elle peut être posée de telle sorte que leurs longueurs sont parallèles

Remarquons ensuite qu'un même parallélépipède ne peut être utilisé au plus que deux fois (dans deux configurations différentes). En effet, considérons le parallélépipède (x_i, y_i, z_i) avec

$x_i \leq y_i \leq z_i$: si on la pose une première fois, alors, dans le meilleur des cas, on peut encore poser sur la tour toutes briques (x_j, y_j, z_j) telles que $x_j < y_i$ et $y_j < z_i$ (en supposant que $x_j \leq y_j \leq z_j$). Ainsi, si les inégalités sont strictes pour x_i, y_i, et z_i, on peut encore poser le parallélépipède. Pour le poser une troisième fois, il faudrait qu'il posséde un côté de longueur strictement inférieure à x_i. Ce n'est pas le cas, est donc on ne peut le poser que deux fois. Cela a pour conséquence que la tour de hauteur maximale sera composée d'au plus $2n$ parallélépipèdes.

Dorénavant, nous allons considérer les briques (L_i, l_i, h_i) avec $L_i \geq l_i$. Nous devons alors considérer $3n$ briques au lieu de n parallélépipèdes, mais il ne reste plus qu'une configuration possible par brique (*Si le parallélépipède est (x_i, y_i, z_i) avec $x_i < y_i < z_i$, les 3 briques associées sont (z_i, y_i, x_i), (z_i, x_i, y_i) et (y_i, x_i, z_i).*).

Ainsi, on peut poser la brique (L_i, l_i, h_i) sur la brique (L_j, l_j, h_j) si et seulement si $\begin{cases} L_i < L_j \\ l_i < l_j \end{cases}$

On pose $\forall i \in \{1, \ldots, 3n\}, H_i =$ hauteur maximale parmi les tours se terminant par la i^{e} brique. On a la relation de récurrence suivante :

$$\forall i \in \{1, \ldots, 3n\}, H_i = h_i + Max_{1 \leq j \leq 3n}(H_j / L_j > L_i \text{ et } l_j > l_i) \tag{3.1}$$

L'idée est de commencer par trier les $3n$ briques par $(L_i,\ l_i)$ décroissant. Celà se fait en $O(nlog(n))$ comparaisons. L'équation précédente peut alors se réécrire :

$$\forall i \in \{1, \ldots, 3n\}, H_i = h_i + Max_{1 \leq j < i}(H_j / L_j > L_i \text{ et } l_j > l_i) \tag{3.2}$$

On calcule ainsi chaque H_i pour i allant de 2 à $3n$ en partant de $H_1 = h_1$. Le calcul de H_i nécessite 1 addition et $3i - 3$ comparaions dans le pire des cas. Le calcul de tous les $H_i, i \in 1, \ldots, 3n$ est donc linéaire en addition et quadratique en comparaison.

Enfin, il faut calculer $H = Max_{1 \leq i \leq 3n}(H_i)$ qui est la solution de notre problème, ce qui coûte $3n - 1$ comparaisons. La complexité totale est par conséquent en $O(n^2)$ pour les comparasions et en $O(n)$ pour les additions et l'algorithme est le suivant :

début
  liste $\leftarrow$ []
  pour i ***de*** 1 ***à*** n **faire**
    Ajouter les 3 configurations possibles du parallélépipède (x_i, y_i, z_i) à *liste*
  Trier $liste = (L_i, l_i, h_i)_{i \in \{1, \ldots, 3n\}}$ par (L_i, l_i) décroissant.
  $H_1 \leftarrow h_1$
  pour i ***de*** 2 ***à*** $3n$ **faire**
    $H_i \leftarrow h_i + Max_{1 \leq j < i}(H_j / L_j > L_i$ et $l_j > l_i)$
  $H \leftarrow Max_{1 \leq i \leq 3n}(H_i)$
  Renvoyer H
fin

Exercice 3.4.3. *Plus grand carré de 1*

Donner un algorithme de programmation dynamique pour résoudre le problème suivant :

Entrée : une matrice A de taille $n \times m$ où les coefficients valent 0 ou 1.

Sortie : la largeur maximum K d'un carré de 1 dans A, ainsi que les coordonnées (I, J) du coin en haut à gauche d'un tel carré (autrement dit pour tout i, j, $I \leq i \leq I + K - 1$, $J \leq j \leq J + K - 1$, $A[i, j] = 1$).

Quelle est sa complexité ?

Indication : considérer $t[i, j]$ la largeur du plus grand carré de 1 ayant (i, j) pour coin en haut à gauche.

A	1	2	3	4	5	6	7	8	9
1	1	0	0	0	0	0	0	1	0
2	1	0	1	1	1	1	1	1	0
3	1	0	1	1	1	1	0	0	0
4	0	1	1	1	1	1	0	1	0
5	0	0	1	1	1	1	0	0	1
6	0	0	1	1	1	0	0	1	1
7	0	1	0	0	0	0	0	1	1

FIG. 3.7 – Exemple où la largeur max. d'un carré de 1 vaut 4 ($(2,3)$ est le coin en haut à gauche).

Correction.

On considère la matrice A de taille $n \times m$ dont les coeficients valent 0 ou 1, et on note $C_{i,j}$ le plus grand carré ayant l'élément $A_{i,j}$ comme coin supérieur gauche, et $t[i,j]$ sa largeur. On peut alors établir la relation de récurrence suivante :

$$\forall i,j < n,m \quad \begin{cases} \text{si } A_{i,j} = 0, & t[i,j] = 0 \\ \text{sinon,} & t[i,j] = \min(t[i,j+1], t[i+1,j], t[i+1,j+1]) + 1 \end{cases}$$

En effet, le premier cas se vérifie de manière triviale, et l'on peut vérifier le second cas, en remarquant déjà que l'on a $t[i+1,j+1] \geq \max(t[i,j+1], t[i+1,j]) - 1$. Ensuite, si l'on pose $l = t[i,j+1]$ et que l'on a $l < t[i+1,j] \leq t[i+1,j+1] - 1$, alors $A_{i,j+l+1} = 0$ et donc $t[i,j] \leq l+1$. D'où, le carré de coin supérieur gauche $A_{i,j}$ et de côté $l+1$ est inclu dans l'ensemble $C_{i,j+1} \cup C_{i+1,j} \cup \{A_{i,j}\}$, il est donc entièrement constitué d'éléments valant 1. On procede de la même manière si $t[i+1,j] < t[i,j+1]$, ce qui permet de vérifier la formule donnée précédemment. Dans le cas où $t[i+1,j] = t[i,j+1] = l$, on a $t[i,j] = l+1$ si $A_{i+l,j+l} = 1$ (et dans ce cas $t[i+1,j+1] \geq l$) mais sinon, on a $t[i,j] = l-1$ (et dans ce cas $t[i+1,j+1] = l-1$). On retrouve bien notre formule. Tout ceci nous fournit l'algorithme suivant :

```
/* initialisation des bords de la matrices */
pour i de 1 à n faire t[i,m] ← A_{i,m}
pour j de 1 à m faire t[n,j] ← A_{n,j}

/* calcul des t[i,j] dans le bon ordre */
pour i de n − 1 à 1 faire
    ▶ pour j de m − 1 à 1 faire
        ▶ si A_{i,j} = 0 alors faire t[i,j] ← 0
          sinon faire t[i,j] ← min(t[i,j+1], t[i+1,j], t[i+1,j+
1]) + 1
retourner max_{i,j} t[i,j]
```

Exercice 3.4.4. *Arbres binaires de recherche optimaux*

Un *arbre binaire de recherche* est une structure de données permettant de stocker un ensemble de clés ordonnées $x_1 < x_2 < \ldots < x_n$, pour ensuite effectuer des opérations du type *rechercher*, *insérer* ou *supprimer* une clé. Il est défini par les propriétés suivantes :
(i) C'est un arbre où chaque nœud a 0, 1 ou 2 fils, et où chaque nœud stocke une des clés.
(ii) Etant donné un nœud avec sa clé x, alors les clés de son sous-arbre gauche sont strictement inférieures à x et celles de son sous-arbre droit sont strictement supérieures à x.

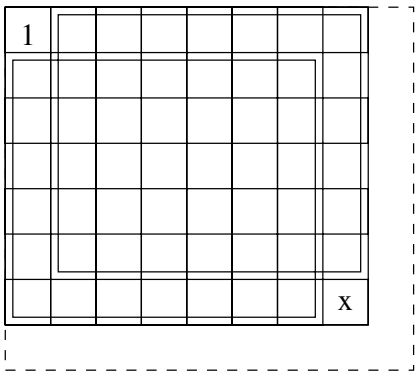


FIG. 3.8 – schéma des carrés à prendre en compte pour le calcul de $t[i, j]$

La figure 3.9 représente un arbre binaire de recherche pour les clés $x_1 < x_2 < x_3 < x_4 < x_5 < x_6 < x_7 < x_8 < x_9$.

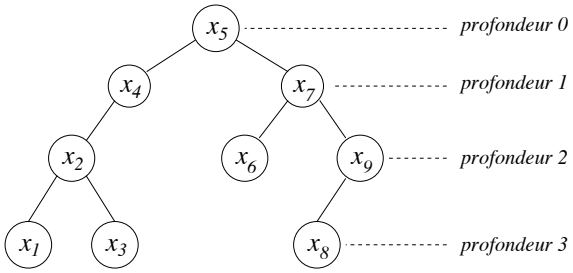


FIG. 3.9 – Exemple d'arbre binaire de recherche.

Les requètes auxquelles on s'intéresse ici sont les *recherches* de clés. Le coût de la recherche d'une clé x correspond au nombre de tests effectués pour la retrouver dans l'arbre en partant de la racine, soit exactement la profondeur de x dans l'arbre, plus 1 (la racine est de profondeur 0).

Pour une séquence fixée de recherches, on peut se demander quel est l'arbre binaire de recherche qui minimise la somme des coûts de ces recherches. Un tel arbre est appelé *arbre binaire de recherche optimal* pour cette séquence.

1 - Pour un arbre binaire de recherche fixé, le coût de la séquence ne dépend clairement que du nombre de recherches pour chaque clé, et pas de leur ordre. Pour $n = 4$ et $x_1 < x_2 < x_3 < x_4$, supposons que l'on veuille accèder une fois à x_1, 9 fois à x_2, 5 fois à x_3 et 6 fois à x_4. Trouver un arbre binaire de recherche optimal pour cet ensemble de requètes.

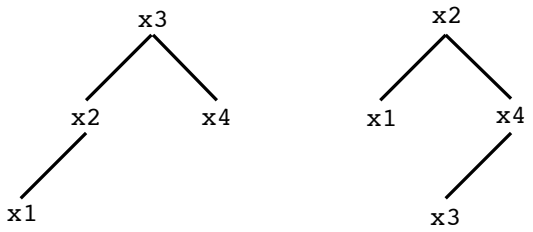
2 - Donner un algorithme en temps $\mathcal{O}(n^3)$ pour construire un arbre binaire de recherche optimal pour une séquence dont les nombres d'accès aux clés sont $c_1, c_2, \ldots, c_n$ (c_i est le nombre de fois que x_i est recherché). Justifier sa correction et sa complexité.

Indication : pour $i \leq j$, considérer $t[i, j]$ le coût d'un arbre de recherche optimal pour les clés $x_i < ... < x_j$ accédées respectivement $c_i, \ldots, c_j$ fois.

Correction.

1 - Arbres binaires optimaux

il suffit de dessiner l'ensemble des graphes possibles (il y en a 14) pour trouver les deux graphes minimaux.



2 - Algorithme

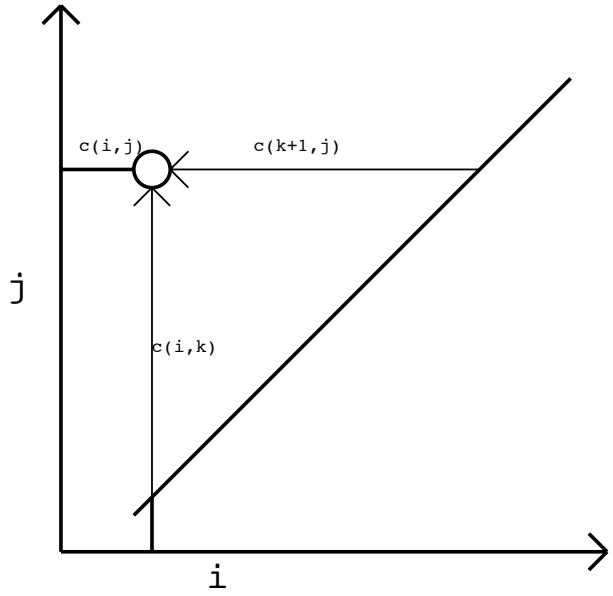
Le poids d'un arbre est le poid de ses fils, plus le poids de sa racine, plus le poids de tous les noeuds de ses fils (on leur rajoute un niveau). On appelle $t[i,j]$ le poid de l'arbre, de poids minimum contenant les sommets de i à j, et C_k le poid du sommet p. donc :

$$t[i,j] = min_{i \leq k \leq j}(t[i,k-1] + C_k + t[k+1,j] + \sum_{p=i}^{k-1}(C_p) + \sum_{p=k+1}^{j}(C_p))$$

soit :

$$t[i,j] = min_{i \leq k \leq j}(t[i,k-1] + t[k+1,j]) + \sum_{p=i}^{j}(C_p)$$

On initialise l'algo avec $t[i,i-1] = 0$.



la complexité est en $O(n^3)$: une boucle sur les i une autre sur les j puis une dernière sur les k.

Exercice 3.4.5. *Impression équilibrée*

Le problème est l'impression équilibrée d'un paragraphe sur une imprimante. Le texte d'entrée est une séquence de n mots de longueurs l_1,l_2,...,l_n (mesurées en caractères). On souhaite imprimer ce paragraphe de manière équilibrée sur un certain nombre de lignes qui contiennent un maximum de M caractères chacune. Le critère d'*équilibre* est le suivant.

Si une ligne donnée contient les mots i à j (avec $i \leq j$) et qu'on laisse exactement un espace entre deux mots, le nombre de caractères d'espacement supplémentaires à la fin de la ligne est

$M - j + i - \sum_{k=i}^{j} l_k$, qui doit être positif ou nul pour que les mots tiennent sur la ligne. L'objectif est de minimiser la somme, sur toutes les lignes *hormis la dernière*, des cubes des nombres de caractères d'espacement présents à la fin de chaque ligne.

1 - Est-ce que l'algorithme glouton consistant à remplir les lignes une à une en mettant à chaque fois le maximum de mots possibles sur la ligne en cours, fournit l'optimum ?

2 - Donner un algorithme de programmation dynamique résolvant le problème. Analyser sa complexité en temps et en espace.

3 - Supposons que pour la fonction de coût à minimiser, on ait simplement choisi la somme des nombres de caractères d'espacement présents à la fin de chaque ligne. Est-ce que l'on peut faire mieux en complexité que pour la question 2 ?

4 - (*Plus informel*) Qu'est-ce qui à votre avis peut justifier le choix de prendre les cubes plutôt que simplement les nombres de caractères d'espacement en fin de ligne ?

Correction.

Laissée au lecteur.

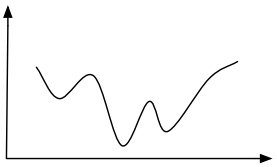
3.5 Références bibliographiques

La présentation du cours et les exercices s'inspirent du Cormen [2].

Chapitre 4

Algorithmes gloutons

Credo du glouton : à chaque étape on choisit l'optimum local.



4.1 Exemple du gymnase

Problème
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 le problème à l'aide d'un programme glouton.

Essai 1
On trie les événements par durée puis on gloutonne, *i.e.* on met les plus courts en premier s'ils sont compatibles avec ceux déjà placés.
Ceci n'est pas optimal comme le montre l'exemple de la Figure 4.1.

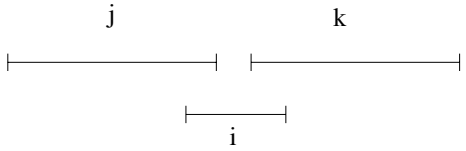


FIG. 4.1 – L'essai 1 ne conduit pas à l'optimal.

On constate que l'événement i, plus court que k et j, est placé avant ceux-ci et les empêche d'être choisis : on obtient alors une situation non optimale où seul un événement a eu lieu alors que deux événements étaient compatibles l'un avec l'autre.

Essai 2
Plutôt que de trier par durée les événements, on peut les classer par date de commencement. Encore une fois, cet algorithme n'est pas optimal, comme le montre l'exemple de la Figure 4.2.

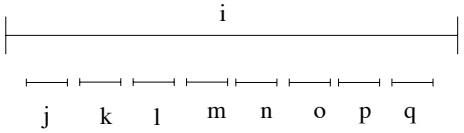


FIG. 4.2 – L'essai 2 ne conduit pas à l'optimal.

On constate que l'événement i, le premier à démarrer, empêche toute une multitude d'événements d'avoir lieu : le second essai est loin d'être optimal.

Essai 3
On peut trier les événements par ceux qui intersectent le moins possible d'autres événements. L'exemple de la Figure 4.3 montre qu'on n'est toujours pas optimal, car i est l'intervalle ayant le moins d'intersection et on n'arrive qu'à 3 événements si l'on choisit i alors qu'on peut en caser 4 : j, k, l et m.

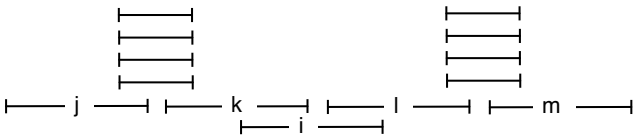


FIG. 4.3 – L'essai 3 ne conduit pas à l'optimal.

Essai 4
On peut enfin trier les événements par date de fin croissante.

Théorème 4. *Cet algorithme glouton est optimal.*

Preuve. 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,

$$O = \{f_{i_1}, f_{i_2}, ..., f_{i_k}\}$$

avec k le maximum d'événements pouvant avoir lieu dans le gymnase. Il y a deux possibilités : soit $f_{i_1} = f_1$ soit $f_{i_1} \neq f_1$. Dans ce dernier cas, alors on remplace f_{i_1} par f_1. Comme f_1 finit avant tout autre événement, alors comme f_{i_2} n'intersectait pas avec f_{i_1}, f_{i_2} n'intersecte pas avec f_1. On peut donc bien trouver une solution optimale ayant comme premier événement l'événement finissant en premier, la classe des solutions ayant f_1 en premier "domine".
Ensuite, après avoir placé f_1, on ne considère que les événements n'intersectant pas avec f_1, et on réitère la procédure sur les événements restants, d'où la preuve par récurrence. □

4.2 Route à suivre pour le glouton

- Décider du "choix" glouton pour optimiser localement le problème.
- Chercher un contre-exemple ou s'assurer que notre glouton est optimal (étapes suivantes).
- Montrer qu'il y a toujours une solution optimale qui effectue le choix glouton.

– Montrer que si l'on combine le choix glouton et une solution optimale du sous-problème qu'il reste à résoudre, alors on obtient une solution optimale.

La stratégie gloutonne est descendante (*top-down*) car il n'y a pas un choix entre plusieurs problèmes restants. On effectue un choix localement et on résout l'unique sous-problème restant ensuite.

A l'opposée, la stratégie de programmation dynamique était ascendante (*bottom-up*) car on avait besoin des résultats des multiples sous-problèmes pour pouvoir effectuer le choix.

Parenthèse sur le sac à dos entier/fractionnel

On a vu que le glouton pour le problème du sac à dos en entiers n'était pas optimal. En revanche on peut montrer qu'il est optimal pour le sac à dos en fractionnel : on peut mettre une partie d'un objet dans le sac (poudre d'or au lieu du lingot d'or).

Exemple avec trois objets, de volume w_i et de valeur c_i, avec un sac de taille $W = 5$.

Objet i	1	2	3
w_i	1	2	3
c_i	5	8	9
Rapport c_i/w_i	5	4	3

En entiers : les trois objets ne rentrent pas tous dans le sac.
- Objets 1+2 → prix 13 ;
- Objets 1+3 → prix 14 ;
- Objets 2+3 → prix 17.

Il ne faut donc pas choisir l'objet de meilleur rapport qualité/prix.

En rationnels :
La meilleur solution choisit l'objet 1, puis l'objet 2, puis les 2/3 de l'objet 3, ce qui donne un coût de $5 + 8 + 6 = 19$.
Montrer que ce glouton est optimal.

4.3 Coloriage d'un graphe

Soit un graphe $G = (V, E)$ (en anglais vertex=sommet et edge=arête). On veut colorier les sommets, mais deux sommets reliés par une arête ne doivent pas avoir la même couleur : formellement, on définit la coloration $c : V \to \{1..K\}$ telle que $(x, y) \in E \Rightarrow c(x) \neq c(y)$. Le but est de minimiser K, le nombre de couleurs utilisé

Théorème 5. *Un graphe est 2-coloriable ssi ses cycles sont de longueur paire.*

Preuve.

$\Longrightarrow$ Supposons que G contienne un cycle de longueur impaire $2k + 1$, et que, par l'absurde, il soit 2-coloriable. Pour les sommets du cycle, si 1 est bleu, alors 2 est rouge et par récurrence les impairs sont bleus et les pairs rouges ; mais 1 est à côté de $2k + 1$. Contradiction.

$\Longleftarrow$ Supposons que tous les cycles de G soient de longueur paire. On suppose que G est connexe (le problème est indépendant d'une composante connexe à l'autre). On parcourt G en largeur.

Soit $x_0 \in G$, $X_0 = \{x_0\}$ et $X_{n+1} = \bigcup_{y \in X_n} \{\text{fils de y dans l'ordre de parcours}\}$.
Si par l'absurde $\exists k \exists m \exists z \in X_{2k} \cap X_{2m+1}$, alors z est à distance $2k$ de x_0 et $2m + 1$ de x_0. Le cycle correspondant contient$(2k - 1) + (2m) + 2 = 2(m + k) + 1$ éléments. Contradiction.
Donc $\forall k \forall m X_{2k} \cap X_{2m+1} = \emptyset$; on colorie en bleu les éléments des X_i avec i impair et en rouge les éléments des X_i avec i pair.

Il ne peut y avoir deux éléments reliés appartenant à X_i,X_j avec i et j de même parité car sinon ils formeraient en remontant à x_0 un cycle de longueur $i + j + 1$ (impaire).
On a donc 2-colorié G. □

On appelle *biparti* un graphe 2-coloriable : les sommets sont partitionnés en deux ensembles, et toutes les arêtes vont d'un ensemble à l'autre. On s'intéresse maintenant au coloriage d'un graphe général.

4.3.1 Algorithme glouton 1

Pour colorier un graphe général :
- prendre les sommets dans un ordre au hasard ;
- leur attribuer la plus petite valeur possible, i.e. la plus petite valeur qui n'a pas déjà été attribuée à un voisin.

Soit K le nombre de couleurs utilisées. Alors $K \leq \Delta(G) + 1$, où $\Delta(G)$ est le degré maximal d'un sommet, le degré d'un sommet étant le nombre d'arêtes auxquelles appartient ce sommet. En effet, au moment où on traite un sommet donné, il a au plus $\Delta(G)$ voisins déjà coloriés, donc l'algorithme glouton n'est jamais forcé d'utiliser plus de $\Delta(G) + 1$ couleurs.

Pour une clique (un graphe complet, avec toutes les arêtes possibles), il n'est pas possible de faire mieux.

Ce glouton n'est pas optimal, par exemple sur un graphe biparti, si l'on colorie d'abord 1 puis 4, on a besoin de 3 couleurs au lieu de 2.

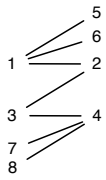
4.3.2 Algorithme glouton 2

- trier les sommets par degré décroissant ;
- prendre les sommets dans cet ordre ;
- leur attribuer la plus petite valeur possible.

Soit $n = |V|$ le nombre de sommets, et d_i le degré du sommet v_i. On montre alors que $K \leq \max_{1 \leq i \leq n} \min(d_i + 1, i)$. En effet, au moment où on colorie le i-ème sommet v_i, il a au plus $\min(d_i, i - 1)$ voisins qui ont déjà été coloriés, et donc sa propre couleur est au plus $1 + \min(d_i, i - 1) = \min(d_i + 1, i)$. En prenant le maximum sur i de ces valeurs, on obtient la borne demandée.

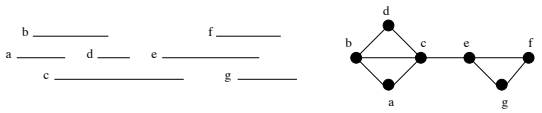
Cette borne suggère de se débarrasser d'abord des plus gros degrés, qui disparaissent ainsi de la complexité tant que $\min(d_i + 1, i) = i$. Comme on ne sait pas a priori jusqu'à quand privilégier les grands degrés, on a trié l'ensemble des sommets par degrés décroissants dans l'algorithme glouton.

On peut ici encore forcer l'algorithme à effectuer un mauvais choix sur un graphe biparti (choisir 1 puis 4, ce qui forcera les 3 couleurs).



4.3.3 Graphe d'intervalles

L'algorithme glouton, avec un ordre astucieux, est optimal pour les graphes d'intervalles. Etant donnée une famille d'intervalles (disons sur la droite réelle), on définit un graphe dont les sommets sont les intervalles, et dont les arêtes relient les sommets représentant les intervalles qui s'intersectent. Voici un exemple :



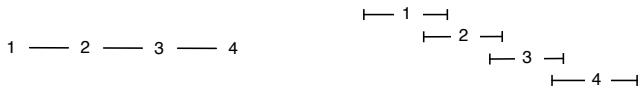
Notons que ce problème est proche de celui du gymnase : on considère que l'on dispose d'un ensemble d'événements et on cherche le nombre minimal de gymnases nécessaires pour caser tous ces événements.

On montre que dans le cas de ces graphes particuliers (bien évidemment, tout graphe ne peut pas être représenté par une famille d'intervalles), l'algorithme glouton qui énumère les sommets du graphe d'intervalles selon l'ordre donné par les extrémités gauches des intervalles (a, b, c, d, e, f, g sur l'exemple) colorie le graphe de manière optimale.

Sur l'exemple, on obtient le coloriage $1, 2, 3, 1, 1, 2, 3$ qui est bien optimal.

Dans le cas général, exécutons l'algorithme glouton avec l'ordre spécifié des sommets sur un graphe G. Supposons que le sommet v reçoive la couleur maximale k. L'extrémité gauche de v doit donc intersecter $k-1$ autres intervalles qui ont reçus les couleurs 1 à $k-1$, sinon on colorierai v d'une couleur $c \leq k-1$. Tous ces intervalles s'intersectent donc (ils ont tous le point a, extrémité gauche de v, en commun), et cela signifie que G possède une clique (un sous-graphe complet) de taille k. Le cardinal maximal d'une clique de G est donc supérieur ou égal à k. Comme tous les sommets d'une clique doivent recevoir une couleur différente, on voit que l'on ne peut pas faire mieux que k couleurs. L'algorithme glouton est donc optimal.

En revanche, on peut montrer que l'ordre est important, car une fois de plus, sur un graphe biparti, on pourrait forcer l'algorithme à faire un mauvais choix (1 puis 4) si l'on ne procédait pas de gauche à droite.

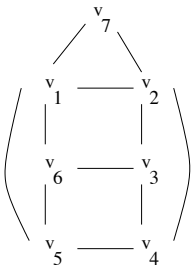


4.3.4 Algorithme de Brelaz

L'idée intuitive est de colorier en priorité les sommets ayant beaucoup de sommets déjà coloriés. On définit le degré-couleur d'un sommet comme son nombre de voisins déjà coloriés. Le degré-couleur, qui va évoluer au cours de l'algorithme, est initialisé à 0 pour tout sommet.

- Prendre parmi les sommets de degré-couleur maximal un sommet v de degré maximal, et lui attribuer la plus petite valeur possible.
- Mettre à jour le degré-couleur des noeuds voisins de v.

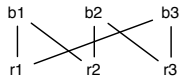
Il s'agit d'un glouton sur (degré-couleur, degré), appelé algorithme de Brélaz. Considérons l'exemple ci-dessous :



On prend au départ un sommet de degré maximal, par exemple v_1, et on lui donne la couleur 1. Le degré-couleur de v_2, v_5, v_6 et v_7 passe à 1, on choisit v_2 qui est de degré maximal parmi ces trois sommets, et on lui donne la couleur 2. Maintenant, v_7 est le seul sommet de degré-couleur 2, on le choisit et on lui attribue la couleur 3. Tous les sommets restants (non coloriés) ont le même degré-couleur 1 et le même degré 3, on choisit v_3 au hasard, et on lui donne la couleur 1. Puis v_4, de degré-couleur 2, reçoit la couleur 3. Enfin v_5 reçoit la couleur 2 et v_6 la couleur 3. Le graphe est 3-colorié, c'est optimal.

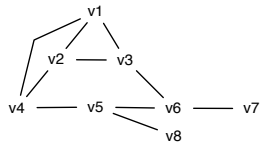
Théorème 6. *L'algorithme de Brelaz est optimal sur les graphes bipartis, i.e. réussit toujours à les colorier avec deux couleurs.*

Preuve. On considère un graphe biparti $G = (V, E)$, $V = B \cup R$: toute arête de E relie un sommet de B (bleu) et un sommet de R (rouge). Remarquons d'abord que les deux premiers algorithmes gloutons peuvent se tromper : par exemple soit G avec $B = \{b_1, b_2, b_3\}$, $R = \{r_1, r_2, r_3\}$, et les six arêtes (b_i, r_i), $1 \leq i \leq 3$, (b_1, r_2), (b_2, r_3) et (b_3, r_1). Tous les sommets sont de degré 2. Si on commence par colorier un sommet de B, disons b_1, puis un sommet non adjacent de R, r_3, avec la même couleur 1, on ne pourra pas terminer avec seulement 2 couleurs. Par contre, en utilisant la variante de Brelaz avec les degrés-couleur sur ce petit exemple, après b_1 on doit colorier soit r_1 soit r_2, et on ne se trompe pas.



Pour le cas général, considérons $G = (V, E)$, $V = B \cup R$ connexe biparti, et colorions le avec l'algorithme de Brelaz. On commence par colorier un sommet, disons, de B, en bleu. Puis on colorie nécessairement un sommet adjacent (de degré-couleur 1), donc de R, en rouge. Par la suite, on ne colorie que des sommets de degré-couleur non nul, dont tous les voisins sont de la même couleur, et c'est l'invariant qui donne le résultat. □

Contre-exemple Brelaz n'est pas optimal sur les graphes quelconques :



Brelaz peut choisir en premier v_4, qui est de degré maximum 3 (colorié 1). Parmi les noeuds de degré-couleur 1, il choisit alors v_5, qui est colorié en 2. Puis v_6 est colorié en 1. Il choisit alors v_1 parmi les noeuds de degré-couleur 1 et de degré 2, qui est colorié en 2. On est alors obligé d'utiliser des couleurs 3 et 4 pour v_2 et v_3 alors que ce graphe est 3-coloriable (v_1, v_5, v_7 en 1, v_2, v_6, v_8 en 2, v_3, v_4 en 3).

Remarque Le problème du coloriage des graphes est NP-complet.

4.4 Théorie des matroïdes

Cette section présente une petite théorie basée sur les matroïdes permettant de savoir si un algorithme glouton est optimal pour un problème. Cela ne couvre pas tous les cas d'application de la méthode gloutonne (comme l'exemple du gymnase par exemple).

4.4.1 Matroïdes

Le mot "matroïde" a été introduit en 1935 par Whitney, dans des travaux sur l'indépendance linéaire des colonnes de matrices. On pourra se référer au papier de Whitney pour en savoir plus : "On the abstract properties of linear dependence", *American Journal of Mathematics*, 57 :509-533, 1935.

Définition 1. *$(S, \mathcal{I})$ est un matroïde si S est un ensemble de n éléments, $\mathcal{I}$ est une famille de parties de S, $\mathcal{I} \subset \mathcal{P}(S)$, vérifiant :*
- l'hérédité *: $X \in \mathcal{I} \Rightarrow \forall Y \subset X, Y \in \mathcal{I}$*
- l'échange *: $(A, B \in \mathcal{I},\ |A| < |B|) \Rightarrow \exists x \in B \setminus A\ tq\ A \cup \{x\} \in \mathcal{I}$*

Si $X \in \mathcal{I}$, on dit que X est un indépendant.

Exemples

1. Les familles libres d'un espace vectoriel
2. Les forêts[1] d'un graphe.

Soit $G = (V, E)$ un graphe , $|V| = n$, on définit alors $S = E$ et $\mathcal{I} = \{A \subset E / A$ sans cycles$\}$, c.a.d. qu'un ensemble d'arêtes est un indépendant *ssi* c'est une forêt.

- L'hérédité est évidente, car un sous-ensemble d'une forêt est une forêt (en enlevant des arêtes à une forêt on ne crée pas de cycles).
- L'échange :
  Soient A et B des forêts de $G\ tq\ |A| < |B|$. $|A|$ représente le nombre d'arbres dans la forêt A, et tous les sommets doivent appartenir à un arbre (*i.e.* un sommet non relié par une arête est un arbre constitué d'un unique sommet). Alors A (resp. B) contient $n-|A|$ (resp. $n-|B|$) arbres (avec chaque arête ajoutée à A, on relie deux arbres, donc on décrémente de un le nombre d'arbres de A).
  Ainsi, B contient moins d'arbres que A. Il existe donc un arbre T de B qui n'est pas inclus dans un arbre de A, c.a.d. qu'il existe deux sommets u et v de T qui n'appartiennent pas au même arbre de A. Sur le chemin de u à v dans T, il existe une paire de sommets consécutifs qui ne sont pas dans le même arbre de A (ils sont de chaque côté du *pont* qui traverse d'un arbre de A à l'autre). Soit (x, y) l'arête en question. Alors $A \cup \{(x, y)\}$ est sans cycle, i.e. $A \cup \{(x, y)\} \in \mathcal{I}$, ce qui complète la preuve.

Définition 2. *Soit $F \in \mathcal{I}$. $x \notin F$ est une extension de F si $F \cup \{x\}$ est un indépendant.*

Sur l'exemple de la forêt, une arête reliant deux arbres distincts est une extension.

Définition 3. *Un indépendant est dit maximal s'il est maximal au sens de l'inclusion,* i.e. *s'il ne possède pas d'extensions.*

Dans l'exemple, une forêt est maximale si elle possède un seul arbre. On parle alors d'arbre couvrant.

Lemme 2. *Tous les indépendants maximaux ont même cardinal.*

Si ce n'était pas le cas, on pourrait les étendre par la propriété d'échange.

[1] Une forêt est un ensemble d'arbres, i.e. un graphe non-orienté sans cycles

4.4.2 Algorithme glouton

Fonction de poids On pondère le matroïde avec une fonction de poids ; on parle alors de matroïde *pondéré* :
$x \in S \mapsto w(x) \in \mathbb{N}$; $\quad X \subset S\ ,\ w(X) = \sum_{x \in X}\ w(x)$

Question Trouver un indépendant de poids maximal.
Par glouton : trier les éléments de S par poids décroissant.
$A \leftarrow \emptyset$[2]
For $i = 1$ *to* $|S|$
si $A \cup \{s_i\} \in \mathcal{I}$ *alors* $A \leftarrow A \cup \{s_i\}$

Théorème 7. *Cet algorithme donne la solution optimale.*

Preuve. Soit s_k le premier élément indépendant de S, *i.e.* le premier indice i de l'algorithme précédent tel que $\{s_i\} \subset \mathcal{I}$.

Lemme 3. *Il existe une solution optimale qui contient* s_k.

En effet : Soit B une solution optimale, i.e. un indépendant de poids maximal.

1. Si $s_k \in B$, c'est bon.
2. Si $s_k \notin B$, soit $A = \{s_k\} \in \mathcal{I}$. On applique $|B| - 1$ fois la propriété d'échange (tant que $|B| > |A|$), d'où l'indépendant $A = B \backslash \{b_i\} \cup \{s_k\}$, où $\{b_i\}$ est l'élément restant de B (car $|A| = |B|$ et A contient déjà s_k).
   On a $w(A) = w(B) - w(b_i) + w(s_k)$, or $w(s_k) \geq w(b_i)$ car b_i est indépendant (par l'hérédité), et b_i a été trouvé après s_k, par ordre de poids décroissants.
   Donc $w(A) \geq w(B)$, d'où $w(A) = w(B)$, l'indépendant A est optimal.

Puis, par récurrence on obtient que glouton donne la solution optimale : on se restreint à une solution contenant $\{s_k\}$, et on recommence avec $S' = S \backslash \{s_k\}$ et $\mathcal{I}' = \{X \subset S' / X \cup \{s_k\} \in \mathcal{I}\}$. □

Retour à l'exemple des forêts d'un graphe Le théorème précédent assure de la correction de l'algorithme glouton dû à Kruskal pour construire un arbre de poids minimal : trier toutes les arêtes par poids croissant, et sélectionner au fur et à mesure celles qui ne créent pas de cycle si les on ajoute à l'ensemble courant. Bien sûr, il faudrait discuter d'une structure de données adaptées pour vérifier la condition "ne crée pas de cycle" rapidement. Avec un tableau, on arrive facilement à vérifier la condition en $O(n^2)$, et on peut faire mieux avec des structures de données adaptées.

4.5 Ordonnancement

On donne ici un exemple d'algorithme glouton, dont on démontre l'optimalité avec la théorie des matroïdes. C'est un problème d'ordonnancement à une machine.

Données : n tâches $T_1 ... T_n$ de durée 1. Deadlines associées : $d_1, ..., d_n$. Pénalités associées si le deadline est dépassé : $w_1, ..., w_n$.

Ordonnancement : $\sigma : \mathcal{T} \rightarrow \mathbb{N}$ qui à chaque tâche associe le top de début d'exécution. Si une tâche T_i finit après son deadline d_i, alors cela crée la pénalité w_i.
But : minimiser la pénalité totale, c'est à dire la somme sur les tâches en retard des pénalités.

[2] $\emptyset$ est un indépendant, par l'hérédité

Intérêt de l'ordonnancement : sur les chaînes de production en usine, optimisation des calculs sur des systèmes multiprocesseurs. Ici, le problème est simplifié (et cela permet que glouton marche).

On va donc chercher le meilleur ordre pour l'exécution des tâches.

Définition 4. *tâche à l'heure : tâche finie avant le deadline ;*
tâche en retard : tâche finie après le deadline.

Pour notre problème on va se restreindre :

1. Aux ordonnancements où les tâches à l'heure précèdent les tâches en retard : en effet, cela ne restreint pas le problème, car si on avait une tâche en retard qui s'exécutait avant une tâche à l'heure, les permuter ne changerait rien à la pénalité totale.
2. Aux ordonnancement où les tâches à l'heure sont classées par ordre de deadline croissant : de même que précédemment cela ne restreint pas le problème, car si on avait une tâche T_i de deadline d_i et une tâche T_j de deadline d_j, avec $d_i > d_j$ et T_i s'exécutant avant T_j, alors si on permute les ordres d'exécution de T_i et de T_j , T_i reste à l'heure, car $d_i > d_j$ et T_j était à l'heure, de plus T_j reste à l'heure car on avance son moment d'exécution.

On dira qu'on a l'**ordre canonique** si les deux conditions précédentes sont vérifiées.

Exemple : 7 tâches :

T_i	1	2	3	4	5	6	7
d_i	4	2	4	3	1	4	6
w_i	7	6	5	4	3	2	1

Remarque : minimiser la pénalité totale des tâches en retard $\Leftrightarrow$ maximiser la pénalité des tâches à l'heure.

Glouton : On commence par trier les tâches par w_i décroissants puis on "gloutonne", c'est à dire qu'on prend la tâche si elle tient sans pénalité (on regarde si l'ordre canonique marche).

Application : $\{1\}; \{2,1\}; \{2,1,3\}; \{2,4,1,3\}$. Par contre on ne peut pas rajouter la tâche 5 car l'ordre canonique $\{5,2,4,1,3\}$ ne respecte pas la deadline de la tâche 3. On ne peut pas rajouter 6 non plus, mais $\{2,4,1,3,7\}$ est un ensemble de tâches à l'heure. La solution optimale est donc 2,4,1,3,7 et la pénalité minimale est 5.

Ici, l'idée a été de *réordonner* à chaque fois que l'on rajoutait une tâche.

Définition 5. *Un ensemble de tâches est indépendant $\Leftrightarrow$ elles peuvent être exécutées en étant toutes à l'heure.*

Lemme 4. *Soit A un ensemble de tâches. $N_t(A)$ = nombre de tâches de deadline $\leq t$.*
Les propositions suivantes sont équivalentes :

1. *A est indépendant.*
2. *$\forall t = 1, 2, ..., n,\ \ N_t(A) \leq t$.*
3. *Ordre canonique $\rightarrow$ pas de tâches en retard.*

Preuve.

- $1 \Rightarrow 2$: Par contraposée : supposons que $N_t > t$, alors il y a au moins une tâche qui se déroulera après le temps t, donc qui sera pénalisée, donc A ne sera pas indépendant.
- $2 \Rightarrow 3$: trivial
- $3 \Rightarrow 1$: définition

□

Proposition 1. *On a un matroïde.*

Preuve.

- Hérédité : trivial
- Echange : Soit A, B indépendants tels que $|A| < |B|$. Existe-t-il x appartenant à B tel que $A \cup \{x\}$ soit indépendant ?
  Idée : Comparer $N_t(A)$ et $N_t(B)$ pour $t = 1..n$.
  Pour $t = 0$, $N_0(A) = N_0(B) = 0$. Pour $t = n$, $N_n(A) = |A| < |B| = N_n(B)$. On cherche le plus grand $0 \leq t \leq n$ tel que $N_t(A) \geq N_t(B)$. On sait que $t < n$, et pour tout $t' > t$, $N_{t'}(A) < N_{t'}(B)$. Notamment, dans B il y a plus de tâches de deadline $t+1 \leq n$ que dans A $\Rightarrow$ On choisit $x \notin A$ de deadline $t + 1$, et alors $A \cup \{x\}$ est un indépendant.

□

Complexité La complexité de l'ordonnancement est en $O(n^2)$ car il y a n itérations au glouton et la vérification d'indépendance se fait en temps n.

Next year Dans le cours d'algorithmique parallèle, plein de nouveaux jolis algorithmes d'ordonnancement dans un cadre un peu plus complexe (plusieurs machines...).

4.6 Exercices

Exercice 4.6.1. *Couverture par intervalles*

Etant donné un ensemble $\{x_1, \ldots, x_n\}$ de n points sur une droite, décrire un algorithme qui détermine le plus petit ensemble d'intervalles fermés de longueur 1 qui contient tous les points donnés. Prouver la correction de votre algorithme et donner sa complexité.

Correction.

début
  Trier dans l'ordre croissant les x_i ;
  $X \leftarrow \{x_1, \ldots, x_n\}$;
  $I \leftarrow \emptyset$;
  tant que $X \neq \emptyset$ **faire**
    $x_k \leftarrow \min(X)$;
    $I \leftarrow I \cup \{[x_k, x_k + 1]\}$;
    $X \leftarrow X \backslash [x_k, x_k + 1]$
  Retourner I ;
fin

La complexité est $n \log n$ (tri) + n (parcours de X), c'est-à-dire $O(n \log n)$

Théorème : Cet algorithme est optimal.

Méthode 1 : avec des matroïdes. Cette méthode ne fontionne pas. Il est impossible d'exhiber un matroïde (on ne peut pas prouver la propriété d'échange)

Méthode 2 : par récurrence sur $|X|$. On ajoute un à un les élements. On montre que le premier est bien contenu dans l'ensemble fournit par l'algorithme.

- **Initialisation :** Si on a qu'un seul point, l'algorithme renvoie un seul intervalle : c'est une solution optimale.
- Soient un recouvrement optimal $I_{opt} = \{[a_1, a_1 + 1], \ldots, [a_p, a_p + 1]\}$ avec $a_1 < \cdots < a_p$ et le recouvrement donné par notre algorithme $I_{glou} = \{[g_1, g_1 + 1], \ldots, [g_m, g_m + 1]\}$ avec $g_1 < \cdots < g_m$. Montrons que $m = p$.

– On pose $I = \Big(I_{opt} \backslash \big\{[a_1, a_1+1]\big\}\Big) \cup \big\{[g_1, g_1+1]\big\}$.
Puisque $g_1 = x_1$, I est un recouvrement de X car $a_1 \leq x_1 \Rightarrow a_1+1 \leq x_1+1 = g_1+1$ donc $X \cap [a_1, a_1+1] \subseteq X \cap [g_1, g_1+1]$; autrement-dit $I_{opt} \backslash \big\{[a_1, a_1+1]\big\}$ est un recouvrement de $X \backslash \big\{[g_1, g_1+1]\big\}$.
– Par hypothèse de récurrence, sur $X \backslash [g_1, g_1+1]$ une solution optimale est donnée par glouton : $\big\{[g_2, g_2+1], \ldots, [g_m, g_m+1]\big\}$ car $|X| \backslash [g_1, g_1+1] \leq n-1$
– Comme $I_{opt} \backslash \big\{[a_1, a_1+1]\big\}$ est un recouvrement de $X \backslash [g_1, g_1+1]$ avec $p-1$ intervalles, on a :

$$\left.\begin{array}{r} p-1 \geq m-1 \Rightarrow p \geq m \\ \left.\begin{array}{l} I_{opt} \text{ optimal sur } X \\ I_{glou} \text{ recouvrement de } X \end{array}\right\} \Rightarrow m \geq p \end{array}\right\} \Rightarrow m = p$$

Puisque p est le nombre d'intervalles que renvoie un algorithme optimal, et que $m = p$, alors, notre glouton est optimal.

Exercice 4.6.2. *Codage de Huffman*

Soit Σ un alphabet fini de cardinal au moins deux. Un *codage binaire* est une application injective de Σ dans l'ensemble des suites finies de 0 et de 1 (les images des lettres de Σ sont appelées *mots de code*). Il s'étend de manière naturelle par concaténation en une application définie sur l'ensemble Σ^* des mots sur Σ. Un codage est dit *de longueur fixe* si toutes les lettres dans Σ sont codées par des mots binaires de même longueur. Un codage est dit *préfixe* si aucun mot de code n'est préfixe d'un autre mot de code.

1 - Le décodage d'un codage de longueur fixe est unique. Montrer qu'il en est de même pour un codage préfixe.

2 - Représenter un codage préfixe par un arbre binaire dont les feuilles sont les lettres de l'alphabet.

On considère un texte dans lequel chaque lettre c apparaît avec une fréquence $f(c)$ non nulle. A chaque codage préfixe de ce texte, représenté par un arbre T, est associé un coût défini par :

$$B(T) = \sum_{c \in \Sigma} f(c) l_T(c)$$

où $l_T(c)$ est la taille du mot binaire codant c. Si $f(c)$ est exactement le nombre d'occurences de c dans le texte, alors $B(T)$ est le nombre de bits du codage du texte.

Un codage préfixe représenté par un arbre T est *optimal* si, pour ce texte, il minimise la fonction B. Montrer qu'à un codage préfixe optimal correspond un arbre binaire où tout nœud interne a deux fils. Montrer qu'un tel arbre a $|\Sigma|$ feuilles et $|\Sigma| - 1$ nœuds internes.

3 - Montrer qu'il existe un codage préfixe optimal pour lequel les deux lettres de plus faibles fréquences sont soeurs dans l'arbre (autrement dit leurs codes sont de même longueur et ne diffèrent que par le dernier bit) - *Propriété de choix glouton.*

Etant donnés x et y les deux lettres *de plus faibles fréquences* dans Σ, on considère l'alphabet $\Sigma' = \Sigma - \{x, y\} + \{z\}$, où z est une nouvelle lettre à laquelle on donne la fréquence $f(z) = f(x) + f(y)$. Soit T' l'arbre d'un codage optimal pour Σ', montrer que l'arbre T obtenu à partir de T' en remplaçant la feuille associée à z par un nœud interne ayant x et y comme feuilles représente un codage optimal pour Σ - *Propriété de sous-structure optimale.*

4 - En déduire un algorithme pour trouver un codage optimal et donner sa complexité. A titre d'exemple, trouver un codage optimal pour $\Sigma = \{a, b, c, d, e, g\}$ et $f(a) = 45$, $f(b) = 13$, $f(c) = 12$, $f(d) = 16$, $f(e) = 9$ et $f(g) = 5$.

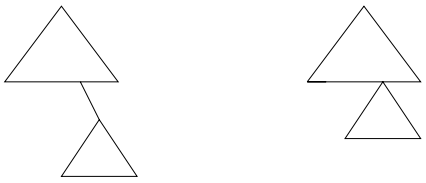


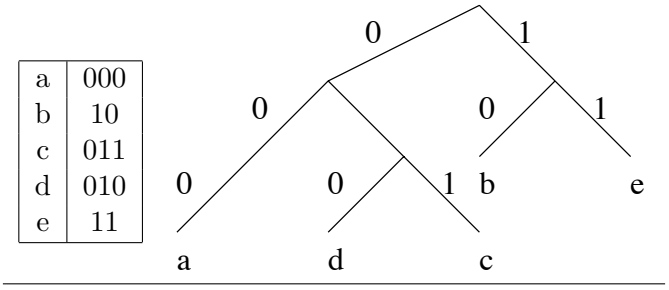
FIG. 4.4 – Chaque codage préfixe optimal correspond à un arbre binaire dont les nœuds internes ont deux fils

Correction.

1 - Dans un codage préfixe, le décodage est unique. En effet, si x peut se décoder de deux façons, alors $x = ua$ et $x = vb$, et donc u est préfixe de x ou v est préfixe de u, donc $u = v$ pour un codage préfixe.

2 - On peut représenter le codage préfixe par un arbre : on associe 0 à chaque branche gauche et 1 à chaque branche droite. Pour chaque caractère, on lit le code à partir de la racine. Ainsi, l'arbre représenté dans la figure 4.1 correspond au codage des lettres a, b, c, d et e.

a	000
b	10
c	011
d	010
e	11



TAB. 4.1 – Codage de a,b,c,d et e

Chaque codage préfixe optimal correspond à un arbre binaire où tout nœud interne a deux fils. En effet, s'il existe un nœud qui n'a qu'un fils, alors on diminue le $L_T(c)$ en remontant le sous-arbre correspondant : soit la feuille correspondant à la lettre c était dans le sous-arbre, dans ce cas $L'_T(c) = L_T(c) - 1$, soit elle n'y était pas et $L'_T(c) = L_T(c)$ (cf figure 4.4).

Un tel arbre a par construction $|\Sigma|$ feuilles, puisqu'il code $|\Sigma|$ lettres. Montrons par récurrence qu'il a $|\Sigma| - 1$ nœuds :

- Un arbre à 2 feuilles a 1 nœud.
- Si l'arbre est constitué de deux sous-arbres de i et j feuilles, alors les deux sous-arbres ont $i-1$ et $j-1$ nœuds (par hypothèse de récurrence), et l'arbre total a donc $i-1+j-1+1 = i+j-1$ nœuds, la propriété est donc vraie.

3 - Montrons qu'il existe un codage préfixe optimal pour lequel les deux lettres de plus faibles fréquences sont sœurs dans l'arbre. Soit T un codage optimal. On a $\forall T', B(T) \leq B(T')$. Soient x et y les lettres de fréquences les plus basses, et a et b deux lettres sœurs de profondeur maximale dans T. Construisons alors un arbre T'' en inversant les positions de x et de a, ainsi que celles de y et de b.

On veut montrer que T'' est optimal, c'est à dire que $B(T) = B(T'')$.

$$\begin{aligned} B(T) - B(T'') &= f(a)l_T(a) + f(b)l_T(b) + f(x)l_T(x) + f(y)l_T(y) \\ &\quad -f(a)l_{T''}(a) - f(b)l_{T''}(b) - f(x)l_{T''}(x) - f(y)l_{T''}(y) \end{aligned}$$

Comme on a les égalités suivantes :

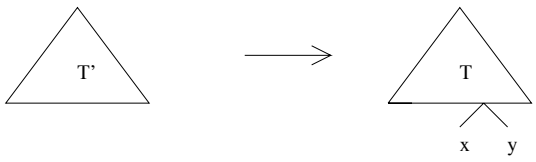
$$\begin{aligned} l_{T''}(a) &= l_T(x) \\ l_{T''}(b) &= l_T(y) \\ l_{T''}(x) &= l_T(a) \\ l_{T''}(y) &= l_T(b) \end{aligned}$$

On en déduit l'égalité :

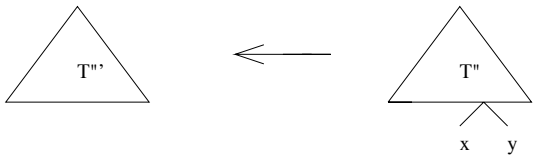
$$B(T) - B(T'') = [f(a) - f(x)][l_T(a) - l_T(x)] + [f(b) - f(y)][l_T(b) - l_T(y)]$$

$f(a) \geq f(x)$ et $f(b) \geq f(y)$, $l_T(a) \geq l_T(x)$ et $l_T(b) \geq l_T(y)$, donc $B(T) - B(T'') \geq 0$.
Donc comme T est un codage optimal, alors T'' aussi. On a donc un codage préfixe optimal dans lequel les deux lettres de plus faibles fréquences sont sœurs dans l'arbre. (*Propriété de choix glouton*).

On considère maintenant $\Sigma' = \Sigma \setminus \{x, y\} + \{z\}$, où z est une nouvelle lettre de fréquence $f(z) = f(x) + f(y)$. Soit T' l'arbre d'un codage optimal pour Σ', on veut montrer que l'arbre T obtenu à partir de T' en remplaçant la feuille associée à z par un nœud de feuilles x et y est optimal pour Σ.



Soit T'' un arbre optimal pour Σ. D'après la propriété de choix glouton, on peut supposer que x et y sont sœurs dans T''. On peut alors construire un arbre T''' en remplaçant le nœud de fils x et y par une feuille z.



On a alors $B(T''') = B(T'') - f(x) - f(y)$.
Or T' est optimal, donc $B(T') \leq B(T''')$. Comme $B(T) = B(T') + f(x) + f(y)$, on en déduit que $B(T) \leq B(T'')$. Donc T est optimal car T'' est optimal.

4 - On en déduit l'algorithme suivant pour le codage de Huffman :

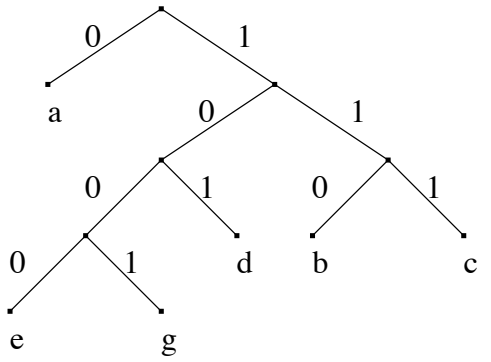
```
Huffman(Σ, f)
début
    F ← tas_binaire(Σ, f) ;
    n ← |Σ| ;
    pour i de 1 à n − 1 faire
        z ←allouer_noeud() ;
        x ←extraire_min(F) ;
        y ←extraire_min(F) ;
        z(gauche) ← x ;
        z(droite) ← y ;
        f(z) ← f(x) + f(y) ;
        Insérer(F, z, f(z)) ;
    Retourner extraire_min(F) ;
fin
```

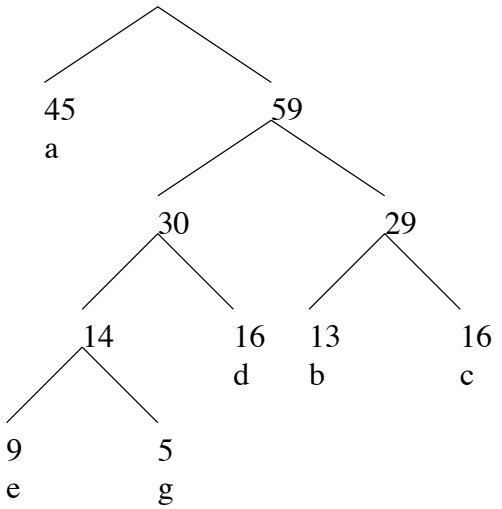
Calculons la complexité de cet algorithme :
Pour un tas binaire, le coût des opérations est le suivant :

- Savoir si la file est vide $O(1)$.
- Insérer un élement $O(\log n)$
- Trouver un élément de clé minimale $O(1)$
- Extraire l'élément de clé minimale $O(\log n)$

La complexité d'une itération de la boucle est donc en $O(\log n)$.
La complexité totale est donc en $O(n \log n)$, car il y a n itérations de la boucle, et que la construction du tas est en $O(n \log n)$.

Voici l'arbre que l'on obtient en appliquant cet algorithme à $\Sigma = \{a, b, c, d, e, g\}$ avec $f(a) = 45$, $f(b) = 13$, $f(c) = 12$, $f(d) = 16$, $f(e) = 9$, $f(g) = 5$:





4.7 Références bibliographiques

Tout le chapitre (cours et exercice *Codage de Huffman*) s'inspire du Cormen [2], à l'exception des algorithmes de coloriage de graphes, tirés du West [11].

Chapitre 5

Tri

5.1 Tri fusion

Tri fusion : n éléments à trier, appel de TF(1,n) → appels récursifs de TF(1,$\lfloor n/2 \rfloor$) et TF($\lfloor n/2 \rfloor + 1$,n) puis fusion des deux sous listes triées.

Algorithme de fusion de deux listes de taille p et q : on compare les deux premiers des deux listes, on prend le plus petit et on recommence avec les deux listes restantes. On fait un total de $p + q - 1$ comparaisons.

Complexité : ici, c'est en terme de comparaisons qu'il faut compter. Comme la somme des tailles des deux listes à fusionner est toujours $\lfloor n/2 \rfloor + \lceil n/2 \rceil = n$, la fusion coûte $n - 1$ comparaisons :

$$comp_{TF}(n) = comp_{TF}(\lfloor n/2 \rfloor) + comp_{TF}(\lceil n/2 \rceil) + n - 1$$

avec $comp_{TF}(1) = 0$.

Donc, d'après le Master Théorème, $comp_{TF}(n) = O(n \log_2 n)$. En fait, on peut montrer par une récurrence un peu technique que

$$comp_{TF}(n) = n\lceil \log_2 n \rceil - 2^{\lceil \log_2 n \rceil} + 1$$

Il faut distinguer les cas $n = 2p$ / $n = 2p + 1$, et dans le cas où $n = 2p + 1$ distinguer les cas $n = 2^r + 1$ / $n \neq 2^r + 1$ pour prouver que cette formule pour $comp_{TF}(n)$ est exacte. L'intuition pour obtenir cette formule est en revanche loin d'être triviale.

- Avantages : Pas de grosse constante devant $(n \log_2 n)$, complexité constante (en moyenne, au pire, etc).
- Inconvénients : Prend beaucoup de place, difficulté de fusionner deux listes "sur place", besoin d'un tableau auxiliaire de taille n, on l'utilise donc peu en pratique.

5.2 Tri par tas : Heapsort

5.2.1 Définitions

- **Arbre binaire parfait** : tous les niveaux de l'arbre sont complets sauf le dernier que l'on remplit de gauche à droite.
- **Arbre partiellement ordonné** : la valeur d'un fils est supérieure ou égale à la valeur du père.
- **Tas** : arbre binaire et partiellement ordonné.

Soit un tas de n éléments numérotés de 1 à n. Considérons un noeud i de l'arbre qui ne soit pas la racine. Son père est $\lfloor \frac{i}{2} \rfloor$. Son fils gauche est $2i$ s'il existe ($2i \leq n$) et son fils droit est $2i+1$ s'il existe ($2i+1 \leq n$). Un noeud i est une feuille si et seulement si il n'a pas de fils gauche, à savoir $2i > n$. Pour tout noeud i autre que la racine, $A[\text{père}(i)] \leq A[i]$.

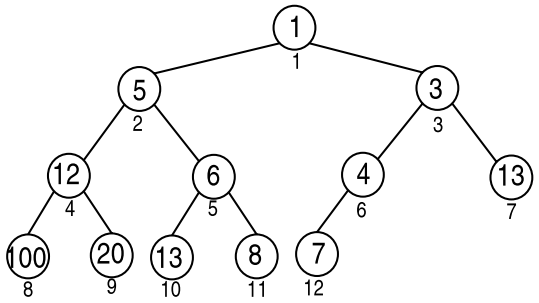


FIG. 5.1 – Exemple de tas

L'avantage du tas est l'utilisation d'un tableau de taille n comme structure de données : on n'a pas besoin de pointeurs fils gauche fils droit.

5.2.2 Tri par tas

Soit $a[1], ..., a[n]$ les n éléments que l'on souhaite trier. On les insère un à un dans le tas (procédure `insertion`). Une fois le tas construit, on extrait l'élément minimum du tas qui se trouve à la racine, et on réorganise le tas (procédure `suppression_min`). On réitère jusqu'à ce que le tas soit vide.

```
proc trier_par_tas
  TAS = tas_vide;
  FOR i = 1 TO n
    insertion(TAS, a[i]);
  FOR i = 1 TO n
    a[i] = suppression_min(TAS);
```

A la fin de la procédure, le tableau a est trié.

5.2.3 Insertion d'un nouvel élément

Il faut garder le caractère partiellement ordonné de l'arbre et, plus dur, son caractère parfait. On place si possible l'élément à droite de la dernière feuille, et si le niveau est complet on le place en dessous, tout à gauche. On procède ensuite par échanges locaux : on compare l'élément avec son père, et on les échange si l'inégalité n'est pas respectée. Le coût de l'insertion est $\log(n)$ puisque l'on fait au plus autant de comparaisons qu'il y a d'étages dans l'arbre.

```
PROC insertion(var TAS::array[1..n], n, x)
n=n+1; TAS[n]=x; fini=FALSE;
pos=n;       // position courante
WHILE NOT(fini) DO
```

```
  IF pos=1 THEN fini=TRUE; // pos est la racine
  ELSE IF TAS[pos/2] <= TAS[pos]
    THEN fini=TRUE;  // élément à la bonne place
    ELSE echange(TAS[pos/2],TAS[pos]);
         pos=pos/2;  // on échange les éléments et on remonte dans l'arbre
```

5.2.4 Suppression d'un élément du tas

On supprime la racine et on réorganise le tas. Pour cela, on fait remonter le dernier élément (le plus bas, tout à droite) en le substituant à la racine, pour conserver la structure de tas. Il s'agit de rétablir l'ordre : on compare l'élément à ses deux fils, et si au moins l'un des fils est inférieur à l'élément, on échange le plus petit des deux fils avec l'élément. On poursuit en descendant dans le tas tant que l'ordre n'est pas respecté ou que l'on n'a pas atteint une feuille.

```
proc suppression_min(var  TAS::array[1..n])
x=TAS[1]; fini=FALSE;
TAS[1]=TAS[n]; n=n-1; pos=1;
WHILE NOT(fini) DO
  IF 2*pos>n  THEN fini = TRUE; // pos est une feuille
  ELSE
    // Identification du plus petit fils i
    IF 2*pos=n or TAS[2*pos] < TAS[2*pos+1]
       THEN i=2*pos ELSE i=2*pos+1
    // Echange si nécessaire, et itération du processus
    IF TAS[pos] > TAS[i]
       THEN echange(TAS[pos],TAS[i]); pos=i;
       ELSE fini = TRUE; // tas ordonné
return x;
```

5.2.5 Complexité du tri par tas

La construction du tas nécessite n insertions, et l'insertion dans un tas de taille i est en $\log i$. On a donc une complexité totale en $\sum_{i=1}^{n} \log i$, qui est en $O(\log(n!))$, et donc en $O(n \log n)$ ($n! \sim \frac{1}{\sqrt{2\pi n}} \left(\frac{n}{e}\right)^n$).

Le coût de `suppression_min` est également logarithmique (nombre d'échanges borné par la hauteur de l'arbre), et donc la complexité des n étapes de suppression est également en $O(n \log n)$.

Il est possible d'améliorer l'algorithme de construction du tas et d'obtenir une complexité en $O(n)$ au lieu de $O(n \log n)$. Cependant, la suppression des n éléments reste en $O(n \log n)$.

5.3 Tri rapide

Tri rapide : On cherche à trier n éléments. On choisit l'un d'entre eux, appelé pivot, et on partage les éléments en les positionnant par rapport au pivot : on place à gauche du pivot ceux qui sont inférieurs ou égaux au pivot, et à droite ceux qui sont supérieurs. On fait deux appels récursifs de la méthode pour continuer, un dans chaque sous-ensemble gauche et droite.

5.3.1 Coût

Le partage se fait en temps linéaire : on compare une fois chaque élément avec le pivot. Si on partage un tableau de taille n en deux sous-ensembles de taille p et q, avec $p + q = n - 1$ (le pivot n'est dans aucun des sous-ensembles), le coût $TR(n)$ de TriRapide est $TR(n) = c.n + TR(p) + TR(q)$.

D'après l'analyse faite pour diviser-pour-régner, on obtiendra un coût $TR(n)$ en $n \log n$ si on sait garantir que p et q ne sont pas trop grands. L'idéal serait d'avoir $p = q$ ou $|p - q| = 1$, donc p et q proches de $\lceil \frac{n}{2} \rceil$.

C'est ce qui se passe en moyenne, et on peut montrer que la version de TriRapide, qui prend comme pivot le premier élément du tableau courant, a un coût moyenné sur les $n!$ permutations en $n \log n$. Ce n'est pas le pire cas de cette version, clairement en n^2.

Pour avoir un pire cas en $n \log n$, on met en oeuvre la version où l'on choisit l'élément médian du tableau courant, autour duquel on va partager les éléments. Comme il existe un algorithme linéaire pour le calcul de la médiane, exposé ci-dessous, le but est atteint.

Il faut noter que TriRapide fonctionne très *rapidement* en pratique, et c'est lui qui sous-tend la routine `qsort()` d'Unix.

5.3.2 Médiane en temps linéaire

Problème

Données : ensemble A de n éléments distincts totalement ordonné.

Résultat : le i-ème élément x de A (suivant leur ordre).

Remarque : on pourrait trier A et sortir le i-ème élément, ce qui se fait en $O(n \log n)$.

Définition : intuitivement, y est médiane de A ssi A possède autant d'éléments supérieurs à y que d'éléments inférieurs à y. On dit que la médiane de A est l'élément de rang $\lfloor \frac{n+1}{2} \rfloor$ pour régler le cas où n est pair. Rechercher la médiane revient donc à trouver le i-ème élément de A, avec $i = \lfloor \frac{n+1}{2} \rfloor$.

Algorithme

- faire des paquets de taille 5 (sauf le dernier éventuellement de taille inférieure) ;
- prendre la médiane de chaque paquet ;
- prendre la médiane Z des médianes ;
- partitionner autour de Z : $\underbrace{\ldots}_{k-1} \leq Z \leq \underbrace{\ldots}_{n-k}$;
  - si $k = i$, renvoyer Z ;
  - si $k > i$, appel récursif : chercher le i-ème élément parmi les $k - 1$ éléments de gauche ;
  - si $k < i$, appel récursif : chercher le $(i - k)$-ème élément parmi les $n - k$ éléments de droite (ce sera donc le i-ème élément de l'ensemble initial, car il y a k éléments plus petits).

Complexité Soit n quelconque et n' le premier multiple de 5 impair supérieur ou égal à n ($n' = 5(2m+1) \geq n$). Alors $n \leq n' \leq n+9$. On rajoute $n' - n$ éléments valant $+\infty$ à l'ensemble, dorénavant de taille n'. Il y a donc $2m + 1$ paquets de 5 éléments, et la liste des médianes de ces paquets est la suivante, avec Z au milieu :

$$Z_1 \leq \ldots \leq Z_m \leq Z \leq Z_{m+2} \leq \ldots \leq Z_{2m+1}$$

On voit qu'il y a $\begin{cases} \text{au moins } 3m+2 \text{ éléments inférieurs à } Z \\ \text{au moins } 3m+2 \text{ éléments supérieurs à } Z \end{cases}$

Il y a en effet 3 éléments inférieurs à $Z_j \leq Z$ dans chaque paquet avec $j = 1..m$, et 2 éléments

inférieurs à Z dans le paquet $m+1$. Le raisonnement est identique pour les éléments supérieurs à Z.

Le coût de l'algorithme ci-dessus $T(n)$ vérifie

$$T(n) = \underbrace{T(\frac{n'}{5})}_{\text{médiane des } \frac{n'}{5} \text{ médianes}} + \underbrace{T(n'-(3m+2))}_{\text{appel récursif}} + an$$

Le terme linéaire an correspond au calcul de la médiane de chaque paquet, qui se fait en temps constant (et il y a un nombre linéaire de paquets), et à la partition autour de Z des n éléments.

Remarques : $\begin{cases} n' \leq n+9 \\ n'-3m-2 \leq n' - \frac{3}{2}(\frac{n'}{5}-1) - 2 \leq \frac{7}{10}(n+9) - \frac{1}{2} \leq \frac{7}{10}n + \frac{58}{10} < \frac{7}{10}n + 6 \end{cases}$

Théorème 8. *Cet algorithme est de complexité linéaire,* i.e. $\exists c,\ T(n) \leq cn$

Preuve. Par récurrence, supposons le résultat vrai $\forall k < n$. Alors

$$T(n) \leq c\frac{n+9}{5} + c(\frac{7}{10}n + 6) + an = (\frac{9}{10}c + a)n + \frac{39}{5}c$$

On veut $(\frac{9}{10}c + a)n + \frac{39}{5}c \leq cn$.

Or $\frac{39c}{5} \leq \frac{cn}{20}$ ssi $20 \leq \frac{5n}{39}$ ssi $n \geq 156$.

Donc pour $n \geq 156$, $(\frac{9c}{10} + a)n + \frac{39c}{5} \leq (\frac{19c}{20} + a)n$.

Ainsi il suffit de prendre $\begin{cases} c \geq 20a \\ c \geq \max_{1 \leq k \leq 156} \frac{T(k)}{k} \end{cases}$

La récurrence est ainsi correcte également pour les cas initiaux. □

Remarque : Si on découpe en paquets de 3 au lieu de 5, on raisonne pour $n = 3(2m+1)$. Alors, dans la récurrence, on obtient $T(n) = T(n/3) + T(n-2m-1) + an$, avec $2m \sim n/3$. Asymptotiquement, $T(n) = T(n/3) + T(2n/3) + an$, et cette récurrence est connue pour être en $n \log n$, il faut développer l'arbre de récurrences.

Remarque : En revanche, l'algorithme marche en découpant en paquets de 7, ou plus gros, mais de toute façon on ne pourra pas obtenir une complexité meilleure que cn.

5.4 Complexité du tri

5.4.1 Les grands théorèmes

Attention. On travaille dans un univers où on ne fait que des comparaisons. On a un ordre total pour les éléments qu'on veut trier, mais pas d'autre opération, ni d'information sur l'univers des éléments. Sans ces hypothèses, il existe des tris différents, sans comparaison du tout à proprement parler. Par exemple RadixSort utilise la décomposition binaire des entiers qu'il trie (penser à un jeu de cartes qu'on trie par couleur avant de trier par valeur). Par exemple encore, si on a l'information que tous les éléments sont des entiers entre 1 et K, on prépare un tableau de taille K et on stocke en case i le nombre d'éléments égaux à i ; remplir ce tableau n'exige qu'un seul parcours des n éléments, qu'on va pouvoir trier en temps linéaire.

Théorème 9. *Il faut au moins* $\lceil \log_2(n!) \rceil = O(n \log(n))$ *comparaisons au pire pour trier* n *éléments.*

Théorème 10. *Il faut au moins* $\log_2(n!) = O(n \log(n))$ *comparaisons en moyenne pour trier* n *éléments.*

Rappel sur la complexité en moyenne : $n!$ données différentes possibles (toutes les permutations des n éléments). Pour chaque donnée on compte le nombre de comparaisons nécessaires. On fait la moyenne de tous les cas possibles (la complexité moyenne est rarement calculable).

Bien sûr le second théorème implique le premier (la partie entière supérieure peut être rajoutée car dans le cas le pire, le nombre de comparaisons est un entier), mais le premier est montré facilement de façon indépendante.

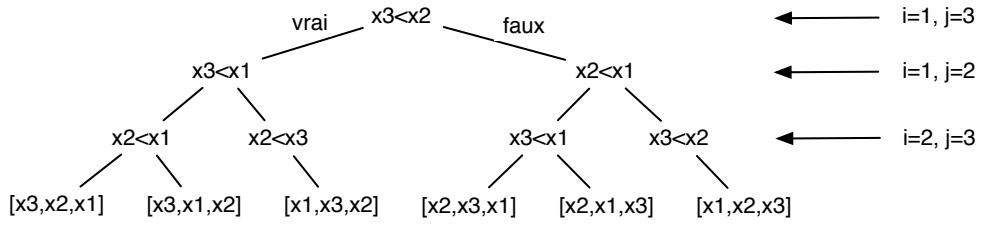
On a vu que la complexité de tri fusion ne dépend pas des données et vaut $comp_{TF}(n) = n\lceil \log_2(n) \rceil - 2^{\lceil \log_2(n) \rceil} + 1$, ce qui tend vers $n \log_2(n)$, montrant ainsi que la borne peut être atteinte asymptotiquement.

5.4.2 Démonstration des théorèmes

Voici le déroulement d'un tri en deux boucles (tri à bulle), dont le code pour trier un tableau $t[1..n]$ est le suivant :

```
for i = 1 to n do
   for j = n downto i+1 do
        if t[j] < t[j-1] then t[j] <-> t[j-1]
```

On a représenté l'arbre de décision : c'est la trace de l'exécution pour toutes les données possibles. C'est un arbre binaire (chaque comparaison renvoie 'vrai' ou 'faux'). Il n'y a pas toujours deux fils (l'arbre n'est pas localement complet), mais il y a exactement $n!$ feuilles.



Lemme 5. *Il y a exactement* $n!$ *feuilles dans tout arbre de décision d'un algorithme qui trie* n *éléments.*

Preuve. On fait des comparaisons entre les éléments. Si l'arbre avait moins de $n!$ feuilles, une même feuille correspondrait à deux permutations différentes, ce qui n'est pas possible. Pour une permutation donnée, il y a toujours le même chemin d'exécution, donc la même feuille, et le nombre de feuilles est au plus $n!$. □

Il faut distinguer 'Hauteur maximale' et 'Hauteur moyenne'. La complexité de l'algorithme pour une donnée correspond à la hauteur de la feuille associée.

Pour démontrer les théorèmes, on a maintenant un problème d'hauteur d'arbre binaire. La *hauteur* d'une feuille est sa distance à la racine (définition), c'est le nombre de comparaisons effectuées pour la donnée correspondante.

Pour tout arbre à f feuilles, la hauteur maximale est au moins celle de l'arbre qui est le moins haut avec toutes ces feuilles : c'est l'arbre complet. On a donc $h_{max} \geq \lceil log_2(f) \rceil$. On

applique le résultat pour $f = n!$ ce qui prouve le premier théorème, en utilisant la formule de Stirling pour estimer $\log_2(n!) = O(n \log(n))$.

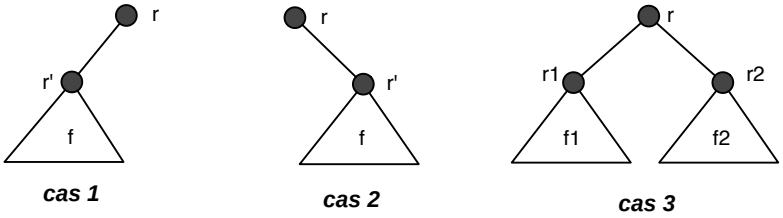
Pour le deuxième théorème, on montre que la hauteur moyenne est supérieure à $\log_2(n!)$ grâce au lemme suivant, et on termine également avec la formule de Stirling.

Lemme 6. *Pour tout arbre à f feuilles, la hauteur moyenne h est supérieure à $log_2(f)$.*

Preuve. On procède par récurrence sur le nombre de feuilles f. On note par h la hauteur moyenne.

• Si $f = 1$, ça marche, car $h \geq 0$ (évident).

• Cas général : supposons la propriété vraie pour tout $f' < f$, et considérons un arbre à f feuilles. Il y a trois cas :



Dans les cas 1 et 2 : profondeur des feuilles (distance à r) = 1 + distance à la sous-racine r'. On va montrer en itérant la construction que dans le sous-arbre enraciné en r', $h_{r'} \geq log_2(f)$, où $h_{r'}$ est la hauteur moyenne dans le sous-arbre. Alors on aura bien $h \geq h_{r'} \geq log_2(f)$. On itère donc la construction jusqu'à tomber sur le cas 3, celui d'une racine qui a deux sous-arbres, qui est plus compliqué.

Pour le cas 3, on considère $f = f_1 + f_2$, avec $f_1 \neq 0$ et $f_2 \neq 0$. On peut alors utiliser l'hypothèse de récurrence : si h_1 et h_2 sont les hauteurs moyennes des deux sous-arbres, alors $h_i \geq \log_2(f_i)$.

La hauteur moyenne d'une feuille a est :
- si a est à gauche, $h(a) = 1 + h_1(a)$;
- si a est à droite, $h(a) = 1 + h_2(a)$.

$h_1(a)$ et $h_2(a)$ représentent respectivement la distance de la feuille a à la racine du sous-arbre de gauche r_1 et de droite r_2, et $h(a)$ est la distance de a à r.
On va alors pouvoir calculer la hauteur moyenne de l'arbre en sommant sur toutes les feuilles. On note par $F = F_1 \cup F2$ l'ensemble des feuilles.

$$h = \frac{\sum_{a \in F} h(a)}{f} = \frac{\sum_{a \in F_1} h(a) + \sum_{a \in F_2} h(a)}{f}$$

$$h = \frac{f_1 + \sum_{a \in F_1} h_1(a)}{f} + \frac{f_2 + \sum_{a \in F_2} h_2(a)}{f} = \frac{f_1 + h_1 * f1}{f} + \frac{f_2 + h_2 * f_2}{f}$$

$$h \geq 1 + \frac{log_2(f_1) * f_1}{f} + \frac{log_2(f_2) * f_2}{f}$$

Il y a 2 manières de démontrer la dernière inégalité :
- On écrit $f_2 = f - f_1$. Soit $g(f1) = 1 + \frac{log_2(f_1) * f_1}{f} + \frac{log_2(f - f_1) * (f - f_1)}{f}$. On obtient le minimum de la fonction g pour $f_1 = f/2$ (c'est en cette valeur que la dérivée s'annule).
- On peut également observer que cette fonction est convexe, et le minimum est donc obtenu pour $f_1 = f_2 = f/2$.

On a donc $h \geq 1 + \log_2(f/2) = 1 + \log_2(f) - 1 = \log_2(f)$, ce qui clôt la démonstration.

□

Autre preuve : preuve de Shannon, basée sur la théorie de l'information. Idée : il y a $n!$ données, donc $\lceil \log_2(n!) \rceil$ bits d'informations à acquérir. Comme une comparaison permet d'acquérir 1 bit, il en faut au moins $\lceil log_2(n!) \rceil$ pour trier les données.

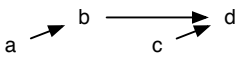
5.4.3 Peut-on atteindre la borne ?

On sait qu'il faut au moins $\lceil \log_2(n!) \rceil$ comparaisons dans le pire des cas, mais cette borne est-elle atteignable ? Asymptotiquement oui, car nous avons trois algorithmes dont la complexité dans le pire des cas était en $n \log(n)$. Regardons si c'est le cas si on regarde à la comparaison près. Dans le tableau ; on note le nombre de comparaisons effectué par TriFusion, la valeur de la borne, et le nombre de comparaisons d'une solution optimale décrite plus bas :

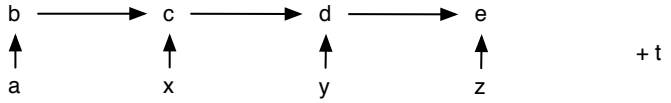
n	2	3	4	5	6	7	8	9	10	11	12
TriFusion(n)	1	3	5	8	11	14	17	21	25	29	33
$\lceil \log_2(n!) \rceil$	1	3	5	7	10	13	16	19	22	26	29
Opt(n)	1	3	5	7	10	13	16	19	22	26	30

Une idée importante est l'insertion par dichotomie. Insérer un élément dans une suite triée de k éléments coûte r comparaisons au pire si $k \leq 2^r - 1$. Il est donc plus *avantageux* d'insérer dans une chaîne de taille 3 que de taille 2, et dans une chaîne de taille 7 que de taille 4 à 6, car le coût au pire est le même.

- trier 3 nombres :
  on en prend 2 au hasard et on compare le 3ème aux deux autres, coût 3.
- trier 4 nombres :
  - tri incrémental : on trie les 3 premiers ($a \leq b \leq c$) (3 comparaisons), puis on insère le 4ème par dichotomie (2 comparaisons) : coût 3+2=5.
  - diviser pour régner (le diagramme ($a \to b$) signifie $a \leq b$) :

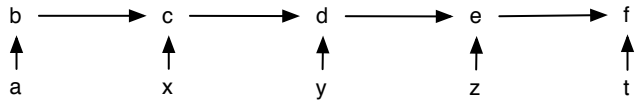


    on forme 2 paires de 2, ($a \to b$) et ($c \to d$), on compare les deux plus grands pour obtenir par exemple ($a \to b \to d$), puis on insère c par dichotomie (2 comparaisons) : coût 2+1+2=5.
- trier 5 nombres :
  faire deux paires sur a, b, c, d et comparer les plus grands (coût = 3) pour obtenir le même dessin que plus haut, puis on insère e dans la chaîne (a, b, d) (coût = 2), et on insère c dans (e, a, b) (si $e < a$), (a, e, b) (si e entre a et b), (a, b, e) (si e entre b et c), ou (a, b) (si $e > d$), avec un coût 2 dans tous les cas. Il y a donc 3+2+2=7 comparaisons à faire.
- trier 6 nombres :
  $6 = 5 + 1$: on insère le 6ème dans les 5 premiers (coût = $7 + 3 = 10$)
- trier 7 nombres :
  $7 = 6 + 1$: on insère le 7ème dans les 6 premiers (coût = $10 + 3 = 13$)
- trier 8 nombres :
  $8 = 7 + 1$: on insère le 8ème dans les 7 premiers (coût = $13 + 3 = 16$)
- trier 9 nombres $(a, b, c, d, e, x, y, z, t)$:



On forme 4 paires : (a, b), (x, c), (y, d), (z, e), t est isolé. Coût : 4 pour former les paires. On trie (b, c, d, e), coût $5 = \text{Opt}(4)$. Puis on insère dans l'ordre y, x, t, z. Coûts respectifs : 2,2,3,3. Coût total : 19.

– trier 10 nombres : $(a, b, c, d, e, f, x, y, z, t)$:



On forme 5 paires (a, b), (x, c), (y, d), (z, e), (t, f) (coût 5), on trie (b, c, d, e, f) (coût $7 = \text{Opt}(5)$) puis on insère y dans (a, b, c) (coût 2), x dans $(a, b, y?)$ (coût 2) puis t dans une chaîne de longueur 7 (coût 3) et z dans une chaîne de longueur 6 ou 7 (coût 3). Coût total 22.

– trier 11 nombres :
$11 = 10 + 1$, on utilise la méthode incrémentale.
– trier 12 nombres :
Il faut au minimum 30 comparaisons (méthode $12 = 11+1$), c'est impossible de le faire en 29 comparaisons, on a testé tous les algorithmes possibles par ordinateur ('brute force'), ce qui était un véritable challenge lorsque ces recherches ont été effectuées, pour être sûr de tout tester (en 1990, 2h de calcul).

La borne du nombre de comparaisons en moyenne ne peut pas être atteinte exactement dès $n = 7$. Il s'agit de compter la longueur des cheminements pour un arbre binaire à n feuilles, que l'on peut minorer de façon exacte.

5.5 Exercices

Exercice 5.5.1. *Hors d'œuvre*

On considère un ensemble S de $n \geq 2$ entiers distincts stockés dans un tableau (S n'est pas supposé trié). Résoudre les questions suivantes :

1 - Proposer un algorithme en $\mathcal{O}(n)$ pour trouver deux éléments x et y de S tels que $|x - y| \geq |u - v|$ pour tout $u, v \in S$.

2 - Proposer un algorithme en $\mathcal{O}(n \log n)$ pour trouver deux éléments x et y de S tels que $x \neq y$ et $|x - y| \leq |u - v|$ pour tout $u, v \in S$, $u \neq v$.

3 - Soit m un entier arbitraire (pas nécessairement dans S), proposer un algorithme en $\mathcal{O}(n \log n)$ pour déterminer s'il existe deux éléments x et y de S tels que $x + y = m$.

4 - Proposer un algorithme en $\mathcal{O}(n)$ pour trouver deux éléments x et y de S tels que $|x - y| \leq \frac{1}{n-1}(\max(S) - \min(S))$.

Correction.

1 - Pour que $|a - b|$ soit maximum il faut que a (Resp b) soit maximum et que b (Resp a) soit minimum. Le problème se résume donc à une recherche du min et du max qui se fait en $O(n)$.

2 - Si $|c-d|$, est minimum pour c et d appartenant à S alors il n'existe pas e appartenant à S tel que $a > e > b$ ou $a < e < b$: sinon $|c-e| < |c-d|$. On trie donc S en $O(n \times logn)$ et on cherche ensuite le min des $s_i - s_{i+1}$, (s_i et s_{i+1} étant deux entier consécutifs de S) en $O(n)$.On obtient donc un algorithme en $O(n \times logn)$.

3 - On commence par trier S en $O(n \times logn)$.Ensuite pour tout y appartenant à S on cherche si $m - y$ appartient à S par dichotomie (S est trié), la dichotomie se faisant en $O(logn)$ cette opération ce fait en $O(n \times logn)$. A la fin on obtient bien un algorithme en $O(n \times logn)$.

4 - Il existe deux méthodes pour résoudre cette question.

4.1 - Première méthode

Cette méthode est basée sur la médiane. On note S_1 l'ensemble des élément de S inféreurs à la médiane et S_2 les éléments de S supérieurs à la médiane. Il est à noter que $\|S_1\| = \|S_2\| = \lceil \frac{\|S\|}{2} \rceil$.

début
  Calculer le min et le max de S ;
  si $n = 2$ **alors** retourner (Min,Max) ;
  sinon
    extraire la médiane de S ; calculer S_1 S_2 ;
    relancer l'algo sur S_1 ou S_2 suivant le cas.
fin

Preuve de l'algorithme : on pose $M(S)$ la moyenne pondérée de S, $n =\| S \|$, a le min de S, b le max de S, et m la médianne de S, deux cas sont alors possibles :

– n impair :

$$n = 2k+1, \text{alors} \|S_1\| = \frac{n+1}{2} = k+1$$
$$\text{donc :} M(S_1) = \frac{m-a}{k+1-1} = \frac{m-a}{k}$$
$$\text{et :} \|S_2\| = \frac{n+1}{2} = k+1$$
$$\text{donc :} M(S_2) = \frac{b-m}{k+1-1} = \frac{b-m}{k}$$
$$\text{donc :} \frac{M(S_1)+M(S_2)}{2} = (\frac{m-a}{k} + \frac{b-m}{k}) \times \frac{1}{2} = \frac{b-a}{2k}$$
$$\text{or :} \frac{b-a}{2k} = \frac{b-a}{n-1} = M(S)$$
$$\text{donc :} 2 \times M(S) = M(S_1) + M(S_2)$$
$$\text{finalement :} M(S_1) \leq M(S) \text{ou} M(S_2) \leq M(S).$$

– n pair :

$$n = 2k, \text{alors} \|S_1\| = \frac{n+1}{2} = k$$
$$\text{donc :} M(S_1) = \frac{m-a}{k-1}$$
$$\text{et :} \|S_2\| = \frac{n+1}{2}$$
$$\text{donc :} M(S_2) = \frac{b-m}{k-1}$$

$$\text{donc} : \frac{M(S_1) + M(S_2)}{2} = (\frac{m-a}{k-1} + \frac{b-m}{k-1}) \times \frac{1}{2} = \frac{b-a}{2k-2}$$

$$\text{or} : \frac{b-a}{2k-2} \leq \frac{b-a}{n-1} = M(S)$$

$$\text{donc} : 2 \times M(S) \geq M(S_1) + M(S_2)$$

$$\text{finalement} : M(S_1) \leq M(S) \text{ou} M(S_2) \leq M(S).$$

Comme on sait que $M(S_1) \leq M(S)$ ou $M(S_2) \leq M(S)$ cela nous permet d'enclencher l'induction : soit sur S_1, soit sur S_2. Rechercher la moyenne se fait en $O(n)$, chercher le min et le max d'un ensemble aussi, donc :

$$C(n) = C(\lceil \frac{n}{2} \rceil) + O(n)$$

Ce qui donne par master théorème

$$C(n) = O(n)$$

4.2 - Deuxième méthode

On divise S en $(n-1)$ boites.

début

- **si** *il n'y a qu'une boite* **alors** on renvoie deux de ses éléments;
- **sinon**
  - **si** *le nombre d'éléments des* $\lfloor \frac{n-1}{2} \rfloor$ *premières boites est supérieur à* $\lfloor \frac{n-1}{2} \rfloor$. **alors** la moyenne pondérée sur les $\lfloor \frac{n-1}{2} \rfloor$ premières boites est inférieure à celle de S, et on relance l'algorithme sur ces boites. ;
  - **sinon**
    - le nombre d'éléments des $\lceil \frac{n+1}{2} \rceil$ dernières boites est supérieur à $\lceil \frac{n+1}{2} \rceil$, et alors la moyenne pondérée sur les $\lceil \frac{n+1}{2} \rceil$ dernières boites est inférieure à celle de S, et on relance l'algorithme sur celles-ci.

fin

Quant à la complexité elle est en $O(n)$:

$$C(n) = C(\lceil \frac{n}{2} \rceil + 1) + O(n)$$

Ce qui donne par master théorème

$$C(n) = O(n)$$

Dans les exercices suivants, on s'intéresse à des ensembles de n entiers tous distincts rangés dans un tableau $T[1], \ldots, T[n]$. Les algorithmes considérés ici effectuent des **affectations** et leur seul critère de décision (ou de bifurcation) est la **comparaison** de deux éléments ($=$ et $<$). En aucun cas ils ne peuvent effectuer des opérations arithmétiques, comme l'addition ou la multiplication.

Exercice 5.5.2. *Maximum de n entiers*

1 - Ecrire un algorithme (naïf !) qui calcule le maximum de n entiers. Quelle en est la complexité (en nombre de comparaisons effectués, en nombre d'affectations effectuées, dans le pire des cas, le meilleur, en moyenne) ?

Indications pour le calcul des affectations en moyenne : soit $P_{n,k}$ le nombre de permutations σ de $\{1,\ldots,n\}$ telles que sur la donnée $T[1] = \sigma(1)$,...,$T[n] = \sigma(n)$ l'algorithme effectue k affectations. Donner une relation de récurrence pour $P_{n,k}$. Soit $G_n(z) = \sum P_{n,k} z^k$. Montrer que $G_n(z) = z(z+1)\cdots(z+n-1)$. En déduire le résultat.

2 - L'algorithme que vous avez proposé est-il optimal pour la complexité en comparaisons dans le pire des cas ?

Correction.

1- On donne ici **un algorithme naïf** pour le calcul du maximum de n entiers :

début
  max $\leftarrow T[1]$
  pour i **de** *1* **à** n **faire**
    si $T[i] > max$ **alors** max $\leftarrow T[i]$
  fin
  Retourner max
fin

Complexité en nombre de comparaisons : quel que soit le cas, on effectue $n-1$ comparaisons dans l'algorithme, chaque élément après le premier étant comparé une fois.

- dans le pire des cas : n (si le tableau est ordonné de manière croissante),
- dans le meilleur des cas : 1 (si le maximum est en première position),
- en moyenne : pour le calcul de la complexité en moyenne, il va falloir faire appel à un calcul plus poussé :
  - **Rappel :** *complexité en moyenne d'un algorithme A de données de taille n :*

$$moy_A(n) = \sum_{\text{n données de taille m}} p(d)\text{coût}_A(d)$$

    où $p(d)$ est la probabilité que d soit une entrée de l'algorithme A.
  - les données sont les permutations de [1,...,n] stockées dans T. Les permutations étant totalement aléatoires et n'étant régies par aucune loi mathématique, on peut les supposer équiprobables.
  - le coût d'un algorithme A(T) est son nombre d'affectation. Ainsi, pour chaque cas de $P_{n,k}$, l'algorthime effectue k affectations. On obtient donc ainsi que le coût d'un algorithme A(T) est de $kP_{n,k}$.
  - d'où :

$$moy_A(n) = \frac{1}{n!} \sum_{T \text{ permutations de } [1,\ldots,n]} cout_A(T) = \frac{1}{n!} \sum_{k=1}^{n} kP_{n,k}$$

  - trouvons $P_{n,k}$:
    - supposons le max en $T[n]$: pour avoir k affectations en tout, il faut : $P_{n-1,k-1}$
    - sinon, les affectations sont réalisées avant et on a alors $n-1$ permutations possibles : $P_{n-1,k}$
    - Donc $P_{n,k} = P_{n-1,k-1} + (n-1)P_{n-1,k-1}$
  - cas limites :
    - $P_{n,0} = 0$
    - si $k > n$: $P_{n,k} = 0$
    - $\forall n \in \mathbb{N}$: $P_{n,n} = 1$

– Utilisons les séries génératrices : $G_n(z) = \sum_{k=1}^{n} P_{n,k} z^k$

$$\begin{aligned} G_{n+1}(z) &= \sum_{k=1}^{n+1} P_{n,k} z^k \\ &= \sum_{k=1}^{n+1} (P_{n,k-1} + nP_{n,k}) z^k \\ &= \sum_{k=1}^{n+1} P_{n,k-1} z^{k-1} + n \sum_{k=1}^{n+1} P_{n,k} z^k \\ &= \sum_{k=0}^{n} P_{n,k} z^k + n \sum_{k=1}^{n+1} P_{n,k} z^k \end{aligned}$$

Or $P_{n,0} = 0$ et $P_{n,n+1} = 0$ d'où :

$$G_{n+1}(z) = z \sum_{k=0}^{n} P\_n, k z^k + n \sum_{k=1}^{n} P_{n,k} z^k$$

d'où :

$$G_{n+1}(z) = (z+n) G_n(z)$$

Or $G_1(z) = P_{1,1} z = z$ d'où $G_n(z) = z(z+1)\dots(z+n-1)$

– de plus :

$$\begin{aligned} G'_n(z) &= \sum_{k=1}^{n} k P_{n,k} z^{k-1} \\ \text{d'où : } G'_n(1) &= \sum_{k=1}^{n} k P_{n,k} \\ \text{de même : } G_n(1) &= \sum_{k=1}^{n} P_{n,k} = n! \end{aligned}$$

– On obtient donc :

$$\begin{aligned} moy_A(n) &= \frac{G'_n(1)}{G_n(1)} \\ &= [lnG(z)]'(1) \\ &= [\sum_{i=0}^{n-1} ln(z+i)]'(1) \\ &= 1 + \frac{1}{2} + \dots + \frac{1}{n} \\ &= H_n \text{ (la suite harmonique à l'ordre } n) \end{aligned}$$

$$moy_A(n) = O(ln\, n)$$

2- **Optimalité en terme de comparaisons**

Une comparaison est un match entre deux entiers, le gagnant étant le plus grand. Si le max est connu, c'est que tout les autres entiers ont perdu au moins une fois un match contre lui, ce qui demande en tout $n-1$ comparaisons. Notre algorithme en faisant toujours $n-1$, il est optimal pour la complexité en nombre de comparaisons.

Exercice 5.5.3. *Plus petit et plus grand*

Dans l'exercice suivant, on ne s'intéresse plus qu'à la **complexité dans le pire des cas et en nombre de comparaisons** des algorithmes.

1- On s'intéresse maintenant au calcul (simultané) du maximum et du minimum de n entiers. Donner un algorithme naïf et sa complexité.

2 - Une idée pour améliorer l'algorithme est de regrouper *par paires* les éléments à comparer, de manière à diminuer ensuite le nombre de comparaisons à effectuer. Décrire un algorithme fonctionnant selon ce principe et analyser sa complexité.

3 - Montrons l'optimalité d'un tel algorithme en fournissant une borne inférieure sur le nombre de comparaisons à effectuer. Nous utiliserons la méthode de l'*adversaire*.

Soit A un algorithme qui trouve le maximum et le minimum. Pour une donnée fixée, au cours du déroulement de l'algorithme, on appelle *novice* (N) un élément qui n'a jamais subi de comparaisons, *gagnant* (G) un élément qui a été comparé au moins une fois et a toujours été supérieur aux éléments auxquels il a été comparé, *perdant* (P) un élément qui a été comparé au moins une fois et a toujours été inférieur aux éléments auxquels il a été comparé, et *moyens* (M) les autres. Le nombre de ces éléments est représenté par un quadruplet d'entiers (i,j,k,l) qui vérifient bien sûr $i+j+k+l=n$.

Donner la valeur de ce quadruplet au début et à la fin de l'algorithme. Exhiber une stratégie pour l'adversaire, de sorte à maximiser la durée de l'exécution de l'algorithme. En déduire une borne inférieure sur le nombre de tests à effectuer.

Correction.

1 - On donne ici **un algorithme naïf** pour le calcul du maximum et du minimum de n entiers :

début
  max $\leftarrow T[1]$
  $i_{max} \leftarrow 1$
  min $\leftarrow T[1]$
  pour i **de** *2* **à** n **faire**
    si $T[i] > max$ **alors** max $\leftarrow T[i]$; $i_{max} \leftarrow i$
  fin
  pour i **de** *2* **à** n **faire**
    si $i \neq i_{max}$ **et** $min < T[i]$ **alors** min $\leftarrow T[i]$
  fin
  Retourner (max,min)
fin

Complexité en nombre de comparaisons : On effectue $n-1$ comparaisons pour trouver le maximum, et $n-2$ pour trouver le minimum. On a donc une complexité en $2n-3$.

2 - **Groupement des éléments par paires**

On regroupe maintenant les éléments par paire pour ensuite effectuer les opérations de comparaisons.

L'algorithme suivant permet de résoudre le problème posé :

début
  pour i **de** *1* **à** $\lfloor \frac{n}{2} \rfloor$ **faire**
    si $T[2i-1] > T[2i]$ **alors** échange $T[2i-1]$ et $T[2i]$
  fin
  max $\leftarrow T[2]$
  pour i **de** *2* **à** $\lfloor \frac{n}{2} \rfloor$ **faire**
    si $T[2i] > max$ **alors** max $\leftarrow T[2i]$
  fin
  min $\leftarrow T[1]$
  pour i **de** *2* **à** $\lfloor \frac{n}{2} \rfloor$ **faire**
    si $T[2i-1] < min$ **alors** min $\leftarrow T[2i-1]$
  fin
  si n *impair* **alors** **si** $T[n] > max$ **alors** max $\leftarrow T[n]$
  sinon si $T[n] < min$ **alors** min $\leftarrow T[n]$
fin

Complexité associée :

- *si n est pair*, on va avoir $\frac{n}{2}$ comparaisons pour faire les paires, $\frac{n}{2} - 1$ comparaisons pour trouver le maximum et autant pour le mimimum. Dans ce cas là, la complexité vaut donc :

$$\frac{3n}{2} - 2$$

- *si n est impair*, le nombre de comparaisons est inférieur ou égal à $3\lfloor \frac{n}{2} \rfloor$, donc inférieur ou égal à $\lceil \frac{3n}{2} \rceil - 2$.

3- **Méthode de l'***adversaire*

Cette méthode va être utilisée pour fournir une borne inférieure sur le nombre de comparaisons à effectuer et ainsi montrer l'optimalité de l'algorithme précédent.

Définitions :
- Novice : élément qui n'a jamais été comparé
- Gagnant : élément qui a gagné tout ses matchs
- Perdant : élément qui a perdu tout ses matchs
- Moyen : élément qui a gagné et perdu

Soit A notre algorithme. On note (i, j, k, l) respectivement le nombre de novices, gagnants, perdants et moyens (avec $i + j + k + l = n$).

Au début de l'algorithme, on a $(i, j, k, l) = (n, 0, 0, 0)$. A la fin, $(i, j, k, l) = (0, 1, 1, n-1)$.

Le but de l'adversaire est de maximiser la durée de l'algorithme tout en conservant une stratégie cohérente. On peut alors donner l'ensemble des opérations résumées dans le tableau 5.5.3

/ représente les cas où l'entier reste inchangé.

Cherchons à donner une borne inférieure sur le nombre de tests à effectuer.
Il faut que chaque novice soit comparé au moins une fois. La comparaison la plus efficace est alors $N : N$ car elle diminue le nombre de novice de deux, et il en faut au moins $\lceil \frac{n}{2} \rceil$ pour atteindre le bon nombre de novices.

De plus, il faut à la fin que l'on ait $n - 2$ moyens, ce qui nécessite au moins $n - 2$ autres comparaisons.

D'où au minimum

$$n - 2 + \left\lceil \frac{n}{2} \right\rceil = \left\lceil \frac{3n}{2} \right\rceil - 2 \text{ comparaisons.}$$

comparaison	choix	i	j	k	l
$N:N$		$i-2$	$j+1$	$k+1$	/
$N:G$	$G>N$	$n-1$	/	$k+1$	/
$N:P$	$P<N$	$n-1$	$j+1$	/	/
$N:M$	$M>N$	$i-1$	/	$k+1$	/
	$N>M$ [1]	$i-1$	$j+1$	/	/
$G:G$		/	$j+1$	/	$l+1$
$G:P$	$G>P$	/	/	/	/
$G:M$	$G>M$	/	/	/	/
$P:P$		/	/	$k-1$	$l+1$
$P:M$	$P<M$	/	/	/	/
$M:M$		/	/	/	/

La borne inférieure sur le nombre de tests à effectuer est égale à la complexité de l'algorithme précédent. On montre ainsi que ce dernier est optimal.

Exercice 5.5.4. *Plus grand et deuxième plus grand de n entiers*

On s'intéresse dans cet exercice à la **complexité dans le pire des cas et en nombre de comparaisons** des algorithmes.

1 - Pour rechercher le plus grand et deuxième plus grand élément de n entiers, donner un algorithme naïf et sa complexité.

2 - Pour améliorer les performances, on se propose d'envisager la solution consistant à calculer le maximum suivant le principe d'un *tournoi* (tournoi de tennis par exemple). Plaçons-nous d'abord dans le cas où il y a $n = 2^k$ nombres qui s'affrontent dans le tournoi. Comment retrouve-t-on, une fois le tournoi terminé, le deuxième plus grand ? Quelle est la complexité de l'algorithme ? Dans le cas général, comment adapter la méthode pour traiter n quelconque ?

3 - Montrons l'optimalité de cet algorithme en fournissant une borne inférieure sur le nombre de comparaisons à effectuer. Nous utiliserons la méthode des *arbres de décision*.

3.1 - Montrer que tout arbre de décision qui calcule le maximum de N entiers a au moins 2^{N-1} feuilles.

3.2 - Montrer que tout arbre binaire de hauteur h et avec f feuilles vérifie $2^h \geq f$.

3.3 - Soit A un arbre de décision résolvant le problème du plus grand et deuxième plus grand de n entiers, minorer son nombre de feuilles. En déduire une borne inférieure sur le nombre de comparaisons à effectuer.

Correction.

1 - Algorithme naïf : recherche du premier maximum, puis du second.

Nombre de comparaisons :

Premier maximum, sur n valeurs :	$n-1$
Second maximum, sur n - 1 valeurs :	$n-2$
	$2n-3$

2 - Cas où $n = 2^k$:

On calcule le premier maximum à la façon d'un tournoi de tennis. On cherche ensuite le second maximum, en prenant le maximum parmi les adversaires que le premier à rencontré. Ainsi, dans

```
début
    max_1 ← T[1];
    pour i allant de 2 à n faire
        si T[i] > max_1 alors
            max_1 ← T[i];
            posmax_1 ← i;
    si posmax_1 ≠ 1 alors max_2 ← T[1] sinon max_2 ← T[2];
    pour i allant de 2 à n avec i ≠ posmax_1 faire
        si T[i] > max_2 alors
            max_2 ← T[i];
    retourner max_1, max_2;
fin
```

l'arbre donné à la figure 5.2, le second maximum est nécessairement un élément contenu dans le chemin rouge.

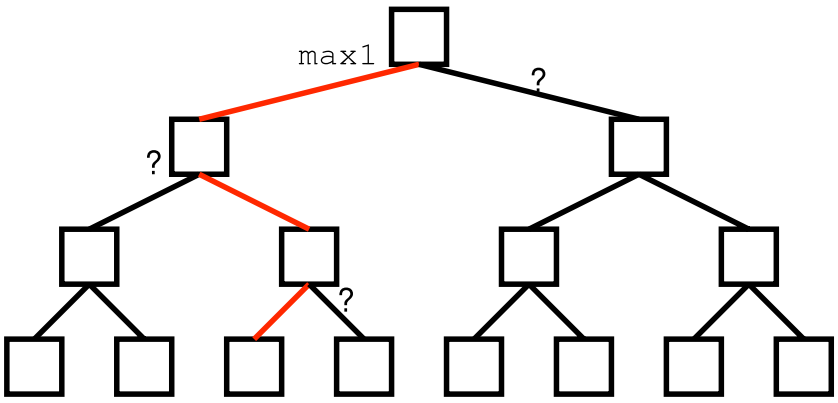


FIG. 5.2 – Arbre des "compétitions" effectuées

Nombre de comparaisons :

Premier maximum :	$n-1 = 2^k - 1$
Second maximum :	$k-1$
	$2^k + k - 2 = n + log_2 n - 2$

Cas général : n quelconque
S'il n'est pas rempli et a une profondeur minimale, la branche de longueur maximale est de longueur $\lceil log_2 n \rceil$.
Au pire, nombre de comparaisons :

Premier maximum :	$n-1$
Second maximum :	$\lceil log_2 n \rceil - 1$
	$n + \lceil log_2 n \rceil - 2$

3.1 - Pour chercher le maximum parmis N valeurs, on doit effectuer $N-1$ comparaisons.
On a donc 2^{N-1} feuilles dans l'arbre de décision.

3.2 - Par récurrence sur la hauteur h de l'arbre
- Pour $h = 0$: 1 feuille au plus
- On considère un arbre de hauteur $h+1$, on a alors deux cas, soit la racine a un seul fils soit il en a deux.
  - un fils : on alors le nombre de feuille qui correspond à celui de l'arbre partant du fils,

qui est de hauteur h , et $2^(h+1) \geq 2^h$ ce qui va bien.
– deux fils : on a alors le nombre de feuilles qui correspond à la somme de celles des deux sous arbresz partants des deux fils : $f = f_1 + f_2$, en ayant pour chacun une hauteur maximale de h soit : $2^(h+1) \geq 2^h + 2^h \geq 2^{h_1} + 2^{h_2} \geq f_1 + f_2 \geq f$ cqfd.

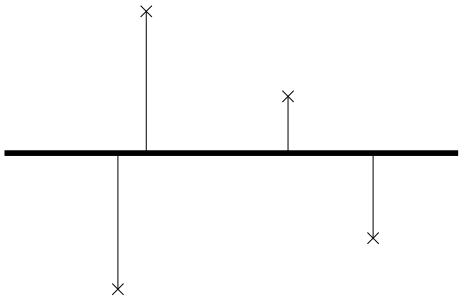
3.3 - On partitionne les feuilles de A selon la valeur du premier maximum pour former les A_i ; A_i est au final l'arbre A dont on a enlevé tous ce qui n'aboutissait pas à une feuille concluant que le premier maximum était i. Ces A_i sont des arbres donnant un maximum parmi n-1 éléments donc ils ont chacun un nombre de feuilles tel que : nombre de feuilles de $A_i \geq 2^{n-2}$ Donc en considérant A comme la 'fusion' de ces arbres qui en forme une partition, on a :

$$\begin{aligned} nombre de feuilles de A &\geq \sum_{i=1}^{n} 2^{n-2} \\ &\geq n2^{n-2} \\ 2^{hauteur} &\geq n2^{n-2} \\ hauteur &\geq \lceil \log_2(n2^{n-2}) \rceil \\ &\geq n - 2 + \lceil \log_2 n \rceil \end{aligned}$$

Exercice 5.5.5. *Du pétrole et des idées*

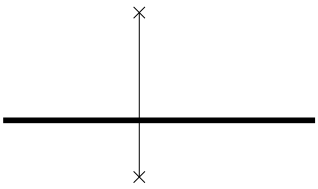
Le professeur Robert Ive est consultant pour une compagnie pétrolière qui projette de construire un grand oléoduc en ligne droite d'Est en Ouest à travers un champ pétrolier. À partir de chaque puits un raccordement doit être connecté directement sur l'oléoduc principal suivant le plus court chemin. Connaissant les coordonnées des puits, comment le professeur trouvera-t'il l'emplacement idéal de l'oléoduc principal qui minimise la longueur totale des raccordements ?

Correction.



Idée simplificatrice : projeter tous les puits sur l'axe Nord-Sud.

Avec deux points seulement, la longueur totale des raccordements est minimale si l'oléoduc passe entre les puits, ou même sur l'un des puits :



Soit y_k l'ordonnée de P_k. L'altitude optimale de l'oléoduc est le médian est y_k. La chose est supposée assez évidente pour se dispenser de la prouver.

Si le nombre n de puits est pair, on a le choix entre deux médians : inférieur ou supérieur.

Si n est impair, l'oléoduc passe par le puit médian.

Exercice 5.5.6. *Médian pondéré*

Soient n éléments distincts $x_1, ..., x_n$ de poids positifs $p_1, ..., p_n$ tels que

$$\sum_{i=1}^{n} p_i = 1$$

Le *médian pondéré (inférieur)* est l'élément x_k satisfaisant :

$$\sum_{x_i<x_k} p_i < \frac{1}{2} \quad \text{et} \quad \sum_{x_i>x_k} p_i \leq \frac{1}{2}$$

1 - Démontrer que le médian de $x_1, ..., x_n$ est le médian pondéré des x_i affectés des poids $p_i = 1/n$ pour i compris entre 1 et n.

2 - Comment calculer *simplement* le médian pondéré de n éléments. Donner la complexité en comparaisons et en additions.

3 - Comment calculer le médian pondéré en temps linéaire dans le pire des cas.

Correction.

1 - On cherche à calculer le médian des x_k.

$\sum_{x_i<x_k} p_i < \frac{1}{2}$ et $\sum_{x_i>x_k} p_i \leq \frac{1}{2}$

$\Longrightarrow$ $\frac{k-1}{n} < \frac{1}{2}$ et $\frac{n-k}{n} \leq \frac{1}{2}$

$\Longrightarrow$ $k - 1 < \frac{n}{2}$ et $n - k \leq \frac{n}{2}$

$\Longrightarrow$ $\frac{n}{2} \leq k < \frac{n}{2} + 1$

$\Longrightarrow$ $k = \lfloor \frac{n}{2} \rfloor$ c'est le médian inférieur.

2 - Algorithme simple de calcul du médian pondéré

début

Trier les éléments : $x_{\sigma(1)} < \cdots < x_{\sigma(n)}$;

Sommer les poids $p_{\sigma(k)}$ jusqu'à avoir : $p_{\sigma(1)} + \cdots + p_{\sigma(k+1)} \geq \frac{1}{2}$;

k est alors le médian pondéré ;

fin

Complexité :

$\emptyset(n \log n)$ comparaisons à cause du tri

$\emptyset(n)$ additions dans la sommation

3 - Algorithme en temps linéaire de calcul du médian pondéré

début
Poser $E = \{ (x_i, p_i) \}$;
si $|E| = 1$ **alors** terminé : renvoyer x_1 ;
Calculer le médian x des x_i, noter p son poids ;
Construire $E_1 = \{ (x_i, p_i) \ / \ x_i < x \}$;
Construire $E_2 = \{ (x_i, p_i) \ / \ x_i > x \}$;
Calculer $S_1 = \sum_{E_1} p_i$;
si $S_1 > \frac{1}{2}$ **alors**
relancer l'algorithme sur E_1 ;
sinon
relancer l'algorithme sur $E_2 \cup \{ (x, p + S_1 \}$;
fin
k est alors le médian pondéré ;
fin

Lorsque l'algorithme est relancé, il faut réajuster tous les poids pour que leur somme fasse toujours 1.

Calcul de $I(n)$, complexité d'une itération de l'algorithme, sans récursivité :
- Le calcul du médian non pondéré s'effectue en $\emptyset(n)$
  (cf. cours : algorithme faisant des paquets de cinq).
- Les autres opérations s'effectuent en $\emptyset(n)$ également.

$\Longrightarrow$ $I(n) = \emptyset(n)$

Complexité totale : $T(n) = T(\frac{n}{2}) + I(n) = T(\frac{n}{2}) + \emptyset(n)$

On peut alors appliquer le Master Theorem :

Notations : $T(n) = a\,T(\frac{n}{b}) + \emptyset(n^\alpha)$
Valeurs : $a = 1$; $b = 2$; $\alpha = 1$
Cas : $1 = a < b^\alpha = 2$
Conclusion : $T(n) = \emptyset(n^\alpha) = \emptyset(n)$

$T(n) = \emptyset(n)$ et on a donc un algorithme linéaire pour calculer le médian pondéré.

Exercice 5.5.7. *Le problème de l'emplacement du bureau de poste*

Considérons un espace métrique (E, d). Soient n éléments distincts $x_1, ..., x_n$ de poids respectifs $p_1, ..., p_n$ tels que

$$\forall i,\ p_i \geq 0 \qquad \text{et} \qquad \sum_{i=1}^{n} p_i = 1$$

On souhaite trouver un point x (qui n'est pas nécessairement un des x_i) qui minimise $\sum_{i=1}^{n} p_i d(x, x_i)$.

1 - **Cas de la dimension 1 :** (E, d) est l'ensemble des réels muni de la distance usuelle. Montrer que le médian pondéré est une des solutions optimales.

2 - **Cas de la dimension 2 :** (E, d) est le plan muni de la distance 1, ou *distance Manhattan*, c'est à dire :

$$d((x_1, y_1), (x_2, y_2)) = |x_1 - x_2| + |y_1 - y_2|$$

Résoudre l'énoncé qui s'appelle dans ce cas ¡¡problème du bureau de poste¿¿.

Correction.

On suppose les x_i triés tels que $x_1 < \cdots < x_n$.

1 - Il est évident que $x \in [\, x_1 \,;\, x_n \,]$.
On note x_m le médian pondéré des (x_i, p_i).

Soit $x \in [\, x_i \,;\, x_{i+1} \,]$ et $\epsilon > 0$ avec $x + \epsilon \in [\, x_i \,;\, x_{i+1} \,]$.
La situation est la suivante : $x_i \leq x < x + \epsilon \leq x_{i+1}$

$S(x) = \sum_{k=1}^{n} p_i \, |x - x_i|$
La définition de x impose que $S(x)$ soit minimal.

$$S(x) = \sum_{x_k < x} p_k \, (x - x_k) + \sum_{x < x_k} p_k \, (x_k - x)$$

$$S(x+\epsilon) = \sum_{x_k \leq x} p_k \, (x + \epsilon - x_k) + \sum_{x_k > x} p_k \, (x_k - x - \epsilon)$$

$$= S(x) + \epsilon \left(\sum_{x_k \leq x} p_k - \sum_{x_k > x} p_k \right)$$

$$= S(x) + \epsilon \left(\sum_{x_k \leq x} p_k + \sum_{x_k \leq x} p_k - 1 \right) \qquad \text{car :} \quad \sum_{k=1}^{n} p_k = 1$$

$$S(x+\epsilon) = S(x) + \underbrace{\epsilon}_{>\,0} \left(2 \sum_{x_k \leq x} p_k - 1 \right)$$

$$S(x+\epsilon) < S(x) \iff \sum_{x_k < x_{i+1}} p_k < \tfrac{1}{2}$$

Si $x < x_{i+1} < x_m$ alors $\sum_{x_k < x_{i+1}} p_k < \sum_{x_k < x_m} p_k < \frac{1}{2} \implies S(x+\epsilon) < S(x)$
$\implies S(x)$ n'est pas minimal donc x n'est pas un médian pondéré.

On en déduit : $x \geq x_m$.

En posant $\epsilon < 0$, on montre avec un raisonnement similaire que $x \leq x_m$, ce qui permet de conclure : $x = x_m$. Corrolaire : l'exercice 2 n'a pas servi à rien !

2 - On calcule indépendamment x_k médian sur les x, et y_l médian sur les y.
La solution au problème est le point (x_k, y_l).

Note 1 : ne pas commettre l'erreur de se limiter au calcul d'un seul médian et de déclarer solution le point (x_k, y_k). En dimension 2, le point solution n'est pas toujours confondable avec un des P_k comme c'était le cas dans la question précédente.

Note 2 : cette solution simple convient car on utilise la distance de Manhattan. Elle ne fonctionnerait pas avec la distance usuelle $d(A,B) = \sqrt{(x_A - x_B)^2 + (y_A - y_B)^2}$.

Exercice 5.5.8. *Un nouvel algorithme de tri*

Soit T un tableau contenant n entiers distincts, la procédure `Nouveau_Tri` est définie de la manière suivante :

```
Nouveau_Tri (T,i,j)
| si T[i] > T[j] alors échanger les valeurs T[i] et T[j] ;
| si i ≤ j − 2 alors faire
|     k ← ⌊(j − i + 1)/3⌋ ;
|     Nouveau_Tri(T, i, j − k) ;
|     Nouveau_Tri(T, i + k, j) ;
|     Nouveau_Tri(T, i, j − k) ;
```

1 - Montrer que `Nouveau_Tri`$(T, 1, n)$ trie correctement le tableau T.

2 - Donner la complexité en nombre de comparaisons de cet algorithme. Comparer avec les complexités des algorithmes de tri vus en cours (*Rappel :* $\ln(2) \simeq 0,7$ et $\ln(3) \simeq 1,1$).

Correction.

1 - L'algorithme trie correctement.
- Pour une longueur inférieure à deux l'algo est réduit à l'échange ce qui donne un tableau trié.
- Pour i supérieur à trois la première ligne ne sert à rien. On notera $X \leq Y$ si $\forall x \in X$ $\forall y \in Y : x \leq y$. On pose $l = \|T\|$,on pose A l'ensemble des $(\lfloor \frac{l+1}{3} \rfloor)$ premier élément de T, C l'ensemble des $(\lceil \frac{l}{3} \rceil)$ élément suivants , on pose B l'ensemble des $(\lfloor \frac{l}{3} \rfloor)$ dernier élément. On commence par trier les éléments de $A \cup C$ par induction : comme $\|A \cup C\| = (\lceil \frac{2l}{3} \rceil) < l$. Donc on obtient $A < C$. On trie maintenant $C \cup B$ par induction : comme $\|A \cup B\| \leq (\lceil \frac{2l}{3} \rceil) < l$.Alors $\|B\| \leq \|C\|$ ce qui implique que si un élément b de B avant le tri se retrouve dans B après, c'est qu'il existe un élément c dans C avant le tri tel que $c < b$ or comme c est supérieur à tous les éléments de A b aussi donc après le tri $C \leq B$ et $A \leq B$ ou encore $C \cup A \leq B$. Finalement on retrie $A \cup C$ et on obtient $A < B < C$ avec A, B, C, trié donc T est trié.

2 - Complexité

Pour trier un ensemble de n (n supérieur à trois) éléments on trie trois ensembles à $\lceil \frac{2n}{3} \rceil$ éléments et on fait un échange. Pour trier un ensemble de 2 éléments on fait un échange. Donc :

$$C(n) = 1 + C(\lceil \tfrac{2n}{3} \rceil) \text{ et } C(1) = 2$$

donc par master théorème :

$$C(n) = O(n^{log_{\frac{3}{2}}(3)}) \simeq O(n^{2,75})$$

C'est mauvais.

Exercice 5.5.9. *Tas faibles & tri*

Une technique classique de tri par comparaisons consiste à utiliser la structure de données appelée *tas binaire*. Le tri correspondant a une complexité totale en $2n \log_2(n) + O(n)$ comparaisons pour n éléments. L'objectif de cet exercice est d'étudier une variante de cette structure de données, les *tas faibles*, pour améliorer la complexité et se rapprocher de la borne inférieure théorique.

Un *tas faible* pour n éléments est un arbre binaire où chaque nœud est étiqueté par un élément, et qui vérifie les conditions suivantes :
(i) Les éléments du sous-arbre droit de chaque nœud sont inférieurs ou égaux à l'élément associé au nœud.
(ii) La racine de l'arbre n'a pas de fils gauche.
(iii) Les feuilles se trouvent uniquement sur les deux derniers niveaux de l'arbre.

Pour coder cette structure de données, nous utiliserons deux tableaux $t[0..n–1]$ et $r[0..n–1]$. Le tableau t contiendra les n éléments et le tableau r est un tableau de bits 0 ou 1 (qui permettra de distinguer fils gauche et fils droit d'un nœud, et ensuite de les permuter à faible coût). Plus précisément, avec le tableau t, le fils gauche de l'élément d'indice i a pour indice $2i + r[i]$ et son fils droit a pour indice $2i + 1 - r[i]$ (on fixe juste $r[0] = 0$ pour que la racine d'indice 0

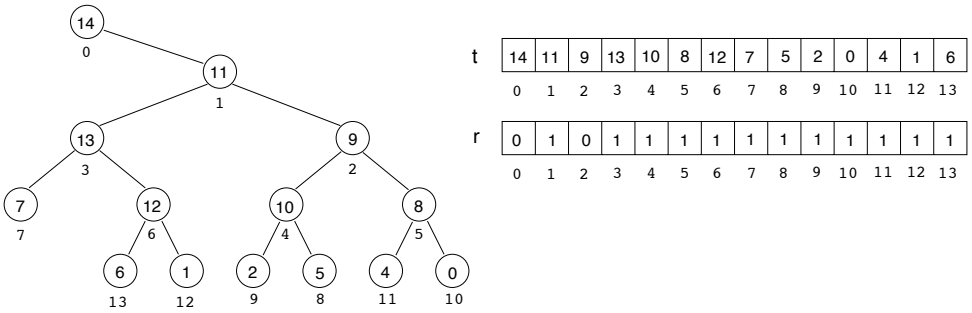


FIG. 5.3 – Exemple de tas faible et un codage de cet arbre.

ait un unique fils qui soit à droite). Changer le bit $r[i]$ revient alors à échanger fils droit et fils gauche de l'élément d'indice i et donc permuter le sous-arbre droit et le sous-arbre gauche de cet élément. La figure 5.3 présente un exemple de tas faible et de codage de cet arbre.

Les successeurs d'un nœud sur la *branche gauche* du *sous-arbre droit* sont appelés ses *petits-fils*. Par exemple sur la figure 5.3, les petits-fils de la racine sont les éléments 11, 13 et 7. On note `Pere(`i`)` l'indice du père du nœud d'indice i, et la fonction `Gpere(`i`)` (pour *Grand-père*) définie par `Gpere(Pere(`i`))` si i est le fils gauche de `Pere(`i`)` et par `Pere(`i`)` si i est le fils droit de `Pere(`i`)`. Par exemple avec le codage de la figure 5.3 le nœud d'indice 9 (de valeur $t[9] = 2$) a pour grand-père `Gpere(9)`$= 1$ (indice du nœud avec la valeur $t[1] = 11$).

1 - Pour le codage de l'arbre avec les tableaux t et r, écrire précisément la fonction `Gpere(`i`)` qui calcule l'indice du grand-père de l'élément d'indice i dans t.

Soit `echange(`i`,`j`)` une fonction qui échange les valeurs de $t[i]$ et $t[j]$. Soit `fusion(`i`,`j`)` la fonction définie par : `si` $(t[i] > t[j])$ `alors echange(`i`,`j`) et` $r[i] \leftarrow 1 - r[i]$.

2 - Etant donné un tableau de n valeurs $t[0..n–1]$, on lui associe le tableau $r = [0, 0, \ldots, 0]$ où tous les bits sont à 0, et on effectue la fonction `ConstruitTas(`t`,`r`)` définie par :

`pour` i `de` $n-1$ `à 1, faire fusion(`i`,Gpere(`i`))`

Montrer que pendant le déroulement de la boucle, après avoir effectué `fusion(`i`,Gpere(`i`))`, alors pour tout $j \geq i$, les valeurs du sous-arbre de racine le nœud d'indice j sont toutes inférieures ou égales à la valeur du nœud d'indice `Gpere(`j`)`. En déduire qu'en fin de boucle, on obtient un tas faible. Combien de comparaisons a-t-on effectué ?

3 - Etant donné un tas faible représenté par les tableaux t et r, supposons que l'on remplace la valeur $t[0]$ de la racine par une valeur quelconque, comment reconstruire rapidement un tas faible avec cette valeur et à quel coût en nombre de comparaisons ?

Indication : mettre à jour en parcourant les petits fils de la racine.

4 - En déduire un algorithme de tri d'un tableau t en $n \log_2(n) + \mathcal{O}(n)$ comparaisons, qui commence par construire un tas faible puis exploite cette structure de données.

Correction.

Laissée au lecteur.

5.6 Références bibliographiques

Ce chapitre est un grand classique. Nous avons surtout utilisé le livre de Froidevaux, Gaudel et Soria [4] (cours et nombreux exercices). L'exercice *Plus petit et plus grand* est dû à Rawlins [8].

Chapitre 6

Graphes

6.1 Définitions

Soit $G = (V, E)$ un graphe non orienté, où V est l'ensemble des sommets et E l'ensemble des arêtes. On rappelle quelques définitions élémentaires :

Degrés Le degré $\delta(v)$ d'un sommet $v \in V$ est le nombre d'arêtes de E incidentes en v. Le degré $\Delta(G)$ du graphe G est le maximum des degrés d'un sommet.

Chemin Soient u, v deux sommets de V. Un chemin de u à v est un ensemble d'arêtes $e_1, e_2, \ldots, e_k$ de E telles que $e_i = (v_{i-1}, v_i)$, $v_0 = u$ et $v_k = v$. Si tous les v_i sont distincts, le chemin est élémentaire. Si $u = v$, le chemin (élémentaire) est un cycle (élémentaire).

Composantes connexes Une composante connexe est une classe d'équivalence de la relation R définie sur $V \times V$ par uRv si et seulement s'il existe un chemin de u à v.Le graphe G est connexe si tous les sommets sont dans la même composante connexe.

Sous-graphe Un sous-graphe de G est un graphe (W, F) où $W \subset V$, $T \subset E$, et toutes les arêtes de F relient deux sommets de W. Le sous-graphe induit par un sous-ensemble de sommets comprend *toutes* les arêtes qui ont leurs deux extrémités dans ce sous-ensemble. De même pour le sous-graphe induit par un sous-ensemble d'arêtes.

Dans le cas orienté, on parle d'arcs au lieu d'arêtes. On définit le degré sortant et le degré entrant d'un sommet. Deux sommets u et v sont dans une même composante connexe s'il existe un chemin de u à v et un chemin de v à u. Pour bien distinguer du cas non-orienté, on parle plutôt de composante fortement connexe.

6.2 Arbres

6.2.1 Caractérisation

Définition 6. *Un arbre est un graphe connexe sans cycle.*

Théorème 11. *Soit G un graphe à n sommets et m arêtes. On a l'équivalence :*

1. *G est connexe sans cycle (définition d'un arbre)*
2. *G est connexe et minimal pour cette propriété (si on supprime une arête, G n'est plus connexe)*
3. *G est connexe et $m = n - 1$*
4. *G est sans cycle et maximal pour cette propriété (si on ajoute une arête, G a un cycle)*
5. *G est sans cycle et $m = n - 1$*

6. Toute paire de sommets est reliée par un chemin élémentaire et un seul

Preuve. La preuve est facile. Tout repose sur la construction suivante. On part des n sommets, sans aucune arête. Il y a alors n composantes connexes, une par sommet. On ajoute les arêtes une par une. Quand une ajoute une nouvelle arête, il y a deux cas :

- Soit cette arête relie deux sommets qui n'étaient pas dans la même composante connexe. Les composantes connexes de ces deux sommets fusionnent, et le nombre total de composantes connexes diminue d'une unité.
- Soit cette arête relie deux sommets qui étaient déjà dans une même composante connexe. Elle crée un cycle (avec le chemin qui reliait les deux sommets). Le nombre total de composantes connexes reste inchangé.

On voit en particulier que pour passer de n composantes connexes à une seule, il faut au moins $n-1$ arêtes, et qu'aucune de celles-ci en crée de cycle.

Ce bout de preuve permet de montrer *proprement* que trouver le maximum de n éléments dans un ensemble muni d'une relation d'ordre total exige de faire au moins $n-1$ comparisons. En effet, on ajoute une arête entre deux éléments qu'on compare. S'il reste deux (ou plus) composantes connexes à la fin, le maximum peut se trouver dans n'importe laquelle.

□

6.2.2 Parcours d'arbres binaires

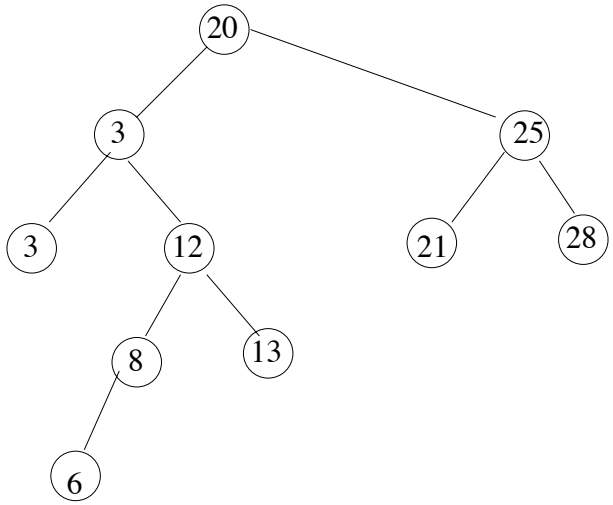
Il y a trois parcours classiques en profondeur :

Préfixe (père, fils gauche, fils droit)

Postfixe (fils gauche, fils droit, père)

Infixe (fils gauche, père, fils droit)

Par contre, le parcours en largeur est un parcours niveau par niveau, de la racine jusqu'aux feuilles. C'est une exploration "hiérarchique" de l'arbre. Considérons l'exemple :



Pour cet exemple, si on imprime le numéro du sommet *père* au moment où on le visite, on trouve :

Préfixe 20, 3, 3, 12, 8, 6, 13, 25, 21, 28

Postfixe 3, 6, 8, 13, 12, 3,21, 28, 25, 20

Infixe 3, 3, 6, 8, 12, 13, 20, 21, 25, 28

Largeur 20, 3, 25, 3, 12, 21, 28, 8, 13, 6

On donne ci-dessous des programmes Java pour tous ces parcours.

Parcours préfixe

```
class Arbre {
  int      contenu;
  Arbre    filsG;
  Arbre    filsD;

  Arbre (int v, Arbre a, Arbre b) {
      this.contenu = v;
      this.filsG = a;
      this.filsD = b;
  }

  static int taille(Arbre a) {
    if (a == null)
        return 0;
    else
        return 1 + taille (a.filsG) + taille (a.filsD);
  }

  static Arbre nouvelArbre(int v, Arbre a, Arbre b) {
      return new Arbre (v, a, b);
  }

  public static void main(String args[]) {
    Arbre a5, a7;
    a5 = nouvelArbre (12, nouvelArbre (8, nouvelArbre (6, null, null), null),
                          nouvelArbre (13, null, null));
    a7 = nouvelArbre (20, nouvelArbre (3, nouvelArbre (3, null, null), a5),
                          nouvelArbre (25, nouvelArbre (21, null, null),
                                           nouvelArbre (28, null, null)));
   prefixe(a7);
  }

  static void prefixe(Arbre a) {
    if (a==null) return;
    System.out.println(a.contenu+" ");
    prefixe(a.filsG);
    prefixe(a.filsD);
  }
}
```

Les trois parcours en profondeur

```
class Arbre {
...
```

```
  // parcours préfixe P G D
  static void prefixe(Arbre a) {
    if (a==null) return;
    System.out.println(a.contenu+" ");
    prefixe(a.filsG);
    prefixe(a.filsD);
  }

  // parcours postfixe G D P
  static void postfixe(Arbre a) {
    if (a==null) return;
    postfixe(a.filsG);
    postfixe(a.filsD);
    System.out.println(a.contenu+" ");
  }

  // parcours infixe G P D
  static void infixe(Arbre a) {
    if (a==null) return;
    infixe(a.filsG);
    System.out.println(a.contenu+" ");
    infixe(a.filsD);
  }
}
```

En profondeur (préfixe) avec une pile

```
   static void prefixeIter(Arbre a) {
     if (a==null) return;
     Pile P = new Pile();
     P.push(a);
     while (!P.estVide()) {
         a = P.pop(); // imprime a.contenu
         if (a.filsD != null) P.push(a.filsD);
         if (a.filsG != null) P.push(a.filsG);
  }
```

En largeur avec une file

```
   static void largeurIter(Arbre a) {
     if (a==null) return;
     File F = new File();
     F.fush(a);
     while (!F.estVide()) {
         a = F.pop(); // imprime a.contenu
         if (a.filsG != null) F.fush(a.filsG);
         if (a.filsD != null) F.fush(a.filsD);
  }
```

6.2.3 Arbres binaires de recherche

Dans un arbre binaire de recherche, on impose un ordre partiel sur les contenus : les nœuds du sous-arbre gauche (resp : droit) ont une valeur inférieure ou égale (resp : supérieure) à la valeur du père :

$$\text{valeur[sous-arbre gauche]} \leq \text{valeur[nœud]} < \text{valeur[sous-arbre droit]}$$

On veut que les procédures de recherche, d'insertion d'un nouveau nœud, et de suppression d'un nœud, aient un coût proportionnel à la hauteur de l'arbre. C'est facile pour les deux premières, mais cela demande une analyse plus fine pour la suppression : comment reconstruire l'arbre ? Voici la méthode :

- Pour supprimer la clé x de l'arbre a, on cherche d'abord le sommet s dont la clé vaut x.
- Quand on l'a trouvé, il faut disposer d'une méthode pour supprimer la racine du sous-arbre de s :
  1. Si s n'a qu'un fils unique, on le remplace par celui-ci.
  2. Sinon, il faut chercher le sommet t qui est le prédécesseur de la racine (au sens de l'ordre sur les clés). On remplace alors la clé de s par celle de t, et on supprime t.
- Lemme : t n'a pas de fils droit.
- Le prédécesseur est donc le "dernier" fils droit en descendant la chaîne à partir du fils gauche de la racine.

On donne ci-dessous des programmes Java pour les trois opérations.

Recherche

```
 static Arbre recherche(int v, Arbre a) {
  if (a == null || v == a.contenu)
       return a;
  else
       if (v < a.contenu)
           return recherche(v, a.filsG);
       else
           return recherche(v, a.filsD);
}

static boolean estDans(int v, Arbre a) {
   return (recherche(v,a) != null);
}
```

Insertion

```
static Arbre ajouter(int v, Arbre a) {
   if (a == null)
       a = nouvelArbre(v, null, null);
   else if (v <= a.contenu)
      a.filsG = ajouter(v, a.filsG);
  else
      a.filsD = ajouter(v, a.filsD);
  return a;
}
```

```
static Arbre ajouterSansModif(int v, Arbre a) {
   if (a == null)
       a = nouvelArbre(v, null, null);
   else if (v <= a.contenu)
      return nouvelArbre(v, ajouter(v, a.filsG), a.filsD);
  else
      return nouvelArbre(v, a.filsG, ajouter(v, a.filsD));
}
```

Suppression

```
static Arbre supprimer(int v, Arbre a) {
   if (a == null) return a;
   if (v == a.contenu)
       return eliminerRacine(a);
   if (v < a.contenu)
       a.filsG = supprimer(v, a.filsG);
   else
       a.filsD = supprimer(v, a.filsD);
   return a;
}

static Arbre dernierFils(Arbre a) {
   if (a.filsD == null)
       return a;
   else
       return dernierFils(a.filsD);
}

static Arbre eliminerRacine(Arbre a) {
   if (a.filsG == null)
       return a.filsD;
   if (a.filsD == null)
       return a.filsG;
   Arbre b = dernierFils(a.filsG);
   a.contenu = b.contenu;
   a.filsG = supprimer(a.contenu, a.filsG);
   return a;
}
```

Version itérative de la suppression

```
static Arbre supprimerI(int v, Arbre a) {
   if (a == null) return a;
   if (v == a.contenu)
       return eliminerRacineI(a);
   if (v < a.contenu)
       a.filsG = supprimerI(v, a.filsG);
   else
```

```
        a.filsD = supprimerI(v, a.filsD);
    return a;
}

static Arbre eliminerRacineI(Arbre a) {
    if (a.filsG == null)
        return a.filsD;
    if (a.filsD == null)
        return a.filsG;
    Arbre g = a.filsG;
    Arbre h = g.filsD;
    if (h == null) { // g n'a pas de fils droit
        a.contenu = g.contenu; // transfert de clé
        a.filsG = g.filsG; // suppression de g
        return a;
    }
    while (h.filsD != null) { // descente branche droite
        g = h;
        h = h.filsD; // h fils droit de g
    }
    a.contenu = h.contenu; // transfert de clé
    g.filsD = h.filsG; // suppression de h
    return a;
}
```

Pour terminer avec les arbres de recherche, voici un petit exercice. Soit A un arbre de recherche. Montrer que :

1. Le parcours infixe ordonne les noeuds par valeur croissante.
2. Si un noeud a deux fils, son successeur dans l'ordre infixe n'a pas de fils gauche et son prédécesseur (dans l'ordre infixe) n'a pas de fils droit.
3. Montrer que le successeur d'un noeud n est le sommet le plus à gauche dans le sous-arbre droit issu de n.

6.3 Structures de données pour les graphes

Pour représenter un graphe en machine, plusieurs structures de données sont utilisées, notamment matrices ou listes de successeurs. La complexité des algorithmes dépendra de la représentation choisie, d'où la nécessité de la choisir soigneusement.

Dans cette section, on utilise les notations suivantes :
- $G = (V, E)$, V sommets et E arêtes (ou arcs dans le cas orienté),
- Le nombre de sommets est $n = |V|$,
- Le nombre d'arêtes est $m = |E|$ ($m = O(n^2)$).

Matrice d'adjacence
- $M_{i,j} = \begin{cases} O & \text{si } (i,j) \notin E \\ 1 & \text{si } (i,j) \in E \end{cases}$
- M est symétrique si le graphe est non-orienté
- Utilisation : accessibilité, plus courts chemins (propriétés globales du graphe).

Matrice d'incidence

- Pour un graphe orienté sans boucle
- Taille $n \times m$
- $M_{i,a} = \begin{cases} 1 & \text{si } i \text{ est l'origine de } a \\ -1 & \text{si } i \text{ est l'extrémité de } a \\ 0 & \text{sinon} \end{cases}$
- Utilisation : branchements, programmation linéaire
- Exercice : M est totalement unimodulaire : le déterminant de toute sous-matrice carrée de M vaut 0, 1 ou -1.

Listes de successeurs

- Pour chaque sommet, la liste de ses successeurs
- On a un tableau de taille n pour les débuts de liste
- Deux variantes :
  - GrapheSucc si tous les sommets ont à peu près le même nombre de sucesseurs : on stocke ceux-ci dans un tableau, avec une sentinelle Omega = -1 pour marquer le dernier successeur
  - GrapheListe une vraie structure de liste
- Utilisation : parcours dans tous les sens, connexité

Structures de données en Java

```
class GrapheMat {
  int m[][];
  int nb;

  GrapheMat(int n1) {
  // Initialisation: n1 sommets, matrice vide
      nb = n1;
      m = new int[n1][n1];
  }

  GrapheMat (int n1, int p) {
  // Initialisation: n1 sommets, matrice
  // Pour un arc entre i et j:  proba 1/p d'exister
      nb = n1;
      m = new int[n1][n1];
      for (int i = 0; i < nb; ++i)
        for (int j = 0; j < nb; ++j) {
          int a  = (int) (Math.random() * p);
          if (a == p - 1 ) m[i][j] = 1;
            else m[i][j] = 0;
        }
  }

  static void imprimer (GrapheMat g) {
      System.out.println ("nombre de sommets " + g.nb);
      for (int i = 0; i < g.nb ; ++i) {
          for (int j = 0; j < g.nb; ++j)
              System.out.print (g.m[i][j] + " ");
```

```
        System.out.println ();
      }
      System.out.println ();
  }
}


class GrapheSucc{
  int succ[][];
  int nb;
  final static int Omega = -1;

  GrapheSucc(int n) {
      nb = n;
      succ = new int[n][n];
  }

  static void imprimer(GrapheSucc g) {
     System.out.println ("nombre de sommets " + g.nb + " ");
     for (int i = 0; i < g.nb; ++i)
       { System.out.println ("sommet " + i + ", : successeurs: ");
         for (int j = 0; g.succ[i][j] != Omega; ++j)
         System.out.print (g.succ[i][j] +" ");
         System.out.println ();
       }
     System.out.println();
  }
}


class GrapheListe{
  Liste listeSucc[];
  int nb;

  GrapheListe (int n1) {
      nb = n1;
      listeSucc = new Liste[n1];
  }

  static void imprimer (GrapheListe g) {
      System.out.println ("nombre de sommets " + g.nb);
      for (int i = 0; i < g.nb; ++i) {
          System.out.println ("sommet " + i + ", : successeurs: ");
          for (Liste u = g.listeSucc[i]; u != null; u = u.suivant)
              System.out.print (" " + u.contenu);
         System.out.println ();
      }
      System.out.println ();
  }
```

```
}
```

Comparaison Pour un graphe à n sommets et m arcs, la taille mémoire nécessaire est :
- Matrice d'adjacence : n^2.
- Matrice d'incidence : $n \times m$.
- Liste de successeurs : $n + m$.

Conversions

```
GrapheMat (GrapheSucc h) {
// Matrice de successeurs -> matrice d'adjacence
    nb = h.nb;
    m= new int[nb][nb];
    for (int i = 0; i < nb ; ++i)
       for (int j = 0; j < nb ; ++j) m[i][j] = 0;
    for (int i = 0; i < nb ; ++i)
        for (int k = 0; h.succ[i][k] != GrapheSucc.Omega; ++k)
            m[i][h.succ[i][k]] = 1;
}

GrapheSucc (GrapheMat g) {
// Matrice d'adjacence -> matrice de successeurs
  nb = g.nb;
  int nbMaxSucc = 0;
  int temp = 0;
  for (int i = 0; i < nb ; ++i) {
      for (int j = 0; j < nb  ; ++j)
          if (g.m[i][j] != 0) temp++;
      nbMaxSucc = Math.max (nbMaxSucc, temp);
      temp = 0;
  }
  succ = new int[nb][nbMaxSucc + 1];
  for (int i = 0; i < nb ; ++i) {
      int k = 0;
      for (int j = 0; j < nb  ; ++j)
          if (g.m[i][j] != 0)
              succ[i][k++] = j;
      succ[i][k] = Omega;
  }
}

GrapheListe (GrapheSucc g) {
// Matrice de successeurs -> tableau de listes de successeurs
  nb = g.nb;
  listeSucc = new Liste[nb];
  for (int i = 0; i < nb ; ++i) {
      listeSucc[i] = null;
      for (int k = 0; g.succ[i][k] != GrapheSucc.Omega; ++k)
          listeSucc[i] = Liste.ajouter(g.succ[i][k], listeSucc[i]);
  }
```

```
}
```

6.4 Accessibilité

6.4.1 Rappels sur les relations binaires

Soit E un ensemble fini, $n = |E|$, et R une relation binaire (c'est-à-dire une partie de $E \times E$). Pour un élément $\{(x, y)\} \in R$ on écrira aussi $x\,R\,y$. On définit par récurrence l'élévation à la puissance R^i :

$$\left| \begin{array}{l} R^i = R \cdot R^{i-1} \\ R^0 = id \end{array} \right.$$

Par exemple, $R^2 = \{\,(x, z)\,/\,\exists y\,,\, xRy \wedge yRz\}$.

Lemme 7. $\exists\, p, q/\ 0 \leq p < q \leq 2^{(n^2)}$ *tq* $R^p = R^q$

Preuve. Il y a autant de relations binaires que de parties de $E \times E$. Comme $|E \times E| = n^2$, il y a exactement $2^{(n^2)}$ relations binaires. □

Définition 7. *- Soit S est une relation binaire ; on dit que S est transitive* $\Leftrightarrow S^2 \subseteq S$.
- Soit R est une relation binaire ; on définit la fermeture transitive R^+ par :

$$R^+ = \bigcap_{\substack{S \supseteq R \\ S \text{transitive}}} S$$

Remarque 1.
- *il existe au moins une relation transitive S contenant R (notamment $\forall x, y, xSy$).*
- *l'intersection de deux relations transitives est transitive.*

Proposition 2. $xR^+y \Leftrightarrow$

$$\exists p > 0 \,|\, \exists x_1 \ldots x_p \in E \text{ avec } x_1 = x, x_p = y \text{ et } x_i R x_{i+1} \text{ pour } 1 \leq i \leq p$$

$$\text{ce qui peut aussi s'écrire : } R^+ = \bigcup_{p \geq 1} R^p$$

Preuve. Nommons xCy la relation *chemin de* R. Montrons que $C = R^+$:
- $R^+ \subseteq C$: C contient R et C est transitive, donc C est un des termes de l'intersection dans la définition de R^+.
- $C \subseteq R^+$: fixons un chemin de x à y, et soit S transitive et contenant R. Comme S contient R, S doit contenir tous les arcs du chemin considéré ; et comme S est transitive, S doit contenir le couple (x, y). Maintenant R^+ est l'intersection de tels relations S, donc (x, y) est dans R^+.

□

Proposition 3.

$$(1)\quad R^n \subseteq \bigcup_{0 \leq i \leq n-1} R^i \qquad et \qquad (2)\quad \forall p \geq 1, R^p \subseteq \bigcup_{1 \leq i \leq n} R^i$$

Preuve. 1. Soit $S = \bigcup_{0 \leq i \leq n-1} R^i$ et supposons $aR^n b$, on va montrer qu'alors aSb. Il existe $x_0, \ldots, x_n \in E$, $x_0 = a$, $x_n = b$, $x_i R x_{i+1}$ pour $0 \leq, i < n$. On a $n+1$ valeurs de x donc il existe p et q tels que $x_p = x_q$, avec $0 \leq p < q \leq n$. Si $p = 0$ et $q = n$, alors $a = b$ donc $aR^0 b$ donc aSb. Sinon, on a $aR^p x_p = x_q = R^{n-q} x_n$, donc $aR^s b$ avec $s = n + p - q \leq n - 1$, et aSb.

2. Par récurrence sur p :
   - $p = 1$: $R^1 \subseteq \bigcup_{1 \leq i \leq n} R^i$
   - $p \to p+1$:
     on vient de montrer que $R^n \subseteq \bigcup_{0 \leq i \leq n-1} R^i$, donc $\bigcup_{0 \leq i \leq n} R^i \bigcup_{0 \leq i \leq n-1} R^i$
     si par hypothèse de récurrence $R^p \subseteq \bigcup_{1 \leq i \leq n} R^i \subseteq \bigcup_{0 \leq i \leq n-1} R^i$,
     alors en composant par R on a
     $R^{p+1} \subseteq \bigcup_{1 \leq i \leq n} R^i$

□

Définition 8. *Fermeture reflexive et transitive :*

$$R^* = Id \cup R^+ = \bigcup_{0 \leq i \leq n-1} R^i$$

6.4.2 Chemins dans les graphes

- Soit M la matrice d'adjacence de $G = (V, E)$. On considère ses coefficients comme des entiers.
- **Lemme** : soit M^p la puissance p-ème de M. Le coefficient M^p_{ij} est égal au nombre de chemins de longueur p de v_i à v_j
- Preuve par récurrence :
  - Vrai par définition pour $p = 1$
  - $M^p_{ij} = \sum_{k=1}^{n} M^{p-1}_{ik} . M_{kj}$, cette relation exprime toutes les décompositions possibles d'un chemin de longueur p en un chemin de longueur $p-1$ suivi d'une dernière arête
- Notons que si on considère les coefficients de M comme des booléens, M^p donne l'existence d'un chemin de longueur p (cf. ce qui précède).

6.4.3 Fermeture transitive

Algorithme naïf

- $G^+ = (V, A^+)$ défini par

$$(i, j) \in A^+ \Leftrightarrow \text{il y a un chemin non vide de } v_i \text{ à } v_j \text{ dans } G$$

- $M^+ = M + M^2 + \ldots + M^n$ est la fermeture transitive
- $M^* = I + M^+$ est la fermeture réflexive et transitive
- Coût de calcul de M^+ en $O(n^4)$ car on doit faire n multiplications de matrices
- Coût de calcul de M^* en $O(n^3 \log n)$ car on peut calculer "plus" : $\lceil \log n \rceil$ élévations successives au carré de $I + M$ (c'est l'algorithme d'exponentiation binaire vu Section 1.3)

Algorithme de Roy-Warshall L'algorithme de Roy-Warshall résout le problème de fermeture transitive d'un graphe. Au passage, ce problème est clairement un sous-problème du problème de plus court chemin. Il suffit pour le montrer de prendre tous les poids des arêtes égaux à 0. Dans une telle situation, $\delta(u, v)$ vaut 0 s'il existe un chemin entre u et v, et vaut $+\infty$ dans le cas contraire. C'est ce qui permetttra d'étendre facilement l'algorithme pour traiter des plus courts chemins dan sun graphe sans cycle de poids négatif.

Proposition 4. *La récurrence* $A_{ij}^{(p)} = A_{ij}^{(p-1)} + A_{ip}^{(p-1)} A_{pj}^{(p-1)}$ *initialisée avec* $A^{(0)} = M$*, calcule* $M^+ = A^{(n)}$*.*

Preuve. La preuve est fondée sur l'idée que $A_{ij}^{(p)} = 1$ si et seulement s'il existe un chemin de i à j dont tous les sommets intermédiaires sont de numéro inférieur ou égal à p. Ce qui se démontre par récurrence sur p.

- Initialisation $p = 0$: on n'utilise aucun sommet intermédiaire, donc les chemins sont réduits aux arêtes du graphe, et on a bien posé $A^{(0)} = M$.
- Le premier sens de l'équivalence est vérifié par construction. Si $A_{ij}^{(p)} = 1$, alors on a au moins une des égalités $A_{ij}^{(p-1)} = 1$ ou $A_{ip}^{(p-1)} A_{pj}^{(p-1)} = 1$, qui traduisent toutes deux l'existence d'un chemin de longueur inférieure ou égale à p, entre i et j, par hypothèse de récurrence.
- Pour la réciproque, prenons un chemin de i à j de longueur inférieure ou égale à p, et montrons que $A_{ij}^{(p)} = 1$:
  - Si ce chemin passe par le sommet p, on a $i \rightsquigarrow p \rightsquigarrow j$, avec $i \rightsquigarrow p$ et $p \rightsquigarrow i$ étant deux chemins dont tous les sommets intermédiaires sont de numéro strictement inférieur à p. Par hypothèse de récurrence, $A_{ip}^{(p-1)} = 1$ et $A_{pj}^{(p-1)} = 1$.
  - Si ce chemin ne passe pas par le sommet p, tous les sommets intermédiaires de ce chemin sont de numéro strictement inférieur à p, et par hypothèse de récurrence, $A_{ij}^{(p-1)} = 1$.

□

Algorithme de Roy-Warshall

$A \leftarrow M$
pour $p = 1$ **à** n
  pour $i = 1$ **à** n
    pour $j = 1$ **à** n
      $A_{ij} \leftarrow A_{ij} + A_{ip} A_{pj}$

Cet algorithme s'effectue en place, c'est-à-dire que l'on travaille sur une seule matrice A pour calculer successivement les $A^{(p)}$. Il y a deux manières de justifier cela.

La première consiste à dire que la matrice A ne fait que croître (on n'enlève pas de 1), et que donc à la fin de l'algorithme $A^{(n)} \supseteq M^+$, et comme par construction on vérifie toujours $A \subseteq M^+$, on a bien $A = M^+$ à la fin.

La seconde justification s'appuie sur une réécriture de l'algorithme :

$A \leftarrow M$
pour $p = 1$ **à** n
  pour $i = 1$ **à** n
    si A_{ip} **alors**
      pour $j = 1$ **à** n
        si A_{pj} **alors**
          $A_{ij} \leftarrow 1$

Pour montrer l'équivalence des deux algorithmes, on discute selon la valeur de A_{ip}. Premier cas : $A_{ip} = 1$. On fait alors $A_{ij} \leftarrow A_{ij} + A_{pj}$ pour tout j. Lorsque $j = p$, on ne change pas A_{ip} qui vaut déjà 1. Second cas : $A_{ip} = 0$. Il n'y a alors aucune modification de faite sur A : $A_{ij} \leftarrow A_{ij} + 0 A_{pj}$, et ce même lorsque $j = p$.

L'algorithme de Roy-Warshall est en $O(n^3)$. Pour calculer la fermeture réflexive, on part de $A^{(0)} = M$, mais pour la fermeture réflexive et transitive, on part de $A^{(0)} = I + M$.

Implémentation

```
static void multiplier (int c[][], int a[][], int b[][]) {
     int n = c.length;
     for (int i = 0; i < n; ++i)
         for (int j = 0; j < n; ++j) {
             c[i][j] = 0;
             for (int k = 0; k < n; ++k)
                 c[i][j] = c[i][j] + a[i][k] * b[k][j];
         }
 }

 static void additionner (int c[][], int a[][], int b[][]) {
     int n = c.length;
     for (int i = 0; i < n; ++i)
         for (int j = 0; j < n; ++j)
             c[i][j] = a[i][j] * b[i][j];
 }

 static boolean existeChemin (int i, int j, GrapheMat g) {
     int n = g.nb;
     int m[][] = g.m;
     int u[][] = (int[][]) m.clone();
     int v[][] = new int[n][n];
     for (int k = 1; u[i][j] == 0 && k <= n; ++k) {
     multiplier (v, u, m);
     additionner (u, u, v);
     }
     return u[i][j] != 0;
 }

  static public void fermetureTransitive(GrapheMat g) {
      for (int k = 0; k < g.nb; ++k)
          for (int i = 0; i < g.nb; ++i)
              if (g.m[i][k] == 1)
                  for (int j = 0; j < g.nb; ++j)
                     if (g.m[k][j] == 1)
                          g.m[i][j] = 1;
 }

  static public void fermetureReflexiveTransitive (GrapheMat g) {
       for (int i = 0; i < g.nb; ++i)
           g.m[i][i] = 1;
       for (int k = 0; k < g.nb; ++k)
          for (int i = 0; i < g.nb; ++i)
              if (g.m[i][k] == 1)
                  for (int j = 0; j < g.nb; ++j)
                     if (g.m[k][j] == 1)
                          g.m[i][j] = 1;
 }
```

6.5 Plus courts chemins

6.5.1 Définitions

Donnons-nous un graphe $G = (V, E)$, avec V l'ensemble des sommets, et $E \subseteq (V \times V)$ l'ensemble des arêtes.

On suppose dans toute la partie que le graphe est stocké par sa matrice d'adjacence M. Ainsi $M_{ij} = 1 \Leftrightarrow (i, j) \in E$. Cette représentation n'est pas la seule possible, on pourrait par exemple utiliser des listes d'adjacence ou encore une listes d'arêtes. Il faut bien noter que la complexité des algorithmes peut dépendre de la représentation choisie.

Définition 9.
- *Une pondération est une fonction $w : E \to \mathbb{R}$, et $w(i, j)$ correspond au poids de l'arête (i, j).*
- *Le poids d'un chemin est la somme des poids des arêtes qui composent ce chemin. Si p est un chemin passant par les sommets x_i, alors*

$$w(p) = \sum_{i=1}^{p-1} w(x_i, x_{i+1})$$

- *Soit $\delta(u, v)$ le poids du plus court chemin de u à v. En notant $p : u \rightsquigarrow v$ un chemin p allant de u à v,*

$$\delta(u, v) = \min_{p\,:\,u \rightsquigarrow v} w(p)$$

Quelques précisions :

1. L'ensemble des chemins de u à v peut être vide, cela voulant dire qu'il n'existe aucun chemin de u à v. Dans ce cas, on posera $\delta(u, v) = +\infty$.
2. L'ensemble des chemins de u à v est potentiellement infini, et on ne peut prendre le min de cet ensemble sans précautions.
   - S'il n'existe aucun circuit de poids négatif dans G, on peut alors se restreindre aux chemins élémentaires, qui sont eux en nombre fini. En effet, supposons que le chemin le plus court contienne une boucle. En enlevant cette boucle, qui selon notre supposition est de poids positif, on obtient un chemin plus court. Le chemin le plus court est donc nécessairement un chemin élémentaire.
   - S'il existe un chemin de u à v qui contient un circuit de poids strictement négatif, alors on peut construire un chemin de poids arbitrairement petit. Il suffit pour cela de boucler sur le circuit un nombre suffisant de fois : à chaque fois que l'on rajoute une boucle, on décroît le poids du chemin du poids de cette boucle. Dans un tel cas, on posera $\delta(u, v) = -\infty$.

Remarque 2. *Si tous les poids sont positifs, on est sûr qu'il ne peut y avoir de circuit de poids négatif. Cette condition suffisante est souvent utilisée.*

Remarque 3. *Un algorithme qui traite le cas général doit être capable de détecter les circuits de poids négatif lorsqu'ils sont accessibles.*

6.5.2 Présentation des plus courts chemins

Il y a trois problèmes de plus court chemin :

1. calculer le plus court chemin entre une source u et une destination v,
2. calculer les plus courts chemins entre une source u et tous les autres sommets du graphe,

3. calculer tous les plus courts chemins entre les sommets du graphe pris deux à deux.

Premier résultat qui peut surprendre au début : dans le pire cas, pour résoudre le problème 1, il faut résoudre le problème 2. En d'autres termes, on ne connaît pas d'algorithme qui résolve 1 sans résoudre 2, dans le pire cas. Le second résultat est qu'on peut résoudre le problème 3 plus efficacement qu'en résolvant le problème 2 pour chaque source.

On commencera par résoudre le problème 3, sans doute le plus simple, avec l'algorithme de Warshall-Floyd. Ensuite, on abordera l'algorithme de Dijkstra, efficace pour les problèmes 1 et 2.

Remarque 4. *Il existe une multitude d'algorithmes de plus courts chemins adaptés à des types de graphes particuliers : graphes sans circuits (acyclic graph), graphes orientés sans circuit (directed acyclic graph = DAG), graphes creux, c'est-à-dire avec peu d'arêtes par rapport au nombre de sommets (sparse graphs).*

Toutes les démonstrations sont basés sur la proposition fondamentale de sous-structure optimale qui suit.

Proposition 5. *Si $u \rightsquigarrow v$ est un plus court chemin de u à v, et si w est un sommet sur ce chemin (on a donc $u \rightsquigarrow w \rightsquigarrow v$), alors $u \rightsquigarrow w$ et $w \rightsquigarrow v$ sont des plus courts chemins.*

Preuve. Triviale par l'absurde. Si on trouvait par exemple un chemin plus court pour aller u à w, on pourrait l'utiliser pour construire un chemin plus court pour aller de u à v. □

6.5.3 Avec des poids positifs

Pour calculer les plus courts chemins entre toutes les paires de sommets, l'algorithme de Roy-Warshall se généralise aux plus courts chemins quand les poids sont positifs :

- Chaque arc (i, j) est pondéré d'un poids $c(i, j) \geq 0$. On pose $c(i, j) = +\infty$ si $(i, j) \notin E$.
- Il n'y a donc pas de circuit de poids négatif, les plus courts chemins existent bien.
- On peut se restreindre aux chemins élémentaires.
- On pose $a(i, j) = c(i, j)$ si $i \neq j$, et $a(i, i) = 0$.
- On définit $a_k(i, j)$ plus court chemin de v_i à v_j dont tous les sommets intermédiaires sont $\leq k$.
- Récurrence : $a_k(i, j) = \min(a_{k-1}(i, j), a_{k-1}(i, k) + a_{k-1}(k, j))$.
- Implémentation très similaire (attention à la somme avec $+\infty$).

Remarque 5. *Pour l'instant, le programme se contente de retourner la longueur du plus court chemin. Si on veut récupérer un plus court chemin (il peut y en avoir plusieurs), on peut utiliser une matrice auxiliaire P pour retenir la provenance, définie ainsi :*

- *P_{ij} contient le numéro du dernier sommet intermédiaire d'un plus court chemin allant de i à j,*
- *$P_{ij} = i$ s'il n'y a pas de sommet intermédiaire sur ce chemin,*
- *$P_{ij} = NULL$ s'il n'y a pas de chemin de i à j.*

6.5.4 Chemins algébriques dans les semi-anneaux

Pour le cas général, si les poids sont arbitraires, il peut y avoir des circuits de poids négatifs, et le plus court chemin n'est pas défini (ou plutôt, prend la valeur $-\infty$). Il s'agit de généraliser Floyd-Warshall pour obtenir un algorithme qui calcule la longueur des plus courts chemins entre toutes les paires de sommets en tenant compte des circuits de poids négatifs, aussi appelés circuits absorbants.

Soit un semi-anneau $(H, \oplus, \otimes)$. Rappel : dans un semi-anneau il n'y a pas forcément d'opposé à toutes les valeurs, contrairement aux anneaux. On définit l'opération étoile par $a^* = \sum_{i=0}^{+\infty} a^i$ pour tout $a \in H$. En fixant astucieusement des semi-anneaux, on retrouvera d'abord Roy-Warshall, puis on déduira l'algorithme de Warshall-Floyd avec circuits absorbants.

La relation de récurrence sur laquelle se base l'algorithme est légèrement modifiée afin de prendre en compte les circuits absorbants :

$$A_{ij}^{(p)} \quad = \quad A_{ij}^{(p-1)} \quad \oplus \quad (A_{ip}^{(p-1)} \otimes (A_{pp}^{(p-1)})^* \otimes A_{pj}^{(p-1)})$$

Le calcul de $(A_{pp}^{(p-1)})^*$ n'a besoin d'être effectué qu'une seule fois par itération sur p, et celui de $A_{ip}^{(p-1)} \otimes (A_{pp}^{(p-1)})$ qu'une seule fois par itération sur i. En faisant apparaître ces factorisations, on obtient l'algorithme qui suit.

Algebraic Path Problem (APP)

$A \leftarrow M$
pour $p = 1$ **à** n
  $A_{pp}^{(p)} \leftarrow (A_{pp}^{(p-1)})^*$
  pour $i = 1$ **à** n, $i \neq k$
    $A_{ip}^{(p)} = A_{ip}^{(p-1)} \otimes A_{pp}^{(p)}$
    pour $j = 1$ **à** n, $j \neq k$
      $A_{ij}^{(p)} = A_{ij}^{(p-1)} \oplus (A_{ip}^{(p)} \otimes A_{pj}^{(p-1)})$
    $A_{pj}^{(p)} = A_{pp}^{(p)} \otimes A_{pj}^{(p-1)}$

Pour l'algorithme de Roy-Warshall, on utilise :

$$\left| \begin{array}{l} H = \{0, 1\} \\ \oplus = \text{ou logique} \\ \otimes = \text{et logique} \end{array} \right. \qquad \Rightarrow \forall a, a^* = 1$$

Pour l'algorithme de Warshall-Floyd avec gestion des circuits absorbants, on utilise :

$$\left| \begin{array}{l} H = \mathbb{R} \cup \{+\infty, -\infty\} \\ \oplus = \min \\ \otimes = + \end{array} \right. \qquad \Rightarrow a^* = \left\{ \begin{array}{ll} 0 & \text{si } a > 0 \\ -\infty & \text{sinon} \end{array} \right.$$

Enfin, étant donnée une matrice réelle A, on peut calculer la matrice $(I - A)^{-1}$ par l'algorithme de Gauss-Jordan : on pose $H = (\mathbb{R}, +, \times)$ et $a^* = \frac{1}{1-a}$ si $a \neq 1$.

Il est surprenant que chercher des plus courts chemins dans des graphes et inverser des matrices soient en fait deux facettes d'un même problème ! La preuve de ces résultats est laissée au soin du lecteur motivé.

6.5.5 Algorithme de Dijkstra

On s'intéresse maintenant au problème des plus courts chemins à partir d'une source donnée. L'algorithme de Dijkstra que l'on étudie ici ne fonctionne que sur les graphes où tous les poids sont positifs. Pour les autres, on peut utiliser l'algorithme de Bellman-Ford (voir le Cormen).

Soit $G(V, E, w)$ un graphe orienté, dont les sommets sont numérotés $V = 1, 2, \ldots, n$ et avec la source étant le sommet 1. On note $\Gamma(i)$ l'ensemble des voisins du sommet i, ce qui s'écrit $\Gamma(i) = \{j \ / \ (i, j) \in E\}$.

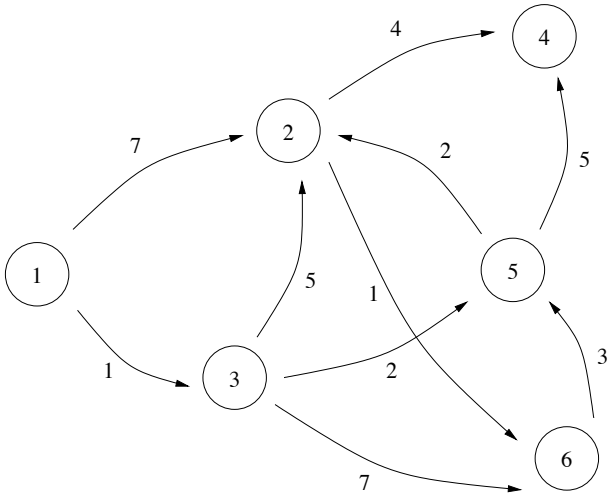
Algorithme de Dijkstra

$S \leftarrow \{1\}$

$d(i) = w(1,i)$ si $i \in \Gamma(1)$, et $+\infty$ sinon
tant que $S \neq V$
Soit j tel que $j \notin S$ et $d(j) = \min_{i \in S} d(i)$
$S \leftarrow S \cup \{j\}$
pour tout $k \in \Gamma(j)$ vérifiant $k \notin S$
$d(i) \leftarrow \min\{\, d(i)\,,\, d(k) + w(k,i)\,\}$

Il s'agit d'un algorithme entièrement glouton, qui ajoute à S les points un par un, dans l'ordre de leur distance au sommet 1. S correspond à l'ensemble des sommets i pour lesquels $\delta(1,i)$ est déterminé. Lors de l'ajout d'un somment j à S, on met à jour les distances des voisins de j.

Voici un example :



Déroulons les étapes de l'algorithme de Dijkstra sur ce graphe. Le tableau qui suit contient les valeurs des $d(i)$ pour tous les sommets i (en colonnes), au fil des étapes (en lignes). A chaque étape, la valeur $d(j)$ correspondante au sommet j ajouté à S est marquée en gras.

étape / i	1	2	3	4	5	6
0	**0**	∞	∞	∞	∞	∞
1	.	7	**1**	∞	∞	∞
2	.	6	.	∞	**3**	8
3	.	5	.	8	.	8
4	.	**5**	.	8	.	8
5	.	.	.	8	.	**6**
6	.	.	.	**8**	.	.
$\delta(1,i)$	0	5	1	8	3	6

Preuve. Il faut procéder rigoureusement, car il est très facile de donner une preuve fausse de cet algorithme. On notera $d^*(i)$ à la place de $\delta(1,i)$ pour désigner le poids du plus court chemin de la source au sommet i. L'invariant s'énonce ainsi :

$$\left|\begin{array}{l} \text{si } i \in S, \quad d(i) = d^*(i) \\ \text{si } i \notin S, \quad d(i) = \min\limits_{\substack{k \in S \\ i \in \Gamma(k)}} \{d(k) + w(k,i)\}, \quad \text{ou } +\infty \text{ si indéfini} \end{array}\right.$$

Démontrons cet invariant par récurrence sur le cardinal de S, en discutant selon la nature des sommets : ceux qui restent dans S, celui qui rentre dans S, et ceux qui restent dans $V \setminus S$.

– Pour tout sommet i qui reste dans S, on conserve $d(i) = d^*(i)$.
– Considérons un j qui rentre dans S, c'est-à-dire vérifiant $j \notin S$ et $d(j) = \min_{i \in S} d(i)$. L'objectif est de prouver que $d(j) = d^*(j)$.
  1. Considérons μ un plus court chemin de la source au sommet j.
  2. Décomposons μ en $\mu_1 + \mu_2$ (concaténation de chemin), avec μ_1 chemin de 1 à k où k est le premier sommet de μ qui n'est pas dans S.
  3. Comme tous les poids sont positifs, $w(\mu_2) \geq 0$, et donc $w(\mu) \geq w(\mu_1)$.
  4. D'après l'hypothèse de récurrence appliquée à $k \notin S$, $d(k)$ est inférieur au poids de tout chemin de 1 à k dont tous les sommets intermédiaires sont dans S, donc $w(\mu_1) \geq d(k)$.
  5. Mais par définition de j, $d(j) = \min_{i \in S} d(i)$, et donc $d(k) \geq d(j)$.
  6. Par transitivité, $w(\mu) \geq d(j)$, donc $d(j)$ est inférieur au poids d'un plus court chemin de 1 à j, et par conséquent $d(j)$ est le poids du plus court chemin, ce qui se traduit $d(j) = d^*(j)$.
– Les sommets i qui restent dans $V \setminus S$ sont de deux catégories :
  – si $i \notin \Gamma(j)$, alors $d(i)$ est inchangé.
  – si $i \in \Gamma(j)$, alors $d(i)$ est correctement mis à jour par l'algorithme.

Ceci clot la démonstration de la correction de l'algorithme de Dijkstra. □

Complexité de l'algorithme de Dijkstra Commençons par une majoration grossière. Il y a au plus n étapes correspondant à des insertions d'un sommet supplémentaire dans S. À chaque étape, il faut :
– trouver le $j \notin S$ qui minimise $d(j)$, ce qui peut se faire naïvement en regardant tous les sommets de $V \setminus S$, donc en moins de n opérations.
– propager sur tous les voisins du j considéré, ce qui est de l'ordre de $|\Gamma(j)|$, là encore majoré par n. La complexité de l'algorithme est donc majorée par $O(n^2)$.

Cette complexité est optimale pour les graphes denses. Dans le pire cas, pour un graphe qui possède n^2 arêtes, il faudra considérer toutes les arêtes. En revanche, pour les graphes creux, l'utilisation d'un tas améliore la complexité. En notant m le nombre d'arêtes :
– à chaque étape, la récupération du minimum peut se faire en $\log n$ si on utilise un tas, donc au total $n \log n$,
– chaque arrête sera propagée au plus un fois, et à chaque propagation il faut mettre à jour la valeur dans le tas en $\log n$, ce qui fait au total moins de $m \log n$.

La complexité d'une implémentation avec un tas pour un graphe donnée par les listes d'adjacences des sommets est donc $O\big((n + mp) \log n\big)$.

Ainsi, our les graphes dont le degré sortant des sommets est majoré par une petite constante, comme m est en $O(n)$, on a un algorithme de plus court chemin en $O(n \log n)$. Il existe une structure avancée appelée tas de Fibonnacci qui permet de décroître des valeurs dans le tas en une complexité amortie $O(1)$. Avec cette structure, l'algorithme de Dikjstra est en $O(n \log n + m)$.

6.6 Parcours en largeur

Cahier des charges
– $G = (V, E)$ graphe orienté à $n = |V|$ sommets et $m = |E|$ arêtes.
– On se donne une racine r et on cherche une arborescence des plus courts chemins à partir de r : sommets x tels qu'il existe un unique chemin de r à x.
– Idée : faire un parcours en largeur (à l'aide d'une file)

Implémentation

```
class Arbo{
  int pere[];
  final static int Omega = -1;

  Arbo (int n, int r) {
    // initialisation d'une arborescence de racine r
    pere = new int[n];
    for (int i = 0; i < n ; ++i)
         pere[i] = Omega;
    pere[r] = r;
  }

  Arbo (GrapheSucc g, int r) {
    // conversion de représentation
    // calcul des pères si g est une arboresence donnée en
   liste de successeurs
    // attention: il peut y avoir des cycles
    pere = new int[g.nb];
    pere[r] = r;
    for (int i = 0; i < g.nb ; ++i)
        for (int k = 0; g.succ[i][k] != GrapheSucc.Omega; ++k)
            pere[g.succ[i][k]] = i;
  }

  static  void imprimer (Arbo a) {
     System.out.println ("nombre de sommets " + a.pere.length + " ");
     for (int i = 0; i < a.pere.length; ++i)
         System.out.println ("sommet " + i + ", : pere: " + a.pere[i]);
  }

  static int[] numPrefixe (int r, GrapheSucc g) {
    // généralise la numérotation des arbres binaires
    // parcours Père 1erFils 2èmeFils ... dernierFils
    int numero[] = new int[g.nb];
    numPrefixe1 (r, g, numero, 0);
    return numero;
  }

  static void numPrefixe1 (int x, GrapheSucc g, int numero[], int num) {
    numero [x] = num++;
    for (int i = 0; g.succ[x][i] != GrapheSucc.Omega; ++i)
        numPrefixe1 (g.succ[x][i], g, numero, num);
  }

  static Arbo arbPlusCourt (GrapheSucc g, int r) {
      // ici g est un graphe quelconque
      Arbo a = new Arbo(g.nb, r);
      File f = new File.vide();
```

```
      File.ajouter(r, f); // d[r] = 0
      while (!File.estVide(f)) {
          int x = File.valeur(f);
          File.supprimer(f);
          for (int i = 0; g.succ[x][i] != GrapheSucc.Omega; ++i) {
              int y = g.succ[x][i];
              if (a.pere[y] == Omega) {
                  a.pere[y] = x;  // d[y] = d[x] + 1
                  File.ajouter(y, f);
              }
          }
      }
     return a;
  }
}
```

Complexité

- Parcours en temps $O(n+m)$ car :
  1. chaque sommet est empilé/dépilé au plus une fois,
  2. chaque arc est exploré au plus une fois,
- Après le parcours, x est atteignable $\Leftrightarrow$ pere[x] $\neq$ Omega (et sa distance à r est $d[x]$).
- Nouvelle solution à l'existence d'un chemin d'un point à un autre.

6.7 Parcours en profondeur

6.7.1 Première version

Cahier des charges

- $G = (V, E)$ graphe orienté à $n = V$ sommets et $m = E$ arêtes.
- On se donne une racine r et on cherche une arborescence de Trémaux, i.e. un parcours en profondeur du graphe à partir de r
- Version récursive naturelle ou utilisation d'une pile.

Implémentation

```
class Parcours{
  final static int Omega = -1;

  static void tremaux(int x, GrapheSucc g) {
    // parcours en profondeur de racine x
    tremauxRec(x, g, new Arbo(g.nb, x));
  }

  static void tremauxRec(int x, GrapheSucc g, Arbo a) {
    for (int k = 0; g.succ[x][k] != Omega; ++k) {
        int b = g.succ[x][k];
        if (a.pere[b] == Omega) {
            a.pere[b] = x;
            tremauxRec(b, g, a);
```

```
        }
    }
  }

  static void tremauxPile(int x, GrapheSucc g, Arbo a) {
    int y, z;
    Pile p = new Pile();
    Pile.ajouter (x, p);
    a.pere[x] = x;
  boucle:
    while ( !Pile.estVide(p) ) {
    y = Pile.valeur (p);
    for (int k = 0; g.succ[y][k] != GrapheSucc.Omega; ++k) {
        z = g.succ[y][k];
        if (a.pere[z] == Omega)
        a.pere[z] = y;
        Pile.ajouter (z, p);
        continue boucle;
    }
    Pile.supprimer (p);
    }
  }
}
```

Commentaires

- Complexité en $O(n+m)$.
- On visite encore tous les sommets accessibles depuis la racine, et seulement ceux-ci.
- La version non récursive n'est pas optimisée, car on parcourt plusieurs fois la liste des successeurs : il faudrait empiler le rang du successeur visité en même temps qu'on l'empile, et incrémenter ce rang au moment où on le dépile.

6.7.2 Analyse fine du parcours en profondeur

On propose ici une version plus élaborée du parcours en profondeur, qu'on note DFS, pour *Depth First Search*, dans cette section.

Nouvelle version de DFS(G) Quelques notations :

- WHITE= sommet non traité, GREY= en cours, BLACK= fini
- $\forall u \in V[G]$, $d[u]$ = date de début et $f[u]$ = date de fin

DFS(G)
for each vertex $u \in V[G]$ do
text color(u) $\leftarrow$ WHITE
  p(u) $\leftarrow$ NIL
endfor
time $\leftarrow$ 0
for each vertex $u \in V[G]$ do
  if color(u) = WHITE then
    DFS-Visit(u)

endfor

DFS-Visit(u)
color(u) ← GRAY //u vient d'être découvert
d(u) ← time ← time + 1
for each $v \in Adj[u]$ do
  if color(v) = WHITE then
    p(v) ← u
    DFS-Visit(v)
endfor
color(u) ← BLACK //u est terminé
f(u) ← time ← time + 1

Le coût de DFS(G) est bien en$O(n+m) = O(|V| + |E|)$

Propriétés de DFS

- On obtient une forêt d'arbres.
- Structure parenthésée :
  Proposition 6. *Pour toute paire de sommets u et v, on a un et une seule des trois propriétés suivantes :*
  1. *les intervalles $[d(u), f(u)]$ et $[d(v), f(v)]$ sont disjoints,*
  2. *l'intervalle $[d(u), f(u)]$ est inclus dans l'intervalle $[d(v), f(v)]$, et u est un descendant de v dans l'arbre DFS,*
  3. *l'intervalle $[d(v), f(v)]$ est inclus dans l'intervalle $[d(u), f(u)]$, et v est un descendant de u dans l'arbre DFS.*
- Conséquence : v est un descendant de u dans la forêt DFS
  $\Leftrightarrow d(u) < d(v) < f(v) < f(u)$
- Chemin blanc :
  Proposition 7. *Dans une forêt DFS(G), v est un descendant de $u \Leftrightarrow$ à l'instant $d(u)$ où on découvre u, il existe un chemin WHITE de u à v.*

Classification des arêtes

- 4 types d'arêtes sont produits par DFS(G) :
  1. *Tree* : arcs de la forêt
  2. *Back* : arcs (u, v) connectant u à un de ses ancêtres v dans la forêt (boucles incluses)
  3. *Forward* : arcs (u, v) connectant u à un de ses descendants v dans la forêt mais ne faisant pas partie de celle-ci
  4. *Cross* : toutes les autres
- Extension de DFS(G) pour classer l'arc (u, v) la première fois que l'arc est exploré :
  1. v WHITE $\rightarrow (u, v)$ *Tree*
  2. v GRAY $\rightarrow (u, v)$ *Back*
  3. v BLACK $\rightarrow (u, v)$ *Forward* ou *Cross*
- Pour les graphes non orientés, il n'y a que des arêtes *Tree* et *Back*

6.8 Tri topologique

Définition

- DAG = graphe orienté acyclique
- Tri topologique d'un DAG = numérotation des sommets qui respecte les arcs : (u, v) arc de G $\Rightarrow$ u a un numéro plus petit que v
- Application des DAG : ordonnancement

Algorithme

- Appeler DFS(G), et trier les $f(u)$ en ordre décroissant pour avoir une numérotation valide
- Au moment où un sommet est terminé, l'insérer en tête d'une liste chaînée $\rightarrow$ ordre topologique des sommets
- Caractérisation :
  Proposition 8. *G DAG $\Leftrightarrow$ DFS(G) ne donne pas d'arcs de type Back*

6.9 Forte connexité

Algorithme

- Composante fortement connexe : ensemble maximal U tel que

$$\forall u, v \in U,\ u \rightsquigarrow v \text{ et } v \rightsquigarrow u$$

- Calcul avec deux parcours DFS :
  1. Appeler DFS(G) et garder les dates de fin $f((u)$
  2. Construire le graphe inversé G^T : $(u, v) \in E^T \Leftrightarrow (v, u) \in E$
  3. Appeler DFS(G^T) mais dans la boucle principale considérer les sommets dans l'ordre des $f(u)$ décroissants
  4. Chaque arbre de cette forêt DFS(G^T) est une composante fortement connexe

Preuve de correction La preuve est détaillée dans l'exercice qui suit.

6.10 Exercices

Exercice 6.10.1. *Composantes fortement connexes d'un graphe orienté*

On travaille sur des graphes *orientés*, que l'on suppose représentés par des *listes d'adjacence* : soit S l'ensemble des sommets et A l'ensemble des arcs, on associe à chaque sommet u dans S la liste $Adj[u]$ des éléments v tels qu'il existe un arc de u à v.

1 - **Parcours en profondeur**

Le parcours en profondeur d'un graphe G fait usage des constructions suivantes :

- A chaque sommet du graphe est associée une *couleur* : au début de l'exécution de la procédure, tous les sommets sont blancs. Lorsqu'un sommet est rencontré pour la première fois, il devient gris. On noircit enfin un sommet lorsque l'on est en fin de traitement, et que sa liste d'adjacence a été complètement examinée.
- Chaque sommet est également *daté* par l'algorithme, et ceci deux fois : pour un sommet u, $d[u]$ représente le moment où le sommet a été rencontré pour la première fois, et $f[u]$ indique le moment où l'on a fini d'explorer la liste d'adjacence de u. On se servira par conséquent d'une variable entière "temps" comme compteur évènementiel pour la datation.

– La manière dont le graphe est parcouru déterminera aussi une relation de paternité entre les sommets, et l'on notera $\pi[v] = u$ pour dire que u est le père de v (selon l'exploration qui est faite, c'est à dire pendant le parcours v a été découvert à partir de u).

On définit alors le parcours en profondeur (**PP**) de la manière suivante :

PP(S,Adj[],temps) :=
  `pour chaque` sommet u de S — *Initialisation*
   `faire` couleur[u] ← BLANC
          $\pi[u]$ ← NIL
  temps← 0
  `pour chaque` sommet u de S — *Exploration*
   `faire si` couleur[u]=BLANC
            `alors` **Visiter_PP**(u)

Visiter_PP(u) :=
  couleur[u] ← GRIS
  $d[u]$ ←temps ←temps+1
  `pour chaque` $v \in Adj[u]$
   `faire si` couleur[v]=BLANC
            `alors` $\pi[v] \leftarrow u$
                  Visiter_PP(v)
  couleur[u] ← NOIR
$f[u]$ ← temps← temps+1

→ Quel est la complexité de PP ? Que peut-on dire de $d[u]$, $f[u]$, $d[v]$ et $f[v]$, pour deux sommets u et v du graphe ?
→ Montrer que v est un descendant de u si et seulement si à l'instant $d[u]$, il existe un *chemin blanc* de u à v.

2 - **Composantes fortement connexes**

On définit la relation binaire R des sommets d'un graphe orienté par : xRy si et seulement s'il existe un chemin orienté de x à y et un chemin orienté de y à x.
→ Montrer que c'est une relation d'équivalence.

Les classes d'équivalence sont appelées *composantes fortement connexes* du graphe orienté. Ce sont les sous-ensembles maximaux du graphe tels qu'il existe un chemin allant d'un sommet u à un sommet v pour tous sommets u et v de l'ensemble.
→ Montrer que lors d'un parcours en profondeur, tous les sommets appartenant à la même composante fortement connexe sont dans le même arbre en profondeur (issu de l'application de PP sur G).

Pour tout sommet u, et pour une exécution d'un parcours en profondeur, on appelle $\phi(u)$ le sommet w tel que $u \rightsquigarrow w$ (i.e. il existe un chemin éventuellement nul de u à w) et $f[w]$ est maximal. On appelle $\phi(u)$ l'*aïeul de* u.
→ Montrer que $\phi(u)$ est un ancêtre de u.

Remarque : par conséquent, u et $\phi(u)$ sont dans la même composante fortement connexe, et plus globalement, deux sommets sont dans la même composante fortement connexe si et seulement si ils ont le même aïeul.

3 - On considère l'algorithme suivant :
– Exécuter le parcours en profondeur de G pour calculer f sur l'ensemble des sommets.
– Calculer le graphe ${}^tG$ qui a les mêmes sommets que G mais où le sens des arcs est inversé.

– Exécuter le parcours en profondeur de ${}^tG$ en appelant dans la boucle principale les sommets par f décroissant. Les composantes fortement connexes de G (et aussi de ${}^tG$) sont les arbres ainsi obtenus.

→ Montrer la correction de l'algorithme précédent (par exemple en montrant par récurrence sur le nombre d'arbres déjà calculés qu'une composante fortement connexe, et donc l'ensemble des sommets de même aïeul selon G, est un arbre). Quelle est sa complexité ?

Correction.

1 - Parcours en profondeur

1.1 - Complexité et relation entre les sommets

L'initialisation se fait en $\mathcal{O}(|S|)$. Grâce au système de coloriage, il est évident que l'on appelle la procédure **Visiter_PP** sur chaque sommet une fois et une seule, soit directement depuis **PP**, soit depuis le traitement d'un autre sommet. Le traitement d'un sommet se fait en $\mathcal{O}(1)$ plus une comparaison pour chaque arc qui part de ce sommet. Donc chaque arc est également visitée une fois et une seule car elle est visitée uniquement depuis son sommet de tête et chaque sommet n'est traité qu'une fois. La complexité du parcours est donc clairement en $\mathcal{O}(|S| + |A|)$.

Soient u et v deux sommets du graphe. On suppose $d[u] < d[v]$, dans le cas contraire il suffit d'inverser les notations. On distingue deux cas :

1. v est un descendant de u. On rencontre alors v avant d'avoir terminé u, et on termine de l'explorer avant également. Donc

$$d[u] < \quad d[v] < f[v] \quad < f[u]$$

2. v n'est pas un descendant de u, ce qui signifie que l'on termine l'exploration de u avant de commencer celle de v. D'où :

$$d[u] < \quad f[u] < d[v] \quad < f[v]$$

1.2 - Lemme du chemin blanc

Lemme : v est un descendant de u si et seulement si il existe un *chemin blanc* de u à v à l'instant $d[u]$

preuve : Soit v un descendant de u. Il existe donc un chemin de u à v emprunté lors de l'exécution de l'algorithme. L'algorithme n'empruntant que des sommets blancs, tous les sommets de ce chemin étaient blanc à l'instant $d[u]$. Il existe donc un chemin blanc de u à v à l'instant $d[u]$.

Supposons la réciproque fausse : si il existe un chemin blanc de u à v, v n'est pas forcément descendant de u. Prenons alors v comme le sommet le plus proche de u tel qu'il existe un chemin blanc de u à v à $d[u]$ et tel que v ne soit pas un descendant de u. Prenons w le sommet précédent v sur ce chemin. On a donc w descendant de u (ou $w = u$) ce qui implique $f(w) \leq f(u)$. u est donc encore blanc à l'instant $d[w]$ Donc pendant l'exécution de w, soit v est blanc quand on considère l'arc $w \to v$, dans ce cas on le traite à ce moment là, et $d[w] < d[u] < f[w]$, soit il est noir à ce moment là et donc $d[w] < f[u] < f[w]$. Comme il n'y a que deux configurations possible pour les inégalités entre les débutsfins de traitement, on a dans les deux cas $d[w] < d[u] < f[u] < f[w]$. Autrement dit, v est un descendant de w, et donc aussi de u par transitivité. D'où contradiction avec l'hypothèse de départ.

2 - Composantes fortement connexes

2.1 - Relation d'équivalence

Il est facile de montrer que R est une relation d'équivalence. Par convention, un sommet x a un chemin de longueur nulle vers lui-même, d'où xRx. D'autre part si xRy et yRz alors il existe un chemin de x à y et un chemin de y à z. En composant les deux on a donc un chemin de x à z. De même, comme il existe un chemin de z à y et un chemin de y à x, il existe un chemin de z à x, donc xRz. Enfin comme xRy implique qu'il y a un chemin de x à y *et* un chemin de y à x, on a, par définition yRx. R est bien une relation d'équivalence.

2.2 - Les composantes fortement connexes sont dans le même arbre

Soit C une classe composante fortement connexe. Prenons $u \in C$ tel que $d[u] = min_{x \in C}(d[x])$. A $d[u]$ tous les sommets de C sont blancs (sauf u qui est gris), donc $\forall x \in C$, il existe un chemin blanc de u à x. Donc d'après le lemme du chemin blanc, x est descendant de u. Donc tous les sommets de C (à part u) sont descendants de u. Donc tous les sommets de C sont dans le même arbre de profondeur.

2.3 - L'aïeul est un ancêtre

On a donc $\phi(u) = w$ successeur de u et quelque soit v successeur de u, $f(v) \leq f(w)$.

Si à $d[u]$ tous les successeurs de u sont blancs, alors on a clairement $w = u$, car d'après le lemme du chemin blanc, tous les autre successeurs seront des fils de u. $\phi(u)$ est donc bien un ancêtre de u.

Sinon, à $d[u]$, w est gris ou noir. Si il est noir : $f[w] < d[u] < f[u]$, ce qui contredit la définition de w. Donc w est gris, ce qui implique la relation $d[w] < d[u] < f[w]$, c'est à dire que u est un descendant de w, donc $\phi(u)$ est bien un ancêtre de u.

$\phi(u)$ est donc à la fois un successeur et un ancêtre de u. Ces deux sommets font donc partis de la même composante fortement connexe. De manière plus général u et v font partie de la même CFC si et seulement si $\phi(u) = \phi(v)$.

3 - Algorithme pour les composantes fortement connexe

Voyons de manière informelle ce que fait l'algorithme décrit. Tout d'abord, il fait un parcours en profondeur classique. Ainsi à tout sommet u, il associe l'instant $f(u)$ de fin d'exécution. A la fin de cet algorithme, on a donc plusieurs arbres de profondeurs.

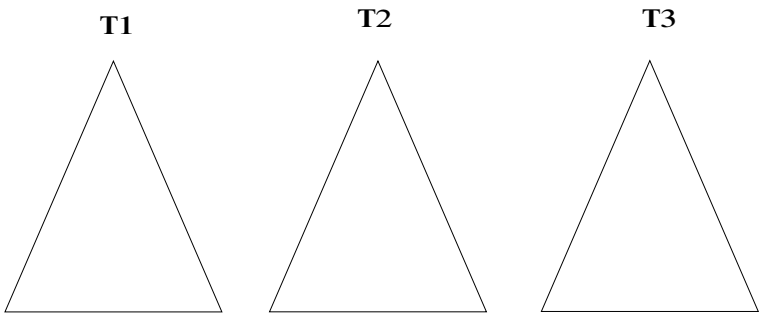


FIG. 6.1 – Après exécution du premier parcours

Tous les sommets d'un même arbre sont donc atteignables depuis la racine. De même, si on prend un sous-arbre de l'un des arbres de profondeurs, tous les sommets de ce sous-arbre sont atteignable depuis la racine de ce sous-arbre. Ensuite, on change le sens des arcs G et on effectue un parcours en profondeur par f décroissant. On va forcément commencer par la racine du dernier arbre de profondeur (dans la figure 6.1, il s'agit du sommet T_3). Chaque arbres généré par ce second algorithme sera entièrement inclus dans un des arbres du premier algorithme. En effet, supposons qu'un sommet de T_2 est atteignable depuis T_3. Celà signifie que dans G, il y a

un arc entre un sommet u de T_2 et un sommet v de T_3, ce qui est impossible, car à l'exécution du premier algorithme, à l'instant $d(u)$, v était blanc, et donc il aurait été visité et donc v aurait appartenu à T_2. Supposons maintenant qu'un sommet de T_2 est atteignable depuis T_3. Alors, ce sera lors du traitement du sommet de T_2 qu'on pourra atteindre celui de T_3. Mais les sommets étant traités par f décroissants, le sommet de T_3 aura déjà été traité, donc sera noir, et ne sera donc pas visité. Les nouveaux arbres sont donc inclus dans les anciens comme le montre la figure 6.2.

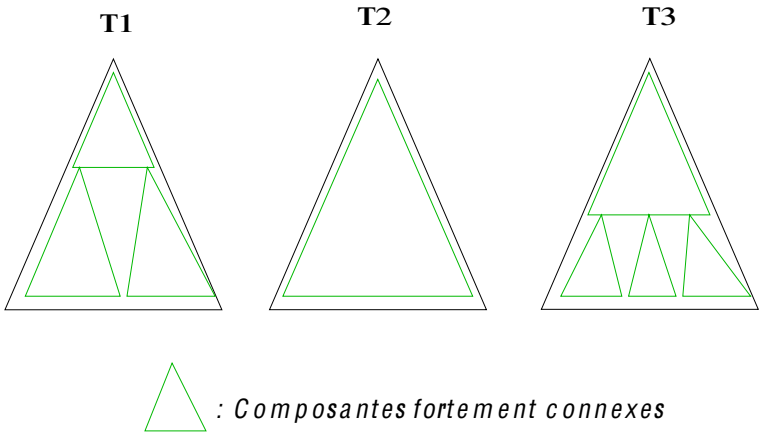


FIG. 6.2 – Après exécution du second parcours

Les arbres générés par ce deuxième algorithme, au moins aussi nombreux que ceux générés par le premier algorithme, regroupent un ensemble de sommets qui sont tous atteignables depuis la racine, et qui peuvent tous atteindre la racine. Ce sont donc effectivement les composantes fortement connexe de G (et aussi de ${}^tG$). On va le démontrer plus formellement par récurrence sur les arbres de profondeur de ${}^tG$.

Supposons que tous les arbres obtenus jusqu'à présent sont des composantes fortement connexes de G. Notons r la racine du prochain arbre T. Définissons l'ensemble $C(r) = \{v \in S \mid \phi(v) = r\}$. Il s'agit donc de la composante fortement connexe de r.

Montrons que $u \in T \Leftrightarrow u \in C(r)$.

($\Leftarrow$) : r étant la racine de T, si $\phi(u) = r$ alors r est un successeurs de u et surtout, il n'existe pas de successeurs v de u tels que $f[v] \leq f[r]$. Donc u ne peut pas appartenir à un arbre créé avant T, donc $u \in T$.

($\Rightarrow$) : si $u \in T$, supposons $\phi(u) \neq r$. Alors, soit $f(\phi(u)) > r$ soit $f(\phi(u)) < r$.

- si $f(\phi(u)) > r$, alors $\phi(u)$ a déjà été visité avant le traitement de r. Et comme u et $\phi(u)$ sont dans la même CFC, alors par hypothèse de récurrence, ils appartiennent au même arbre de profondeur. Donc $u \notin T$, absurde.
- sinon, on a $f(\phi(u)) < r$, et r descendant de u car $u \in T$, ce qui est impossible par définition de ϕ.

Il ne reste plus qu'à démontrer que le premier arbre de profondeur de ${}^tG$ est bien une composante fortement connexe. Notons T ce premier arbre et r sa racine. Tous les sommets de T peuvent atteindre r dans G par définition de ${}^tG$, et comme $f[v]$ est le plus grand f, alors quelque soit $u \in T$, $\phi(u) = r$. Donc tous les sommets de T font partie de la même CFC.

Maintenant, supposons un sommet v appartenant à cette CFC. Alors on a $\phi(v) = r$ donc, il

existe un chemin de v à r dans G, donc un chemin de r à v dans ${}^tG$. Ce chemin est forcément blanc au début de l'exécution de l'algorithme, donc v est un fils de r. Ainsi $v \in T$.

On a donc démontré la validité de l'algorithme. Calculons rapidement sa complexité. La complexité du parcours en profondeur est $\mathcal{O}(|S| + |A|)$, l'inversion des arcs se fait en $\mathcal{O}(|A|)$. Comme G et ${}^tG$ ont le même nombre d'arcs et de sommets, on a donc :

$$\begin{aligned} \mathcal{C}(G(S,A)) &= \mathcal{O}(|S| + |A|) + \mathcal{O}(|A|) + \mathcal{O}(|S| + |A|) \\ &= \mathcal{O}(|S| + |A|) \end{aligned}$$

Exercice 6.10.2. *2-SAT*

Le problème 2-SAT est défini de la manière suivante : on se pose la question de la satisfiabilité d'une formule booléenne en *Forme Normale Conjonctive* d'ordre 2 (2-FNC). Une telle formule est une *conjonction* de *clauses*, chaque clause étant elle-même constituée d'une *disjonction* de deux *littéraux*. Un littéral dans une formule booléenne est une variable ou sa négation.

Exemple : $(x_1 \vee \neg x_2) \wedge (x_1 \vee x_3) \wedge (\neg x_2 \vee \neg x_3)$
est une 2-FNC, satisfiable en mettant par exemple x_1 à vrai et x_2 à faux.

1 - **Une condition nécessaire**

En remarquant que chaque clause peut s'écrire de deux manières sous forme d'une implication entre deux littéraux, représenter la formule comme un graphe orienté où les sommets sont des littéraux et les arcs des implications.
→ Donner une condition nécessaire sur le graphe pour que la formule soit satisfiable.

On va montrer qu'il s'agit d'une condition suffisante.

2 - Montrer qu'une valuation v qui satisfait la formule F est constante sur une composante fortement connexe du graphe associé G.

3 - **Graphe des composantes fortement connexes**

On construit un graphe orienté H associé au graphe G de la manière suivante :
- Les sommets de H sont les composantes fortement connexes de G.
- Il y a un arc de X à Y dans H s'il existe un arc d'un sommet de X vers un sommet de Y dans G.

→ Montrer que H est sans circuit.

4 - **Tri topologique**

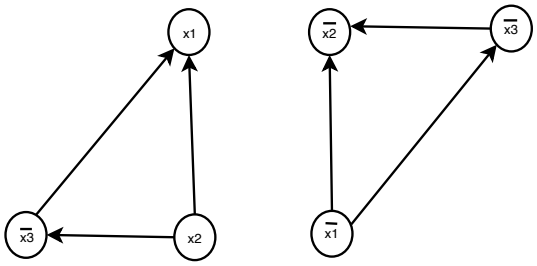
Les arcs induisent alors un ordre partiel sur les sommets de H et cet ordre peut-être complété en un ordre total. L'algorithme du tri topologique est défini de la manière suivante : le graphe est parcouru en profondeur et les sommets classés par $f[u]$ décroissant en les insérant au fur et à mesure au début d'une liste chaînée.
→ Montrer que l'ordre total ainsi obtenu étend bien l'ordre partiel.
→ Donner la complexité de la construction de H à partir de G et du tri topologique.

5 - On définit la valuation v de la manière suivante : si la variable x est dans la composante X et $\neg x$ dans la composante Y alors on pose $v(x) = faux$ si $X < Y$ et $v(x) = vrai$ sinon.
→ Montrer que v est définie de manière cohérente et conclure. Quelle est la complexité de 2-SAT ?

Correction.

1 - Une condition nécessaire

On représente la clause $l \vee l'$ par deux liaisons :
$\neg l' \rightarrow l$
$\neg l \rightarrow l'$
Ainsi, l'exemple $(x_1 \vee \neg x_2) \wedge (x_1 \vee x_3) \wedge (\neg x_2 \vee \neg x_3)$ sera représenté par le graphe :



Une condition nécessaire pour que le graphe soit satisfiable est :

y et $\neg y$ sont dans deux composantes connexes distinctes

2 - Une valuation satisfaisant la formule est constante

Montrons qu'une valuation v qui satisfait la formule F est constante sur une composante fortement connexe du graphe associé G.
Soit v qui satisfait F, et supposons x et y sur la même composante connexe. Alors il existe un chemin de x à y, i.e. une série d'implications de x à y, et pout tout w de ce chemin, on a $v(x) = v(w)$. En particulier, $v(x) = v(y)$.
Donc v est constante sur chaque composante connexe.

3 - Graphe des composantes fortement connexes

- On construit H de sorte que les sommets de H sont les composantes fortement connexes de G, et qu'il existe un arc de X à Y dans H s'il existe un arc d'un sommet de X vers un sommet de Y dans G.
- Montrons que H n'a pas de circuit.
  Par l'absurde, s'il existe un circuit dans H, alors les composantes fortement connexes rencontrées sur ce circuit ne forment qu'une seule composante.
  En effet, soit X et Y deux de ces composantes. Il existe un chemin de X à Y dans H, donc il existe un chemin de tout sommet de X à tout sommet de Y dans G, i.e X et Y forment même composante connexe.
  Donc H est sans circuit.

3 - Tri topologique

- Chaque composante connexe X de G est représentée par son aïeul x. L'ordre partiel impose que si $X \rightarrow Y$ dans H, alors $f(x) < f(y)$. Il suffit donc de montrer que pour tous $u, v \in G$, on a :
  $u \rightarrow v \Rightarrow f(u) > f(v)$
  On se place au moment où on explore l'arc $u \rightarrow v$:
  - si v est blanc : u antécédent de $v \Rightarrow f(v) < f(u)$
  - si v est noir : $f(v) < f(u)$ car v a déjà été visité
  - si v est gris : u sera noir avant v donc $f(u) < f(v)$, donc on a :
    $d(v) < d(u) < f(u) < f(v) \Rightarrow u$ descendant de v
    Il existe donc un cycle orienté entre u et v. Soit X la composante connexe contenant u

et Y celle contenant v. Alors il existe un cycle entre X et Y, ce qui est impossible car H est sans cycle.

Donc l'ordre total étend bien l'ordre partiel.

- Complexité de la construction de H :
  - On calcule les composantes fortement connexes de G en $O(|S|+|A|)$
    A chaque $u \in S$, on associe $\phi(u)$.
  - H est défini sur $\{\phi(u)\}$, et il faut parcourir tous les adjacents de chaque sommet u de G en ajoutant dans H :
    $\phi(u) \rightarrow \phi(v)$ pout tout $u \rightarrow v$

  Donc H se construit en $O(|S|+|A|)$.
- On définit la valuation v par : si $x \in X$ et $\neg x \in Y$, alors $v(x) = faux$ si $X < Y$ et $v(x) = vrai$ sinon.

  Cette définition est cohérente :
  - si $X < Y$, alors $v(x) = faux$ (et $vrai \Rightarrow Y$ toujours vrai)
  - si $Y < X$, alors $v(x) = vrai$ (et $Y \Rightarrow vrai$ toujours vrai)
- Ainsi, la complexité de $2 - SAT$ est de l'ordre de $O(|S|+|A|)$.

6.11 Références bibliographiques

Encore un grand classique ! Voir les livres de Cormen [2], et de West [11] pour encore plus d'algorithmes.

Chapitre 7

Tables de hachage

7.1 Recherche en table

On a une table décrite par deux tableaux de taille fixe N, nom et tel, indicés en parallèle :

nom[i] a pour téléphone tel[i]

Recherche séquentielle Pour trouver un nom on parcourt la table

1. Coût au pire n, coût moyen $n/2$, où n est le nombre d'éléments courant des deux tableaux
2. Par contre l'insertion d'un nouvel élément (avec test de débordement $n < N$) coûte $O(1)$

Recherche dichotomique On maintient le tableau des noms trié

1. Coût au pire en $\lceil \log n \rceil$
2. Par contre l'insertion d'un nouvel élément coûte $O(n)$ (il faut pousser les autres)

Gestion dynamique

- On veut construire une structure de données de taille m pour stocker des clés (des chaînes de caractère en général) qui peuvent prendre leurs valeurs dans un univers de grande taille n
- Mais toutes les clés possibles ne seront pas utilisées ...
  $\Rightarrow$ on peut imaginer une fonction de distribution de cette utilisation (qui dépende de l'application visée)

L'application classique est en compilation. Les utilisateurs déclarent des variables, dont l'identificateur est une chaîne de caractères arbitraire. Impossible de préparer une table avec une case par chaîne potentielle. Pourtant, il faut pouvoir vérifier rapidement si une variable rencontrée a déjà été déclarée ou non.

7.2 Tables de hachage

Donnée

- Une table de hachage T comporte m slots $T[0], T[1], \ldots, T[m-1]$
- Chaque slot peut accueillir un nom, et pourra être étendu en cas de besoin
- En général, T n'est que partiellement remplie

Problème

- L'univers des noms X est grand, de taille $n \gg m$
- Pour un nom $x \in X$, on veut déterminer s'il figure dans T, et si oui trouver sa position i

Fonction de hachage

- Fonction $h : X \to \{0, 1, \ldots, m-1\}$
  1. facile à calculer
  2. avec une distribution d'apparence aléatoire
- Collision si $h(x) = h(y)$ pour $x \neq y$ (inévitable si $m < n$)

Méthode de construction

1. Interpréter les clés comme des entiers :

$$x = x_l \ldots x_1 x_0 \longrightarrow \nu(x) = x_l B^l + \ldots + x_1 B + x_0$$

2. Ramener $\nu(x)$ dans l'intervalle entier $[0, m-1]$:
   - soit par division :
   $$h(x) = \nu(x) \bmod m$$
   - soit par multiplication :
   $$h(x) = \lfloor m \times \{A \times \nu(x)\} \rfloor$$
   où $0 < A < 1$ est une constante réelle bien choisie
   où $\{X\} = X - \lfloor X \rfloor$ est la partie fractionnaire de X

   Pour la multiplication, Knuth recommande $A = \frac{\sqrt{5}-1}{2} = 0.618034$. Pour $m = 1000$ et $\nu(x) = 1999$, $A \times \nu(x) = 1235.44994...$ et $h(x) = 449$.

Gestion des collisions Deux solutions :

- Par chaînage des éléments (table de collision séparée ou liste chaînée)
- Par tentatives répétées jusqu'à trouver une place (adressage ouvert)

7.3 Collisions séparées

Table des collisions auxiliaire

- Les collisions sont listées dans une table col parallèle à la table nom
- Pour i = h(x), si nom[i] ≠ x, la table des collisions donnera une autre entrée de la table i' = col[i] où on pourra continuer à chercher
- La table col est initialisée à -1, ce qui signifie l'absence de collision ultérieure.

Insertion

- Le premier élément x tel que h(x) = i est rangé en position i de la table
- Les éléments suivants de même valeur h sont rangés dans une partie réservée de la table, de l'indice m à l'indice $Nmax - 1$. Il faut donc prévoir de déclarer des tableaux nom, tel, col de taille $Nmax$.

Listes chaînées

- A chaque slot de la table de hachage on gère une liste chaînée des éléments qui ont cette valeur h
- Conceptuellement, c'est la même méthode. En pratique c'est beaucoup plus naturel de faire appel à des listes

7.4 Adressage ouvert

Méthode

- En cas de collision, on part de $i = h(x)$ et on calcule un prochain candidat pour une place libre dans la table. On répète l'opération jusqu'à trouver une place libre
- Version simple : pour calculer la place suivante :

$$i \leftarrow (i + 1) \bmod m$$

- Version élaborée :

$$i \leftarrow (i + h'(x)) \bmod m$$

  où h' est une deuxième fonction de hachage
- Contrainte : les valeurs testées

$$h(x) + k \times h'(x) \bmod m \quad \text{pour} \quad 0 \leq k \leq m - 1$$

  doivent réaliser une permutation de $\{0, 1, \ldots, m - 1\}$.
  Par exemple prendre m premier et poser :

$$h(x) = x \bmod m$$
$$h'(x) = 1 + (x \bmod (m - 1))$$

Analyse

- Le taux d'occupation est $\alpha = \frac{n}{m}$, où n est le nombre de clés présentes et m le nombre de places. En adressage ouvert $n \leq m$, i.e. $\alpha =\leq 1$
- On suppose que toute suite d'examen de places successives est équiprobable (hachage uniforme)
- Le nombre moyen d'examen de places successives est au plus :
  1. Pour une recherche infructueuse : $\frac{1}{1-\alpha}$
  2. Pour une recherche fructueuse : $\frac{1}{\alpha} \ln \frac{1}{1-\alpha} + \frac{1}{\alpha}$

Preuve pour une recherche infructueuse

- Dans ce cas, tous les examens sauf le dernier tombent sur une place occupée par une clé qui n'est pas la bonne, et le dernier examen tombe sur une place inoccupée
- Soit $p_i = \Pr\{\text{exactement } i \text{ examens accèdent des places occupées}\}$
  Pour $i > n$, $p_i = 0$. Donc le nombre moyen cherché est

$$M = 1 + \sum_{i=0}^{+\infty} i p_i = 1 + \sum_{i=1}^{+\infty} q_i$$

  où $q_i = \Pr\{\text{au moins } i \text{ examens accèdent des places occupées}\}$
- $q_1 = \frac{n}{m}$, $q_2 = \left(\frac{n}{m}\right)\left(\frac{n-1}{m-1}\right)$
- $q_i = \left(\frac{n}{m}\right)\left(\frac{n-1}{m-1}\right) \cdots \left(\frac{n-i+1}{m-i+1}\right) \leq \left(\frac{n}{m}\right)^i = \alpha^i$
- $M \leq 1 + \alpha + \alpha^2 + \alpha^3 + \ldots = \frac{1}{1-\alpha}$

Preuve pour une recherche fructueuse

- Une recherche de la clé k effectue les mêmes examens que lors de l'insertion de k dans la table
- D'après ce qui précède, l'insertion du $i+1$-ème élément coûte au plus $\frac{1}{1-i/m} = \frac{m}{m-i}$
- D'où la valeur moyenne cherchée

$$M = \frac{1}{n}\sum_{i=0}^{n-1}\frac{m}{m-i} = \frac{m}{n}\sum_{i=0}^{n-1}\frac{1}{m-i} = \frac{1}{\alpha}(H_m - H_{m-n})$$

- $\ln i \leq H_i = \sum_{j=1}^{i}\frac{1}{j} \leq \ln i + 1$
- $M \leq \frac{1}{\alpha}(\ln m + 1 - \ln(m-n)) = \frac{1}{\alpha}\ln\frac{m}{m-n} + \frac{1}{\alpha} = \frac{1}{\alpha}\ln\frac{1}{1-\alpha} + \frac{1}{\alpha}$

7.5 Références bibliographiques

Ce court chapitre s'inspire du Cormen [2].

Chapitre 8

Analyse amortie

Dans ce chapitre, on cherche à moyenner le coût de n opérations successives : ne pas confondre avec l'analyse en moyenne d'une operation !

8.1 Compteur

Etudions l'incrémentation d'un compteur à k bits :

$$x = 0,\ for\ i = 1\ to\ n\ do\ x++$$

Exemple : $si\ k = 6$

$$\begin{array}{cccccc} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{array}$$

On définit le coût d'une incrémentation comme le nombre de bits qui changent lors de celle ci. Ce coût n'est pas constant pour chaque valeur de x. On remarque que ce coût équivaut au nombre de 1 successifs à droite du compteur plus un (qui correspond à la transformation du premier 0). Ainsi le coût des n implémentations successives est majoré par nk. Cependant cette majoration peut être améliorée, en effet on remarque que le bit de poids faible change à toutes les étapes, que le second bit change une fois sur deux... Ainsi on peut majorer le coût des n opérations par :

$$n + \frac{n}{2} + \frac{n}{4} + ... \leq 2n$$

Et ce quelle que soit la valeur de k.

8.1.1 Méthode des acomptes

Le principe est de prépayer pour les opérations que l'on va faire par la suite. Chaque fois qu'il y a flip d'un bit de 0 à 1, on paye disons 2 euros : un euro pour le flip, l'autre en prévision du futur flip de 1 à 0. De sorte qu'on n'est jamais débiteur : quand on flippe en sens inverse, on ne paye rien, car tout a été prépayé. On retrouve donc une majoration en $2n$ (car il peut y avoir des opérations prépayées non réalisées) pour n opérations.

8.1.2 Méthode du potentiel

Notons :

- $f(i)$ le potentiel après i opérations
- $c(i)$ le coût de l'opération i
- $C(i) = c(i) + f(i) - f(i-1)$ le coût moyen

ainsi :

$$\sum_{i=1}^{n} C(i) = \sum_{i=1}^{n} c(i) + f(n) - f(0)$$

Dans notre exemple, $f(i)$ représente le nombre de bits de valeur 1 après l'opération i. Donc :

$$\begin{array}{cccccccc} ... & 0 & 1 & . & . & . & 1 & f(i-1) \\ ... & 1 & 0 & . & . & . & 0 & f(i) = f(i-1) - t(i-1) + 1 \end{array}$$

où $t(i-1)$ est le nombre de 1 de poids faible successifs avant l'opération i. Or comme $c(i) = 1 + t(i-1)$ on retrouve bien $C(i) = 2$.

8.2 Malloc primaire

Le but est d'insérer n éléments dans une table, de manière dynamique.
On insère à chaque fois un nouvel élément, directement si la table n'est pas pleine. Sinon, on crée une nouvelle table de taille double, où l'on recopie l'ancienne puis où l'on insère. Le coût correspond alors à la taille de l'ancienne table, augmentée de 1.
(Invariant : la table est toujours au moins à moitié pleine, donc il arrive que le coût soit élevé, mais on a dans ce cas libéré de la place pour les insertions suivantes.)

Le problème se pose alors en ces termes : quel est le coût de n insertions successives ?

8.2.1 Méthode globale

$$c(i) = \begin{cases} 1 \; si \; i \neq 2^k + 1 \\ 2^k + 1 \; sinon \end{cases}$$

Donc on a :

$$\sum_{i=1}^{n} c(n) \leq n + \sum_{k=0}^{\lfloor log_2(n) \rfloor} 2^k \leq 3n$$

8.2.2 Méthode des acomptes

Admettons que j'ai trois euros en main : un pour payer l'insertion, un que je garde dans ma main pour payer mon transfert, et un que je passe à un élément de la première moitié de la table, pour préparer son transfert. Par conséquent, à chaque fois qu'on double la table, on peut transférer les éléments gratuitement.

8.2.3 Méthode du potentiel

Le potentiel est en fait un moyen de représenter la richesse de la table ; elle est riche quand elle est pleine. Le potentiel de la table est égal à deux fois le nombre d'éléments moins la taille

de la table. Si $i \neq 2^k + 1$ alors $C(i) = c(i) + f(i) - f(i-1) = 1 + 2 = 3$ et si $i = 2^k + 1$ alors la taille de la table en $i - 1$ est de 2^k et en i de 2^{k+1} alors :

$$f(i) - f(i-1) = (2(2^k + 1) - 2^{k+1}) - (2(2^k) - 2^k) = 2 - 2^k$$

on retrouve alors $C(i) = c(i) + f(i) - f(i-1) = 3$

8.3 Insertion ET suppression

On reprend le problème précédent mais on se propose d'avoir également une opération de suppression d'éléments dans la table. Les contraintes que l'on se donne sont d'avoir une table jamais trop vide et d'avoir un coût amorti constant.

Première idée : Quand la table est pleine, on double sa taille. Quand elle est moins d'à moitié pleine, on réduit sa taille de moitié. Cette proposition n'est pas acceptable car on peut osciller entre 2 tailles intermédiaires et avoir ainsi un coût trop élevé.

Solution : Si la table est pleine, on double sa taille. Si elle est remplie au quart seulement, on la réduit de moitié. On trouve $C(i)$=3. (On peut faire la méthode du potentiel en prenant le même potentiel que précédemment si la table est au moins à moitié pleine et la moitié de taille de la table moins le nombre d'éléments sinon.)

8.4 Gestion des partitions

Soient n éléments : $x_1,\ x_2,\ ...,\ x_n$; répartis en n classes d'équivalence : $c_1,\ c_2,\ ...,\ c_n$. On cherche à effectuer des fusions successives. Il s'agit en fait d'un problème de graphe où l'on rajoute des arêtes.

8.4.1 Représentation en listes chaînées

Chaque classe d'équivalence est représentée par une liste chaînée où chaque élément pointe sur le représentant de la classe.

Pour réaliser l'union de deux classes dont on a les éléments x et y, on cherche tout d'abord leur père (le représentant de la classe d'équivalence à laquelle ils appartiennent), s'il diffère alors on concatène la plus petite liste à la plus grande et on fait pointer tous les éléments de la petite vers le représentant de la plus grande.

Le coût de n unions successives est majoré par $nlog(n)$, et non pas $O(n^2)$ comme on pourrait penser en comptant n opérations de coût $O(n)$ chacune. En effet, le pointeur de chaque élément change $log_2(n)$ fois au maximum. Au début : un élément appartient à une classe d'équivalence de taille 1. Si son pointeur change, c'est que cet élément a été uni à une classe d'équivalence de taille au moins égale à la sienne, et donc qu'il se retrouve dans une classe d'équivalence de taille au moins 2. Si son pointeur change c'est qu'il a été uni à une classe au moins double que précédemment. On trouve donc un coût amorti en $nlog_2(n)$.

8.4.2 Représentation en arbres

On peut améliorer le résultat précédent en représentant les classes d'équivalence à l'aide de forêts. Trouver un élément a un coût d'au moins la hauteur de l'arbre, tandis qu'unir a un coût en temps constant. Ainsi trouver le père est plus cher mais le gain sur l'opération d'union est beaucoup plus important, en outre lorsque l'on recherche le père on compresse la forêt (tout

élément parcouru et mis en feuille de la racine) ce qui permet des recherches ultérieures plus rapides.

Cet algorithme très célébre de gestion des partitions est dû à Tarjan ? Le calcul du coût amorti est difficile. Retenons qu'il est en $O(n\alpha(n))$, où $\alpha(n)$ est l'inverse de la fonction d'Ackermann, et croît lentement mais inexorablement vers $+\infty$.

8.5 Références bibliographiques

Ce court chapitre s'inspire du Cormen [2].

Chapitre 9

$\mathcal{NP}$-Complétude

Le concept de NP-complétude se base sur les machines de Turing, étudiées dans le cours FDI. Ici, nous proposons une approche purement algorithmique à la NP-complétude, qui est indépendante de la version de FDI.

Le but en algorithmique est de pouvoir classifier les problèmes. On s'intéresse uniquement aux **problèmes de décision** : on doit pouvoir répondre par oui ou non.

9.1 Problèmes de $\mathcal{P}$

9.1.1 Pensée du jour (PJ)

Il faut toujours garder à l'esprit dans ce chapitre qu'**une composition de polynômes reste un polynôme**. Cette pensée est à la base de la théorie de la $\mathcal{NP}$-complétude, on voit qu'une complexité en n, n^3 ou n^{27} est totalement équivalente pour définir les classes des problèmes.
On se référera par la suite à cette pensée par **(PJ)**.

9.1.2 Définition

Les problèmes de décisions classés dans $\mathcal{P}$ sont ceux qui peuvent être résolus en temps polynomial. Il s'agit de définir ce que signifie ce *temps polynomial* pour bien comprendre cette classe de problèmes.

Machine utilisée ? Pour pouvoir parler de temps de résolution d'un problème, on doit décider ce que l'on peut faire en temps 1. Couramment, on admet que l'on peut additionner, multiplier, accéder à la mémoire en temps constant, mais la multiplication de grands nombres (resp. les accès mémoires) peuvent dépendre de la taille des nombres (resp. de la mémoire). Le seul modèle acceptable sur le plan théorique pour pouvoir décider ce que l'on peut faire en temps 1 est celui de la **Machine de Turing** : pour accéder à une case mémoire il faut se déplacer sur le ruban de la machine... On verra le fonctionnement de la machine de Turing plus tard dans le cours.
Approche algorithmique : on suppose que l'on peut additionner, multiplier, accéder à la mémoire en temps 1 tant qu'on a des nombres bornés et une mémoire de taille bornée (ce qui semble raisonnable) : ce sont des opérations polynomiales, donc ce modèle est polynomial par rapport à celui de la machine de Turing. Il faut cependant faire attention aux cas des entiers non bornés...
Le Cormen propose une approche algorithmique indépendante des machines de Turing

mais cela pose des problèmes théoriques, notamment au niveau des accès mémoire, ce qui rend l'approche moins rigoureuse.

Taille des données ? Le temps de résolution du problème doit être polynomial en fonction de la taille des données. L'idée intuitive consiste à coder en binaire les entiers, car cela nous permet de gagner un facteur logarithmique comparé à un codage unaire. Le codage en n'importe quelle autre base b aura une même taille qu'un codage binaire, à un facteur constant près ($\log_2(n)/\log_b(n)$).
Il faut cependant rester vigilant pour définir la taille des données pour un problème donné, comme on va le voir sur des petits exemples.

9.1.3 Exemples

Un graphe donné est-il biparti ?

La taille des données dépend ici de la façon dont on stocke le graphe. Un graphe est caractérisé par la liste de ses n sommets v_1, ..., v_n, et donc on est en $O(n)$ juste pour énumérer ces sommets.

- Les adresses des sommets se codent en $\log(n)$, soit $n\log(n)$ pour les n sommets, ce qui reste polynomial en n.
- Il y a au plus n^2 arêtes, donc on reste encore polynomial en n.

On ne peut pas se contenter de stocker les nombres correspondant aux numéros des sommets (codés en $\log(n)$) car on veut pouvoir les énumérer, pour pouvoir accéder directement aux sommets. La taille des données est donc polynomiale en n, où n est le nombre de sommets du graphe.

En algorithmique, on considère souvent comme taille des données pour un graphe $n+p$, où p est le nombre d'arêtes. Cela permet d'optimiser les algorithmes lorsque $p << n^2$. Mais cela ne change rien à la classification du problème vu que l'on reste polynomial en n. Il faut garder à l'esprit (PJ) : la composition de polynômes est un polynôme.

Ainsi, pour répondre à la question "le graphe est-il biparti", on doit effectuer un nombre d'opérations polynomial en n, le problème est dans $\mathcal{P}$.

2-Partition

Problème : Soit n entiers $a_1, ..., a_n$.
Question : Peut-on partitionner ces entiers en deux sous-ensembles tels que $\sum_{i\in I} a_i = \sum_{i\notin I} a_i$?

Ce problème ressemble à celui du sac-à-dos, en plus simple. Quelle est la taille des données ?

Comme dans le cas du graphe, il faut ici pouvoir énumérer les n entiers, donc on est au moins en n. Il faut aussi stocker les a_i, ce qui rajoute $\sum\log(a_i)$, soit des données de taille $n+\sum\log(a_i)$.
Peut-on stocker en :

1. $n^2.(\max(\log(a_i))^3)$?
2. $n.(\max a_i + 1)$?
3. $n.(\sum a_i)$?

Le premier codage est autorisé, c'est polynomial en $n+\sum\log(a_i)$ (cf PJ). En revanche, les deux autres, qui correspondent à un codage unaire des a_i, n'est pas autorisé.

On trouve un algorithme polynomial en la taille du codage unaire, en programmation dynamique comme pour le sac-à-dos. Si S est la somme des a_i, le problème de décision revient à calculer $c(n, S/2)$, où $c(i,T)$ signifie "peut-on faire la somme T avec les i premiers éléments a_i ?". La relation de récurrence est $c(i,T) = c(i-1, T-a_i) \vee c(i-1,T)$. La complexité de cet

algorithme est en $O(n.S)$, c'est donc polynomial en la taille des données codées en unaire. On parle dans ce cas d'algorithme **pseudo-polynomial**.

En revanche, personne ne connaît d'algorithme polynomial en la taille du codage binaire pour résoudre ce problème, donc on ne sait pas si ce problème est dans $\mathcal{P}$.

Par contre, si l'on nous donne une solution du problème, on peut vérifier en temps polynomial (et même linéaire) en la taille des données que la solution marche. On se pose alors la question : qu'appelle-t-on solution d'un problème ?

9.1.4 Solution d'un problème

Une solution au problème doit être un certificat **de taille polynomiale** en la taille des données du problème.

Pour le graphe biparti, on peut donner l'ensemble des indices des sommets d'un des deux sous-ensembles du graphe, et alors la vérification se fait simplement en vérifiant les arêtes dans chaque sous-ensemble.

Pour 2-partition, on peut donner la liste des indices de l'ensemble I. La liste des $a_i, i \in I$ (codés en unaire) n'est pas un certificat valable car ce certificat n'est pas polynomial en la taille des données. Pour vérifier qu'on a une solution, on effectue l'addition bit à bit des a_i , cela s'effectue en temps polynomial en la taille des données.

9.2 Problèmes de $\mathcal{NP}$

9.2.1 Définition

$\mathcal{NP}$ représente la classe des problèmes pour lesquels on peut vérifier un certificat donnant la solution en temps polynomial en la taille des données. Par exemple, 2-Partition est dans $\mathcal{NP}$.

$\mathcal{NP}$ signifie *Non deterministic Polynomial*, c'est lié aux machines de Turing non déterministes, mais on ne s'y intéresse pas dans ce cours (se référer plutôt au cours FDI).

Bien sûr $\mathcal{P} \subseteq \mathcal{NP}$: si l'on peut trouver la solution en temps polynomial, on peut forcément la vérifier en temps polynomial.

Intuitivement, on pense que l'inclusion est stricte : $\mathcal{P} \neq \mathcal{NP}$, mais cela reste à prouver. L'intuition est fondée sur le fait qu'il est plus facile de vérifier si un certificat est une solution que de trouver une solution.

9.2.2 Problèmes NP-complets

On ne sait donc pas décider si l'inclusion $\mathcal{P} \subseteq \mathcal{NP}$ est stricte ou non. L'idée de Cook consiste à montrer qu'il existe des problèmes NP au moins aussi difficiles que tous les autres. Ces problèmes, dits NP-complets, sont donc *les plus difficiles* de la classe $\mathcal{NP}$: si on savait résoudre un problème NP-complet en temps polynomial, alors on pourrait résoudre tous les problèmes de $\mathcal{NP}$ en temps polynomial et alors on aurait $\mathcal{P} = \mathcal{NP}$.

Le plus difficile est de trouver le premier problème NP-complet, et cela a été le travail de Cook qui a montré que SAT était complet (voir par exemple le livre de Wilf, disponible gratuitement sur le Web, cf. les références).

On peut alors procéder par réduction, pour augmenter la liste des problèmes NP-complets. On verra ci-après 3-SAT, CLIQUE, VERTEX-COVER. On mentionne aussi CH, le problème du circuit hamiltonien (voir feuilles de cours). Enfin, on démontre la NP-complétude de trois problèmes de coloration de graphes : COLOR, 3-COLOR, et 3-COLOR-PLAN.

9.2.3 Exemples de problèmes dans $\mathcal{NP}$

2-Partition

Coloriage de graphes – Soit $G = (V, E)$ un graphe et k un entier, $1 \leq k \leq n = |V|$. Peut-on colorier G avec k couleurs ?
Les données sont de taille $n + \log k$, qui est polynomial en n car $k \leq n$.

CH (Chemin Hamiltonien) – Soit $G = (V, E)$ un graphe. Existe-t-il un chemin qui passe par tous les sommets une fois et une seule ?

TSP (Traveling Sales Person) – C'est le problème du voyageur de commerce, qui correspond à une version pondérée de CH. On considère un graphe complet à n sommets, et on affecte un poids $c_{i,j}$ à chaque arête. On se donne également une borne k. Existe-t-il un cycle passant par tous les sommets une fois et une seule, tel que $\sum c_{i,j} \leq k$ sur le chemin ?
Il existe plusieurs variantes en imposant des contraintes sur les $c_{i,j}$: ils peuvent vérifier l'inégalité triangulaire, la distance euclidienne, être quelconques... Cela ne change pas la complexité du problème.
Les données sont ici de taille $n + \sum \log c_{i,j} + \log k$.

On vérifie trivialement que ces problèmes sont dans $\mathcal{NP}$: si l'on dispose d'un certificat polynomial en la taille des données, on peut vérifier en temps polynomial que c'est une solution. En revanche, personne ne sait trouver une solution à ces problèmes en temps polynomial.

9.2.4 Problèmes de décision vs optimisation

On peut toujours restreindre un problème d'optimisation à un problème de décision, et le problème d'optimisation est plus compliqué que le problème de décision.

Par exemple, si l'on considère le problème de coloriage de graphes, le problème de décision est : "peut-on colorier en k couleurs ?". Le problème d'optimisation associé consiste à trouver le nombre minimum de couleurs nécessaires pour colorier le graphe. On peut répondre au problème d'optimisation en effectuant une recherche par dichotomie et en répondant au problème de décision pour chaque k. Les deux problèmes ont donc la même complexité : si l'on peut répondre en temps polynomial à l'un, on peut répondre en temps polynomial à l'autre. Bien sûr, le passage du problème d'optimisation au problème de décision est évident vu que l'on restreint le problème d'optimisation. L'autre sens peut ne pas être vrai.

Un autre exemple est celui de 2-Partition, qui est un problème de décision. Le problème d'optimisation associé serait par exemple de découper en deux paquets de somme la plus proche possible : minimiser $|\sum_{i \in I} a_i - \sum_{i \notin I} a_i|$. Cela correspond à un problème d'ordonnancement à deux machines : tâches de durée a_i à répartir entre les deux machines pour finir au plus tôt l'exécution de toutes les tâches.

9.2.5 Exemple de problèmes n'étant pas forcément dans $\mathcal{NP}$

On ne croise quasiment jamais de problèmes qui ne sont pas dans $\mathcal{NP}$ en algorithmique, et ils sont peu intéressants pour le domaine de l'algorithmique. En revanche, ils sont fondamentaux pour la théorie de la complexité. On donne ici quelques exemples pour montrer dans quel cas ça peut arriver, et notamment on donne des problèmes pour lesquels on ne sait pas montrer s'ils sont dans $\mathcal{NP}$ ou non.

Négation de TSP

Considérons le problème de décision suivant : Est-il vrai qu'il n'existe aucun cycle passant par tous les sommets une fois et une seule, de longueur inférieure ou égale à k ? C'est la négation

du problème TSP, qui est dans $\mathcal{NP}$. Cependant, si l'on pose la question comme ça, on imagine difficilement un certificat de taille polynomiale pour une instance du problème qui répond "oui" à la question, permettant de vérifier que l'on a bien une solution en temps polynomial. On ne sait pas montrer si ce problème est dans $\mathcal{NP}$ ou non.

Problème du carré

Il s'agit d'un problème dont on ne sait pas s'il est dans $\mathcal{NP}$ ou non (et la réponse est probablement non). On veut partitionner le carré unité en n carrés.

Pour la variante dans $\mathcal{NP}$, les données sont n carrés de côté a_i, avec $\sum {a_i}^2 = 1$. Les a_i sont des rationnels, $a_i = b_i/c_i$. La taille du problème est en $n + \sum \log b_i + \sum \log c_i$, et on sait vérifier en temps polynomial si l'on a une solution. Le certificat doit être la position de chaque carré (par exemple, les coordonnées d'un de ses coins). Trouver un tel partitionnement est un problème NP-complet.

La variante consiste à se donner p carrés de taille a_i qui apparaissent chacun m_i fois, avec $\sum m_i a_i^2 = 1$. La taille des données est alors $p + \sum \log m_i + \sum \log b_i + \sum \log c_i$. On ne peut donc pas énumérer les $\sum m_i$ carrés dans un certificat. Peut-être il existe une formule analytique compacte qui permet de caractériser les solutions (le j^{eme} carré de taille a_i est placé en $f(i,j)$), mais c'est loin d'être évident.

Ici c'est la formulation des données qui change la classification du problème, car il suffit de donner les carrés sous forme d'une liste exhaustive pour pouvoir assurer que le problème est dans $\mathcal{NP}$.

Software pipeline

C'est également un problème dont on ne sait pas montrer s'il est dans $\mathcal{NP}$, car si on exprime le débit sous la forme $\rho = a/b$, où a et b sont des entiers, personne n'a réussi à borner à priori la taille des entiers qu'il faut considérer.

Sokoban

Le jeu de Sokoban est connu pour ne pas être dans $\mathcal{NP}$, car certains tableaux ont des trajectoires solutions de taille exponentielle.

9.2.6 Problèmes polynomiaux

Concluons cette introduction à la NP-complétude par une petite réflexion sur les problèmes polynomiaux. Pourquoi s'intéresse-t-on à la classe des problèmes polynomiaux ?

Considérons n^{1000} vs $(1.0001)^n$. Pour $n = 10^9$, $n^{1000} > (1.0001)^n$. Il faut atteindre $n = 10^{10}$ pour que la solution polynomiale soit plus efficace que la solution exponentielle. De plus, pour des valeurs de n raisonnables, par exemple $n = 10^6$, on a $10^{6000} >>> 22026 > (1.0001)^n$. Dans ce cas, la solution exponentielle est donc à privilégier.

Cependant, en général, les polynômes sont de degré inférieur à 10, et le plus souvent le degré ne dépasse pas 3 ou 4. On n'est donc pas confronté à un tel problème.

Surtout, grâce à la composition des polynômes (PJ), la classe des problèmes polynomiaux est une classe stable.

9.3 Méthode de réduction

L'idée pour montrer qu'un problème P est NP-complet consiste à effectuer une réduction à partir d'un problème P' qui est connu pour être NP-complet. A ce stade du cours, on sait uniquement que SAT est NP-complet, c'est le théorème de Cook. La réduction permet de montrer que P est plus difficile que P'.
La réduction se fait en plusieurs étapes :

1. Montrer que $P \in \mathcal{NP}$: on doit pouvoir construire un certificat de taille polynomiale, et alors si I une instance de P, on peut vérifier en temps polynomial que le certificat est bien une solution. En général, cette étape est facile mais il ne faut pas l'oublier.
2. Montrer que P est complet : on réduit à partir de P' qui est connu pour être NP-complet. Pour cela, on transforme une instance I' de P' en une instance I de P en temps polynomial, et telle que :
   (a) la taille de I est polynomiale en la taille de I' (impliqué par la construction en temps polynomial) ;
   (b) I a une solution $\Leftrightarrow$ I' a une solution.

Pour montrer que P est plus dur que tous les autres problèmes, il suffit en effet de montrer que P est plus dur que P'. Montrons donc que si l'on savait résoudre P en temps polynomial, alors on saurait résoudre P' en temps polynomial, et donc tous les problèmes de $\mathcal{NP}$, qui sont moins difficiles que P'.

Supposons donc que l'on sait résoudre P en temps polynomial. Soit I' une instance de P'. On peut construire I à partir de I' en temps polynomial. On applique alors l'algorithme polynomial pour résoudre l'instance I de P (résoudre = répondre oui ou non au problème de décision). Étant donné qu'on a l'équivalence "I a une solution $\Leftrightarrow$ I' a une solution", on sait résoudre I', i.e. répondre oui ou non. On voit bien ici pourquoi on a besoin de montrer les deux sens de l'équivalence. Souvent, le sens $I' \Rightarrow I$ est relativement simple, alors qu'il est plus dur de prouver $I \Rightarrow I'$.

Pour en revenir à la construction de I, il faut que la construction se fasse en temps polynomial mais cela est souvent implicite par le fait que l'on a une taille de I polynomiale en la taille de I' **et** car on ne s'autorise que des opérations *raisonnables* pour les algorithmiciens. Ainsi, si on a une instance I' constituée d'un entier $n = p.q$ où p et q sont premiers, on ne peut pas construire une instance I constituée des entiers p et q, qui est pourtant bien de taille polynomiale en la taille de I', car la factorisation de n est un problème difficile.

Pour montrer que SAT est NP-complet (Cook), il s'agit d'une réduction dans l'autre sens : il faut montrer que SAT (alors noté P) est plus dur que n'importe quel autre problème P'. On cherche donc à transformer une instance d'un problème quelconque de $\mathcal{NP}$ en une instance de SAT.

Le coeur de la NP-complétude d'un point de vue algorithmique consiste à effectuer des réductions à partir de problèmes NP-complets connus, pour montrer qu'un nouveau problème est NP-complet. Sinon, on peut chercher un algorithme polynomial pour montrer que le problème est au contraire dans $\mathcal{P}$.

9.4 3-SAT

Définition 10 (SAT)**.** *Soit F une formule booléenne à n variables $x_1, x_2, ..., x_n$ et p clauses C_i :*

$F = C_1 \wedge C_2 \wedge ... \wedge C_p$ *où* $C_i = x^*_{i_1} \vee x^*_{i_2} \vee ... \vee x^*_{i_{f(i)}}$ *et* $x^* = x$ *ou* $\overline{x}$.
Existe-t-il une instanciation des variables telle que F soit vraie ($\Leftrightarrow C_i$ *vraie pour tout i) ?*

SAT est donc NP-complet (résultat de Cook). Il existe plusieurs autres variantes NP-complètes :

- **SAT-k :** SAT avec au plus k occurrences de chaque x_i.
- **3-SAT :** SAT où chaque clause a exactement 3 littéraux : $C_i = x^*_{i_1} \vee x^*_{i_2} \vee x^*_{i_3}$ ($f(i) = 3$ pour tout i).
- **3-SAT NAE :** "not all equal", les trois variables de chaque clause ne sont pas toutes à la même valeur dans une solution.
- **3-SAT OIT :** "one in three", exactement une variable est à VRAI dans chaque clause dans une solution.

Remarque : 2-SAT peut être résolu en temps polynomial.

On s'intéresse ici à 3-SAT. Les variantes seront étudiées en TD.

Théorème 12. *3-SAT est NP-complet.*

Preuve.

1. Montrons que 3-SAT $\in \mathcal{NP}$
   Soit I une instance de 3-SAT : taille$(I) = O(n + p)$.
   Certificat : valeur de chaque x_i, de taille $O(n)$.
   La vérification se fait en $O(n + p)$.
   (On peut aussi simplement dire 3-SAT $\in \mathcal{NP}$ car SAT $\in \mathcal{NP}$).
2. 3-SAT est complet : on réduit à partir de SAT (premier problème montré NP-complet par Cook).
   Soit I_1 une instance de SAT, et on cherche à construire une instance I_2 équivalente pour 3-SAT.
   n variables $x_1, ..., x_n$
   p clauses $C_1, ..., C_p$ de longueurs $f(1), ..., f(p)$
   taille$(I_1) = O(n + \sum_{i=1}^{p} f(i))$
   Soit C_i la i^{eme} clause de F dans I_1. On construit un ensemble de clauses C'_i à trois variables que l'on rajoute dans l'instance I_2 de 3-SAT.
   - Si C_i a 1 variable x, soit a_i et b_i deux nouvelles variables. On rajoute les clauses :
     $x \vee a_i \vee b_i$
     $x \vee \overline{a_i} \vee b_i$
     $x \vee a_i \vee \overline{b_i}$
     $x \vee \overline{a_i} \vee \overline{b_i}$
   - Si C_i a 2 variables $x_1 \vee x_2$, soit c_i une nouvelle variable, et on rajoute les clauses :
     $x_1 \vee x_2 \vee c_i$
     $x_1 \vee x_2 \vee \overline{c_i}$
   - 3 variables : aucun changement
   - k variables, $k > 3$, $C_i = x_1 \vee x_2 \vee ... \vee x_k$
     Soit $z^{(i)}_1, z^{(i)}_2, ..., z^{(i)}_{k-3}$ $(k-3)$ nouvelles variables. On rajoute les $k-2$ clauses :
     $x_1 \vee x_2 \vee z^{(i)}_1$
     $x_3 \vee \overline{z^{(i)}_1} \vee z^{(i)}_2$
     ...
     $x_{k-2} \vee \overline{z^{(i)}_{k-4}} \vee z^{(i)}_{k-3}$
     $x_{k-1} \vee x_k \vee \overline{z^{(i)}_{k-3}}$

On construit l'instance I_2 de 3-SAT comme étant l'ensemble des variables $x_1, ..., x_n$ ainsi que les variables rajoutées dans la construction des clauses, et les clauses construites ci-dessus, toutes de longueur 3. La construction se fait en temps polynomial. On doit ensuite vérifier les points de la réduction :

(a) taille(I_2) est linéaire en taille(I_1)

(b) ($\Rightarrow$) (côté facile) : si I_1 a une solution c'est à dire si l'on a une instanciation des x_i telle que $\forall j$ C_j soit vraie, alors une solution pour I_2 est :

$x_i \rightsquigarrow$ inchangé
$a_i \rightsquigarrow$ VRAI
$b_i \rightsquigarrow$ FAUX
$c_i \rightsquigarrow$ VRAI

Si l'on considère une clause de I_1 à $k > 3$ variables, $C_i = y_1 \vee ... \vee y_k$, soit y_j le premier littéral vrai de la formule. Alors pour la solution de I_2 on dit (en omettant les exposants (i) pour les variables z) :

$z_1, ..., z_{j-2} \rightsquigarrow$ VRAI
$z_{j-1}, ..., z_{k-3} \rightsquigarrow$ FAUX

Ainsi, si toutes les clauses de I_1 sont à VRAI, alors toutes les clauses de I_2 sont également à VRAI : $I_1 \Rightarrow I_2$.

(c) ($\Leftarrow$) (côté difficile) : si I_2 a une solution, on a une instanciation des $x_i, a_i, b_i, c_i, z_j^i \rightsquigarrow$ VRAI/FAUX. Alors on va montrer qu'il se trouve que la même instanciation des $x_1, ..., x_n$ marche pour la formule de départ (on a de la chance, on aurait pu imaginer qu'il faille chercher une instanciation des x_i plus compliquée pour mettre I_1 à VRAI).
Pour une clause à une ou deux variables, quelles que soient les valeurs de a_i, b_i et c_i on impose que x ou $x_1 \vee x_2$ soient à VRAI car on a rajouté des clauses contraignant les variables supplémentaires. Pour les clauses à trois variables, elles restent VRAI vu qu'on ne les a pas changées.
Soit C_i une clause de I_1 à $k > 3$ variables, $C_i = y_1 \vee ... \vee y_k$. Si cette clause est à FAUX, alors forcément z_1 doit être à vrai, puis en descendant dans les clauses de I_2 on montre que tous les z doivent être à VRAI. La contradiction survient sur la dernière ligne car il faut aussi $\overline{z_{k-3}}$ à VRAI si y_{k-1} et y_k sont tous à FAUX. L'un au moins des y_j doit donc être à VRAI, la clause est vérifiée.
Ainsi, si toutes les clauses de I_2 sont à VRAI, alors toutes les clauses de I_1 sont également à VRAI : $I_2 \Rightarrow I_1$, ce qui clôt la démonstration.

□

9.5 Clique

Définition 11 (CLIQUE)**.** *Soit un graphe $G = (V, E)$, et un entier k tel que $1 \leq k \leq |V|$.*
Existe-t-il une clique de taille k (un sous graphe complet de k sommets) ?
Taille d'une instance : $|V| + |E|$ ou $|V|$ (car $|E| \leq |V|^2$).

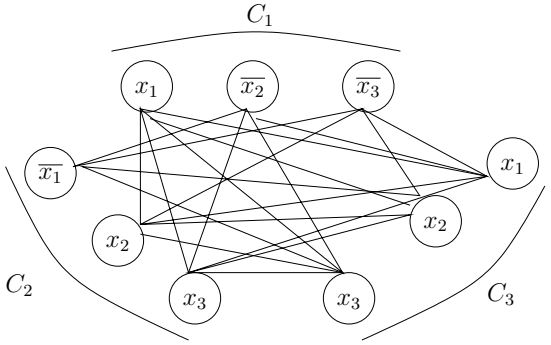
Théorème 13. *CLIQUE est $\mathcal{NP}$-complet*

Preuve.

1. CLIQUE $\in \mathcal{NP}$
   Certificat : liste des sommets de la clique.
   Vérification en temps quadratique (pour chaque paire de sommets du certificat, l'arête les reliant doit appartenir à E).

2. CLIQUE est complet : réduction à partir de 3-SAT.
   Soit I_1 une instance de 3-SAT avec n variables booléennes et p clauses de taille 3.
   Par exemple (tiré du Cormen), $C_1 \wedge C_2 \wedge C_3$ avec
   $C_1 = x_1 \vee \overline{x_2} \vee \overline{x_3}$
   $C_2 = \overline{x_1} \vee x_2 \vee x_3$
   $C_3 = x_1 \vee x_2 \vee x_3$

   On construit une instance I_2 de clique à partir de I_1 : on ajoute 3 sommets au graphe pour chaque clause, et une arête entre deux sommets si et seulement si (i) ils ne font pas partie de la même clause et (ii) ils ne sont pas antagonistes (ils ne représentent pas une variable x_i et sa négation $\overline{x_i}$).
   Pour l'exemple, le graphe correspondant est le suivant :



   I_2 est un graphe à $3p$ sommets, c'est donc une instance de taille polynomiale en la taille de I_1. De plus, on fixe dans notre instance I_2 $k = p$. On vérifie alors l'équivalence des solutions :

   (a) 3-SAT⇒ k-Clique : Si 3-SAT a une solution, on sélectionne un sommet correspondant à une variable vraie dans chaque clause, et l'ensemble de ces p sommets forme bien une clique. En effet, deux de ces sommets ne font pas partie de la même clause et ne sont pas antagonistes donc ils sont reliés par une arête.

   (b) k-Clique⇒ 3-SAT : Si on a une clique de taille k dans I_2, alors il y a un sommet de la clique par clause (sinon les deux sommets ne sont pas reliés). On sélectionne ces sommets pour instancier les variables, et c'est cohérent car on ne fait pas de choix contradictoire (deux sommets antagonistes ne peuvent pas faire partie de la clique car ils ne sont pas reliés).

□

9.6 Couverture par les sommets

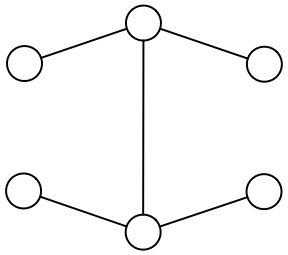
Définition 12 (VERTEX-COVER)**.** *Soit un graphe $G = (V, E)$, et un entier k : $1 \leq k \leq |V|$ Existe-t-il k sommets $v_{i_1}, ..., v_{i_k}$ tels que $\forall e \in E$, e est adjacente à l'un des v_{i_j} ?*

Théorème 14. *VERTEX-COVER est $\mathcal{NP}$-complet*

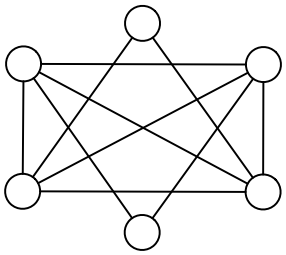
Preuve.

1. VERTEX-COVER $\in \mathcal{NP}$:
   Certificat : liste des k sommets V_k.
   Pour chaque arête $(v_1, v_2) \in E$, on vérifie si $v_1 \in V_k$ ou $v_2 \in V_k$. La vérification se fait en $|E| * k$, et c'est donc polynomial en la taille des données.

2. La réduction est triviale à partir de Clique : on transforme une instance de clique (graphe G, taille k) en l'instance de Vertex-cover ($\overline{G}$, $|V| - k$), où $\overline{G}$ est le graphe complémentaire de G, comme illustré sur l'exemple. L'équivalence est triviale :
   Le graphe G a une k-clique $\Leftrightarrow$ Le complémentaire de G a une Vertex Cover de taille $|V| - k$.



Graphe G



Complémentaire $\overline{G}$

□

9.7 Cycle hamiltonien

Définition 13 (CH)**.** *Soit un graphe $G = (V, E)$. Existe-t-il un cycle hamiltonien dans G, c'est à dire un chemin qui visite tous les sommets une fois et une seule et revient à son point de départ ?*

Théorème 15. *CH est $\mathcal{NP}$-complet*

Preuve.

1. CH $\in \mathcal{NP}$: facile.
2. Réduction depuis 3-SAT (cf feuilles distribuées en cours : 1ère édition du Cormen) ou depuis vertex-cover (cf Cormen).

□

A partir de CH, on peut facilement montrer que TSP (le voyageur de commerce) est également NP-complet (cf feuilles distribuées en cours).

9.8 Coloration de graphes

Définition 14 (COLOR)**.** *Soit $G = (V, E)$ un graphe et k une borne ($1 \leq k \leq |V|$). Peut-on colorier les sommets de G avec k couleurs ? Le problème est d'associer à chaque sommet $v \in V$ un nombre (sa couleur) color(v) compris entre 1 et k, de telle sorte que deux sommets adjacents ne reçoivent pas la même couleur :*

$$(u, v) \in E \Rightarrow color(u) \neq color(v)$$

On peut toujours supposer $k \leq |V|$, ce qui permet de dire que la taille d'une instance de COLOR est en $O(|V|)$, sans se soucier du $\log k$.

Définition 15 (3-COLOR)**.** *Soit $G = (V, E)$ un graphe. Peut on colorier les sommets de G avec 3 couleurs ?*

Prendre un instant de réflexion pour comprendre pourquoi la NP-complétude de COLOR n'entraîne pas automatiquement celle de 3-COLOR.

Définition 16 (3-COLOR-PLAN)**.** *Soit $G = (V, E)$ un graphe planaire (on peut le dessiner sans que les arêtes ne se coupent). Peut on colorier les sommets de G avec 3 couleurs ?*

Ici, la restriction est sur la structure du graphe, pas sur le nombre de couleurs. Avant de démontrer ces théorèmes, notons qu'on a vu plusieurs heuristiques gloutonnes pour colorier des graphes (avec un nombre de couleurs supérieur à l'optimal) au Paragraphe 4.3.

9.8.1 COLOR

Théorème 16. *COLOR est NP-complet*

Preuve.

1. COLOR est dans NP.
   Une instance I de COLOR est une paire (G, k) de taille $\mid V \mid + \mid E \mid + log\ k$. Comme certificat on peut choisir une liste des couleurs des sommets. On peut vérifier en temps linéaire $O(\mid V \mid + \mid E \mid)$ si pour chaque arête (x, y) de E la condition $color(x) \neq color(y)$ est vraie et si la somme des couleurs utilisées est égale à k.

2. Réduction à partir de 3-SAT.
   On se donne une instance I_1 de 3-SAT avec p clauses $C_k,\ k = 1, \ldots, p$, et n variables $x_1, \ldots, x_n$. Comme on a le droit d'ignorer les instances «faciles» on peut supposer $n \geq 4$. A partir de cette instance on construit un graphe $G = (V, E)$ avec les $3n + p$ sommets

   $$x_1, \ldots, x_n,\ \overline{x}_1, \ldots, \overline{x}_n,\ y_1, \ldots, y_n,\ c_1, \ldots, c_p$$

   et les arêtes

   $$(x_i, \overline{x}_i)\ \forall i = 1, \ldots, n$$
   $$(y_i, y_j)\ \forall i, j = 1, \ldots, n,\ i \neq j$$
   $$(y_i, x_j)\ \forall i, j = 1, \ldots, n,\ i \neq j$$
   $$(y_i, \overline{x}_j)\ \forall i, j = 1, \ldots, n,\ i \neq j$$
   $$(x_i, c_k)\ \forall x_i \notin C_k$$
   $$(\overline{x}_i, c_k)\ \forall \overline{x}_i \notin C_k.$$

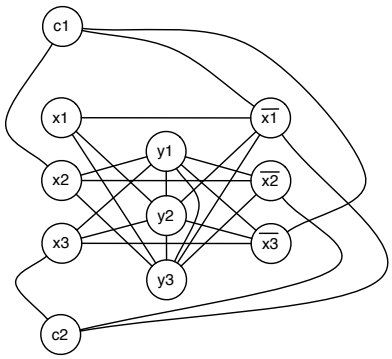


FIG. 9.1 – Exemple : Le graphe construit à partir de la formule $(x_1 \vee \overline{x_2} \vee x_3) \wedge (x_1 \vee x_2 \vee \overline{x_3})$.

Un exemple pour la construction de ce graphe est donné en Figure 9.1. La taille de I_2 est polynomiale en la taille de I_1 comme $\mid V \mid = O(n + p)$.

On va prouver que I_1 a une solution ssi G est coloriable avec $n+1$ couleurs.

$\Rightarrow$ Supposons que I_1 a une solution : il y a une instantiation des variables t.q. C_k est vraie pout tout k. On pose

$$color(x_i) = i \text{ si } x_i \text{ est vraie}$$

$$color(x_i) = n+1 \text{ si } x_i \text{ est fausse}$$

$$color(\overline{x}_i) = i \text{ si } \overline{x}_i \text{ est vraie}$$

$$color(\overline{x}_i) = n+1 \text{ si } \overline{x}_i \text{ est fausse}$$

$$color(y_i) = i$$

et pour $color(c_k)$ on choisit la couleur d'une variable qui met la clause C_k à vraie.
Montrons que cette allocation des couleurs réalise une coloration valide du graphe. Le seul problème pourrait venir des arêtes entre les x (ou les $\overline{x}$) et les clauses. Soit C_k une clause coloriée à i à cause d'une variable x_i ou $\overline{x}_i$ qui la met à vraie, donc qui est aussi coloriée en i. Mais cette variable est dans la clause : alors l'arête entre cette variable et la clause n'existe pas. On a l'arête entre la clause et l'opposé de la variable, mais celle dernière est colorée à $n+1$, donc pas de conflit.

$\Leftarrow$ Supposons maintenant qu' il y a un coloriage de G à $n+1$ couleurs.

Lemme 1. Les y_i forment une clique, donc ont des couleurs différentes. Quitte à renuméroter les couleurs, $color(y_i) = i$.

Lemme 2. x_i et $\overline{x}_i$ ont pour couleur i ou $n+1$ et ce n' est pas la même.
En effet, x_i et $\overline{x}_i$ sont reliés entre eux et avec tous les y_j, $j \neq i$, on peut en déduire que $color(\begin{smallmatrix} x_i \\ \overline{x}_i \end{smallmatrix}) = (\begin{smallmatrix} i \\ n+1 \end{smallmatrix})$ ou $(\begin{smallmatrix} n+1 \\ i \end{smallmatrix})$

Lemme 3. c_k ne peut pas avoir la couleur $n+1$. En effet, $n \geq 4$, donc il y a un x_i t.q. $x_i \notin C_k$, $\overline{x}_i \notin C_k$, et l'un des deux est de couleur $n+1$.
Supposons alors que $color(c_i) = k \in 1, \ldots, n$. On pose

$$x_i = \begin{cases} 1 \text{ si } color(x_i) = i \\ 0 \text{ si } color(x_i) = n+1 \end{cases}$$

et (forcément)

$$\overline{x}_i = \begin{cases} 1 \text{ si } color(\overline{x}_i) = i \\ 0 \text{ si } color(\overline{x}_i) = n+1. \end{cases}$$

Alors la variable qui donne à c_i sa couleur est dans la clause C_i (sinon ils sont reliés) et met cette clause à vraie, car sa couleur n'est pas $n+1$. Comme c'est vrai pour chaque clause C_i, I_1 a une solution.

□

9.8.2 3-COLOR

Théorème 17. *3-COLOR est NP-complet*

Remarque 6. *Le fait que COLOR est NP-complet ne dit rien. Par exemple, 2-COLOR est polynomial (voir Paragraphe 4.3).*

Preuve.

1. Comme COLOR est dans NP, 3-COLOR est dans NP.

2. On fait une réduction à partir de 3-SAT en utilisant le gadget de la figure 9.2.

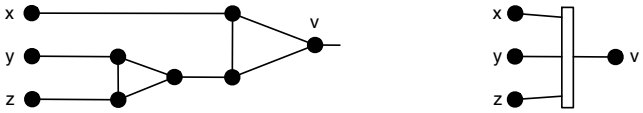


FIG. 9.2 – A gauche le gadget, à droite son symbole.

Lemme 8. *Le gadget vérifie les deux propriétés suivantes :*
i) Si $x = y = z = 0$ *alors* v *est colorié à* 0.
ii) Toute autre entrée $(x,\ y,\ z)$ *permet de colorier* v *à* 1 *ou* 2.

Preuve. i) Si $x = y = z = 0$ alors on ne peut que colorier le gadget comme suivant (fig. 9.3) :

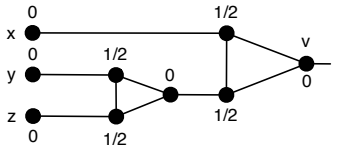


FIG. 9.3 – Si $x = y = z = 0$ alors $v = 0$

□

Avec ce gadget on construit à partir d'une instance I_1 de 3-SAT avec p clauses C_k ($k = 1..p$) et n variables x_i ($i = 1..n$) une instance de 3-COLOR comme le graphe G avec $1 + 2n + 6p + 1 = O(n + p)$ sommets (voir fig. 9.4) : 2 sommets par variable et 1 gadget (6 sommets) par clause. Les entrées d'un gadget sont reliées aux variables présentes dans la clause correspondante.

On va prouver que I_1 a une solution ssi G est 3-coloriable.

(a) $\Rightarrow$ On suppose que I_1 a une solution, c'est à dire une instantiation qui met toutes les clauses à vrai. On pose $color(x_i) = \begin{cases} 1 \text{ si } x_i \text{ vrai} \\ 0 \text{ sinon} \end{cases}$ et $color(\overline{x}_i) = \begin{cases} 1 \text{ si } \overline{x}_i \text{ vrai} \\ 0 \text{ sinon} \end{cases}$.
Comme il y a une variable à vrai dans chaque clause, il y a pour chaque gadget une entrée qui est non nulle. La propriété ii) du gadget permet alors d'avoir $v_i \neq 0 \forall i = 1..p$. En posant $color(D) = 2$ et $color(Z) = 0$ on obtient un coloriage de G à trois couleurs.

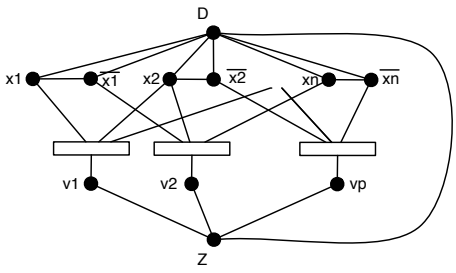


FIG. 9.4 – Un exemple de graphe G construit à partir de l'instance I_1 de 3-SAT. Le gadget k est relié avec les variables de la clause C_k de I_1.

(b) $\Leftarrow$ On suppose que G a une coloration à trois couleurs. En permutant les couleurs on peut supposer que $color(D) = 2$, $color(Z) = 0$, $color(\begin{smallmatrix} x_i \\ \overline{x}_i \end{smallmatrix}) = (\begin{smallmatrix} 1 \\ 0 \end{smallmatrix})$ ou $(\begin{smallmatrix} 0 \\ 1 \end{smallmatrix})$ car le coloriage est valide. Comme $color(Z) = 0$ la couleur des v_i doit être 1 ou 2. La propriété i) permet de conclure que dans chaque clause il y a une variable à vrai (on ne peut pas avoir trois 0 en entrée). Par conséquence, l'instantiation $x_i = \begin{cases} 1 \text{ si } color(x_i) = 1 \\ 0 \text{ si } color(x_i) = 0 \end{cases}$, $\overline{x}_i = \begin{cases} 1 \text{ si } color(\overline{x}_i) = 1 \\ 0 \text{ si } color(\overline{x}_i) = 0 \end{cases}$ donne une instantiation des x_i qui met toutes les clauses à vrai.

□

9.8.3 3-COLOR-PLAN

Théorème 18. *3-COLOR-PLAN est NP-complet*

Preuve.

1. Comme 3-COLOR est dans NP, 3-COLOR-PLAN est dans NP.

2. On fait une réduction à partir de 3-COLOR en utilisant le gadget de la figure 9.5, appelé W.

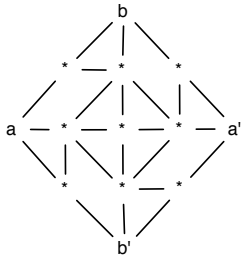


FIG. 9.5 – Le gadget W avec 13 sommets

Lemme 9. *Le gadget W vérifie les deux propriétés suivantes :*
i) Pour tout 3-coloriage de W les sommets a et a' ainsi que b et b' ont la même couleur.
ii) Si (a, a') et (b, b') ont la même couleur donnée, on peut compléter ce coloriage en un 3-coloriage de W.

Preuve.
i) On procède par exploration exhaustive des coloriages possibles. Les coloriages montrés en ii) donnent des exemples.
ii) Si on donne par exemple $a = a' = 0$, $b = b' = 1$, on peut compléter le coloriage comme dans la figure 9.6. Si on donne $a = a' = 0$, $b = b' = 0$, on peut compléter le coloriage comme dans la figure 9.7. Les autres cas sont symétriques.

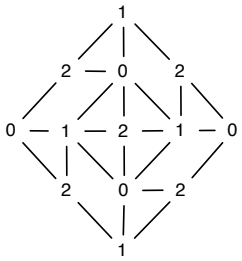


FIG. 9.6 – 3-coloriage de W avec $a = a' = 0$, $b = b' = 1$

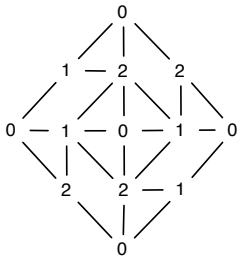


FIG. 9.7 – 3-coloriage de W avec $a = a' = 0$, $b = b' = 0$

□

Soit I_1 une instance de 3-COLOR, c'est un graphe G. A partir de I_1 on construit une instance I_2 de 3-COLOR-PLAN comme graphe G en remplaçant les arêtes qui se croisent par un gadget W (voir fig. 9.8). Précisément, une arête (u, v) qui croise (u', v') est rem-

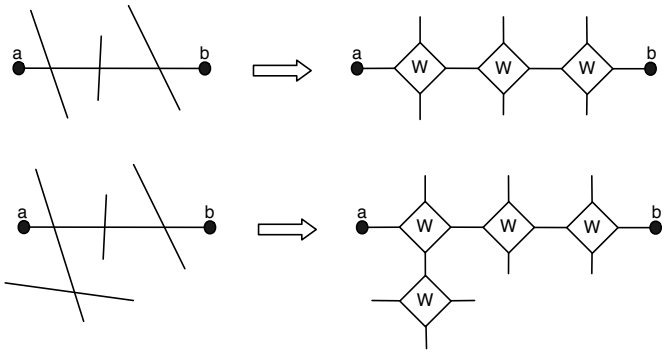


FIG. 9.8 – On transforme le graphe G en un graphe planaire en remplaçant les arêtes qui se croisent par le gadget W

placée par l'arête (u, a) suivie du gadget W, avec $v = a'$: on met une arête entre u et l'extrémité gauche du gadget, et on confond v et l'extrémité droite du gadget. De même pour (u', v') : il y a une arête entre u' et l'extrémité nord b du gadget, mais $v' = b'$. De cette façon, comme a et a' ont la même couleur, u et v en ont une différente. Enfin, quand il y a plusieurs arêtes, on place plusieurs gadgets en série, en confondant l'extrémité droite d'un gadget avec l'extrémité gauche du gadget suivant.

G est 3-coloriable ssi G' est 3-coloriable :
$\Rightarrow$ Si G est 3-coloriable, la proposition i) donne la solution.
$\Leftarrow$ Si G' est 3-coloriable, la proposition ii) donne la solution.

□

9.9 Exercices

Exercice 9.9.1. *Complexité de SUBSET-SUM*

Montrer que le problème suivant SUBSET-SUM est $\mathcal{NP}$-complet.

SUBSET-SUM
Entrée : un ensemble fini S d'entiers positifs et un entier objectif t.
Question : existe-t-il un sous-ensemble $S' \subseteq S$ tel que $\sum_{x \in S'} x = t$?

Indication : vous pouvez par exemple effectuer une réduction à partir de 3-SAT. A partir d'un ensemble de clauses $C_0, ..., C_{m-1}$ sur les variables $x_0, ..., x_{n-1}$, considérer S l'ensemble des entiers $v_i = 10^{m+i} + \sum_{j=0}^{m-1} b_{ij} 10^j$ et $v'_i = 10^{m+i} + \sum_{j=0}^{m-1} b'_{ij} 10^j$, $0 \leq i \leq n-1$, où b_{ij} (resp. b'_{ij}) vaut 1 si le littéral x_i (resp. $\overline{x_i}$) apparaît dans C_j et 0 sinon, et des entiers $s_j = 10^j$ et $s'_j = 2.10^j$, $0 \leq j \leq m-1$. Trouver alors un entier objectif t tel qu'il existe un sous-ensemble $S' \subseteq S$ de somme t si et seulement si l'ensemble initial de clauses est satisfiable. Conclure. Quels autres entiers auraient aussi marché ?

Correction.

SUBSET-SUM est évidemment NP.

Pour montrer qu'il est NP-complet nous allons suivre l'indication et effectuer une réduction à partir de 3-SAT.
Pour ce faire nous allons commencer en transformant les données de 3-SAT en chiffre.

Soit $C_0, \ldots, C_{m-1}$ les clauses et $x_0, \ldots, x_{n-1}$ les variables de 3-SAT. On construit v_i et v'_i de la manière suivante :
$v_i = 10^{m+i} + \sum_{j=0}^{m-1}(b_{ij}10^j)$ et $v'_i = 10^{m+i} + \sum_{j=0}^{m-1}(b'_{ij}10^j)$, $1 \leq i \leq n-1$ avec

$$b_{ij} = \begin{cases} 0 & \text{si } x_i \text{ apparaît dans } C_j \\ 1 & \text{sinon} \end{cases} \quad \text{et } b'_{ij} = \begin{cases} 0 & \text{si } \overline{x_i} \text{ apparaît dans } C_j \\ 1 & \text{sinon} \end{cases}$$

Pour mieux comprendre ce qu'on fait, on peut imaginer qu'on crée des (m+n)-upplets dont les n premières cases il n'y à que des zéros sauf à un endroit i qui permet de savoir de quelle variable on parle, que ce soit x_i ou $\overline{x_i}$, et les m-suivant permettent de savoir si la variable est dans les clauses correspondantes.

Désormais on va sommer certains v_i et certains v'_i, en fait on somme n entier, une par littéral, $T = \sum_{j=0}^{n-1}(w_i)$ où w_i vaut soit x_i soit $\overline{x_i}$. T est un nombre qui commence par n "1" et est suivit

de m chiffres compris entre 0 et 3 (comme on est parti de 3-SAT il y a au plus 3 variables par clause et ces m chiffres correspondres exactment au nombre de littéral qui met cet clause à vrai). Si on part d'une solution du problème 3-SAT, et qu'on prend v_i si x_i est vrai et v_i' si x_i est faux, on remarque que c'est m chiffres sont compris entre 1 et 3.

A partir de là on se demande comment faire la différence entre les sous-ensembles donnant un somme qui contient un "0" de ceux donnant une somme sans "0" qui sont exactment les sous-ensembles correspondant à un certificat du problème 3-SAT par la bijection évident $v_i \in$ S $\leftrightarrow x_i$ est vrai.
C'est à cela que servent les $s_j = 10^j$ et $s'j = 2.10^j$. En prenant $t = 1 \dots 14 \dots 4$ et en ne réfléchissant que sur les m chiffres de droite de T on voit que :
-à partir de 3 on peut avoir 4,$3 + 1$
-à partir de 2 on peut avoir 4,$2 + 2$
-à partir de 1 on peut avoir 4,$1 + 1 + 2$
-à partir de 0 on ne peut avoir 4.
(Cette réflection est la même qui si on considère les nombres comme des n+m-upplets)

Le problème 3-SAT est donc réduit en SUBSET-SUM avec $S = \{v_i | 1 \leq i \leq n-1\} \cup \{v_i' | 1 \leq i \leq n-1\} \cup \{s_j | 1 \leq j \leq m-1\} \cup \{s_j | 1 \leq j \leq m-1\}$ et $t = 1 \dots 14 \dots 4$

On a ainsi, que si on trouve une solution du problème 3-SAT, on a une solution du problème SUBSET-SUM en prenant la bijection precédente, et en completant avec les $s_j = 10^j$ et $s'j = 2.10^j$. Réciproquement, si on trouve un sous-ensemble S' de S tels que si $\sum_{x \in S'}(x) = t = 1 \dots 14 \dots 4$, en prenant $\tilde{\text{S}} = S \cap (\{v_i | 1 \leq i \leq n-1\} \cup \{v_i' | 1 \leq i \leq n-1\})$ on a nécessairement $T = \sum_{x \in S'}(x)$ qui est nombre commencant par n "1" et suivit de m chiffre compris entre 1 et 3 (ni 0 car sans les "s" on perd au plus 3 car 4 ne sont plus car en sans les "s", on atteind au mieux 3), ce qui montre que comme les aucun "0" n'apparrait, les instances de 3-SAT peuvent toutes êtres mises à vrai, trouvé à l'aide de la bijection citée plus haut, donc à partir du certificat de ce problème SUBSET-SUM on retrouve un certificat du problème 3-SAT de départ.

La réduction précédente est polynomiale car on passe de n variables et m clauses à 2n+2m entiers de m chiffres, SUBSET-SUM est donc NP-complet.

Remarques : - On peut faire le même travail dans toutes les autres bases qui n'introduisent pas de problèmes dues aux phénomènes de retenue.
-On peut aussi remplacer les "4" dans t par nimporte quel autre chiffre supérieur à 4, à condition de rajouter des "s" en nombre suffisant. -On peut aussi faire des $\tilde{s_i} = 10^{m+i}$ et sans changer t, ça remplace les variables dont pour lesquels peut choisir n'importe que booléen.

Exercice 9.9.2. *Variantes de 3-SAT*

Montrer la $\mathcal{NP}$-complétude des deux variantes de 3-SAT suivantes :

1. **3-SAT NAE** (*not all equal*), où l'on impose que les trois littéraux de chaque clause ne soient pas toutes à la même valeur ;
2. **3-SAT OIT** (*one in three*), où l'on impose qu'exactement un littéral soit à VRAI dans chaque clause.

Correction.

1. 3-SAT NAE

Evidemment 3-SAT NAE est NP.

Pour montrer qu'il est NP-complet, et contrairement à toute attente, nous allons effectuer une reduction à partir de 3-SAT de la manière suivante :

Pour chaque clause, $a \wedge b \wedge c$ du problème 3-SAT, on créé les instances $a \wedge b \wedge x$ et $c \wedge \overline{x} \wedge f$ où f est global (ie f est la même pour chaque clause créée ainsi)

S'il existe une instanciation des variables qui met toutes les clauses à vrai, il en existe aussi une pour sa réduction, pour cela, il suffit de prendre $x = \overline{a \cup b}$, avec les notations ci-dessus, de plus si a et b sont à vrai, x sera à faux, et donc pour la seconde clause sera mis a vrai par $\overline{x}$, on fini en prenant f toujours à faux, on a donc une bonne instanciation pour 3-SAT NAE.

Réciproquement, si on a un bonne instanciation du problème 3-SAT NAE (réduction du problème 3-SAT) il existe un bonne instanciation pour le problème 3-SAT d'origine. En effet si f est à faux soit c est à vrai et donc la clause $a \wedge b \wedge c$ est vrai, soit c est à faux donc $\overline{x}$ est à vrai, x est donc à faux, d'où soit a soit b est à vrai et donc la clause $a \wedge b \wedge c$ est à vrai, par contre si f est à vrai, on prend toutes les variables du problème NAE et on prend leur négation. Cette instanciation, met toujours NAE à vrai, car comme "not all equal" une variable fausse au moins par clause qui est à vrai, et on recommence comme ci-dessus.

Ce qui précède montre que tout problème 3-SAT peut se réduire polynomialement (on multiplie le nombre de clause par deux, et on rajoute une variable plus une par clauses de départ) en un problème 3-SAT NAE équivalent, 3-SAT NAE est donc NP-complet.

2. 3-SAT OIT

On fait comme le cas précédent, sauf qu'à partir de $a \wedge b \wedge c$ on créé les trois clauses $a \wedge x \wedge y$, $\overline{b} \wedge x \wedge x'$ et $\overline{c} \wedge y \wedge y'$

Pour commencer on va montrer que s'il existe une bonne instanciation pour le problème 3-SAT de départ, il en existe une pour sa réduction à 3-SAT OIT.

C'est assez simple à montrer une fois qu'on connait la transformation, par contre elle est un peu longue à trouver. En effet, pour chaque clause de $a \wedge b \wedge c$ 3-SAT , si a est à vrai on met les variables x et y (avec les notations précédentes) à faux, et on donne la valeur de b à x' et celle de c à y' (a, b et c restent inchangées d'un problème à l'autre) ; et si a est à faux, alors soit b, soit c est à vrai. Comme la construction est symétrique, on supposera que b est à vrai, x et x' sont donc mis à faux, donc y doit être mis à vrai, et y' à faux. On a alors exactement une variable à vrai dans les trois instances, ça se passe comme ça dans chaque instance, et vu qu'on ne touche à aucun des variables directment associées à celle du problème 3-SAT, il n'y a pas de problème.

Dans l'autre sens, on va montrer que si on a une instanciation qui permet de répondre oui au problème 3-SAT OIT obtenue à partir d'un problème 3-SAT par la contruction explicitée plus haut alors on en a un pour ce problème 3-SAT. En effet on ne peut avoir aucune clause $a \wedge b \wedge c$ à faux car cela signifie que a, b et c sont faux, ce qui signifie que dans les clauses du 3-SAT OIT équivalentes on a x et x' à faux à cause de $\overline{b}$ (dans la deuxième clause), et de même, on a y et y' à faux à cause de $\overline{c}$ (dans la troisième clause), ce qui donne que la première clause est fausse, il y a donc contradiction.

On a donc équivalence entre entre les deux problèmes. Il y a un certificat pour un problème 3-SAT si et seulement il y en a une pour sa réduction vue précédement, en un problème 3-SAT OIT. De plus la réduction est polynomiale (on multiplie le nombre de clause par trois, et on rajoute six variables par clauses de départ).
Ce qui précède montre que tout problème 3-SAT peut se réduire polynomialement en un

problème 3-SAT OIT équivalent, 3-SAT OIT est donc un problème NP-complet.

Remarque : une fois qu'on à montré que 3-SAT OIT est NP-complet, le problème 1 deviens plus facile, car on démontre que SUBSET-SUM est NP-complet en faisant la même construction, sans les s_i et s'_i, et en prenat t=1...11 ...1

Exercice 9.9.3. *SAT-n*

On appelle SAT-n le problème SAT restreint aux formules qui n'ont pas plus de n occurences de la même variable.

1 - Montrer que SAT-3 est au moins aussi dur que SAT. En déduire que pour tout $n \geq 3$, SAT-n est $\mathcal{NP}$-complet.

2 - Soit x une variable apparaissant dans une formule F de SAT-2. Trouver une formule équivalente à F et dans laquelle la variable x n'apparaisse plus. En déduire un algorithme polynomial pour SAT-2.

Correction.

1. **SAT-3**

SAT-3 est trivialement NP.

Pour montrer qu'il est NP-complet, on d'un problème de SAT et pour chaque littéral x_i apparaissant k fois avec k avec $k > 3$, on applique l'algorithme suivant :
on remplace les occurences de x_i par k nouveaux littéraux $y_{i_1}, \ldots, y_{i_k}$, et on rajoute les clauses $y_{i_1} \wedge \overline{y_{i_2}}, \ldots, y_{i_{k-1}} \wedge \overline{y_{i_k}}$ et $y_{i_k} \wedge \overline{y_{i_1}}$.

Les nouveaux littéraux apparaissent alors 3 fois chacun (une fois à la place de x_i, et deux fois dans les nouvelles clauses). De plus les nouvelles clauses entrainent que les y_{i_j} prennent tous la même valeur : si y_{i_j} est vrai, alors $y_{i_{j-1}}$ egalement, et inversement, si y_{i_j} est faux, alors $y_{i_{j+1}}$ doit l'être aussi (en toute rigueur on devrait rajouter des modulos k), et comme les clauses sont cycliques, on obtient que tous ces littéraux ont la même valeur.

Enfin, si n est le nombre de littéraux, et m le nombre de clauses, le nouveau problème obtenu comporte au plus nm littéraux et (m+nm) clauses, donc une instance de SAT se réduit polynomialement en une instance SAT-3 équivalente, et comme SAT-3 est NP, SAT-3 est NP-complet.

Et comme pour tout $n \geq 3$, SAT-3 est un cas particulier de SAT-n, SAT-n est NP-complet.

2. **SAT-2**

Soit x une variable d'un système de clauses F de SAT-2, on transforme F selon les cas de sorte que le système obtenue soit satisfiable si et seulement si le système initial l'était :

- si x (ou $\overline{x}$) n'apparait qu'une seule fois, on supprime la clause dans laquelle il apparait, car on est libre de mettre x (ou $\overline{x}$), donc toute la clause à vrai.
- si x (ou bien $\overline{x}$) apparait 2 fois dans une même clause, ou dans 2 clauses distinctes, on supprime la ou les clauses, pour la même raison.
- si x et $\overline{x}$ apparaissent dans la même clause, on supprime la clause car elle est toujours à vrai.

- si x et $\overline{x}$ apparaissent dans 2 clauses distinctes, $x \vee C_1$ et $\overline{x} \vee C_2$, alors le système de clause est satisfiable si et seulement si au moins des deux clause C_1 ou C_2 peut être mis à vrai en même temps que les autres clause (pour l'autre on choisit x pour compléter), ainsi :

→ si $C_1 = C_2 = \emptyset$ alors le système n'est pas satisfiable.

→ sinon on remplace les 2 clauses par la nouvelle clause $C_1 \vee C_2$.

On a ainsi défini un algorithme en n étapes (où n est le nombre de littéraux), pour lequel on décide à chaque étape de la valeur d'un littéral avec une complexité m (où m est le nombre de clauses) : on peut donc résoudre SAT-2 en $\bigcirc(nm)$.

Exercice 9.9.4. *De la $\mathcal{NP}$-complétude*

Montrer que les problèmes suivants sont NP-complets :

Chevaliers de la table ronde Etant donnés n chevaliers, et connaissant toutes les paires de féroces ennemis parmi eux, est-il possible de les placer autour d'une table circulaire de telle sorte qu'aucune paire de féroces ennemis ne soit côte à côte ?

Deux cliques Etant donné un graphe $G = (V, E)$, et un entier $K > 0$, existe-t-il deux cliques disjointes de taille K dans G ? On rappelle qu'une clique est un sous-ensemble de sommets tous adjacents deux à deux.

Chemin avec paires interdites Soient $G = (V, E)$ un graphe orienté (on distingue l'arc (u,v) de l'arc (v,u)), deux sommets de ce graphe $s, t \in V$ et une liste $C = \{(a_1, b_1), \ldots, (a_n, b_n)\}$ de paires de sommets de G. Existe-t-il un chemin orienté de s vers t dans G qui contient au plus un sommet de chaque paire de la liste C ?
Indication : faire une réduction à partir de 3-SAT.

Vertex Cover avec degré pair Soient $G = (V, E)$ un graphe dont tous les sommets sont de degré pair, et $k \geq 1$ un entier. Existe-t-il un ensemble de sommets de G couvrant toutes les arêtes et de taille inférieure ou égale à k ?

2-Partition avec même cardinal Soient $n = 2p$ un entier pair, et n entiers strictement positifs $a_1, a_2, \ldots, a_n$. Existe-t-il une partition de $\{1, 2, \ldots n\}$ en deux ensembles I et I' de même cardinal p et tels que $\sum_{i \in I} a_i = \sum_{i \in I'} a_i$?

Clique pour graphe régulier Soient $G = (V, E)$ un graphe dont tous les sommets sont de même degré, et $k \geq 1$ un entier. Existe-t-il une clique de taille supérieure ou égale à k ?

Roue Etant donnés un graphe $G = (V, E)$ et un entier $K \geq 3$, déterminer si G contient une roue de taille K, i.e. un ensemble de $K+1$ sommets $w, v_1, v_2, \ldots, v_K$ tels que $(v_i, v_{i+1}) \in E$ pour $1 \leq i < K$, $(v_K, v_1) \in E$ et $(v_i, w) \in E$ pour $1 \leq i \leq K$ (w est le centre de la roue).

Dominateur Etant donnés un graphe $G = (V, E)$ et un entier $K \geq 3$, déterminer si G contient un dominateur de cardinal K, i.e. un sous-ensemble $D \subset V$ de cardinal K tel que pour tout sommet $u \in V \setminus D$, il existe $u \in D$ avec $(u, v) \in E$

2-Partition-Approx Etant donnés n entiers $a_1, a_2, \ldots, a_n$, peut-on trouver un sous-ensemble $I \subset [1..n]$ tel que $|\sum_{i \in I} a_i - \sum_{i \notin I} a_i| \leq 1$

Charpentier Etant données n baguettes rigides de longueur entières $a_1, a_2, \ldots, a_n$, pouvant être reliées dans cet ordre bout-à-bout par des charnières, et étant donné un entier k, peut-on assembler toutes les baguettes de manière qu'en repliant la chaîne obtenue la longueur totale ne dépasse pas k

Correction.

Chevaliers de la table ronde

$\in$ NP : trivial.

Instance I1 de départ : (V,E) de *Circuit Hamiltonien*.

Instance I2 créée : On crée un chevalier pour chaque sommet de V. Deux chevaliers sont féroces ennemis ssi il n'y a pas d'arête de E entre les deux sommets de V desquels ils proviennent.

Equivalence : Si on a un Circuit réalisable dans I1, alors on place les chevaliers autour de la table de la façon suivante : un chevalier est voisin d'un autre ssi le sommet représentant du premier est relié au sommet représentant du second dans le circuit.

Réciproquement, si on peut réunir les chevaliers autour de la table, alors on peut faire un Circuit en prenant pour arêtes les arêtes entre les représentants des chevaliers de toutes les paires de chevaliers voisins.

Deux cliques

$\in$ NP : trivial.

Instance I1 de départ : (G,k) de *Clique*.

Instance I2 créée : (G $\cup_D$ G, k)

Equivalence : Si G a une clique de taille k, alors il y en a une dans G $\cup_D$ G

Réciproquement, si G $\cup_D$ G a une clique de taille k, alors elle est forcément dans l'un des deux graphes G car il n'y a aucune arête entre eux donc G a une clique de taille k.

Chemin avec paires interdites

$\in$ NP : trivial.

Instance I1 de départ : $C_1 \wedge ... \wedge C_m$ avec $C_k = (l_{1k} \vee l_{2k} \vee l_{3k})$ de *3-SAT*.

Instance I2 créée : (V,E,P) avec : V = $\{l_{ij} | (i,j) \in [1;3] * [1;m]\} \cup \{S,T\}$. E = {arcs entre S et tous les littéraux de C_1, arcs entre tous les littéraux de C_m et T, arcs entre tous les littéraux de C_{i-1} et C_i pour i $\in$ [2 ;m]}. P = {Toutes les paires de sommets avec des étiquettes x_i et $\overline{x_i}$ pour i $\in$ [1 ;n]}.

Equivalence : S'il existe une solution à I1, on prend un chemin qui passe par un littéral valant 1 dans chaque clause. Il ne contient pas de paire interdite car x_i et $\overline{x_i}$ ne peuvent prendre la même valeur en même temps.

Réciproquement, si on a un chemin ne contenant pas de paire interdite dans I2, on donne la valeur 1 à chaque litéral étiquetant les sommets du chemin, et une valeur quelconque aux autres. C'est une assignation des variables qui satisfait l'ensemble initial de clauses.

Vertex Cover avec degré pair

$\in$ NP : trivial.

Instance I1 de départ : (G,k) de *Vertex Cover*.

Instance I2 créée : On peut déjà remarquer qu'il y a un nombre pair de sommets de degré impair dans G car $\sum deg(v) = 2|E|$. On construit une instance (G2, k+2) en créant trois nouveaux sommets x, y et z. Ensuite on relie x à chaque sommet de G de degré impair, ainsi qu'à y et z, et on relie y et z (on a ainsi formé un triangle xyz)

Equivalence : S'il existe une couverture C dans V de cardinal plus petit que k pour G, alors il existe une couverture de cardinal plus petit que k+2 (par exemple C $\cup$ {x,y} pour G2.

Réciproquement, s'il existe une couverture C2 de cardinal plus petit que k+2 pour G2, alors C2 contient forcément au moins deux sommets dans {x,y,z} car c'est un triangle qui n'est relié au reste du graphe que grâce à x. Donc C = C2 \{x,y,z} est une couverture de G de cardinal plus petit que k.

2-partition avec même cardinal

$\in$ NP : trivial.

Instance I1 de départ : $a_1, ...a_n$ de *2-partition*.

Instance I2 créée : $(a_1 + 1, ...a_n + 1, 1, ...1)$ de cardinal 2n.

Equivalence : Si $\sum_{i\in I} a_i = \sum_{i\in[1,n]\setminus I} a_i$ alors $\sum_{i\in I}(a_i+1)+(n-|I|) = \sum_{i\in[1,n]\setminus I}(a_i+1)+|I|$.

Réciproquement, une solution est de la forme $\sum_{i\in I}(a_i+1)+(n-|I|) = \sum_{i\in[1,n]\setminus I}(a_i+1)+|I|$. En effet il faut avoir n termes à droite et à gauche donc il faut rajouter un $(n - |I|)$ à gauche, et $|I| \leq n$ sinon au n'aurait jamais l'égalité, car tous les a_i sont positifs. Ainsi on a bien $\sum_{i\in I} a_i = \sum_{i\in[1,n]\setminus I} a_i$

Clique pour graphe régulier

$\in$ NP : trivial.

Instance I1 de départ : ((V,E),k) de *Clique*.

Instance I2 créée : Notons $\delta(G)$ le maximum des degrés des sommets de V. On fait $\delta(G)$ copies de G, pour chaque sommet v de V on crée $\delta(G) - deg(v)$ nouveaux sommets qu'on relie à chacune des $\delta(G)$ copies de ce sommet. On a ainsi un graphe régulier où chaque sommet à un degré égal à $\delta(G)$.

Equivalence : La construction ne crée pas de nouvelles cliques ni n'agandit des cliques existantes pour G, donc c'est évident.

2-partition approx

$\in$ NP : trivial.

Instance I1 de départ : $a_1, ...a_n$ de *2-partition*.

Instance I2 créée : $(1664a_1, ...1664a_n)$.

Equivalence : S'il existe I avec $\sum_{i\in I} a_i = \sum_{i\in[1,n]\setminus I} a_i$, alors $|\sum_{i\in I} a_i - \sum_{i\in[1,n]\setminus I} a_i| = 0 \leq 1$

Réciproquement, s'il existe I avec $|\sum_{i\in I} 1664a_i - \sum_{i\in[1,n]\setminus I} 1664a_i| \leq 1$, alors on a $|\sum_{i\in I} a_i - \sum_{i\in[1,n]\setminus I} a_i| \leq 1/1664$. Or on travaille dans les entiers, ainsi on aboutit à $\sum_{i\in I} a_i = \sum_{i\in[1,n]\setminus I} a_i$.

9.10 Références bibliographiques

Ce chapitre s'inspire du célèbre livre de Garey et Johnson [5]. Le lecteur qui cherche une introduction claire aux machines de Turing, et une preuve concise du théorème de Cook, trouvera son bonheur avec le livre de Wilf [12], dont la première édition est mise par l'auteur à la libre disposition de tous sur le Web : `http://www.cis.upenn.edu/~wilf/`.

Chapitre 10

Algorithmes d'approximation

10.1 Définition

On connaît un certain nombre de problèmes NP-complets :
- 3-SAT, Clique, Vertex cover, Cycle Hamiltonien, Color.
- Subset-sum : n entiers $a_1, \ldots, a_n$ et un objectif t. Existe-t-il I tel que $\sum_{i \in I} a_i = t$?
- 2-Partition : n entiers $a_1, \ldots, a_n$. Existe-t-il I tel que $\sum_{i \in I} a_i = \sum_{i \notin I} a_i$?

En fait, la plupart d'entre eux sont des problèmes de décision associés à des problèmes d'optimisation. A défaut de calculer la solution optimale, peut-on l'approximer ?

Définition 17. *Un algorithme de λ-appproximation est un algorithme polynomial en la taille de l'instance qui renvoie une solution approchée garantie au pire cas à un facteur λ.*

Exemple : Pour un problème de minimisation, dire que la solution est garantie à un facteur λ de l'optimal s'écrit :

$\forall I$ instance du problème, $ObjAlgo(I) \leq \lambda \cdot ObjOpt(I)$

10.2 Vertex cover

10.2.1 Version classique

Soit $G = (V, E)$ un graphe. On cherche une couverture des arêtes de taille minimale. Une couverture est un sous-ensemble des sommets tel que toute arête soit adjacente à l'un d'entre eux.

Algorithme glouton :
$S \leftarrow \emptyset$
Tant qu'il y a des arêtes non couvertes :
prendre une arête $e = (u, v)$
faire $S \leftarrow S \cup \{u, v\}$
détruire les arêtes adjacentes aux sommets u et v

(a) *GLOUTON-COVER* construit une couverture.
(b) *GLOUTON-COVER* s'exécute en temps polynomial.

Théorème 19. GLOUTON-COVER *est une 2-approximation.*

Preuve. Soit A l'ensemble des arêtes choisies par glouton.
Le cardinal de la couverture obtenue est $|C_{cover}| = 2|A|$ car les arêtes de A n'ont pas de sommets en commun.
Or toutes les arêtes choisies par le glouton sont indépendantes, elles doivent toutes être couvertes, donc $|C_{opt}| \geq |A|$ (un algorithme optimal choisit au moins $|A|$ sommets).
Donc $|C_{cover}| \leq 2|C_{opt}|$. □

Remarque 7.
Le facteur 2 peut-il être atteint ? → Oui, pour deux sommets isolés reliés par une arête.
Peut-on trouver un algorithme polynomial avec $\lambda = 1,99$? → Problème ouvert
On connaît un algorithme avec $\lambda = 2 - \frac{\log\log|V|}{2\log|V|}$

10.2.2 Version pondérée

Chaque sommet $i \in V$ a maintenant un poids w_i.
Problème : minimiser $\sum_{i\in I} w_i$, $I \subseteq V$ étant une couverture.
On pose $x_i = 1$ si le sommet i est pris dans la couverture ($i \in I$), et $x_i = 0$ sinon.
On formule alors le problème sous la forme :

$$min \sum_{i\in V} x_i w_i$$

sous les contraintes

$$x_i + x_j \geq 1 \text{ pour toute arête } (i,j) \in E$$

On peut résoudre ce problème linéaire sur les rationnels, i.e. trouver des $x_i^* \in [0,1]$ qui minimisent la somme des $x_i^* w_i$ sous les contraintes précédentes, et ensuite on pose $x_i = 1 \Leftrightarrow x_i^* \geq \frac{1}{2}$.

(a) C'est une solution (c'est une couverture) : soit $(i,j) \in E$.
Alors $x_i^* + x_j^* \geq 1$, et forcément l'un des x^* est $\geq \frac{1}{2}$. On a donc soit $x_i = 1$ soit $x_j = 1$ dans la solution de notre problème, et donc pour toute arête $(i,j) \in E$ on a bien $x_i + x_j \geq 1$.

(b) C'est une 2-approximation : $x_i \leq 2x_i^*$ que i ait été choisi ou non, et la solution optimale entière a forcément un coût plus élevé que la solution du programme rationnel :

$$C_{algo} = \sum_i x_i w_i \leq \sum_i (2x_i^*) w_i \leq 2C_{opt}$$

10.3 Voyageur de commerce : TSP

10.3.1 Définition

Définition : Soit $G = (V,E)$ un graphe complet et une borne $K \geq 0$.
c : $E \longmapsto \mathbf{N}$: fonction de poids sur les arêtes.
Existe-t-il une tournée (un cycle hamiltonien) de poids $\leq K$?

Théorème 20. *TSP est NP-complet.*

Preuve. – TSP ∈ NP, le certificat est la liste des arêtes de la tournée.

– Complétude : réduction à partir de CH (cycle hamiltonien).

Soit une instance I_1 de CH : graphe $G = (V, E)$. Existe-t-il un cycle hamiltonien de G ? On construit une instance I_2 de TSP : on prend $G' = (V, E')$ complet, K=0, et on définit c tel que $c(e) = \begin{cases} 1 & \text{si e} \in \text{E} \\ 0 & \text{sinon} \end{cases}$

On a bien une réduction :

– Taille polynomiale (trivial)

– Solution de $I_1 \Leftrightarrow$ Solution de I_2 (trivial aussi)

□

10.3.2 Inapproximabilité de TSP

Problème : le problème d'optimisation de TSP consiste à trouver une tournée de poids minimal.

Définition : une λ-approximation est un algorithme polynomial tel que :
$\forall I$ instance du problème, $Sol_{algo}(I) \leq \lambda \cdot Sol_{opt}(I)$

Théorème 21. *$\forall \lambda > 0$, il n'existe pas de λ-approximation sauf si $P = NP$.*

Preuve. *(par l'absurde) :*
Supposons qu'il existe une λ-approximation. On montre que dans ce cas, on peut résoudre cycle hamiltonien en temps polynomial, ce qui prouve que $P = NP$.
Soit $G = (V, E)$ une instance de CH. Dans le graphe complété, on construit c tel que :

$$c(e) = \begin{cases} 1 & \text{si e} \in \text{E} \\ \lambda \cdot |V| + 1 & \text{sinon} \end{cases}$$

On utilise ensuite l'algorithme de λ-approximation pour résoudre TSP. L'algorithme renvoie une solution de coût $C_{algo} \leq \lambda \cdot C_{opt}$.

– Si $C_{algo} \geq \lambda \cdot |V| + 1$, alors $C_{opt} > |V|$ et donc il n'existe pas de CH (car un CH est un TSP de coût $|V|$ pour cette instance).

– Sinon, $C_{algo} \leq \lambda \cdot |V|$ et donc le chemin renvoyé est un CH car il n'utilise aucune arête de coût $\lambda \cdot |V| + 1$. Il existe donc un CH.

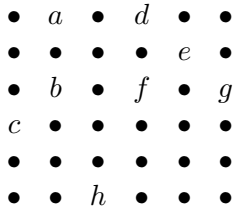
On peut donc répondre au problème CH suivant le résultat de l'algorithme, ce qui prouve que $P = NP$. □

Remarque 8. *On a ici supposé λ constant, mais on peut aller jusqu'à $\lambda = 2^{|V|}$ car alors λ est encore polynomial en la taille de l'instance. Par contre, l'existence d'une $2^{2^{|V|}}$-approximation n'est pas contredite par le résultat précédent.*

10.3.3 2-approximation dans le cas où c vérifie l'inégalité triangulaire

On a un graphe $G = (V, E)$ et un poids c tel que $c(u, v) \leq c(u, w) + c(w, v)$ (c vérifie l'inégalité triangulaire).

Voila un petit exemple (ici c est la distance euclidienne) :



Principe :

- Construire T un arbre couvrant de poids minimum de G (un algorithme glouton le fait en temps polynomial). Ici on trouve l'arbre avec les arêtes a-b, b-c, b-h, a-d, d-e, e-f et e-g.
- Doubler les arêtes de T et faire un parcours de ces arêtes (ex : w = abcbhbadefegeda)
- On ne garde que la première occurrence de chaque lettre pour obtenir une tournée (ex : abchdefga)

Théorème 22. *Cet algorithme est une 2-approximation de TSP, i.e.* $c(\textit{Tournée}) \leq 2 \cdot c(Opt)$

Preuve.
$c(w) = 2 \cdot c(T)$, où w est le chemin qui parcours deux fois chaque arête de l'arbre couvrant de poids minimum.
c(Tournée) $\leq c(w)$ à cause de l'inégalité triangulaire.
$c(T) \leq c(Opt)$ car une tournée optimale à laquelle on supprime une arête est un arbre, et T est l'arbre couvrant de poids minimum.
Donc c(Tournée) $\leq 2 \cdot c(Opt)$, ce qui prouve le résultat. □

Remarque : Il existe une heuristique efficace pour résoudre TSP, l'heuristique de Lin Kernighan, qui fournit de bons résultats (en général à moins de 5% de l'optimal). Cette heuristique est particulièrement efficace lorsque l'inégalité triangulaire est vérifiée.

10.4 Bin Packing : BP

10.4.1 Définition

Données : boites de tailles B et n objets de volume $a_1, \ldots, a_n$.
But : placer ces objets dans un nombre minimal de boites.

Définition : Soit n rationnels $a_1, \ldots, a_n$, avec $0 < a_i \leq 1$. Peut-on les partitionner en K boites $B_1, \ldots, B_K$ de capacité 1, c'est à dire :

$$\sum_{j \in B_k} a_j \leq 1$$

Théorème 23. *Bin Packing est NP-complet.*

Preuve.

- BP $\in$ NP (facile). Certificat : liste des entiers de chaque boite.
- Réduction à partir de 2-Partition. Soit une instance I_1 de 2P : $b_1, \ldots, b_n$. On construit une instance I_2 de BP : $a_i = b_i / \frac{S}{2}$, avec $S = \sum_{i=1}^{n} b_i$, et on prend K=2.
  - Taille polynomiale (trivial)
  - Solution de $I_1 \Leftrightarrow$ Solution de I_2 (trivial aussi)

□

Théorème 24. *Pour tout* $\epsilon > 0$ *il n'existe pas de* $\frac{3}{2} - \epsilon$ *approximation sauf si* $P = NP$.

Preuve. On suppose qu'il existe une $\frac{3}{2} - \epsilon$ approximation pour Bin-Packing. Alors il existerait un algorithme polynomial pour résoudre 2-Partition.

On soumet à l'algorithme les a_i définis précédemment ; s'il existe une 2-Partition, alors l'algorithme renvoie au plus $2(\frac{3}{2} - \epsilon) = 3 - 2\epsilon$ boîtes, donc renvoie 2 boîtes. Sinon, il renvoie au moins 3 boîtes. Notre algorithme polynomial permet donc de résoudre 2-Partition, ce qui implique $P = NP$. □

10.4.2 Next Fit

Principe
- Prendre les objets dans un ordre quelconque.
- Placer l'objet courant dans la dernière boîte utilisée s'il tient, sinon créer une nouvelle boîte.

Variante : First Fit
- Prendre les objets dans un ordre quelconque.
- Placer l'objet courant dans la première boîte utilisée où il tient, sinon créer une nouvelle boîte.

Théorème 25. *Next Fit est une 2-approximation.*

Preuve.
On pose $A = \sum_{i=1}^{n} a_i$.
Par construction, la somme des contenus de toute paire de boîtes consécutives est strictement supérieure à 1. Donc si on a K boîtes, en sommant sur les paires de boîtes consécutives,

$$K - 1 < \sum_{i=1}^{K-1} (contenu(B_i) + contenu(B_{i+1}))$$

En outre, on a $\sum_{i=1}^{K-1} (contenu(B_i) + contenu(B_{i+1})) \leq 2A$, et donc $K - 1 < 2A \leq 2\lceil A \rceil$. Finalement, $K \leq 2\lceil A \rceil$.
Next Fit utilise donc au plus $2\lceil A \rceil$ boîtes. Or l'optimal utilise au moins $\lceil A \rceil$ boîtes. Next Fit est bien une 2-approximation. □

Remarque : La borne 2 est stricte.
Si l'on considère $4n$ éléments de taille $\frac{1}{2}, \frac{1}{2n}$ ($2n$ fois), alors Next Fit utilise $2n$ boîtes, alors que l'optimal en utilise $n + 1$.

10.4.3 Dec First Fit (DFF)

Principe
- Trier les a_i par ordre décroissant.
- Placer l'objet courant dans la première boîte utilisée où il tient, sinon créer une nouvelle boîte.

Théorème 26. $K_{DFF} \leq \frac{3}{2} K_{Opt} + 1$

Preuve. On sépare les a_i en quatre catégories :

$$A = \{a_i > \frac{2}{3}\} \qquad B = \{\frac{2}{3} \geq a_i > \frac{1}{2}\} \qquad C = \{\frac{1}{2} \geq a_i > \frac{1}{3}\} \qquad D = \{\frac{1}{3} \geq a_i\}$$

Dans la solution de DFF, s'il existe au moins une boîte ne contenant que des éléments de D, alors au plus une boîte (la dernière) a un taux d'occupation inférieur ou égal à $\frac{2}{3}$. En effet, si les éléments de D de la dernière boîte n'ont pas pu être mis dans les boîtes précédentes, c'est que celles-ci sont remplies aux $\frac{2}{3}$ au moins. On a donc $K_{opt} \geq \frac{2}{3}(K_{DFF} - 1)$ si l'on ignore la dernière boîte, d'où la borne.

Sinon, la solution de DFF est la même que celle de l'instance où on enlève les éléments de D car on les rajoute à la fin et ils ne rajoutent pas de boîte. Or, montrons que la solution de DFF pour les éléments de A, B et C est optimale. En effet, dans toute solution,

– les éléments de A sont tout seuls.

– les éléments de B et C sont au plus 2 par boîtes et on a au plus un élément de B par boîte.

Or DFF fait la meilleure répartition des C en les appariant au plus grand B ou C compatible. DFF renvoie donc la solution optimale dans ce cas. □

Remarque : On peut montrer $K_{DFF} \leq \frac{11}{9} K_{Opt} + 4$.

10.5 2-Partition

Définition 18. *Soit $a_1 \dots a_n$ des entiers. Existe-il I tel que $\sum_{i \in I} a_i = \sum_{i \notin I} a_i$?*

10.5.1 NP-complétude au sens faible et au sens fort

Quelle est la taille d'une instance I de 2-Partition ? On peut la définir par

$$n + \sum_{i=1}^{n} \lceil log a_i \rceil \qquad n + max_{i \in \{1..n\}} \lceil log a_i \rceil \qquad n \lceil log \sum_{i=1}^{n} a_i \rceil$$

Attention : *Résolution de 2-Partition par programmation dynamique* On définit $m(i, T)$ par : il existe un sous-ensemble I des $a_1 \dots a_i$ tel que $\sum_{j \in I} a_j = T$. On a :

$$m(i, T) = m(i-1, T) \vee m(i, T - a_i)$$

On a la solution du problème en $m(n, \frac{P_{sum}}{2})$ avec $P_{sum} = \sum_{i=1}^{n} a_i$.

La complexité est en $O(nS)$. On a donc un algorithme pseudo-polynômial.

Pour une instance I on définit $Max(I)$ par (au choix) :

$$\sum_{i=1}^{n} a_i \qquad Max_{i \in \{1 \dots n\}} a_i \qquad \frac{\sum_{i=1}^{n} a_i}{n}$$

Pour un polynôme P, on définit $Probleme|_P$ en restreignant $Probleme$ à ses instances I telles que $Max(I) \leq P(taille(I))$

Un problème est NP-complet au sens fort si et seulement si il existe un polynôme P tel que $Probleme|_P$ reste NP-complet.

Un problème de graphes, sans nombres, est NP-complet au sens fort. Pour les problèmes avec des nombres (dont les problèmes de graphes pondérés), il faut faire attention. Ainsi 2-Partition est NP-complet au sens faible, de même que Subset-Sum, ou Knapsack : intuitivement, cela veut dire que le problème n'est difficile que si on ne borne pas la taille des données dans l'instance examinée. Un exemple de problèmes avec des nombres qui est NP-complet au sens fort est 3-Partition, dont le nom est trompeur car il ne ressemble guère à 2-Partition : étant donnés un entier B, $3n$ entiers a_i, peut-on les partitionner en n triplets, chacun de somme B ? On peut supposer que la somme de tous les a_i vaut nB (sinon il n'y a pas de solution) et qu'ils tous compris strictement entre $B/4$ et $B/2$ (pour qu'il faille exactement trois éléments pour avoir une somme égale à B).

10.5.2 Approximation gloutonnes

Le problème d'optimisation associé à 2-Partition est le suivant : soit $a_1 \dots a_n$ des entiers. Minimiser

$$\max_{I \subset \{1,2,\dots,n\}} \left(\sum_{i \in I} a_i, \sum_{i \notin I} a_i \right)$$

(bien sûr le minimum sera toujours au moins $P_{sum}/2$, où $P_{sum} = \sum_{i=1}^{n} a_i$).

On peut voir le problème précédent comme un problème d'ordonnancement à deux machines (processeurs) identiques. Il y a n tâches indépendantes, qui peuvent être exécutées sur l'une des deux machines en temps a_i. L'objectif est de minimiser le temps total d'exécution : c'est bien le problème précédent. Il y a deux variantes naturelles si on veut le résoudre avec un algorithme glouton :
- GLOUTON : prendre les tâches dans n'importe quel ordre. Les allouer au processeur le moins chargé.
- GLOUTON-TRI : le même que précédemment, mais après avoir trié les tâches par poids décroissant.
L'idée de trier est que l'arrivée d'une grosse tâche à la fin risque de déséquilibrer toute l'exécution. Par contre, il faut connaître toutes les tâches avant de commencer l'ordonnancement (algorithme off-line). Le glouton sans tri peut s'appliquer à un problème on-line, où les tâches arrivent dynamiquement (comme des jobs d'utilisateurs à exécuter sur un serveur bi-pro).

Théorème 27. *GLOUTON est une* 4/3 *approximation, GLOUTON-TRI est une* 7/6 *approximation, et ces facteurs d'approximation ne peuvent pas être améliorés.*

Preuve. On montre d'abord que les bornes sont strictes. Pour GLOUTON, prendre trois tâches de poids 1, 1, et 2, GLOUTON ordonnance en temps 3 alors que l'optimal est 2. Pour GLOUTON-TRI, prendre cinq tâches, trois de poids 2 et deux de poids 3. GLOUTON-TRI ordonnance en temps 7, alors que l'optimal est 6.

Pour l'approximation, soit $P_{sum} = \sum_{i=1}^{n} a_i$. En termes d'ordonnancement, P_{sum} est le temps séquentiel. Soit Opt le temps optimal avec deux machines, alors $Opt \geq P_{sum}/2$. Notons aussi que $a_i \leq Opt$ pour toute tâche a_i.

Commençons par GLOUTON. Soient P_1 et P_2 les deux processeurs. Disons que P_1 termine l'exécution, et soit a_j la dernière tâche exécutée sur cette machine. Soit M_1 le temps d'exécution sur P_1 (la somme des tâches qui lui sont allouées), et M_2 le temps d'exécution sur P_2. Comme P_1 termine, $M_1 \geq M_2$. Bien sûr $M_1 + M_2 = P_{sum}$.

On a supposé que a_j est la dernière tâche de P_1. Soit $M_0 = M_1 - a_j$ la charge de P_1 avant que GLOUTON ne lui alloue la tâche a_j. Pourquoi GLOUTON choisit-il P_1 ? Parce qu'à ce moment, P_2 est plus chargé que P_1 : M_0 est inférieur ou égal à la charge de P_2 à ce moment, elle-même inférieure à sa charge finale M_2 (noter qu'on a pu rajouter des tâches à P_2 après avoir ordonnancé a_j) : donc $M_0 \leq M_2$. Pour résumer, le temps de GLOUTON est M_1, avec

$$M_1 = M_0 + a_j = 1/2((M_0 + M_0 + a_j) + a_j) \leq 1/2((M_0 + M_2 + a_j) + a_j) = 1/2(P_{sum} + a_j)$$
$$M_1 \leq Opt + a_j/2 \leq Opt + Opt/2,$$

d'où la 3/2-approximation pour GLOUTON.

Pour GLOUTON-TRI on reprend la preuve précédente mais on va être plus précis que la majoration $a_j \leq Opt$. D'abord, si $a_j \leq Opt/3$, on obtient juste ce qu'il faut, $M_1 \leq 7/6Opt$. Mais si $a_j > Opt/3$, alors $j \leq 4$: en effet, si a_j était la cinquième tâche, comme les tâches sont triées il y aurait au moins 5 taches de poids supérieur à $Opt/3$, et dans tout ordonnancement, y compris l'optimal, l'une des machines aurait la charge de trois d'entre elles, contradiction. On remarque alors que le temps de GLOUTON-TRI est le même pour ordonnancer toutes les tâches que pour ordonnancer les $j \leq 4$ premières. Mais pour tout ensemble d'au plus quatre tâches, GLOUTON-TRI est optimal, et $M_1 = Opt$. □

10.5.3 Une $(1+\epsilon)$-approximation

On va montrer que 2-Partition admet un schéma d'approximation polynomial PTAS, pour *Polynomial Time Approximation Scheme* : pour toute constante $\lambda = 1 + \varepsilon > 1$, il existe une

λ-approximation, i.e. un algorithme dont la performance est garantie à un facteur λ, et dont le temps d'exécution est polynomial. Attention : polynomial en la taille de l'instance de 2-Partition, mais pas forcément en $1/\varepsilon$, c'est là toute la subtilité. On verra plus loin un FPTAS, pour *Fully Polynomial Time Approximation Scheme*, où le temps est polynomial à la fois en la taille de l'instance et en $1/\varepsilon$.

Théorème 28. $\forall\, \varepsilon > 0$, 2-*Partition admet une* $(1+\epsilon)$-*approximation.*

Preuve. On pose :

$$P_{sum} = \sum_{i=1}^{n} a_i$$

$$P_{max} = \max a_i$$

$$L = \max(\frac{1}{2} P_{sum}, P_{max})$$

On a déjà vu plus haut que :

Lemme 10. $L \leq Opt$.

Soit I une instance de 2-partition : $a_1, ..., a_n$. et $\epsilon > 0$. On appelle big jobs, les a_i tels que : $a_i > \epsilon L$ et small jobs ceux tels que : $a_i \leq \epsilon L$. Considérons l'instance I^* qui regroupe uniquement les big jobs et $\lfloor \frac{S}{\epsilon L} \rfloor$ jobs de taille ϵL, où $S = \sum_{a_i \text{ small job}} a_i$.

Lemme 11. $Opt^* \leq (1+\epsilon)Opt$ *(inégalité (A)), où Opt est un ordonnancement optimal pour I et* Opt^* *un ordonnancement optimal pour* I^*.

Preuve. Considérons un ordonnancement optimal Opt pour l'instance de départ I. Soient S_1 la somme des small jobs sur la machine P_1 et S_2 celle des small jobs sur la machine P_2.

P_1 : ——— oo ——- ooo ——-

P_2 : —— oo ———- o ——

où o est un small job et —— un big job.

Sur chaque machine P_i, on laisse les gros jobs où ils sont, et on remplace les petits jobs par $\left\lceil \frac{S_i}{\epsilon L} \right\rceil$ jobs de taille ϵL. On a symboliquement :

P_i : ——- —— ——- oooo oooo oo

Comme

$$\left\lceil \frac{S_1}{\epsilon * L} \right\rceil + \left\lceil \frac{S_2}{\epsilon * L} \right\rceil \geq \left\lfloor \frac{S_1 + S_2}{\epsilon * L} \right\rfloor = \left\lfloor \frac{S}{\epsilon * L} \right\rfloor$$

on a ordonnancé au moins autant de jobs de taille ϵL qu'il y en a dans I^*. En faisant ainsi, on a augmenté le temps de P_i d'au plus :

$$\left\lceil \frac{S_i}{\epsilon L} \right\rceil \cdot \epsilon L \; - \; S_i \leq \epsilon L$$

Le temps d'exécution est au moins égal à l'optimal pour I^*, donc :

$$Opt^* \leq Opt + \epsilon L \leq Opt + \epsilon . Opt$$

□

Comment ordonnancer I^* ? Combien y a-t-il de jobs dans I^* ? Quand on a remplacé les petits jobs de I par des jobs de taille ϵL, on n'a pas augmenté le temps d'exécution total, qui est borné par $P_{sum} \leq 2L$. Chaque job de I^* a une durée au moins ϵL, donc il y en a au plus $\frac{2L}{\epsilon L} = \frac{2}{\epsilon}$, c'est une constante. Au passage, la taille de I^*, constante donc, est bien polynomiale en la taille de l'instance de départ I. Pour odonnancer I^*, il suffit d'essayer toutes les $2^{\frac{2}{\epsilon}}$ possibilités d'ordonnancement et de garder la meilleure. Bien sûr ceci a un coût non polynomial en $1/\epsilon$, mais c'est une constante en la taille de l'instance de départ.

Retour à l'instance de départ I On a désormais l'ordonnancement optimal σ^* pour I^*. Soit L_i^* la charge de la machine P_i avec σ^*, qu'on décompose en $L_i^* = B_i^* + S_i^*$, où B_i^* sont les big jobs et S_i^* les small jobs. Par construction de I^*, on a l'inégalité (B) :

$$S_1^* + S_2^* = \epsilon L \left\lfloor \frac{S}{\epsilon L} \right\rfloor > S - \epsilon L$$

On peut alors considérer l'ordonnancement σ pour I où les big jobs sont inchangés et où on fait un glouton avec les small jobs : on les alloue d'abord à P_1 dans l'espace $S_1^* + 2 * \epsilon * L$, puis quand il n'y a plus de place, on les alloue sur P_2 dans l'espace S_2^* :

Lemme 12. *En procédant ainsi, on case tous les small jobs.*

Preuve. En effet, on case au moins une quantité $S_1^* + \epsilon L$ de small jobs sur P_1 : chacun d'entre eux prenant moins de ϵL, il faut qu'il reste moins de ϵL place dans l'intervalle alloué sur P_1 pour qu'on soit obligé de passer à P_2. Mais alors il reste au plus $S - (S_1^* + \epsilon L) \leq S_2^*$ (d'après l'inégalité (B)) à allouer à P_2, donc on a bien la place.

Au final, on a construit σ ordonnancement pour I en temps polynomial. Son temps d'exécution excède celui de l'ordonnancement optimal pour I^* d'au plus $2\epsilon L$, donc

$$Opt \leq ordo(\sigma) \leq Opt^* + 2\epsilon L \leq (1 + 3\epsilon).Opt,$$

la dernière inégalité venant de (A). □

Ceci conclut la preuve du théorème. □

Remarque 9. *On peut proposer une preuve plus simple en utilisant la technique des big jobs et celle du GLOUTON du paragraphe précédent pour les petits jobs. On ne considère que les bigs jobs et tous leurs ordonnancements possible. Ils sont en nombre constant, on garde le meilleur. Le temps de cet ordonnancement Opt^{big} est inférieur à Opt, l'ordonnancement optimal pour l'instance de départ I. On ordonnance alors les small jobs sur les deux machines en GLOUTON, à la suite des big jobs. Soit le temps total de l'ordonnancement ne change pas, i.e. reste égal à Opt^{big}, et alors il est optimal. Soit la machine qui termine termine avec un small job a_j. Mais alors sa charge avant cette dernière allocation ne pouvait pas dépasser $1/2P_{sum}$, sinon on aurait attribué a_j à l'autre machine (même idée et preuve que pour GLOUTON). Et donc le temps est au plus $1/2P_{sum} + a_j \leq L + \epsilon L \leq (1 + \epsilon)Opt$.*

10.5.4 FPTAS pour 2-Partition

On va maintenant montrer que 2-Partition admet même un FPTAS (*Fully Polynomial Time Approximation Scheme*), soit que pour toute instance I de 2-Partition et tout $\varepsilon > 0$, il existe une $(1 + \varepsilon)$-approximation qui soit polynomiale à la fois en la taille de I *et en* $1/\varepsilon$.

La relation entre PTAS et FPTAS est la même qu'entre continuité et uniforme continuité : on fait changer le $\forall\varepsilon$ de côté. Et bien entendu, admettre un FPTAS étant une propriété plus forte, (FPTAS $\Rightarrow$ PTAS).

Théorème 29. *$\forall\varepsilon > 0$, 2-Partition admet une $(1+\varepsilon)$-approximation polynômiale en $(1/\varepsilon)$*

Preuve. On va coder les différents ordonnancements sous formes de *Vector Sets* où le premier (resp. deuxième) élément de chaque vecteur représente le temps de travail de la première (resp. deuxième) machine :

$$\begin{cases} VS_1 = \{[a_1, 0], [0, a_1]\} \\ VS_{k-1} \to VS_k : \forall [x, y] \in VS_{k-1}, [x + a_k, y] \text{ et } [x, y + a_k] \in VS_k \end{cases}$$

Entrée : les tâches $a_1 \dots a_n$.

Sortie : le vecteur $[x, y] \in VS_n$ qui minimise $\max(x, y)$.

Idée : on va énumérer tous les ordonnancements possibles, mais en "élaguant" au passage les branches qui ne nous intéressent pas.

Soit $\Delta = 1 + \frac{\varepsilon}{2n}$.
On effectue un quadrillage du plan selon les puissances de Δ, entre 0 et Δ^M.

$$M = \lceil \log_\Delta P_{sum} \rceil = \left\lceil \frac{\log P_{sum}}{\log \Delta} \right\rceil \leq \left\lceil \left(1 + \frac{2n}{\varepsilon}\right) \ln(P_{sum}) \right\rceil$$

En effet, si $z \geq 1, \ln(z) \geq 1 - \frac{1}{z}$.

L'idée est de ne rajouter les vecteurs que s'ils tombent dans une case vide.

Comme M est polynômial en $\frac{1}{\varepsilon}$ et en $\ln(P_{sum})$ et que $taille(I) \geq \ln(P_{sum})$, il suffit de démontrer que notre algorithme est bien une $(1+\varepsilon)$-approximation.

Deux vecteurs $[x_1, y_1]$ et $[x_2, y_2]$ tombent dans la même case ssi

$$\begin{cases} \dfrac{x_1}{\Delta} \leq x_2 \leq \Delta x_1 \\ \\ \dfrac{y_1}{\Delta} \leq y_2 \leq \Delta y_1 \end{cases}$$

$$\begin{cases} VS_1^\# = VS_1 \\ VS_{k-1}^\# \to VS_k^\# \end{cases} :$$

$\forall [x, y] \in VS_{k-1}^\#, [x + a_k, y] \in VS_k^\#$ sauf si un vecteur de $VS_k^\#$ est déjà dans la même case. De même pour $[x, y + a_k]$.

On ne garde donc qu'au plus un vecteur par case à chaque étape, ce qui donne une complexité en $n \times M^2$, soit polynômiale en $taille(I)$.

Lemme 13. *$\forall [x, y] \in VS_k, \exists [x^\#, y^\#] \in VS_k^\#$ tel que $x^\# \leq \Delta^k x$ et $y^\# \leq \Delta^k y$*

Preuve. **Par récurrence**

k = 1 : trivial

k - 1 → k : Soit $[x, y] \in VS_k$,

$$\begin{cases} x = u + a_k \text{ et } y = v \\ \text{ou} \qquad\qquad\qquad\qquad \text{où } [u, v] \in VS_{k-1} \\ x = u \text{ et } y = v + a_k \end{cases}$$

Par hypothèse de récurrence, $\exists [u^\#, v^\#] \in VS_{k-1}^\#$ où $u^\# \leq \Delta^{k-1} u$ et $v^\# \leq \Delta^{k-1} v$

$[u^\# + a_k, v]$ n'est pas nécessairement dans $VS_k^\#$ mais on sait qu'il y a au moins un élément de la même case dans $VS_k^\#$:

$$\exists [x^\#, y^\#] \in VS_k^\# \text{ tel que } x^\# \leq \Delta \left(u^\# + a_k \right) \text{ et } y^\# \leq \Delta v^\#$$

$$\Rightarrow \begin{cases} x^\# \leq \Delta^k u + \Delta a_k \leq \Delta^k (u + a_k) = \Delta^k x \\ y^\# \leq \Delta v^\# \leq \Delta^k y \end{cases}$$

et symétriquement pour $[u, y + a_k]$. □

On en déduit donc $max(x^\#, y^\#) \leq \Delta^n max(x, y)$.

Reste à montrer $\Delta^n \leq (1 + \varepsilon)$:

$$\Delta^n = \left(1 + \frac{\varepsilon}{2n}\right)^n$$

Lemme 14. *Pour* $0 \leq z \leq 1$, $f(z) = \left(1 + \frac{z}{n}\right)^n - 1 - 2z \leq 0$

Preuve. $f'(z) = \frac{1}{n} n \left(1 + \frac{z}{n}\right)^{n-1} - 2$.

On en déduit que f est convexe et atteint son minimum en $\lambda_0 = n \left(\sqrt[n-1]{2} - 1 \right)$. Or $f(0) = -1$ et $f(1) = g(n) = \left(1 + \frac{1}{n}\right)^n - 3$. On compare une fonction linéaire à une fonction convexe : si la relation est vraie aux bornes, elle est vraie partout. □

Ceci conclut la démonstration.

□

10.6 Exercices

Exercice 10.6.1. *Approximabilité de SUBSET-SUM*

Dans un précédent TD, on s'est intéressé au problème de décision SUBSET-SUM consistant à savoir s'il existe un sous-ensemble d'entiers de S dont la somme vaut exactement t. Le problème d'optimisation qui lui est associé prend aussi en entrée un ensemble d'entiers strictement positifs S et un entier t et il consiste à trouver un sous-ensemble de S dont la somme est la plus grande possible sans dépasser t (cette somme qui doit donc approcher le plus possible t sera appelée *somme optimale*).

On suppose que $S = \{x_1, x_2, \ldots, x_n\}$ et que les ensembles sont manipulés sous forme de listes triées par ordre croissant. Pour une liste d'entiers L et un entier x, on note $L+x$ la liste d'entiers

obtenue en ajoutant x à chaque entier de L. Pour listes L et L', on note **Fusion**(L, L') la liste contenant l'union des éléments des deux listes.

1 - *Premier algorithme*
Algorithm:Somme(S, t)

---

début
  $n \leftarrow |S|$;
  $L_0 \leftarrow \{0\}$;
  pour i **de** *1* **à** n **faire**
    $L_i \leftarrow$ **Fusion**$(L_{i-1}, L_{i-1} + x_i)$;
    Supprimer de L_i tout élément supérieur à t ;
  retourner le plus grand élément de L_n
fin

Quel est la distance entre la valeur retournée par cet algorithme et la somme optimale ? Quelle est la complexité de cet algorithme dans le cas général ? Et si pour un entier $k \geq 1$ fixé, on ne considère que les entrées telles que $t = \mathcal{O}(|S|^k)$, quelle est la complexité de cet algorithme ?

2 - *Deuxième algorithme*

Cet algorithme prend en entrée un paramètre ϵ en plus, où ϵ est un réel vérifiant $0 < \epsilon < 1$.
Algorithm:Somme-avec-seuillage(S, t)

---

début
  $n \leftarrow |S|$;
  $L_0 \leftarrow \{0\}$;
  pour i **de** *1* **à** n **faire**
    $L_i \leftarrow$ **Fusion**$(L_{i-1}, L_{i-1} + x_i)$;
    $L_i \leftarrow$ **Seuiller**$(L_i, \epsilon/2n)$;
    Supprimer de L_i tout élément supérieur à t ;
  retourner le plus grand élément de L_n
fin

L'opération **Seuiller** décrite ci-dessous réduit une liste $L =< y_0, y_1, \ldots, y_m >$ (supposée triée par ordre croissant) avec le seuil δ :
Algorithm:Seuiller(L, δ)

---

début
  $m \leftarrow |L|$;
  $L' \leftarrow < y_0 >$;
  $dernier \leftarrow y_0$;
  pour i **de** *1* **à** m **faire**
    si $y_i > (1 + \delta) dernier$ **alors**
      Insérer y_i à la fin de L' ;
      $dernier \leftarrow y_i$;
  retourner L'
fin

Evaluer le nombre d'éléments dans L_i à la fin de la boucle. En déduire la complexité totale de l'algorithme. Pour donner la qualité de l'approximation fournie par cet algorithme, borner le ratio valeur retournée/somme optimale.

Correction.

On a un ensemble S = $\{x_1, \ldots, x_n\}$ d'entiers strictement positifs ainsi qu'un entier t . On cherche une somme partielle d'éléments de S maximale et inférieure à t. On appellera somme optimale ce nombre.

1 - Premier Algorithme

Dans le premier algorithme , L_i représente l'ensemble des sommes partielles des éléments de $\{x_1, \ldots, x_i\}$ inférieures à t. L_n représente donc l'ensemble des sommes partielles de S inférieures à t. Par conséquent, max(L_n) renvoie exactement la somme optimale de S. De plus, cet algorithme a testé toutes les sommes partielles inférieures à t.

On en déduit donc que :

- La différence entre le résultat trouvé et la somme optimale est 0
- Dans le pire des cas, on teste toutes les sommes partielles de S soit une complexité de O(2^n)
- Si t = O($|S|^k$) pour k un entier fixé, alors $|L_i|$ = O($|S|^k$). Au pas i de la boucle, la fusion et la suppression se font en $O(|L_i|)$. Comme il y a n pas dans la boucle, la complexité totale est O($|S|^{k+1}$)

2 - Deuxième Algorithme

Considérons une liste l = $\{y_0, \ldots, y_m\}$ d'entiers triés par ordre croissant et $\delta > 0$. L'opération de seuillage va consister à dire : soit y' le dernier élément gardé de l (on garde y_0 au départ) si $y_i > y' * (1 + \delta)$ alors on va garder y_i. En gros, on enlève des éléments qui sont trop proches d'autres proches d'autres éléments.

L'algorithme proposé va fonctionner de la même façon que le précédent sauf qu'il va seuiller à chaque pas de la boucle L_i par rapport à $\epsilon/2n$. On a alors $L_n = \{y_0, \ldots, y_m\}$. On va essayer de majorer m. On sait que :

$t \geq y_m \geq (1 + \epsilon/2n)^{m-1} * y_0 \geq (1 + \epsilon/2n)^{m-1}$

D'où : $m \leq \ln t / \ln(1 + \epsilon/2n) \leq (\ln t * 2n)/(\ln 2 * \epsilon) + 1$

m est donc polynomial en ln t et en $1/\epsilon \ln t$ polynomial en la taille des données l'algorithme est en O(m * n) donc polynomial en la taille des données.

Il faut maintenant vérifier qu'on a bien trouvé une $(1+\epsilon)$ Approximation en montrant que :

$maxL_n \geq (1 + \epsilon) * Opt$(somme optimale)

On note P_i l'ensemble des sommes partielles de $\{x_1, \ldots, x_i\}$ inférieures à t.

Invariant : $\forall x \in P_i, \exists y \in L_i,\ x/(1 + \delta)^i \leq y \leq x$

Par récurrence :

- i = 0 : OK

- On suppose que c'est vrai pour (i-1) et on le montre pour i :

Soit $x \in P_i$

$P_i = P_{i-1} \cup (P_{i-1} + x_i)$

$x = x' + e$ avec $x' \in P_{i-1},\ e = 0\ ou\ e = x_i$

$x' \in P_{i-1}$ donc $\exists y',\ x'/(1 + \delta)^i \leq y' \leq x'$

Si y' + e est conservé par le seuillage :

$(x' + e)/(1 + \delta)^i \leq x'/(1 + \delta)^{i-1} + e \leq y' + e \leq x' + e = x$

Si y' + e n'est pas conservé par le seuillage :

$\exists y'' \in L_i,\ y'' \leq y' + x_i \leq y'' * (1 + \delta)$

$(y' + e)/(1 + \delta) \leq y'' \leq y' + e \leq x' + e \leq x$

$(x'/(1 + \delta)^{i-1} + e)/(1 + \delta) \leq y''$

$(x' + e)/(1 + \delta)^i \leq y''$

C'est l'encadrement recherché.

Grâce à l'invariant, on obtient : $Algo \geq Opt/(1+\epsilon/2n)^n \geq Opt(1+\epsilon)$

On a donc une $1+\epsilon$ *approximation de* $Subset_sum$ *polynomial en la taille des donnees et polynomial en* $1/\epsilon$

10.7 Références bibliographiques

Ce chapitre s'inspire des livres de Cormen [2], Garey et Johnson [5] et Wilf [12]. Les schémas d'approximation pour 2-PARTITION sont tirés de [9].

Bibliographie

[1] G. Ausiello, P. Crescenzi, G. Gambosi, V. Kann, A. Marchetti-Spaccamela, and M. Protasi. *Complexity and Approximation.* Springer Verlag, 1999.

[2] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* The MIT Press, 1990.

[3] A. Darte and S. Vaudenay. *Algorithmique et optimisation : exercices corrigés.* Dunod, 2001.

[4] D. Froidevaux, M. Gaudel, and D. Soria. *Types de données et algorithmes.* McGraw-Hill, 1990.

[5] M. R. Garey and D. S. Johnson. *Computers and Intractability, a Guide to the Theory of NP-Completeness.* W. H. Freeman and Company, 1979.

[6] D. E. Knuth. *The Art of Computer Programming ; volumes 1-3.* Addison-Wesley, 1997.

[7] D. C. Kozen. *The Design and Analysis of Algorithms.* Springer-Verlag, 1992.

[8] G. J. E. Rawlins. *Compared to what ? : an introduction to the analysis of algorithms.* Computer Science Press, Inc, 1992.

[9] P. Schuurman and G. Woeginger. Approximation schemes - a tutorial. Research Report Woe-65, CS Department, TU Graz, Austria, 2001. Available at `www.diku.dk/undervisning/2003e/404/ptas_schu_woeg.ps`.

[10] V. V. Vazirani. *Approximation Algorithms.* Springer-Verlag, 2001.

[11] D. B. West. *Introduction to Graph Theory.* Prentice Hall, 1996.

[12] H. S. Wilf. *Algorithms and Complexity.* A.K. Peter, 1985. Available at `http://www.cis.upenn.edu/~wilf/`.