

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

**Cours**

Anne Benoit

**Travaux Dirigés (2008-2009)**

Benjamin Depardon, Christophe Moulleron, Clément Resvoy

Septembre 2009



# Table des matières

<b>1</b>	<b>Introduction : calcul de <math>x^n</math></b>	<b>9</b>
1.1	Énoncé du problème . . . . .	9
1.2	Algorithme naïf . . . . .	9
1.3	Méthode binaire . . . . .	9
1.4	Méthode des facteurs . . . . .	10
1.5	Arbre de Knuth . . . . .	10
1.6	Résultats sur la complexité . . . . .	11
1.7	Exercices . . . . .	12
1.8	Références bibliographiques . . . . .	14
<b>2</b>	<b>Diviser pour régner</b>	<b>15</b>
2.1	Algorithme de Strassen . . . . .	15
2.2	Produit de deux polynômes . . . . .	17
2.3	Master theorem . . . . .	18
2.4	Résolution des récurrences . . . . .	19
2.4.1	Résolution des récurrences homogènes . . . . .	19
2.4.2	Résolution des récurrences avec second membre . . . . .	19
2.5	Multiplication et inversion de matrices . . . . .	20
2.6	Exercices . . . . .	21
2.7	Références bibliographiques . . . . .	23
<b>3</b>	<b>Programmation dynamique</b>	<b>25</b>
3.1	Pièces de Monnaies . . . . .	25
3.2	Le problème du sac à dos . . . . .	26
3.2.1	En glouton . . . . .	26
3.2.2	Par programmation dynamique . . . . .	26
3.3	Quelques exemples de programmation dynamique . . . . .	27
3.3.1	Chaînes de matrices . . . . .	27
3.3.2	Plus longue sous-suite . . . . .	28
3.3.3	Location de skis . . . . .	30
3.4	Exercices . . . . .	32
3.5	Références bibliographiques . . . . .	34
<b>4</b>	<b>Algorithmes gloutons</b>	<b>35</b>
4.1	Exemple du gymnase . . . . .	35
4.2	Route à suivre pour le glouton . . . . .	36
4.3	Coloriage d'un graphe . . . . .	37
4.3.1	Algorithme glouton 1 . . . . .	38
4.3.2	Algorithme glouton 2 . . . . .	38

4.3.3	Graphe d'intervalles . . . . .	39
4.3.4	Algorithme de Brelaz . . . . .	39
4.4	Théorie des matroïdes . . . . .	41
4.4.1	Matroïdes . . . . .	41
4.4.2	Algorithme glouton . . . . .	42
4.5	Ordonnancement . . . . .	42
4.6	Exercices . . . . .	44
4.7	Références bibliographiques . . . . .	45
<b>5</b>	<b>Tri</b> . . . . .	<b>47</b>
5.1	Tri fusion . . . . .	47
5.2	Tri par tas : Heapsort . . . . .	47
5.2.1	Définitions . . . . .	47
5.2.2	Tri par tas . . . . .	48
5.2.3	Insertion d'un nouvel élément . . . . .	48
5.2.4	Suppression d'un élément du tas . . . . .	49
5.2.5	Complexité du tri par tas . . . . .	49
5.3	Tri rapide . . . . .	49
5.3.1	Coût . . . . .	50
5.3.2	Médiane en temps linéaire . . . . .	50
5.4	Complexité du tri . . . . .	51
5.4.1	Les grands théorèmes . . . . .	51
5.4.2	Démonstration des théorèmes . . . . .	52
5.4.3	Peut-on atteindre la borne ? . . . . .	54
5.5	Exercices . . . . .	55
5.6	Références bibliographiques . . . . .	57
<b>6</b>	<b>Graphes</b> . . . . .	<b>59</b>
6.1	Définitions . . . . .	59
6.2	Arbres . . . . .	59
6.2.1	Caractérisation . . . . .	59
6.2.2	Parcours d'arbres binaires . . . . .	60
6.2.3	Arbres binaires de recherche . . . . .	63
6.3	Structures de données pour les graphes . . . . .	65
6.4	Accessibilité . . . . .	69
6.4.1	Rappels sur les relations binaires . . . . .	69
6.4.2	Chemins dans les graphes . . . . .	70
6.4.3	Fermeture transitive . . . . .	70
6.5	Plus courts chemins . . . . .	73
6.5.1	Définitions . . . . .	73
6.5.2	Présentation des plus courts chemins . . . . .	74
6.5.3	Avec des poids positifs . . . . .	74
6.5.4	Chemins algébriques dans les semi-anneaux . . . . .	75
6.5.5	Algorithme de Dijkstra . . . . .	76
6.6	Parcours en largeur . . . . .	78
6.7	Parcours en profondeur . . . . .	80
6.7.1	Première version . . . . .	80
6.7.2	Analyse fine du parcours en profondeur . . . . .	81
6.8	Tri topologique . . . . .	82
6.9	Forte connexité . . . . .	83

6.10	Exercices	83
6.11	Références bibliographiques	88
<b>7</b>	<b>Tables de hachage</b>	<b>89</b>
7.1	Recherche en table	89
7.2	Tables de hachage	89
7.3	Collisions séparées	90
7.4	Adressage ouvert	91
7.5	Références bibliographiques	92
<b>8</b>	<b>Analyse amortie</b>	<b>93</b>
8.1	Compteur	93
8.1.1	Méthode des acomptes	93
8.1.2	Méthode du potentiel	94
8.2	Malloc primaire	94
8.2.1	Méthode globale	94
8.2.2	Méthode des acomptes	94
8.2.3	Méthode du potentiel	94
8.3	Insertion ET suppression	95
8.4	Gestion des partitions	95
8.4.1	Représentation en listes chaînées	95
8.4.2	Représentation en arbres	95
8.5	Références bibliographiques	96
<b>9</b>	<b><math>\mathcal{NP}</math>-Complétude</b>	<b>97</b>
9.1	Problèmes de $\mathcal{P}$	97
9.1.1	Pensée du jour (PJ)	97
9.1.2	Définition	97
9.1.3	Exemples	98
9.1.4	Solution d'un problème	99
9.2	Problèmes de $\mathcal{NP}$	99
9.2.1	Définition	99
9.2.2	Problèmes NP-complets	99
9.2.3	Exemples de problèmes dans $\mathcal{NP}$	100
9.2.4	Problèmes de décision vs optimisation	100
9.2.5	Exemple de problèmes n'étant pas forcément dans $\mathcal{NP}$	100
9.2.6	Problèmes polynomiaux	101
9.3	Méthode de réduction	102
9.4	3-SAT	102
9.5	Clique	104
9.6	Couverture par les sommets	105
9.7	Cycle hamiltonien	106
9.8	Coloration de graphes	106
9.8.1	COLOR	107
9.8.2	3-COLOR	109
9.8.3	3-COLOR-PLAN	110
9.9	Exercices	112
9.10	Références bibliographiques	114

<b>10 Algorithmes d'approximation</b>	<b>115</b>
10.1 Définition . . . . .	115
10.2 Vertex cover . . . . .	115
10.2.1 Version classique . . . . .	115
10.2.2 Version pondérée . . . . .	116
10.3 Voyageur de commerce : TSP . . . . .	116
10.3.1 Définition . . . . .	116
10.3.2 Inapproximabilité de TSP . . . . .	117
10.3.3 2-approximation dans le cas où $c$ vérifie l'inégalité triangulaire . . . . .	117
10.4 Bin Packing : BP . . . . .	118
10.4.1 Définition . . . . .	118
10.4.2 Next Fit . . . . .	119
10.4.3 Dec First Fit (DFF) . . . . .	119
10.5 2-Partition . . . . .	120
10.5.1 NP-complétude au sens faible et au sens fort . . . . .	120
10.5.2 Approximation gloutonnes . . . . .	120
10.5.3 Une $(1 + \epsilon)$ -approximation . . . . .	121
10.5.4 FPTAS pour 2-Partition . . . . .	123
10.6 Exercices . . . . .	125
10.7 Références bibliographiques . . . . .	127

# 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 Hirschhoff, 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  $y_1, y_2, \dots, y_{i-1}$ , 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 multiplications 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  $n$  :

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

**Exemple :**  $x^{15} = (x^3)^5 = x^3 \cdot (x^3)^4 = \dots$  (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.

$$\begin{aligned} x^{33} &= (x^3)^{11} = x^3 \cdot (x^3)^{10} = x^3 \cdot ((x^3)^2)^5 = x^3 \cdot y \cdot y^4 \text{ avec } y = (x^3)^2. \\ x^{33} &= x \cdot x^{25}. \end{aligned}$$

**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.

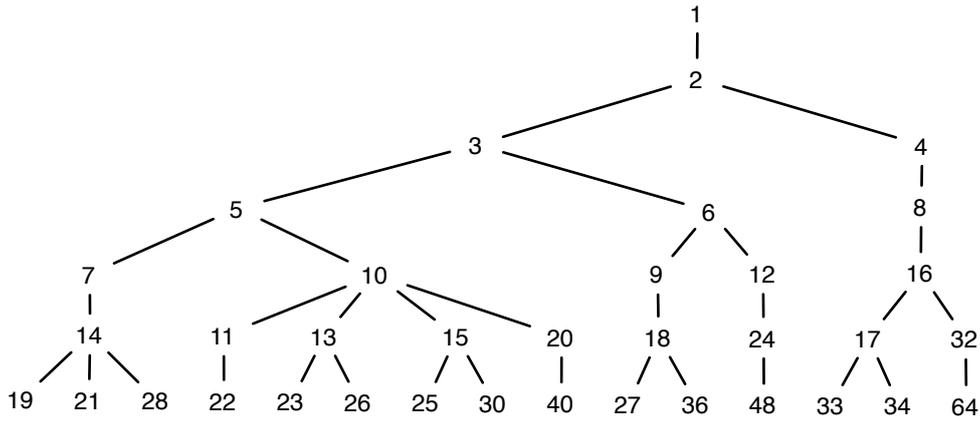


FIG. 1.1: Les sept premiers niveaux de l'arbre de Knuth.

**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

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

(dans cet ordre), où  $1, a_1, \dots, a_{k-1} = n$  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.**  $Opt(n) \geq \lceil \log n \rceil$

*Démonstration.* 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.  $\square$

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 :

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

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

*Démonstration.* 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  $m$  :

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

où chaque  $\alpha_i$  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  $x^{\alpha_i}$ , mais comme on les calcule "au vol" on peut les calculer tous sans surcoût.

Ensuite on calcule successivement :

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

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  $\lceil \log_m n \rceil$ , on trouve un coût total de

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

(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  $2^k \leq \sqrt{\log n}$ ).  $\square$

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*

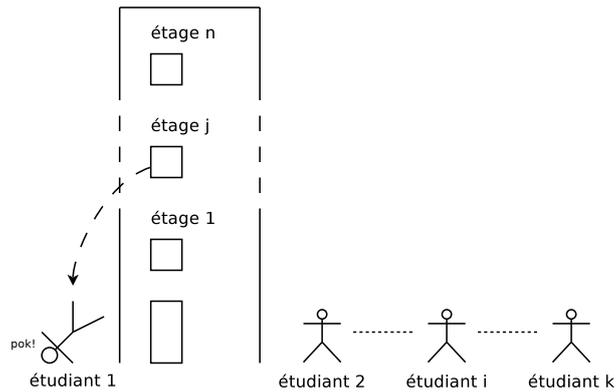
Le problème est de déterminer à partir de quel étage d'un immeuble, sauter par une fenêtre est fatal. Vous êtes dans un immeuble à  $n$  étages (numérotés de 1 à  $n$ ) et vous disposez de  $k$  étudiants. Les étudiants sont classés par notes de partiel croissantes. Il n'y a qu'une opération possible pour tester si la hauteur d'un étage est fatale : faire sauter le premier étudiant de la liste par la fenêtre. S'il survit, vous pouvez le réutiliser ensuite (évidemment, l'étudiant survivant reprend sa place initiale dans la liste triée), 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.

**Question 1.1** Si  $k \geq \lceil \log_2(n) \rceil$ , proposer un algorithme en  $\mathcal{O}(\log_2(n))$  sauts.

**Question 1.2** Si  $k < \lceil \log_2(n) \rceil$ , proposer un algorithme en  $\mathcal{O}(k + \frac{n}{2^{k-1}})$  sauts.

**Question 1.3** Si  $k = 2$ , proposer un algorithme en  $\mathcal{O}(\sqrt{n})$  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 connaît 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 " $i \rightarrow j$  ?"), 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.

**Question 1.4** Combien peut-il y avoir de stars dans le groupe ?

**Question 1.5** Ecrire le meilleur algorithme que vous pouvez et donner sa complexité en nombre de questions (on peut y arriver en  $\mathcal{O}(n)$  questions).

**Question 1.6** 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  $3n - \lfloor \log_2(n) \rfloor - 3$ ).

**Exercice 1.7.3.** *Bricolage*

Dans une boîte 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.

**Question 1.7** Ecrire un algorithme simple en  $\mathcal{O}(n^2)$  essais qui appareille chaque écrou avec son boulon.

**Question 1.8** 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.

**Question 1.9** 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.

**Exercice 1.7.4.** *Maximum de  $n$  entiers*

**Question 1.10** Écrire un algorithme (naïf!) qui calcule le maximum de  $n$  entiers. Quelle est la complexité (en nombre de comparaisons effectuées, en nombre d'affectations effectuées, dans le pire des cas, le meilleur) ?

**Question 1.11** Quelle est la complexité en moyenne en nombre d'affectations effectuées, en nombre de comparaisons effectuées ?

*Indications pour le calcul des affectations en moyenne :* soit  $P_{n,k}$  le nombre de permutations  $\sigma$  de  $\{1, \dots, n\}$  telles que sur la donnée  $T[1] = \sigma(1), \dots, T[n] = \sigma(n)$  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.

**Question 1.12** L'algorithme que vous avez proposé est-il optimal pour la complexité en nombre de comparaisons dans le pire des cas ?

## 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 ?<sup>1</sup> 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 :

<i>Classique</i>	<i>Strassen</i>
$Mult(2) = 8$	$Mult(2) = 7$
$Add(2) = 4$	$Add(2) = 18$

On a gagné une multiplication, mais en perdant 14 additions ; donc, pour des matrices  $2 \times 2$ , 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.

---

<sup>1</sup>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 :

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

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 :

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

(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 :

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

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,  $n = 2m$

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

Soient :

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

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.

$$\begin{cases} M(1) = 1 \\ M(n) = 3 \times M(n/2) \end{cases} \implies M(n) = 3^s = n^{\log_2(3)} \approx n^{1.58}$$

$$\begin{cases} A(1) = 0 \\ A(n) = \underbrace{3 \times A(n/2)}_{\text{appels récursifs}} + \underbrace{2 \times n/2}_{\text{pour avoir } R_3} + \underbrace{2 \times (n - 1)}_{\text{pour avoir } R_3 - R_2 - R_1} + \underbrace{(n - 2)}_{\text{construction de } R} \end{cases}$$

En effet, pour la construction de  $R$  :

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

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 :

$$\begin{cases} A(1) = 0 \\ A(n) = 3 \times A(n/2) + 4 \times n - 4 \end{cases} \implies A(n) = 6n^{\log_2(3)} - 8n + 2$$

(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 Fourier rapide (FFT).

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

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 régner : 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 :

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

	a	b	$\alpha$	c
Strassen	7	2	2	18/4
Polynomes	3	2	1	4

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

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

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

et :

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

On distingue alors plusieurs cas :

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

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  $n/2$  ?

$$\left( \begin{array}{|c|c|c|} \hline & & \\ \hline & & \\ \hline & & \\ \hline \end{array} \right) \times \left( \begin{array}{|c|c|c|} \hline & & \\ \hline & & \\ \hline & & \\ \hline \end{array} \right)$$

On a :

a	b	$\alpha$	c
a	3	2	

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

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

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

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

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  $r_i$

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

### 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 :

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

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

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

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

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

<i>suite</i>	<i>annulateur</i>
$\{c\}$	$E - 1$
$\{Q_k(n)\}$	$(E - 1)^{k+1}$
$\{c^n\}$	$E - c$
$\{c^n \times Q_k(n)\}$	$(E - c)^{k+1}$

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

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

(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  $A_s = A(2^s)$

$$\begin{aligned} A_{s+1} &= 7 \times A_s + 18 \times \frac{(2^{s+1})^2}{4} = 7 \times A_s + 18 \times 4^s \\ (E - 4) \underbrace{(E - 7)\{A_s\}}_{A_{s+1} - 7A_s = 18 \times 4^s} &= \bar{0} \\ \Rightarrow A_s &= k_1 \times 7^s + k_2 \times 4^s \end{aligned}$$

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 :  $(E - 1)(E - 2)(E - 3)\{A_s\} = \bar{0}$

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

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 :
  - $O(n^2) \leq \frac{M(n)}{I(n)} \leq O(n^3)$
  - $M(n)$  et  $I(n)$  croissants

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

*Démonstration.* 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

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

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

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

Il suffit de construire :

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

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

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

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

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

□

## 2.6 Exercices

### Exercice 2.6.1. Matrice 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  $2 \leq i, j \leq n$ .

**Question 2.1** La somme de deux matrices de Tœplitz est-elle une matrice de Tœplitz ? Et le produit ?

**Question 2.2** Trouver un moyen d'additionner deux matrices de Tœplitz en  $\mathcal{O}(n)$ .

**Question 2.3** 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 ?

**Exercice 2.6.2.** *Plus grand et deuxième plus grand de  $n$  entiers*

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. On suppose que  $n$  est une puissance de 2, et on s'intéresse à la complexité dans le pire des cas.

**Question 2.4 Algorithme naïf**

Écrire un algorithme calculant le cardinal  $c_x$  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 ?

**Question 2.5 Diviser pour régner** Donner un autre algorithme récursif basé sur un découpage de  $E$  en deux tableaux de même taille. Quelle est sa complexité ?

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  $E$ .

**Question 2.6** Donner un algorithme récursif possédant cette propriété. Quelle est sa complexité ?

**Question 2.7** En déduire un algorithme efficace vérifiant si  $E$  possède un élément majoritaire.

**Algorithme des balles -** 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.

**Question 2.8** 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 posées 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.

**Question 2.9** (*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.

**Question 2.10** (*Complexité*) Donner la complexité dans le pire des cas en nombre de comparaisons de couleurs des balles.

**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  $\text{couleur}(x) = \text{couleur}(y)$ ?

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  $d = B + t$ .
  - $d > m$  : cela signifie que les balles sont dans des troupeaux singletons. La réponse est non et les balles deviennent un nouveau binôme.
  - $d = m$  : la réponse est oui et les troupeaux de  $x$  et  $y$  fusionnent.

**Question 2.11** 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.

**Question 2.12** Montrer qu'à tout moment les deux coloriage 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.

**Question 2.13** Montrer que si un algorithme correct s'arrête alors l'arène ne contient qu'une seule composante connexe qui est un troupeau de taille  $m$ .

**Question 2.14** Montrer qu'à tout moment le nombre de comparaisons inégales effectuées par l'algorithme est d'au moins  $2g + B$  et que le nombre de comparaisons égales est d'au moins  $t - T$ .

**Question 2.15** Considérons un algorithme qui résout la majorité. Montrer qu'il existe une donnée pour laquelle l'algorithme effectue au moins  $2(n - m) = 2\lfloor n/2 \rfloor - 1$  comparaisons d'inégalité et au moins  $\lfloor n/2 \rfloor$  comparaisons d'égalité, et donc au moins au total  $3\lfloor n/2 \rfloor - 2$  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 Darté et Vaudenay [3].



## 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 à 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  $(v_1, v_2, \dots, v_n)$ . Pour cela on introduit la fonction  $z$  telle que :

- $z(T, i)$  = nombre minimum de pièces choisies parmi les  $i$  premières  $(v_1, \dots, v_i)$  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  $T = 0$  ou  $i = 0$ , et  $z(T, i) = +\infty$  pour  $T < 0$ .

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  $1, B, B^2, B^3, \dots$  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  $c_1, \dots, c_n$  et pour poids (ou volume)  $w_1, \dots, w_n$ . 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 glotonne 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.

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

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

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

avec les conditions initiales  $C(v, i) = 0$  si  $v = 0$  ou  $i = 0$ , et  $C(v, i) = -\infty$  si  $v < 0$ .

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.

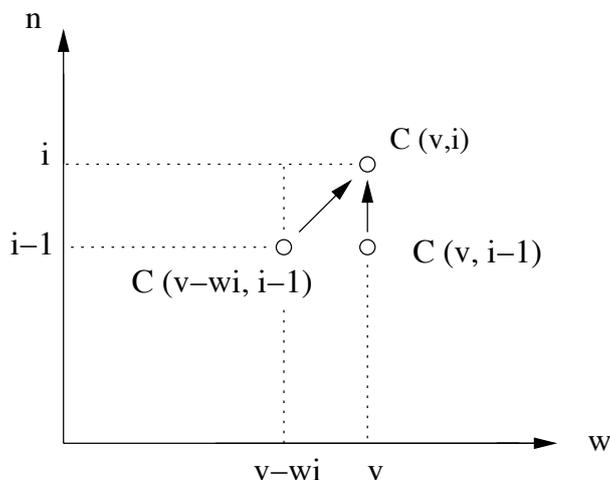


FIG. 3.1: Attention au respect des dépendances !

**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  $A_1 \times A_2 \times \dots \times A_n$ . 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<sup>1</sup>). 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.

<sup>1</sup>En cas de besoin urgent de compiler une référence sur les séries, se référer à Flajolet 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 cherchons à calculer  $A_1 \times \dots \times A_n$ , cherchons donc le coût optimal du calcul de  $A_i \times \dots \times A_j$  (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  $(A_i, \dots, A_j)$  est de couper à un endroit  $k$  :

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

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  $C(k+1, j)$

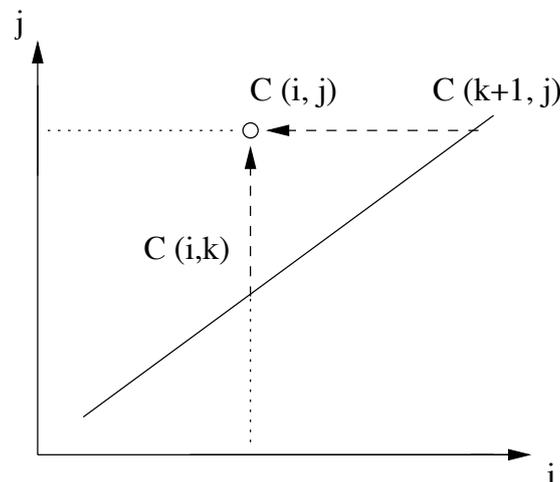


FIG. 3.2: L'initialisation se fait sur  $C(i, i+1) = P_{i-1} * P_i * P_{i+1}$  et  $C(i, i) = 0$  sur la diagonale

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

$$A = a_1 \dots a_n \quad B = b_1 \dots b_m$$

On définit une sous-chaîne de  $A$  comme étant  $a_{i_1} a_{i_2} \dots a_{i_k}$ .

On recherche les sous-chaînes communes à  $A$  et  $B$ , et plus particulièrement la Plus Longue Sous-Chaîne Commune ou *PLSCC*.

*Exemple :*

$$A = aabaababaa, \quad B = ababaaabb$$

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  $p(i, j)$ .

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

*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*  
 $A = a b c b d a b$   
 $B = b d c a b a$

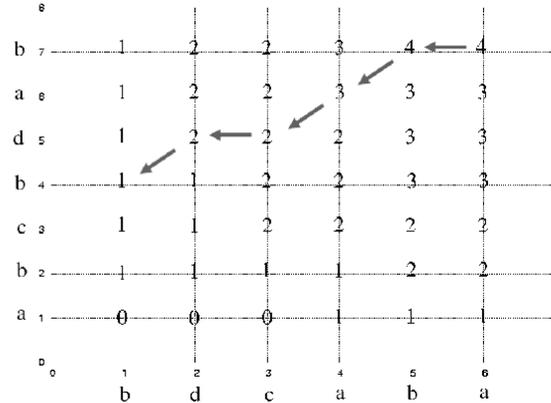


FIG. 3.3: La PLSCC est *bdab*.

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  $s_1, \dots, s_m$ , respectivement, à  $n$  skieurs ( $m \geq n$ ) de taille  $h_1, \dots, h_n$ , respectivement, via une fonction (injective)  $f : \{1, \dots, n\} \rightarrow \{1, \dots, m\}$ ,  $f$  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  $s_1, \dots, s_m$  et  $h_1, \dots, h_n$  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  $f : \{1, \dots, i\} \rightarrow \{1, \dots, j\}$  (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).

*Démonstration.* 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

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

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  $f : \{1, \dots, n\} \rightarrow S$  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  $s_{j_1}, \dots, s_{j_n}$  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.

**Question 3.1** Montrer qu'une triangulation d'un polygone à  $n$  côtés a  $(n - 3)$  cordes et  $(n - 2)$  triangles.

**Question 3.2** Le problème est celui de la triangulation optimale de polygones. On part d'un polygone convexe  $P = \langle v_0, \dots, v_n \rangle$ , où  $v_0, \dots, v_n$  sont les sommets du polygone donnés dans l'ordre direct, et d'une fonction de pondération  $w$  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  $\langle v_{i-1}, \dots, v_j \rangle$ , avec  $t[i, i] = 0$  pour tout  $1 \leq i \leq n$ .

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

**Question 3.3** 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.

**Question 3.4** Si le poids d'un triangle est égal à son aire, que pensez-vous de l'algorithme que vous avez proposé?

### 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.

**Question 3.5** Proposer un algorithme efficace pour construire une tour de hauteur maximale.

### Exercice 3.4.3. Plus grand carré de 1

#### Question 3.6

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  $i, j, I \leq i \leq I+K-1, J \leq j \leq J+K-1, A[i, j] = 1$ ).

**Question 3.7** Quelle est sa complexité?

### Exercice 3.4.4. Arbre binaire de recherche optimal

Un *arbre binaire de recherche* est une structure de données permettant de stocker un ensemble de clés ordonnées  $x_1 < x_2 < \dots < x_n$ , 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  $x$ .

La figure 3.4 représente un arbre binaire de recherche pour les clés  $x_1 < x_2 < x_3 < x_4 < x_5 < x_6 < x_7 < x_8 < x_9$ .

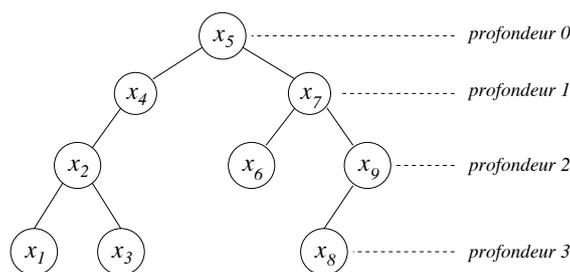


FIG. 3.4: Exemple d'arbre binaire de recherche.

Les requêtes 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.

**Question 3.8** 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 accéder 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 requêtes.

**Question 3.9** 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  $c_1, c_2, \dots, c_n$  ( $c_i$  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  $x_i < \dots < x_j$  accédées respectivement  $c_i, \dots, c_j$  fois.

### 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  $l_1, l_2, \dots, l_n$  (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 une espace<sup>2</sup> 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.

**Question 3.10** 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 ?

**Question 3.11** Donner un algorithme de programmation dynamique résolvant le problème. Analyser sa complexité en temps et en espace.

**Question 3.12** 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 ?

**Question 3.13** (*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 ?

## 3.5 Références bibliographiques

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

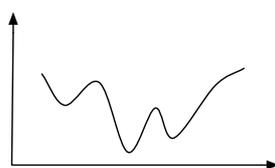
---

<sup>2</sup>En typographie *espace* est un mot féminin.

## 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.

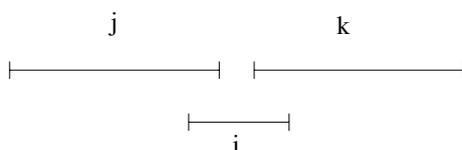


FIG. 4.1: L’essai 1 ne conduit pas à l’optimal.

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.

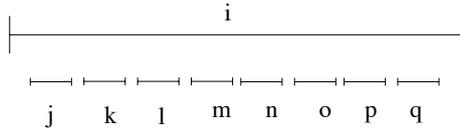


FIG. 4.2: L'essai 2 ne conduit pas à l'optimal.

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  $m$ .

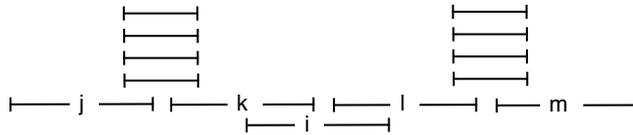


FIG. 4.3: L'essai 3 ne conduit pas à l'optimal.

#### Essai 4

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

**Théorème 4.** *Cet algorithme glouton est optimal.*

*Démonstration.* 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  $O$ ,

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

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.  $\square$

## 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 meilleure 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  $c : V \rightarrow \{1..K\}$  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.*

*Démonstration.*

$\Rightarrow$  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.

$\Leftarrow$  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  $X_{n+1} = \bigcup_{y \in X_n} \{\text{fils de } y \text{ dans l'ordre de parcours}\}$ .

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

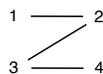
Pour colorier un graphe général :

- 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.



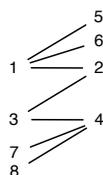
### 4.3.2 Algorithme glouton 2

- 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.

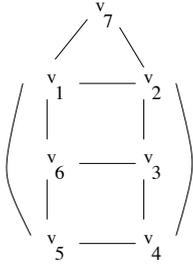


### 4.3.4 Algorithme de Brélaz

L'idée intuitive est de colorier en priorité les sommets ayant beaucoup de voisins 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  $v$ .

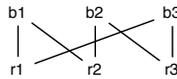
Il s'agit d'un glouton sur (degré-couleur, degré), appelé algorithme de Brélaz. Considérons l'exemple ci-dessous :



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.

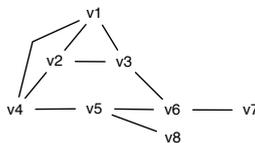
**Théorème 6.** *L'algorithme de Brelaz est optimal sur les graphes bipartis, i.e. réussit toujours à les colorier avec deux couleurs.*

*Démonstration.* 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.  $\square$

**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 :  $(A, B \in \mathcal{I}, |A| < |B|) \Rightarrow \exists x \in B \setminus A \text{ tq } A \cup \{x\} \in \mathcal{I}$

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

#### Exemples

1. Les familles libres d'un espace vectoriel
2. Les forêts<sup>1</sup> d'un graphe.

Soit  $G = (V, E)$  un graphe,  $|V| = n$ , on définit alors  $S = E$  et  $\mathcal{I} = \{A \subset E / A \text{ sans cycles}\}$ , 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.

---

<sup>1</sup>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* :

$$x \in S \mapsto w(x) \in \mathbb{N}; \quad X \subset S, \quad w(X) = \sum_{x \in X} w(x)$$

**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.*

*Démonstration.* 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  $A = B \setminus \{b_i\} \cup \{s_k\}$ , 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  $S' = S \setminus \{s_k\}$  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 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 on les ajoute à l'ensemble courant. Bien sûr, il faudrait discuter d'une structure de données adaptée 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  $T_1 \dots T_n$  de durée 1. Deadlines associées :  $d_1, \dots, d_n$ . Pénalités associées si le deadline est dépassé :  $w_1, \dots, w_n$ .

*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.

---

<sup>2</sup>  $\emptyset$  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 le 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.  $\forall t = 1, 2, \dots, n, N_t(A) \leq t$ .
3. *Ordre canonique  $\rightarrow$  pas de tâches en retard.*

*Démonstration.*

- $1 \Rightarrow 2$  : Par contraposée : supposons que  $N_t(A) > t$ , 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.*

*Démonstration.*

- Hérité : trivial
- Echange : Soit  $A, B$  indépendants tels que  $|A| < |B|$ . Existe-t-il  $x$  appartenant à  $B$  tel que  $A \cup \{x\}$  soit indépendant ?

Idee : 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

**Question 4.1** Etant donné un ensemble  $\{x_1, \dots, x_n\}$  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.

### Exercice 4.6.2. Codage de Huffman

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

**Question 4.2** Le décodage d'un codage de longueur fixe est unique. Montrer qu'il en est de même pour un codage préfixe.

**Question 4.3** Représenter un codage préfixe par un arbre binaire dont les feuilles sont les lettres de l'alphabet.

**Question 4.4** 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'occurrences 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.

**Question 4.5** 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  $\Sigma$ , 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  $\Sigma'$ , 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  $\Sigma$  - *Propriété de sous-structure optimale*.

**Question 4.6** 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  $f(g) = 5$ .

## 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  $\text{TF}(1,n) \rightarrow$  appels récursifs de  $\text{TF}(1, \lfloor n/2 \rfloor)$  et  $\text{TF}(\lfloor n/2 \rfloor + 1, n)$  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 :

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

avec  $\text{comp}_{TF}(1) = 0$ .

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

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

Il faut distinguer les cas  $n = 2^r$  /  $n = 2^r + 1$ , et dans le cas où  $n = 2^r + 1$  distinguer les cas  $n = 2^r + 1$  /  $n \neq 2^r + 1$  pour prouver que cette formule pour  $\text{comp}_{TF}(n)$  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,  $A[\text{père}(i)] \leq A[i]$ .

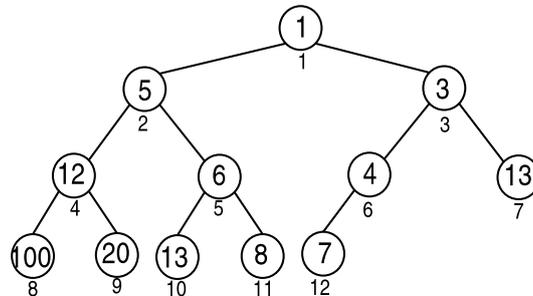


FIG. 5.1: Exemple de tas

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  $a[1], \dots, a[n]$  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  $Z$  :  $\underbrace{\dots}_{k-1} \leq Z \leq \underbrace{\dots}_{n-k}$ ;
- 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  $n \leq n' \leq n + 4$ . 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 :

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

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

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

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

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

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

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

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

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

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

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.  $\square$

**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.

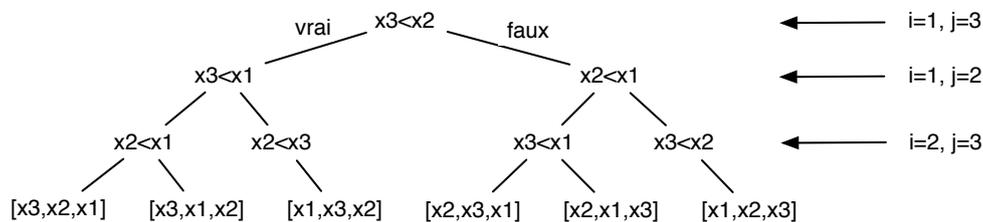
On a vu que la complexité de tri fusion ne dépend pas des données et vaut  $comp_{TF}(n) = n \lceil \log_2(n) \rceil - 2^{\lceil \log_2(n) \rceil} + 1$ , ce qui tend vers  $n \log_2(n)$ , montrant ainsi que la borne peut être atteinte asymptotiquement.

### 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.

*Démonstration.* 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  $n!$ .  $\square$

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

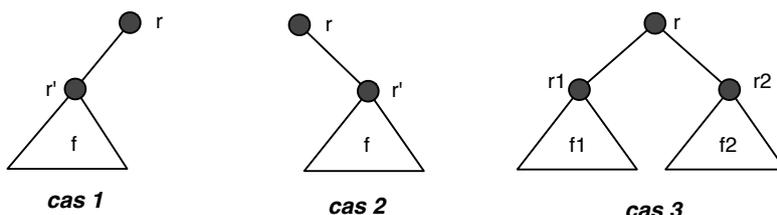
applique le résultat pour  $f = n!$  ce qui prouve le premier théorème, en utilisant la formule de Stirling pour estimer  $\log_2(n!) = O(n \log(n))$ .

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  $\log_2(f)$ .*

*Démonstration.* 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  $F = F_1 \cup F_2$  l'ensemble des feuilles.

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

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

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

Il y a 2 manières de démontrer la dernière inégalité :

- On écrit  $f_2 = f - f_1$ . Soit  $g(f_1) = 1 + \frac{\log_2(f_1) * f_1}{f} + \frac{\log_2(f - f_1) * (f - f_1)}{f}$ . 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.  $\square$

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

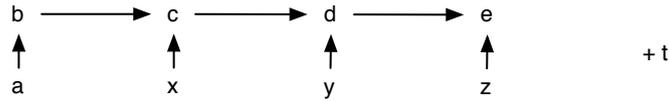
Une idée importante est l'insertion par dichotomie. Insérer un élément dans une suite triée de  $k$  éléments coûte  $r$  comparaisons au pire si  $k \leq 2^r - 1$ . Il est donc plus *avantageux* d'insérer dans une chaîne de taille 3 que de taille 2, et dans une chaîne de taille 7 que de taille 4 à 6,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



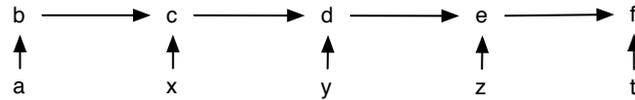
on forme 2 paires de 2, ( $a \rightarrow b$ ) et ( $c \rightarrow d$ ), on compare les deux plus grands pour obtenir par exemple ( $a \rightarrow b \rightarrow d$ ), puis on insère  $c$  par dichotomie (2 comparaisons) : coût  $2+1+2=5$ .

- 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 :

**Question 5.1** 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  $u, v \in S$ .

**Question 5.2** 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  $u, v \in S, u \neq v$ .

**Question 5.3** 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  $x + y = m$ .

**Question 5.4** Proposer un algorithme en  $\mathcal{O}(n)$  pour trouver deux éléments  $x$  et  $y$  de  $S$  tels que  $|x - y| \leq \frac{1}{n-1}(\max(S) - \min(S))$ .

### Exercice 5.5.2. *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 ?

**Exercice 5.5.3. Médian Pondéré**

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

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

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

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

**Question 5.5** Démontrer que le médian de  $x_1, \dots, x_n$  est le médian pondéré des  $x_i$  affectés des poids  $p_i = 1/n$  pour  $i$  compris entre 1 et  $n$ .

**Question 5.6** Comment calculer *simplement* le médian pondéré de  $n$  éléments. Donner la complexité en comparaisons et en additions.

**Question 5.7** Comment calculer le médian pondéré en temps linéaire dans le pire des cas.

**Exercice 5.5.4. Tas faible et 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.

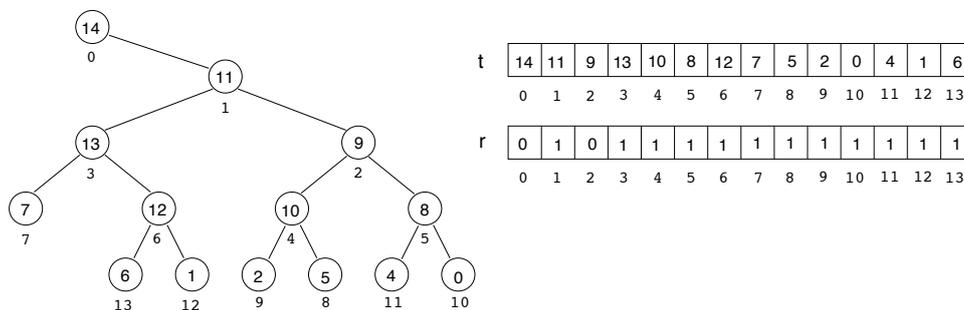


FIG. 5.2: Exemple de tas faible et un codage de cet arbre.

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 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 5.2 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 5.2, les petits-fils de la racine sont les éléments 11, 13 et 7. On note  $\text{Pere}(i)$  l'indice du père du nœud d'indice  $i$ , et la fonction  $\text{Gpere}(i)$  (pour *Grand-père*) définie par  $\text{Gpere}(\text{Pere}(i))$  si  $i$  est le fils gauche de  $\text{Pere}(i)$  et par  $\text{Pere}(i)$  si  $i$  est le fils droit de  $\text{Pere}(i)$ . Par exemple avec le codage de la figure 5.2 le nœud d'indice 9 (de valeur  $t[9] = 2$ ) a pour grand-père  $\text{Gpere}(9) = 1$  (indice du nœud avec la valeur  $t[1] = 11$ ).

**Question 5.8** 1 - Pour le codage de l'arbre avec les tableaux  $t$  et  $r$ , écrire précisément la fonction  $\text{Gpere}(i)$  qui calcule l'indice du grand-père de l'élément d'indice  $i$  dans  $t$ .

Soit  $\text{echange}(i, j)$  une fonction qui échange les valeurs de  $t[i]$  et  $t[j]$ . Soit  $\text{fusion}(i, j)$  la fonction définie par : si  $(t[i] > t[j])$  alors  $\text{echange}(i, j)$  et  $r[i] \leftarrow 1 - r[i]$ .

**Question 5.9** 2 - Etant donné un tableau de  $n$  valeurs  $t[0..n-1]$ , on lui associe le tableau  $r = [0, 0, \dots, 0]$  où tous les bits sont à 0, et on effectue la fonction  $\text{ConstruitTas}(t, r)$  définie par :

pour  $i$  de  $n - 1$  à 1, faire  $\text{fusion}(i, \text{Gpere}(i))$

Montrer que pendant le déroulement de la boucle, après avoir effectué  $\text{fusion}(i, \text{Gpere}(i))$ , 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  $\text{Gpere}(j)$ . En déduire qu'en fin de boucle, on obtient un tas faible. Combien de comparaisons a-t-on effectué ?

**Question 5.10** 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.

**Question 5.11** 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.

### Exercice 5.5.5. Tri par tas

Le tri par tas est réalisé en deux étapes : insertion des  $n$  éléments dans le tas, puis déconstruction du tas en extrayant à chaque fois le plus petit élément.

La construction du tas se fait en  $O(n \log n)$ , et la suppression des éléments du tas se fait également en  $O(n \log n)$ .

Il est possible d'améliorer la complexité de la construction du tas, elle peut se faire en  $O(n)$  (cependant la suppression des  $n$  éléments du tas reste en  $O(n \log n)$ ). Proposez un algorithme pour construire le tas en temps linéaire.

## 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  $e_1, e_2, \dots, e_k$  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  $W \subset V$ ,  $F \subset E$ , 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

*Démonstration.* 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 on 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  $n - 1$  comparaisons. 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

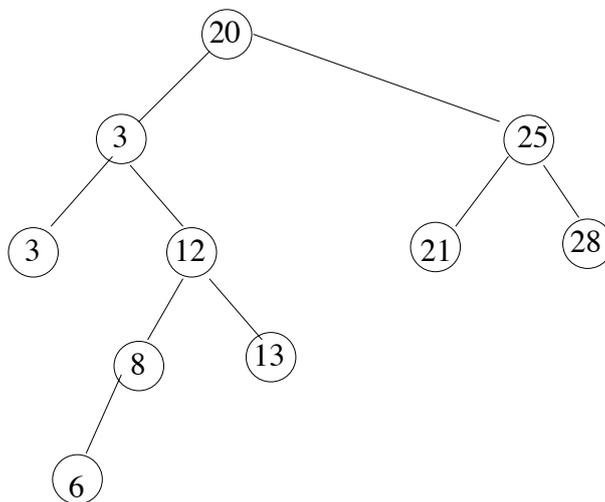
Il y a trois parcours classiques en profondeur :

**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 :

$$\text{valeur}[\text{sous-arbre gauche}] \leq \text{valeur}[\text{nœud}] < \text{valeur}[\text{sous-arbre droit}]$$

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

- $M_{i,j} = \begin{cases} 0 & \text{si } (i, j) \notin E \\ 1 & \text{si } (i, j) \in E \end{cases}$
- $M$  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 successeurs : 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  $xRy$ . On définit par récurrence l'élevation à la puissance  $R^i$  :

$$\begin{cases} R^i = R \cdot R^{i-1} \\ R^0 = id \end{cases}$$

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  $R^p = R^q$

*Démonstration.* 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.  $\square$

**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.**  $xR^+y \Leftrightarrow$

$$\exists p > 0 \mid \exists x_1 \dots x_p \in E \text{ avec } x_1 = x, x_p = y \text{ et } x_i R x_{i+1} \text{ pour } 1 \leq i \leq p$$

$$\text{ce qui peut aussi s'écrire : } R^+ = \bigcup_{p \geq 1} R^p$$

*Démonstration.* 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  $R^+$ .

$\square$

**Proposition 3.**

$$(1) \quad R^n \subseteq \bigcup_{0 \leq i \leq n-1} R^i \quad \text{et} \quad (2) \quad \forall p \geq 1, R^p \subseteq \bigcup_{1 \leq i \leq n} R^i$$

*Démonstration.* 1. Soit  $S = \bigcup_{0 \leq i \leq n-1} R^i$  et supposons  $aR^nb$ , on va montrer qu'alors  $aSb$ .

Il existe  $x_0, \dots, x_n \in E$ ,  $x_0 = a$ ,  $x_n = b$ ,  $x_i R x_{i+1}$  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  $aR^0b$  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  $p$  :

–  $p = 1$  :  $R^1 \subseteq \bigcup_{1 \leq i \leq n} R^i$

–  $p \rightarrow p + 1$  :

on vient de montrer que  $R^n \subseteq \bigcup_{0 \leq i \leq n-1} R^i$ , donc  $\bigcup_{0 \leq i \leq n} R^i \subseteq \bigcup_{0 \leq i \leq n-1} R^i$

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 réflexive 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  $M_{ij}^p$  est égal au nombre de chemins de longueur  $p$  de  $v_i$  à  $v_j$
- Preuve par récurrence :
  - Vrai par définition pour  $p = 1$
  - $M_{ij}^p = \sum_{k=1}^n M_{ik}^{p-1} \cdot M_{kj}$ , 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

$$(i, j) \in A^+ \Leftrightarrow \text{il y a un chemin non vide de } v_i \text{ à } v_j \text{ dans } G$$

- $M^+ = M + M^2 + \dots + M^n$  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 permettra d'étendre facilement l'algorithme pour traiter des plus courts chemins dans un 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  $M^+ = A^{(n)}$ .

*Démonstration.* 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  $p \rightsquigarrow j$  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

début

$A \leftarrow M$ ;

pour  $p$  de 1 à  $n$  faire

pour  $i$  de 1 à  $n$  faire

pour  $j$  de 1 à  $n$  faire

$A_{ij} \leftarrow A_{ij} + A_{ip}A_{pj}$ ;

fin

fin

fin

fin

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 :

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  $A$  :  $A_{ij} \leftarrow A_{ij} + 0A_{pj}$ , 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) {
```

début

```
A ← M;
pour p de 1 à n faire
  pour i de 1 à n faire
    si  $A_{ip}$  alors
      pour j de 1 à n faire
        si  $A_{pj}$  alors
           $A_{ij} ← 1$ ;
        fin
      fin
    fin
  fin
fin
```

```
    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  $w : E \rightarrow \mathbb{R}$ , 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.

*Démonstration.* 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  $v$ .  $\square$

### 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,
- $P_{ij} = \text{NULL}$  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 :

$$A_{ij}^{(p)} = A_{ij}^{(p-1)} \oplus (A_{ip}^{(p-1)} \otimes (A_{pp}^{(p-1)})^* \otimes A_{pj}^{(p-1)})$$

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)

début

$A \leftarrow M;$

**pour**  $p$  **de** 1 **à**  $n$  **faire**

$A_{pp}^{(p)} \leftarrow (A_{pp}^{(p-1)})^*;$

**pour**  $i$  **de** 1 **à**  $n$ ,  $i \neq p$  **faire**

$A_{ip}^{(p)} = A_{ip}^{(p-1)} \otimes A_{pp}^{(p)};$

**pour**  $j$  **de** 1 **à**  $n$ ,  $j \neq p$  **faire**

$A_{ij}^{(p)} = A_{ij}^{(p-1)} \oplus (A_{ip}^{(p)} \otimes A_{pj}^{(p-1)});$

**fin**

$A_{pj}^{(p)} = A_{pp}^{(p)} \otimes A_{pj}^{(p-1)};$

**fin**

**fin**

**fin**

Pour l'algorithme de Roy-Warshall, on utilise :

$$\left| \begin{array}{l} H = \{0, 1\} \\ \oplus = \text{ou logique} \\ \otimes = \text{et logique} \end{array} \right. \Rightarrow \forall a, a^* = 1$$

Pour l'algorithme de Warshall-Floyd avec gestion des circuits absorbants, on utilise :

$$\left| \begin{array}{l} H = \mathbb{R} \cup \{+\infty, -\infty\} \\ \oplus = \min \\ \otimes = + \end{array} \right. \Rightarrow a^* = \begin{cases} 0 & \text{si } a > 0 \\ -\infty & \text{sinon} \end{cases}$$

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  $V = 1, 2, \dots, n$  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

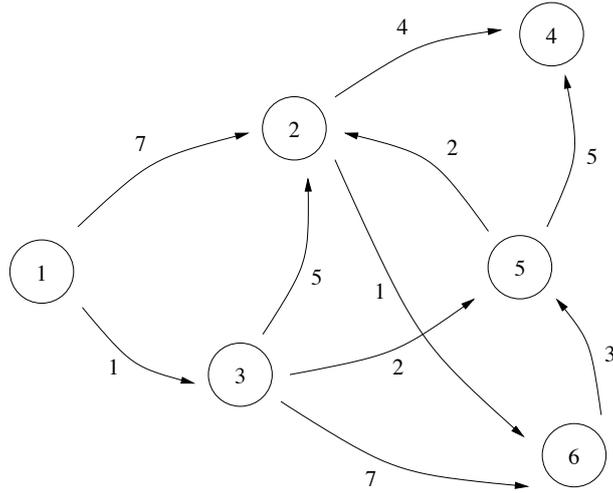
```

début
   $S \leftarrow \{1\}$ ;
  si  $i \in \Gamma(1)$  alors
     $d(i) = w(1, i)$ ;
  sinon
     $d(i) = +\infty$ 
  fin
  tant que  $S \neq V$  faire
    Soit  $j$  tel que  $j \notin S$  et  $d(j) = \min_{i \in S} d(i)$ ;
     $S \leftarrow S \cup \{j\}$ ;
    pour  $k \in \Gamma(j)$  vérifiant  $k \notin S$  faire
       $d(i) \leftarrow \min\{d(i), d(k) + w(k, i)\}$ ;
    fin
  fin
fin

```

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 sommet  $j$  à  $S$ , on met à jour les distances des voisins de  $j$ .

Voici un exemple :



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	<b>0</b>	$\infty$	$\infty$	$\infty$	$\infty$	$\infty$
1	.	<b>7</b>	<b>1</b>	$\infty$	$\infty$	$\infty$
2	.	<b>6</b>	.	$\infty$	<b>3</b>	8
3	.	5	.	8	.	8
4	.	<b>5</b>	.	8	.	8
5	.	.	.	8	.	<b>6</b>
6	.	.	.	<b>8</b>	.	.
$\delta(1, i)$	0	5	1	8	3	6

*Démonstration.* Il faut procéder rigoureusement, car il est très facile de donner une preuve fautive 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 :

$$\left| \begin{array}{l} \text{si } i \in S, \quad d(i) = d^*(i) \\ \text{si } i \notin S, \quad d(i) = \min_{\substack{k \in S \\ i \in \Gamma(k)}} \{d(k) + w(k, i)\}, \quad \text{ou } +\infty \text{ si indéfini} \end{array} \right.$$

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  $\mu$  un plus court chemin de la source au sommet  $j$ .
2. Décomposons  $\mu$  en  $\mu_1 + \mu_2$  (concaténation de chemin), avec  $\mu_1$  chemin de 1 à  $k$  où  $k$  est le premier sommet de  $\mu$  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 arête 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  $O((n + mp) \log n)$ .

Ainsi, pour 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 Fibonacci qui permet de décroître des valeurs dans le tas en une complexité amortie  $O(1)$ . Avec cette structure, l'algorithme de Dijkstra 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 arborescence 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 ("sommets " + 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) ← WHITE
        p(u) ← NIL
endfor
time ← 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.**  $G \text{ DAG} \Leftrightarrow \text{DFS}(G) \text{ ne donne pas d'arcs de type Back}$

## 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  $v$ .

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 événementiel 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 :

```

début
  pour tous les sommet  $u \in S$  /*Initialisation */
  faire
     $couleur[u] \leftarrow BLANC$ ;
     $\pi[u] \leftarrow NIL$ ;
  fin
   $temps \leftarrow 0$ ;
  pour tous les somme  $u \in S$  /*Exploration */
  faire
    si  $couleur[u] = BLANC$  alors
      Visiter_PP( $u$ );
    fin
  fin
fin

```

**Algorithme 1** :  $PP(S, Adj[], temps)$

```

début
   $couleur[u] \leftarrow GRIS$ ;
   $d[u] \leftarrow temps$ ;
   $temps \leftarrow temps + 1$ ;
  pour tous les  $v \in Adj[u]$  faire
    si  $couleur[v] = BLANC$  alors
       $\pi[v] \leftarrow u$ ;
      Visiter_PP( $v$ );
    fin
  fin
   $couleur[u] \leftarrow NOIR$ ;
   $f[u] \leftarrow temps$ ;
   $temps \leftarrow temps + 1$ ;
fin

```

**Algorithme 2** :  $Visiter\_PP(u)$

**Question 6.1** 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 ?

**Question 6.2** 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  $v$ .

### 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.

**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.

**Question 6.3** Donner une condition nécessaire sur le graphe pour que la formule soit satisfiable.

On va montrer qu'il s'agit d'une condition suffisante.

**Question 6.4** Montrer qu'une valuation  $v$  qui satisfait la formule  $F$  est constante sur une composante fortement connexe du graphe associé  $G$ .

### 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  $G$ .

**Question 6.5** Montrer que  $H$  est sans circuit.

### 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.

**Question 6.6** Montrer que l'ordre total ainsi obtenu étend bien l'ordre partiel.

**Question 6.7** Donner la complexité de la construction de  $H$  à partir de  $G$  et du tri topologique.

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.

**Question 6.8** Montrer que  $v$  est définie de manière cohérente et conclure. Quelle est la complexité de 2-SAT ?

### Exercice 6.10.3. L'algorithme de Bellman-Ford

L'algorithme de Bellman-Ford résout le problème des plus courts chemins avec origine unique dans le cas le plus général où les poids des arcs peuvent avoir des valeurs négatives. Étant donné un graphe orienté pondéré  $G = (V, E)$ , de fonction de poids  $w$ , et une origine  $s$ , l'algorithme retourne une valeur booléenne indiquant s'il existe un circuit de poids négatif accessible depuis  $s$ . S'il n'en existe pas, l'algorithme donne les plus courts chemins ainsi que leurs poids.

Les notations sont les suivantes :  $\pi[v]$  contient le prédécesseur de  $v$  sur le chemin (NIL s'il n'y en a pas),  $\delta(u, v)$  est le poids du plus court chemin de  $u$  vers  $v$  ( $\infty$  s'il n'existe pas),  $d[v]$  est une variable qui est une borne supérieure du poids du plus court chemin de  $s$  vers  $v$  (cf. question 12). On définit le poids d'un chemin  $p = \langle v_0, v_1, \dots, v_k \rangle$ , est  $w(p) = \sum_{i=1}^k w(v_{i-1}, v_i)$ .

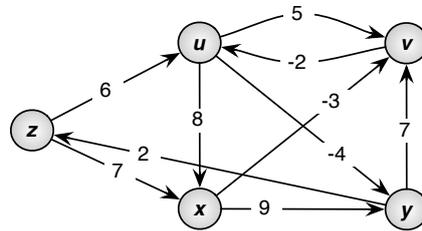
```

début
  pour tous les sommet  $v \in V$  /*Initialisation */
  faire
     $d[v] \leftarrow \infty, \pi[v] \leftarrow \text{NIL};$ 
  fin
   $d[s] \leftarrow 0;$ 
  pour  $i$  de 1 à  $|V| - 1$  faire
    pour tous les arc  $(u, v) \in E$  /*Relâchement de l'arc  $(u, v)$  */
    faire
      si  $d[v] > d[u] + w(u, v)$  alors
         $d[v] \leftarrow d[u] + w(u, v), \pi[v] \leftarrow u;$ 
      fin
    fin
  fin
  pour tous les arc  $(u, v) \in E$  /*Détection des circuits négatifs */
  faire
    si  $d[v] > d[u] + w(u, v)$  alors
      retourner faux;
    fin
  fin
  retourner vrai;
fin

```

**Algorithme 3** : Bellman-Ford( $G, w, s$ )

**Question 6.9** On considère le graphe ci-dessous. Faire tourner l'algorithme en prenant comme source le sommet  $z$ . Même question en changeant le poids de l'arc  $(y, v)$  à 4.



**Question 6.10** Quelle est la complexité de cet algorithme ?

**Question 6.11** Montrer qu'après le relâchement de l'arc  $(u, v)$ ,  $d[v] \leq d[u] + w(u, v)$ .

**Question 6.12 Propriété du majorant** : Montrer que  $d[v] \geq \delta(s, v)$  pour tout sommet  $v$  à tout instant de l'algorithme. Montrer que lorsque  $d[v]$  a atteint sa borne inférieure  $\delta(s, v)$ , il n'est plus jamais modifié.

**Question 6.13 Propriété de convergence** : Considérons un plus court chemin  $(s, \dots, u, v)$  pour un sommet  $v$  donné. Montrer que si  $d[u] = \delta(s, u)$  à un moment précédent le relâchement de  $(u, v)$  dans l'algorithme, alors on a toujours  $d[v] = \delta(s, v)$  après l'appel.

**Question 6.14** Montrer que, si  $G$  ne contient aucun circuit de poids négatif accessible depuis  $s$ , alors, à la fin de l'exécution de l'algorithme,  $d[v] = \delta(s, v)$  pour tout sommet  $v$  accessible depuis  $s$ .

**Question 6.15** Montrer que pour chaque sommet  $v$ , il existe un chemin de  $s$  vers  $v$  si et seulement si BELLMAN-FORD se termine avec  $d[v] < \infty$ .

Nous pouvons à présent démontrer la validité de l'algorithme.

**Question 6.16** Montrer que si  $G$  ne contient aucun circuit de poids négatif accessible depuis  $s$ , alors l'algorithme retourne VRAI, on a  $d[v] = \delta(s, v)$  pour tous les sommets  $v \in V$ , et le

sous-graphe  $G_\pi$  des sommets  $v$  dont  $\pi(v) \neq \text{NIL}$  et des arcs  $(\pi(v), v)$  est une arborescence des plus courts chemins de racine  $s$ . Montrer que si  $G$  contient un circuit de poids négatif accessible à partir de  $s$ , l'algorithme renvoie FAUX (on pourra raisonner par l'absurde).

#### Exercice 6.10.4. L'algorithme de Johnson

On s'intéresse au problème consistant à calculer les plus courts chemins pour tout couple de sommets du graphe. L'algorithme de Johnson a une bonne complexité pour les graphes creux (peu d'arêtes en comparaison du nombre de sommets). Il utilise les algorithmes de Dijkstra et de Bellman-Ford, et renvoie soit la matrice des poids des plus courts chemins, soit l'assertion que le graphe possède un circuit de poids négatif.

La technique utilisée consiste à se ramener au cas avec uniquement des poids positifs, puis de faire tourner l'algorithme de Dijkstra en partant de chaque sommet. Si les poids ne sont pas tous positifs, on les redéfinit pour pouvoir se ramener au premier cas.

Soit  $w$  la fonction des poids avant redéfinition. La nouvelle fonction de poids  $\hat{w}$  doit respecter deux propriétés :

1. Pour chaque couple de sommets  $u, v \in V$ , un plus court chemin de  $u$  à  $v$  en utilisant la fonction de poids  $w$  est également un plus court chemin pour la fonction  $\hat{w}$ .
2. Pour chaque arête  $(u, v)$ , le nouveau poids  $\hat{w}(u, v)$  n'est pas négatif.

Dans la suite, on note comme dans l'exercice précédent  $\delta$  pour la fonction des poids des plus courts chemins (avec  $w$ ), et  $\hat{\delta}$  représente la fonction des poids des plus courts chemins en utilisant la fonction de poids  $\hat{w}$ .

Voici l'algorithme de Dijkstra. Avec  $\forall u \in V, \text{adjacent}[u]$  la liste des sommets adjacents à  $u$ .  
**début**

```

pour tous les sommet  $v \in V$  /*Initialisation */
faire
     $d[v] \leftarrow \infty, \pi[v] \leftarrow \text{NIL};$ 
fin
 $d[s] \leftarrow 0;$ 
 $E \leftarrow \emptyset;$ 
 $F \leftarrow V[G];$ 
/*ensemble des sommets de  $G$  */
tant que  $F \neq \emptyset$  faire
     $u \leftarrow v | d[v] = \min\{d[x] | x \in F\};$ 
    /*on choisit le sommet avec la plus petite valeur de  $d$  */
     $F \leftarrow F - \{u\};$ 
     $E \leftarrow E \cup \{u\};$ 
    pour tous les sommet  $v \in \text{adjacent}[u]$  /*relâchement de l'arc  $(u, v)$  */
    faire
        si  $d[v] > d[u] + w(u, v)$  alors
             $d[v] \leftarrow d[u] + w(u, v), \pi[v] \leftarrow u;$ 
        fin
    fin
fin
fin

```

**Algorithme 4** : Dijkstra( $G, w, s$ )

#### Question 6.17 Propriété 1

Soit  $h : V \rightarrow \mathbb{R}$  une fonction quelconque faisant correspondre un réel à chaque sommet. Pour chaque arête  $(u, v) \in E$ , on définit  $\hat{w}(u, v) = w(u, v) + h(u) - h(v)$ .

Soit  $p = \langle v_0, v_1, \dots, v_k \rangle$  un chemin allant de  $v_0$  à  $v_k$ . Montrer que  $w(p) = \delta(v_0, v_k)$  si et seulement si  $\hat{w}(p) = \hat{\delta}(v_0, v_k)$ . De plus, montrer que  $G$  possède un circuit de poids négatif en utilisant  $w$  si et seulement si  $G$  possède un circuit de poids négatif en utilisant  $\hat{w}$ .

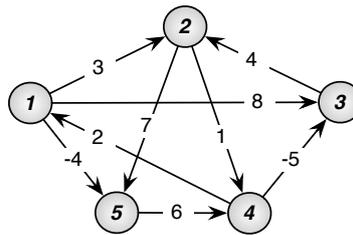
**Question 6.18 Propriété 2**

On désire maintenant définir  $\hat{w}$  vérifiant également la deuxième propriété (fonction non négative). On crée un nouveau graphe  $G' = (V', E')$  avec  $V' = V \cup \{s\}$  ( $s \notin V$ ) et  $E' = E \cup \{(s, v) : v \in V\}$ . La fonction de poids  $w$  est étendue de manière à ce que  $w(s, v) = 0$  pour chaque  $v \in V$ .

De plus, si  $G$  n'a pas de circuit de poids négatif, on définit  $h(v) = \delta(s, v)$  pour chaque  $v \in V'$ .

**6.18.1** A partir des indications précédentes, définir une nouvelle fonction de poids et montrer qu'elle vérifie les deux propriétés.

**6.18.2** Générer  $G'$  à partir du graphe  $G$  ci-dessous, et calculer les nouveaux poids.



**Question 6.19 Algorithme de Johnson**

Écrire l'algorithme de Johnson en s'inspirant des questions précédentes. Faire tourner l'algorithme sur le graphe ci-dessus. Justifier la correction de l'algorithme et calculer sa complexité. On rappelle que l'algorithme de Dijkstra implémenté par un tas de Fibonacci a une complexité en  $O(V \log V + E)$ .

**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 ...  
⇒ 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  $T[0], T[1], \dots, T[m-1]$
- 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  $h : X \rightarrow \{0, 1, \dots, m - 1\}$ 
  1. facile à calculer
  2. avec une distribution d'apparence aléatoire
- Collision si  $h(x) = h(y)$  pour  $x \neq y$  (inévitables si  $m < n$ )

### Méthode de construction

1. Interpréter les clés comme des entiers :

$$x = x_l \dots x_1 x_0 \longrightarrow \nu(x) = x_l B^l + \dots + x_1 B + x_0$$

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  $\nu(x) = 1999$ ,  $A \times \nu(x) = 1235.44994\dots$  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  $nom[i] \neq x$ , 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

$$h(x) + k \times h'(x) \bmod m \quad \text{pour } 0 \leq k \leq m - 1$$

doivent réaliser une permutation de  $\{0, 1, \dots, m - 1\}$ .

Par exemple prendre  $m$  premier et poser :

$$\begin{aligned} h(x) &= x \bmod m \\ h'(x) &= 1 + (x \bmod (m - 1)) \end{aligned}$$

### 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  $q_i = \Pr\{\text{au moins } i \text{ examens accèdent des places occupées}\}$

- $q_1 = \frac{n}{m}$ ,  $q_2 = \left(\frac{n}{m}\right) \left(\frac{n-1}{m-1}\right)$
- $q_i = \left(\frac{n}{m}\right) \left(\frac{n-1}{m-1}\right) \dots \left(\frac{n-i+1}{m-i+1}\right) \leq \left(\frac{n}{m}\right)^i = \alpha^i$
- $M \leq 1 + \alpha + \alpha^2 + \alpha^3 + \dots = \frac{1}{1-\alpha}$

### 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 opération !

### 8.1 Compteur

Etudions l'incrémentation d'un compteur à  $k$  bits :

$x = 0$ , for  $i = 1$  to  $n$  do  $x++$

Exemple : si  $k = 6$

```
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
```

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 :

$$n + \frac{n}{2} + \frac{n}{4} + \dots \leq 2n$$

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 :

$$\begin{array}{cccccccc} \dots & 0 & 1 & \dots & \dots & 1 & & f(i-1) \\ \dots & 1 & 0 & \dots & \dots & 0 & f(i) = f(i-1) - t(i-1) + 1 & \end{array}$$

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

$$c(i) = \begin{cases} 1 & \text{si } i \neq 2^k + 1 \\ 2^k + 1 & \text{sinon} \end{cases}$$

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 :  $x_1, x_2, \dots, x_n$  ; répartis en  $n$  classes d'équivalence :  $c_1, c_2, \dots, c_n$ . 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  $n \log(n)$ , 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  $n \log_2(n)$ .

#### 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 célèbre 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  $v_1, \dots, v_n$ , 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  $p \ll n^2$ . 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  $a_1, \dots, a_n$ .

**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.  $n^2 \cdot (\max(\log(a_i)))^3$  ?
2.  $n \cdot (\max a_i + 1)$  ?
3.  $n \cdot (\sum a_i)$  ?

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  $10^{6000} \gg \gg 22026 > (1.0001)^n$ . 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  $x_1, x_2, \dots, x_n$  et  $p$  clauses  $C_i$  :

$F = C_1 \wedge C_2 \wedge \dots \wedge C_p$  où  $C_i = x_{i_1}^* \vee x_{i_2}^* \vee \dots \vee x_{i_{f(i)}}^*$  et  $x^* = x$  ou  $\bar{x}$ .

Existe-t-il une instantiation 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.

Démonstration.

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  $x_1, \dots, x_n$

$p$  clauses  $C_1, \dots, C_p$  de longueurs  $f(1), \dots, f(p)$

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 :

$$x \vee a_i \vee b_i$$

$$x \vee \bar{a}_i \vee b_i$$

$$x \vee a_i \vee \bar{b}_i$$

$$x \vee \bar{a}_i \vee \bar{b}_i$$

- Si  $C_i$  a 2 variables  $x_1 \vee x_2$ , soit  $c_i$  une nouvelle variable, et on rajoute les clauses :

$$x_1 \vee x_2 \vee c_i$$

$$x_1 \vee x_2 \vee \bar{c}_i$$

- 3 variables : aucun changement

- $k$  variables,  $k > 3$ ,  $C_i = x_1 \vee x_2 \vee \dots \vee x_k$

Soit  $z_1^{(i)}, z_2^{(i)}, \dots, z_{k-3}^{(i)}$  ( $k - 3$ ) nouvelles variables. On rajoute les  $k - 2$  clauses :

$$x_1 \vee x_2 \vee z_1^{(i)}$$

$$x_3 \vee z_1^{(i)} \vee z_2^{(i)}$$

...

$$x_{k-2} \vee z_{k-4}^{(i)} \vee z_{k-3}^{(i)}$$

$$x_{k-1} \vee x_k \vee z_{k-3}^{(i)}$$

On construit l'instance  $I_2$  de 3-SAT comme étant l'ensemble des variables  $x_1, \dots, x_n$  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)  $\text{taille}(I_2)$  est linéaire en  $\text{taille}(I_1)$
- (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 :

$$\begin{aligned} x_i &\rightsquigarrow \text{inchangé} \\ a_i &\rightsquigarrow \text{VRAI} \\ b_i &\rightsquigarrow \text{FAUX} \\ c_i &\rightsquigarrow \text{VRAI} \end{aligned}$$

Si l'on considère une clause de  $I_1$  à  $k > 3$  variables,  $C_i = y_1 \vee \dots \vee y_k$ , 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  $z$ ) :

$$\begin{aligned} z_1, \dots, z_{j-2} &\rightsquigarrow \text{VRAI} \\ z_{j-1}, \dots, z_{k-3} &\rightsquigarrow \text{FAUX} \end{aligned}$$

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  $x_1, \dots, x_n$  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,  $C_i = y_1 \vee \dots \vee y_k$ . 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

*Démonstration.*

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

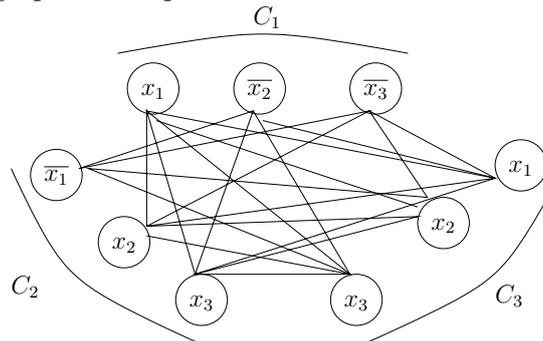
$$C_1 = x_1 \vee \bar{x}_2 \vee \bar{x}_3$$

$$C_2 = \bar{x}_1 \vee x_2 \vee x_3$$

$$C_3 = x_1 \vee x_2 \vee x_3$$

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  $\bar{x}_i$ ).

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  $\Rightarrow$  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  $\Rightarrow$  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  $v_{i_1}, \dots, v_{i_k}$  tels que  $\forall e \in E$ ,  $e$  est adjacente à l'un des  $v_{i_j}$  ?*

**Théorème 14.** *VERTEX-COVER est  $\mathcal{NP}$ -complet*

*Démonstration.*

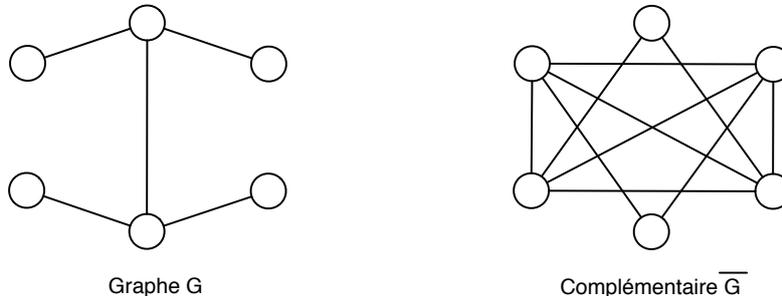
1. VERTEX-COVER  $\in \mathcal{NP}$  :

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  $|V| - k$ .



□

## 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 NP-complet

*Démonstration.*

1. CH  $\in$  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

*Démonstration.*

1. COLOR est dans NP.

Une instance  $I$  de COLOR est une paire  $(G, k)$  de taille  $|V| + |E| + \log k$ . Comme certificat on peut choisir une liste des couleurs des sommets. On peut vérifier en temps linéaire  $O(|V| + |E|)$  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  $C_k$ ,  $k = 1, \dots, p$ , et  $n$  variables  $x_1, \dots, x_n$ . 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

$$x_1, \dots, x_n, \bar{x}_1, \dots, \bar{x}_n, y_1, \dots, y_n, c_1, \dots, c_p$$

et les arêtes

$$(x_i, \bar{x}_i) \quad \forall i = 1, \dots, n$$

$$(y_i, y_j) \quad \forall i, j = 1, \dots, n, \quad i \neq j$$

$$(y_i, x_j) \quad \forall i, j = 1, \dots, n, \quad i \neq j$$

$$(y_i, \bar{x}_j) \quad \forall i, j = 1, \dots, n, \quad i \neq j$$

$$(x_i, c_k) \quad \forall x_i \notin C_k$$

$$(\bar{x}_i, c_k) \quad \forall \bar{x}_i \notin C_k.$$

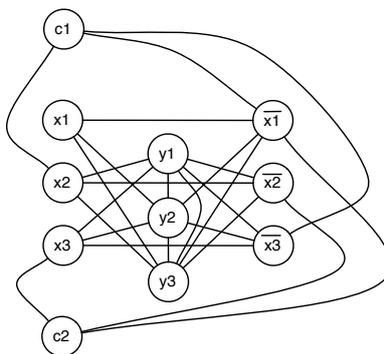


FIG. 9.1: Exemple : Le graphe construit à partir de la formule  $(x_1 \vee \bar{x}_2 \vee x_3) \wedge (x_1 \vee x_2 \vee \bar{x}_3)$ .

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  $|V| = O(n + p)$ .

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 pour tout  $k$ . On pose

$$color(x_i) = i \text{ si } x_i \text{ est vraie}$$

$$color(x_i) = n + 1 \text{ si } x_i \text{ est fausse}$$

$$color(\bar{x}_i) = i \text{ si } \bar{x}_i \text{ est vraie}$$

$$color(\bar{x}_i) = n + 1 \text{ si } \bar{x}_i \text{ est fausse}$$

$$color(y_i) = i$$

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  $\bar{x}$ ) et les clauses. Soit  $C_k$  une clause coloriée à  $i$  à cause d'une variable  $x_i$  ou  $\bar{x}_i$  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 cette dernière est coloriée 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  $\bar{x}_i$  ont pour couleur  $i$  ou  $n + 1$  et ce n'est pas la même.

En effet,  $x_i$  et  $\bar{x}_i$  sont reliés entre eux et avec tous les  $y_j$ ,  $j \neq i$ , on peut en déduire que

$$color\left(\begin{matrix} x_i \\ \bar{x}_i \end{matrix}\right) = \begin{pmatrix} i \\ n + 1 \end{pmatrix} \text{ ou } \begin{pmatrix} n + 1 \\ i \end{pmatrix}$$

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.  $x_i \notin C_k$ ,  $\bar{x}_i \notin C_k$ , et l'un des deux est de couleur  $n + 1$ .

Supposons alors que  $color(c_i) = k \in 1, \dots, n$ . 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)

$$\bar{x}_i = \begin{cases} 1 & \text{si } color(\bar{x}_i) = i \\ 0 & \text{si } color(\bar{x}_i) = n + 1. \end{cases}$$

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).*

*Démonstration.*

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.

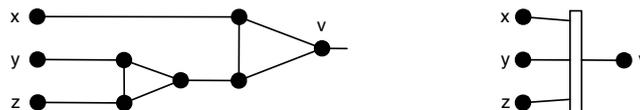


FIG. 9.2: A gauche le gadget, à droite son symbole.

**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.*

*Démonstration.* i) Si  $x = y = z = 0$  alors on ne peut que colorier le gadget comme suivant (fig. 9.3) :

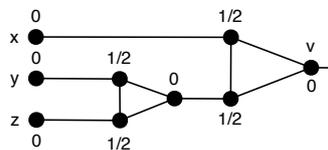


FIG. 9.3: Si  $x = y = z = 0$  alors  $v = 0$

□

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  $color(\bar{x}_i) = \begin{cases} 1 & \text{si } \bar{x}_i \text{ vrai} \\ 0 & \text{sinon} \end{cases}$ .

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.

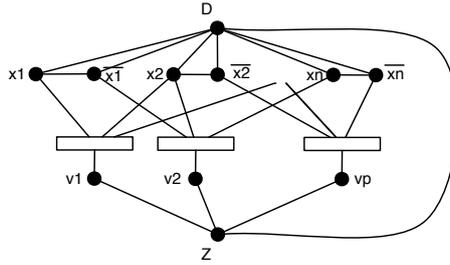


FIG. 9.4: Un exemple de graphe  $G$  construit à partir de l'instance  $I_1$  de 3-SAT. Le gadget  $k$  est relié avec les variables de la clause  $C_k$  de  $I_1$ .

(b)  $\Leftarrow$  On suppose que  $G$  a une coloration à trois couleurs. En permutant les couleurs on peut supposer que  $color(D) = 2$ ,  $color(Z) = 0$ ,  $color(\begin{smallmatrix} x_i \\ \bar{x}_i \end{smallmatrix}) = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$  ou  $\begin{pmatrix} 0 \\ 1 \end{pmatrix}$  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 conséquent, l'instantiation  $x_i = \begin{cases} 1 & \text{si } color(x_i) = 1 \\ 0 & \text{si } color(x_i) = 0 \end{cases}$ ,  $\bar{x}_i = \begin{cases} 1 & \text{si } color(\bar{x}_i) = 1 \\ 0 & \text{si } color(\bar{x}_i) = 0 \end{cases}$  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*

*Démonstration.*

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  $W$ .

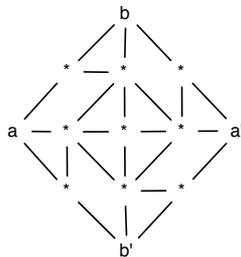


FIG. 9.5: Le gadget  $W$  avec 13 sommets

**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  $W$ .*

*Démonstration.*

i) On procède par exploration exhaustive des coloriage possibles. Les coloriage 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.

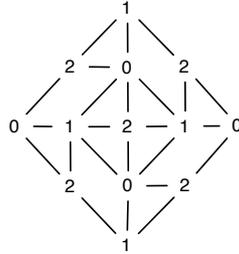


FIG. 9.6: 3-coloriage de  $W$  avec  $a = a' = 0, b = b' = 1$

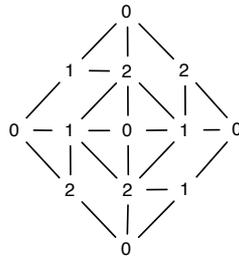


FIG. 9.7: 3-coloriage de  $W$  avec  $a = a' = 0, b = b' = 0$

□

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-

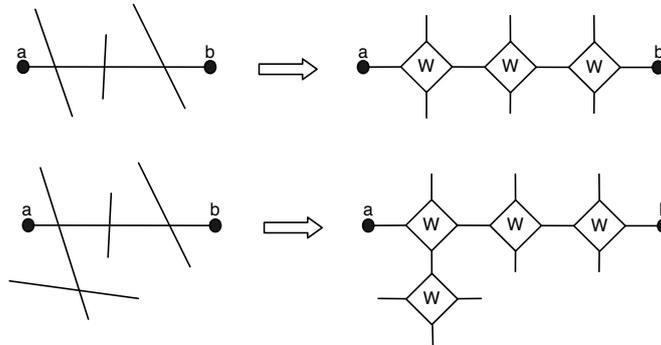


FIG. 9.8: On transforme le graphe  $G$  en un graphe planaire en remplaçant les arêtes qui se croisent par le gadget  $W$

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

*Instance* : 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  $C_0, \dots, C_{m-1}$  sur les variables  $x_0, \dots, x_{n-1}$ , 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.  $\bar{x}_i$ ) apparaît dans  $C_j$  et 0 sinon, et des entiers  $s_j = 10^j$  et  $s'_j = 2 \cdot 10^j$ ,  $0 \leq j \leq m-1$ . 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 ?

### Exercice 9.9.2. Variantes de 3-SAT

Montrer la  $\mathcal{NP}$ -complétude des deux variantes de 3-SAT suivantes :

**Question 9.1 3-SAT NAE** (*not all equal*), où l'on impose que les trois littéraux de chaque clause ne soient pas tous à la même valeur.

**Question 9.2 3-SAT OIT** (*one in three*), où l'on impose qu'exactement un littéral soit à VRAI dans chaque clause.

### Exercice 9.9.3. Variantes de SAT-N

On appelle SAT- $N$  le problème SAT restreint aux formules qui n'ont pas plus de  $N$  occurrences de la même variable.

**Question 9.3** 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.

**Question 9.4** 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.

### Exercice 9.9.4. De la $\mathcal{NP}$ -complétude

Montrer que les problèmes suivants sont NP-complets :

#### Question 9.5 Chevaliers de la table ronde

Étant donné  $n$  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 ?

**Question 9.6** Étant 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.

**Question 9.7** 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  $P = \{(a_1, b_1), \dots, (a_n, b_n)\}$  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  $P$  ?

*Indication : faire une réduction à partir de 3-SAT.*

**Question 9.8** 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  $k$  ?

**Question 9.9** Soient  $n = 2p$  un entier pair, et  $n$  entiers strictement positifs  $a_1, a_2, \dots, a_n$ . Existe-t-il une partition de  $\{1, 2, \dots, n\}$  en deux ensembles  $I$  et  $I'$  de même cardinal  $p$  et tels que  $\sum_{i \in I} a_i = \sum_{i \in I'} a_i$  ?

**Question 9.10** 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  $k$  ?

**Question 9.11** Étant donné 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  $w, v_1, v_2, \dots, v_K$  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).

**Question 9.12** Étant donné 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  $(u, v) \in E$

**Question 9.13** Étant donné  $n$  entiers  $a_1, a_2, \dots, a_n$ , peut-on trouver un sous-ensemble  $I \subset [1..n]$  tel que  $|\sum_{i \in I} a_i - \sum_{i \notin I} a_i| \leq 1$

### Exercice 9.9.5. Sous-chaîne transitive

Dans un graphe orienté, on dit que  $\{x_1, \dots, x_k\}$  est une chaîne transitive de longueur  $k$  si et seulement si pour tout  $1 \leq i < j \leq k$ ,  $(x_i, x_j) \in E$ . Montrer que le problème suivant est  $\mathcal{NP}$ -complet.

**Sous-chaîne transitive :**

*Instance :* Un graphe orienté  $D = (V, E)$ .

*Question :*  $D$  contient-il une sous-chaîne transitive de longueur au moins  $\lfloor \frac{|V|}{2} \rfloor$  ?

*Indication :* Vous pouvez par exemple effectuer une réduction à partir de 3-SAT. Si on se donne un ensemble de clauses  $C_1, \dots, C_k$ , avec  $C_i = (x_i^1 \vee x_i^2 \vee x_i^3)$ , on peut construire une instance du problème de sous-chaîne transitive en posant  $V = \{C_0\} \cup_{1 \leq i \leq k} \{C_i, x_i^1, x_i^2, x_i^3\}$  et en choisissant avec soin les arêtes à mettre dans  $E$ .

## 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  $a_1, \dots, a_n$  et un objectif  $t$ . Existe-t-il  $I$  tel que  $\sum_{i \in I} a_i = t$  ?
- 2-Partition :  $n$  entiers  $a_1, \dots, a_n$ . 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  $\lambda$ -approximation est un algorithme polynomial en la taille de l'instance qui renvoie une solution approchée garantie au pire cas à un facteur  $\lambda$ .*

**Exemple :** Pour un problème de minimisation, dire que la solution est garantie à un facteur  $\lambda$  de l'optimal s'écrit :

$$\forall I \text{ instance du problème, } ObjAlgo(I) \leq \lambda \cdot ObjOpt(I)$$

### 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.*

*Démonstration.* 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  $K \geq 0$ .

$c : E \mapsto \mathbf{N}$  : fonction de poids sur les arêtes.

Existe-t-il une tournée (un cycle hamiltonien) de poids  $\leq K$  ?

**Théorème 20.** *TSP est NP-complet.*

*Démonstration.* – TSP  $\in$  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

$$c(e) = \begin{cases} 1 & \text{si } e \in E \\ 0 & \text{sinon} \end{cases}$$

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  $\lambda$ -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  $\lambda$ -approximation sauf si  $P = NP$ .

*Démonstration. (par l'absurde) :*

Supposons qu'il existe une  $\lambda$ -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 :

$$c(e) = \begin{cases} 1 & \text{si } e \in E \\ \lambda \cdot |V| + 1 & \text{sinon} \end{cases}$$

On utilise ensuite l'algorithme de  $\lambda$ -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.

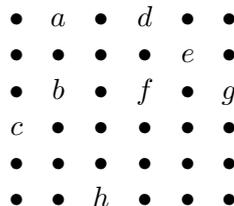
On peut donc répondre au problème CH suivant le résultat de l'algorithme, ce qui prouve que  $P = NP$ . □

**Remarque 8.** On a ici supposé  $\lambda$  constant, mais on peut aller jusqu'à  $\lambda = 2^{|V|}$  car alors  $\lambda$  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).

Voilà 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 :  $w = \text{abcbhbadefegeda}$ )
- On ne garde que la première occurrence de chaque lettre pour obtenir une tournée (ex :  $\text{abchdefga}$ )

**Théorème 22.** *Cet algorithme est une 2-approximation de TSP, i.e.  $c(\text{Tournée}) \leq 2 \cdot c(\text{Opt})$*

*Démonstration.*

$c(w) = 2 \cdot c(T)$ , où  $w$  est le chemin qui parcourt deux fois chaque arête de l'arbre couvrant de poids minimum.

$c(\text{Tournée}) \leq c(w)$  à cause de l'inégalité triangulaire.

$c(T) \leq c(\text{Opt})$  car une tournée optimale à laquelle on supprime une arête est un arbre, et  $T$  est l'arbre couvrant de poids minimum.

Donc  $c(\text{Tournée}) \leq 2 \cdot c(\text{Opt})$ , ce qui prouve le résultat.  $\square$

**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 : boîtes de tailles  $B$  et  $n$  objets de volume  $a_1, \dots, a_n$ .

But : placer ces objets dans un nombre minimal de boîtes.

**Définition :** Soit  $n$  rationnels  $a_1, \dots, a_n$ , avec  $0 < a_i \leq 1$ . Peut-on les partitionner en  $K$  boîtes  $B_1, \dots, B_K$  de capacité 1, c'est à dire :

$$\sum_{j \in B_k} a_j \leq 1$$

**Théorème 23.** *Bin Packing est NP-complet.*

*Démonstration.*

- $\text{BP} \in \text{NP}$  (facile). Certificat : liste des entiers de chaque boîte.
- Réduction à partir de 2-Partition. Soit une instance  $I_1$  de 2P :  $b_1, \dots, b_n$ . 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)

$\square$

**Théorème 24.** *Pour tout  $\epsilon > 0$  il n'existe pas de  $\frac{3}{2} - \epsilon$  approximation sauf si  $P = \text{NP}$ .*

*Démonstration.* 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  $P = \text{NP}$ .  $\square$

### 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.*

*Démonstration.*

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,

$$K - 1 < \sum_{i=1}^{K-1} (\text{contenu}(B_i) + \text{contenu}(B_{i+1}))$$

En outre, on a  $\sum_{i=1}^{K-1} (\text{contenu}(B_i) + \text{contenu}(B_{i+1})) \leq 2A$ , 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.  $\square$

**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.**  $K_{DFF} \leq \frac{3}{2}K_{Opt} + 1$

*Démonstration.* On sépare les  $a_i$  en quatre catégories :

$$A = \{a_i > \frac{2}{3}\} \quad B = \{\frac{2}{3} \geq a_i > \frac{1}{2}\} \quad C = \{\frac{1}{2} \geq a_i > \frac{1}{3}\} \quad D = \{\frac{1}{3} \geq a_i\}$$

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.  $\square$

**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

$$n + \sum_{i=1}^n \lceil \log a_i \rceil \quad n + \max_{i \in \{1..n\}} \lceil \log a_i \rceil \quad n \lceil \log \sum_{i=1}^n a_i \rceil$$

**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) :

$$\sum_{i=1}^n a_i \quad \max_{i \in \{1..n\}} a_i \quad \frac{\sum_{i=1}^n a_i}{n}$$

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 donné 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.*

*Démonstration.* 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 ordonné  $a_j$ ) : 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,  $M_1 \leq 7/6 Opt$ . 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 tâches 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 ordonner toutes les tâches que pour ordonner les  $j \leq 4$  premières. Mais pour tout ensemble d'au plus quatre tâches, GLOUTON-TRI est optimal, et  $M_1 = Opt$ .  $\square$

### 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  $\lambda = 1 + \epsilon > 1$ , il existe une

$\lambda$ -approximation, i.e. un algorithme dont la performance est garantie à un facteur  $\lambda$ , et dont le temps d'exécution est polynomial. Attention : polynomial en la taille de l'instance de 2-Partition, mais pas forcément en  $1/\epsilon$ , 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  $1/\epsilon$ .

**Théorème 28.**  $\forall \epsilon > 0$ , 2-Partition admet une  $(1 + \epsilon)$ -approximation.

*Démonstration.* On pose :

$$P_{sum} = \sum_{i=1}^n a_i$$

$$P_{max} = \max a_i$$

$$L = \max\left(\frac{1}{2}P_{sum}, P_{max}\right)$$

On a déjà vu plus haut que :

**Lemme 10.**  $L \leq Opt$ .

Soit  $I$  une instance de 2-partition :  $a_1, \dots, a_n$ . 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  $\epsilon L$ , 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  $I^*$ .

*Démonstration.* 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  $P_2$ .

$P_1$  : ——— o o ——— o o o ———

$P_2$  : ——— o o ——— 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  $\lfloor \frac{S_i}{\epsilon L} \rfloor$  jobs de taille  $\epsilon L$ . On a symboliquement :

$P_i$  : ——— ——— ——— o o o o o o o o

Comme

$$\left\lceil \frac{S_1}{\epsilon * L} \right\rceil + \left\lceil \frac{S_2}{\epsilon * L} \right\rceil \geq \left\lceil \frac{S_1 + S_2}{\epsilon * L} \right\rceil = \left\lceil \frac{S}{\epsilon * L} \right\rceil$$

on a ordonnancé au moins autant de jobs de taille  $\epsilon L$  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 :

$$Opt^* \leq Opt + \epsilon L \leq Opt + \epsilon \cdot Opt$$

□

**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  $\epsilon L$ , 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  $\epsilon L$ , 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 ordonnancer  $I^*$ , il suffit d'essayer toutes les  $2^{\frac{2}{\epsilon}}$  possibilités d'ordonnement et de garder la meilleure. Bien sûr ceci a un coût non polynomiale 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'ordonnement optimal  $\sigma^*$  pour  $I^*$ . Soit  $L_i^*$  la charge de la machine  $P_i$  avec  $\sigma^*$ , 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'ordonnement  $\sigma$  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.*

*Démonstration.* 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  $\epsilon L$ , il faut qu'il reste moins de  $\epsilon L$  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  $\sigma$  ordonnancement pour  $I$  en temps polynomiale. Son temps d'exécution excède celui de l'ordonnement 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 big 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'ordonnement 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'ordonnement 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  $\epsilon > 0$ , il existe une  $(1 + \epsilon)$ -approximation qui soit polynomiale à la fois en la taille de  $I$  et en  $1/\epsilon$ .

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  $(1/\varepsilon)$

*Démonstration.* 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 :

$$\begin{cases} VS_1 = \{[a_1, 0], [0, a_1]\} \\ VS_{k-1} \rightarrow VS_k : \forall [x, y] \in VS_{k-1}, [x + a_k, y] \text{ et } [x, y + a_k] \in VS_k \end{cases}$$

**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  $\Delta$ , entre 0 et  $\Delta^M$ .

$$M = \lceil \log_{\Delta} P_{sum} \rceil = \left\lceil \frac{\log P_{sum}}{\log \Delta} \right\rceil \leq \left\lceil \left(1 + \frac{2n}{\varepsilon}\right) \ln(P_{sum}) \right\rceil$$

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

$$\begin{cases} \frac{x_1}{\Delta} \leq x_2 \leq \Delta x_1 \\ \frac{y_1}{\Delta} \leq y_2 \leq \Delta y_1 \end{cases}$$

$$\begin{cases} VS_1^{\#} = VS_1 \\ VS_{k-1}^{\#} \rightarrow VS_k^{\#} \end{cases} :$$

$\forall [x, y] \in VS_{k-1}^{\#}, [x + a_k, y] \in VS_k^{\#}$  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  $y^{\#} \leq \Delta^k y$

*Démonstration. Par récurrence*

$\mathbf{k} = 1$  : trivial

$\mathbf{k} - 1 \rightarrow \mathbf{k}$  : Soit  $[x, y] \in VS_k$ ,

$$\begin{cases} x = u + a_k \text{ et } y = v \\ \text{ou} \\ x = u \text{ et } y = v + a_k \end{cases} \quad \text{où } [u, v] \in VS_{k-1}$$

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  $VS_k^\#$  :

$$\begin{aligned} \exists [x^\#, y^\#] \in VS_k^\# \text{ tel que } x^\# \leq \Delta(u^\# + a_k) \text{ et } y^\# \leq \Delta v^\# \\ \Rightarrow \begin{cases} x^\# \leq \Delta^k u + \Delta a_k \leq \Delta^k(u + a_k) = \Delta^k x \\ y^\# \leq \Delta v^\# \leq \Delta^k y \end{cases} \end{aligned}$$

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  $0 \leq z \leq 1$ ,  $f(z) = \left(1 + \frac{z}{n}\right)^n - 1 - 2z \leq 0$

*Démonstration.*  $f'(z) = \frac{1}{n}n \left(1 + \frac{z}{n}\right)^{n-1} - 2$ .

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 le TD précédent, 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  $t$ , 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  $S = \{x_1, x_2, \dots, x_n\}$  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 les listes  $L$  et  $L'$ , on note **Fusion**( $L, L'$ ) la liste contenant l'union des éléments des deux listes.

## Premier algorithme

```
début
   $n \leftarrow |S|;$ 
   $L_0 \leftarrow \{0\};$ 
  pour  $i$  de 1 à  $n$  faire
     $L_i \leftarrow \text{Fusion}(L_{i-1}, L_{i-1} + x_i);$ 
    Supprimer de  $L_i$  tout élément supérieur à  $t$ ;
  fin
  retourner le plus grand élément de  $L_n$ ;
fin
```

**Algorithme 5** : Somme( $S, t$ )

**Question 10.1** Quelle est la distance entre la valeur retournée par cet algorithme et la somme optimale ?

**Question 10.2** 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 ?

**Deuxième algorithme** Cet algorithme prend en entrée un paramètre  $\epsilon$  en plus, où  $\epsilon$  est un réel vérifiant  $0 < \epsilon < 1$ .

```
début
   $n \leftarrow |S|;$ 
   $L_0 \leftarrow \{0\};$ 
  pour  $i$  de 1 à  $n$  faire
     $L_i \leftarrow \text{Fusion}(L_{i-1}, L_{i-1} + x_i);$ 
     $L_i \leftarrow \text{Seuiller}(L_i, \epsilon/2n);$ 
    Supprimer de  $L_i$  tout élément supérieur à  $t$ ;
  fin
  retourner le plus grand élément de  $L_n$ ;
fin
```

**Algorithme 6** : Somme-avec-seuillage( $S, t, \epsilon$ )

L'opération **Seuiller** décrite ci-dessous réduit une liste  $L = \langle y_0, y_1, \dots, y_m \rangle$  (supposée triée par ordre croissant) avec le seuil  $\delta$  :

```
début
   $m \leftarrow |L|;$ 
   $L' \leftarrow \langle y_0 \rangle;$ 
   $dernier \leftarrow y_0;$ 
  pour  $i$  de 1 à  $m$  faire
    si  $y_i > (1 + \delta)dernier$  alors
      Insérer  $y_i$  à la fin de  $L'$ ;
       $dernier \leftarrow y_i;$ 
    fin
  fin
  retourner  $L'$ ;
fin
```

**Algorithme 7** : Seuiller( $L, \delta$ )

**Question 10.3** Évaluer 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.

## 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](http://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/>.