Algorithmique avancée

IUP 2

Frédéric Vivien

24 avril 2002

Table des matières

Table des figures

Avertissement

Ce cours ne traite pas :

– la recherche de motifs dans une chaîne de caractères (problème supposé être abondamment traité dans le cours de *Programmation Orientée Objet*) ;
– l'algorithmique géométrique (supposée être traitée dans les cours liés au traitement d'image).

Chapitre 1

Introduction

1.1 Qu'est-ce que l'algorithmique ?

Définition 1 (Algorithme). *Un algorithme est suite finie d'opérations élémentaires constituant un schéma de calcul ou de résolution d'un problème.*

Historique : Le mot « algorithme » provient de la forme latine (*Algorismus*) du nom du mathématicien arabe AL-KHAREZMI ou AL-KHWĀRIZMĪ auteur –entre autres mais ce n' est pas le plus important– d'un manuel de vulgarisation sur le calcul décimal positionnel indien (v. 830) expliquant son utilisation et, surtout, la manipulation des différents algorithmes permettant de réaliser les opérations arithmétiques classiques (addition, soustraction, multiplication, division, extraction de racines carrées, règle de trois, etc.).

Double problématique de l'algorithmique

1. **Trouver une méthode** de résolution (exacte ou approchée) du problème.
   - Soient trois nombres réels a, b et c, quelles sont les solutions de l'équation $ax^2 + bx + c$? (Résultat bien connu.)
   - Soient cinq nombres réels a, b, c, d et e, quelles sont les solutions de l'équation $ax^5 + bx^4 + cx^3 + dx^2 + ex + f$? (Pas de méthode générale, cf. la théorie de GALOIS.)
2. Trouver une méthode **efficace**.

Savoir résoudre un problème est une chose, le résoudre efficacement en est une autre, comme nous allons le voir à la section 1.2.

Différences entre algorithmes et programmes

Un **programme** est la réalisation (l'implémentation) d'un algorithme au moyen d'un langage donné (sur une architecture donnée). Il s'agit de la mise en œuvre du principe. Par exemple, lors de la programmation on s'occupera parfois explicitement de la gestion de la mémoire (allocation dynamique en C) qui est un problème d'implémentation ignoré au niveau algorithmique.

1.2 Motivation : calcul de x^n

1.2.1 Problème

Données : un entier naturel n et un réel x. On veut calculer x^n.

Moyens : Nous partons de $y_1 = x$. Nous allons construire une suite de valeurs y_1, ..., y_m telle que la valeur y_k soit obtenue par multiplication de deux puissances de x précédemment calculées : $y_k = y_u \times y_v$, avec $1 \leq u, v < k$, $k \in [2, m]$.

But : $y_m = x^n$. Le **coût** de l'algorithme sera alors de $m-1$, le nombre de multiplications faites pour obtenir le résultat recherché.

1.2.2 Algorithme trivial

$y_i = y_{i-1} \times y_1, i \in [2, n]$. Résultat : $y_n = x^n$. Coût : $m - 1 = n - 1$ multiplications.

Algorithme

y[1] = x
Pour $i \leftarrow 2$ **à** n **faire**
  y[i] = y[$i-1$] × y[1]
renvoyer y[n]

1.2.3 Méthode binaire

Algorithme

1. Écrire n sous forme binaire
2. Remplacer chaque :
   – « 1 » par la paire de lettres « `SX` » ;
   – « 0 » par la lettre « `S` ».
3. Éliminer la paire « `SX` » la plus à gauche.
4. Résultat : un mode de calcul de x^n où
   – `S` signifie « élever au carré » (*squaring*) ;
   – `X` signifie « multiplier par x ».
   Le tout en partant de x.

Illustration avec $n = 23$

1. $n = 10111$
   `1  0  1  1  1`
2. `SX S SX SX SX`
3. `S SX SX SX`
4. Nous partons de x et nous obtenons successivement :
   x^2, x^4, x^5, x^{10}, x^{11}, x^{22}, x^{23}.
   Nous sommes donc capables de calculer x^{23} en 7 multiplications au lieu de 22 !

Explication de la méthode

– Écriture binaire de n : $n = \sum_{i=0}^{i=p} a_i 2^i$.
– Plaçons nous au cours du calcul de puissances de x. Soit j le dernier bit de la représentation binaire de n qui ait été « décodé » et soit y_j le dernier résultat obtenu. Initialement, $j = p$ et $y_p = x = x^{a_p}$.
– Deux cas sont possibles pour a_{j-1} :
   1. $a_{j-1} = 1$. a_{j-1} est remplacé par `SX`, nous élevons y_j au carré puis multiplions le résultat par x. Le nouveau résultat est $y_{j-1} = y_j^2 \times x$.
   2. $a_{j-1} = 0$. a_{j-1} est remplacé par `S` et nous élevons simplement y_j au carré. Le nouveau résultat est $y_{j-1} = y_j^2$.

   Dans tous les cas nous avons : $y_{j-1} = y_j^2 \times (x^{a_{j-1}})$.
– D'où, $y_{p-1} = y_p^2 \times (x^{a_{p-1}}) = (x^{a_p})^2 \times (x^{a_{p-1}}) = (x^{2 \times a_p}) \times (x^{a_{p-1}}) = (x^{2 \times a_p + a_{p-1}})$. Par récurrence, nous pouvons montrer que $y_1 = x^{\sum_{i=0}^{i=p} a_i 2^i} = x^n$...

Complexité (coût)

Note : les nombres dont la représentation binaire a exactement p chiffres forment exactement l'intervalle $[2^{p-1}, 2^p - 1]$.

Nombres de chiffres dans l'écriture binaire de n : $1 + [\log_2 n]$. Notons $\nu(n)$ le nombre de « 1 » dans l'écriture binaire de n. Nombre d'opérations effectuées :
- $(1 + [\log_2 n]) - 1$ élévations au carré (ne pas oublier l'étape 3) ;
- $\nu(n) - 1$ multiplications par x (ne pas oublier l'étape 3).

Soit en tout $T(n) = [\log_2 n] + \nu(n) - 1$ multiplications. Trivialement, $1 \leq \nu(n) \leq [\log_2 n]$ et $[\log_2 n] \leq T(n) \leq 2[\log_2 n]$.

Pour $n = 1000$, l'algorithme trivial effectue 999 multiplications, et la méthode binaire moins de 20.

Historique

Cette méthode a été présentée avant 200 avant J.C. en Inde, mais il semblerait qu'il ait fallu attendre un millénaire avant que cette méthode ne soit connue en dehors de l'Inde [3, p. 441].

Peut-on faire mieux ?

Prenons le cas $n = 15$.

1. $n = 1111$
```
   1  1  1  1
```
2.
```
   SX SX SX SX
```
3.
```
      SX SX SX
```
4. Nous partons de x et nous obtenons successivement : x^2, x^3, x^6, x^7, x^{14}, x^{15}. Nous sommes donc capables de calculer x^{15} en 6 multiplications.

Autre schéma de calcul : x^2, x^3, x^6, x^{12}, $x^{15} = x^{12} \times x^3$. Nous obtenons ainsi x^{15} en 5 multiplications et la méthode binaire n'est donc pas *optimale* (c'est-à-dire que l'on peut faire mieux).

1.2.4 Algorithme des facteurs

Algorithme

$$x^n = \begin{cases} x & \text{si } n = 1; \\ x^{n-1} \times x & \text{si } n \text{ premier}; \\ (x^p)^{n'} & \text{si } n = p \times n' \text{ avec } p \text{ plus petit diviseur premier de } n. \end{cases}$$

Illustration avec $n = 15$

1. $15 = 3 \times 5$, 3 étant le plus petit diviseur (facteur) premier de 15. Donc $x^{15} = (x^3)^5$.
   Nous réappliquons l'algorithme pour calculer x^3 et y^5, où $y = x^3$.
2. Calcul de x^3 :
   (a) 3 est premier. Donc $x^3 = x^2 \times x$. Nous réappliquons l'algorithme pour calculer x^2.
   (b) 2 est premier. Donc $x^2 = x \times x$.
   (c) Finalement, x^3 est calculé comme suit : $x^3 = x \times x \times x$, soit en deux multiplications.
3. Calcul de y^5 :
   (a) 5 est premier. Donc $y^5 = y^4 \times y$. Nous réappliquons l'algorithme pour calculer y^4.
   (b) $4 = 2 \times 2$, où 2 est le plus petit facteur premier de 4. Donc $y^4 = (y^2)^2$.
   (c) Finalement y^5 est calculé comme suit : $t = y \times y$, $u = t \times t$, $y^5 = u \times y$, soit en 3 multiplications.
4. Finalement, x^{15} est calculé en 5 multiplications.

Peut-on faire mieux ?

Oui...

1.2.5 Algorithme de l'arbre

Le $k+1^{e}$ niveau de l'arbre est défini comme suit :

- on suppose que l'on a déjà les k premiers niveaux ;
- on construit le $k+1^{e}$ de la gauche vers la droite en ajoutant sous le nœud n les nœuds de valeur $n+1$, $n+a_1$, ..., $n+a_{k-1}$ où 1, a_1, ..., a_{k-1} est le chemin de la racine au nœud n ;
- on supprime tous les nœuds qui dupliquent une valeur déjà obtenue.

Cf. la figure 1.1.

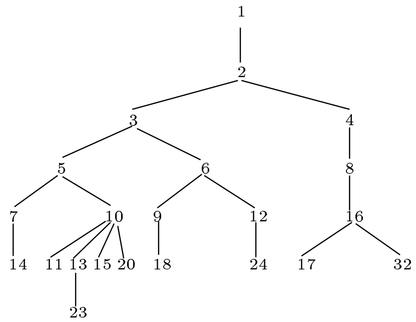


FIG. 1.1 – Arbre de puissances (minimisant le nombre de multiplications pour $n \leq 76$ [3]).

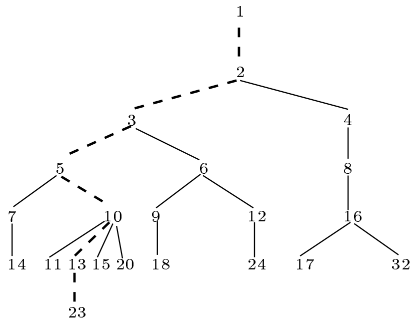


FIG. 1.2 – Schéma de calcul pour $n = 23$.

Illustration avec $n = 23$

Sur la figure 1.2 nous pouvons constater que cette méthode permet de calculer x^{23} en 6 multiplications, au lieu de 7 pour la méthode binaire et celle des facteurs... Cette méthode n'est optimale que pour $n \leq 76$.

1.2.6 Et après ?

KNUTH [3] consacre près de 26 pages à ce problème...

Moralité : nous avons affaire à un problème simple, que tout le monde sait résoudre, mais qu'il est très difficile de résoudre efficacement...

Dans ce cours nous verrons des problèmes classiques, des méthodes classiques de résolutions (qui ne résoudrons pas tout, loin s'en faut), des structures de données classiques.

1.3 Conclusion

Pour conclure, citons [2] : « Un bon algorithme est comme un couteau tranchant —il fait exactement ce que l'on attend de lui, avec un minimum d'efforts. L'emploi d'un mauvais algorithme pour résoudre un problème revient à essayer de couper un steak avec un tournevis : vous finirez sans doute par obtenir un résultat digeste, mais vous accomplirez beaucoup plus d'efforts que nécessaire, et le résultat aura peu de chances d'être esthétiquement satisfaisant. »

Chapitre 2

Complexité et optimalité ; premier algorithme de tri

2.1 Définition de la complexité

2.1.1 Notations de Landau

Quand nous calculerons la complexité d'un algorithme, nous ne calculerons généralement pas sa complexité exacte, mais son ordre de grandeur. Pour ce faire, nous avons besoin de notations asymptotiques.

$$\begin{array}{cccccl} \mathbf{O} & : & f = O(g) & \Leftrightarrow & \exists n_0, \exists c \geq 0, \forall n \geq n_0, f(n) \leq c \times g(n) \\ \boldsymbol{\Omega} & : & f = \Omega(g) & \Leftrightarrow & g = O(f) \\ \mathbf{o} & : & f = o(g) & \Leftrightarrow & \forall c \geq 0, \exists n_0, \forall n \geq n_0, f(n) \leq c \times g(n) \\ \boldsymbol{\Theta} & : & f = \Theta(g) & \Leftrightarrow & f = O(g) \text{ et } g = O(f) \end{array}$$

Exemples

O : $n = O(n)$, $2n = O(3n)$, $n + 2 = O(n)$ (pour s'en convaincre, prendre $n_0 = 2$ et $c = 2$), $\sqrt{n} = O(n)$, $\log(n) = O(n)$, $n = O(n^2)$.

o : $\sqrt{n} = o(n)$, $\log(n) = o(n)$, $n = o(n^2)$, $\log(n) = o(\sqrt{n})$.

Θ : $n + \log(n) = \Theta(n + \sqrt{n})$.

2.1.2 Complexité

Définition 2 (Complexité). *La complexité d'un algorithme est la mesure du nombre d'opérations fondamentales qu'il effectue sur un jeu de données. La complexité est exprimée comme une fonction de la taille du jeu de données.*

Nous notons D_n l'ensemble des données de taille n et $T(d)$ le coût de l'algorithme sur la donnée d.

Complexité au meilleur : $T_{\min}(n) = \min_{d \in D_n} C(d)$. C'est le plus petit nombre d'opérations qu'aura à exécuter l'algorithme sur un jeu de données de taille fixée, ici à n. C'est une borne inférieure de la complexité de l'algorithme sur un jeu de données de taille n.

Complexité au pire : $T_{\max}(n) = \max_{d \in D_n} C(d)$. C'est le plus grand nombre d'opérations qu'aura à exécuter l'algorithme sur un jeu de données de taille fixée, ici à n.

Avantage : il s'agit d'un maximum, et l'algorithme finira donc toujours avant d'avoir effectué $T_{\max}(n)$ opérations.

Inconvénient : cette complexité peut ne pas refléter le comportement « usuel » de l'algorithme, le pire cas pouvant ne se produire que très rarement, mais il n'est pas rare que le cas moyen soit aussi mauvais que le pire cas.

Complexité en moyenne : $T_{\text{moy}}(n) = \frac{\sum_{d \in D_n} C(d)}{|D_n|}$. C'est la moyenne des complexités de l'algorithme sur des jeux de données de taille n (en toute rigueur, il faut bien évidemment tenir compte de la probabilité d'apparition de chacun des jeux de données).

Avantage : reflète le comportement « général » de l'algorithme si les cas extrêmes sont rares ou si la complexité varie peu en fonction des données.

Inconvénient : la complexité en pratique sur un jeu de données particulier peut être nettement plus importante que la complexité en moyenne, dans ce cas la complexité en moyenne ne donnera pas une bonne indication du comportement de l'algorithme.

En pratique, nous ne nous intéresserons qu'à la complexité au pire et à la complexité en moyenne.

Définition 3 (Optimalité). *Un algorithme est dit* **optimal** *si sa complexité est la complexité minimale parmi les algorithmes de sa classe.*

Nous nous intéresserons quasi exclusivement à la *complexité en temps* des algorithmes. Il est parfois intéressant de s'intéresser à d'autres de leurs caractéristiques, comme la *complexité en espace* (taille de l'espace mémoire utilisé), la largeur de bande passante requise, etc.

2.1.3 Modèle de machine

Pour que le résultat de l'analyse d'un algorithme soit pertinent, il faut avoir un modèle de la machine sur laquelle l'algorithme sera implémenté (sous forme de programme). On prendra comme référence un modèle de **machine à accès aléatoire (RAM)** et à processeur unique, où les instructions sont exécutées l'une après l'autre, sans opérations simultanées.

2.2 Illustration : cas du tri par insertion

2.2.1 Problématique du tri

Entrée : une séquence de n nombres, a_1, ..., a_n.

Sortie : une permutation, a'_1, ..., a'_n, de la séquence d'entrée, telle que $a'_1 \leq a'_2 \leq ... \leq a'_n$.

2.2.2 Principe du tri par insertion

De manière répétée, on retire un nombre de la séquence d'entrée et on l'insère à la bonne place dans la séquence des nombres déjà triés (ce principe est le même que celui utilisé pour trier une poignée de cartes).

2.2.3 Algorithme

Tri-Insertion

Pour $j \leftarrow 2$ **à** n **faire**	
clé $\leftarrow$ A[j]	*On retire un nombre de la séquence d'entrée*
$i \leftarrow j - 1$	*Les $j-1$ premiers éléments de A sont déjà triés.*
tant que $i > 0$ **et** A[i] > clé **faire**	*Tant que l'on n'est pas arrivé au début du tableau, et que l'élément courant est plus grand que celui à insérer.*
A[$i+1$] $\leftarrow$ A[i]	*On décale l'élément courant (on le met dans* **la** *place vide).*
$i \leftarrow i - 1$	*On s'intéresse à l'élément précédent.*
A[$i+1$] $\leftarrow$ clé	*Finalement, on a trouvé où insérer notre nombre.*

2.2.4 Exemple

Les différentes étapes de l'exécution de l'algorithme Tri-Insertion sur le tableau $[5; 2; 4; 6; 1; 3]$ sont présentées figure 2.1.

Entrée	5	2	4	6	1	3
$j = 2$	5	2	4	6	1	3
$j = 3$	2	5	4	6	1	3
$j = 4$	2	4	5	6	1	3
$j = 5$	2	4	5	6	1	3
$j = 6$	1	2	4	5	6	3
Sortie	1	2	3	4	5	6

Fig. 2.1 – Action de Tri-Insertion sur le tableau $[5; 2; 4; 6; 1; 3]$; l'élément à insérer est entouré par un cercle.

2.2.5 Complexité

Nous passons en revue les différentes étapes de notre algorithme afin d'évaluer son temps d'exécution. Pour ce faire, nous attribuons un coût en temps à chaque instruction, et nous comptons le nombre d'exécutions de chacune des instructions. Pour chaque valeur de $j \in [2, n]$, nous notons t_j le nombre d'exécutions de la boucle `tant que` pour cette valeur de j. Il est à noter que la valeur de t_j **dépend des données**...

Tri-Insertion	*Coût*	*Nombre d'exécutions*
Pour $j \leftarrow 2$ **à** n **faire**	c_1	n
clé $\leftarrow$ A$[j]$	c_2	$n-1$
$i \leftarrow j-1$	c_3	$n-1$
tant que $i > 0$ **et** A$[i] >$ clé **faire**	c_4	$\sum_{j=2}^{n} t_j$
A$[i+1] \leftarrow$ A$[i]$	c_5	$\sum_{j=2}^{n}(t_j-1)$
$i \leftarrow i-1$	c_6	$\sum_{j=2}^{n}(t_j-1)$
A$[i+1] \leftarrow$ clé	c_7	$n-1$

Le temps d'exécution total de l'algorithme est alors :

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4\sum_{j=2}^{n} t_j + c_5\sum_{j=2}^{n}(t_j-1) + c_6\sum_{j=2}^{n}(t_j-1) + c_7(n-1)$$

Complexité au meilleur : le cas le plus favorable pour l'algorithme Tri-Insertion est quand le tableau est déjà trié, comme le montre le cas $j = 4$ de la figure 2.1. Dans ce cas $t_j = 1$ pour tout j.

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_3(n-1) + c_4(n-1) + c_7(n-1) \\ &= (c_1 + c_2 + c_3 + c_4 + c_7)n - (c_2 + c_3 + c_4 + c_7). \end{aligned}$$

$T(n)$ peut ici être écrit sous la forme $T(n) = an + b$, a et b étant des *constantes* indépendantes des entrées, et $T(n)$ est donc une **fonction linéaire** de n.

Le plus souvent, comme c'est le cas ici, le temps d'exécution d'un algorithme est fixé pour une entrée donnée ; mais il existe des algorithmes « aléatoires » intéressants dont le comportement peut varier même pour une entrée fixée. Nous verrons un algorithme de ce style au chapitre 4 : une version « aléatoire » du *tri rapide*

Complexité au pire : le cas le plus défavorable pour l'algorithme TRI-INSERTION est quand le tableau est déjà trié dans l'ordre inverse, comme le montre le cas $j = 5$ de la figure 2.1. Dans ce cas $t_j = j$ pour tout j.

Rappel : $\sum_{j=1}^{n} j = \frac{n(n+1)}{2}$. Donc $\sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1$ et $\sum_{j=2}^{n}(j-1) = \frac{n(n-1)}{2}$.

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_3(n-1) + c_4\left(\frac{n(n+1)}{2} - 1\right) + c_5\left(\frac{n(n-1)}{2}\right) + c_6\left(\frac{n(n-1)}{2}\right) + c_7(n-1) \\ &= \left(\frac{c_4}{2} + \frac{c_5}{2} + \frac{c_6}{2}\right) n^2 + \left(c_1 + c_2 + c_3 + \frac{c_4}{2} - \frac{c_5}{2} - \frac{c_6}{2} + c_7\right) n - (c_2 + c_3 + c_4 + c_7). \end{aligned}$$

$T(n)$ peut ici être écrit sous la forme $T(n) = an^2 + bn + c$, a, b et c étant des constantes, et $T(n)$ est donc une **fonction quadratique** de n.

Complexité en moyenne : supposons que l'on applique l'algorithme de tri par insertion à n nombres choisis au hasard. Quelle sera la valeur de t_j ? C'est-à-dire, où devra-t-on insérer A[j] dans le sous-tableau A[1..$j-1$] ? En moyenne, pour moitié les éléments de A[1..$j-1$] sont inférieurs à A[j], et pour moitié supérieurs. Donc $t_j = j/2$. Si l'on reporte cette valeur dans l'équation définissant $T(n)$, on obtient, comme dans le pire cas, une fonction quadratique en n.

Caveat : ce raisonnement est partiellement faux ; un raisonnement précis doit bien évidemment tenir compte des *valeurs* des éléments déjà triés. Pour un calcul précis, voir KNUTH [4, p. 82]. CORI et LÉVY [1, p. 26] font un autre raisonnement et trouve un autre résultat (de même ordre de grandeur). Les deux sont justes : tout dépend de l'hypothèse que l'on prend sur les jeux de données. Ainsi [1] suppose que les permutations sont équiprobables, et [4] que les valeurs à trier sont équiprobables...

Ordre de grandeur

Ce qui nous intéresse vraiment, c'est l'ordre de grandeur du temps d'exécution. Seul le terme dominant de la formule exprimant la complexité nous importe, les termes d'ordres inférieurs n'étant pas significatifs quand n devient grand. On ignore également le coefficient multiplicateur constant du terme dominant. On écrira donc, à propos de la complexité du tri par insertion :

meilleur cas : $\Theta(n)$.

pire cas : $\Theta(n^2)$.

en moyenne : $\Theta(n^2)$.

En général, on considère qu'un algorithme est plus efficace qu'un autre si sa complexité dans le pire cas a un ordre de grandeur inférieur.

Classes de complexité

Les algorithmes usuels peuvent être classés en un certain nombre de grandes classes de complexité :

- Les algorithmes sub-linéaires dont la complexité est en général en $O(\log n)$.
- Les algorithmes linéaires en complexité $O(n)$ et ceux en complexité en $O(n \log n)$ sont considérés comme rapides.
- Les algorithmes polynomiaux en $O(n^k)$ pour $k > 3$ sont considérés comme lents, sans parler des algorithmes exponentiels (dont la complexité est supérieure à tout polynôme en n) que l'on s'accorde à dire impraticables dès que la taille des données est supérieure à quelques dizaines d'unités.

Chapitre 3

La récursivité et le paradigme « diviser pour régner »

3.1 Récursivité

De l'art d'écrire des programmes qui résolvent des problèmes que l'on ne sait pas résoudre soi-même !

3.1.1 Définition

Définition 4 (Définition récursive, algorithme récursif). *Une définition récursive est une définition dans laquelle intervient ce que l'on veut définir. Un algorithme est dit récursif lorsqu'il est défini en fonction de lui-même.*

Dans le cadre de ce cours, nous ne nous intéresserons qu'aux programmes et algorithmes récursifs. Mais la notion de définition récursive est beaucoup plus générale :

en mathématiques : définition de l'exponentielle : $\forall x \in \mathbb{R}, f'(x) = f(x)$ et $f(0) = 1$.

en programmation : définition en `Ocaml` d'une liste infinie dont tous les éléments valent 1 :

let rec z = 1::z ; ;

3.1.2 Récursivité simple

Revenons à la fonction puissance $x \mapsto x^n$. Cette fonction peut être définie récursivement :

$$x^n = \begin{cases} 1 & \text{si } n = 0; \\ x \times x^{n-1} & \text{si } n \geq 1. \end{cases}$$

L'algorithme correspondant s'écrit :

PUISSANCE (x, n)
Si $n = 0$ **alors** renvoyer 1
sinon renvoyer $x \times$PUISSANCE$(x, n-1)$

3.1.3 Récursivité multiple

Une définition récursive peut contenir plus d'un appel récursif. Nous voulons calculer ici les combinaisons C_n^p en se servant de la relation de Pascal :

$$C_n^p = \begin{cases} 1 & \text{si } p = 0 \text{ ou } p = n; \\ C_{n-1}^p + C_{n-1}^{p-1} & \text{sinon.} \end{cases}$$

L'algorithme correspondant s'écrit :

COMBINAISON (n, p)
Si $p = 0$ ou $p = n$ **alors** renvoyer 1
sinon renvoyer COMBINAISON $(n-1, p)$ + COMBINAISON $(n-1, p-1)$

Bref, rien de particulier...

3.1.4 Récursivité mutuelle

Des définitions sont dites *mutuellement récursives* si elles dépendent les unes des autres. Ça peut être le cas pour la définition de la parité :

$$\text{pair}(n) = \begin{cases} \textit{vrai} & \text{si } n = 0; \\ \text{impair}(n-1) & \text{sinon;} \end{cases} \quad \text{et} \quad \text{impair}(n) = \begin{cases} \textit{faux} & \text{si } n = 0; \\ \text{pair}(n-1) & \text{sinon.} \end{cases}$$

Les algorithmes correspondants s'écrivent :

PAIR (n)
Si $n = 0$ **alors** renvoyer *vrai*
sinon renvoyer IMPAIR $(n-1)$

IMPAIR (n)
Si $n = 0$ **alors** renvoyer *faux*
sinon renvoyer PAIR $(n-1)$

3.1.5 Récursivité imbriquée

La fonction d'Ackermann est définie comme suit :

$$A(m,n) = \begin{cases} n+1 & \text{si } m = 0 \\ A(m-1,1) & \text{si } m > 0 \text{ et } n = 0 \\ A(m-1, A(m, n-1)) & \text{sinon} \end{cases}$$

d'où l'algorithme :

ACKERMANN(m, n)
si $m = 0$
alors $n + 1$
sinon si $n = 0$ **alors** ACKERMANN$(m-1, 1)$
sinon ACKERMANN$(m-1,$ ACKERMANN$(m, n-1))$

En résumé : on peut utiliser la récursivité comme l'on veut, à peu près n'importe comment...

3.1.6 Principe et dangers de la récursivité

Principe et intérêt : ce sont les mêmes que ceux de la démonstration par récurrence en mathématiques. On doit avoir :
- un certain nombre de cas dont la résolution est connue, ces « cas simples » formeront les cas d'arrêt de la récursion ;
- un moyen de se ramener d'un cas « compliqué » à un cas « plus simple ».

La récursivité permet d'écrire des algorithmes concis et élégants.

Difficultés :
- la définition peut être dénuée de sens :
  Algorithme A(n)
  renvoyer A(n)
- il faut être sûrs que l'on retombera toujours sur un cas connu, c'est-à-dire sur un cas d'arrêt ; il nous faut nous assurer que la fonction est complètement définie, c'est-à-dire, qu'elle est définie sur tout son domaine d'applications.

Moyen : existence d'un ordre strict tel que la suite des valeurs successives des arguments invoqués par la définition soit strictement monotone et finit toujours par atteindre une valeur pour laquelle la solution est explicitement définie.

L'algorithme ci-dessous teste si a est un diviseur de b.

DIVISEUR (a,b)
  Si $a \leq 0$ **alors** *Erreur*
    sinon si $a \geq b$ **alors** $a = b$ (test d'égalité)
      sinon DIVISEUR$(a,b-a)$

La suite des valeurs b, $b-a$, $b-2\times a$, etc. est strictement décroissante, car a est strictement positif, et on finit toujours pas aboutir à un couple d'arguments (a,b) tel que $b-a$ est négatif, cas défini explicitement.

Cette méthode ne permet pas de traiter tous les cas :

SYRACUSE(n)
  Si $n = 0$ ou $n = 1$ **alors** 1
    sinon si $n \mod 2 = 0$ **alors** SYRACUSE $(n/2)$
      sinon SYRACUSE $(3\times n+1)$

Problème ouvert : l'algorithme est bien défini et vaut 1 sur $\mathbb{N}$.

Question : N'y a-t-il vraiment aucun moyen de déterminer automatiquement si un algorithme récursif quelconque va terminer ? Réponse à la section suivante...

3.1.7 Non décidabilité de la terminaison

Question : peut-on écrire un programme qui vérifie automatiquement si un programme donné `P` termine quand il est exécuté sur un jeu de données `D` ?

Entrée Un programme `P` et un jeu de données `D`.

Sortie *vrai* si le programme `P` termine sur le jeu de données `D`, et *faux* sinon.

Démonstration de la non décidabilité

Supposons qu'il existe un tel programme, nommé `termine`, de vérification de la terminaison. À partir de ce programme on conçoit le programme `Q` suivant :

`programme Q`
  résultat = `termine`(`Q`,$\emptyset$)
  tant que résultat = *vrai* **faire** attendre une seconde **fin tant que**
  renvoyer résultat

Supposons que le programme `Q` —qui ne prend pas d'arguments— termine. Donc `termine`(`Q`,$\emptyset$) renvoie *vrai*, la deuxième instruction de `Q` boucle indéfiniment et `Q` ne termine pas. Il y a donc contradiction et le programme `Q` ne termine pas. Donc, `termine`(`Q`,$\emptyset$) renvoie *faux*, la deuxième instruction de `Q` ne boucle pas, et le programme `Q` termine normalement. Il y a une nouvelle fois contradiction : par conséquent, il n'existe pas de programme tel que `termine`, c'est-à-dire qui vérifie qu'un programme termine ou non sur un jeu de données...

Le problème de la terminaison est indécidable !

Petit historique : cf. [1, p. 48].

3.1.8 Importance de l'ordre des appels récursifs

Fonction qui affiche les entiers par ordre décroissant, de n jusqu'à 1 :

DÉCROISSANT(n)
Si $n = 0$ **alors** ne rien faire
  sinon afficher n
    DÉCROISSANT$(n-1)$

Exécution pour $n = 2$:
  Appel de DÉCROISSANT(2)
    Affichage de 2.
    Appel de DÉCROISSANT(1)
      Affichage de 1.
      Appel de DÉCROISSANT(0)
        L'algorithme ne fait rien.

Résultat affichage d'abord de 2 puis de 1 : l'affichage a lieu dans l'ordre décroissant.

Intervertissons maintenant l'ordre de l'affichage et de l'appel récursif :

CROISSANT(n)
Si $n = 0$ **alors** ne rien faire
    sinon CROISSANT($n - 1$)
        afficher n

Exécution pour $n = 2$:
  Appel de CROISSANT(2)
    Appel de CROISSANT(1)
      Appel de CROISSANT(0)
        L'algorithme ne fait rien.
      Affichage de 1.
    Affichage de 2.

Résultat affichage d'abord de 1 puis de 2 : l'affichage a lieu dans l'ordre croissant.

3.1.9 Exemple d'algorithme récursif : les tours de Hanoï

Le problème

Le jeu est constitué d'une plaquette de bois où sont plantées trois tiges. Sur ces tiges sont enfilés des disques de diamètres tous différents. Les seules règles du jeu sont que l'on ne peut déplacer qu'un seul disque à la fois, et qu'il est interdit de poser un disque sur un disque plus petit.

Au début tous les disques sont sur la tige de gauche, et à la fin sur celle de droite.

Résolution

Voir la figure 3.1.

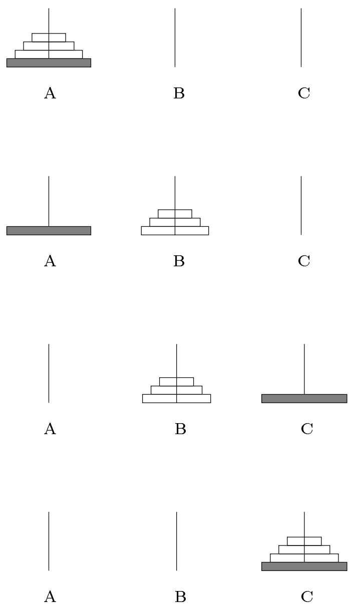
Algorithme

HANOÏ(n, `départ`, `intermédiaire`, `destination`)
Si $n = 1$ **alors** déplacer le disque supérieur de la tige `départ` vers la tige `destination`
    sinon HANOÏ($n - 1$, `départ`, `destination`, `intermédiaire`)
        déplacer le disque supérieur de la tige `départ` vers la tige `destination`
        HANOÏ($n - 1$, `intermédiaire`, `départ`, `destination`)

Exécution avec trois disques

1. Déplace un disque de la tige `départ` vers la tige `destination`
2. Déplace un disque de la tige `départ` vers la tige `intermédiaire`
3. Déplace un disque de la tige `destination` vers la tige `intermédiaire`
4. Déplace un disque de la tige `départ` vers la tige `destination`
5. Déplace un disque de la tige `intermédiaire` vers la tige `départ`
6. Déplace un disque de la tige `intermédiaire` vers la tige `destination`
7. Déplace un disque de la tige `départ` vers la tige `destination`

Il ne faut pas chercher à comprendre *comment* ça marche, mais *pourquoi* ça marche...



Hypothèse : on suppose que l'on sait résoudre le problème pour $(n-1)$ disques.

Principe : pour déplacer n disques de la tige `A` vers la tige `C`, on déplace les $(n-1)$ plus petits disques de la tige `A` vers la tige `B`, puis on déplace le plus gros disque de la tige `A` vers la tige `C`, puis on déplace les $(n-1)$ plus petits disques de la tige `B` vers la tige `C`.

Validité : il n'y a pas de viol des règles possible puisque le plus gros disque est toujours en « bas » d'une tige et que l'hypothèse (de récurrence) nous assure que nous savons déplacer le « bloc » de $(n-1)$ disques en respectant les règles.

FIG. 3.1 – Méthode de résolution du jeu des tours de Hanoï.

Complexité

On compte le nombre de déplacements de disques effectués par l'algorithme HANOÏ invoqué sur n disques.

$$C(n) = \begin{cases} 1 & \text{si } n = 1 \\ C(n-1) + 1 + C(n-1) & \text{sinon} \end{cases} = \begin{cases} 1 & \text{si } n = 1 \\ 1 + 2 \times C(n-1) & \text{sinon} \end{cases}$$

d'où l'on en déduit que $C(n) = 2^n - 1$. On a donc ici un algorithme de complexité exponentielle.

3.2 Dérécursivation

Dérécursiver, c'est transformer un algorithme récursif en un algorithme équivalent ne contenant pas d'appels récursifs.

3.2.1 Récursivité terminale

Définition 5 (Récursivité terminale). *Un algorithme est dit récursif terminal s'il ne contient aucun traitement après un appel récursif.*

Exemple :

ALGORITHME P(U)
  si C **alors** $D; \mathrm{P}(\alpha(U))$
      sinon T

où :
- U est la liste des paramètres ;
- C est une condition portant sur U ;
- D est le traitement de base de l'algorithme (dépendant de U) ;
- $\alpha(U)$ représente la transformation des paramètres ;
- T est le traitement de terminaison (dépendant de U).

Avec ces notations, l'algorithme P équivaut à l'algorithme suivant :

ALGORITHME P'(U)
  tant que C **faire** $D; U \leftarrow \alpha(U)$
  T

L'algorithme P' est une version **dérécursivée** de l'algorithme P.

3.2.2 Récursivité non terminale

Ici, pour pouvoir dérécursiver, il va falloir sauvegarder le contexte de l'appel récursif, typiquement les paramètres de l'appel engendrant l'appel récursif. Originellement, l'algorithme est :

ALGORITHME Q(U)
si $C(U)$ **alors** $D(U); \mathrm{Q}(\alpha(U)); F(U)$
  sinon $T(U)$

Les piles sont des structures de stockage (via les primitives `empiler` et `dépiler`) qui fonctionnent sur le principe « le dernier entré est le premier sorti » (cf. chapitre 5). Les compilateurs utilisent des piles pour stocker les paramètres des appels de fonctions, et en particulier lors de la transcription des fonctions récursives. Nous mimons ici l'utilisation des piles pour dérécursiver l'algorithme.

Après dérécursivation on obtiendra donc :

ALGORITHME Q'(U)
  empiler(nouvel_appel, U)
  tant que pile non vide **faire**
    dépiler(état, V)
    si état = nouvel_appel **alors** $U \leftarrow V$
      si $C(U)$ **alors** $D(U)$
        empiler(fin, U)
        empiler(nouvel_appel, $\alpha(U)$)
      sinon $T(U)$
    si état = fin **alors** $U \leftarrow V$
      $F(U)$

Illustration de la dérécursivation de l'algorithme Q

Exemple d'exécution de Q :

Appel $\mathrm{Q}(U_0)$
  $C(U_0)$ *vrai*
  $D(U_0)$
  Appel $\mathrm{Q}(\alpha(U_0))$
    $C(\alpha(U_0))$ *vrai*
    $D(\alpha(U_0))$
    Appel $\mathrm{Q}(\alpha(\alpha(U_0)))$
      $C(\alpha(\alpha(U_0)))$ *faux*
      $T(\alpha(\alpha(U_0)))$
    $F(\alpha(U_0))$
  $F(U_0)$

L'exécution correspondante de Q' est présentée figure 3.2. Les instructions de gestion de piles y figurent en italic, et les instructions de l'algorithme originel (ce qui nous importe) y figurent en gras.

Appel Q'(U_0)
empiler(nouvel_appel, U))
pile = [(nouvel_appel, U_0)]
dépiler(état, V))
état ← nouvel_appel ; $V \leftarrow U_0$; pile = []
$U \leftarrow U_0$
C($\mathbf{U_0}$) ***vrai***
D($\mathbf{U_0}$)
empiler(fin, U))
pile = [(fin, U_0)]
empiler(nouvel_appel, $\alpha(U)$))
pile = [(fin, U_0) ; (nouvel_appel, $\alpha(U_0)$)]
dépiler(état, V))
état ← nouvel_appel ; $V \leftarrow \alpha(U_0)$; pile = [(fin, U_0)]
$U \leftarrow \alpha(U_0)$
C($\alpha(\mathbf{U_0})$) ***vrai***
D($\alpha(\mathbf{U_0})$)
empiler(fin, U))
pile = [(fin, U_0) ; (fin, $\alpha(U_0)$)]
empiler(nouvel_appel, $\alpha(U)$))
pile = [(fin, U_0) ; (fin, $\alpha(U_0)$) ; (nouvel_appel, $\alpha(\alpha(U_0))$)]
dépiler(état, V))
état ← nouvel_appel ; $V \leftarrow \alpha(\alpha(U_0))$; pile = [(fin, U_0) ; (fin, $\alpha(U_0)$)]
$U \leftarrow \alpha(\alpha(U_0))$
C($\alpha(\alpha(\mathbf{U_0}))$) ***faux***
T($\alpha(\alpha(\mathbf{U_0}))$)
dépiler(état, V))
état ← fin ; $V \leftarrow \alpha(U_0)$; pile = [(fin, U_0)]
F($\alpha(\mathbf{U_0})$)
dépiler(état, V))
état ← fin ; $V \leftarrow U_0$; pile = []
F($\mathbf{U_0}$)

FIG. 3.2 – Exemple d'exécution de l'algorithme dérécursivé.

3.2.3 Remarques

Les programmes itératifs sont souvent plus efficaces, mais les programmes récursifs sont plus faciles à écrire. Les compilateurs savent, la plupart du temps, reconnaître les appels récursifs terminaux, et ceux-ci n'engendrent pas de surcoût par rapport à la version itérative du même programme.

Il est toujours possible de dérécursiver un algorithme récursif.

3.3 Diviser pour régner

3.3.1 Principe

Nombres d'algorithmes ont une structure récursive : pour résoudre un problème donné, ils s'appellent eux-mêmes récursivement une ou plusieurs fois sur des problèmes très similaires, mais de tailles moindres, résolvent les sous-problèmes de manière récursive puis combinent les résultats pour trouver une solution au problème initial.

Le paradigme « diviser pour régner » donne lieu à trois étapes à chaque niveau de récursivité :

Diviser : le problème en un certain nombre de sous-problèmes ;

Régner : sur les sous-problèmes en les résolvant récursivement ou, si la taille d'un sous-problème est assez réduite, le résoudre directement ;

Combiner : les solutions des sous-problèmes en une solution complète du problème initial.

3.3.2 Premier exemple : multiplication naïve de matrices

Nous nous intéressons ici à la multiplication de matrices carrés de taille n.

Algorithme naïf

L'algorithme classique est le suivant :

Multiplier-Matrices(A, B)
  Soit n la taille des matrices carrés A et B
  Soit C une matrice carré de taille n
  Pour $i \leftarrow 1$ **à** n **faire**
    Pour $j \leftarrow 1$ **à** n **faire**
      $c_{i,j} \leftarrow 0$
      Pour $k \leftarrow 1$ **à** n **faire**
        $c_{i,j} \leftarrow c_{i,j} + a_{i,k}.b_{k,j}$
  renvoyer C

Cet algorithme effectue $\Theta(n^3)$ multiplications et autant d'additions.

Algorithme « diviser pour régner » naïf

Dans la suite nous supposerons que n est une puissance exacte de 2. Décomposons les matrices A, B et C en sous-matrices de taille $n/2 \times n/2$. L'équation $C = AB$ peut alors se récrire :

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

En développant cette équation, nous obtenons :

$$r = ae + bf, \quad s = ag + bh, \quad t = ce + df \quad \text{et} \quad u = cg + dh.$$

Chacune de ces quatre opérations correspond à deux multiplications de matrices carrés de taille $n/2$ et une addition de telles matrices. À partir de ces équations on peut aisément dériver un algorithme « diviser pour régner » dont la complexité est donnée par la récurrence :

$$T(n) = 8T(n/2) + \Theta(n^2),$$

l'addition des matrices carrés de taille $n/2$ étant en $\Theta(n^2)$.

3.3.3 Analyse des algorithmes « diviser pour régner »

Lorsqu'un algorithme contient un appel récursif à lui-même, son temps d'exécution peut souvent être décrit par une équation de récurrence qui décrit le temps d'exécution global pour un problème de taille n en fonction du temps d'exécution pour des entrées de taille moindre.

La récurrence définissant le temps d'exécution d'un algorithme « diviser pour régner » se décompose suivant les trois étapes du paradigme de base :

1. Si la taille du problème est suffisamment réduite, $n \leq c$ pour une certaine constante c, la résolution est directe et consomme un temps constant $\Theta(1)$.
2. Sinon, on divise le problème en a sous-problèmes chacun de taille $1/b$ de la taille du problème initial. Le temps d'exécution total se décompose alors en trois parties :

(a) $D(n)$: le temps nécessaire à la division du problème en sous-problèmes.

(b) $aT(n/b)$: le temps de résolution des a sous-problèmes.

(c) $C(n)$: le temps nécessaire pour construire la solution finale à partir des solutions aux sous-problèmes.

La relation de récurrence prend alors la forme :

$$T(n) = \begin{cases} \Theta(1) & \text{si } n \leq c, \\ aT(n/b) + D(n) + C(n) & \text{sinon,} \end{cases}$$

où l'on interprète n/b soit comme $\lfloor n/b \rfloor$, soit comme $\lceil n/b \rceil$.

3.3.4 Résolution des récurrences

Théorème 1 (Résolution des récurrences « diviser pour régner »).
Soient $a \geq 1$ et $b > 1$ deux constantes, soit $f(n)$ une fonction et soit $T(n)$ une fonction définie pour les entiers positifs par la récurrence :

$$T(n) = aT(n/b) + f(n),$$

où l'on interprète n/b soit comme $\lfloor n/b \rfloor$, soit comme $\lceil n/b \rceil$.

$T(n)$ peut alors être bornée asymptotiquement comme suit :

1. *Si $f(n) = O(n^{(\log_b a) - \epsilon})$ pour une certaine constante $\epsilon > 0$, alors $T(n) = \Theta(n^{\log_b a})$.*
2. *Si $f(n) = \Theta(n^{\log_b a})$, alors $T(n) = \Theta(n^{\log_b a} \log n)$.*
3. *Si $f(n) = \Omega(n^{(\log_b a) + \epsilon})$ pour une certaine constante $\epsilon > 0$, et si $af(n/b) \leq cf(n)$ pour une constante $c < 1$ et n suffisamment grand, alors $T(n) = \Theta(f(n))$.*

Remarques :

1. Le remplacement des termes $T(n/b)$ par $T(\lfloor n/b \rfloor)$ ou $T(\lceil n/b \rceil)$ n'affecte pas le comportement asymptotique de la récurrence [2, section 4.4.2]. On omettra donc en général les parties entières.
2. Le théorème 1 ne couvre pas toutes les possibilité pour $f(n)$. Par exemple, il y a un « trou » entre les cas 1 et 2 quand $f(n)$ est plus petite que $n^{\log_b a}$, mais pas polynomialement. Dans un tel cas, on ne peut tout simplement pas appliquer le théorème 1.

Retour sur le premier exemple

Utilisons le théorème 1 pour calculer la complexité de notre algorithme de multiplication de matrices « diviser pour régner » naïf. Ici, $a = 8$, $b = 2$ et $f(n) = \Theta(n^2)$. Donc $\log_b a = 3$, nous nous trouvons dans le cas 1 du théorème (avec $\epsilon = 1$), l'algorithme a une complexité en $\Theta(n^3)$ et nous n'avons rien gagné...

3.3.5 Deuxième exemple : algorithme de Strassen pour la multiplication de matrices

L'algorithme de Strassen est un algorithme « diviser pour régner » qui n'effectue que 7 multiplications de matrices, contrairement à 8 dans l'algorithme précédent, mais qui effectue plus d'additions et de soustractions de matrices, ce qui est sans conséquence une addition de matrices étant « gratuite » par rapport au coût d'une multiplication.

Complexité

La complexité de l'algorithme de Strassen est donnée par la récurrence :

$$T(n) = 7T(n/2) + \Theta(n^2).$$

En utilisant le théorème 1, nous obtenons comme complexité : $T(n) = \Theta(n^{\log_2 7}) = O(n^{2,81})$.

Algorithme

Il se décompose en quatre étapes :

1. Diviser les matrices A et B en matrices carrés de taille $n/2$.
2. Au moyen de $\Theta(n^2)$ additions et soustractions scalaires, calculer 14 matrices (à préciser) A_1, ..., A_7, B_1, ..., B_7 carrés de taille $n/2$.
3. Calculer récursivement les 7 produits de matrices $P_i = A_i B_i$, $i \in [1; 7]$.
4. Calculer les sous-matrices désirées r, s, t et u en additionnant et/ou soustrayant les combinaisons idoines des matrices P_i *ad-hoc*, à l'aide de $\Theta(n^2)$ additions et soustractions scalaires.

Produits de sous-matrices

Nous supposons que chaque matrice produit P_i peut s'écrire sous la forme :

$$P_i = A_i B_i = (\alpha_{i,1}a + \alpha_{i,2}b + \alpha_{i,3}c + \alpha_{i,4}d).(\beta_{i,1}e + \beta_{i,2}f + \beta_{i,3}g + \beta_{i,4}h),$$

où les coefficients $\alpha_{i,j}$ et $\beta_{i,j}$ sont tous pris dans l'ensemble $\{-1; 0; 1\}$. Nous supposons donc que chaque produit peut être obtenu en additionnant et soustrayant certaines des sous-matrices de A, en additionnant et soustrayant certaines des sous-matrices de B, et en multipliant les deux matrices ainsi obtenues.

Récrivons l'équation définissant r :

$$r = ae + bf = (a\ b\ c\ d)\begin{pmatrix} +1 & 0 & 0 & 0 \\ 0 & +1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}\begin{pmatrix} e \\ f \\ g \\ h \end{pmatrix} = \begin{array}{cc} & \begin{array}{cccc} e & f & g & h \end{array} \\ \begin{array}{c} a \\ b \\ c \\ d \end{array} & \begin{pmatrix} + & . & . & . \\ . & + & . & . \\ . & . & . & . \\ . & . & . & . \end{pmatrix} \end{array}$$

où « + » représente « +1 », « . » représente « 0 » et « - » représente « -1 ». Nous récrivons de même les équations définissant s, t et u :

$$s = ag + bh = \begin{array}{cc} & \begin{array}{cccc} e & f & g & h \end{array} \\ \begin{array}{c} a \\ b \\ c \\ d \end{array} & \begin{pmatrix} . & . & + & . \\ . & . & . & + \\ . & . & . & . \\ . & . & . & . \end{pmatrix} \end{array},$$

$$t = ce + df = \begin{array}{cc} & \begin{array}{cccc} e & f & g & h \end{array} \\ \begin{array}{c} a \\ b \\ c \\ d \end{array} & \begin{pmatrix} . & . & . & . \\ . & . & . & . \\ + & . & . & . \\ . & + & . & . \end{pmatrix} \end{array},$$

$$u = cg + dh = \begin{array}{cc} & \begin{array}{cccc} e & f & g & h \end{array} \\ \begin{array}{c} a \\ b \\ c \\ d \end{array} & \begin{pmatrix} . & . & . & . \\ . & . & . & . \\ . & . & + & . \\ . & . & . & + \end{pmatrix} \end{array}.$$

On remarque que l'on peut calculer s par $s = P_1 + P_2$ où P_1 et P_2 sont calculées chacune au moyen d'une unique multiplication de matrice :

$$P_1 = A_1 B_1 = a.(g - h) = ag - ah = \begin{pmatrix} . & . & + & - \\ . & . & . & . \\ . & . & . & . \\ . & . & . & . \end{pmatrix}, P_2 = A_2 B_2 = (a + b).h = ah + bh = \begin{pmatrix} . & . & . & + \\ . & . & . & + \\ . & . & . & . \\ . & . & . & . \end{pmatrix}.$$

De même la matrice t peut être calculée par $t = P_3 + P_4$ avec :

$$P_3 = A_3B_3 = (c+d).e = ce+de = \begin{pmatrix} . & . & . & . \\ . & . & . & . \\ + & . & . & . \\ + & . & . & . \end{pmatrix}, P_4 = A_4B_4 = d.(f-e) = df-de = \begin{pmatrix} . & . & . & . \\ . & . & . & . \\ . & . & . & . \\ - & + & . & . \end{pmatrix}.$$

Pour calculer r et u on introduit une matrice P_5 définie comme suit :

$$P_5 = A_5B_5 = (a+d).(e+h) = \begin{pmatrix} + & . & . & + \\ . & . & . & . \\ . & . & . & . \\ + & . & . & + \end{pmatrix},$$

et on cherche à obtenir r à partir de P_5 :

$$\begin{aligned} r &= \begin{pmatrix} + & . & . & . \\ . & + & . & . \\ . & . & . & . \\ . & . & . & . \end{pmatrix} \\ &= \begin{pmatrix} + & . & . & + \\ . & . & . & . \\ . & . & . & . \\ + & . & . & + \end{pmatrix} + \begin{pmatrix} . & . & . & - \\ . & + & . & . \\ . & . & . & . \\ - & . & . & - \end{pmatrix} \\ &= \begin{pmatrix} + & . & . & + \\ . & . & . & . \\ . & . & . & . \\ + & . & . & + \end{pmatrix} + \begin{pmatrix} . & . & . & . \\ . & . & . & . \\ . & . & . & . \\ - & + & . & . \end{pmatrix} + \begin{pmatrix} . & . & . & - \\ . & + & . & . \\ . & . & . & . \\ . & - & . & - \end{pmatrix} \\ &= \begin{pmatrix} + & . & . & + \\ . & . & . & . \\ . & . & . & . \\ + & . & . & + \end{pmatrix} + \begin{pmatrix} . & . & . & . \\ . & . & . & . \\ . & . & . & . \\ - & + & . & . \end{pmatrix} + \begin{pmatrix} . & . & . & - \\ . & . & . & - \\ . & . & . & . \\ . & . & . & . \end{pmatrix} + \begin{pmatrix} . & . & . & . \\ . & + & . & + \\ . & . & . & . \\ . & - & . & - \end{pmatrix}. \end{aligned}$$

D'où, en posant

$$P_6 = A_6B_6 = (b-d).(f+h) \begin{pmatrix} . & . & . & . \\ . & + & . & + \\ . & . & . & . \\ . & - & . & - \end{pmatrix},$$

on obtient $r = P_5 + P_4 - P_2 + P_6$.

De même, on cherche à obtenir u à partir de P_5 :

$$
\begin{aligned}
u &= \begin{pmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & + & \cdot \\ \cdot & \cdot & \cdot & + \end{pmatrix} \\
&= \begin{pmatrix} + & \cdot & \cdot & + \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ + & \cdot & \cdot & + \end{pmatrix} + \begin{pmatrix} - & \cdot & \cdot & - \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & + & \cdot \\ - & \cdot & \cdot & \cdot \end{pmatrix} \\
&= \begin{pmatrix} + & \cdot & \cdot & + \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ + & \cdot & \cdot & + \end{pmatrix} + \begin{pmatrix} \cdot & \cdot & + & - \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix} + \begin{pmatrix} - & \cdot & - & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & + & \cdot \\ - & \cdot & \cdot & \cdot \end{pmatrix} \\
&= \begin{pmatrix} + & \cdot & \cdot & + \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ + & \cdot & \cdot & + \end{pmatrix} + \begin{pmatrix} \cdot & \cdot & + & - \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix} + \begin{pmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ - & \cdot & \cdot & \cdot \\ - & \cdot & \cdot & \cdot \end{pmatrix} + \begin{pmatrix} - & \cdot & - & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ + & \cdot & + & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix}.
\end{aligned}
$$

D'où, en posant

$$
P_7 = A_7 B_7 = (a-c).(e+g) = ae + ag - ce - cg = \begin{pmatrix} + & \cdot & + & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ - & \cdot & - & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix},
$$

on obtient $u = P_5 + P_1 - P_3 - P_7$.

Discussion

L'algorithme de Strassen n'est intéressant en pratique que pour de grandes matrices ($n > 45$) denses (peu d'éléments non nuls).

La meilleure borne supérieure connue pour la multiplication de matrices carrés de taille n est environ en $O(n^{2,376})$. La meilleure borne inférieure connue est en $\Omega(n^2)$ (il faut générer n^2 valeurs). On ne connaît donc toujours pas le niveau de difficulté réel d'une multiplication de matrices !

Chapitre 4

Algorithmes de tri

4.1 Tri par fusion

4.1.1 Principe

L'algorithme de tri par fusion est construit suivant le paradigme « diviser pour régner » :

1. Il divise la séquence de n nombres à trier en deux sous-séquences de taille $n/2$.
2. Il trie récursivement les deux sous-séquences.
3. Il fusionne les deux sous-séquences triées pour produire la séquence complète triée.

La récursion termine quand la sous-séquence à trier est de longueur 1... car une telle séquence est toujours triée.

4.1.2 Algorithme

La principale action de l'algorithme de tri par fusion est justement la fusion des deux listes triées.

La fusion

Le principe de cette fusion est simple : à chaque étape, on compare les éléments minimaux des deux sous-listes triées, le plus petit des deux étant l'élément minimal de l'ensemble on le met de côté et on recommence. On conçoit ainsi un algorithme FUSIONNER qui prend en entrée un tableau A et trois entiers, p, q et r, tels que $p \leq q < r$ et tels que les tableaux $A[p..q]$ et $A[q+1..r]$ soient triés. L'algorithme est présenté figure 4.1.

Complexité de la fusion

Étudions les différentes étapes de l'algorithme :
- les initialisations ont un coût constant $\Theta(1)$;
- la boucle *tant que* de fusion s'exécute au plus $r-p$ fois, chacune de ses itérations étant de coût constant, d'où un coût total en $O(r-p)$;
- les deux boucles *tant que* complétant C ont une complexité respective au pire de $q-p+1$ et de $r-q$, ces deux complexités étant en $O(r-p)$;
- la recopie finale coûte $\Theta(r-p+1)$.

Par conséquent, l'algorithme de fusion a une complexité en $\Theta(r-p)$.

Le tri

Écrire l'algorithme de tri par fusion est maintenant une trivialité (cf. figure 4.2).

```
Fusionner(A, p, q, r)
  i ← p                                      indice servant à parcourir le tableau A[p..q]
  j ← q + 1                                  indice servant à parcourir le tableau A[q + 1..r]
  Soit C un tableau de taille r − p + 1      tableau temporaire dans lequel on construit le résultat
  k ← 1                                      indice servant à parcourir le tableau temporaire
  tant que i ≤ q et j ≤ r faire              boucle de fusion
    si A[i] < A[j] alors C[k] ← A[i]
                         i ← i + 1
                   sinon C[k] ← A[j]
                         j ← j + 1
    k ← k + 1
  tant que i ≤ q faire C[k] ← A[i]           on incorpore dans C les éléments de A[p..q]
                       i ← i + 1               qui n'y seraient pas encore ; s'il y en a,
                       k ← k + 1               les éléments de A[q + 1..r] sont déjà tous dans C
  tant que j ≤ r faire C[k] ← A[j]           on incorpore dans C les éléments de A[q + 1..r]
                       j ← j + 1               qui n'y seraient pas encore ; s'il y en a,
                       k ← k + 1               les éléments de A[p..q] sont déjà tous dans C
  pour k ← 1 à r − p + 1 faire               on recopie le résultat dans le tableau originel
      A[p + k − 1] ← C[k]
```

Fig. 4.1 – Algorithme de fusion de deux sous-tableaux adjacents triés.

```
Tri-Fusion(A, p, r)
  si p < r alors q ← ⌊(p + r)/2⌋
                 Tri-Fusion(A, p, q)
                 Tri-Fusion(A, q + 1, r)
                 Fusionner(A, p, q, r)
```

Fig. 4.2 – Algorithme de tri par fusion.

4.1.3 Complexité

Pour déterminer la formule de récurrence qui nous donnera la complexité de l'algorithme Tri-Fusion, nous étudions les trois phases de cet algorithme « diviser pour régner » :

Diviser : cette étape se réduit au calcul du milieu de l'intervalle $[p; r]$, sa complexité est donc en $\Theta(1)$.

Régner : l'algorithme résout récursivement deux sous-problèmes de tailles respectives $\frac{n}{2}$, d'où une complexité en $2T(\frac{n}{2})$.

Combiner : la complexité de cette étape est celle de l'algorithme de fusion qui est de $\Theta(n)$ pour la construction d'un tableau solution de taille n.

Par conséquent, la complexité du tri par fusion est donnée par la récurrence :

$$T(n) = \begin{cases} \Theta(1) & \text{si } n = 1, \\ 2T(\frac{n}{2}) + \Theta(n) & \text{sinon.} \end{cases}$$

Pour déterminer la complexité du tri par fusion, nous utilisons de nouveau le théorème 1. Ici $a = 2$ et $b = 2$, donc $\log_b a = 1$ et nous nous trouvons dans le deuxième cas du théorème : $f(n) = \Theta(n^{\log_b a}) = \Theta(n)$. Par conséquent :

$$T(n) = \Theta(n \log n).$$

Pour des valeurs de n suffisamment grandes, le tri par fusion avec son temps d'exécution en $\Theta(n \log n)$ est nettement plus efficace que le tri par insertion dont le temps d'exécution est en $\Theta(n^2)$.

4.2 Tri par tas

4.2.1 Définition d'un tas

Définition 6 (Tas). *Un tas est un arbre binaire parfait dont tous les niveaux sont complets sauf le dernier qui est rempli de la gauche vers la droite. Dans un tas, un père est toujours plus grand que ses deux fils.*

Pour un exemple de tas, voir la figure 4.3.

Les tas sont généralement représentés et manipulés sous la forme d'un tableau :

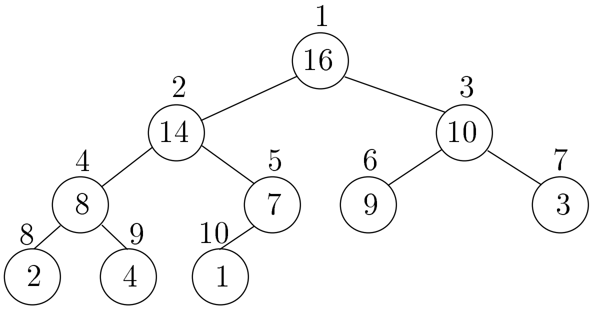
– Un tableau A qui représente un tas est un objet à deux attributs :
  1. `longueur`(A) qui est le nombre d'éléments qui peuvent être stockés dans le tableau A ;
  2. `taille`(A) qui est le nombre d'éléments stockés dans le tableau A.
– La racine est stockée dans la première case du tableau $A[1]$.
– Les éléments de l'arbre sont rangés dans l'ordre, niveau par niveau, et de gauche à droite. Les fonctions d'accès aux éléments du tableau sont alors :

  Père(i)
  renvoyer $\lfloor i/2 \rfloor$

  Fils-Gauche(i)
  renvoyer $2i$

  Fils-Droit(i)
  renvoyer $2i + 1$

– Propriété des tas : $A[\text{Père}(i)] \geq A[i]$.



16	14	10	8	7	9	3	2	4	1
1	2	3	4	5	6	7	8	9	10

Fig. 4.3 – Un tas vu comme un arbre binaire (à gauche) et comme un tableau (à droite). Le nombre à l'intérieur d'un nœud de l'arbre est la valeur contenue dans ce nœud ; le nombre au-dessus est l'indice correspondant dans le tableau.

4.2.2 Conservation de la structure de tas

L'algorithme Entasser (cf. figure 4.4) prend en entrée un tableau A et un indice i. On suppose que les sous-arbres de racines Gauche(i) et Droit(i) sont des tas. Par contre, il est possible que $A[i]$ soit plus petit que ses fils (violant ainsi la propriété de tas). Entasser doit faire « descendre » la valeur de $A[i]$ de sorte que le sous-arbre de racine i soit un tas. L'action de cet algorithme est illustré par la figure 4.5.

Correction

Le résultat de l'algorithme Entasser est bien un tas car :

– La structure de l'arbre n'est pas modifiée.

ENTASSER(A, i)
  $g \leftarrow$ GAUCHE(i)
  $d \leftarrow$ DROIT(i)
  $max \leftarrow i$
  si $g \leq$ `taille`(A) **et** $A[g] > A[max]$ **alors** $max \leftarrow g$
  si $d \leq$ `taille`(A) **et** $A[d] > A[max]$ **alors** $max \leftarrow d$
  si $max \neq i$ **alors** échanger $A[i] \leftrightarrow A[max]$
    ENTASSER(A, max)

FIG. 4.4 – Algorithme ENTASSER

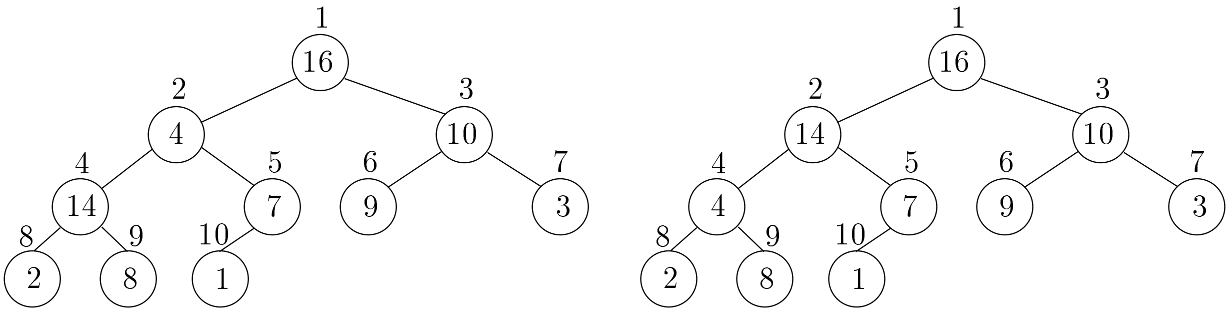


FIG. 4.5 – Action de ENTASSER(A, 2) : la configuration initiale (figure gauche) viole la propriété du tas ; pour $i = 2$ cette propriété est restaurée par interversion de la clé avec celle du fils gauche (figure de droite) ; le résultat n'est toujours pas un tas, et l'appel récursif ENTASSER(A, 4) intervertit la clé du nœud $i = 4$ avec celle de son fils droit ; on obtient finalement le tas de la figure 4.3.

– Un échange de valeurs entre un père et un fils n'a lieu que si la valeur du fils est supérieure à celle du père. Or la valeur du père était supérieure à celles stockées dans ses deux arbres fils exceptée la valeur ajoutée à l'arbre. La nouvelle clé de la racine est donc bien plus grande que l'intégralité de celles stockées dans l'arbre dont elle devient la racine.

Complexité

Le temps d'exécution de ENTASSER sur un arbre de taille n est en $\Theta(1)$ plus le temps de l'exécution récursive de ENTASSER sur un des deux sous-arbres, or ces deux sous-arbres ont une taille en au plus $\frac{2n}{3}$ (le pire cas survient quand la dernière rangée de l'arbre est exactement remplie à moitié). Le temps d'exécution de ENTASSER est donc décrit par la récurrence :

$$T(n) \leq T\left(\frac{2n}{3}\right) + \Theta(1)$$

ce qui, d'après le cas 2 du théorème 1, nous donne : $T(n) = \Theta(\log n)$, car $a = 1$, $b = \frac{3}{2}$ et $\log_b a = 0$.

4.2.3 Construction d'un tas

La construction se fait simplement par utilisation successive de l'algorithme ENTASSER, comme le montre l'algorithme à la figure 4.6.

Complexité

Première borne : chaque appel à entasser coûte $O(\log_2 n)$ et il y a $O(n)$ appels de ce type. La complexité de CONSTRUIRE-TAS est donc en $O(n \log_2 n)$. On peut en fait obtenir une borne plus fine.

```
CONSTRUIRE-TAS(A, Valeurs)
   taille[A] ← longueur[A]
   Pour i ← ⌊longueur(A)/2⌋ à 1 faire ENTASSER(A,i)
```

FIG. 4.6 – Algorithme CONSTRUIRE-TAS.

En effet, un tas à n éléments est de hauteur $\lfloor \log_2 n \rfloor$ et à une hauteur h il contient au maximum $\lceil \frac{n}{2^{h+1}} \rceil$ nœuds. De plus, l'algorithme ENTASSER requiert un temps d'exécution en $O(h)$ quand il est appelé sur un tas de hauteur h. D'où :

$$T(n) = \sum_{j=0}^{\lfloor \log_2 n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O\left(n \sum_{h=0}^{\lfloor \log_2 n \rfloor} \frac{h}{2^h}\right),$$

or

$$\sum_{h=0}^{\infty} \frac{h}{2^h} = 2.$$

D'où :

$$T(n) = O(n).$$

On peut donc construire un tas à partir d'un tableau en temps linéaire.

Illustration de l'algorithme CONSTRUIRE-TAS

Voir la figure 4.7.

4.2.4 Algorithme du tri par tas

```
TRIER-TAS(A)
   CONSTRUIRE-TAS(A)
   Pour i ← longueur(A) à 2 faire
      échanger A[1] ↔ A[i]
      taille(A) ← taille(A) − 1
      ENTASSER(A,1)
```

Illustration de l'algorithme TRI-TAS

Voir la figure 4.8.

Complexité

La procédure TRIER-TAS prend un temps $O(n \log_2 n)$ car l'appel à CONSTRUIRE-TAS prend un temps $O(n)$ et que chacun des $n-1$ appels à ENTASSER prend un temps $O(\log_2 n)$.

4.3 Tri rapide (*Quicksort*)

4.3.1 Principe

Le tri rapide est fondé sur le paradigme « diviser pour régner », tout comme le tri fusion, il se décompose donc en trois étapes :

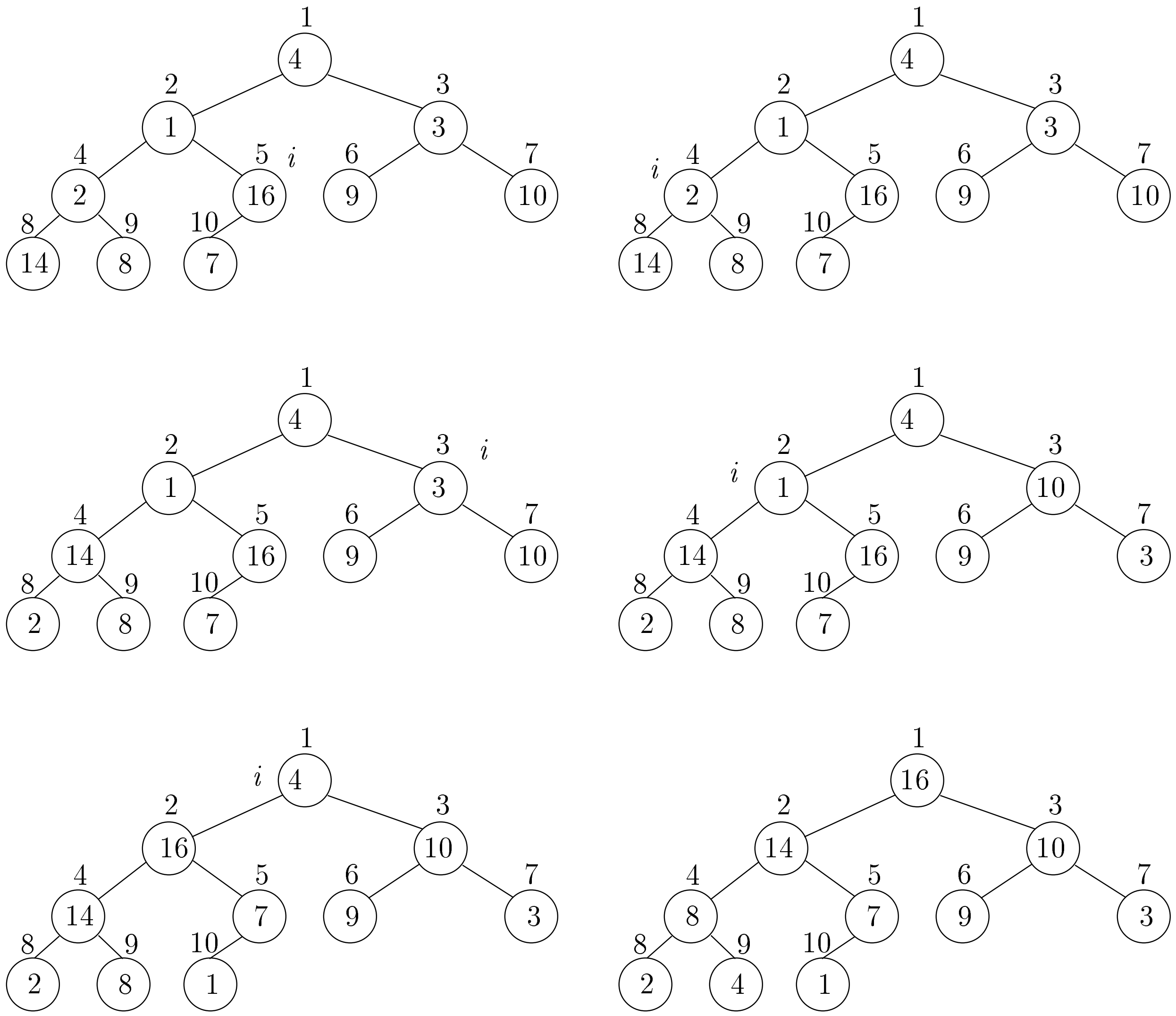


Fig. 4.7 – Action de Construire-Tas sur le tableau $[4; 1; 3; 2; 16; 9; 10; 14; 8; 7]$.

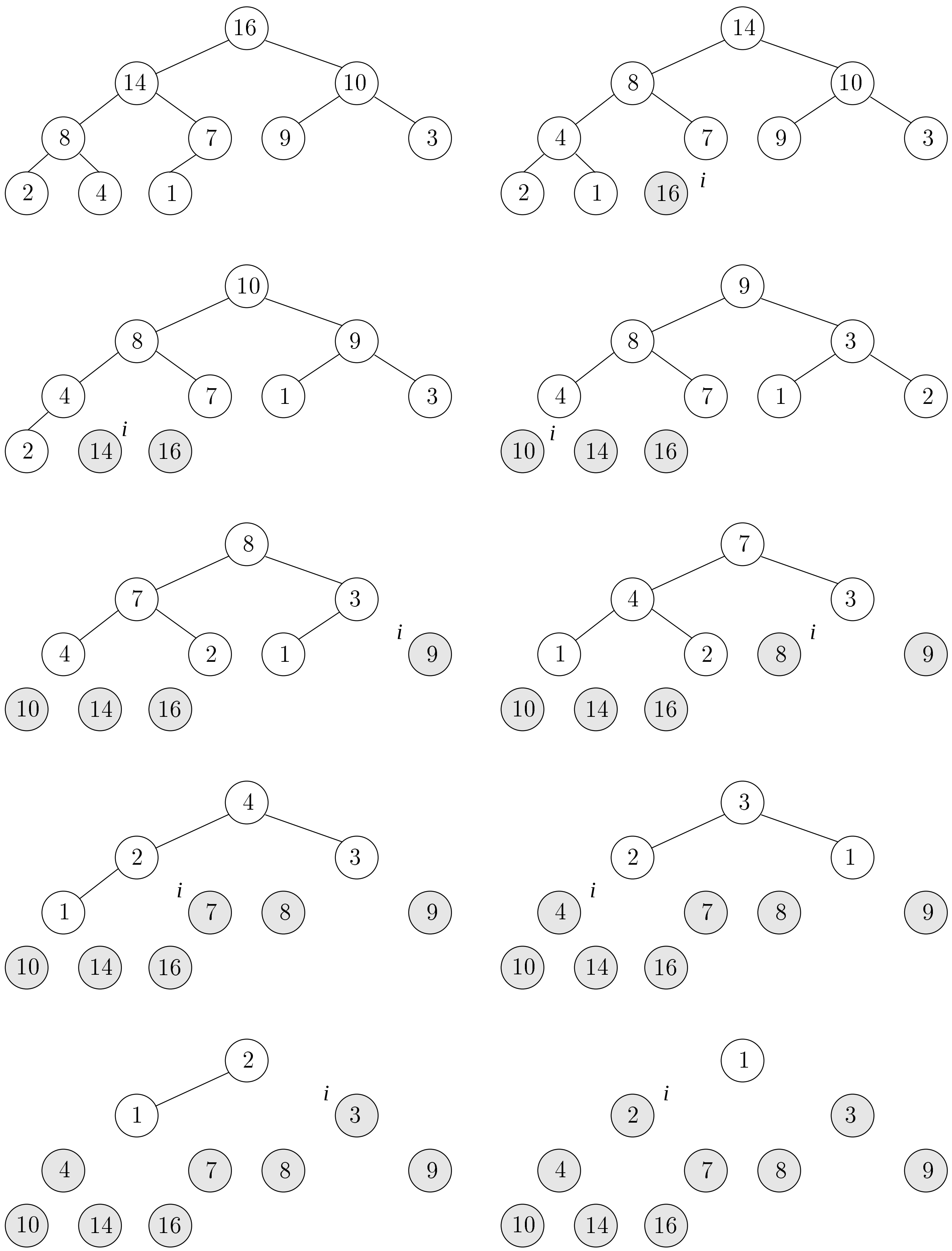


FIG. 4.8 – Action de TRIER-TAS sur le tableau [4; 1; 3; 2; 16; 9; 10; 14; 8; 7].

Diviser : Le tableau $A[p..r]$ est partitionné (et réarrangé) en deux sous-tableaux non vides, $A[p..q]$ et $A[q+1..r]$, tels que chaque élément de $A[p..q]$ soit inférieur ou égal à chaque élément de $A[q+1..r]$. L'indice q est calculé pendant la procédure de partitionnement.

Régner : Les deux sous-tableaux $A[p..q]$ et $A[q+1..r]$ sont triés par des appels récursifs.

Combiner : Comme les sous-tableaux sont triés sur place, aucun travail n'est nécessaire pour les recombiner, le tableau $A[p..r]$ est déjà trié !

4.3.2 Algorithme

```
TRI-RAPIDE(A, p, r)
  si p < r alors q ←PARTITIONNEMENT(A, p, r)
                 TRI-RAPIDE(A, p, q)
                 TRI-RAPIDE(A, q + 1, r)
```

L'appel TRI-RAPIDE(A, 1, *longueur*(A)) trie le tableau A. Le point principal de l'algorithme est bien évidemment le partitionnement qui réarrange le tableau A sur place :

```
PARTITIONNEMENT(A, p, r)
  x ← A[p]
  i ← p − 1
  j ← r + 1
  tant que VRAI faire
    répéter j ← j − 1 jusqu'à A[j] ≤ x
    répéter i ← i + 1 jusqu'à A[i] ≥ x
    si i < j alors échanger A[i] ↔ A[j]
           sinon renvoyer j
```

Exemple de partitionnement :

1. Situation intiale :

1	2	3	4	5	6	7
4	3	6	2	1	5	7

Nous avons donc $x = 4$, $i = 0$ et $j = 8$.

2. On exécute la boucle « **répéter** $j \leftarrow j-1$ **jusqu'à** $A[j] \leq x$ » et on obtient $j = 5$.
3. On exécute la boucle « **répéter** $i \leftarrow i+1$ **jusqu'à** $A[i] \geq x$ », et on obtient $i = 1$.
4. Après l'échange on obtient le tableau :

1	2	3	4	5	6	7
1	3	6	2	4	5	7

5. On exécute la boucle « **répéter** $j \leftarrow j-1$ **jusqu'à** $A[j] \leq x$ » et on obtient $j = 4$.
6. On exécute la boucle « **répéter** $i \leftarrow i+1$ **jusqu'à** $A[i] \geq x$ », et on obtient $i = 3$.
7. Après l'échange on obtient le tableau :

1	2	4	3	5	6	7
1	3	2	6	4	5	7

8. On exécute la boucle « **répéter** $j \leftarrow j-1$ **jusqu'à** $A[j] \leq x$ » et on obtient $j = 3$.
9. On exécute la boucle « **répéter** $i \leftarrow i+1$ **jusqu'à** $A[i] \geq x$ », et on obtient $i = 3$.
10. Comme $i = j$, l'algorithme se termine et renvoie la valeur « 3 ».

4.3.3 Complexité

Pire cas

Le pire cas intervient quand le partitionnement produit une région à $n-1$ éléments et une à un élément, comme nous le montrerons ci-après. Comme le partitionnement coûte $\Theta(n)$ et que $T(1) = \Theta(1)$, la récurrence pour le temps d'exécution est :

$$T(n) = T(n-1) + \Theta(n).$$

D'où par sommation :

$$T(n) = \sum_{k=1}^{n} \Theta(k) = \Theta\left(\sum_{k=1}^{n} k\right) = \Theta\left(n^2\right).$$

Pour montrer que cette configuration est bien le pire cas, montrons que dans tous les cas $T(n) = O(n^2)$, c'est-à-dire qu'il existe une constante c telle que $T(n) \leq c \times n^2$. Si $T(n)$ est la complexité au pire :

$$T(n) = \max_{1 \leq q \leq n-1} (T(q) + T(n-q)) + \Theta(n),$$

où le paramètre q est dans l'intervalle $[1..n-1]$ puisque la procédure Partitionnement génère deux régions de tailles chacune au moins égale à un. D'où :

$$T(n) \leq \max_{1 \leq q \leq n-1} \left(cq^2 + c(n-q)^2\right) + \Theta(n).$$

Or l'expression $q^2 + (n-q)^2$ atteint son maximum à l'une des extrémités de l'intervalle (dérivée négative puis positive). D'où

$$\max_{1 \leq q \leq n-1} \left(q^2 + (n-q)^2\right) = 1^2 + (n-1)^2 = n^2 - 2(n-1).$$

et

$$T(n) \leq cn^2 - 2c(n-1) + \Theta(n) \leq cn^2,$$

puisque l'on peut choisir la constante c assez grande pour que le terme $2c(n-1)$ domine le terme $\Theta(n)$. Du coup, le temps d'exécution du tri rapide (dans le pire cas) est $\Theta(n^2)$.

Meilleur cas

On subodore que le meilleur cas apparaît quand la procédure de partitionnement produit deux régions de taille $\frac{n}{2}$. La récurrence est alors :

$$T(n) = 2T(\frac{n}{2}) + \Theta(n),$$

ce qui, d'après le cas 2 du théorème 1 nous donne

$$T(n) = \Theta(n \log n).$$

Complexité en moyenne

On suppose que le tableau A ne contient pas deux fois le même élément.

Version stochastique du tri rapide. Un algorithme est dit **stochastique** si son comportement est déterminé non seulement par son entrée mais aussi par les valeurs produites par un **générateur de nombres aléatoires**. On modifie la procédure Partitionnement pour qu'elle est un comportement stochastique en utilisant une fonction Hasard(a, b) qui renvoie de manière équiprobable un entier entre les nombres a et b.

Partitionnement-Stochastique(A, p, r)
  $i \leftarrow$ Hasard(p, r)
  échanger $A[p] \leftrightarrow A[i]$
  renvoyer Partitionnement(A, p, r)

Le but ici est de faciliter l'analyse de l'algorithme et de minimiser l'influence des configurations pathologiques.

Analyse du partitionnement. La valeur q renvoyée par PARTITIONNEMENT ne dépend que du rang de $x = A[p]$ parmi les éléments de $A[p..r]$ (le **rang** d'un nombre dans un ensemble étant le nombre d'éléments qui lui sont inférieurs ou égaux). Du fait de l'encapsulation de PARTITIONNEMENT dans PARTITIONNEMENT-STOCHASTIQUE et de l'interversion de $A[p]$ et d'un élément aléatoire de $A[p..r]$, $rang(x) = i$ pour $i = 1$, 2, ..., n avec une probabilité $\frac{1}{n}$ en posant $n = r - p + 1$ (c'est le nombre d'éléments de l'intervalle $[p..r]$).

Ce qui nous intéresse, c'est la taille des partitions. Nous avons deux cas à considérer :

1. $rang(x) = 1$. L'algorithme PARTITIONNNEMENT s'arrête alors avec $i = j = 1$ et la région « inférieure » de la partition comprend l'unique élément $A[p]$ et est de taille 1.
2. $rang(x) \geq 2$. Il existe alors au moins un élément (strictement) plus petit que $x = A[p]$. Par conséquent, lors du passage dans la boucle *tant que*, l'indice i s'arrête à la valeur $i = p$ mais l'indice j s'arrête à une valeur strictement inférieure à p. Un échange est alors effectué et $A[p]$ est placé dans la région supérieure. Lorsque PARTITIONNEMENT se termine, chacun des $rang(x) - 1$ éléments de la région « inférieure » de la partition est strictement inférieur à x. Ainsi pour chaque $i = 1$, 2, ..., $n - 1$, la probabilité pour que la région inférieure ait i élément est de $\frac{1}{n}$

Récurrence pour le cas moyen. Vu ce qui précède, le temps moyen requis pour le tri d'un tableau de longueur n vaut donc :

$$T(n) = \frac{1}{n}\left(T(1) + T(n-1) + \sum_{q=1}^{n-1}(T(q) + T(n-q))\right) + \Theta(n).$$

Comme $T(1) = \Theta(1)$ et $T(n-1) = O(n^2)$ (vue l'étude du pire cas), on a :

$$\frac{1}{n}(T(1) + T(n-1)) = \frac{1}{n}\left(\Theta(1) + O(n^2)\right) = O(n),$$

et ce terme peut être absorbé par le terme $\Theta(n)$ de la formule. Ainsi :

$$T(n) = \frac{1}{n}\sum_{q=1}^{n-1}(T(q) + T(n-q)) + \Theta(n) = \frac{2}{n}\sum_{q=1}^{n-1}T(q) + \Theta(n).$$

Résolution de la récurrence. On suppose par induction qu'il existe des constantes strictement positives a et b telles que $T(n) \leq an\log n + b$. Si a et b sont tels que l'hypothèse est vraie pour $n = 1$ alors, si l'on suppose l'hypothèse vraie jusqu'au rang $n - 1$, on a :

$$T(n) = \frac{2}{n}\sum_{q=1}^{n-1}T(q) + \Theta(n) \leq \frac{2}{n}\sum_{k=1}^{n-1}(ak\log k + b) + \Theta(n) = \frac{2a}{n}\sum_{k=1}^{n-1}k\log k + \frac{2b}{n}(n-1) + \Theta(n).$$

Si l'on sait que (cf. [2, P. 164]) :

$$\sum_{k=1}^{n-1}k\log k \leq \frac{1}{2}n^2\log n - \frac{1}{8}n^2,$$

on obtient :

$$T(n) \leq \frac{2a}{n}\left(\frac{1}{2}n^2\log n - \frac{1}{8}n^2\right) + \frac{2b}{n}(n-1) + \Theta(n) \leq an\log n - \frac{a}{4}n + 2b + \Theta(n) = an\log n + b + \left(\Theta(n) + b - \frac{a}{4}n\right)$$

d'où

$$T(n) \leq an\log n + b,$$

puisque l'on peut choisir a suffisamment grand pour que $\frac{a}{4}n$ domine $\Theta(n) + b$. On en conclut que le temps d'exécution moyen du tri rapide est $O(n\log n)$.

Chapitre 5

Structures de données élémentaires

5.1 Introduction

En informatique, il existe plusieurs manières de représenter la notion mathématique d'ensemble. Il n'existe pas une représentation qui soit « meilleure » que les autres dans l'absolu : pour un problème donné, la meilleure représentation sera celle qui permettra de concevoir le « meilleur » algorithme, c'est-à-dire celui le plus esthétique et de moindre complexité. On parlera parfois d'*ensembles dynamiques* car nos ensembles seront rarement figés.

Chaque élément de ces ensembles pourra comporter plusieurs *champs* qui peuvent être examinés dès lors que l'on possède un *pointeur* —ou une *référence* si on préfère utiliser une terminologie plus proche de *Java* que de *C*— sur cet élément. Certains ensembles dynamiques supposent que l'un des champs de l'objet contient une **clé** servant d'identifiant.

Ces ensembles supportent potentiellement tout une série d'opérations :

- RECHERCHE(S, k) : étant donné un ensemble S et une clé k, le résultat de cette requête est un pointeur sur un élément de S de clé k, s'il en existe un, et la valeur NIL sinon —NIL étant un pointeur ou une référence sur « rien ».
- INSERTION(S, x) : ajoute à l'ensemble S l'élément pointé par x.
- SUPPRESSION(S, x) : supprime de l'ensemble S son élément pointé par x (si l'on souhaite supprimer un élément dont on ne connaît que la clé k, il suffit de récupérer un pointeur sur cet élément via un appel à RECHERCHE(S, k)).

Si l'ensemble des clés, ou l'ensemble lui-même, est totalement ordonné, d'autres opérations sont possibles :

- MINIMUM(S) : renvoie l'élément de S de clé minimale.
- MAXIMUM(S) : renvoie l'élément de S de clé maximale.
- SUCCESSEUR(S, x) : renvoie, si celui-ci existe, l'élément de S immédiatement plus grand que l'élément de S pointé par x, et NIL dans le cas contraire.
- PRÉDÉCESSEUR(S, x) : renvoie, si celui-ci existe, l'élément de S immédiatement plus petit que l'élément de S pointé par x, et NIL dans le cas contraire.

5.2 Piles et files

5.2.1 Piles

Définition 7 (Pile). *Une pile est une structure de données mettant en œuvre le principe « dernier entré, premier sorti » (*LIFO : Last-In, First-Out *en anglais).*

L'élément ôté de l'ensemble par l'opération SUPPRESSION est spécifié à l'avance (et donc cette opération ne prend alors que l'ensemble comme argument) : l'élément supprimé est celui le plus récemment inséré. L'opération INSERTION dans une pile est communément appelée EMPILER, et l'opération SUPPRESSION, DÉPILER. La figure 5.1 montre les conséquences des opérations EMPILER et DÉPILER sur une pile.

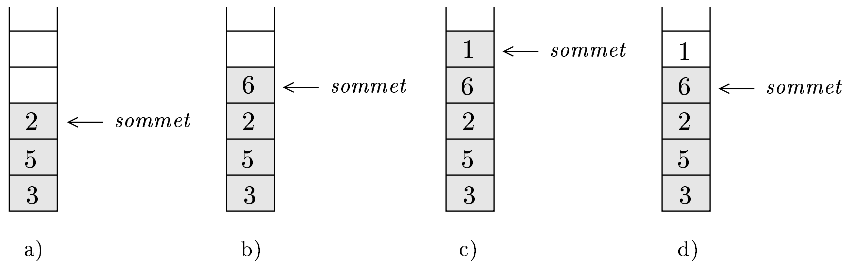


Fig. 5.1 – Exemple de pile : a) initialement la pile contient les valeurs 3, 5 et 2 ; b) état de la pile après l'opération Empiler(6) ; c) état de la pile après l'opération Empiler(1) ; d) état de la pile après l'opération Dépiler, qui a renvoyé la valeur 1.

Il est facile d'implémenter une pile au moyen d'un tableau, comme le montre la figure 5.2. La seule difficulté dans cette implémentation est la gestion des débordements de pile qui interviennent quand on tente d'effecteur l'opération Dépiler sur une pile vide et l'opération Empiler sur un tableau codant la pile qui est déjà plein. Ce dernier problème n'apparaît pas lorsque l'on implémente les piles au moyen d'une structure de données dont la taille n'est pas fixée *a priori* (comme une liste chaînée). Les algorithmes réalisant les fonctions Empiler et Dépiler, ainsi que la nécessaire fonction auxiliaire Pile-Vide, sont présentés figure 5.3.

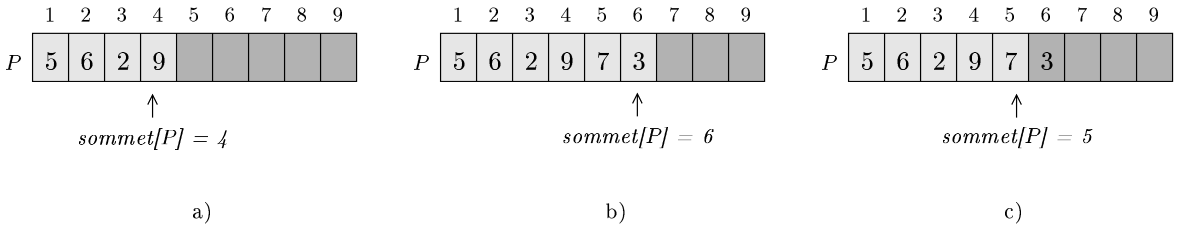


Fig. 5.2 – Implémentation d'une pile par un tableau : a) état initial de la pile ; b) nouvel état après les actions Empiler(7) et Empiler(3) ; c) nouvel état après l'opération Dépiler qui a renvoyé la valeur 3.

5.2.2 Files

Définition 8 (File). *Une file est une structure de données mettant en œuvre le principe « premier entré, premier sorti » (*FIFO : First-In, First-Out *en anglais).*

L'élément ôté de l'ensemble par l'opération Suppression est spécifié à l'avance (et donc cette opération ne prend alors que l'ensemble comme argument) : l'élément supprimé est celui qui est resté le plus longtemps dans la file. Une file se comporte exactement comme une file d'attente de la vie courante. La figure 5.4 montre les conséquences des opérations Insertion et Suppression sur une file.

On peut implémenter les files au moyen de tableaux. La figure 5.5 illustre l'implémentation de files à $n-1$ éléments au moyen d'un tableau à n éléments et de deux attributs :

- *tête*(F) qui indexe (ou pointe) vers la tête de la file ;
- *queue*(F) qui indexe le prochain emplacement où sera inséré un élément nouveau.

Les éléments de la file se trouvent donc aux emplacements *tête*(F), *tête*(F)+1, ..., *queue*(F)-1 (modulo n). Quand *tête*(F) = *queue*(F), la liste est vide. Les algorithmes réalisant les fonctions Insertion et Suppression, ainsi que la nécessaire fonction auxiliaire File-Vide, sont présentés figure 5.6. La seule difficulté dans cette implémentation est la gestion des débordements de file qui interviennent quand on tente d'effectuer l'opération Suppression sur une pile vide et l'opération Insertion sur un tableau codant la file qui est

PILE-VIDE(P)
si $sommet(P)$=0 **alors renvoyer** VRAI
sinon renvoyer FAUX

EMPILER(P, x)
si $sommet(P) = longueur(P)$ **alors erreur** « débordement positif »
sinon $sommet(P) \leftarrow sommet(P)+1$
$P[sommet(P)] \leftarrow x$

DÉPILER(P)
si PILE-VIDE(P) **alors erreur** « débordement négatif »
sinon $sommet(P) \leftarrow sommet(P)-1$
renvoyer $P[sommet(P)+1]$

FIG. 5.3 – Algorithmes de manipulation des piles implémentées par des tableaux.

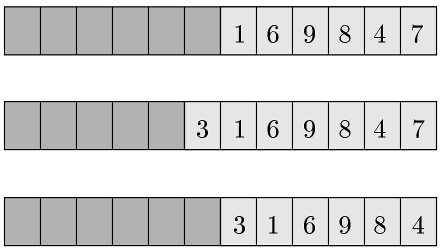


FIG. 5.4 – Exemple de file : a) initialement la file contient les valeurs 7, 4, 8, 9, 6 et 1 (de la plus anciennement à la plus récemment insérée) ; b) état de la file après l'opération INSERTION(3) ; c) état de la file après l'opération SUPPRESSION qui a renvoyé la valeur 7.

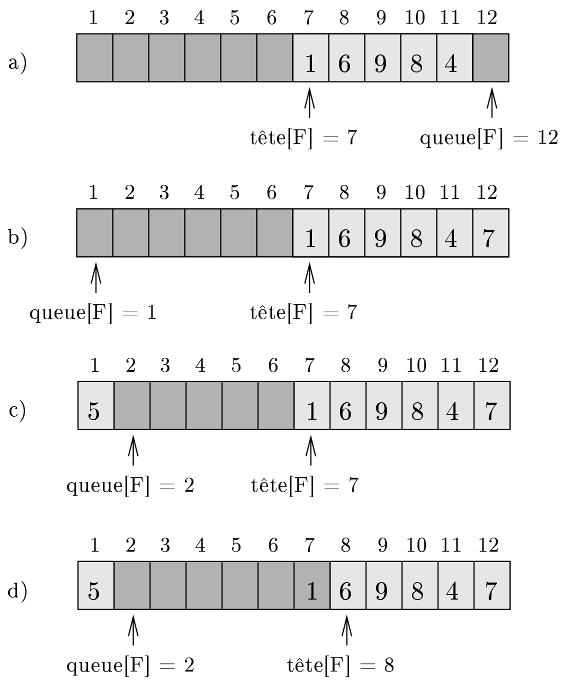


FIG. 5.5 – Implémentation d'une file par un tableau : a) état initial de la file ; b) nouvel état après l'action INSERTION(7) ; d) nouvel état après l'action INSERTION(5) ; d) nouvel état après l'opération SUPPRESSION qui a renvoyé la valeur 1.

déjà plein. Ce dernier problème n'apparaît pas lorsque l'on implémente les files au moyen d'une structure de donnée dont la taille n'est pas fixée *a priori* (comme une liste doublement chaînée).

File-Vide(F)
  si $tête(F)=queue(F)$ **alors renvoyer** Vrai
                sinon renvoyer Faux

Insertion(F, x)
  si $queue(F) + 1$ (modulo n) $=$ $tête(F)$ **alors erreur** « débordement positif »
                sinon $F[queue(F)] \leftarrow x$
                    $queue(F) \leftarrow queue(F)+1$

Suppression(F)
  si File-Vide(F) **alors erreur** « débordement négatif »
                sinon $tête(F) \leftarrow tête(F)+1$
                    renvoyer $F[tête(F)-1]$

Fig. 5.6 – Algorithmes de manipulation des files implémentées par des tableaux.

5.3 Listes chaînées

5.3.1 Définitions

Définition 9 (Liste chaînée). *Une liste chaînée est une structure de données dans laquelle les objets sont arrangés linéairement, l'ordre linéaire étant déterminé par des pointeurs sur les éléments.*

Chaque élément de la liste, outre le champ *clé*, contient un champ *successeur* qui est pointeur sur l'élément suivant dans la liste chaînée. Si le champ *successeur* d'un élément vaut Nil, cet élément n'a pas de successeur et est donc le dernier élément ou la **queue** de la liste. Le premier élément de la liste est appelé la **tête** de la liste. Une liste L est manipulée via un pointeur vers son premier élément, que l'on notera Tête(L). Si Tête(L) vaut Nil, la liste est vide.

La figure 5.7 présente un exemple de liste chaînée et montre les conséquences des opérations Insertion et Suppression sur une telle structure de données.

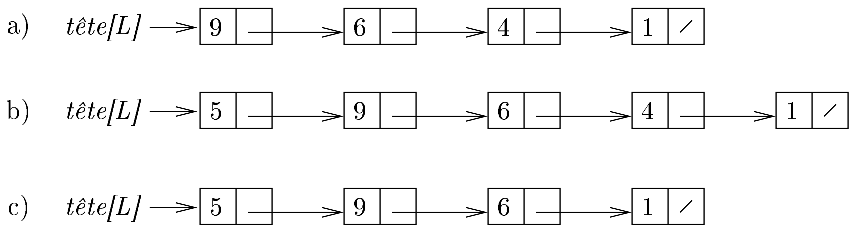


Fig. 5.7 – Exemple de liste chaînée : a) initialement la liste chaînée contient les valeurs 9, 6, 4 et 1 ; b) état de la liste chaînée après l'opération Insertion(5) ; c) état de la liste chaînée après l'opération Suppression(4).

Une liste chaînée peut prendre plusieurs formes :

- **Liste doublement chaînée** : en plus du champ *successeur*, chaque élément contient un champ *prédécesseur* qui est un pointeur sur l'élément précédant dans la liste. Si le champ *prédécesseur* d'un élément vaut Nil, cet élément n'a pas de prédécesseur et est donc le premir élément ou la **tête** de la liste. Une liste qui n'est pas doublement chaînée est dite **simplement chaînée**.

La figure 5.8 présente un exemple de liste doublement chaînée et montre les conséquences des opérations Insertion et Suppression sur une telle structure de données.

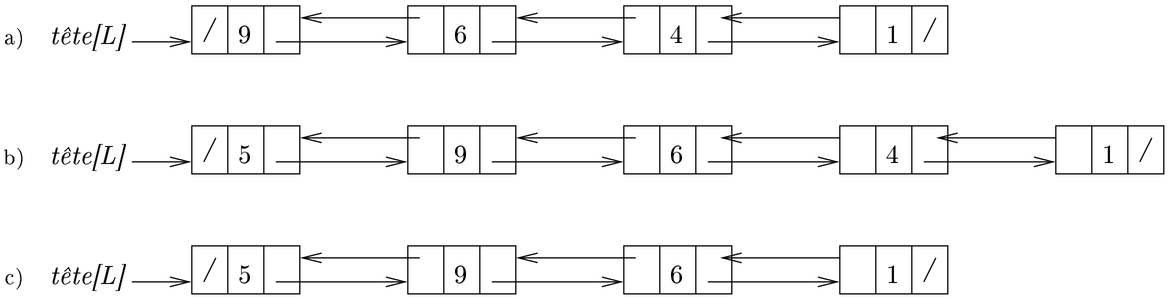


Fig. 5.8 – Exemple de liste doublement chaînée : a) initialement la liste contient les valeurs 9, 6, 4 et 1 ; b) état de la liste après l'opération Insertion(5) ; c) état de la liste après l'opération Suppression(4).

- **Triée** ou **non triée** : suivant que l'ordre linéaire des éléments dans la liste correspond ou non à l'ordre linéaire des clés de ces éléments.
- **Circulaire** : si le champ *précécesseur* de la tête de la liste pointe sur la queue, et si le champ *successeur* de la queue pointe sur la tête. La liste est alors vue comme un anneau.

5.3.2 Algorithmes de manipulation des listes chaînées

Recherche

L'algorithme Recherche-Liste(L, k) trouve le premier élément de clé k dans la liste L par une simple recherche linéaire, et retourne un pointeur sur cet élément. Si la liste ne contient aucun objet de clé k, l'algorithme renvoie Nil.

Recherche-Liste(L, k)
  $x \leftarrow$ Tête(L)
  tant que $x \neq$ Nil et *clé*(x) $\neq k$ **faire**
    $x \leftarrow$ *successeur*(x)
  renvoyer x

Cet algorithme manipule aussi bien des listes simplement que doublement que simplement chaînées.

Insertion

Étant donné un élément x et une liste L, l'algorithme Insertion-Liste insère x en tête de L.

Insertion-Liste(L, x)
  successeur(x) $\leftarrow$ Tête(L)
  si Tête(L) $\neq$ Nil **alors** *prédecesseur*(Tête(L)) $\leftarrow x$
  Tête(L) $\leftarrow x$
  prédecesseur(x) $\leftarrow$ Nil

Cet algorithme est écrit pour les listes doublement chaînées. Il suffit d'ignorer les deux instructions concernant le champ *prédécesseur* pour obtenir l'algorithme équivalent pour les listes simplement chaînées.

Suppression

L'algorithme Suppression-Liste élimine un élément x d'une liste chaînée L. Cet algorithme a besoin d'un pointeur sur l'élément x à supprimer. Si on ne possède que la clé de cet élément, il faut préalablement utiliser l'algorithme Recherche-Liste pour obtenir le pointeur nécessaire.

SUPPRESSION-LISTE(L, x)
  si *prédécesseur*(x) $\neq$ NIL
    alors *successeur*(*prédécesseur*(x)) $\leftarrow$ *successeur*(x)
    sinon TÊTE(L) $\leftarrow$ *successeur*(x)
  si *successeur*(x) $\neq$ NIL
    alors *prédécesseur*(*successeur*(x)) $\leftarrow$ *prédécesseur*(x)

Cet algorithme est écrit pour les listes doublement chaînées. L'algorithme équivalent pour les listes simplement chaînées est plus compliqué puisqu'avec les listes simplement chaînées nous n'avons pas de moyen simple de récupérer un pointeur sur l'élément qui précède celui à supprimer...

SUPPRESSION-LISTE(L, x)
  si x = TÊTE(L)
    alors TÊTE(L) $\leftarrow$ *successeur*(x)
    sinon $y \leftarrow$ TÊTE(L)
        tant que *successeur*(y) $\neq x$ **faire** $y \leftarrow$ *successeur*(y)
        successeur(y) $\leftarrow$ *successeur*(x)

5.3.3 Comparaison entre tableaux et listes chaînées

Aucune structure de données n'est parfaite, chacune a ses avantages et ses inconvénients. La figure 5.9 présente un comparatif des listes simplement chaînées, doublement chaînées et des tableaux, triés ou non, sur des opérations élémentaires. Les complexités indiquées sont celles du *pire cas*. Suivant les opérations que nous aurons à effectuer, et suivant leurs fréquences relatives, nous choisirons l'une ou l'autre de ces structures de données.

	liste chaînée simple non triée	liste chaînée simple triée	liste chaînée double non triée	liste chaînée double triée	tableau non trié	tableau trié
RECHERCHE(L, k)	$\Theta(n)$[a]	$\Theta(n)$[a]	$\Theta(n)$[a]	$\Theta(n)$[a]	$\Theta(1)$[b]	$\Theta(1)$[b]
INSERTION(L, x)	$\Theta(1)$	$\Theta(n)$[e]	$\Theta(1)$	$\Theta(n)$[e]	$\Theta(n)$[c] ou erreur[f]	$\Theta(n)$[d] $\Theta(n)$[c] ou erreur[f]
SUPPRESSION(L, x)	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$[g]	$\Theta(n)$[g]
SUCCESSEUR(L, x) [h]	$\Theta(n)$[i]	$\Theta(1)$	$\Theta(n)$[i]	$\Theta(1)$	$\Theta(n)$[i]	$\Theta(1)$
PRÉDÉCESSEUR(L, x) [h]	$\Theta(n)$[i]	$\Theta(n)$[j]	$\Theta(n)$[i]	$\Theta(1)$	$\Theta(n)$[i]	$\Theta(1)$
MINIMUM(L)	$\Theta(n)$[i]	$\Theta(1)$	$\Theta(n)$[i]	$\Theta(1)$	$\Theta(n)$[i]	$\Theta(1)$
MAXIMUM(L)	$\Theta(n)$[i]	$\Theta(n)$[k]	$\Theta(n)$[i]	$\Theta(n)$[k]	$\Theta(n)$[i]	$\Theta(1)$

[a] Dans le pire cas il faut parcourir tous les éléments pour se rendre compte que la clef n'était pas dans l'ensemble.
[b] La clé étant l'indice de l'élément dans le tableau.
[c] Dans le pire cas, il faut allouer un nouveau tableau et recopier tous les éléments de l'ancien tableau dans le nouveau.
[d] Dans le pire cas, l'insertion a lieu dans la première cas du tableau, et il faut décaler tous les éléments déjà présents.
[e] Au pire, l'insertion a lieu en fin de liste.
[f] Au cas où l'on veut effectuer une insertion dans un tableau déjà plein et qu'il n'est pas possible d'effectuer une allocation dynamique de tableau, comme en FORTRAN 77 ou en PASCAL.
[g] Dans le pire cas on supprime le premier élément du tableau et il faut décaler tous les autres éléments.
[h] Au sens de l'ordre sur la valeur des clés.
[i] Complexité de la recherche du maximum (ou du minimum) dans un ensemble à n éléments...
[j] Complexité de la recherche du maximum dans un ensemble à n éléments... car il faut entreprendre la recherche du prédécesseur depuis le début de la liste.
[k] Il faut parcourir la liste en entier pour trouver son dernier élément.

FIG. 5.9 – Efficacités respectives des listes chaînées et des tableaux.

Chapitre 6

Programmation dynamique

La programmation dynamique, comme la méthode « diviser pour régner » (cf. section 3.3), résout les problèmes en combinant les solutions de sous-problèmes. La programmation dynamique s'applique quand les sous-problèmes ne sont pas indépendants mais ont des sous-sous-problèmes en commun. Dans ce cas, un algorithme « diviser pour régner » fait plus de travail que nécessaire, en résolvant plusieurs fois les sous-sous-problèmes communs. Un algorithme de programmation dynamique résout chaque sous-sous-problème une unique fois et mémorise sa solution dans un tableau, s'épargnant ainsi le recalcul de la solution chaque fois que le sous-sous-problème est rencontré.

La programmation dynamique est en général appliquée aux **problèmes d'optimisation** : ces problèmes peuvent admettre plusieurs solutions, parmi lesquelles on veut choisir *une* solution optimale (maximale ou minimale pour une certaine fonction de coût).

Le *développement* d'un algorithme de programmation dynamique peut être planifié en quatre étapes :

1. Caractériser la structure d'une solution optimale.
2. Définir récursivement la valeur d'une solution optimale.
3. Calculer la valeur d'une solution optimale partant des cas simples (cas d'arrêt des récursions) et en remontant progressivement jusqu'à l'énoncé du problème initial.
4. Construire une solution optimale pour les informations calculées (si l'on souhaite avoir une solution et pas seulement la valeur d'une solution optimale).

6.1 Multiplication d'une suite de matrices

On suppose que l'on a une suite de n matrices, A_1, ..., A_n, et que l'on souhaite calculer le produit :

$$A_1 A_2 ... A_n.$$

On peut évaluer cette expression en utilisant comme sous-programme l'algorithme classique de multiplications de matrices (cf. section 3.3.2), après avoir *complètement parenthésé* cette expression afin de lever toute ambiguïté sur l'ordre des multiplications de matrices —un produit de matrices complètement parenthésé est soit une matrice unique soit le produit de deux produits de matrice complètement parenthésés). La multiplication de matrices étant associative, le résultat de la multiplication est indépendant du parenthésage. Il y a ainsi cinq manières différentes de calculer le produit de quatre matrices :

$$\begin{aligned} A_1 A_2 A_3 A_4 \quad &= (A_1(A_2(A_3 A_4))) \\ &= (A_1((A_2 A_3)A_4)) \\ &= ((A_1 A_2)(A_3 A_4)) \\ &= ((A_1(A_2 A_3))A_4) \\ &= (((A_1 A_2)A_3)A_4) \end{aligned}$$

Le parenthésage du produit peut avoir un impact crucial sur le coût de l'évaluation du produit. Le produit d'une matrice A de taille $p \times q$ par une matrice B de taille $q \times r$ produit une matrice C de taille $p \times r$ en pqr

multiplications scalaires. Considérons trois matrices A_1, A_2 et A_3 de dimensions respectives 10×100, 100×5 et 5×50. Si on effectue la multiplication de ces trois matrices suivant le parenthésage $((A_1 A_2)A_3$, on effectue $10 \times 100 \times 5 = 5\ 000$ multiplications dans un premier temps, puis $10 \times 5 \times 50 = 2\ 500$ dans un deuxième temps, soit 7 500 au total. Si, au contraire, on effectue la multiplication suivant le parenthésage $(A_1(A_2 A_3))$ on effectue $100 \times 5 \times 50 = 25\ 000$ multiplications dans un premier temps, puis $10 \times 100 \times 50 = 50\ 000$ dans un deuxième temps, soit 75 000 au total et 10 fois plus qu'avec le premier parenthésage !

Problématique

Problématique de la multiplication d'une suite de matrices : étant donnée une suite A_1, ..., A_n de n matrices, où pour $i = 1, 2, ..., n$ la matrice A_i est de dimensions $p_{i-1} \times p_i$, parenthéser complètement le produit $A_1 A_2 ... A_n$ de façon à minimiser le nombre de multiplications scalaires.

Nombre de parenthésages

Le passage en revue de tous les parenthésages possibles ne donnera pas un algorithme efficace, c'est pourquoi il faut avoir recours à une technique plus sophistiquée.

Soit $P(n)$ le nombre de parenthésages possibles d'une séquence de n matrices. On peut couper une séquence de n matrices entre la k^{e} et la $(k+1)^{\text{e}}$, pour k prenant n'importe quelle valeur dans l'intervalle $[1, n-1]$, puis parenthéser les deux sous-séquences résultantes indépendamment. D'où la récurrence :

$$P(n) = \begin{cases} 1 & \text{si } n = 1, \\ \sum_{k=1}^{n-1} P(k)P(n-k) & \text{si } n \geq 2. \end{cases}$$

On peut montrer que

$$P(n) = \frac{1}{n} C_{2n-2}^{n-1} = \Omega\left(\frac{4^n}{n^{3/2}}\right).$$

Le nombre de solutions est donc au moins exponentiel en n et la méthode directe consistant à effectuer une recherche exhaustive est donc une stratégie médiocre...

Structure d'un parenthésage optimal

La première étape du paradigme de la programmation dynamique consiste à caractériser la structure d'une solution optimale.

Nous notons $A_{i..j}$ la matrice résultant de l'évaluation du produit $A_i A_{i+1} ... A_{j-1} A_j$. Un parenthésage optimal de $A_1 A_2 ... A_n$ sépare le produit entre A_k et A_{k+1} pour une certaine valeur k. Dans notre solution optimale on commence donc par calculer les matrices $A_{1..k}$ et $A_{k+1..n}$ puis on les multiplie pour obtenir la matrice $A_{1..n}$ recherchée. Le coût du calcul est donc la somme des coûts des calculs des matrices $A_{1..k}$ et $A_{k+1..n}$ et de leur produit. Par conséquent le parenthésage de la sous-suite $A_1 ... A_k$ (et celui de la sous-suite $A_{k+1} .. A_n$) doit être optimal : sinon, on le remplace par un parenthésage plus économique, et on obtient un parenthésage global plus efficace que... le parenthésage optimal !

Par conséquent, une solution optimale à une instance du problème de multiplication d'une suite de matrices utilise uniquement des solutions optimales aux instances des sous-problèmes. La sous-structure optimale à l'intérieur d'une solution optimale est l'une des garanties de l'applicabilité de la programmation dynamique.

Résolution récursive

La deuxième étape du paradigme de la programmation dynamique consiste à définir récursivement la valeur d'une solution optimale en fonction de solutions optimales aux sous-problèmes.

Pour le problème de la multiplication d'une suite de matrices, on prend comme sous-problèmes les problèmes consistant à déterminer le coût minimum d'un parenthésage de $A_i A_{i+1} ... A_j$, pour $1 \leq i < j \leq n$. Soit $m[i, j]$ le nombre minimum de multiplications scalaires nécessaires au calcul de $A_i A_{i+1} ... A_j = A_{i..j}$.

Pour tout i, $m[i, i] = 0$ car $A_{i..i} = A_i$ et aucune multiplication n'est nécessaire. Considérons un couple (i, j) avec $i < j$. Supposons qu'un parenthésage optimal sépare le produit $A_i A_{i+1} ... A_j$ entre A_k et A_{k+1}.

Alors, $m[i,j]$, le coût du calcul de $A_{i..j}$ est égal au coût du calcul de $A_{i..k}$, plus celui de $A_{k+1..j}$, plus celui du produit de ces deux matrices. Nous avons donc :

$$m[i,j] = m[i,k] + m[k+1,j] + p_{i-1}p_kp_j.$$

Cette équation nécessite la connaissance de la valeur de k, connaissance que nous n'avons pas. Il nous faut donc passer en revue tous les cas possibles et il y en a $j-i$:

$$m[i,j] = \begin{cases} 0 & \text{si } i = j, \\ \min_{i \leq k < j} \{m[i,k] + m[k+1,j] + p_{i-1}p_kp_j\} & \text{si } i < j. \end{cases} \tag{6.1}$$

$m[i,j]$ nous donne le coût d'une solution optimale. Pour pouvoir construire une telle solution on note $s[i,j]$ une valeur k telle que $m[i,j] = m[i,k] + m[k+1,j] + p_{i-1}p_kp_j$.

Algorithme récursif

Une première solution à notre problème pourrait être l'algorithme ChaîneDeMatrices-Récursif ci-dessous qui est une utilisation directe de la récursion 6.1

```
ChaîneDeMatrices-Récursif(p, i, j)
  si i = j alors retourner 0
  m[i,j] ← +∞
  pour k ← 1 à j − 1 faire
    q ← ChaîneDeMatrices-Récursif(p, i, k)
      + ChaîneDeMatrices-Récursif(p, k + 1, j)
      + p_{i−1} p_k p_j
    si q < m[i,j] alors m[i,j] ← q
  renvoyer m[i,j]
```

La complexité de cet algorithme est donné par la récurrence :

$$T(n) = \begin{cases} 1 & \text{si } n = 1, \\ 1 + \sum_{k=1}^{n-1} (T(k) + T(n-k) + 1) & \text{pour } n > 1. \end{cases}$$

Dans le cas général, cette complexité peut se récrire :

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

Par conséquent $T(n) \geq 2T(n-1)$ et $T(n) = \Omega(2^n)$. La quantité totale de travail effectué par l'appel ChaîneDeMatrices-Récursif(P, 1, n) est donc au moins exponentiel et est donc prohibitif... Heureusement, on peut mieux faire.

Calcul des coûts optimaux

En fait, le nombre de sous-problèmes est assez réduit : un problème pour chaque choix de i et de j tels que $1 \leq i \leq j \leq n$, soit au total $C_n^2 + n = \Theta(n^2)$ choix. L'algorithme récursif rencontre chaque sous-problème un grand nombre de fois (ici, un nombre exponentiel de fois) dans différentes branches de l'arbre des appels récursifs. Cette propriété, dite des sous-problèmes superposés (des sous-problèmes ont des sous-sous-problèmes en commun), est le deuxième indice de l'applicabilité de la programmation dynamique.

Plutôt que d'implémenter de manière récursive l'équation 6.1, on aborde la troisième étape du paradigme de la programmation dynamique : on calcule le coût optimal en utilisant une approche ascendante. L'entrée de l'algorithme Ordonner-ChaîneDeMatrices ci-dessous est la séquence $p_0, p_1, ..., p_n$ des dimensions des matrices. Cet algorithme calcul le coût optimal $m[i,j]$ et enregistre un indice $s[i,j]$ permettant de l'obtenir.

Ordonner-ChaîneDeMatrices(p)
  $n \leftarrow longueur(p) - 1$
  pour $i \leftarrow 1$ **à** n **faire** $m[i,i] \leftarrow 0$
  pour $l \leftarrow 2$ **à** n **faire**
    pour $i \leftarrow 1$ **à** $n-l+1$ **faire**
      $j \leftarrow i+l-1$
      $m[i,j] \leftarrow +\infty$
      pour $k \leftarrow 1$ **à** $j-1$ **faire**
        $q \leftarrow m[i,k] + m[k+1,j] + p_{i-1}p_kp_j$
        si $q < m[i,j]$ **alors** $m[i,j] \leftarrow q$
                    $s[i,j] \leftarrow k$
  renvoyer m **et** s

L'algorithme remplit le tableau m en considérant des suites de matrices de longueur croissante. L'équation 6.1 nous montre en effet que le calcul du coût d'un produit de m matrices ne dépend que des coûts de calcul de suites de matrices de longueur strictement inférieure. La boucle sur l est une boucle sur la longueur des suites considérées.

La figure 6.1 présente un exemple d'exécution de l'algorithme Ordonner-ChaîneDeMatrices. Comme $m[i,j]$ n'est défini que pour $i \leq j$, seule la partie du tableau m strictement supérieure à la diagonale principale est utilisée. Les deux tableaux sont présentés de manière à faire apparaître la diagonale principale de m horizontalement, chaque rangée horizontale contenant les éléments correspondants à des chaînes de matrices de même taille. Ordonner-ChaîneDeMatrices calcule les rangées de m du bas vers le haut, et chaque rangée de la gauche vers la droite. Dans notre exemple, un parenthésage optimal coûte 15 125 multiplications scalaires.

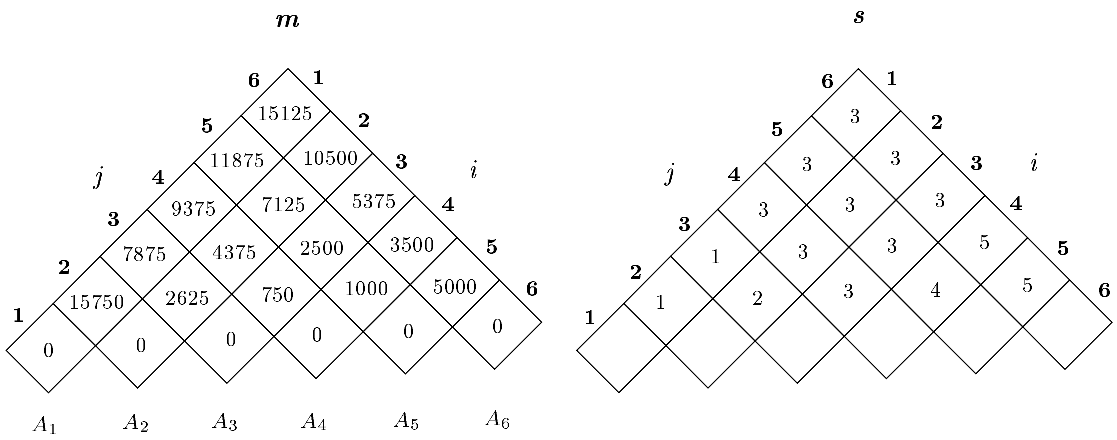


Fig. 6.1 – Tableaux m et s calculés par Ordonner-ChaîneDeMatrices pour $n = 6$ et les dimensions : $30, 35, 15, 5, 10, 20, 25$.

Complexité

Un simple coup d'œil à l'algorithme montre que sa complexité est en $O(n^3)$. Plus précisément :

$$\begin{aligned} T(n) &= \sum_{l=2}^{n} \sum_{i=1}^{n-l+1} \sum_{k=i}^{i+l-2} 1 \\ &= \sum_{l=2}^{n} \sum_{i=1}^{n-l+1} (l-1) \\ &= \sum_{l=2}^{n} (n-l+1)(l-1) \\ &= \sum_{l=2}^{n} (-l^2 + l(n+2) - (n+1)) \\ &= \frac{n^3+5n+12}{6} \end{aligned}$$

sachant que $\sum_{i=1}^{n} i^3 = \frac{n(n+1)(2n+1)}{6}$. La complexité de l'algorithme Ordonner-ChaîneDeMatrices est donc en $\Theta(n^3)$ ce qui est infiniment meilleur que la solution naïve énumérant tous les parenthésages ou que la solution récursive, toutes deux de complexité exponentielle.

Construction d'une solution optimale

L'algorithme Ordonner-ChaîneDeMatrices calcule le coût d'un parenthésage optimal, mais n'effectue pas la multiplication de la suite de matrices. Par contre, l'information nécessaire à la réalisation d'un calcul suivant un parenthésage optimal est stockée au fur et à mesure dans le tableau s : $s[i,j]$ contient une valeur k pour laquelle une séparation du produit $A_i A_{i+1} ... A_j$ entre A_k et A_{k+1} fourni un parenthésage optimal. L'algorithme Multiplier-ChaîneDeMatrices ci-dessous réalise la multiplication et résout donc notre problème.

```
Multiplier-ChaîneDeMatrices(A, s, i, j)
  si j > i
    alors X ← Multiplier-ChaîneDeMatrices(A, s, i, s[i,j])
          X ← Multiplier-ChaîneDeMatrices(A, s, s[i,j] + 1, j)
          renvoyer Multiplier-Matrices(X, Y)
    sinon renvoyer A_i
```

Dans l'exemple de la figure 6.1, Multiplier-ChaîneDeMatrices(A, s, 1, 6) calcule le produit de la suite de matrices en suivant le parenthésage :

$$((A_1(A_2A_3))((A_4A_5)A_6)),$$

car $s[1,6] = 3$, $s[1,3] = 1$ et $s[4,6] = 5$.

6.2 Éléments de programmation dynamique

On examine ici les deux caractéristiques principales que doit posséder un problème d'optimisation pour que la programmation dynamique soit applicable : une sous-structure optimale et des sous-problèmes superposés. On examinera aussi une variante de ce paradigme : le recensement.

6.2.1 Sous-structure optimale

Un problème fait apparaître une **sous-structure optimale** si une solution optimale au problème fait apparaître des solutions optimales aux sous-problèmes. La présence d'une sous-structure optimale est un bon indice de l'utilité de la programmation dynamique (mais cela peut aussi signifier qu'une stratégie gloutonne est applicable, cf. chapitre 7). La sous-structure optimale d'un problème suggère souvent une classe de sous-problèmes pertinents auxquels on peut appliquer la programmation dynamique.

6.2.2 Sous-problèmes superposés

La seconde caractéristique que doit posséder un problème d'optimisation pour que la programmation dynamique soit applicable est « l'étroitesse » de l'espace des sous-problèmes, au sens où un algorithme récursif doit résoudre constamment les mêmes sous-problèmes, plutôt que d'en engendrer toujours de nouveaux. En général, le nombre de sous-problèmes distincts est polynomial par rapport à la taille de l'entrée. Quand un algorithme récursif repasse sur le même problème constamment, on dit que le problème d'optimisation contient des **sous-problèmes superposés**. *A contrario*, un problème pour lequel l'approche « diviser pour régner » est plus adaptée génère le plus souvent des problèmes nouveaux à chaque étape de la récursivité. Les algorithmes de programmation dynamique tirent parti de la superposition des sous-problèmes en résolvant chaque sous-problème une unique fois, puis en conservant la solution dans un tableau où on pourra la retrouver au besoin avec un temps de recherche constant.

6.2.3 Recensement

Il existe une variante de la programmation dynamique qui offre souvent la même efficacité que l'approche usuelle, tout en conservant une stratégie descendante. Son principe est de **recenser** les actions naturelles, mais inefficaces, de l'algorithme récursif. Comme pour la programmation dynamique ordinaire, on conserve dans un tableau les solutions aux sous-problèmes, mais la structure de remplissage du tableau est plus proche de l'algorithme récursif. Un algorithme récursif de recensement maintient à jour un élément de tableau pour la solution de chaque sous-problème. Chaque élément contient au départ une valeur spéciale pour indiquer qu'il n'a pas encore été rempli. Lorsque le sous-problème est rencontré pour la première fois durant l'exécution de l'algorithme récursif, sa solution est calculée puis stockée dans le tableau. À chaque nouvelle confrontation avec ce sous-problème, la valeur stockée est simplement récupérée.

RECENSEMENT-CHAÎNEDEMATRICES(p)
  $n \leftarrow longueur(p) - 1$
  pour $i \leftarrow 1$ **à** n **faire**
    pour $j \leftarrow i$ **à** n **faire**
      $m[i, j] \leftarrow +\infty$
  renvoyer RÉCUPÉRATION-CHAÎNE(p, 1, n)

RÉCUPÉRATION-CHAÎNE(p, i, j)
  si $m[i, j] < +\infty$ **alors renvoyer** $m[i, j]$
  si $i = j$
    alors $m[i, j] \leftarrow 0$
    sinon pour $k \leftarrow i$ **à** $j - 1$ **faire**
      $q \leftarrow$ RÉCUPÉRATION-CHAÎNE(p, i, k) + RÉCUPÉRATION-CHAÎNE($p, k+1, j$) + $p_{i-1}p_kp_j$
      si $q < m[i, j]$ **alors** $m[i, j] \leftarrow q$
                  $s[i, j] \leftarrow k$

Chacun des $\Theta(n^2)$ éléments du tableau m est rempli une unique fois par RÉCUPÉRATION-CHAÎNE et chacun de ces $\Theta(n^2)$ appels à RÉCUPÉRATION-CHAÎNE requiert un temps en $O(n)$ —en excluant le temps passé à calculer d'autres éléments éventuels. La complexité de RÉCUPÉRATION-CHAÎNE est donc en $O(n^3)$.

En pratique, si tous les sous-problèmes doivent être résolus au moins une fois, un algorithme ascendant de programmation dynamique bat en général un algorithme descendant avec recensement d'un facteur constant car il élimine le temps pris par les appels récursifs et prend moins de temps pour gérer le tableau. En revanche, si certains sous-problèmes de l'espace des sous-problèmes n'ont pas besoin d'être résolus du tout, la solution du recensement présente l'avantage de ne résoudre que ceux qui sont vraiment nécessaires.

Chapitre 7

Algorithmes gloutons

Les algorithmes qui résolvent les problèmes d'optimisation parcourent en général une série d'étapes, au cours desquelles ils sont confrontés à un ensemble d'options. Pour de nombreux problèmes d'optimisation la programmation dynamique est une approche trop lourde pour déterminer les meilleures solutions ; d'autres algorithmes plus simples et efficaces y arriveront. Un **algorithme glouton** fait toujours le choix qui semble le meilleur sur le moment. Autrement dit, il fait un choix optimal localement, dans l'espoir que ce choix mènera à la solution optimale globalement.

Les algorithmes gloutons n'aboutissent pas toujours à des solutions optimales, mais la méthode gloutonne est très puissante et fonctionne correctement pour des problèmes variés.

7.1 Location d'une voiture

On considère le problème de la location d'une unique voiture. Des clients formulent un ensemble de demandes de location avec, pour chaque demande, le jour du début de la location et le jour de restitution du véhicule. Notre but ici est d'affecter le véhicule de manière à satisfaire le *maximum de clients* possible (et non pas de maximiser la somme des durées des locations).

Nous disposons donc d'un ensemble E des demandes de location avec, pour chaque élément e de E, la date $d(e)$ du début de la location et la date $f(e)$ de la fin de cette location. Nous voulons obtenir un ensemble F maximal de demandes satisfaites. Cet ensemble F doit vérifier une unique contrainte : deux demandes ne doivent pas se chevaucher dans le temps, autrement dit une location doit se terminer avant que la suivante ne commence. Cette contrainte s'écrit mathématiquement :

$$\forall e_1 \in F, \forall e_2 \in F, \quad d(e_1) \leq d(e_2) \Rightarrow f(e_1) \leq d(e_2).$$

Algorithme

LocationDUneVoiture(E)
  Tri des éléments de E par date de fin croissante.
    On obtient donc une suite $e_1, e_2, ..., e_n$ telle que $f(e_1) \leq f(e_2) \leq ... \leq f(e_n)$.
  $F[1] \leftarrow e_1$
  $j \leftarrow 1$
  pour $i \leftarrow 1$ **à** n **faire**
    si $d(e_i) \geq f(F[j])$ **alors** $j \leftarrow j + 1$
                  $F[j] \leftarrow e_j$
  renvoyer F

Cet algorithme est glouton car à chaque étape il prend la location « la moins coûteuse » : celle qui finit le plus tôt parmi celles qui sont satisfiables.

Preuve de l'optimalité de l'algorithme

Soit $F = \{x_1, x_2, ..., x_p\}$ la solution obtenue par l'algorithme glouton, et soit $G = \{y_1, y_2, ..., y_q\}$, $q \geq p$, une solution optimale. Nous voulons montrer que F est optimal, et donc que $q = p$.

Nous supposons que les ensembles F et G sont classés par dates de fins de location croissantes. Si G ne contient pas F, il existe un entier k tel que : $\forall i < k, x_i = y_i$ et $x_k \neq y_k$. Par construction de F, x_k est une demande de location qui à la date de fin minimale et dont la date de début soit postérieure à la date de fin de $x_{k-1} = y_{k-1}$. Par conséquent, $f(y_k) \geq f(x_k)$. On peut alors remplacer G par $G' = \{y_1, y_2, ..., y_{k-1}, x_k, y_{k+1}, ..., y_q\}$ tout en satisfaisant la contrainte de non chevauchement des demandes. G' est une autre solution optimale mais ayant strictement plus d'éléments en commun avec F que G. En répétant autant que faire se peut ce procédé, on obtient un ensemble H de même cardinalité que G et qui contient F. Cet ensemble H ne peut contenir d'autres éléments que ceux de F car ceux-ci, débutants après la fin de x_p, auraient été ajoutés à F par l'algorithme glouton. Donc $H = F$, et F et G ont le même nombre d'éléments.

Limites de l'algorithme

Il est primordial, ici, que les demandes soit classées par dates de fin croissantes. Le tableau 7.1 présente trois demandes de location classées par dates de début croissantes pour lesquelles l'algorithme glouton présenté ci-dessus n'est pas optimal. Pour d'évidentes raisons de symétries, classer les demandes par dates de début décroissantes donne par contre un résultat optimal.

	e_1	e_2	e_3
d	2	3	5
F	8	4	8

TAB. 7.1 – Demandes classées par dates de début croissantes.

	e_1	e_2	e_3
d	3	5	3
F	6	7	5

TAB. 7.2 – Demandes classées par durées décroissantes.

L'algorithme glouton ne donne pas l'optimum si notre but est de maximiser la durée totale de location du véhicule. Même si on classe les demandes de location par durées décroissantes, un algorithme glouton ne donnera pas une solution optimale, le tableau 7.2 présentant un contre-exemple. En fait, le problème de la maximisation de cette durée totale est NP-complet (cf. chapitre 11) et on ne connaît pas d'algorithme de complexité polynomiale pour le résoudre.

Si nous disposons de deux voitures et non plus d'une seule, l'algorithme précédent ne donne plus l'optimum.

7.2 Éléments de la stratégie gloutonne

Un algorithme glouton détermine une solution après avoir effectué une série de choix. Pour chaque point de décision, le choix qui semble le meilleur à cet instant est retenu. Cette stratégie ne produit pas toujours une solution optimale. Il existe cependant deux caractéristiques qui indiquent qu'un problème se prête à une stratégie gloutonne : la propriété du choix glouton et une sous-structure optimale.

7.2.1 Propriété du choix glouton

Propriété du choix glouton : on peut arriver à une solution globalement optimale en effectuant un choix localement optimal (ou choix glouton). En programmation dynamique on fait un choix à chaque étape, mais ce choix dépend de la solution de sous-problèmes, au contraire, dans un algorithme glouton, on fait le choix qui semble le meilleur sur le moment *puis* on résout les sous-problèmes qui surviennent une fois le choix fait. Une stratégie gloutonne progresse en général de manière descendante en faisant se succéder les choix gloutons pour ramener itérativement chaque instance du problème à une instance « plus petite ».

7.2.2 Sous-structure optimale

Montrer qu'un choix glouton aboutit à un problème similaire mais « plus petit » ramène la démonstration de l'optimalité à prouver qu'une solution optimale doit faire apparaître une sous-structure optimale.

Un problème fait apparaître une **sous-structure optimale** si une solution optimale contient la solution optimale de sous-problèmes. Cette propriété est un indice important de l'applicabilité de la programmation dynamique comme des algorithmes gloutons. Le sujet du TD 6 montre un exemple de problème qui peut être résolu par programmation dynamique mais pas par un algorithme glouton.

7.3 Fondements théoriques des méthodes gloutonnes

La théorie des matroïdes ne couvre pas tous les cas d'applications de la méthode gloutonne, mais elle couvre de nombreux cas intéressants en pratique.

7.3.1 Matroïdes

Définition 10 (Matroïde). *Un matroïde est un couple $M = (E, I)$ vérifiant les conditions suivantes :*

1. *E est un ensemble fini non vide.*
2. *I est une famille non vide de sous-ensembles de E, appelés sous-ensembles **indépendants** de E, telle que si $H \in I$ et si $F \subset H$ alors $F \in I$ (on dit que I est **héréditaire**). Autrement dit, si I contient un sous-ensemble H de E, I contient tous les sous-ensembles de H. On remarque que l'ensemble vide est obligatoirement membre de I.*
3. *Si F et H sont deux éléments de I, avec $|F| < |H|$, alors il existe (au moins) un élément $x \in H \setminus F$ tel que $F \cup \{x\} \in I$ (**propriété d'échange**).*

Un premier résultat sur les matroïdes :

Théorème 2. *Tous les sous-ensembles indépendants maximaux d'un matroïde ont la même taille.*

Ce résultat est une conséquence directe de la propriété d'échange : si un de ces ensembles, H, est strictement plus petit que les autres, la propriété d'échange nous garantit que I contient un sur-ensemble strict H' de H, ce qui contredit la maximalité de H.

Définition 11 (Matroïde pondéré). *Un matroïde $M = (E, I)$ est dit **pondéré** si l'on dispose d'une fonction de pondération w qui affecte un poids strictement positif $w(x)$ à chaque élément x de E. La fonction de pondération w s'étend aux sous-ensembles de E. Soit F un sous-ensemble quelconque de E :*

$$w(F) = \sum_{x \in F} w(x).$$

7.3.2 Algorithmes gloutons sur un matroïde pondéré

De nombreux problèmes pour lesquels une approche gloutonne donne les solutions optimales peuvent être ramenés à une recherche d'un sous-ensemble indépendant de pondération maximale dans un matroïde pondéré. Autrement dit, on dispose d'un matroïde pondéré $M = (E, I)$ et on souhaite trouver un ensemble indépendant $F \in I$ pour lequel $w(F)$ est maximisé. Un tel sous-ensemble indépendant et qui possède la plus grande pondération possible est appelé sous-ensemble **optimal** du matroïde. Comme la pondération est strictement positive par définition, un sous-ensemble optimal est toujours un sous-ensemble indépendant maximal.

L'algorithme ci-dessous prend en entrée un matroïde pondéré $M = (E, I)$ et sa fonction de pondération w et retourne un sous-ensemble optimal F.

GLOUTON($M = (E, I)$, w)
  $F \leftarrow \emptyset$
  Trier E par ordre de poids décroissant

pour $x \in E$ par ordre de poids décroissant **faire**
  si $F \cup \{x\} \in I$ **alors** $F \leftarrow F \cup \{x\}$
renvoyer F

Cet algorithme est glouton parce qu'il considère les éléments de E par ordre de poids décroissant et qu'il ajoute immédiatement un élément x à F si $F \cup \{x\}$ est indépendant. Si E contient n éléments et si la vérification de l'indépendance de $F \cup \{x\}$ prend un temps $O(f(n)$, l'algorithme tout entier s'exécute en $O(n \log n + nf(n))$ —rappel : le tri d'un ensemble de n éléments coûte $O(n \log n)$.

Le sous-ensemble F de E est indépendant par construction. Nous allons maintenant établir l'optimalité de F.

Théorème 3 (Les matroïdes satisfont à la propriété du choix glouton). *Soit $M = (E, I)$ un matroïde pondéré de fonction de pondération w. Supposons que E soit trié par ordre de poids décroissant. Soit x le premier élément de E tel que $\{x\}$ soit indépendant, s'il existe. Si x existe, il existe un sous-ensemble optimal F de E contenant x.*

Si x n'existe pas, le seul élément de I est l'ensemble vide. Soit H un sous-ensemble optimal. On utilise H pour construire, au moyen de la propriété d'échange, un ensemble F maximal (de même cardinalité que H) et contenant x. Par construction, F et H ne diffèrent que d'un élément et il existe donc un élément y tel que : $F = (H \setminus \{y\}) \cup \{x\}$. Par maximalité du poids de x, $w(y) \leq w(x)$, $w(H) \leq w(F)$ et F est optimal.

Théorème 4. *Soit $M = (E, I)$ un matroïde quelconque. Si x est un élément de E tel que $\{x\}$ n'est pas élément de I, alors x n'appartient à aucun sous-ensemble indépendant F de E.*

Autrement dit, un élément qui n'est pas utilisable immédiatement ne pourra jamais être utilisé : l'algorithme GLOUTON ne fait donc pas d'erreur en ne considérant pas les éléments de E qui ne sont pas extension de $\emptyset$.

Théorème 5 (Les matroïdes satisfont la propriété de sous-structure optimale). *Soit x le premier élément de E choisi par* GLOUTON *pour le matroïde pondéré $M = (E, I)$. Le reste du problème —trouver un sous-ensemble indépendant contenant x et de poids maximal— se réduit à trouver un sous-ensemble indépendant et de poids maximal du matroïde pondéré $M' = (E', I')$, où :*

$$E' = \{y \in E : \{x, y\} \in I\},$$
$$I' = \{H \subset E \setminus \{x\} : H \cup \{x\} \in I\},$$

et où la fonction de pondération de M' est celle de M restreinte à E'.

Une solution de poids maximum sur M contenant x engendre une solution de poids maximum sur M', et vice versa.

Théorème 6 (Validité de l'algorithme glouton sur les matroïdes). *Si $M = (E, I)$ est un matroïde pondéré de fonction de pondération w, alors l'appel* GLOUTON*($M = (E, I)$, w) renvoie un sous-ensemble optimal.*

Chapitre 8

Graphes et arbres

8.1 Graphes

Un **graphe orienté** G est représenté par un couple $(S,\ A)$ où S est un ensemble fini et A une relation binaire sur S. L'ensemble S est **l'ensemble des sommets** de G et A est **l'ensemble des arcs** de G. La figure 8.1 est une représentation graphique du graphe orienté $G = (S, A)$ avec l'ensemble de sommets $S = \{1, 2, 3, 4, 5, 6\}$ et l'ensemble d'arcs $A = \{(1, 2), (2, 2), (2, 4), (2, 5), (4, 1), (4, 5), (5, 4), (6, 3)\}$; les sommets étant représentés par des cercles et les arcs par des flèches. On notera que les **boucles** — une boucle étant un arc qui relie un sommet à lui-même — sont ici possibles.

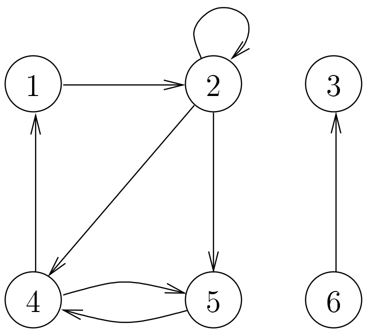


FIG. 8.1 – Exemple de graphe orienté.

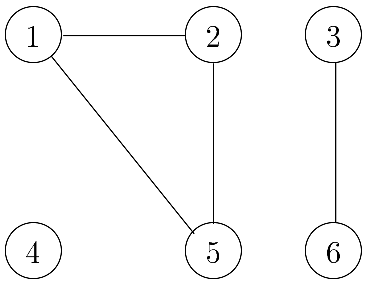


FIG. 8.2 – Exemple de graphe non orienté.

Dans un **graphe non orienté** $G = (S, A)$, l'ensemble des **arêtes** A n'est pas constitué de *couples* mais de *paires* de sommets — une paire étant non ordonnée contrairement à un couple. Par convention, on représente l'arête entre les sommets u et v non par la notation $\{u, v\}$ mais, indifféremment, par les notations (u, v) ou (v, u). Dans un graphe non orienté les boucles sont interdites et chaque arête est donc constituée de deux sommets distincts. La figure 8.2 est une représentation graphique du graphe non orienté $G = (S, A)$ avec l'ensemble de sommets $S = \{1, 2, 3, 4, 5, 6\}$ et l'ensemble d'arcs $A = \{(1, 2), (2, 5), (5, 1), (6, 3)\}$.

Si (u, v) est un arc d'un *graphe orienté* $G = (S, A)$, on dit que (u, v) **part** du sommet u et **arrive** au sommet v. Si (u, v) est une arête d'un graphe *non orienté* $G = (S, A)$, on dit que l'arête (u, v) est **incidente** aux sommets u et v.

Dans un *graphe non orienté*, le **degré** d'un sommet est le nombre d'arêtes qui lui sont incidentes. Si un sommet est de degré 0, comme le sommet 4 de la figure 8.2, il est dit **isolé**. Dans un *graphe orienté*, le **degré sortant** d'un sommet est le nombre d'arcs qui en partent, le **degré (r)entrant** est le nombre d'arcs qui y arrivent et le **degré** est la somme du degré entrant et du degré sortant.

Dans un *graphe orienté* $G = (S, A)$, un **chemin** de **longueur** k d'un sommet u à un sommet v est une séquence $(u_0, u_1, ..., u_k)$ de sommets telle que $u = u_0$, $v = u_k$ et $(u_{i-1}, u_i) \in A$ pour tout i dans $\{1, ..., k\}$. Un chemin est **élémentaire** si ces sommets sont tous distincts. Dans la figure 8.1, le chemin

$(1, 2, 5, 4)$ est élémentaire et de longueur 3, mais le chemin $(2, 5, 4, 5)$ n'est pas élémentaire. Un **sous-chemin** p' d'un chemin $p = (u_0, u_1, ..., u_k)$ est une sous-séquence contiguë de ses sommets. Autrement dit, il existe i et j, $0 \leq i \leq j \leq k$, tels que $p' = (u_i, u_{i+1}, ..., u_j)$. On définit dans les *graphes non orientés* la notion correspondante de **chaîne**.

Dans un *graphe orienté* $G = (S, A)$, un chemin $(u_0, u_1, ..., u_k)$ forme un **circuit** si $u_0 = u_k$ et si le chemin contient au moins un arc. Ce circuit est **élémentaire** si les sommets u_1, ..., u_k sont distincts. Une boucle est un circuit de longueur 1. Dans un *graphe non orienté* $G = (S, A)$, une chaîne $(u_0, u_1, ..., u_k)$ forme un **cycle** si $k \geq 3$ et si $u_0 = u_k$. Ce cycle est **élémentaire** si les sommets u_1, ..., u_k sont distincts. Un graphe sans cycle est dit **acyclique**.

Un *graphe non orienté* est **connexe** si chaque paire de sommets est reliée par une chaîne. Les **composantes connexes** d'un graphe sont les classes d'équivalence de sommets induites par la relation « est accessible à partir de ». Le graphe de la figure 8.2 contient trois composantes connexes : $\{1, 2, 5\}$, $\{3, 6\}$ et $\{4\}$.

Un *graphe orienté* est **fortement connexe** si chaque sommet est accessible à partir de n'importe quel autre. Les **composantes fortement connexes** d'un graphe sont les classes d'équivalence de sommets induites par la relation « sont accessibles l'un à partir de l'autre ». Le graphe de la figure 8.1 contient trois composantes connexes : $\{1, 2, 4, 5\}$, $\{3\}$ et $\{6\}$.

On dit qu'un graphe $G' = (S', A')$ est un **sous-graphe** de $G = (S, A)$ si $S' \subset S$ et si $A' \subset A$.

8.2 Arbres

Un graphe non orienté acyclique est une **forêt** et un graphe non orienté connexe acyclique est un **arbre**. La figure 8.3 présente un graphe qui n'est ni un arbre ni une forêt car contenant un cycle ; la figure 8.4 présente un graphe qui est une forêt mais pas un arbre, puisque n'étant pas connexe ; la figure 8.5 présente un arbre.

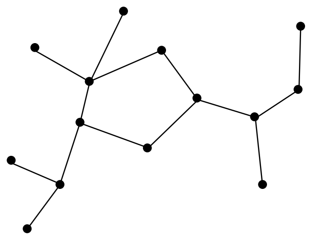


FIG. 8.3 – Exemple de graphe contenant un cycle.

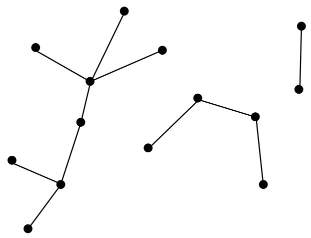


FIG. 8.4 – Exemple de forêt.

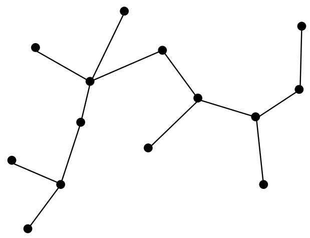


FIG. 8.5 – Exemple d'arbre.

Théorème 7 (Propriétés des arbres). *Soit $G = (S, A)$ un graphe non orienté. Les affirmations suivantes sont équivalentes.*

1. *G est un arbre.*
2. *Deux sommets quelconques de G sont reliés par un unique chemin élémentaire.*
3. *G est connexe, mais si une arête quelconque est ôtée de A, le graphe résultant n'est plus connexe.*
4. *G est connexe et $|A| = |S| - 1$.*
5. *G est acyclique et $|A| = |S| - 1$.*
6. *G est acyclique, mais si une arête quelconque est ajoutée à A, le graphe résultant contient un cycle.*

Pour la démonstration de ce théorème, voir [2, pp. 89–91].

Un **arbre enraciné** est un arbre dans lequel l'un des sommets se distingue des autres. On appelle ce sommet la **racine**. Ce sommet particulier impose en réalité un sens de parcours de l'arbre et l'arbre se retrouve orienté par l'utilisation qui en est faite... Dans la suite de ce cours, et sauf avis contraire, tous les arbres que nous manipulerons seront des arbres enracinés et nous omettrons de le préciser. En outre, on appellera souvent **nœuds** les sommets des arbres (enracinés). La figure 8.6 présente deux arbres qui ne diffèrent que s'ils sont considérés comme des arbres enracinés.

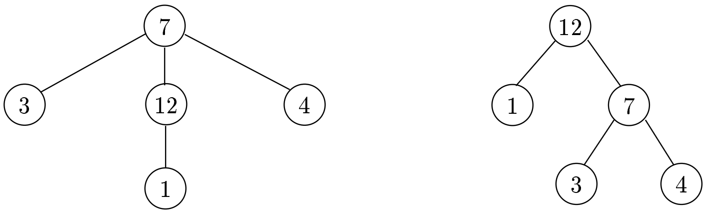


FIG. 8.6 – Exemple d'arbres qui ne diffèrent que s'ils sont enracinés.

Soit x un nœud d'un arbre T de racine r. Un nœud quelconque y sur l'unique chemin allant de r à x est appelé **ancêtre** de x. Si T contient l'arête (y, x) alors y est le **père** de x et x est le **fils** de y. La racine est le seul nœud qui n'ait pas de père. Un nœud sans fils est un nœud externe ou une **feuille**. Un nœud qui n'est pas une feuille est un **nœud interne**. Si y est un ancêtre de x, alors x est un **descendant** de y.

Le **sous-arbre de racine** x est l'arbre composé des descendants de x, enraciné en x. Par exemple, dans le premier arbre de la figure 8.7, le sous-arbre de racine 8 contient les nœuds 8, 6, 5 et 9.

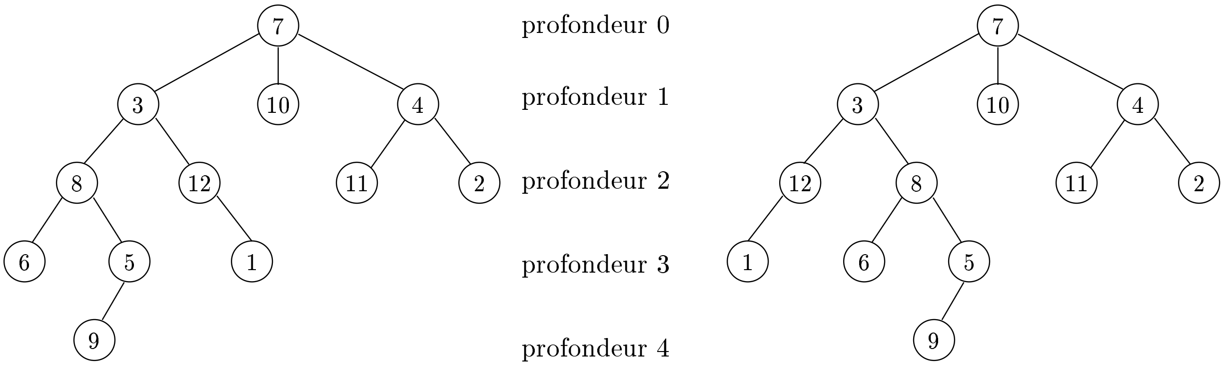


FIG. 8.7 – Exemple d'arbres (enracinés) qui ne diffèrent que s'ils sont ordonnés.

Le nombre de fils du nœud x est appelé le **degré** de x. Donc, suivant qu'un arbre (enraciné) est vu comme un arbre (enraciné) ou un graphe, le degré de ses sommets n'a pas la même valeur ! La longueur du chemin entre la racine r et le nœud x est la **profondeur** de x. La plus grande profondeur que puisse avoir un nœud quelconque de l'arbre T est la **hauteur** de T. Les deux arbres présentés figure 8.7 sont de hauteur 4.

Un **arbre ordonné** est un arbre enraciné dans lequel les fils de chaque nœud sont ordonnés entre eux. Les deux arbres de la figure 8.7 sont différents si on les regarde comme des arbres ordonnés... mais ils sont identiques si on les regarde comme de simples arbres (enracinés).

Les arbres binaires se décrivent plus aisément de manière récursive. Un **arbre binaire** T est une structure définie sur un ensemble fini de nœuds et qui :

- ne contient aucun nœud, ou
- est formé de trois ensembles disjoints de nœuds : une racine, un arbre binaire appelé son **sous-arbre gauche** et un arbre binaire appelé son **sous-arbre droit**.

Un arbre binaire est plus qu'un arbre ordonné dont chaque nœud serait de degré au plus deux : dans un arbre binaire, si un nœud n'a qu'un seul fils, la position de ce fils — qu'il soit **fils gauche** ou **fils droit** — est importante. La figure 8.8 présente deux arbres ordonnés qui ne sont différents que quand ils sont vus comme des arbres binaires.

Dans un **arbre binaire complet** chaque nœud est soit une feuille, soit de degré deux — aucun nœud n'est donc de degré un.

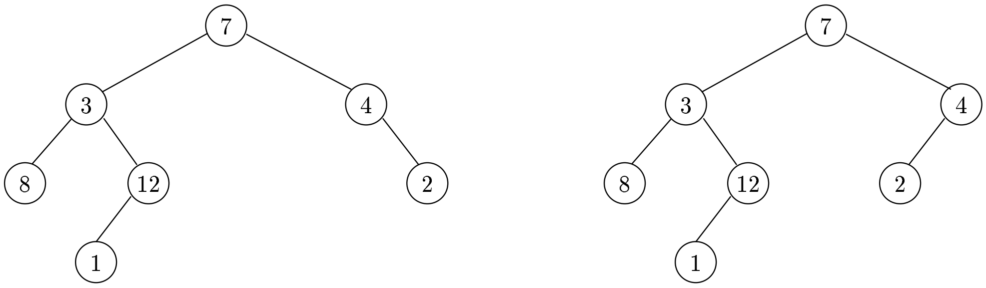


FIG. 8.8 – Exemple d'arbres ordonnés qui ne diffèrent que quand ils sont vus comme des arbres binaires.

Un arbre **k-aire** est une généralisation de la notion d'arbre binaire où chaque nœud est de degré au plus k et non plus simplement de degré au plus 2.

8.3 Parcours

8.3.1 Parcours des arbres

Nous ne considérons ici que des **arbres ordonnés**.

Parcours en profondeur

Dans un *parcours en profondeur d'abord*, on descend le plus profondément possible dans l'arbre puis, une fois qu'une feuille a été atteinte, on remonte pour explorer les autres branches en commençant par la branche « la plus basse » parmi celles non encore parcourues ; les fils d'un nœud sont bien évidemment parcourus suivant l'ordre sur l'arbre.

PP(A)
  si A n'est pas réduit à une feuille **faire**
    pour tous les fils u de `racine`(A) **faire dans l'ordre** PP(u)

FIG. 8.9 – Algorithme de parcours en profondeur d'un arbre.

Les parcours permettent d'effectuer tout un ensemble de traitement sur les arbres. La figure 8.10 présente trois algorithmes qui affichent les valeurs contenues dans les nœuds d'un arbre binaire, suivant des parcours en profondeur préfixe, infixe et postfixe, respectivement.

PRÉFIXE(A)
  si $A \neq$ NIL **faire**
    affiche `racine`(A)
    PRÉFIXE(FILS-GAUCHE(A))
    PRÉFIXE(FILS-DROIT(A))

INFIXE(A)
  si $A \neq$ NIL **faire**
    INFIXE(FILS-GAUCHE(A))
    affiche `racine`(A)
    INFIXE(FILS-DROIT(A))

POSTFIXE(A)
  si $A \neq$ NIL **faire**
    POSTFIXE(FILS-GAUCHE(A))
    POSTFIXE(FILS-DROIT(A))
    affiche `racine`(A)

FIG. 8.10 – Parcours préfixe, infixe et postfixe d'un arbre.

Parcours en largeur

Dans un *parcours en largeur d'abord*, tous les nœuds à une profondeur i doivent avoir été visités avant que le premier nœud à la profondeur $i + 1$ ne soit visité. Un tel parcours nécessite que l'on se souvienne de l'ensemble des branches qu'il reste à visiter. Pour ce faire, on utilise une *file* (ici notée F).

PL(A)
  $F \leftarrow \{\texttt{racine}(A)\}$
  tant que $F \neq \emptyset$ **faire**
    $u \leftarrow$ Suppression(F)
    pour tous les fils v de u **faire dans l'ordre** Insertion(F, v)

Fig. 8.11 – Algorithme de parcours en largeur d'un arbre.

8.3.2 Parcours des graphes

Le parcours des graphes se révèle être un peu plus compliqué que celui des arbres. En effet, les graphes peuvent contenir des cycles et nous voulons éviter de parcourir indéfiniment ces cycles ! Pour éviter cet écueil on colorie les sommets des graphes : initialement les sommets sont tous blancs ; lorsqu'il est rencontré pour la première fois un sommet est peint en gris ; lorsque tous ses successeurs dans l'ordre de parcours ont été visités, un sommet est repeint en noir.

Parcours en profondeur

PP(G)
  pour chaque sommet u de G **faire** *couleur*[u] $\leftarrow$ Blanc
  pour chaque sommet u de G **faire si** *couleur*[u] = Blanc **alors** Visiter-PP(G, u, *couleur*)

Visiter-PP(G, s, *couleur*)
  couleur[s] $\leftarrow$ Gris
  pour chaque voisin v de s **faire**
      si *couleur*[v] = Blanc **alors** Visiter-PP(G, v, *couleur*)
  couleur[s] $\leftarrow$ Noir

Fig. 8.12 – Algorithme de parcours en profondeur d'un graphe.

Parcours en largeur

PL(G, s)
  couleur[s] $\leftarrow$ Gris
  pour chaque sommet u de G, $u \neq s$ **faire** *couleur*[u] $\leftarrow$ Blanc
  $F \leftarrow \{s\}$
  tant que $F \neq \emptyset$ **faire**
    $u \leftarrow$ Suppression(F)
    pour chaque voisin v de u **faire**
      si *couleur*[v] = Blanc
        alors *couleur*[v] $\leftarrow$ Gris
            Insertion(F, v)
    couleur[u] $\leftarrow$ Noir

Fig. 8.13 – Algorithme de parcours en largeur d'un graphe.

Chapitre 9

Arbres de recherche et arbres de recherche équilibrés

9.1 Arbres binaires de recherche

9.1.1 Définition

Définition 12 (Arbre binaire de recherche). *Un arbre binaire de recherche est un arbre binaire vérifiant la propriété suivante : soient x et y deux nœuds de l'arbre, si y est un nœud du sous-arbre gauche de x, alors $clef(y) \leq clef(x)$, si y est un nœud du sous-arbre droit de x, alors $clef(y) \geq clef(x)$.*

La figure 9.1 présente deux exemples d'arbres binaires de recherche. Bien que différents, ces deux arbres contiennent exactement les mêmes valeurs.

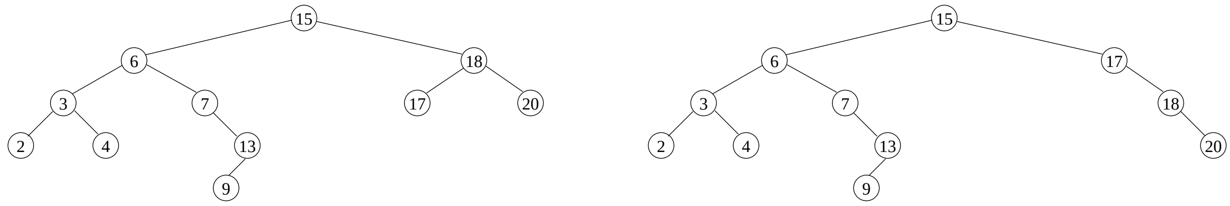


FIG. 9.1 – Deux arbres binaires de recherche contenant les mêmes valeurs.

9.1.2 Recherches

Recherche d'un élément

ARBRE-RECHERCHER(x, k)
  si x = NIL ou k = $clé(x)$ **alors renvoyer** x
  si $k < clé(x)$
    alors renvoyer ARBRE-RECHERCHER($gauche(x)$, k)
    sinon renvoyer ARBRE-RECHERCHER($droit(x)$, k)

Minimum et maximum

ARBRE-MINIMUM(x)
  tant que $gauche(x) \neq$ NIL **faire** $x \leftarrow gauche(x)$
  renvoyer x

ARBRE-MAXIMUM(x)
  tant que $droit(x) \neq$ NIL **faire** $x \leftarrow droit(x)$
  renvoyer x

Successeur et prédécesseur

Si toutes les clés dans l'arbre sont distinctes, le successeur d'un nœud x est le nœud contenant la plus petite clé supérieure à x.

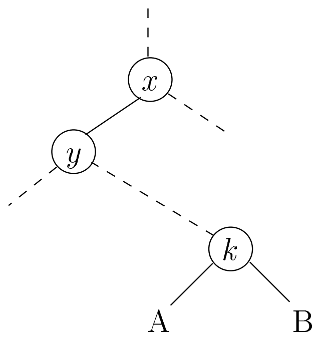


Fig. 9.2 – Localisation du successeur.

Considérons le fragment d'arbre présenté figure 9.2. De par la propriété des arbres binaires de recherche :

1. Le sous-arbre `A` ne contient que des clés inférieures à k et ne peut contenir le successeur de k.
2. Le sous-arbre `B` ne contient que des clés supérieures à k et peut contenir le successeur de k s'il n'est pas vide.
3. y désigne le plus proche ancêtre de k qui soit le fils gauche de son père ($y = k$ si k est fils gauche de son père). Tous les ancêtres de k jusqu'à y sont inférieurs ou égaux à k et leurs sous-arbres gauches ne contiennent que des valeurs inférieures à k.

   x est le père de y. Sa valeur est supérieure à toutes celles contenues dans son sous-arbre gauche (de racine y) et donc à k et à celles de B. Toutes les valeurs de son sous-arbre droit sont supérieures à x.

En résumé : si `B` est non vide, son minimum est le successeur de k, sinon le successeur de k est le premier ancêtre de k dont le fils gauche est aussi ancêtre de k.

Arbre-Successeur(x)
  si $droit(x) \neq$ Nil **alors renvoyer** Arbre-Minimum($droit(x)$)
  $y \leftarrow père(x)$
  tant que $y \neq$ Nil et $x = droit(y)$ **faire**
    $x \leftarrow y$
    $y \leftarrow père(x)$
  renvoyer y

9.1.3 Insertion d'un élément

L'élément à ajouter est inséré là où on l'aurait trouvé s'il avait été présent dans l'arbre. L'algorithme d'insertion, présenté figure 9.3, recherche donc l'élément dans l'arbre et, quand il aboutit à la conclusion que l'élément n'appartient pas à l'arbre (l'algorithme aboutit sur Nil), il insère l'élément comme fils du dernier nœud visité.

9.1.4 Suppression d'un élément

La figure 9.4 présente les différents cas de figure que doit traiter l'algorithme de suppression d'un élément dans un arbre binaire de recherche. L'algorithme, présenté figure 9.5, tient en plus compte des conditions aux limites (changement de racine).

ARBRE-INSERTION(T, z)
  $x \leftarrow racine(T)$
  père_de_x $\leftarrow$ NIL
  tant que $x \neq$ NIL **faire**
    père_de_x $\leftarrow x$
    si $clef(z) < clef(x)$
      alors $x \leftarrow gauche(x)$
      sinon $x \leftarrow droit(x)$
  père$(z) \leftarrow$ *père_de_x*
  si *père_de_x* $=$ NIL
    alors $racine(T) \leftarrow z$
    sinon si $clef(z) < clef(x)$
      alors *gauche*(*père_de_x*) $\leftarrow z$
      sinon *droit*(*père_de_x*) $\leftarrow z$

FIG. 9.3 – Algorithme d'insertion dans un arbre binaire de recherche.

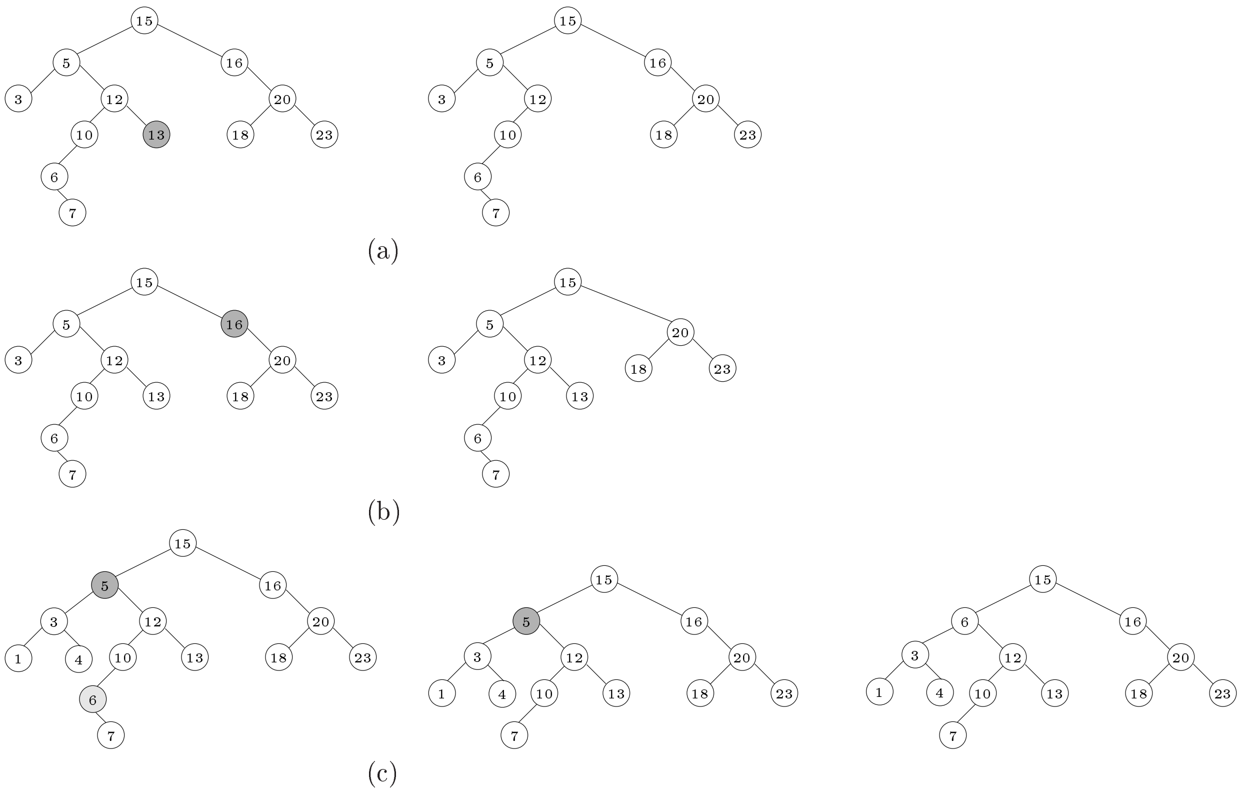


FIG. 9.4 – Les différents cas de figure possibles lors de la suppression d'un nœud d'un arbre binaire de recherche (les nœuds à supprimer sont en gris foncé) : a) le nœud à supprimer n'a pas de fils et on l'élimine simplement de l'arbre ; b) le nœud à supprimer a un unique fils, on détache le nœud et on relie directement son père et son fils ; c) le nœud à supprimer a deux fils, on le remplace par son successeur (qui, dans ce cas, est toujours le minimum de son fils droit, nœud qui est ici légèrement grisé).

```
ARBRE-SUPPRESSION(T, x)
  si gauche(x) = NIL et droit(x) = NIL
    alors
      si père(x) = NIL
        alors racine(T)← NIL
        sinon si x = gauche(père(x))
          alors gauche(père(x)) ← NIL
          sinon droit(père(x)) ← NIL
    sinon si gauche(x) = NIL ou droit(x) = NIL
      alors
        si gauche(x) ≠ NIL
          alors filsde_x ← gauche(x)
          sinon filsde_x ← droit(x)
        père(filsde_x)←père(x)
        si père(x) = NIL
          alors racine(T)← filsde_x
          sinon si gauche(père(x)) = x
            alors gauche(père(x)) ← filsde_x
            sinon droit(père(x)) ← filsde_x
      sinon
        min ← ARBRE-MINIMUM(droit(x))
        clé(y) ← clé(min)
        ARBRE-SUPPRESSION(T, min)
  renvoyer racine(T)
```

FIG. 9.5 – Suppression d'un élément dans un arbre binaire de recherche.

9.1.5 Complexité

Si h est la hauteur de l'arbre, on peut aisément montrer que tous les algorithmes précédents ont une complexité en $O(h)$. Malheureusement, un arbre binaire quelconque à n nœuds a une hauteur comprise, en ordre de grandeur, entre $\log_2 n$ et n. Pour éviter les cas les plus pathologiques, on s'intéresse à des arbres de recherches *équilibrés*.

9.2 Arbres rouge et noir

Les arbres rouge et noir sont *un* des schémas d'arbres de recherche dits *équilibrés*.

9.2.1 Définition

Définition 13 (Arbre rouge et noir). *Un arbre binaire de recherche est un arbre rouge et noir s'il satisfait les propriétés suivantes :*

1. *Chaque nœud est soit rouge, soit noir.*
2. *Chaque feuille (*NIL*) est noire.*
3. *Si un nœud est rouge, alors ses deux fils sont noirs.*
4. *Tous les chemins descendants reliant un nœud donné à une feuille (du sous-arbre dont il est la racine) contiennent le même nombre de nœuds noirs.*

   On appelle hauteur noire *d'un nœud x le nombre de nœuds noirs sur un chemin descendant de x à une feuille.*

La figure 9.6 présente un exemple d'arbre rouge et noir.

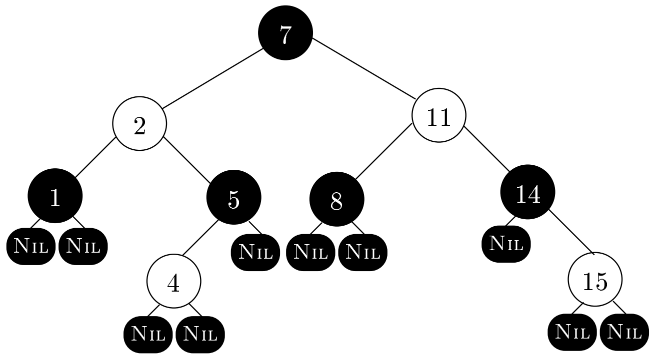


FIG. 9.6 – Exemple d'arbre rouge et noir. Les hauteurs noires sont indiquées à côté des nœuds.

9.2.2 Rotations

La figure 9.7 présente les transformations d'arbres binaires appelées « rotations » pour d'évidentes raisons. La figure 9.8 présente l'algorithme réalisant la rotation gauche (la rotation droite étant bien évidemment symétrique). Les rotations préservent la propriété des arbres de recherche mais pas la propriété des arbres rouge et noir.

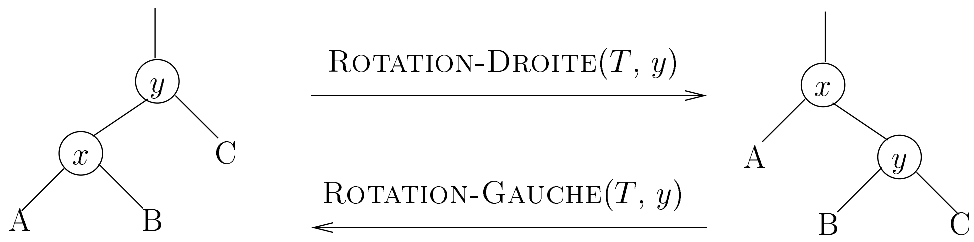


FIG. 9.7 – Rotations sur un arbre binaire de recherche.

9.2.3 Insertion

Pour réaliser l'insertion dans un arbre rouge et noir, on peut essayer d'insérer le nouveau nœud comme si l'arbre n'était qu'un vulgaire arbre de recherche. Pour ce qui est du choix de la couleur du nœud inséré, le noir est *a priori* à proscrire puisqu'il provoquerait *systématiquement* une violation de la propriété 4. On choisit donc de colorier le nouveau nœud en rouge ce qui provoquera *parfois* des violations de la propriété 3 : un père et un fils étant tous les deux rouges. Nous étudions les différents cas de violations. La figure 9.9 présente une première série de configurations pathologiques qui sont transformées en d'autres configurations. La figure 9.10 présente une deuxième série de configurations pathologiques, ces configurations là pouvant être résolues par l'application d'une ou de deux rotations suivant les cas. La figure 9.11 présente l'algorithme d'insertion d'un élément dans un arbre rouge et noir obtenu suite à cette étude de cas, et la figure 9.12 présente de nouveau les différents cas pathologiques sur un exemple.

9.2.4 Suppression

Pour supprimer un élément dans un arbre rouge et noir, on commence par appliquer l'algorithme de suppression pour les arbres de recherche. Si l'élément supprimé était de couleur rouge, aucune des propriétés des arbres rouge et noir n'est violée. Par contre, si le nœud supprimé était noir la propriété 4 (tous les chemins descendants d'un nœud à une feuille contiennent le même nombre de nœuds noirs) peut être violée. Il nous faut donc *rajouter* un noir sur tous les chemins perturbés. Pour ce faire, on rajoute un noir à l'unique

Rotation-Gauche(T, x)
  $y \leftarrow droit(x)$
  $droit(x) \leftarrow gauche(y)$
  si $gauche(y) \neq$ Nil **alors** $père(gauche(y)) \leftarrow x$
  $père(y) \leftarrow père(x)$
  si $père(y) =$ Nil
    alors $racine(T) \leftarrow y$
    sinon si $x = gauche(père(x))$
      alors $gauche(père(x)) \leftarrow y$
      sinon $droit(père(x)) \leftarrow y$
  $gauche(y) \leftarrow x$
  $père(x) \leftarrow y$

Fig. 9.8 – Algorithme de rotation gauche pour un arbre binaire.

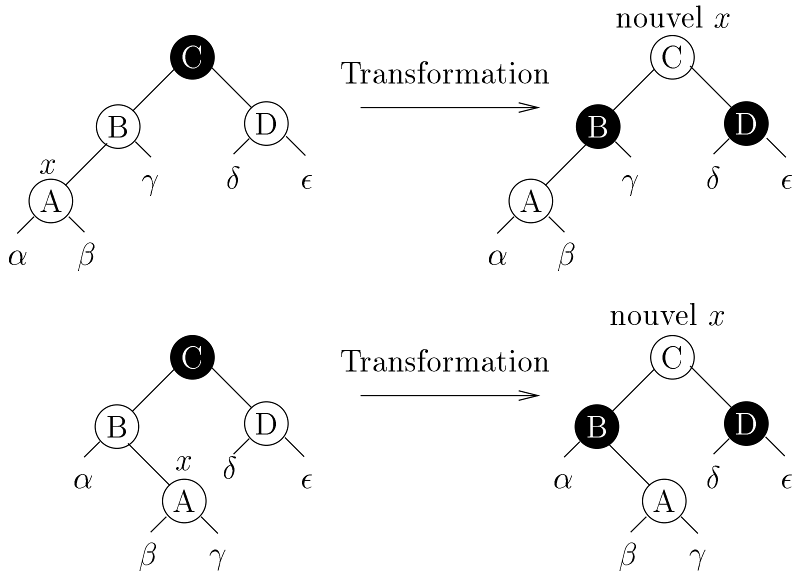


Fig. 9.9 – Première série de cas pathologiques rencontrés par l'algorithme d'insertion. Les nœuds « noirs » sont à fond noir, et les « rouges » sont à fond blanc. Ici, seule la propriété 3 est violée, x étant le nœud source du problème. Par conséquent, les sous-arbres α, β, γ, δ et ϵ sont tous de racine noire et ont tous la même hauteur noire. L'algorithme fait « descendre » la couleur noire du grand-père de x sur le père et l'oncle de x, ce qui revient à faire remonter le problème dans l'arbre, seule la propriété 3 pouvant être violée par cette transformation. Les cas non traités sont symétriques de ceux présentés.

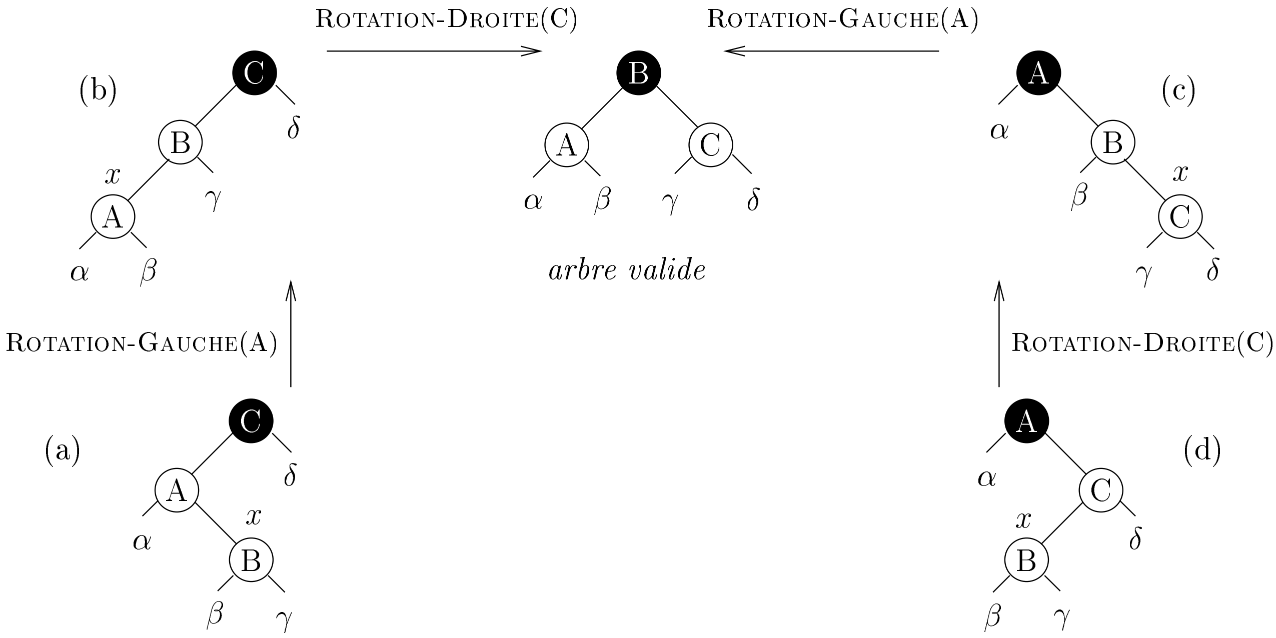


Fig. 9.10 – Deuxième série de cas pathologiques rencontrés par l'algorithme d'insertion. Les nœuds « noirs » sont à fond noir, et les « rouges » sont à fond blanc. Ici, seule la propriété 3 est violée, x est le nœud source du problème et nous ne sommes pas dans un des cas traités par la figure 9.9. Par conséquent, les sous-arbres α, β, γ et δ sont tous de racine noire et ont tous la même hauteur noire (et les transformations d'un cas à l'autre préservent cette propriété). Ici on conjugue des rotations et des changements de couleur des nœuds.

fils du nœud supprimé (pour l'unicité du fils, voir l'algorithme de suppression figure 9.5). Si ce fils était rouge, l'arbre obtenu est un arbre rouge et noir. Si ce fils était déjà noir, on a deux « noirs » empilés sur un même nœud et il nous faut les répartir. La figure 9.13 présente les différents cas de figure possibles et les méthodes de résolutions associées. Si le nœud supprimé n'avait pas de fils, on rajoute un « noir » à la feuille Nil correspondante de son père. Pour pouvoir réaliser cette manipulation, on utilise une sentinelle : un nœud spécial valant Nil et qui permet de ne pas traiter à part les feuilles Nil. La figure 9.14 présente l'algorithme de suppression avec utilisation de sentinelles et appel de l'algorithme de correction — celui qui répartit les « noirs » surnuméraires — lui-même présenté figure 9.15.

9.2.5 Complexité

Théorème 8 (Hauteur des arbres rouge et noir). *Un arbre rouge et noir contenant n nœuds internes a une hauteur au plus égale à* $2\log(n+1)$.

On peut montrer par induction que le sous-arbre (d'un arbre rouge et noir) enraciné en un nœud x quelconque contient au moins $2^{hn(x)}$ nœuds internes, où $hn(x)$ est la hauteur noire de x. Sachant que la hauteur est toujours inférieure au double de la hauteur noire on en déduit la borne du théorème 8.

Ce théorème montre bien que les arbres rouge et noir sont relativement *équilibrés* : la hauteur d'un arbre rouge et noir est au pire du double de celle d'un arbre binaire parfaitement équilibré.

Toutes les opérations sur les arbres rouge et noir sont de coût $O(h)$, c'est-à-dire $O(\log n)$, ce qui justifie leur utilisation par rapport aux arbres binaires de recherche classiques.

```
RougeNoir-Insertion(T, x)
   Arbre-Insertion(T, x)
   couleur(x) ← Rouge
   tant que x ≠ racine(T) et couleur(père(x)) = Rouge faire
      si père(x) = gauche(grand-père(x))
         alors
            y ← droit(grand-père(x))
            si couleur(y) = Rouge
               alors                                    ▷ cas de la figure 9.9
                  couleur(père(x))←Noir
                  couleur(y)←Noir
                  couleur(grand-père(x))←Rouge
                  x ←grand-père(x)
               sinon                                    ▷ cas de la figure 9.10
                  si x =droit(père(x))
                     alors                              ▷ cas a) de la figure 9.10
                        x ←père(x)
                        Rotation-Gauche(T, x)
                                                        ▷ cas b) de la figure 9.10
                  couleur(père(x))←Noir
                  couleur(grand-père(x))←Rouge
                  Rotation-Droite(T, grand-père(x))
         sinon (même chose que précédemment en échangeant droit et gauche)
   couleur(racine(T))←Noir
```

Fig. 9.11 – Algorithme d'insertion dans un arbre rouge et noir.

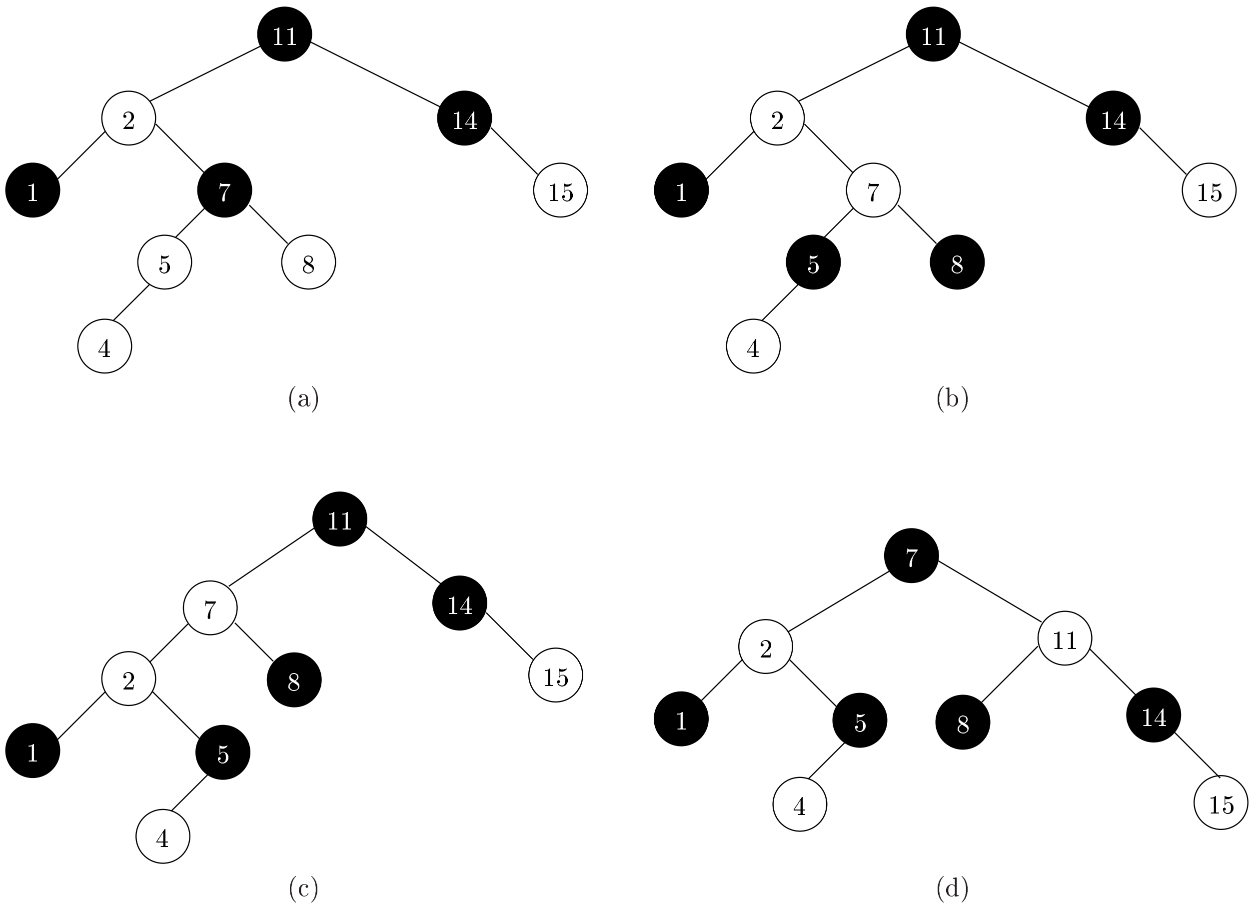


FIG. 9.12 – Exemple d'insertion dans un arbre rouge et noir faisant apparaître les différents cas de figure pathologiques pouvant apparaître après l'insertion. Les nœuds « noirs » sont à fond noir, et les « rouges » sont à fond blanc. À chaque fois le nœud x et son père sont tous les deux rouges : a) l'oncle de x est également rouge, nous nous trouvons donc dans le cas de la figure 9.9, le père et l'oncle de x sont donc repeints en rouge et son grand-père en noir, le problème est alors remonté de deux crans ; b) l'oncle de x est noir et x est le fils gauche de son père, nous nous trouvons donc dans le cas a) de la figure 9.10, nous appliquons alors une rotation gauche sur le père de x et nous aboutissons à la situation du cas suivant ; c) l'oncle de x est noir, x est le fils droit de son père et son père est le fils gauche de son grand-père, nous nous trouvons donc dans le cas b) de la figure 9.10, nous appliquons alors une rotation droite sur le grand-père de x, l'ancien grand-père de x est alors repeint en noir et le père de x en rouge, ce qui nous donne un arbre rouge et noir valide.

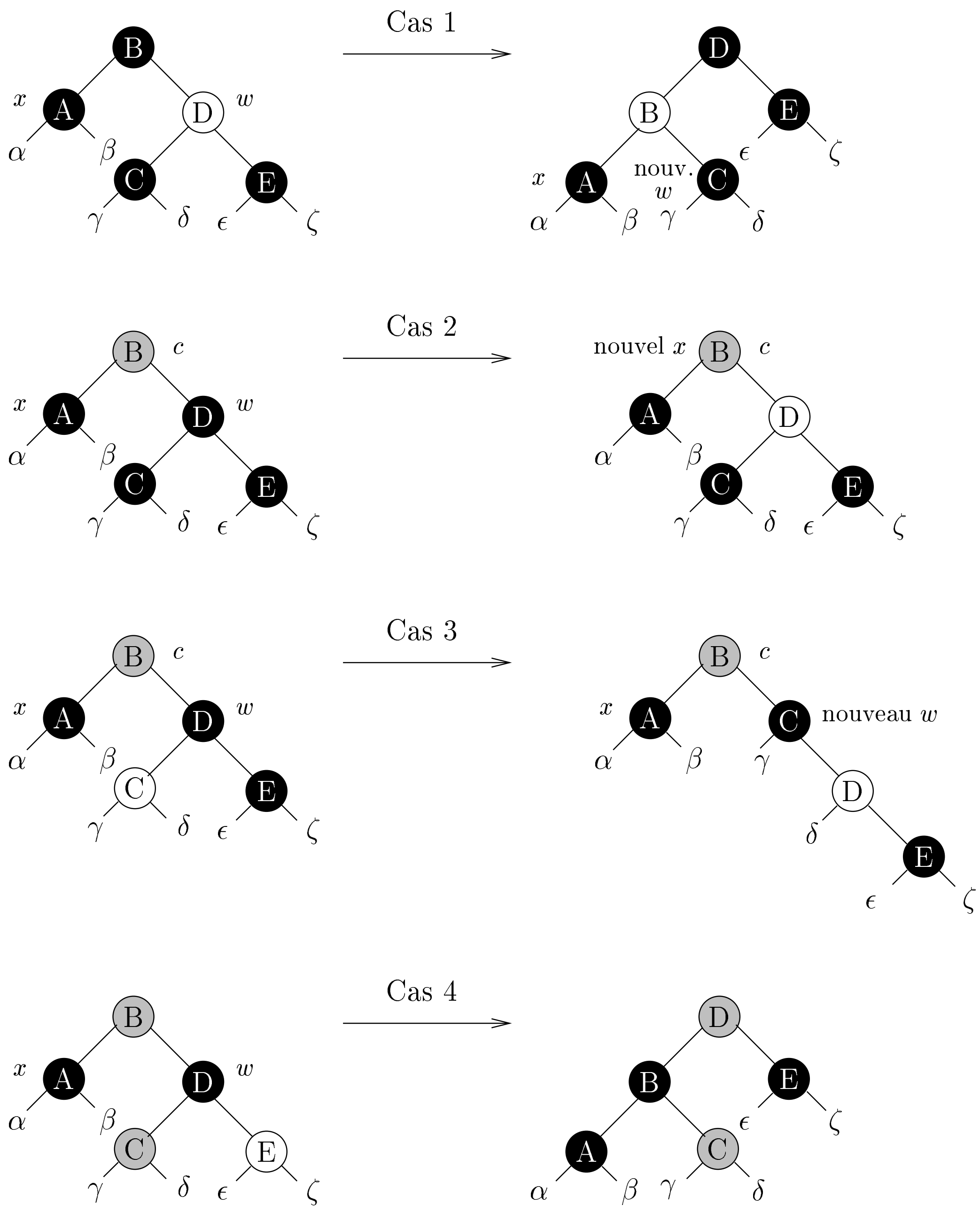


Fig. 9.13 – Configurations pathologiques pour la suppression dans un arbre rouge et noir. Les nœuds à fond noir sont des nœuds « noirs », ceux à fond blanc sont « rouges » et ceux à fond grisé sont soit « noirs » soit « rouges ». α, β, γ, δ, ϵ et ζ désignent des arbres quelconques. Le nœud x comporte un noir supplémentaire. **(1)** Ce cas est transformé en cas 2, 3 ou 4. **(2)** Le noir supplémentaire est remonté sur le père de x, l'oncle de x étant repeint en rouge ; si le père de x était rouge l'arbre est de nouveau valide, sinon on rapplique l'algorithme de correction cette fois-ci sur le père de x. **(3)** Ce cas est transformé en cas 4. **(4)** Le noir supplémentaire est éliminé par rotation gauche sur le père de x et recoloriage du père et de l'oncle de x.

```
ARBRE-RN-SUPPRESSION(T, x)
  si gauche(x) = NIL(T) et droit(x) = NIL(T)
    alors
      si père(x) = NIL(T)
        alors racine(T) ← NIL(T)
        sinon
          si x = gauche(père(x))
            alors gauche(père(x)) ← NIL(T)
            sinon droit(père(x)) ← NIL(T)
          si couleur(x) = NOIR
            alors
              père(NIL(T)) ← père(x)
              RN-CORRECTION(T, x)
    sinon si gauche(x) = NIL(T) ou droit(x) = NIL(T)
      alors
        si gauche(x) ≠ NIL(T)
          alors filsde_x ← gauche(x)
          sinon filsde_x ← droit(x)
        père(filsde_x) ← père(x)
        si père(x) = NIL(T)
          alors racine(T) ← filsde_x
          sinon si gauche(père(x)) = x
            alors gauche(père(x)) ← filsde_x
            sinon droit(père(x)) ← filsde_x
        si couleur(x) = NOIR alors
          RN-CORRECTION(T, filsde_x)
      sinon
        min ← ARBRE-MINIMUM(droit(x))
        clé(y) ← clé(min)
        ARBRE-RN-SUPPRESSION(T, min)
  renvoyer racine(T)
```

FIG. 9.14 – Suppression d'un élément dans un arbre rouge et noir.

```
RN-Correction(T, x)
  tant que x ≠ racine(T) et couleur(x) = Noir faire
    si x = gauche(père(x))
      alors
        w ← droit(père(x))
        si couleur(w) = Rouge
          alors                                              ▷ cas 1 de la figure 9.13
            couleur(w) ← Noir
            couleur(père(w)) ← Rouge
            Rotation-Gauche(T, père(x))
            w ← droit(père(x))
        si couleur(gauche(w)) = Noir et couleur(droit(w)) = Noir
          alors                                              ▷ cas 2 de la figure 9.13
            couleur(w) ← Rouge
            x ← père(x)
          sinon
            si couleur(droit(w)) = Noir
              alors                                          ▷ cas 3 de la figure 9.13
                couleur(gauche(w)) ← Noir
                couleur(w) ← Rouge
                Rotation-Droite(T, w)
                w ← droit(père(x))
                                                             ▷ cas 4 de la figure 9.13
            couleur(w) ← couleur(père(x))
            couleur(père(x)) ← Noir
            couleur(droit(w))) ← Noir
            Rotation-Gauche(T, père(x))
            x ← racine(T)
      sinon (même chose que précédemment en échangeant droit et gauche)
  couleur(x) ← Noir
```

Fig. 9.15 – Correction d'un arbre rouge et noir après suppression d'un élément.

Chapitre 10

Plus courts chemins

Dans un problème de plus courts chemins, on possède en entrée un graphe orienté pondéré $G = (S, A)$ de fonction de pondération $w : A \to \mathbb{R}$. Le poids du chemin $p = \langle v_0, v_1, ..., v_k \rangle$ est la somme des poids de ses arcs :

$$w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i).$$

Le poids $\delta(u, v)$ d'un plus court chemin d'un sommet u à un sommet v est bien évidemment le minimum des poids des chemins de u à v (si celui-ci est défini, ce qui peut ne pas être le cas si le graphe contient un circuit de poids strictement négatif). Un **plus court chemin** d'un sommet u à un sommet v est alors un chemin de u à v de poids $\delta(u, v)$.

10.1 Plus courts chemins à origine unique

On souhaite dans cette section trouver les plus courts chemins depuis un sommet origine s et vers n'importe quel autre sommet.

Dans la suite, $\pi[u]$ désignera le prédécesseur de u dans l'estimation du plus court chemin de s à u et $d[u]$ désignera la longueur de ce chemin.

10.1.1 Algorithme de Dijkstra

L'algorithme de Dijkstra résout le problème de la recherche d'un plus court chemin à origine unique pour un graphe orienté pondéré $G = (S, A)$ dans le cas où **tous les arcs ont un poids positif ou nul** : $\forall (u, v) \in A, w(u, v) \geq 0$.

L'algorithme de Dijkstra maintient à jour un ensemble E des sommets de G dont le plus court chemin à partir de l'origine s est connu et calculé. À chaque itération, l'algorithme choisit parmi les sommets de $S \setminus E$ — c'est-à-dire parmi les sommets dont le plus court chemin à partir de l'origine n'est pas connu — le sommet u dont l'estimation de plus court chemin est minimale. Cet algorithme est donc **glouton**. Une fois un sommet u choisi, l'algorithme met à jour, si besoin est, les estimations des plus courts chemins de ses successeurs (les sommets qui peuvent être atteint directement à partir de u).

Source-Unique-Initialization initialise les valeurs de $\pi[u]$ et de $d[u]$ pour chaque sommet u : initialement, il n'y a pas de chemin connu de s à u (si $u \neq s$) et u est estimé être à une distance infinie de s.

Relâcher(u, v, w) compare le plus court chemin de s à v connu avec une nouvelle proposition (chemin estimé de s à u puis arc de u à v), et met les différentes données à jour si besoin est.

L'algorithme est présenté figure 10.1.

Cet algorithme fourni effectivement les plus courts chemins. L'algorithme glouton fonctionne uniquement parce que les poids sont positifs. On montre la correction de l'algorithme par récurrence.

Source-Unique-Initialization$[G, s]$
  pour chaque sommet v de G **faire**
    $d[v] \leftarrow +\infty$
    $\pi[v] \leftarrow$ Nil
  $d[s] \leftarrow 0$

Relâcher(u, v, w)
  si $d[v] > d[u] + w(u,v)$ **alors**
    $d[v] \leftarrow d[u] + w(u,v)$
    $\pi[v] \leftarrow u$

Dijkstra(G, w, s)
  Source-Unique-Initialization(G, s)
  $E \leftarrow \emptyset$
  $F \leftarrow S$
  tant que $F \neq \emptyset$ **faire**
    $u \leftarrow$ Extraire-Min(F)
    $E \leftarrow E \cup \{u\}$
    pour chaque arc (u,v) de G **faire**
      Relâcher(u, v, w)

Fig. 10.1 – Algorithme de Dijkstra pour le calcul des plus courts chemins.

– Le premier sommet ajouté à E est s car $d[s]$ vaut alors 0 quand toutes les autres distances estimées sont infinies.
– Supposons qu'à un instant donné pour chaque sommet u de E, $d[u]$ est bien la longueur du plus court chemin de s à u. On rajoute alors un sommet v à E. $d[v]$ est alors minimale parmi les sommets de $S \setminus E$. Montrons que $d[v] = \delta(s,v)$.
  Soit p un plus court chemin de s à v Soit y le premier sommet de p n'appartenant pas à E. Par minimalité de $d[v]$ on a : $d[v] \leq d[y]$. De plus on a $d[y] = \delta(s,y)$: parce que p contient le plus court chemin de s à x et donc de s au prédécesseur z de y, parce que $d[z] = \delta(s,z)$ par hypothèse de récurrence, et finalement parce que z a été relâché.
  Par positivité des poids, $\delta(s,y) \leq \delta(s,v)$. Donc $d[v] \leq d[y] = \delta(s,y) \leq \delta(s,v)$ et $d[v] = \delta(s,v)$.

La figure 10.2 présente un exemple d'exécution de l'algorithme de Dijkstra.

Complexité

La complexité de l'algorithme dépend de la complexité de l'opération Extraire-Min. Dans le cas (défavorable) où on implémente l'ensemble F au moyen d'un simple tableau, la recherche du minimum coûte à chaque fois $\Theta(|F|) = O(|S|)$. La boucle « tant que » s'exécutant exactement $|S|$ fois, et chaque arc étant visité une unique fois, la complexité de l'algorithme est $O(|S|^2 + |A|) = O(|S|^2)$.

10.1.2 Algorithme de Bellman-Ford

L'algorithme de Bellman-Ford résout le problème des plus courts chemins avec origine unique dans le cas général où le poids des arcs peut être négatif. Appelé sur un graphe $G = (S, A)$, l'algorithme de Bellman-Ford renvoie un booléen indiquant si le graphe contient ou non un circuit de poids strictement négatif accessible à partir de l'origine. L'algorithme est présenté figure 10.3.

Correction

La correction de l'algorithme de Bellman-Ford peut se montrer par récurrence sur le nombre d'arcs des plus courts chemins : à la fin de la i^e itération de la première boucle, les plus courts chemins contenant au

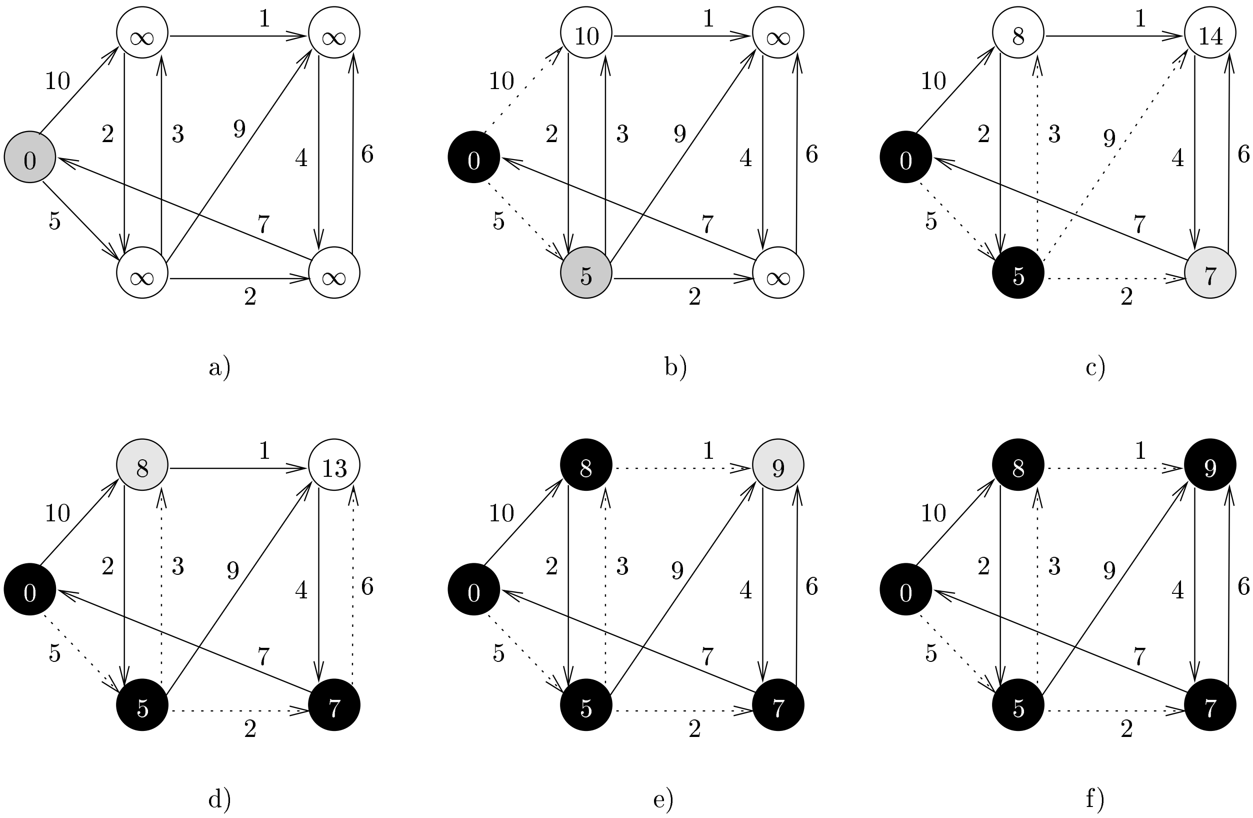


FIG. 10.2 – Exemple d'exécution de l'algorithme de Dijkstra : l'origine est le sommet le plus à gauche ; dans chaque graphe, les sommets noirs sont éléments de E, le sommet grisé est celui qui va être rajouté à E et les arcs en pointillés sont ceux utilisés pour les estimations des plus courts chemins, les longueurs de ces chemins étant indiquées dans les sommets.

BELLMAN-FORD(G, s, w)
  SOURCE-UNIQUE-INITIALIZATION(G, s)
  pour $i \leftarrow 1$ **à** $|S| - 1$ **faire**
    pour chaque arc $(u, v) \in A$ **faire**
      RELÂCHER(u, v, w)
  pour chaque arc $(u, v) \in A$ **faire**
    si $d[v] > d[u] + w(u, v)$ **alors renvoyer** FAUX
  renvoyer VRAI

FIG. 10.3 – Algorithme de Bellman-Ford pour le calcul des plus courts chemins.

plus i arcs sont connus, à la condition que le graphe ne contienne aucun circuit de poids strictement négatif. $|S|-1$ itérations suffisent car un plus court chemin est élémentaire (sans perte de généralité) et contient donc au plus $|S|-1$ arcs.

Vu ce qui précède, l'algorithme renvoie VRAI s'il n'y a pas de circuit de poids strictement négatif. Montrons qu'il renvoie FAUX sinon. Pour s'en convaincre, prenons un circuit c de sommets $u_0, u_1, ..., u_{p-1}, u_p u_0$. Si l'algorithme renvoie VRAI, alors pour tout $i \in [1,p]$, on a $d(u_i) \leq d(u_{i-1} + w(u_{i-1}, u_i)$. Par sommation on obtient :

$$\sum_{i=1}^{p} d(u_i) \leq \sum_{i=1}^{p} d(u_{i-1} + \sum_{i=1}^{p} w(u_{i-1}, u_i) \Leftrightarrow \sum_{i=1}^{p} d(u_i) \leq \sum_{i=0}^{p-1} d(u_i + w(c) \Leftrightarrow d(u_p) \leq d(u_0) + w(c) \Leftrightarrow 0 \leq w(c).$$

Donc, si l'algorithme renvoie VRAI le graphe ne contient pas de circuit de poids strictement négatif.

Complexité

Cet algorithme est en $\Theta(|S|.|A|)$ car l'initialisation et la vérification de la non-existence d'un circuit de poids strictement négatif sont en $\Theta(|S|)$ et $\Theta(|A|)$ respectivement, et car la boucle « pour » s'exécute exactement $(|S|-1)$ fois et que chaque itération visite chaque arc exactement une fois ce qui nous coûte $\Theta(|S|.|A|)$.

La figure 10.4 présente un exemple d'exécution de cet algorithme.

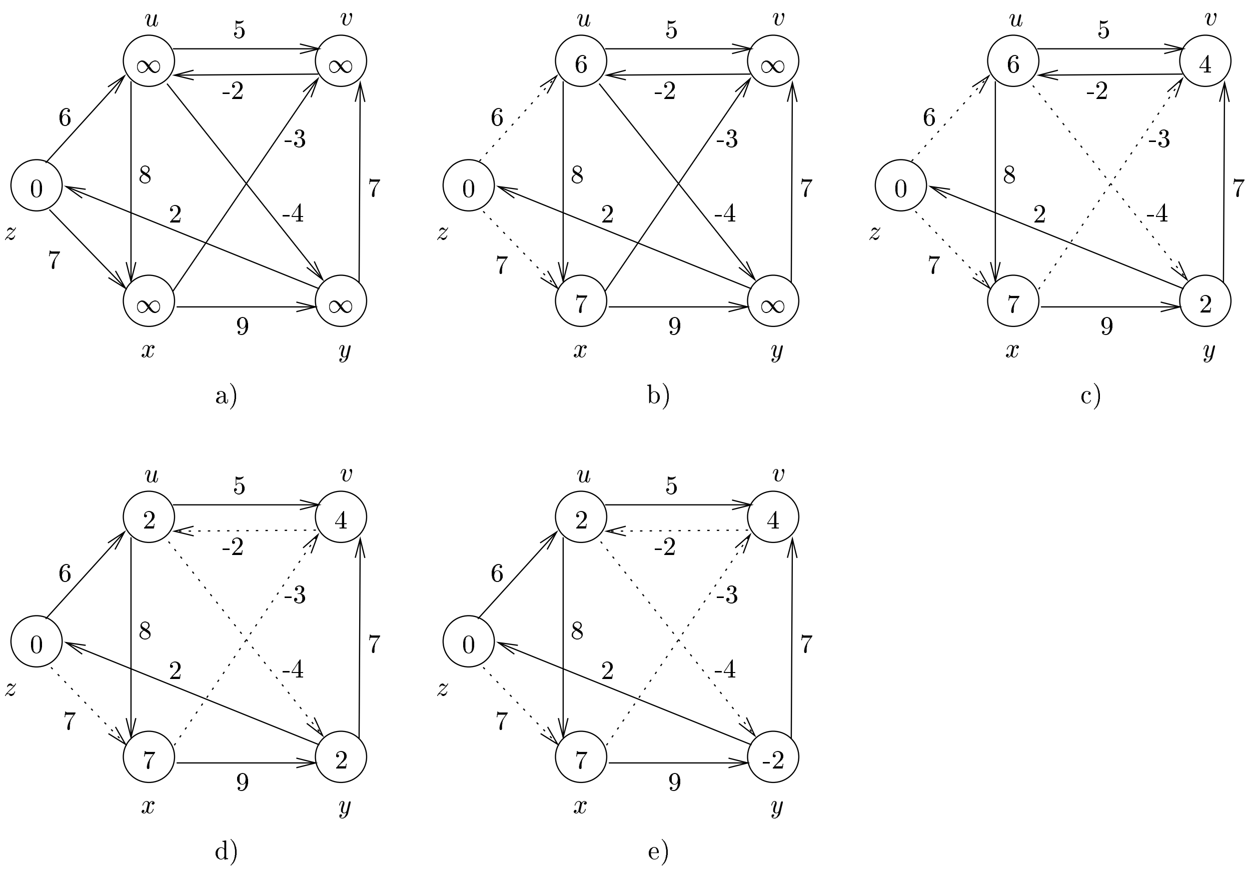


FIG. 10.4 – Exemple d'exécution de l'algorithme de Bellman-Ford : l'origine est le sommet le plus à gauche ; dans chaque graphe les arcs en pointillés sont ceux utilisés pour les estimations des plus courts chemins, les longueurs de ces chemins étant indiquées dans les sommets. Les arcs sont considérés dans l'ordre lexicographique : (u,v), (u,x), (u,y), (v,u), (x,v), (x,y), (y,v), (z,u) et (z,x).

10.2 Plus courts chemins pour tout couple de sommets

Nous nous intéressons ici à la recherche des plus courts chemins entre tous les couples de sommets d'un graphe (typiquement on cherche à élaborer la table des distances entre tous les couples de villes d'un atlas routier). On dispose en entrée d'un graphe $G = (S, A)$ et d'une fonction de pondération w.

Nous supposons dans cette section qu'il peut y avoir des arcs de poids négatifs, mais qu'il n'existe pas de circuits de poids strictement négatifs.

10.2.1 Programmation dynamique naïve

Sous-structure optimale

Comme nous l'avons déjà remarqué précédemment (pour l'algorithme glouton de Dijkstra), tout sous-chemin d'un plus court chemin est lui-même un plus court chemin.

Résolution récursive

La récursion porte ici sur le nombre d'arcs du plus court chemin. On note $d_{i,j}^{(m)}$ le poids minimal d'un chemin d'au plus m arcs du sommet i au sommet j. Pour $m = 0$ il existe un plus court chemin sans arc de i vers j si et seulement si $i = j$:

$$d_{i,j}^{(0)} = \begin{cases} 0 & \text{si } i = j, \\ \infty & \text{sinon.} \end{cases}$$

Pour $m \geq 1$, $d_{i,j}^{(m)}$ est la longueur du plus court chemin de i à j contenant *au plus* m arcs. Soit un tel plus court chemin contient exactement m arcs et il est obtenu par concaténation d'un plus court chemin d'au plus $m-1$ arcs de i à un sommet k et de l'arc de k à j, soit il n'en contient au plus que $m-1$ et sa longueur est égale à $d_{i,j}^{(m-1)}$. Par conséquent :

$$d_{i,j}^{(m)} = \min\left(d_{i,j}^{(m-1)}, \min_{1 \leq k \leq n}\left\{d_{i,k}^{(m-1)} + w_{k,j}\right\}\right) = \min_{1 \leq k \leq n}\left\{d_{i,k}^{(m-1)} + w_{k,j}\right\},$$

la formule étant simplifiée grâce à la propriété : $w_{j,j} = 0$.

Calcul ascendant des poids des plus courts chemins

On note $W = (w_{i,j})_{1 \leq i,j \leq n}$ le matrice des poids et $D^{(m)} = \left(d_{i,j}^{(m)}\right)_{1 \leq i,j \leq n}$ la matrice des poids des plus courts chemins contenant au plus m arcs. Le calcul de $D^{(m)}$ à partir de $D^{(m-1)}$ et de W se fait au moyen de l'algorithme ci-dessous :

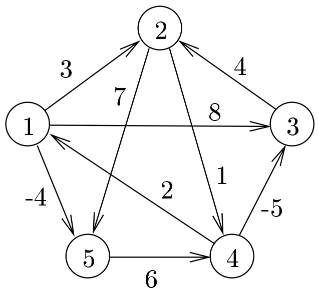
Extension-Plus-Courts-Chemins(D, W)
  $n \leftarrow$ *lignes*(D)
  soit $D' = (d'_{i,j})_{1 \leq i,j \leq n}$ une matrice carrée de taille n
  pour $i \leftarrow 1$ **à** n **faire**
    pour $j \leftarrow 1$ **à** n **faire**
      $d'_{i,j} \leftarrow +\infty$
      pour $k \leftarrow 1$ **à** n **faire**
        $d'_{i,j} \leftarrow \min(d'_{i,j}, d_{i,k} + w_{k,j})$
  renvoyer D'

L'algorithme Extension-Plus-Courts-Chemins s'exécute en $\Theta(n^3)$, à cause des trois boucles imbriquées. À partir de cet algorithme, la résolution du problème est triviale (cf. figure 10.5). Le coût total de résolution est donc en $\Theta(n^4)$.

La figure 10.6 présente un exemple d'exécution de cet algorithme.

PLUS-COURTS-CHEMINS(W)
  $n \leftarrow lignes(W)$
  $D^{(1)} \leftarrow W$
  pour $m \leftarrow 2$ **à** $n-1$ **faire**
    $D^{(m)} \leftarrow$ EXTENSION-PLUS-COURTS-CHEMINS($D^{(m-1)}, W$)
  renvoyer $D^{(n-1)}$

FIG. 10.5 – Algorithme naïf par programmation dynamique pour le calcul des plus courts chemins.



$$D^{(1)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \qquad D^{(2)} = \begin{pmatrix} 0 & 3 & 8 & 2 & -4 \\ 3 & 0 & -4 & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & \infty & 1 & 6 & 0 \end{pmatrix}$$

$$D^{(3)} = \begin{pmatrix} 0 & 3 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \qquad D^{(4)} = \begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix}$$

FIG. 10.6 – Un graphe orienté et la séquence des matrices calculées par PLUS-COURTS-CHEMINS.

10.2.2 Algorithme de Floyd-Warshall

L'algorithme de Floyd-Warshall est un autre algorithme conçu suivant le principe de la programmation dynamique.

Structure d'un plus court chemin

Ici, la récursion n'a pas lieu sur le nombre d'arcs d'un plus court chemin, mais sur les sommets intermédiaires de ces chemins, un sommet intermédiaire étant un sommet autre que les extrémités du chemin. On note $\{1, 2, ..., n\}$ les n sommets de G. Ici, $d_{i,j}^{(k)}$ est la longueur du plus court chemin de i à j n'utilisant comme sommets intermédiaires que des sommets parmi $\{1, 2, ..., k\}$. De deux choses l'une, un plus court chemin de i à j n'ayant comme sommets intermédiaires que des sommets de $\{1, 2, .., k\}$ contient ou ne contient pas le sommet k :

1. Si le plus court chemin p de i à j et n'ayant comme sommets intermédiaires que des sommets de $\{1, 2, .., k\}$ a effectivement comme sommet intermédiaire k, alors p est de la forme $i \overset{p_1}{\leadsto} k \overset{p_2}{\leadsto} j$ où p_1 (resp. p_2) est un plus court chemin de i à k (resp. de k à j) n'ayant comme sommets intermédiaires que des sommets de $\{1, 2, ..., k-1\}$.
2. Si le plus court chemin p de i à j et n'ayant comme sommets intermédiaires que des sommets de $\{1, 2, .., k\}$ ne contient pas k, alors c'est un plus court chemin p de i à j et n'ayant comme sommets intermédiaires que des sommets de $\{1, 2, .., k-1\}$.

Résolution récursive

La structure explicitée aux paragraphes précédent nous donne directement une récursion définissant $d_{i,j}^{(k)}$:

$$d_{i,j}^{(k)} = \begin{cases} w_{i,j} & \text{si } k = 0, \\ \min(d_{i,j}^{(k-1)}, d_{i,k}^{(k-1)} + d_{k,j}^{(k-1)}) & \text{sinon.} \end{cases}$$

Calcul ascendant des poids des plus courts chemins

L'algorithme est présenté figure 10.7.

FLOYD-WARSHALL(W)
  $n \leftarrow$ *lignes*(W)
  $D^{(0)} \leftarrow W$
  pour $k \leftarrow 1$ **à** n **faire**
    pour $i \leftarrow 1$ **à** n **faire**
      pour $j \leftarrow 1$ **à** n **faire**
        $d_{i,j}^{(k)} \leftarrow \min(d_{i,j}^{(k-1)}, d_{i,k}^{(k-1)} + d_{k,j}^{(k-1)})$
  renvoyer $D^{(n)}$

FIG. 10.7 – Algorithme de Floyd-Warshall pour le calcul des plus courts chemins.

Construction des plus courts chemins

Tout comme on a défini récursivement les longueurs des plus courts chemins, on peut définir récursivement les prédécesseurs dans les plus courts chemins : $\pi_{i,j}^{(k)}$ représente ici le prédécesseur du sommet j dans le plus court chemin de i à j n'utilisant comme sommets intermédiaires que des sommets parmi $\{1, 2, ..., k\}$. Pour $k = 0$, un plus court chemin ne possède aucun sommet intermédiaire, donc :

$$\pi_{i,j}^{(0)} = \begin{cases} \text{NIL} & \text{si } i = j \text{ ou } w_{i,j} = \infty, \\ i & \text{si } i \neq j \text{ et } w_{i,j} < \infty. \end{cases}$$

Dans le cas général, si le plus court chemin est de la forme $i \rightsquigarrow k \rightsquigarrow j$ le prédécesseur de j est le même que celui du plus court chemin de k à j et n'utilisant comme sommets intermédiaires que des sommets parmi $\{1, 2, ..., k-1\}$. Autrement, on prend le même prédécesseur de j que celui qui se trouvait sur le plus court chemin de i à j et n'utilisant comme sommets intermédiaires que des sommets parmi $\{1, 2, ..., k-1\}$. Nous avons donc, dans tous les cas :

$$\pi_{i,j}^{(k)} = \begin{cases} \pi_{i,j}^{(k-1)} & \text{si } d_{i,j}^{(k-1)} \leq d_{i,k}^{(k-1)} + d_{k,j}^{(k-1)}, \\ \pi_{k,j}^{(k-1)} & \text{si } d_{i,j}^{(k-1)} > d_{i,k}^{(k-1)} + d_{k,j}^{(k-1)}. \end{cases}$$

Complexité

On remarque aisément que l'algorithme de Floyd-Warshall est de complexité $\Theta(n^3)$.

La figure 10.8 présente le résultat de l'exécution de l'algorithme de Floyd-Warshall sur le graphe de la figure 10.6.

$$D^{(0)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \qquad \Pi^{(0)} = \begin{pmatrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & \text{NIL} & 4 & \text{NIL} & \text{NIL} \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(1)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \qquad \Pi^{(1)} = \begin{pmatrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & 1 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(2)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \qquad \Pi^{(2)} = \begin{pmatrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & 2 & 2 \\ 4 & 1 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(3)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \qquad \Pi^{(3)} = \begin{pmatrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & 2 & 2 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(4)} = \begin{pmatrix} 0 & 3 & -1 & 4 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \qquad \Pi^{(4)} = \begin{pmatrix} \text{NIL} & 1 & 4 & 2 & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(5)} = \begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \qquad \Pi^{(5)} = \begin{pmatrix} \text{NIL} & 3 & 4 & 5 & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{pmatrix}$$

FIG. 10.8 – Séquence des matrice $D^{(k)}$ et $\Pi^{(k)}$ calculées par l'algorithme FLOYD-WARSHALL pour le graphe de la figure 10.6.

Chapitre 11

NP-complétude

Tous les algorithmes que nous avons vu jusqu'à présent, étaient des algorithmes en temps polynomial : sur des entrées de taille n, leur temps d'exécution dans le pire cas était en $O(n^k)$ pour une certaine constante k. D'où la question : **tous les problèmes peuvent-ils être résolus en temps polynomial ?**

– **Non**, car certains ne peuvent pas être résolus (non décidabilité de la terminaison) ;
– **Non**, *a priori*, car il y a des problèmes pour lesquels on ne connaît que des algorithmes de coût exponentiel.

On aimerait donc savoir si un problème peut ou non être résolu par un algorithme polynomial : s'il ne peut exister d'algorithme polynomial pour le résoudre, il vaudra alors sans doute mieux développer un **algorithme d'approximation** (ou **heuristique**) polynomial qu'un algorithme de résolution exact à la complexité super-polynomiale.

La question de l'existence d'un algorithme de résolution de complexité polynomiale nous amène à définir des **classes de complexité** : intuitivement on aimerait avoir une classe des programmes que l'on peut résoudre en temps polynomial, une classe de problème plus compliqués, et un moyen de déterminer à quelle classe appartient un problème.

11.1 La classe P

11.1.1 Problèmes abstraits

Définition

On définit un **problème abstrait** Q comme une relation binaire sur un ensemble I d'**instances** d'un problème et un ensemble S de **solutions** de ce problème.

Exemple : prenons le problème PLUS-COURT-CHEMIN qui consiste à trouver le plus court chemin entre deux sommets d'un graphe.

– Une instance de ce problème est un triplet composé d'un graphe et de deux sommets.
– Une solution du problème est une séquence de sommets du graphe (si la séquence est vide, il n'existe pas de chemin du graphe reliant les deux sommets).
– Le problème lui-même est la relation qui associe à une instance donnée une ou plusieurs solutions.

Restriction aux problèmes de décision

Dans le cadre de la théorie de la NP-complétude, nous nous restreindrons aux **problèmes de décision**, c'est-à-dire ceux dont la solution est soit *vrai* soit *faux*.

Exemple : prenons le problème CHEMIN qui répond à la question « étant donné un graphe G, deux sommets u et v et un entier positif k, existe-t-il dans G un chemin de u à v de longueur au plus k ? ».

Problèmes d'optimisation

De nombreux problèmes abstraits ne sont pas des problèmes de décisions mais des **problèmes d'optimisation**. Pour leur appliquer la théorie de la NP-complétude, le plus souvent on les reformulera sous la forme d'un problème d'optimisation en imposant une borne sur la valeur à optimiser, comme nous l'avons fait en passant du problème PLUS-COURT-CHEMIN au problème CHEMIN.

11.1.2 Codage

Définition

Pour qu'un programme informatique puisse résoudre un problème abstrait, il faut que ces instances soient représentées sous une forme compréhensible par le programme. On appelle **codage** d'un ensemble S d'objets abstraits une application e de S dans l'ensemble des chaînes binaires (ou dans l'ensemble des chaînes d'un alphabet fini quelconque). Exemple : le classique codage des entiers sous forme binaire...

Un algorithme informatique qui « résout » un certain problème de décision prend en fait en entrée un codage d'une instance de ce problème. Un problème dont les instances forment l'ensemble des chaînes binaires est appelé **problème concret**. On dit qu'un algorithme **résout** un problème concret en $O(T(n))$ quand, sur une instance i du problème de longueur $n = |i|$, l'algorithme est capable de produire la solution en au plus $O(T(n))$. Un problème concret est donc résoluble en temps polynomial s'il existe un algorithme permettant de le résoudre en temps $O(n^k)$ pour une certaine constante k.

Définition 14 (Classe de complexité P). *La classe de complexité P est l'ensemble des problèmes concrets de décision qui sont résolubles en temps polynomial.*

L'importance des codages

Pour quoi s'embêter avec des codages plutôt que de définir directement la complexité d'un problème abstrait ? Parce que la complexité dépend du codage... Pour le voir, considérons un algorithme qui prend comme unique entrée un entier k, et dont le temps d'exécution est en $\Theta(k)$.

- Si l'entier k est fourni en **unaire** (son codage est alors une chaîne de k 1), le temps d'exécution de l'algorithme est en $O(n)$ sur des entrées de longueur n, et l'algorithme est de complexité **polynomiale**.
- Si l'entier k est fourni en **binaire**, la longueur du codage est alors de $n = \lfloor \log_2 k \rfloor + 1$, et le temps d'exécution de l'algorithme est en $\Theta(k) = \Theta(2^n)$, et l'algorithme est de complexité **superpolynomiale**.

On ne peut donc pas parler de la complexité de la résolution d'un problème abstrait sans spécifier son codage.

Relativiser cette importance

Définition 15 (Fonction calculable en temps polynomial). *Une fonction $f : \{0,1\}^* \rightarrow \{0,1\}^*$ est **calculable en temps polynomial** s'il existe un algorithme polynomial qui, étant donné une entrée $x \in \{0,1\}^*$ quelconque, produit le résultat $f(x)$.*

Deux codages e_1 et e_2 définis sur un même ensemble S sont reliés polynomialement s'il existe deux fonctions calculables en temps polynomial, f_{12} et f_{21} telles que pour tout $s \in S$ on a $f_{12}(e_1(s)) = e_2(s)$ et $f_{21}(e_2(s)) = e_1(s)$. Autrement dit, un codage peut être calculé à partir de l'autre en temps polynomial, et réciproquement.

Théorème 9. *Soit Q un problème de décision abstrait et soient e_1 et e_2 deux codages (des instances de Q) reliés polynomialement. Alors, le problème concret défini par Q et e_1 appartient à la classe P si et seulement si il en va de même du problème concret défini par Q et e_2.*

11.2 La classe NP

11.2.1 Algorithme de validation

Considérons le problème CHEMIN et une de ses instances (G, u, v, k). La question qui nous intéresse est donc : existe-t-il dans le graphe G un chemin reliant les sommets u et v dont la longueur est inférieure ou égale à k ? Si l'on se donne également un chemin p de u vers v, on peut facilement vérifier que la longueur de p est au plus égale à k et, le cas échéant on peut voir p comme un **certificat** que le problème de décision CHEMIN renvoie *vrai* sur cette instance.

Ici, la validation du fait que le problème concret de décision CHEMIN renvoie *vrai* sur l'instance (G, u, v, k), validation effectuée à partir du certificat p, prend autant de temps que la résolution du problème à partir de rien. Ce n'est pas toujours le cas.

Exemple : il est trivial de vérifier qu'un chemin est un *cycle hamiltonien* (cycle simple contenant tous les sommets) d'un graphe donné alors que l'on ne sait résoudre ce problème qu'en temps super polynomial.

Définition 16 (Algorithme de validation). *Soit un* ***problème concret*** *de décision Q. Un* ***algorithme de validation*** *pour Q est un algorithme de décision $\mathtt{A}$ à deux arguments, où un argument est une instance x du problème, et où l'autre argument est un certificat y. L'algorithme $\mathtt{A}$ valide l'entrée x si et seulement si il existe un certificat y tel que $\mathtt{A}(x, y) = vrai$. Bien évidemment, l'algorithme A ne doit valider que les instances x de Q pour lesquelles $Q(x)$ est vrai. Si $Q(x) = faux$, il ne doit pas y avoir de certificat validant x.*

Exemple : dans le problème du cycle hamiltonien, le certificat est la liste des sommets du cycle hamiltonien. Si un graphe est hamiltonien, le cycle lui-même offre toute l'information nécessaire pour le prouver. Réciproquement, si un graphe n'est pas hamiltonien, il n'existe aucune liste de sommets capable de faire croire à l'algorithme de validation que le graphe est hamiltonien : l'algorithme de validation se rend bien compte que le cycle décrit par la liste des sommets n'est pas un cycle du graphe étudié.

Remarque : dans l'immense majorité des cas le certificat sera une « solution » du problème considéré...

11.2.2 La classe de complexité NP

Définition 17 (Classe de complexité NP). *La classe de complexité NP est l'ensemble des problèmes concrets de décision Q pour lesquels il existe un algorithme polynomial de validation A.*

$$\exists c \geq 0 \text{ telle que pour tout } x \text{ instance de } Q : \quad Q(x) = vrai \;\Leftrightarrow\; \exists y \text{ certificat}, |y| = O(|x|^c), A(x, y) = vrai$$

Remarques

- D'après cette définition et ce qui précède, le problème CYCLE-HAMILTONIEN appartient à NP.
- P $\subset$ NP (soit Q un problème de la classe P, il existe donc un algorithme polynomial qui résout Q, on le convertit facilement en algorithme de validation qui ignore le certificat).
- P = NP ? On n'en sait rien. La majorité des chercheurs pense que P $\neq$ NP, et donc que P $\subsetneq$ NP.
  La classe de complexité P est la classe des problèmes qui peuvent être résolus rapidement. La classe de complexité NP est celle des problèmes pour lesquels une solution peut être rapidement validée (vérifiée). Intuitivement, P $\subsetneq$ NP signifierait qu'il existe des algorithmes difficiles à résoudre mais dont une solution peut être facilement vérifiée...

11.3 NP-complétude

Une des raisons qui laissent à penser que P $\neq$ NP est l'existence de la classe des problèmes *NP-complets* : si *un seul* problème NP-complet peut être résolu en temps polynomial, alors *tous* les problèmes de NP peuvent être résolus en temps polynomial et P = NP. Mais aucun algorithme polynomial n'a jamais été découvert pour aucun problème NP-complet. *Les problèmes NP-complets sont, dans un certain sens, les problèmes les plus « difficiles » de NP.*

11.3.1 Réductibilité

Nous avons besoin de pouvoir comparer la difficulté de problèmes. Intuitivement, un problème Q_1 peut être ramené à un problème Q_2 si une instance quelconque x de Q_1 peut être « facilement reformulée » comme une certaine instance y de Q_2. Dans ce cas, la résolution du problème $Q_2(y)$ nous fournira la solution du problème $Q_1(x)$ et le problème Q_1 n'est, dans un certain sens, « pas plus difficile à résoudre » que le problème Q_2.

Exemple trivial : le problème de la résolution d'équations linéaires à une inconnue ($a \times x + b = 0$) peut être ramenée à la résolution d'équations quadratiques ($a \times x^2 + b \times x + c = 0$).

Définition 18 (Problème réductible à un autre en temps polynomial). *Soient Q_1 et Q_2 deux problèmes concrets. Q_1 est* ***réductible en temps polynomial*** *à Q_2 (ce que l'on note $Q_1 \leq_P Q_2$) s'il existe une fonction calculable en temps polynomial $f : \{0,1\}^* \rightarrow \{0,1\}^*$ telle que pour tout $x \in \{0,1\}^*$:*

$$Q_1(x) = vrai \quad \text{si et seulement si} \quad Q_2(f(x)) = vrai.$$

Exemple non trivial :

1. Problème Q_1 : problème de l'existence d'un cycle Hamiltonien (cycle simple qui comprend tous les sommets) dans un graphe donné G_1.
2. Problème Q_2 : étant donné un graphe G_2 de villes valué des distances inter-villes, existe-t-il un cycle (pas forcément simple) passant par toutes les villes, et de longueur inférieure à une valeur fixée M ?
3. Réduction de Q_1 à Q_2 :
   - On crée un graphe G_2 de villes contenant autant de villes que G_1 de sommets. On associe chaque sommet de G_1 à une ville de G_2.
   - Si deux sommets de G_1 sont reliés par une arête, on relie les deux villes correspondantes de G_2 par une arête valuée de la distance interville « 1 », et sinon valuée par la distance interville « 2 ».
4. On exécute l'algorithme résolvant Q_2 sur G_2 avec $M = n$, le nombre de sommets de G_1. S'il existe un tel cycle... il est hamiltonien ! Et si G_1 admet un cycle hamiltonien, G_2 admet un cycle tel que recherché.
5. G_2 contient autant de villes que G_1 de sommets, et le nombre d'arêtes de G_2 est égal au nombre de paires de sommets de G_1 : $\frac{n(n-1)}{2}$. La réduction est linéaire en la taille de G_2 et est donc bien polynomiale en la taille de G_1.

11.3.2 Définition de la NP-complétude

Les réductions en temps polynomial fournissent un moyen formel de montrer qu'un problème est au moins aussi difficile qu'un autre, à un facteur polynomial près : si $Q_1 \leq_P Q_2$, alors Q_1 n'est pas plus difficile à résoudre à un facteur polynomial près, que Q_2. Les réductions nous permettent de définir l'ensemble des problèmes NP-complets, qui sont les problèmes les plus difficiles de NP.

Définition 19 (Problème NP-complet). *Un problème Q est* ***NP-complet*** *si*

1. *$Q \in NP$.*
2. *$\forall Q' \in NP,\ Q' \leq_P Q$.*

On note NPC la classe des problèmes NP-complets.
Un problème concret qui vérifie la propriété 2 mais pas nécessairement la propriété 1 est dit ***NP-difficile****.*

Théorème 10. *Si un problème de NP est résoluble en temps polynomial, alors $P = NP$. De façon équivalente, si un problème quelconque de NP n'est pas résoluble en temps polynomial, alors aucun problème NP-complet ne peut se résoudre en temps polynomial.*

11.3.3 Exemples de problèmes NP-complets

Il existe des problèmes NP-complets :

- SAT : soit une formule booléenne composée de variables x_1, ..., x_n et de connecteurs (et, ou, non, implication, équivalence) et de parenthèses ; existe-t-il une affectation des variables x_1, ..., x_n pour laquelle la formule soit vraie ?
  Premier problème dont la NP-complétude ait été démontrée, par Cook en 1971.
- 3-SAT : même problème que SAT, la formule étant sous forme normale conjonctive à trois littéraux, c'est-à-dire de la forme : $\mathrm{ET}_{i \in I}(t_{i,1}$ ou $t_{i,2}$ ou $t_{i,3})$ avec $\forall i, j,\ \exists k,\ t_{i,j} = x_k$ ou $t_{i,j} = \neg x_k$.
- PARTITION : peut-on diviser un ensemble d'entier en deux ensembles de même somme ?
- CLIQUE : un graphe donné contient-il une clique (un sous-graphe complet) de taille k ?
- CYCLE-HAMILTONIEN.
- VOYAGEUR-DE-COMMERCE : le voyageur de commerce veut faire la tournée d'un ensemble de villes (cycle hamiltonien) la plus courte possible.
- 3-COLORIAGE D'UN GRAPHE : peut-on colorier à l'aide de trois couleurs les sommets d'un graphe de sorte que deux sommets adjacents aient des couleurs différentes ?

11.3.4 Preuves de NP-complétude

Comment démontrer qu'un problème est NP-complet ?

Théorème 11. *Si Q_1 est un problème tel que $Q_2 \leq_P Q_1$ pour un certain problème $Q_2 \in NPC$, alors Q_1 est NP-difficile. De plus, si $Q_1 \in NP$, alors $Q_1 \in NPC$.*

Commentaire : la première assertion montre que Q_1 est polynomialement plus difficile qu'un problème polynomialement plus difficile que tous les problèmes de NP.

Méthode pour montrer la NP-complétude d'un problème Q_1

1. Prouver que $Q_1 \in$ NP.
2. Choisir un problème NP-complet Q_2.
3. Décrire un algorithme polynomial capable de calculer une fonction f faisant correspondre toute instance de Q_2 à une instance de Q_1.
4. Démontrer que la fonction f satisfait la propriété :

$$Q_2(x) = vrai \quad \text{si et seulement si} \quad Q_1(f(x)) = vrai.$$

5. Démontrer que l'algorithme calculant f s'exécute en temps polynomial.

Preuve de la NP-complétude de Cycle-Ham

1. On a vu à la section 11.2.1 un algorithme polynomial de validation de CYCLE-HAM. Par conséquent, CYCLE-HAM $\in$ NP.
2. On a choisi le problème de l'existence d'un cycle passant par tous les sommets et de taille bornée (on suppose que l'on sait que ce problème est NP-complet).
3. On a vu à la section 11.3.1 un algorithme de réduction.
4. On a montré à la section 11.3.1 la correction de la réduction.
5. On a montré à la section 11.3.1 que l'algorithme de réduction était polynomial.

Donc le problème CYCLE-HAM est NP-complet.

Chapitre 12

Heuristiques

Si le problème à résoudre est NP-complet, plutôt que d'élaborer un algorithme de complexité super-polynomiale, on peut avoir intérêt à recourir à un **algorithme d'approximation** — ou **heuristique** — c'est-à-dire à un algorithme qui ne construira que des solutions *presque* optimales. On recherchera bien évidemment des algorithmes d'approximations de complexité polynomiale .

Bornes de performances

Supposons que l'on cherche à résoudre un problème d'optimisation dans lequel chaque solution potentielle a un coût positif et que l'on cherche à trouver une solution de coût minimal. Un algorithme d'approximation a une **borne** $\rho(n)$ si pour toute entrée de taille n, le coût d'une solution produite par l'algorithme est au plus $\rho(n)$ fois le coût C^* d'une solution optimale :

$$\frac{C}{C^*} \leq \rho(n),$$

autrement dit, un algorithme d'approximation a une borne $\rho(n)$ si pour toute entrée de taille n, une solution produite par l'algorithme est au pire $\rho(n)$ fois plus coûteuse qu'une solution optimale. Un algorithme d'approximation qui admet une borne est appelé **heuristique garantie**.

On peut, de même définir l'erreur relative d'un algorithme d'approximation par :

$$\frac{|C - C^*|}{C^*}.$$

Un algorithme d'approximation a une borne d'erreur relative égale à $\epsilon(n)$ si et seulement si :

$$\frac{|C - C^*|}{C^*} \leq \epsilon(n).$$

Un **schéma d'approximation** est un algorithme d'approximation qui prend en entrée, en plus d'une instance du problème, une valeur $\epsilon > 0$ et qui renvoie une solution approchée avec une borne d'erreur relative égale à ϵ. Il s'agit donc d'une heuristique dont on peut contraindre la précision.

12.1 Le problème de la couverture de sommet

Soit un graphe non orienté $G = (S, A)$. Une **couverture de sommet** est un sous ensemble S' de S ($S' \subset S$) tel que si (u, v) est une arête de G, alors soit $u \in S'$, soit $v \in S'$ (soit u et v appartiennent tous deux à S'). La taille d'une couverture est le nombre de sommets qu'elle contient.

Le **problème de la couverture de sommet** consiste à trouver une couverture de sommet de taille minimale. Ce problème est NP-difficile (le problème de décision associé est NP-complet).

12.1.1 Heuristique

Bien que le problème de la couverture de sommet soit compliqué, on peut facilement concevoir une heuristique garantie pour le résoudre :

Couverture-Sommet-Approchée(G)
  $C \leftarrow \emptyset$
  $A' \leftarrow A$
  tant que $A' \neq \emptyset$ **faire**
    soit (u, v) une arête arbitraire de A'
    $C \leftarrow C \cup \{u, v\}$
    supprimer de A' toutes les arêtes incidentes soit à u soit à v
  renvoyer C

12.1.2 Exemple d'utilisation

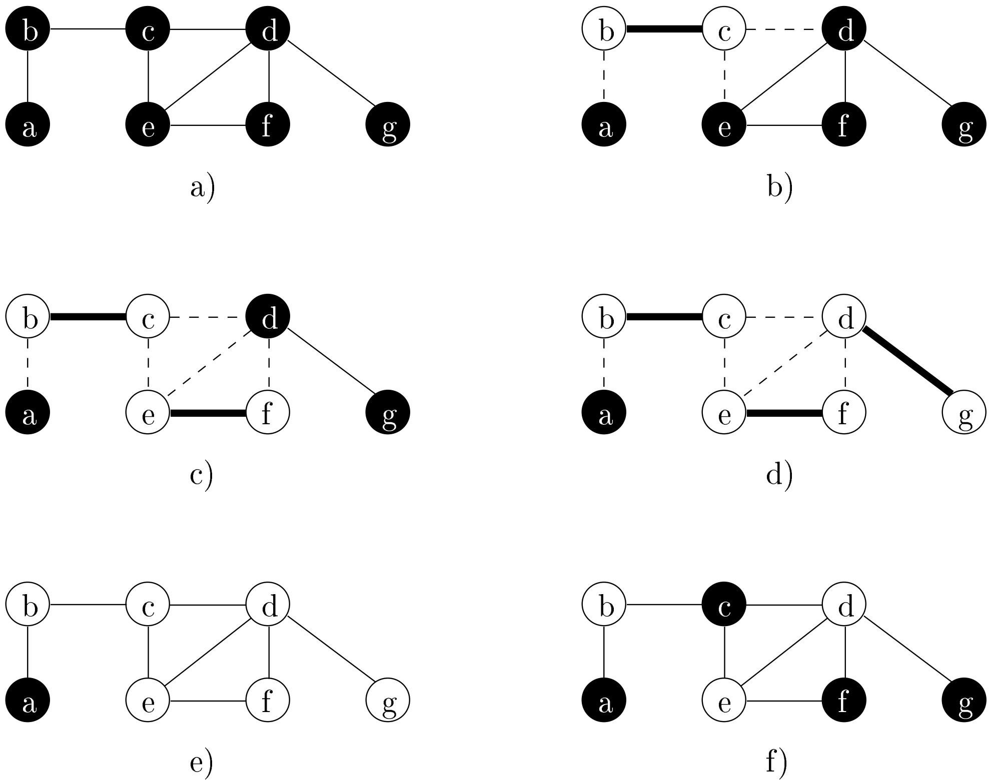


Fig. 12.1 – Exemple d'utilisation de l'algorithme Couverture-Sommet-Approchée, les sommets sur fond blanc étant ceux appartenant à la couverture : a) le graphe G de départ ; b) l'arête (`b`, `c`), en gras, est la première choisie par Couverture-Sommet-Approchée, les sommets `b` et `c` sont rajoutés à la couverture ; c) l'arête (`e`, `f`) est choisie et les sommets `e` et `f` sont rajoutés à la couverture ; d) l'arête (`d`, `g`) est choisie et les sommets `d` et `g` sont rajoutés à la couverture ; e) la couverture produite contient donc les sommets `b`, `c`, `d`, `e`, `f` et `g` ; f) la couverture optimale ne contient que trois sommets : `b`, `d` et `e`.

12.1.3 Garantie de performances

Théorème 12. *L'heuristique* Couverture-Sommet-Approchée *possède une borne de 2.*

Démonstration

Soit E l'ensemble des arêtes qui ont été choisies à la ligne 4 de l'heuristique. Par construction, deux arêtes quelconque de E ne peuvent pas avoir un sommet en commun. Donc chaque exécution de la ligne 5 de l'heuristique ajoute deux nouveaux sommets à C et $|C| = 2 \times |E|$. Soit C^* une couverture de sommet de taille minimale. Comme deux arêtes de E ne peuvent avoir de sommets en commun, un sommet de C^* est incident à au plus une arête de E. Par ailleurs, par définition d'une couverture de sommets, C^* doit contenir au moins une des deux extrémités de chacune des arêtes de E. Donc $|E| \leq |C^*|$. D'où, $|C| \leq 2 \times |C^*|$.

12.2 Le problème du voyageur de commerce

Nous considérons ici aussi un graphe non orienté $G = (S, A)$. Mais ici le graphe est *complet* : chaque paire de sommets est reliée par une arête. On a un poids positif ou nul $w(u, v)$ associé à chaque arête (u, v) du graphe. Le problème est ici de trouver un cycle hamiltonien (une tournée) de poids minimal.

Nous restreignons ici le problème en supposant que la fonction de poids w vérifie **l'inégalité triangulaire** : soient u, v et w trois sommets quelconques, alors :

$$w(u, w) \leq w(u, v) + w(v, w).$$

TOURNÉE-APPROCHÉE(G, w)

Choisir arbitrairement un sommet r de G qui servira de « racine »
Construire un arbre couvrant minimal T pour G à partir de la racine r
Soit L la liste des sommets visités lors d'un parcours préfixe de T
renvoyer le cycle hamiltonien H qui visite les sommets dans l'ordre de L.

Un **arbre couvrant minimal** est un arbre qui contient tous les sommets du graphe (= couvrant) et dont la somme des poids des arêtes est minimale.

Un parcours préfixe visite tous les sommets d'un arbre. L'arbre T étant ici couvrant, la liste L contient bien tous les sommets du graphe et G est bien défini. Le parcours est préfixe : un nœud est donc visité avant que ses fils ne le soient.

La complexité de cet algorithme est en $\Theta(S^2)$ car le graphe est complet (c'est la complexité de la construction d'un arbre couvrant minimal dans ce cas).

12.2.1 Exemple d'utilisation

Voir la figure 12.2.

12.2.2 Garantie de performances

Théorème 13. TOURNÉE-APPROCHÉE *est un algorithme d'approximation ayant une borne égale à deux pour le problème du voyageur de commerce avec inégalité triangulaire.*

Démonstration

Nous devons donc montrer que, si H^* est une tournée optimale, on a $w(H) \leq 2w(H^*)$.

En supprimant certaines arêtes de H^* (n'importe laquelle dans notre exemple), on peut toujours obtenir un arbre couvrant T'. D'où $w(T') \leq w(H^*)$. Comme T est, par définition, un arbre couvrant de poids minimal, on a $w(T) \leq w(T') \leq w(H^*)$.

Un **parcours complet** de T liste les sommets dès qu'ils sont visités pour la première fois et également quand ils sont à nouveau traversés après la visite d'un sous-arbre. Soit W ce parcours. Le parcours complet dans notre exemple a pour résultat la liste :

$$a, b, c, b, h, b, a, d, e, f, e, g, e, d, a.$$

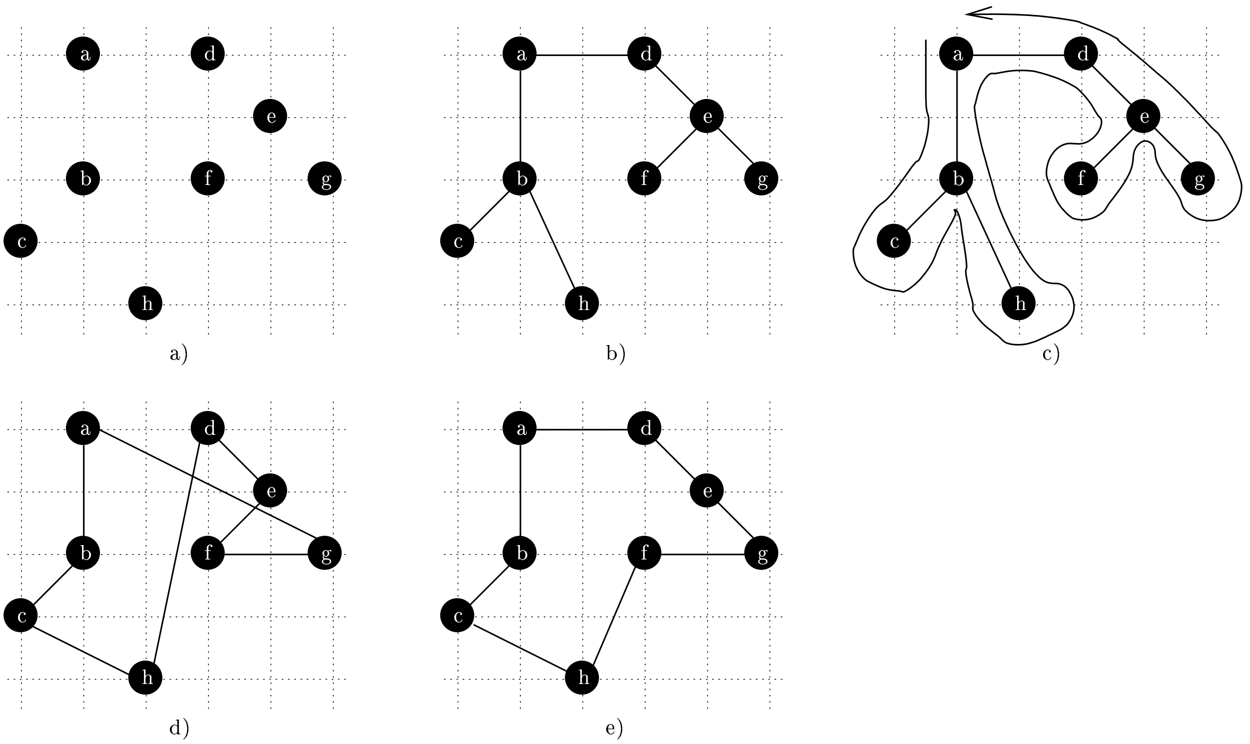


FIG. 12.2 – Exemple d'utilisation de l'algorithme TOURNÉE-APPROCHÉE : a) l'ensemble des sommets auxquels on fait correspondre les sommets d'une grille, le poids d'une arête étant égal à la distance euclidienne des deux sommets qu'elle relie ; b) arbre couvrant de poids minimal et de racine `a` ; c) parcours de l'arbre partant de `a` ; un parcours complet visite les sommets dans l'ordre : `a`, `b`, `c`, `b`, `h`, `b`, `a`, `d`, `e`, `f`, `e`, `g`, `e`, `d` et `a` ; un parcours préfixe visite les sommets dans l'ordre : `a`, `b`, `c`, `h`, `d`, `e`, `f` et `g` ; d) tournée des sommets obtenue à partir du parcours préfixe et de coût environ 19,074 ; e) tournée optimale de coût environ 14,715.

Un parcours complet traverse toutes les arêtes de T exactement deux fois. Par conséquent :

$$w(W) = 2w(T) \leq 2w(H^*).$$

W n'est pas une tournée, et notre démonstration n'est pas terminée ! Grâce à l'inégalité triangulaire, on peut supprimer de W la visite d'un sommet quelconque sans augmenter le poids : si un sommet v est supprimé de W entre les visites à u et w, la nouvelle liste va directement de u à w (avec un poids $w(u,w) \leq w(u,v) + w(v,w)$). En appliquant plusieurs fois ce résultat, on peut supprimer de W toutes les visites à chaque sommet sauf la première, et sauf la dernière visite du premier sommet (la racine). Dans notre exemple, on se retrouve avec la liste réduite :

$$a, b, c, h, d, e, f, g, a.$$

Cette liste est correspond exactement au parcours H et H est donc obtenu en supprimant (en utilisant l'inégalité triangulaire) des sommets du parcours complet W. Par conséquent :

$$w(H) \leq w(W) \text{ et } w(H) \leq 2 \times w(H^*).$$

Bibliographie

[1] Robert Cori and Jean-Jacques Lévy. Algorithmes et programmation. `http://www.enseignement.polytechnique.fr/profs/informatique/Jean-Jacque%s.Levy/poly/`. Cours de l'École Polytechnique.

[2] Thomas Cormen, Charles Leiserson, and Ronald Rivest. *Introduction à l'algorithmique*. Dunod, 1994.

[3] Donald E. Knuth. *Seminumerical Algorithms*, volume 2 of *The Art of Computer Programming*. Addison Wesley, 1969.

[4] Donald E. Knuth. *Sorting and searching*, volume 3 of *The Art of Computer Programming*. Addison Wesley, 1973.