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

Cours
Yves Robert, Anne Benoit

Rédaction
Etudiants-scribes (2003 et 2004)

Version corrigée, Novembre 2006

Table des matières

Chapitre 1

Introduction : calcul de x^n

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

1.1 Énoncé du problème

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

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

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

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

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

1.2 Algorithme naïf

Considérons l'algorithme *naïf* suivant :

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

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

1.3 Méthode binaire

On trouve facilement un algorithme plus efficace :

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

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

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

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

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

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

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

1.4 Méthode des facteurs

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

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

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

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

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

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

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

1.5 Arbre de Knuth

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

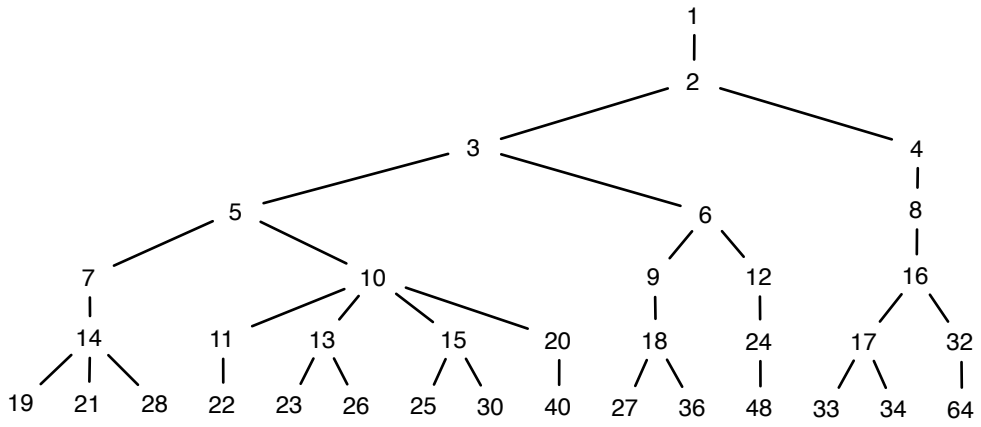


FIG. 1.1 – Les sept premiers niveaux de l'arbre de Knuth.

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

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

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

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

1.6 Résultats sur la complexité

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

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

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

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

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

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

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

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

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

Ensuite on calcule successivement :

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

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

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

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

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

(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 $2^k \le \sqrt{\log n}$). □

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

1.7 Références bibliographiques

La présentation du cours s'inspire de Knuth [5]. Les exercices *Le grand saut* et *Bricolage* sont tirés de Rawlins [6].

Chapitre 2

Diviser pour régner

2.1 Algorithme de Strassen

Calculons un produit de matrices :

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

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

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

alors on peut écrire

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

Comptons maintenant les opérations :

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

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

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

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

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

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

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

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

Soient :

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

On a :

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

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

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

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

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

2.2 Produit de deux polynômes

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

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

Soient :

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

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

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

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

On suppose n pair, $n = 2m$

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

Soient :

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

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

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

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

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

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

En effet, pour la construction de R :

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

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

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

(on verra Section 2.4 comment résoudre cette récurrence)

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

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

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

2.3 Master theorem

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

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

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

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

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

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

et :

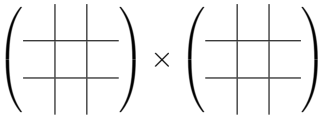
$$\sum = c \times n^\alpha \sum_{i=0}^{k-1} (a/b^\alpha)^i$$

On distingue alors plusieurs cas :

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

La preuve complète est dans le Cormen.

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



On a :

a	b	α	c
a	3	2	

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

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

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

2.4 Résolution des récurrences

2.4.1 Résolution des récurrences homogènes

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

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

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

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

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

2.4.2 Résolution des récurrences avec second membre

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

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

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

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

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

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

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

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

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

(par hypothèse de récurrence)

Résolution pour les additions dans l'algorithme de Strassen :

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

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

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

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

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

avec :

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

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

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

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

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

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

avec :

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

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

2.5 Multiplication et inversion de matrices

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

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

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

La multiplication est au moins aussi complexe que l'inversion

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

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

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

D'où : $M(n) \leq I(3n)$

L'inversion est au moins aussi complexe que la multiplication

On procède en deux étapes :

Si A est symétrique définie positive

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

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

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

Il suffit de construire :

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

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

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

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

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

□

2.6 Références bibliographiques

La présentation du cours, et l'exercice *Matrices de Toeplitz*, s'inspirent du Cormen [1]. L'exercice *Recherche d'un élément majoritaire* est tiré de Darte et Vaudenay [2].

Chapitre 3

Programmation dynamique

3.1 Pièces de Monnaies

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

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

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

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

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

Mais on a une relation de récurrence :

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

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

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

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

  fin
fin

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

3.2 Le problème du sac à dos

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

3.2.1 En glouton

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

Question : Est-ce optimal ? Non.

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

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

est un exemple d'un tel cas de figure.

3.2.2 Par programmation dynamique

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

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

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

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

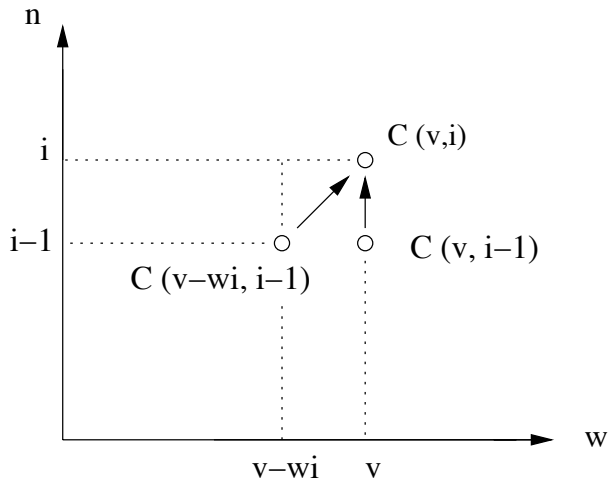


FIG. 3.1 – Attention au respect des dépendances !

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

3.3 Quelques exemples de programmation dynamique

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

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

3.3.1 Chaînes de matrices

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

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

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

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

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

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

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

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

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

et le coût optimal s'obtient par

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

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

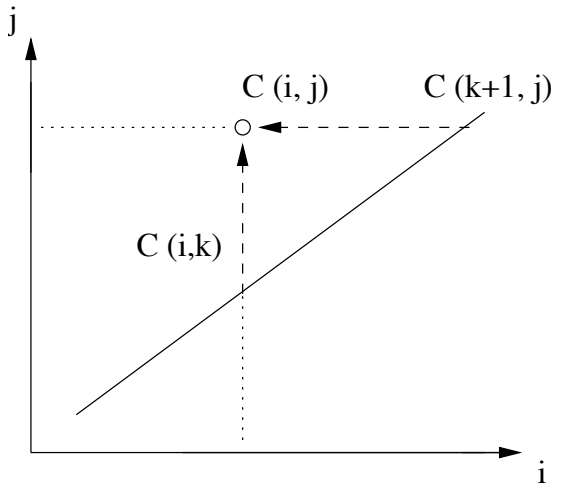


FIG. 3.2 – L'initialisation se fait sur $C(i, i+1) = P_{i-1} * P_i * P_{i+1}$ et $C(i,i) = 0$ sur la diagonale

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

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

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

3.3.2 Plus longue sous-suite

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

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

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

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

Solution exhaustive

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

Programmation dynamique

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

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

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

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

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

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

Preuve
On constate d'abord que :

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

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

1. Si $a_i \neq b_j$, a_i et b_j ne peuvent pas appartenir tous deux à la même sous-suite commune. On a donc deux cas : soit a_i appartient à la sous-suite et $p(i,j) = p(i,j-1)$, soit c'est b_j qui y appartient, et on a $p(i,j) = p(i-1,j)$.

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

On a donc prouvé la relation de récurrence.

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

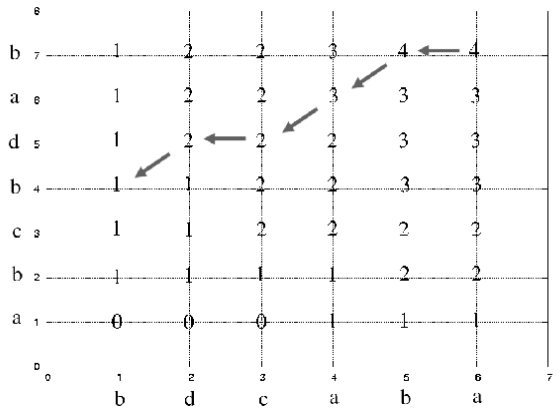


FIG. 3.3 – La *PLSCC* est *bdab*.

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

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

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

On trouve alors des algorithmes en $O((n-r)m)$ ou en $O(rm)$, avec r la longueur de la *PLSCC* de deux mots de longueurs n et m.

3.3.3 Location de skis

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

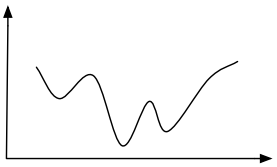
3.4 Références bibliographiques

La présentation du cours et les exercices s'inspirent du Cormen [1].

Chapitre 4

Algorithmes gloutons

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



4.1 Exemple du gymnase

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

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

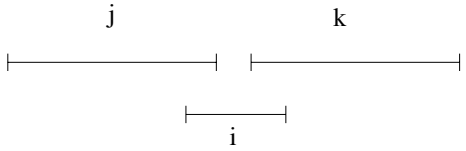


FIG. 4.1 – L'essai 1 ne conduit pas à l'optimal.

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

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

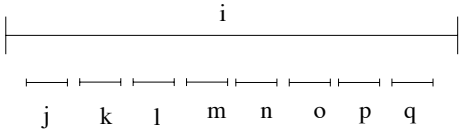


FIG. 4.2 – L'essai 2 ne conduit pas à l'optimal.

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

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

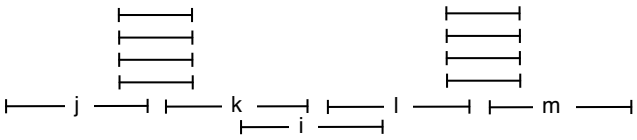


FIG. 4.3 – L'essai 3 ne conduit pas à l'optimal.

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

Théorème 4. *Cet algorithme glouton est optimal.*

Preuve. Soit f_1 l'élément finissant le plus tôt. On va montrer qu'il existe une solution optimale contenant cet événement. Soit donc une solution optimale arbitraire O,

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

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

4.2 Route à suivre pour le glouton

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

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

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

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

Parenthèse sur le sac à dos entier/fractionnel

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

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

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

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

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

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

4.3 Coloriage d'un graphe

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

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

Preuve.

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

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

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

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

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

4.3.1 Algorithme glouton 1

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

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

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

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

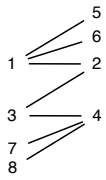
4.3.2 Algorithme glouton 2

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

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

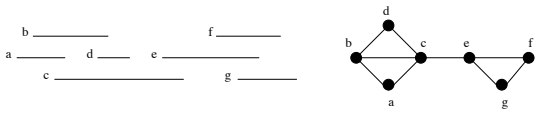
Cette borne suggère de se débarrasser d'abord des plus gros degrés, qui disparaissent ainsi de la complexité tant que $\min(d_i + 1, i) = i$. Comme on ne sait pas a priori jusqu'à quand privilégier les grands degrés, on a trié l'ensemble des sommets par degrés décroissants dans l'algorithme glouton.

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



4.3.3 Graphe d'intervalles

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



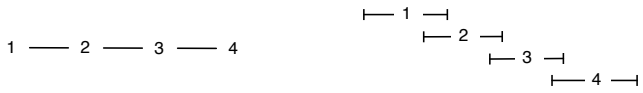
Notons que ce problème est proche de celui du gymnase : on considère que l'on dispose d'un ensemble d'événements et on cherche le nombre minimal de gymnases nécessaires pour caser tous ces événements.

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

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

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

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

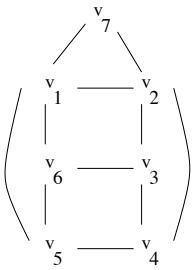


4.3.4 Algorithme de Brelaz

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

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

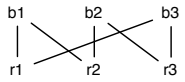
Il s'agit d'un glouton sur (degré-couleur, degré), appelé algorithme de Brélaz. Considérons l'exemple ci-dessous :



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

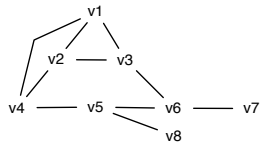
Théorème 6. *L'algorithme de Brelaz est optimal sur les graphes bipartis, i.e. réussit toujours à les colorier avec deux couleurs.*

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



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

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



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

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

4.4 Théorie des matroïdes

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

4.4.1 Matroïdes

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

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

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

Exemples

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

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

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

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

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

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

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

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

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

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

4.4.2 Algorithme glouton

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

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

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

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

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

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

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

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

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

4.5 Ordonnancement

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

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

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

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

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

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

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

Pour notre problème on va se restreindre :

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

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

Exemple : 7 tâches :

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

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

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

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

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

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

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

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

Preuve.

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

□

Proposition 1. *On a un matroïde.*

Preuve.

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

□

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

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

4.6 Références bibliographiques

Tout le chapitre (cours et exercice *Codage de Huffman*) s'inspire du Cormen [1], à l'exception des algorithmes de coloriage de graphes, tirés du West [7].

Chapitre 5

Tri

5.1 Tri fusion

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

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

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

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

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

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

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

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

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

5.2 Tri par tas : Heapsort

5.2.1 Définitions

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

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

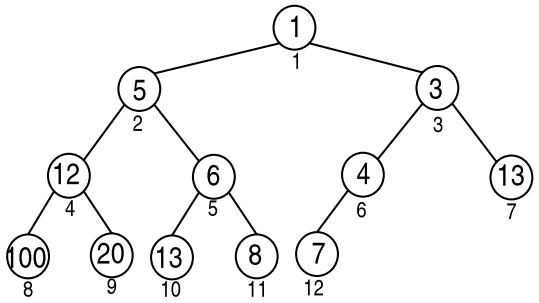


FIG. 5.1 – Exemple de tas

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

5.2.2 Tri par tas

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

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

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

5.2.3 Insertion d'un nouvel élément

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

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

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

5.2.4 Suppression d'un élément du tas

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

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

5.2.5 Complexité du tri par tas

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

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

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

5.3 Tri rapide

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

5.3.1 Coût

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

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

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

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

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

5.3.2 Médiane en temps linéaire

Problème

Données : ensemble A de n éléments distincts totalement ordonné.

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

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

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

Algorithme

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

5.4 Complexité du tri

5.4.1 Les grands théorèmes

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

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

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

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

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

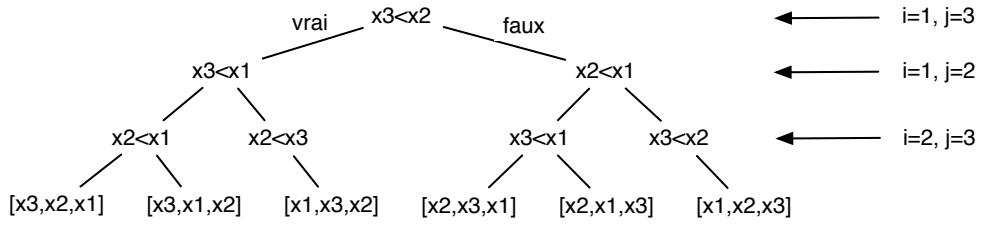
On a vu que la complexité de tri fusion ne dépend pas des données et vaut $comp_{TF}(n) = n\lceil \log_2(n) \rceil - 2^{\lceil \log_2(n) \rceil} + 1$, ce qui tend vers $n \log_2(n)$, montrant ainsi que la borne peut être atteinte asymptotiquement.

5.4.2 Démonstration des théorèmes

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

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

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



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

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

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

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

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

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

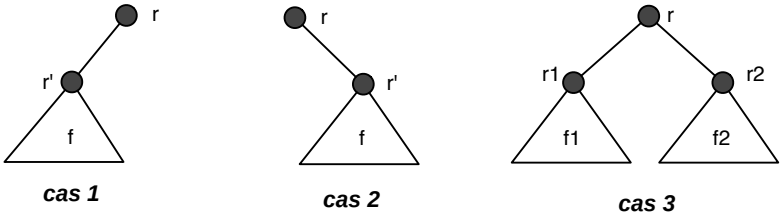
Pour le deuxième théorème, on montre que la hauteur moyenne est supérieure à $\log_2(n!)$ grâce au lemme suivant, et on termine également avec la formule de Stirling.

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

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

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

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



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

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

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

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

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

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

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

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

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

□

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

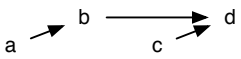
5.4.3 Peut-on atteindre la borne ?

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

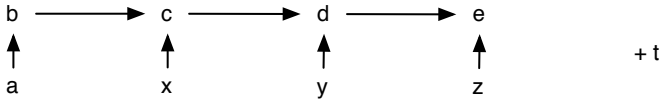
n	2	3	4	5	6	7	8	9	10	11	12
TriFusion(n)	1	3	5	8	11	14	17	21	25	29	33
$\lceil \log_2(n!) \rceil$	1	3	5	7	10	13	16	19	22	26	29
Opt(n)	1	3	5	7	10	13	16	19	22	26	30

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

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

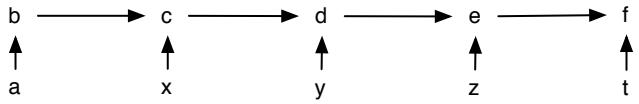


    on forme 2 paires de 2, ($a \to b$) et ($c \to d$), on compare les deux plus grands pour obtenir par exemple ($a \to b \to d$), puis on insère c par dichotomie (2 comparaisons) : coût 2+1+2=5.
- trier 5 nombres :
  faire deux paires sur a, b, c, d et comparer les plus grands (coût = 3) pour obtenir le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



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

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



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

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

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

5.5 Références bibliographiques

Ce chapitre est un grand classique. Nous avons surtout utilisé le livre de Froidevaux, Gaudel et Soria [3] (cours et nombreux exercices). L'exercice *Plus petit et plus grand* est dû à Rawlins [6].

Chapitre 6

$\mathcal{NP}$-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.

6.1 Problèmes de $\mathcal{P}$

6.1.1 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)**.

6.1.2 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.

6.1.3 Exemples

Un graphe donné est-il biparti ?

La taille des données dépend ici de la façon dont on stocke le graphe. Un graphe est caractérisé par la liste de ses n sommets v_1, ..., v_n, et donc on est en $O(n)$ juste pour énumérer ces sommets.

- Les adresses des sommets se codent en $\log(n)$, soit $n\log(n)$ pour les n sommets, ce qui reste polynomial en n.
- Il y a au plus n^2 arêtes, donc on reste encore polynomial en n.

On ne peut pas se contenter de stocker les nombres correspondant aux numéros des sommets (codés en $\log(n)$) car on veut pouvoir les énumérer, pour pouvoir accéder directement aux sommets. La taille des données est donc polynomiale en n, où n est le nombre de sommets du graphe.

En algorithmique, on considère souvent comme taille des données pour un graphe $n+p$, où p est le nombre d'arêtes. Cela permet d'optimiser les algorithmes lorsque $p << n^2$. Mais cela ne change rien à la classification du problème vu que l'on reste polynomial en n. Il faut garder à l'esprit (PJ) : la composition de polynômes est un polynôme.

Ainsi, pour répondre à la question "le graphe est-il biparti", on doit effectuer un nombre d'opérations polynomial en n, le problème est dans $\mathcal{P}$.

2-Partition

Problème : Soit n entiers $a_1, ..., a_n$.
Question : Peut-on partitionner ces entiers en deux sous-ensembles tels que $\sum_{i\in I} a_i = \sum_{i\notin I} a_i$?

Ce problème ressemble à celui du sac-à-dos, en plus simple. Quelle est la taille des données ?

Comme dans le cas du graphe, il faut ici pouvoir énumérer les n entiers, donc on est au moins en n. Il faut aussi stocker les a_i, ce qui rajoute $\sum\log(a_i)$, soit des données de taille $n+\sum\log(a_i)$.
Peut-on stocker en :

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

Le premier codage est autorisé, c'est polynomial en $n+\sum\log(a_i)$ (cf PJ). En revanche, les deux autres, qui correspondent à un codage unaire des a_i, n'est pas autorisé.

On trouve un algorithme polynomial en la taille du codage unaire, en programmation dynamique comme pour le sac-à-dos. Si S est la somme des a_i, le problème de décision revient à calculer $c(n, S/2)$, où $c(i,T)$ signifie "peut-on faire la somme T avec les i premiers éléments a_i ?". La relation de récurrence est $c(i,T) = c(i-1, T-a_i) \vee c(i-1,T)$. La complexité de cet

algorithme est en $O(n.S)$, c'est donc polynomial en la taille des données codées en unaire. On parle dans ce cas d'algorithme **pseudo-polynomial**.

En revanche, personne ne connaît d'algorithme polynomial en la taille du codage binaire pour résoudre ce problème, donc on ne sait pas si ce problème est dans $\mathcal{P}$.

Par contre, si l'on nous donne une solution du problème, on peut vérifier en temps polynomial (et même linéaire) en la taille des données que la solution marche. On se pose alors la question : qu'appelle-t-on solution d'un problème ?

6.1.4 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.

6.2 Problèmes de $\mathcal{NP}$

6.2.1 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.

6.2.2 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.

6.2.3 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.

6.2.4 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.

6.2.5 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.

6.2.6 Problèmes polynomiaux

Concluons cette introduction à la NP-complétude par une petite réflexion sur les problèmes polynomiaux. Pourquoi s'intéresse-t-on à la classe des problèmes polynomiaux ?

Considérons n^{1000} vs $(1.0001)^n$. Pour $n = 10^9$, $n^{1000} > (1.0001)^n$. Il faut atteindre $n = 10^{10}$ pour que la solution polynomiale soit plus efficace que la solution exponentielle. De plus, pour des valeurs de n raisonnables, par exemple $n = 10^6$, on a $10^{6000} >>> 22026 > (1.0001)^n$. Dans ce cas, la solution exponentielle est donc à privilégier.

Cependant, en général, les polynômes sont de degré inférieur à 10, et le plus souvent le degré ne dépasse pas 3 ou 4. On n'est donc pas confronté à un tel problème.

Surtout, grâce à la composition des polynômes (PJ), la classe des problèmes polynomiaux est une classe stable.

6.3 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 $\mathcal{P}$.

6.4 3-SAT

Définition 6 (SAT). *Soit F une formule booléenne à n variables $x_1, x_2, ..., x_n$ et p clauses C_i : $F = C_1 \wedge C_2 \wedge ... \wedge C_p$ où $C_i = x^*_{i_1} \vee x^*_{i_2} \vee ... \vee x^*_{i_{f(i)}}$ et $x^* = x$ ou $\overline{x}$.*
Existe-t-il une instanciation des variables telle que F soit vraie ($\Leftrightarrow C_i$ vraie pour tout i) ?

SAT est donc NP-complet (résultat de Cook). Il existe plusieurs autres variantes NP-complètes :

- **SAT-k :** SAT avec au plus k occurrences de chaque x_i.
- **3-SAT :** SAT où chaque clause a exactement 3 littéraux : $C_i = x^*_{i_1} \vee x^*_{i_2} \vee x^*_{i_3}$ ($f(i) = 3$ pour tout i).
- **3-SAT NAE :** "not all equal", les trois variables de chaque clause ne sont pas toutes à la même valeur dans une solution.
- **3-SAT OIT :** "one in three", exactement une variable est à VRAI dans chaque clause dans une solution.

Remarque : 2-SAT peut être résolu en temps polynomial.

On s'intéresse ici à 3-SAT. Les variantes seront étudiées en TD.

Théorème 11. *3-SAT est NP-complet.*

Preuve.

1. Montrons que 3-SAT $\in \mathcal{NP}$
   Soit I une instance de 3-SAT : taille$(I) = O(n + p)$.
   Certificat : valeur de chaque x_i, de taille $O(n)$.
   La vérification se fait en $O(n + p)$.
   (On peut aussi simplement dire 3-SAT $\in \mathcal{NP}$ car SAT $\in \mathcal{NP}$).
2. 3-SAT est complet : on réduit à partir de SAT (premier problème montré NP-complet par Cook).
   Soit I_1 une instance de SAT, et on cherche à construire une instance I_2 équivalente pour 3-SAT.
   n variables $x_1, ..., x_n$
   p clauses $C_1, ..., C_p$ de longueurs $f(1), ..., f(p)$
   taille$(I_1) = O(n + \sum_{i=1}^{p} f(i))$
   Soit C_i la i^{eme} clause de F dans I_1. On construit un ensemble de clauses C'_i à trois variables que l'on rajoute dans l'instance I_2 de 3-SAT.
   - Si C_i a 1 variable x, soit a_i et b_i deux nouvelles variables. On rajoute les clauses :
     $x \vee a_i \vee b_i$
     $x \vee \overline{a_i} \vee b_i$
     $x \vee a_i \vee \overline{b_i}$
     $x \vee \overline{a_i} \vee \overline{b_i}$
   - Si C_i a 2 variables $x_1 \vee x_2$, soit c_i une nouvelle variable, et on rajoute les clauses :
     $x_1 \vee x_2 \vee c_i$
     $x_1 \vee x_2 \vee \overline{c_i}$
   - 3 variables : aucun changement
   - k variables, $k > 3$, $C_i = x_1 \vee x_2 \vee ... \vee x_k$
     Soit $z_1^{(i)}, z_2^{(i)}, ..., z_{k-3}^{(i)}$ $(k-3)$ nouvelles variables. On rajoute les $k-2$ clauses :
     $x_1 \vee x_2 \vee z_1^{(i)}$
     $x_3 \vee \overline{z_1^{(i)}} \vee z_2^{(i)}$
     ...
     $x_{k-2} \vee \overline{z_{k-4}^{(i)}} \vee z_{k-3}^{(i)}$
     $x_{k-1} \vee x_k \vee \overline{z_{k-3}^{(i)}}$

   On construit l'instance I_2 de 3-SAT comme étant l'ensemble des variables $x_1, ..., x_n$ ainsi que les variables rajoutées dans la construction des clauses, et les clauses construites ci-dessus, toutes de longueur 3. La construction se fait en temps polynomial. On doit ensuite vérifier les points de la réduction :

(a) taille(I_2) est linéaire en taille(I_1)

(b) ($\Rightarrow$) (côté facile) : si I_1 a une solution c'est à dire si l'on a une instanciation des x_i telle que $\forall j$ C_j soit vraie, alors une solution pour I_2 est :

$x_i \rightsquigarrow$ inchangé
$a_i \rightsquigarrow$ VRAI
$b_i \rightsquigarrow$ FAUX
$c_i \rightsquigarrow$ VRAI

Si l'on considère une clause de I_1 à $k > 3$ variables, $C_i = y_1 \vee ... \vee y_k$, soit y_j le premier littéral vrai de la formule. Alors pour la solution de I_2 on dit (en omettant les exposants (i) pour les variables z) :

$z_1, ..., z_{j-2} \rightsquigarrow$ VRAI
$z_{j-1}, ..., z_{k-3} \rightsquigarrow$ FAUX

Ainsi, si toutes les clauses de I_1 sont à VRAI, alors toutes les clauses de I_2 sont également à VRAI : $I_1 \Rightarrow I_2$.

(c) ($\Leftarrow$) (côté difficile) : si I_2 a une solution, on a une instanciation des $x_i, a_i, b_i, c_i, z^i_j \rightsquigarrow$ VRAI/FAUX. Alors on va montrer qu'il se trouve que la même instanciation des $x_1, ..., x_n$ marche pour la formule de départ (on a de la chance, on aurait pu imaginer qu'il faille chercher une instanciation des x_i plus compliquée pour mettre I_1 à VRAI).
Pour une clause à une ou deux variables, quelles que soient les valeurs de a_i, b_i et c_i on impose que x ou $x_1 \vee x_2$ soient à VRAI car on a rajouté des clauses contraignant les variables supplémentaires. Pour les clauses à trois variables, elles restent VRAI vu qu'on ne les a pas changées.
Soit C_i une clause de I_1 à $k > 3$ variables, $C_i = y_1 \vee ... \vee y_k$. Si cette clause est à FAUX, alors forcément z_1 doit être à vrai, puis en descendant dans les clauses de I_2 on montre que tous les z doivent être à VRAI. La contradiction survient sur la dernière ligne car il faut aussi $\overline{z_{k-3}}$ à VRAI si y_{k-1} et y_k sont tous à FAUX. L'un au moins des y_j doit donc être à VRAI, la clause est vérifiée.
Ainsi, si toutes les clauses de I_2 sont à VRAI, alors toutes les clauses de I_1 sont également à VRAI : $I_2 \Rightarrow I_1$, ce qui clôt la démonstration.

□

6.5 Clique

Définition 7 (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 $|E| \leq |V|^2$).

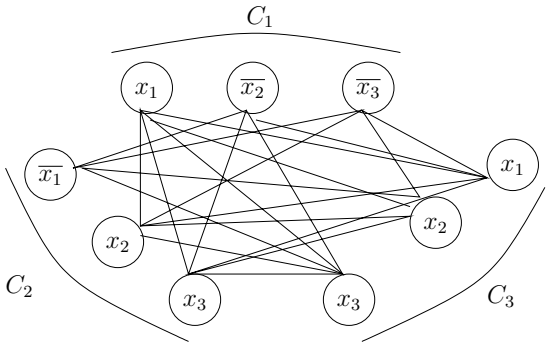
Théorème 12. *CLIQUE est $\mathcal{NP}$-complet*

Preuve.

1. CLIQUE $\in \mathcal{NP}$
   Certificat : liste des sommets de la clique.
   Vérification en temps quadratique (pour chaque paire de sommets du certificat, l'arête les reliant doit appartenir à E).
2. CLIQUE est complet : réduction à partir de 3-SAT.
   Soit I_1 une instance de 3-SAT avec n variables booléennes et p clauses de taille 3.
   Par exemple (tiré du Cormen), $C_1 \wedge C_2 \wedge C_3$ avec
   $C_1 = x_1 \vee \overline{x_2} \vee \overline{x_3}$

$C_2 = \overline{x_1} \vee x_2 \vee x_3$
$C_3 = x_1 \vee x_2 \vee x_3$

On construit une instance I_2 de clique à partir de I_1 : on ajoute 3 sommets au graphe pour chaque clause, et une arête entre deux sommets si et seulement si (i) ils ne font pas partie de la même clause et (ii) ils ne sont pas antagonistes (ils ne représentent pas une variable x_i et sa négation $\overline{x_i}$).
Pour l'exemple, le graphe correspondant est le suivant :



I_2 est un graphe à $3p$ sommets, c'est donc une instance de taille polynomiale en la taille de I_1. De plus, on fixe dans notre instance I_2 $k = p$. On vérifie alors l'équivalence des solutions :

(a) 3-SAT⇒ k-Clique : Si 3-SAT a une solution, on sélectionne un sommet correspondant à une variable vraie dans chaque clause, et l'ensemble de ces p sommets forme bien une clique. En effet, deux de ces sommets ne font pas partie de la même clause et ne sont pas antagonistes donc ils sont reliés par une arête.

(b) k-Clique⇒ 3-SAT : Si on a une clique de taille k dans I_2, alors il y a un sommet de la clique par clause (sinon les deux sommets ne sont pas reliés). On sélectionne ces sommets pour instancier les variables, et c'est cohérent car on ne fait pas de choix contradictoire (deux sommets antagonistes ne peuvent pas faire partie de la clique car ils ne sont pas reliés).

□

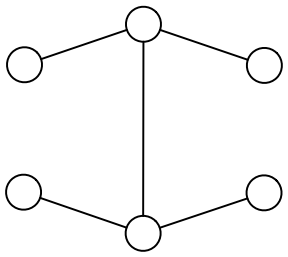
6.6 Couverture par les sommets

Définition 8 (VERTEX-COVER)**.** *Soit un graphe $G = (V, E)$, et un entier k : $1 \leq k \leq |V|$ Existe-t-il k sommets $v_{i_1}, ..., v_{i_k}$ tels que $\forall e \in E$, e est adjacente à l'un des v_{i_j} ?*

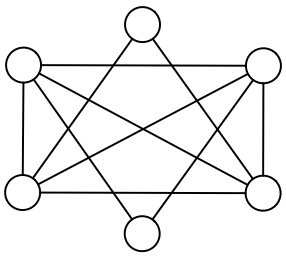
Théorème 13. *VERTEX-COVER est $\mathcal{NP}$-complet*

Preuve.

1. VERTEX-COVER $\in \mathcal{NP}$:
   Certificat : liste des k sommets V_k.
   Pour chaque arête $(v_1, v_2) \in E$, on vérifie si $v_1 \in V_k$ ou $v_2 \in V_k$. La vérification se fait en $|E| * k$, et c'est donc polynomial en la taille des données.
2. La réduction est triviale à partir de Clique : on transforme une instance de clique (graphe G, taille k) en l'instance de Vertex-cover ($\overline{G}$, $|V| - k$), où $\overline{G}$ est le graphe complémentaire de G, comme illustré sur l'exemple. L'équivalence est triviale :
   Le graphe G a une k-clique ⇔ Le complémentaire de G a une Vertex Cover de taille $|V| - k$.



Graphe G



Complémentaire $\overline{G}$

□

6.7 Cycle hamiltonien

Définition 9 (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 14. *CH est $\mathcal{NP}$-complet*

Preuve.

1. CH $\in \mathcal{NP}$: facile.
2. Réduction depuis 3-SAT (cf feuilles distribuées en cours : 1ère édition du Cormen) ou depuis vertex-cover (cf Cormen).

□

A partir de CH, on peut facilement montrer que TSP (le voyageur de commerce) est également NP-complet (cf feuilles distribuées en cours).

6.8 Coloration de graphes

Définition 10 (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 11 (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 12 (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.

6.8.1 COLOR

Théorème 15. *COLOR est NP-complet*

Preuve.

1. COLOR est dans NP.
   Une instance I de COLOR est une paire (G, k) de taille $\mid V \mid + \mid E \mid + log\ k$. Comme certificat on peut choisir une liste des couleurs des sommets. On peut vérifier en temps linéaire $O(\mid V \mid + \mid E \mid)$ si pour chaque arête (x, y) de E la condition $color(x) \neq color(y)$ est vraie et si la somme des couleurs utilisées est égale à k.

2. Réduction à partir de 3-SAT.
   On se donne une instance I_1 de 3-SAT avec p clauses $C_k,\ k = 1, \ldots, p$, et n variables $x_1, \ldots, x_n$. Comme on a le droit d'ignorer les instances «faciles» on peut supposer $n \geq 4$.
   A partir de cette instance on construit un graphe $G = (V, E)$ avec les $3n + p$ sommets

$$x_1, \ldots, x_n,\ \overline{x}_1, \ldots, \overline{x}_n,\ y_1, \ldots, y_n,\ c_1, \ldots, c_p$$

   et les arêtes

$$(x_i, \overline{x}_i)\ \forall i = 1, \ldots, n$$
$$(y_i, y_j)\ \forall i, j = 1, \ldots, n,\ i \neq j$$
$$(y_i, x_j)\ \forall i, j = 1, \ldots, n,\ i \neq j$$
$$(y_i, \overline{x}_j)\ \forall i, j = 1, \ldots, n,\ i \neq j$$
$$(x_i, c_k)\ \forall x_i \notin C_k$$
$$(\overline{x}_i, c_k)\ \forall \overline{x}_i \notin C_k.$$

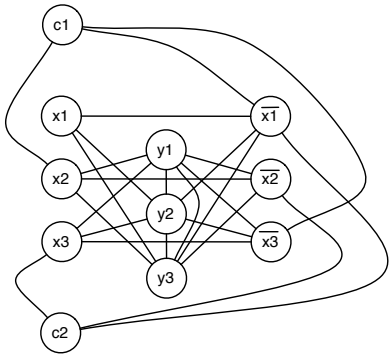


FIG. 6.1 – Exemple : Le graphe construit à partir de la formule $(x_1 \vee \overline{x_2} \vee x_3) \wedge (x_1 \vee x_2 \vee \overline{x_3})$.

   Un exemple pour la construction de ce graphe est donné en Figure 6.1. La taille de I_2 est polynomiale en la taille de I_1 comme $\mid V \mid = O(n + p)$.
   On va prouver que I_1 a une solution ssi G est coloriable avec $n + 1$ couleurs.

   $\Rightarrow$ Supposons que I_1 a une solution : il y a une instantiation des variables t.q. C_k est vraie pout tout k. On pose

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

$$color(x_i) = n+1 \text{ si } x_i \text{ est fausse}$$
$$color(\overline{x}_i) = i \text{ si } \overline{x}_i \text{ est vraie}$$
$$color(\overline{x}_i) = n+1 \text{ si } \overline{x}_i \text{ est fausse}$$
$$color(y_i) = i$$

et pour $color(c_k)$ on choisit la couleur d'une variable qui met la clause C_k à vraie.
Montrons que cette allocation des couleurs réalise une coloration valide du graphe. Le seul problème pourrait venir des arêtes entre les x (ou les $\overline{x}$) et les clauses. Soit C_k une clause coloriée à i à cause d'une variable x_i ou $\overline{x}_i$ qui la met à vraie, donc qui est aussi coloriée en i. Mais cette variable est dans la clause : alors l'arête entre cette variable et la clause n'existe pas. On a l'arête entre la clause et l'opposé de la variable, mais celle dernière est colorée à $n+1$, donc pas de conflit.

$\Leftarrow$ Supposons maintenant qu' il y a un coloriage de G à $n+1$ couleurs.

Lemme 1. Les y_i forment une clique, donc ont des couleurs différentes. Quitte à renuméroter les couleurs, $color(y_i) = i$.

Lemme 2. x_i et $\overline{x}_i$ ont pour couleur i ou $n+1$ et ce n' est pas la même.
En effet, x_i et $\overline{x}_i$ sont reliés entre eux et avec tous les y_j, $j \neq i$, on peut en déduire que $color(\begin{smallmatrix} x_i \\ \overline{x}_i \end{smallmatrix}) = (\begin{smallmatrix} i \\ n+1 \end{smallmatrix})$ ou $(\begin{smallmatrix} n+1 \\ i \end{smallmatrix})$

Lemme 3. c_k ne peut pas avoir la couleur $n+1$. En effet, $n \geq 4$, donc il y a un x_i t.q. $x_i \notin C_k$, $\overline{x}_i \notin C_k$, et l'un des deux est de couleur $n+1$.
Supposons alors que $color(c_i) = k \in 1, \ldots, n$. On pose

$$x_i = \begin{cases} 1 \text{ si } color(x_i) = i \\ 0 \text{ si } color(x_i) = n+1 \end{cases}$$

et (forcément)

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

Alors la variable qui donne à c_i sa couleur est dans la clause C_i (sinon ils sont reliés) et met cette clause à vraie, car sa couleur n'est pas $n+1$. Comme c'est vrai pour chaque clause C_i, I_1 a une solution.

□

6.8.2 3-COLOR

Théorème 16. *3-COLOR est NP-complet*

Remarque 1. *Le fait que COLOR est NP-complet ne dit rien. Par exemple, 2-COLOR est polynomial (voir Paragraphe 4.3).*

Preuve.

1. Comme COLOR est dans NP, 3-COLOR est dans NP.

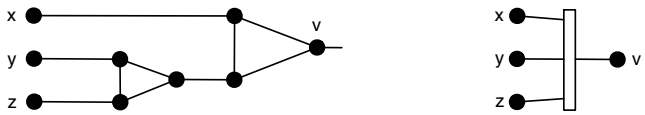


FIG. 6.2 – A gauche le gadget, à droite son symbole.

2. On fait une réduction à partir de 3-SAT en utilisant le gadget de la figure 6.2.

Lemme 7. *Le gadget vérifie les deux propriétés suivantes :*
i) Si $x = y = z = 0$ *alors* v *est colorié à* 0.
ii) Toute autre entrée $(x,\ y,\ z)$ *permet de colorier* v *à* 1 *ou* 2.

Preuve. i) Si $x = y = z = 0$ alors on ne peut que colorier le gadget comme suivant (fig. 6.3) :

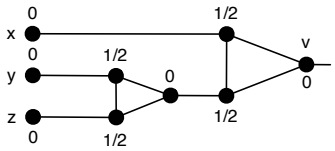


FIG. 6.3 – Si $x = y = z = 0$ alors $v = 0$

□

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. 6.4) : 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.

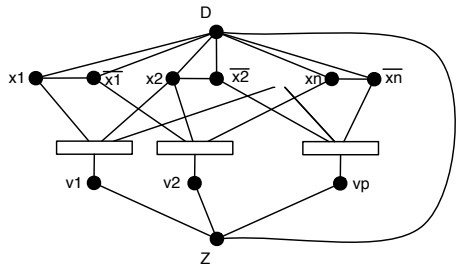


FIG. 6.4 – Un exemple de graphe G construit à partir de l'instance I_1 de 3-SAT. Le gadget k est relié avec les variables de la clause C_k de I_1.

On va prouver que I_1 a une solution ssi G est 3-coloriable.

(a) $\Rightarrow$ On suppose que I_1 a une solution, c'est à dire une instantiation qui met toutes les clauses à vrai. On pose $color(x_i) = \begin{cases} 1 \text{ si } x_i \text{ vrai} \\ 0 \text{ sinon} \end{cases}$ et $color(\overline{x}_i) = \begin{cases} 1 \text{ si } \overline{x}_i \text{ vrai} \\ 0 \text{ sinon} \end{cases}$.
Comme il y a une variable à vrai dans chaque clause, il y a pour chaque gadget une entrée qui est non nulle. La propriété ii) du gadget permet alors d'avoir $v_i \neq 0 \forall i =$

1..p. En posant $color(D) = 2$ et $color(Z) = 0$ on obtient un coloriage de G à trois couleurs.

(b) $\Leftarrow$ On suppose que G a une coloration à trois couleurs. En permutant les couleurs on peut supposer que $color(D) = 2$, $color(Z) = 0$, $color(\begin{smallmatrix} x_i \\ \overline{x}_i \end{smallmatrix}) = (\begin{smallmatrix} 1 \\ 0 \end{smallmatrix})$ ou $(\begin{smallmatrix} 0 \\ 1 \end{smallmatrix})$ car le coloriage est valide. Comme $color(Z) = 0$ la couleur des v_i doit être 1 ou 2. La propriété i) permet de conclure que dans chaque clause il y a une variable à vrai (on ne peut pas avoir trois 0 en entrée). Par conséquence, l'instantiation $x_i = \begin{cases} 1 \text{ si } color(x_i) = 1 \\ 0 \text{ si } color(x_i) = 0 \end{cases}$, $\overline{x}_i = \begin{cases} 1 \text{ si } color(\overline{x}_i) = 1 \\ 0 \text{ si } color(\overline{x}_i) = 0 \end{cases}$ donne une instantiation des x_i qui met toutes les clauses à vrai.

□

6.8.3 3-COLOR-PLAN

Théorème 17. *3-COLOR-PLAN est NP-complet*

Preuve.

1. Comme 3-COLOR est dans NP, 3-COLOR-PLAN est dans NP.

2. On fait une réduction à partir de 3-COLOR en utilisant le gadget de la figure 6.5, appelé W.

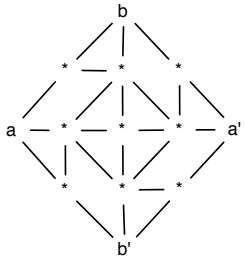


FIG. 6.5 – Le gadget W avec 13 sommets

Lemme 8. *Le gadget W vérifie les deux propriétés suivantes :*
i) Pour tout 3-coloriage de W les sommets a et a' ainsi que b et b' ont la même couleur.
ii) Si (a, a') et (b, b') ont la même couleur donnée, on peut compléter ce coloriage en un 3-coloriage de W.

Preuve.
i) On procède par exploration exhaustive des coloriages possibles. Les coloriages montrés en ii) donnent des exemples.
ii) Si on donne par exemple $a = a' = 0$, $b = b' = 1$, on peut compléter le coloriage comme dans la figure 6.6. Si on donne $a = a' = 0$, $b = b' = 0$, on peut compléter le coloriage comme dans la figure 6.7. Les autres cas sont symétriques.

□

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

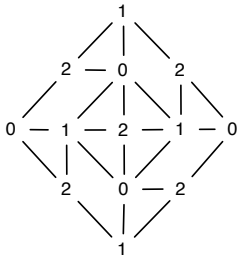


FIG. 6.6 – 3-coloriage de W avec $a = a' = 0$, $b = b' = 1$

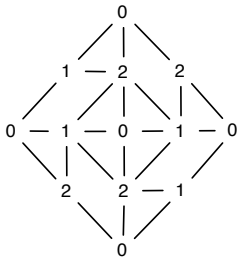


FIG. 6.7 – 3-coloriage de W avec $a = a' = 0$, $b = b' = 0$

par un gadget W (voir fig. 6.8). Précisément, une arête (u, v) qui croise (u', v') est remplacée 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.

□

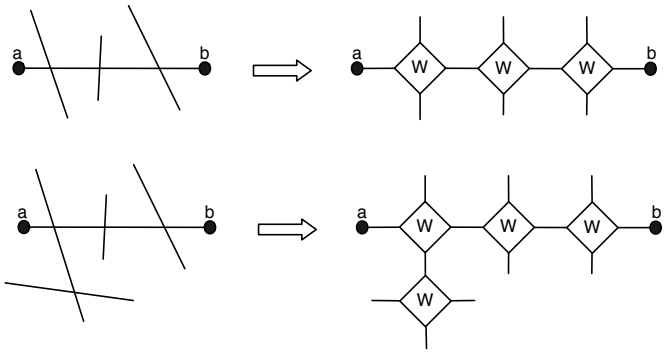


FIG. 6.8 – On transforme le graphe G en un graphe planaire en remplaçant les arêtes qui se croisent par le gadget W

6.9 Références bibliographiques

Ce chapitre s'inspire du célèbre livre de Garey et Johnson [4]. 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 [8], dont la première édition est mise par l'auteur à la libre disposition de tous sur le Web : `http://www.cis.upenn.edu/~wilf/`.

Bibliographie

[1] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. The MIT Press, 1990.

[2] A. Darte and S. Vaudenay. *Algorithmique et optimisation : exercices corrigés*. Dunod, 2001.

[3] D. Froidevaux, M. Gaudel, and D. Soria. *Types de données et algorithmes*. McGraw-Hill, 1990.

[4] M. R. Garey and D. S. Johnson. *Computers and Intractability, a Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, 1979.

[5] D. E. Knuth. *The Art of Computer Programming ; volumes 1-3*. Addison-Wesley, 1997.

[6] G. J. E. Rawlins. *Compared to what ? : an introduction to the analysis of algorithms*. Computer Science Press, Inc, 1992.

[7] D. B. West. *Introduction to Graph Theory*. Prentice Hall, 1996.

[8] H. S. Wilf. *Algorithms and Complexity*. A.K. Peter, 1985. Available at `http://www.cis.upenn.edu/~wilf/`.