**ENSIMAG Année Spéciale Informatique**
Année 2009/2010

# Algorithmique I

**Structures de données**
**Parcours de graphes**
**Programmation récursive**

**Pierre Berlioux**
**Marie-Paule Cani**
**Augustin Lux**
**Roger Mohr**
**Denis Naddef**
**Jean-Louis Roch**

# Table des matières

**1 Analyse du coût d'un algorithme** **3**
1.1 Mesures de complexité . . . 3
1.2 Taille des données . . . 4
1.3 Calcul de coûts par résolution d'équations aux récurrences . . . 5
1.3.1 Rappels sur les notations asymptotiques . . . 5
1.3.2 Algorithme itératif . . . 5
1.3.3 Algorithme récursif . . . 6
1.4 Cas le meilleur, cas le pire, complexité en moyenne . . . 7
1.4.1 Définitions . . . 7
1.4.2 Comment calculer une complexité en moyenne ? . . . 7
1.5 Exemples de calculs de complexité en moyenne . . . 8
1.5.1 Cas de la recherche séquentielle . . . 8
1.5.2 Algorithme de la collection complète . . . 8
1.5.3 Analyse en moyenne du tri par segmentation (quicksort) . . . 9
1.5.4 Un calcul de maximum . . . 11
1.5.5 Calcul des deux éléments maximaux . . . 11
1.6 Coût amorti . . . 14
1.6.1 Définition. . . . 14
1.6.2 Implantation amortie d'un tableau de taille variable . . . 15

**2 Files d'attente avec priorité** **23**
2.1 Représentations simples . . . 23
2.1.1 Ensemble fini de valeurs de priorité . . . 23
2.1.2 Ensemble infini de valeurs de priorité . . . 24
2.2 Représentation par tas . . . 24
2.2.1 Arbres parfaits partiellement ordonnés . . . 25
2.2.2 Représentation par tas . . . 26
2.2.3 Application : le tri par tas . . . 28

**3 Exemple : implantations de l'algorithme de Dijkstra** **33**
3.1 Introduction et rappels . . . 33
3.2 Recherche séquentielle . . . 34
3.3 Une remarque essentielle . . . 34
3.4 L'implantation de Dial . . . 35
3.5 L'implantation par tas binaire . . . 35
3.6 Les listes de Radix . . . 37

3.7 Conclusion . . . . . 38

**4 Dictionnaires 39**
4.1 Fonction caractéristique représentée par un tableau de booléens . . . . . 39
4.2 Tableaux et listes d'éléments . . . . . 40
4.2.1 Tableaux : recherche séquentielle et dichotomique . . . . . 40
4.2.2 Listes chaînées . . . . . 43
4.3 Tables de hachage (hash code) . . . . . 43
4.3.1 Représentation classique d'une table de hachage . . . . . 43
4.3.2 Analyse du coût . . . . . 44
4.3.3 Choix de la fonction de hachage . . . . . 45
4.3.4 Hachage universel . . . . . 46
4.3.5 Table de hachage à adressage dispersé (open addressing) . . . . . 47
4.4 Arbres binaires de recherche . . . . . 49
4.4.1 Opérations sur un ABR . . . . . 49
4.4.2 Complexités en moyenne des opérations . . . . . 51
4.5 Autres arbres de recherche - Les AVL . . . . . 53
4.6 Les "splay trees" . . . . . 56

**5 Bilan sur les structures de données 59**
5.1 Utilisation des structures classiques . . . . . 59
5.1.1 Tableau récapitulatif . . . . . 59
5.1.2 Structures hybrides (exercice) . . . . . 61

**6 Graphes : vocabulaire et représentations classiques 67**
6.1 Vocabulaire . . . . . 67
6.2 Calcul relationnel . . . . . 68
6.3 Représentation des graphes . . . . . 69
6.3.1 Représentation matricielle . . . . . 69
6.3.2 Représentation par une table des successeurs . . . . . 69

**7 Exploration arborescente 71**
7.1 Rappels sur l'exploration d'un arbre . . . . . 71
7.1.1 Spécification d'un arbre . . . . . 71
7.1.2 Exploration générique d'un arbre . . . . . 72
7.1.3 Application : exploration d'arbres combinatoires . . . . . 74
7.2 Exploration d'un DAG . . . . . 75
7.2.1 Exploration des sommets d'un DAG . . . . . 75
7.2.2 Application: tri topologique d'un DAG . . . . . 75
7.3 Parcours arborescent de graphes . . . . . 77
7.3.1 Exploration arborescente des sommets d'un graphe . . . . . 77
7.3.2 Enumération de tous les chemins dans un graphe . . . . . 77
7.3.3 Recherche de chemins optimaux par exploration . . . . . 78
7.3.4 Exploration des chemins optimaux en profondeur d'abord . . . . . 79
7.3.5 Exploration des chemins optimaux en largeur d'abord . . . . . 80



**8 Fermeture transitive d'un graphe 83**
8.1 Algorithme de produits sucessifs . . . . . 83
8.2 Construction par exploration arborescente . . . . . 83
8.2.1 Complexité de la fermeture transitive par exploration . . . . . 84
8.2.2 Etude de la complexité en moyenne . . . . . 84
8.3 Algorithme de Warshall . . . . . 86
8.4 Extension : calcul de tous les plus courts chemins par Floyd-Warshall . . . . . 87
8.5 Bilan pour la fermeture transitive . . . . . 87

**9 Recherche du plus court chemin 89**
9.1 Recherche par exploration arborescente . . . . . 89
9.2 Graphe orienté sans circuit: algorithme de Bellman . . . . . 90
9.3 Graphe orienté, coûts positifs : algorithme de Dijkstra . . . . . 93
9.4 Bilan de la recherche du plus court chemin . . . . . 94

**10 Applications à des problèmes de recherche opérationnelle 95**
10.1 Approximation linéaire par morceau d'une courbe . . . . . 95
10.2 Répartition des moyens de contrôle sur une chaîne . . . . . 96
10.3 Le problème du caboteur . . . . . 96
10.4 Système de contraintes formées de différences . . . . . 97
10.4.1 Arborescence des plus courts chemins . . . . . 97

**12 Diviser pour résoudre et applications 101**
12.1 Complexité des algorithmes "diviser pour régner" . . . . . 101
12.1.1 Cas où la résolution d'un seul sous problème suffit . . . . . 101
12.1.2 Cas général : division des données puis regroupement . . . . . 102
12.2 Exemples classiques . . . . . 103
12.2.1 La multiplication de grands entiers . . . . . 103
12.2.2 La multiplication de polynomes (exercice) . . . . . 104
12.2.3 Multiplication de matrices: algorithme de Strassen . . . . . 104
12.3 Transformée de Fourier rapide discrète (FFT) . . . . . 105
12.3.1 Transformée de Fourier et transformée discrète . . . . . 106
12.3.2 Algorithme de FFT : diviser pour résoudre . . . . . 107
12.3.3 Vers une bonne implantation . . . . . 108
12.3.4 Version itérative . . . . . 110
12.3.5 Application : produit de polynômes en $n \log(n)$ . . . . . 111
12.4 Algorithmes d'enveloppe convexe . . . . . 111
12.4.1 Algorithme de "l'enveloppe rapide" (diviser pour résoudre) . . . . . 112
12.4.2 Algorithme itératif de Jarvis . . . . . 113

**13 Programmation efficace des formules récursives 115**
13.1 Introduction . . . . . 115
13.2 Elimination des calculs redondants . . . . . 116
13.2.1 Exemple : Fibonacci . . . . . 116
13.2.2 Synthèse . . . . . 117
13.3 Exemples d'applications . . . . . 118
13.3.1 Combinaisons $C_n^p$ . . . . . 118

**14 Application : programmation dynamique 123**
14.1 Reconnaissance de chaînes de caractères bruitées . . . 123
14.2 Calcul d'un ABR optimal . . . 126
14.3 Sous-suites ascendantes (sujet d'examen corrigé) . . . 131
14.4 Produit de $n$ matrices . . . 136
14.5 Plus longue sous-séquence commune . . . 137
14.6 Modélisation et programmation dynamique . . . 137
14.6.1 Problèmes des cageots de fraises . . . 137
14.6.2 Sac à dos . . . 137
14.6.3 Jeu télévisé . . . 138
14.6.4 Fonction d'Ackermann . . . 138
14.7 Programmation du parcours d'un chariot automatisé . . . 138
14.7.1 Modélisation . . . 139
14.7.2 Résolution par la programmation dynamique . . . 140
14.7.3 Parcours du chariot . . . 141
14.7.4 Remarques sur l'efficacité de cet algorithme . . . 141

**15 Résolution par séparation et évaluation 143**
15.1 Arborescence des solutions possibles . . . 143
15.2 La méthode du "Branch and Bound" . . . 146
15.2.1 Principe de la méthode . . . 146
15.2.2 Choix du nœud actif à diviser . . . 147
15.3 Exemple du problème du sac à dos . . . 148
15.4 Relaxation de programmes linéaires en nombres entiers . . . 150
15.5 Un exemple d'ordonnancement de tâches . . . 151

**A Mesures sur les arbres 157**

**B Petit lexique d'algo-R.O. 159**

**C Formulaire de calcul de coûts 163**
C.1 Rappels sur les notations asymptotiques . . . 163
C.2 Théorème magique . . . 163
C.3 Découpe récursive à pas dynamique . . . 164

# Introduction

# Chapitre 1

# Analyse du coût d'un algorithme

Évaluer le coût (ou, par abus de langage, la complexité) d'un algorithme consiste à évaluer le nombre de ressources qu'il requiert : nombre d'opérations, espace mémoire, ... D'une part, cela permet de déterminer son champ d'application sur ordinateur. Par exemple :

- les algorithmes de tri en $O(n^2)$ sont inutilisables en pratique lorsque $n$ (taille du tableau à trier) dépasse $10^9$ ;
- l'invention de la transformée de Fourier rapide (FFT), qui fonctionne en $n \log(n)$, a révolutionné le domaine du traitement du signal ;
- la complexité des algorithmes est un handicap majeur pour la programmation des jeux, comme le jeu d'échec.

D'autre part, cela permet également de comparer, au niveau de leur coût, différents algorithmes qui résolvent un même problème.

Après un rappel des notions de coût dans le meilleur et dans le pire cas, ce chapitre définit la notion de complexité en moyenne. Différents exemples sont proposés. L'évaluation pratique sur ordinateur est illustrée sur un exemple. En annexe C, les formules classiques pour évaluer un coût sont rappelées.

## 1.1 Mesures de complexité

Si $D$ est l'ensemble des données d'un algorithme ou d'un programme, les mesures de complexité sont des fonction de type $D \to \mathbb{N} \cup \{\infty\}$, ou $D \to \mathbb{R} \cup \{\infty\}$. On peut penser à différentes manières de définir ces mesures :

- temps d'exécution,
- nombre d'opérations, c'est à dire d'exécutions de certaines primitives (par exemple nombre de comparaisons et nombre d'affectations pour un algorithme de tri),
- taille mémoire utilisée.

Comparer des algorithmes d'après leur temps d'exécution, fortement corrélé à la machine sur laquelle on fait le calcul, est difficile. Nous utiliserons dans le cadre de ce cours les complexités en nombre d'opérations et en place mémoire.





**Propriété :** Pour toute mesure de complexité $m$ d'un programme $p$, si $m(d) = \infty$ alors $p$ ne s'arrête pas pour la donnée $d$.

Remarquons que la réciproque de cette propriété, qui est vraie pour le temps d'exécution, est fausse pour la taille de la mémoire (un algorithme peut boucler tout en n'utilisant qu'une place mémoire finie).

## 1.2 Taille des données

La plupart des mesures de complexité dépendent des données sur lesquelles on exécute l'algorithme[1]. Cependant, en général, seule *une partie* des données est significative. Dans ce cas, elle seule doit être prise en compte.

Par exemple, l'opération de fusion utilisée dans l'algorithme de tri fusion a la particularité d'utiliser toujours le même nombre de comparaisons pour des tailles de tableaux à fusionner données. Le temps d'exécution du tri par fusion ne dépend donc que du nombre d'éléments du tableau initial à trier, et non pas de l'ordre de ces éléments. Cela simplifie de beaucoup le calcul de la complexité de cet algorithme, dans la mesure où les mesures de complexité deviennent des fonctions de $\mathbb{N}$ dans $\mathbb{N}$, au lieu d'être des fonctions de l'ensemble des permutations des objets à trier dans $\mathbb{N}$.

De manière générale, on s'intéresse dans la plupart des cas à évaluer la complexité d'un algorithme en fonction d'une notion de "taille des données" qui a été associée au problème à résoudre. Par exemple, cette taille est fonction de l'entier $n$ dans le calcul de factorielle($n$), et correspond à la longueur de la suite à trier pour les algorithmes de tri.

Même lorsqu'une taille a été définie sur les données, la complexité d'un algorithme n'admet pas toujours une expression simple. Ceci est illustré par l'exemple ci-dessous.

**Exemple :** tri par segmentation ("quicksort")

```
procedure  tri (inf, sup: entier)
  si  inf < sup  alors
   debut
      i: entier;
      segmentation(inf, sup, i); -- renvoie la place du pivot dans i
      tri(inf, i-1);
      tri(i+1, sup)
   fin
```

Calculons le nombre de comparaisons effectuées par cet algorithme pour trier un tableau de taille $n$, dans le pire et dans le meilleur des cas.

Solution: on remarque que le nombre de comparaisons lors de la segmentation est constant et égal à $n - 1$.

- si le tableau est déjà trié, $C(n) = n - 1 + C(n-1)$ D'où $C(n) = O(n^2)$ dans le cas le pire.

[1]On utilise parfois des mesures de complexité indépendantes des données, comme la longueur du programme (mesurée en nombre de lignes, ou d'instructions). Ces mesures concernent plutôt le génie logiciel que l'algorithmique, et ne seront pas étudiées dans le cadre de ce cours.

- si le tableau est tel qu'on le coupe "au milieu" à chaque étape (meilleur cas), alors $C(n) = 2C(n/2) + n - 1$, d'où $C(n) \sim n \log_2(n)$.

Pour cet exemple, la complexité du tri n'est donc pas uniquement fonction de la taille des données. En fait, on peut montrer que c'est une fonction de l'ensemble des permutations des objets à trier dans $\mathbb{N}$ qui n'admet pas d'expression analytique. D'où la nécessité de définir les notions de complexité dans le cas le pire, dans le cas le meilleur, et en moyenne.

**Exercice 1.1** *Rappeler l'algorithme de tri par insertion. Calculer son coût dans le pire et le meilleur cas.*
*Même question avec le tri par fusion. Que remarquez-vous ?*

## 1.3 Calcul de coûts par résolution d'équations aux récurrences

Très souvent, l'analyse du coût d'un algorithme conduit à une équation aux récurrences : calculer la sortie pour une entrée de taille $n$ se ramène à calculer la sortie pour des entrées de taille $m$ inférieure à $n$. On étudie ci-dessous deux cadres que l'on retrouve très souvent.

### 1.3.1 Rappels sur les notations asymptotiques

Soient $f(n)$ et $g(n)$ deux fonctions à valeurs positives.

- $f = \mathbf{O}(g) \iff \exists A > 0, \exists N_0 > 0 : \forall n > N_0 \quad f(n) \leq Ag(n)$
- $f = \mathbf{\Omega}(g) \iff \exists A > 0, \exists N_0 > 0 : \forall n > N_0 \quad f(n) \geq Ag(n)$
- $f = \mathbf{\Theta(g)} \iff \exists A, B > 0, \exists N_0 > 0 : \forall n > N_0 \quad Ag(n) \leq f(n) \leq Bg(n)$

Exemple: $12.345n^2 + 3.14n \log n + O(n) = \Theta(n^2)$

Les notations suivantes sont aussi utilisées parfois :

- $f = \mathbf{o}(g) \iff \forall C > 0, \exists N_0 > 0 : \forall n > N_0 \quad f(n) \leq Cg(n)$
  $\iff \frac{f(n)}{g(n)} \lim_{n\to\infty} 0$
- $f \sim_{n\to\infty} g \iff f(n) = g(n) + o(f(n))$. Par abus, on note $f \sim g$.

### 1.3.2 Algorithme itératif

Le calcul sur l'entrée de taille $n$ se ramène, avec $f(n)$ opérations au calcul sur une donnée de taille $n-1$. Le coût $C(n)$ de l'algorithme vérifie alors l'équation

$$C(n) = C(n-1) + f(n)$$

où $f$ est une fonction supposée croissante (cas pratique général). On en déduit :

$$C(n) = \sum_{i=0}^{n} f(i).$$

Souvent, $f$ est assez complexe (en particulier dû aux parties entières); on s'intéresse donc à un encadrement assez précis de cette somme. Aussi, plutôt que de travailler en



somme discrète, on se ramène à une approximation par intégration. Comme $f$ est croissante, on a :

$$\int_{i-1}^{i} f(t)dt \leq f(i) \leq \int_{i}^{i+1} f(t)dt.$$

D'où l'on déduit :

$$\int_{0}^{n-1} f(t)dt \leq C(n) \leq \int_{1}^{n+1} f(t)dt.$$

Souvent $f$ vérifie : $\int_0^{n-1} f(t) \sim \int_1^{n+1} f(t)dt$; on a alors :

$$C(n) \sim \int_{1}^{n+1} f(t)dt.$$

**Exemple :** Le tri par insertion d'un tableau de taille $n$ consiste à :

1. trier par insertion le tableau constitué des $n-1$ premiers éléments, en $C(n-1)$ opérations.
2. comparer l'élément de rang $n$ aux $(n-1)$ précédents jusqu'à trouver un supérieur et enfin décaler les suivants, d'où $f(n) = n$ opérations (comparaison ou décalage).

Le coût de cet algorithme est donc : $C(n) = C(n-1) + \Theta(n) \simeq \int_0^n t\,dt \simeq \frac{n^2}{2}$.

### 1.3.3 Algorithme récursif

Pour un algorithme récursif, le calcul sur l'entrée de taille $n$ se ramène, avec $f(n)$ opérations à $a$ calculs sur des données de taille $n/K$ ($K = 2$ ou 3 ou ...). Le coût $C(n)$ de l'algorithme vérifie alors l'équation

$$C(n) = a.C\left(\frac{n}{K}\right) + f(n)$$

et $C(1) = 0$ où $f$ est une fonction supposée croissante (cas pratique général).

On résout l'équation homogène associée: $H(n) = a.H\left(\frac{n}{K}\right)$ et $H(1) = 0$. D'où

$$H(n) = a^j.H(\left(\frac{n}{K^j}\right) = a^{\log_K n} = K^{\log_K a^{\log_K n}} = K^{\log_K n.\log_K a} = n^{\log_K a}.$$

Le résultat de l'équation avec second membre dépend du terme prépondérant: ce peut être par exemple le premier membre $a.C\left(\frac{n}{K}\right)$ (i.e. $n^{\log_K a}$), ou le second membre $f(n)$. On distingue les 3 cas suivants :

Cas 1. si $f(n) = O\left(n^{\log_K a}\right)^{1-\epsilon}$ (avec $\epsilon > 0$) : on a $C(n) = \Theta\left(n^{\log_K a}\right)$.

Cas 2. si $f(n) = \Theta\left(n^{\log_K a}\right)$ : on a $C(n) = \Theta\left(n^{\log_K a}.\log n\right)$.

Cas 3. si $f(n) = \Omega\left(n^{\log_K a}\right)^{1+\epsilon}$ (avec $\epsilon > 0$) et si $\exists n_0 : \forall n > n_0 : f(n) > a.f\left(\frac{n}{K}\right)$; alors on a $C(n) = \Theta\left(f(n)\right)$.

**Exemple :** Le tri par partition-fusion d'un tableau de taille $n$ consiste à :

1. trier récursivement les $n/2$ premiers éléments d'une part et les $n/2$ derniers d'autre part, en effectuant donc $2.C\left(\frac{n}{2}\right)$ opérations.
2. fusionner les 2 tableaux ainsi triés en effectuant au plus $f(n) = n$ comparaisons.

On en déduit : $C(n) \leq C(n/2) + n$. On a ici $K = a = 2$ d'où $n^{\log_K a} = n$. Comme $f(n) = n = \Theta\left(n^{\log_K a}\right)$, on est dans le cas 2, d'où

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

## 1.4 Cas le meilleur, cas le pire, complexité en moyenne

### 1.4.1 Définitions

Soit $f : D \to \mathbb{N} \cup \{\infty\}$ (ou bien $D \to \mathbb{R} \cup \{\infty\}$) la complexité de l'algorithme étudié (par exemple la complexité en nombre d'exécutions, ou en place mémoire). On définit trois nouvelles mesures de complexité de type $\mathbb{N} \to \mathbb{N} \cup \{\infty\}$ (ou $\mathbb{N} \to \mathbb{R} \cup \{\infty\}$) :

- Complexité dans le meilleur cas :
$$f_{\min}(n) = \min\{f(d) / |d| = n\}$$
- Complexité dans le pire cas :
$$f_{\max}(n) = \max\{f(d) / |d| = n\}$$
- Complexité en moyenne :
$$f_{\text{moyenne}}(n) = moyenne\{f(d) / |d| = n\}$$

### 1.4.2 Comment calculer une complexité en moyenne ?

La fonction $f_{moyenne}$ est définie comme une moyenne sur toutes les données de taille $n$. Il faut donc tenir compte de la répartition effective de ces données. On se place dans le cas où $f : D \to \mathbb{N}$, c'est-à-dire dans le cas de programmes qui s'arrêtent toujours. Soit $C(n)$ la complexité pour une donnée de taille $n$. On définit $q_{n,k}$ comme la probabilité pour que $C(n) = k$. La complexité en moyenne correspond alors à l'espérance mathématique de $C(n)$ :

$$f_{moyenne}(n) = \sum_{k \geq 0} k.q_{n,k}$$

Il est parfois plus simple d'exprimer $f_{moyenne}$ en fonction de la probabilité $p_{n,k}$ pour que $C(n) \geq k$. On a alors de manière évidente :

$$f_{moyenne}(n) = \sum_{k \geq 0} p_{n,k}$$

On fait souvent l'hypothèse "d'équiprobabilité", qui consiste à supposer que toutes les données de taille $n$ ont la même probabilité d'occurrence. Si le nombre de données de taille $n$ est fini, on a alors une troisième expression de $f_{moyenne}$ :

$$f_{moyenne}(n) = \frac{\sum_{|d|=n} f(d)}{\text{nb de données de taille n}}$$



## 1.5 Exemples de calculs de complexité en moyenne

### 1.5.1 Cas de la recherche séquentielle

**Exercice 1.2** *On considère la recherche séquentielle dans une liste à $n$ éléments non ordonnés. On est sûr que l'élément cherché est dans la liste, et que chaque élément a la même probabilité d'être recherché, à savoir $1/n$. Calculer la complexité moyenne de l'algorithme, la mesure de complexité choisie étant le nombre de comparaisons.*

Solution :

$$\begin{aligned} f_{\text{moyenne}}(n) &= \sum_{i=1}^{n} i \,.\, \text{proba}(C(n) = i) \\ &= \sum_{i=1}^{n} i \,.\, \text{proba}(x \text{ présent dans la case } i) \\ &= \sum_{i=1}^{n} i \,.\, \frac{1}{n} \\ &= \frac{1}{n} \,.\, \frac{n(n+1)}{2} = \frac{n+1}{2} \end{aligned}$$

### 1.5.2 Algorithme de la collection complète

**Exercice 1.3** *Pour promouvoir un article, une société offre des albums d'images à compléter. La collection complète est formée de $N$ images différentes. Une image est fournie en cadeau avec chaque article acheté. Combien d'achats faudra-t-il faire en moyenne pour avoir une collection complète ? (on supposera que les images sont uniformément distribuées sur les articles).*

**Indications :** La méthode à employer pour compléter une collection s'exprime de la manière suivante : tant qu'on n'a pas les $N$ images, on achète un article jusqu'à ce qu'on trouve une image que l'on n'a pas. On la colle dans l'album, et on s'arrête si l'album est complet. Cela correspond à l'algorithme :

```
perm: Tab(1..N);       -- numeros des images deja trouvees
pour i de 1 a N faire
  tant que x appartient a perm(1..i-1)
    tirage-aleatoire-uniforme(x); -- tirage de l'image x
  fin tant que ;
  perm(i) := x ;
fin pour ;
```

Soit $T(n)$ la complexité en moyenne de cet algorithme. On a $T(n) = \sum_{i=1}^{n} T_{i,n}$, où $T_{i,n}$ est le nombre moyen de tirages à l'étape $i$, pour obtenir un $x$ qui ne soit pas déjà dans le tableau perm.

Calculons les $T_{i,n}$, en utilisant la probabilité $p^i_j$ pour que le coût soit au moins $j$ à l'étape $i$ (au moins $j$ tirages nécessaires pour trouver un $x$ qui ne soit pas dans perm).

**Solution :** On a de façon évidente $T_{1,n} = 1$. D'après la seconde formulation de la complexité en moyenne donnée au paragraphe 1.4.2, $T_{i,n} = \sum_j p^i_j$, pour $i > 1$.

On obtient par récurrence :

$$\begin{aligned} p^i_1 &= 1 \\ p^i_{j+1} &= p^i_j.\frac{i-1}{n} \end{aligned}$$

(il y a en effet $i-1$ chances sur $N$ à chaque tirage pour tomber sur l'une des images que l'on a déjà). Donc $p^i_j = (\frac{i-1}{n})^{j-1}$.

$$\begin{aligned} T_{i,n} &= \sum_j p^i_j \\ &= \sum_j (\frac{i-1}{n})^{j-1} \\ &= \frac{n}{n-i+1} \end{aligned}$$

Le résultat cherché est donc :

$$T(n) = \sum_{i=1}^{n} T_{i,n} = n \sum_{i=1}^{n} \frac{1}{i} \sim n \log n$$

On doit donc en moyenne acheter $N \log N$ articles pour compléter sa collection.

### 1.5.3 Analyse en moyenne du tri par segmentation (quicksort)

Nous supposerons que tous les éléments du tableau à trier sont différents. Si le tableau a $n$ éléments, il y a $n!$ cas possibles, correspondant aux permutations de $1, 2, ..., n$. Notre évaluation en moyenne sera faite sous l'hypothèse que ces $n!$ cas ont la même probabilité d'occurrence.

On va calculer $K(n)$, nombre moyen de comparaisons pour trier un tableau à $n$ éléments. Soit $K_j(n)$ le nombre moyen de comparaisons pour trier un tableau à $n$ éléments, le pivot $A(inf)$ étant égal à $j$ ($1 \le j \le n$). D'après l'hypothèse d'équiprobabilité ci-dessus, on a :

$$K(n) = \frac{K_1(n) + K_2(n) + \ldots + K_n(n)}{n} \tag{1.1}$$

Pour trier un tableau $A$ à $n$ éléments ($n > 1$), dont le pivot est $j$, le nombre de comparaisons est :

$$K_j(n, A) = n - 1 + K(j-1, AG) + K(n-j, AD) \tag{1.2}$$

le terme $n-1$ correspondant au nombre de comparaisons lors de la segmentation.

$K_j(n)$ est égal à la valeur moyenne des $K(n, A)$ pour les $(n-1)!$ configurations possibles du tableau $A$. Pour calculer sa valeur, il suffit donc de connaître la valeur moyenne du second membre de l'égalité (1.2). Pour cela, on évalue la probabilité d'apparition de chaque configuration de $AG$ et $AD$. Nous ferons l'hypothèse que chacune des $j!$ configurations de $AG$ a la même probabilité d'occurrence, et de même pour chacune des $(n-j)!$ configurations de $AD$. Cette propriété peut être prouvée pour toute programmation "raisonnable" de la procédure de segmentation.



Les valeurs moyennes de $K(j-1, AG)$ et $K(n-j, AD)$ sont donc respectivement $K(j-1)$ et $K(n-j)$. D'où :

$$K_j(n) = n - 1 + K(j-1) + K(n-j) \quad \text{pour } n > 1$$

D'où, d'après l'équation (1.1) :

$$\begin{aligned} K(n) &= n - 1 + \frac{2}{n}\sum_{j=0}^{n-1} K(j) \\ K(0) &= K(1) = 0 \end{aligned}$$

**Résolution de l'équation de récurrence :** L'équation précédente peut s'écrire

$$nK(n) = n(n-1) + 2\sum_{j=0}^{n-1} K(j)$$

et de même :

$$(n-1)K(n-1) = (n-1)(n-2) + 2\sum_{j=0}^{n-2} K(j)$$

On obtient, à partir de ces deux égalités, par soustraction :

$$nK(n) - (n+1)K(n-1) = 2(n-1)$$

ou encore :

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

En posant :

$$x_n = \frac{K(n)}{n+1}$$

l'équation récurrente devient :

$$\begin{aligned} x_n - x_{n-1} &= \frac{2(n-1)}{n(n+1)} \\ x_0 = x_1 &= 0 \end{aligned}$$

La solution de cette équation est :

$$x_n = \sum_{j=1}^{n} \frac{2(j-1)}{j(j+1)}$$

c'est-à-dire :

$$x_n = \left(2\sum_{j=1}^{n} \frac{1}{j}\right) - 4 + \frac{4}{n+1}$$

On en déduit

$$\begin{aligned} K(n) &= \left(2(n+1)\sum_{j=1}^{n} \frac{1}{j}\right) - 4n \\ K(0) &= K(1) = 0 \end{aligned}$$

Or, d'après la formule d'Euler, le nombre harmonique $H_n = \sum_{j=1}^{n} \frac{1}{j}$ est égal à $ln(n) + g + e$ (où $g$ est la constante d'Euler et $e$ tend vers zéro quand $n$ tend vers l'infini). Le nombre de comparaisons moyen est donc de l'ordre de $2n \log n$.

**Remarque 1** *Les termes d'ordre le plus élevé de $K_{min}(n)$ (nombre de comparaisons dans le cas le plus favorable) et de $K(n)$ sont respectivement $n\log_2(n)$ et $2nln(n)$. Le rapport $\frac{K(n)}{Kmin(n)}$ tend donc vers $2ln(2)$, soit 1,386. Le nombre de comparaisons en moyenne est donc très proche du nombre de comparaisons du cas le plus favorable. C'est ce qui fait tout l'intérêt de cet algorithme, le cas le pire étant en $O(n^2)$, comme nous l'avons vu plus haut.*

### 1.5.4 Un calcul de maximum

**Exercice 1.4** *On considère la fonction suivante, qui calcule l'élément maximal d'un vecteur $T$. La fonction est écrite en C, l'appel initial est valmax(k) où k est l'indice du dernier élément de T :*

```
int valmax(int i)
 {
  if(i==0)
    return T[0];
   else {
         if(T[i}>valmax(i-1))
           return T[i];
          else return valmax(i-1);
        }
 }
```

*Un utilisateur, qui teste cette fonction sur des vecteurs de tailles croissantes, vous dit : "je ne comprends pas, la fonction marche jusqu'à 27 éléments, mais semble boucler pour 28". Comment interprétez-vous ce comportement ? Que faut-il faire pour y remédier ?*

**Solution :** en fait, la complexité de cette fonction est de $n$ dans le cas le plus favorable (vecteur trié en ordre croissant), car il n'y a alors qu'un appel secondaire à `max` par exécution. Par contre, dans le cas le plus défavorable (vecteur trié en ordre décroissant), $2^n$ appels seront effectués. C'est ce qui a dû se produire au cours des tests effectués par l'utilisateur : les premiers tests ont du tomber sur des configurations pas trop mauvaises. Par contre, si le cas d'une complexité exponentielle se produit pour $n = 28$, le temps de calcul est tel que le programme semble boucler.

La solution consiste à utiliser une variable locale pour le calcul de `valmax(i-1)`.

**Remarque 2** *On se trouve ici dans un cas trivial de calcul redondant, qui est, au fond, un défi pour nos compilateurs. Pourquoi un compilateur qui optimise l'emploi des registres n'optimiserait pas aussi la gestion des appels de fonctions? Il devrait juste savoir que l'appel de valmax est sans effet de bord.*

### 1.5.5 Calcul des deux éléments maximaux

**Exercice 1.5** *On désire trouver les deux éléments les plus grands d'un tableau $T$. On supposera que les données sont équidistribuées. Comparer la complexité des deux versions* `algo1` *et* `algo2` *ci-dessous réalisant ce calcul. Que peut-on en conclure ?*



```
void algo1 ( const int n, const long* x, long& m1, long& m2 ) {
  // Imput : n : precondition  n>=2 ;
  //         x : precondition x[0] ... x[n-1] exist.
  // Output : m1 = first maximum and m2 = second maximum
  //Implem  : if (x[i] > m1) { m1 := x[i] ; } else if (x[i] > m2) then { ...}
  long max1 = x[n-1] ;
  long max2 = x[n-2] ;

  if (max1 < max2) {
    long aux = max1 ;
    max1 = max2 ;
    max2 = aux ;
  }

  for ( int i=n-3; i>=0; --i ) {
    if ( x[i] >= max1 ) {
       max2 = max1 ;
       max1 = x[i] ;
    }
    else if ( x[i] > max2 ) {
       max2 = x[i] ;
    }
  }

  m1 = max1 ;
  m2 = max2 ;
}
```

La seconde version `algo2` consiste à remplacer la partie intérieure à la boucle par :

```
if ( x[i] > max2 ) {
  if ( x[i] >= max1 ) {
    max2 = max1 ;
    max1 = x[i] ;
  }
  else {
    max2 = x[i] ;
  }
}
```

**Solution :** La complexité dans le cas le pire est la même pour les deux versions de l'algorithme, et correspond au cas où le tableau est trié en ordre décroissant. On a alors $2(n-2)+1 = 2n-3$ comparaisons.

Complexité en moyenne :

- Version 1 : Le second test de la boucle est fait dans les cas où $x(i) < \max_1$, soit avec une probabilité $\frac{i-1}{i}$ (il y a en effet $i-1$ chances sur $i$ pour que le plus grand de $i$ éléments ne soit pas placé en dernier).

  En tenant compte des $n-2$ occurrences du premier test de la boucle, on a :

$$\begin{aligned} C_{moy}(n) &= 1 + n - 2 + \sum_{i=3}^{n} \frac{i-1}{i} \\ &= 2n - 2 - H_n + \frac{1}{2} \end{aligned}$$

où $H_n = \sum_{i=1}^{n} \frac{1}{i} \sim log(n)$

- Version 2 : Évaluons la probabilité pour que le second test de la boucle soit effectué : C'est le cas soit si $x(i)$ est le premier maximum parmi les $i$ premiers éléments du tableau, soit s'il est le second. Or, ces évènements ont chacun une probabilité de $1/i$, d'où :

$$\begin{aligned} C_{moy}(n) &= 1 + n - 2 + \sum_{i=3}^{n} \frac{2}{i} \\ &= n - 4 + 2H_n \end{aligned}$$

Pour des grandes valeurs de $n$, la version 2 de l'algorithme tend donc à faire deux fois moins de comparaisons que la première. Si la fonction de comparaison est coûteuse, la version 2 sera donc environ deux fois plus efficace. Cela illustre le fait que, même pour de petits programmes très simples, il est judicieux de bien choisir l'ordre des instructions!

D'un point de vue pratique, les programmes effectuant non seulement des comparaisons mais aussi des affectations, tout dépend du coût de la comparaison et de l'architecture (et du compilateur) utilisée. Le tableau 1.1 compare les performances pratiques des deux programmes sur la plateforme suivante :

- Machine hardware: sun4m
- OS version: SOLARIS 5.5
- Processor type: sparc
- Hardware: sun4m
- Compilateur: gcc version 2.8.0

Le temps du programme 1 est alors environ 1.3 celui du programme 2, (à comparer au facteur 2 en nombre de comparaisons).

| Taille tableau | temps cpu Algo1 | temps cpu Algo2 | Ratio Algo1/Algo2 |
|---|---|---|---|
| 10 | 0.00022 | 0.00021 | 1.11 |
| 50 | 0.00076 | 0.00066 | 1.22 |
| 100 | 0.0013 | 0.00098 | 1.35 |
| 1000 | 0.012 | 0.0088 | 1.37 |
| 10000 | 0.13 | 0.097 | 1.35 |
| 100000 | 1.46 | 1.12 | 1.30 |
| 1000000 | 19.2 | 15.6 | 1.23 |

Figure 1.1: Résultats en moyenne sur 10 exécutions de chaque algorithme. Chaque exécution est faite sur un nouveau tableau dont les éléments sont générés par `rand`. Pour chaque tableau le temps est mesuré sur 100 exécutions.



## 1.6 Coût amorti

Pour certains algorithmes, le coût le plus pertinent n'est pas le coût en pire cas ou en moyenne, mais le coût amorti. Cette notion est similaire à la notion d'amortissement pour un investissement industriel: le surcoût $c$ d'une dépense unique (par exemple coût d'achat ou coût de design) est divisé par le nombre $M$ d'unités produites et un "coût amorti" $\frac{c}{M}$ est attribué à chaque unité produite.

Généralement, le coût amorti est utile pour analyser un algorithme qui est itéré un grand nombre de fois et pour lequel le coût de chaque exécution peut varier fortement. Dans ce cas, le coût total $T_n(M)$ pris par $M$ exécutions consécutives peut fournir une estimation plus précise que simplement $M$ fois le coût en pire cas.

### 1.6.1 Définition.

Soit $M$ exécutions consécutives de l'opération $x_k := A(x_{k-1})$ pour $0 < k \leq M$. Soit $n$ la taille de $x_0$ et $t_k$ le coût de l'exécution $x_{k+1} := A(x_k)$. On appelle *coût amorti* des $M$ exécutions de $A$ la quantité:

$$C_n(M) = \frac{\sum_{k=1}^{M} t_k}{M}.$$

Par exemple, on dit que l'on a un coût amorti en $\Theta(\log n)$ si le coût de $M$ opérations consécutives ($M$ asymptotiquement grand) est $\Theta(M \log n)$.

Autrement dit, on autorise des opérations coûteuses à condition qu'elles soient rares et que cela permette d'optimiser les opérations les plus fréquentes.

**Exemple: gestion d'une pile.** Considérons par exemple une `pile` qui est implémentée par un tableau. L'opération `empiler` prend un coût variable :

- constant si il reste de la place dans le tableau (par exemple si l'opération `empiler` a lieu juste après une opération `dépiler`);
- linéaire si le tableau est plein. Il faut en effet alors allouer un tableau plus grand, et recopier les éléments dedans.

Le coût de l'opération `empiler` est fortement variable selon le contexte; il est donc plus pertinent d'évaluer le coût amorti de l'opération en considérant $n$ opérations `empiler` consécutives.

Ainsi, considérons l'implémentation suivante de `empiler` : Lorsque la zone mémoire allouée pour le tableau $t$ est pleine, on réalloue un tableau $t'$ 2 fois plus grand; on recopie ensuite tous les éléments de $t$ dans $t'$. Avec cette implémentation, une opération "empiler" prend en pire cas un temps linéaire $O(n)$, lorsque les éléments ont été recopiés dans un tableau 2 fois plus grand; mais alors, les $n-1$ opérations "empiler" suivantes prennent toutes un temps constant $O(1)$ (meilleur cas). Le coût amorti est donc : $\frac{0(n)+(n-1).O(1)}{n} = O(1)$.

**Remarque 3** *Bien que le coût amorti soit une moyenne, il n'y a aucune d'hypothèse sur la loi de distribution des entrées comme c'est le cas pour un coût en moyenne (pour lequel on suppose par exemple l'équiprobabilité des entrées) .*

### 1.6.2 Implantation amortie d'un tableau de taille variable

Un tableau $T$ de taille variable fournit les opérations suivantes (en plus de l'accès en temps constant à l'élément $T(i)$ d'indice $i$ :

- ajout d'un élément en fin de tableau;
- suppression du dernier élément en fin de tableau;

Le programme ci-dessous implémente cette structure de données (en C++); les opérations d'ajout et de suppression sont de coût amorti constant $O(1)$; de plus la place mémoire occupée physiquement par le tableau est proportionnelle au nombre d'éléments du tableau (i.e. insérés et non détruits).

```
template< class Elt >
class tableau {  // Implantation d'un tableau d'elements de type Elt
  private :
     Elt* tab;  // Le tableau physique qui contient les elements de la pile.
     int nmax;  // Taille du tableau tab.
     int ncour; // (ncour-1) est l'indice du dernier element insere dans le tableau

     void realloctab ( int size) {
           // reallocation du tableau avec ncour eltis dans un tableau de taille size
           // Precondition : size > ncour
           Elt* newtab = new Elt [ size ] ; // allocation du nouveau tableau
           for (int i=0; i < ncour; i++) newtab[i] = tab[i]; // recopie des elts
           delete tab [] ;
           tab = newtab ;
           nmax = size;
      }

  public :
     pile() {  ncour = 0 ; nmax = 0 ; tab = 0 ; } // Initialisation i: le tableau est vide.

     int NbElements() { return  ncour ; }

     Elt& operator()(int i)  { // Opération d'acces a l'element d'indice i
       assert ( (i >=0) && (i < ncour) ) ; // pre-condition
       return tab[i] ;
     }

     AjoutFin (const Elt x) {
       if (nmax == 0) { // le tableau n'a pas ete alloue: allocation
             nmax = 1 ; // taille initiale; par exemple 1.
             tab = new Elt [ nmax ] ;
       }
       else if (ncour == nmax) {
           // Il n'y a plus de place dans le tableau: on realloue dans un tableau 2 fois plus grand
           realloctab( 2*nmax ) ;
       }
       tab[ncour] = x ;
       ncour += 1 ;
     }

     Elt SuppressionDernier () { // On suppose ncour > 0
       if ( ncour < nmax/4 ) { // Reallocation de la pile dans un tableau plus petit
         realloctab( nmax/2 ) ;
```



```
       }
       ncour -= 1 ;
       return tab [ ncour ] ;
     }
}
```

Le nombre $n$ d'éléments accessibles dans le tableau est: $n$ = `ncour` = #`AjoutFin` - #`SuppressionDernier`. Avec le programme ci-dessus, la place mémoire physique "nmax" (tableau "tab") est liée à $n$ :

$$\frac{\text{nmax}}{4} \leq n \leq \text{nmax}$$

La place mémoire est donc $\Theta(n)$ où $n$ est le nombre d'éléments dans le tableau.

De plus, on vérifie facilement que toute opération "AjoutFin" qui effectue une réallocation d'un tableau de taille $M$ est nécessairement au moins précédée de $M/2$ opérations "AjoutFin" sans réallocation. Le coût amorti de "AjoutFin" est donc $\Theta(1)$.
De même, toute opération "SuppressionDernier" qui effectue une opération de réallocation dans un tableau de taille $M/2$ est nécessairement suivie d'au moins $M/4$ opérations "SuppressionDernier" qui se feront sans déallocation. Le coût amorti de "SuppressionDernier" est donc aussi $\Theta(1)$.
**Remarque.** Le choix des facteurs d'expansion $\alpha = 2$ (en cas de rallongement du tableau) et de compression $\beta = \frac{1}{4}$ (en cas de raccourcissement) est arbitraire. D'un point de vue théorique, il suffit pour avoir un coût amorti $O(1)$ que $\alpha > 1$ et $\alpha.\beta < 1$. Le choix précédent est "simple"; mais on n'aurait pu aussi bien choisir $\alpha = 1.5$ et $\beta = 0.6$ par exemple. En pratique, les choix de taille d'allocation initiale (ici nmax=1) et les facteurs $\alpha$ et $\beta$ peuvent être paramétrés pour être adaptés au cadre d'utilisation, en fonction de compromis temps et mémoire.

**Exercice 1.6** *Écrire deux programmes implémentant une pile (LIFO) non bornée d'entiers telle que toutes les opérations (empiler, dépiler, pilevide) soient de coût amorti constant :*

- *le premier basé sur une liste;*
- *le deuxième basé sur un tableau avec réallocation dynamique.*

*Quelle version est la plus performante en pratique ?*

**Exercice 1.7** *La recherche dichotomique dans un tableau trié de $n$ éléments prend un temps logarithmique, mais l'insertion un temps linéaire. Pour diminuer le coût de l'insertion, on représente le tableau de la façon suivante.*
*Soit $k = \lceil \log_2(n+1) \rceil$ et $n = \sum_{i=0}^{k-1} n_i 2^i$ la décomposition de $n$ en base 2 (i.e. $n_i$ vaut 0 ou 1).*
*Les $n$ éléments triés sont stockés dans $k$ tableaux $A_0, \ldots, A_{k-1}$. Le tableau $A_i$ est de longueur $2^i$ si $n_i = 1$; il est vide si $n_i = 0$. Chaque tableau $A_i$ est trié; mais il n'y a pas d'ordre entre deux tableaux.*
*Exemple : pour 13 caractères dans l'ordre alphabétique, on a 4 tableaux :*

$$A_0 = [e] \quad A_1 = [,] \quad A_2 = [a, c, i, l] \quad A_3 = [b, d, f, g, h, j, k]$$

*Pour l'implantation, les $k$ sous-tableaux $A_i$ sont stockés consécutivement dans un tableau dynamique $T$ de taille $2^k - 1$, donc inférieure à $2n$.*

1. *Donner un algorithme de recherche; analyser le coût en pire cas.*
2. *Analyser le coût en moyenne lors de la recherche d'un élément qui est dans le tableau.*
3. *Donner un algorithme d'insertion. Analyser le coût en pire cas et le coût amorti.*
4. *Étudier la destruction d'un élément.*
5. *On propose l'algorithme de tri suivant, inspiré d'un tri par insertion; il utilise cette structure de données $T$ qui stocke de manière contiguë $T = [A_0, A_1, \ldots]$, chaque sous-tableau $A_i$ de taille $2^i$ étant toujours soit vide soit trié :*

```
T = vide ; // initialisation de T
// Insertion des éléments dans la structure    for (int i=0; i < n; i++)
T.insert ( x(i) ) ;
// Fin de l'insertion; on finalise en fusionnant les tableaux 2 à 2 pour terminer.
int k = ⌈log2(n+1)⌉ ;
for (int j=1; j < k ; j++ ) merge( T[0...2^(j-1) - 1 ], T[ 2^j...2^j - 1] ) ;
T.resize( n) ; // pour réajuster la taille de T
return T ;
```

*L'opération "* `merge( T[0...2^(j-1) - 1 ], T[ 2^j...2^j - 1] )` *" est la fusion classique de deux tableaux (non nécessairement pleins) avec $m_1 \leq 2^{j-1}$ et $m_2 \leq 2^j$. En résultat, les $m_1 + m_2$ éléments sont stockés triés dans le sous tableau $T[0 \ldots m_1 + m_2 - 1]$.*

   (a) *Quel est le coût de la phase d'insertion ?*

   (b) *Quel est le coût de la phase de fusion ?*

**Remarque** *: Ce tri, assez proche d'un tri par insertion, est donc de coût $O(n \log n)$. La recherche dichotomique dans un tableau trié étant en $O(\log n)$, c'est le décalage des éléments dans un tableau de taille $n$ qui entraînent un coût $O(n^2)$ pour le tri par insertion naïf. La représentation ci-dessus permet de conserver un coût total $O(n \log n)$ grâce à un tableau de taille raisonnable, inférieure à $2n$. Cependant, le tableau $T$ ne restant pas trié tout au long de l'algorithme, il ne peut pas être considéré comme un vrai tri par insertion. D'autres représentations d'un tableau, qui utilisent comme ci-dessus des emplacements vides pour amortir le coût des décalages nécessaires, permettent d'implanter un tri par insertion dans un tableau de taille $O(n)$ en coût amorti $O(n \log n)$.*

**Correction:**

1. *Recherche: on recherche successivement dans les $\log_2 n$ tableaux $A_i$ pour $i = k - 1, \ldots, 0$, en commençant par le tableau de plus grande taille;*
   *dans chaque tableau $A_i$ avec $n_i \neq 0$, on effectue une recherche dichotomique de coût $\log_2 \|A_i\| = i$.*
   *Le coût en pire cas est majoré par $\sum_{i=0}^{\log n} i = O(\log^2 n)$.*

2. *Analyse en moyenne de la recherche d'un élément dans le tableau. On se place dans le pire cas où $n = 2^k - 1$, i.e. on a $n_i = 1$ pour tout $i$.*
   *On suppose que l'élément recherché est choisi uniformément parmi les $n$ éléments;*



*la probabilité que l'élément soit dans le tableau $A_i$ est alors $\frac{2^i}{n} \leq 2^{i-k}$. D'où le coût $c_R(n = 2^k - 1)$ en moyenne:*

$$
\begin{aligned}
c_R(n) &\leq Coût_{RechDicho}(A_{k-1}) + \frac{1}{2}.c_R(2^{k-1} - 1) &(1.3)\\
&\leq \log_2 2^{k-1} + \frac{1}{2}\log_2 2^{k-2} + \ldots + \frac{1}{2^i}\log_2 2^{k-i-1} + \ldots + \frac{1}{2^{k-1}} &(1.4)\\
&\leq 2\log_2 n. &(1.5)
\end{aligned}
$$

*Le coût en moyenne est donc $\Theta(\log_2 n)$ si l'on fait la recherche dans l'ordre : d'abord dans $A_{k-1}$, puis dans $A_{k-2}$ etc jusqu'à trouver l'élément.*

3. *Pour l'insertion: soit $j$ tel que $n_j = 0$ et $n_i = 1$ pour tout $i = 0 \ldots j-1$. L'insertion consiste alors à insérer le nouvel élément dans $A_j$; puis à réorganiser la structure de donnée en supprimant les tableaux $A_0$, ..., $A_{j-1}$ pour les déplacer dans $A_j$.*
   *Pour cela, on procède comme suit: on fusionne $A_j$ successivement avec $A_0$ puis $A_1$, etc jusqu'à $A_{j-1}$. Le coût en comparaison de la fusion de deux tableaux de taille $2^i$ est majoré par $2^{i+1}$; le coût de ces fusions successives est donc majoré par $\sum_{i=0}^{j-1} 2^{i+1} \leq 2^j - 1$.*
   *On incrémente alors $n$ de 1: ainsi on a $n_i = 0$ pour $0 \leq i < j$ et $n_j = 1$; on détruit donc – virtuellement – les tableaux $A_i$ devenus inutiles.*
   *Le pire cas est lorsque $j = k - 1$ : dans ce cas, la réorganisation est de coût $O(n)$. Mais alors, on a ensuite $n_0 = 0$ et la prochaine insertion se fera avec 0 comparaisons, en coût $O(1)$. Plus précisément, le coût pour $n$ insertions consécutives est le suivant:*

   - $\frac{n}{2}$ *fois, on a $j = 0$ et l'insertion se fait avec $2^0 - 1 = 0$ comparaisons;*
   - $\frac{n}{2^2}$ *fois, on a $j = 1$ et l'insertion se fait avec $2^1 - 1 = 1$ comparaisons;*
   - $\frac{n}{2^3}$ *fois, on a $j = 2$ et l'insertion se fait avec $2^2 - 1 = 3$ comparaisons;*
   - $\frac{n}{2^i}$ *fois, on a $j = i$ et l'insertion se fait avec $2^i - 1$ comparaisons;*
   - ...
   - *1 fois seulement on a $j = \log_2 n$ et l'insertion se fait avec $n - 1$ comparaisons;*

   *On en déduit le coût amorti pour $n$ insertions (consécutives):*

$$c(n) \leq \frac{\sum_{j=0}^{\log_2 n} \frac{n}{2^{j+1}}.2^j}{n} = O(\log_2 n).$$

   *Le coût amorti de l'insertion est alors logarithmique, comme le coût moyen de la recherche.*

4. *Pour analyser la destruction, supposons que l'élément détruit est dans le tableau $A_d$. Soit de plus $j \leq d$ l'indice tel que $n_j = 1$ et $n_i = 0$ pour $0 \leq i < j$*
   *Pour détruire l'élément, on procède comme suit.*

   - *Étape 1 :* Réorganisation de $A_j$ *: on enlève le dernier élément $y$ dans $A_j$ et on déplace les $2^j - 1$ éléments restants de $A_j$ dans les tableaux $A_0, \ldots, A_j - 1$. Le coût est $O(2^j)$.*

- *Étape 2 :* Insertion dans $A_d$ *: on remplace l'élément $x$ par $y$ et on rétablit l'ordre dans le tableau $A_d$ en permutant l'élément $y$ soit à gauche, soit à droite jusqu'à ce qu'il soit en place. Le coût est $O(2^d)$.*
- *Finalement, on décrémente $n$ de 1 : ainsi $n_i = 1$ pour $0 \leq i < j$ et $n_j = 0$.*

*Le coût de la destruction est donc linéaire $O(n)$ en pire cas.*
*Analyse amortie : le coût de la réorganisation du tableau $A_j$ peut être amorti : il est nul lorsque $j = 0$, i.e. une fois sur deux. Cependant, l'insertion dans le tableau $A_d$ trié est de coût linéaire et n'est pas amortie : une fois sur deux, $d = k$ et l'insertion est de coût moyen $n/4$. Donc, le coût amorti de la destruction est aussi $O(n)$.*
*En conclusion, cette structure est bien adaptée à la recherche et l'insertion (coût amorti $O(\log n)$); mais elle n'est pas adaptée à la destruction.*
*Les structures de dictionnaire (chapitre 4) permettent de garantir un temps d'insertion, de recherche et de suppression de coût $\Theta(\log n)$.*

5. (a) *L'insertion de l'élément $i$ étant de coût amorti $O(\log i)$, le coût de la phase d'insertion est $\sum_{i=1}^{n} \log i \equiv \int_1^n \log t = [t \log t - t]_1^n \equiv n \log n$. Donc $O(n \log n)$.*

   (b) *Quel est le coût de la phase de fusion ? Le coût de la fusion "`merge( T[`$0 \dots 2^{j-1} - 1$`], T[`$2^j \dots 2^j - 1$`] )`" est $2^j$ comparaisons, donc $\Theta(2^j)$. La phase de fusion est donc de coût total $\sum_{j=1}^{\log_2 n} 2^j \equiv 2n$, donc $\Theta(n)$.*

*Le coût de cette variante du tri par insertion est donc $\Theta(n \log n)$.*

# Structures de Données

# Chapitre 2

# Files d'attente avec priorité

Rappelons qu'une file d'attente avec priorité (type `File`) est une structure permettant de stocker un certain nombre d'objets (type `Obj`) dont chacun est muni d'une valeur de priorité (type `Pr`). On suppose le type `Pr` muni d'une relation d'ordre. Les opérations usuelles sont l'ajout d'un élément et la suppression d'un élément de priorité maximale. Plusieurs options sont possibles : pour l'opération ajout, on peut accepter ou non les répétitions d'un même élément ; pour l'opération suppression on peut imposer une stratégie pour choisir entre des éléments de même priorité - par exemple prendre le plus ancien dans la file - ou au contraire laisser ce choix au programmeur. Dans ce chapitre, les répétitions d'objets sont acceptées et on n'impose pas de stratégie de choix à priorité égale.

## 2.1 Représentations simples

### 2.1.1 Ensemble fini de valeurs de priorité

On peut utiliser une structure table de hachage, les éléments étant rangés dans différentes listes en fonction de leur valeur de priorité (voir figure 2.1).

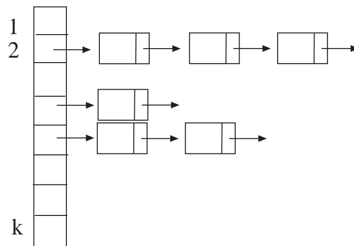


Figure 2.1: Représentation d'une file de priorité construite sur un ensemble de priorités fini.

Etudions les coûts des différentes opérations, en supposant qu'il y ait $k$ valeurs de priorité possibles et $n$ éléments rangés dans la file :

- `file-vide` $O(k)$





- `ajout` $O(1)$ si on insère en tête de liste. On a alors une pile à priorités égales. Il est facile de modifier la structure pour avoir une file d'attente à priorités égales avec l'ajout en temps $O(1)$.
- `premier-ob`, `premiere-pr`, `reste` : $O(k)$

La place mémoire utilisée est $k + 2n$.

On peut proposer des améliorations à la structure (faible augmentation de la place mémoire) pour diminuer le coût des opérations :

- Ajout d'une case donnant la valeur courante de la plus forte priorité ($-\infty$ si la file est vide). `file-vide`, `premier-ob`, et `premiere-pr` s'effectuent alors en temps constant. Par contre, l'opération `reste` est toujours en $O(k)$ dans le cas le pire, puisque cette case supplémentaire doit être mise à jour. se fait alors en $O(1)$.

La place mémoire utilisée après ces améliorations est de $2k+1+2n$, et toutes les opérations se font en temps constant sauf `reste` qui est en $O(k)$. Cette implémentation est donc particulièrement intéressante lorsque $k$ est très petit devant $n$.

### 2.1.2 Ensemble infini de valeurs de priorité

On peut essayer de conserver le même type de représentation. Cependant, les accès aux listes non vides d'éléments qui ont la même priorité doivent maintenant être chaînés entre eux (voir figure 2.2).

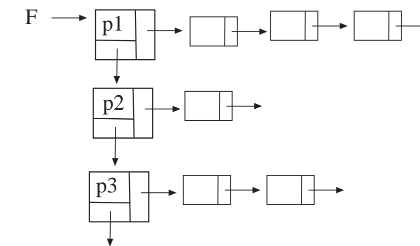


Figure 2.2: File de priorité construite sur un ensemble de priorités infini.

Les opérations `file-vide`, `premier-ob`, `premiere-pr` et `reste` s'effectuent en $O(1)$. Cependant, l'ajout se fait forcément en $O(k)$, $k$ étant le nombre de priorités différentes couramment présentes dans la file (en effet, un parcours est maintenant nécessaire pour trouver la bonne liste). Comme $k$ peut être, en pratique, du même ordre de grandeur que $n$, la structure perd beaucoup de son intérêt.

## 2.2 Représentation par tas

L'implémentation des files de priorité que nous allons maintenant étudier est beaucoup plus intéressante que la précédente lorsque $n$ est grand et que l'on n'a pas d'information particulière sur les valeurs de priorités. En effet, le tas [1] assure que toutes les opérations de la file s'effectueront au maximum en $O(log(n))$.

[1] "heap" en anglais.

### 2.2.1 Arbres parfaits partiellement ordonnés

**Définitions :**

**Arbre partiellement ordonné :** arbre dont la valeur en tout noeud est supérieure ou égale à celles de ses fils.

**Arbre parfait :** Arbre binaire dont les feuilles sont sur deux niveaux seulement, dont l'avant dernier niveau est complet, et dont les feuilles du dernier niveau sont groupées le plus à gauche possible.

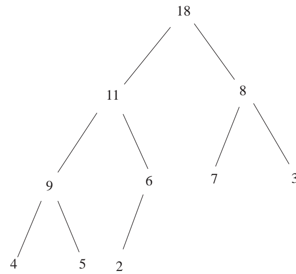


Figure 2.3: Arbre parfait partiellement ordonné.

La structure d'arbre parfait partiellement ordonné est bien adaptée à l'implantation d'une file de priorité. En effet, le fait que l'arbre soit partiellement ordonné facilite l'accès au maximum, toujours stocké dans la racine. De plus, un arbre parfait reste toujours bien équilibré, comme le montre la propriété qui suit.

**Propriété :** la hauteur d'un arbre parfait à $n$ noeuds est $\log_2(n)$.

Démonstration : décomposons $n$ en sommant le nombre de noeuds par niveaux.

$$n = \sum_{i=0}^{h-1} 2^i + reste = \frac{2^h - 1}{2 - 1} + reste = 2^h - 1 + reste$$

Or, $reste - 1 < 2^h$, d'où $2^h \leq n < 2^{h+1}$. On en déduit que $h$ est de l'ordre de $\log(n)$.

**Ajout dans un arbre parfait partiellement ordonné :**

- On ajoute l'élément sur la première feuille libre, de manière à ce que l'arbre reste parfait.
- On rétablit la propriété "partiellement ordonné" en échangeant l'élément avec son père si la valeur de ce dernier est inférieure, et en propageant ce type d'échange aussi loin qu'il le faut vers la racine (voir Figure 2.4).

Le coût de cet algorithme (en nombre de comparaisons ou d'échanges) est égal au pire à la hauteur de l'arbre, soit $\log_2(n)$.



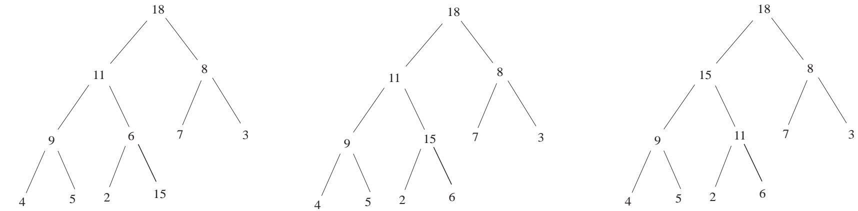


Figure 2.4: Ajout de 15 dans l'arbre de la figure précédente.

**Suppression du maximum :**

- On remplace l'élément racine, qui contient le maximum, par le dernier élément (dernière feuille) de manière à ce que l'arbre reste parfait.
- On rétablit la propriété "partiellement ordonné" en effectuant si nécessaire des échanges de pères avec le plus grand de leurs fils, en partant de la racine de l'arbre et en descendant (voir figure 2.5).

Comme pour l'ajout, le coût de cet algorithme est au pire égal à la hauteur de l'arbre, soit $\log_2(n)$.

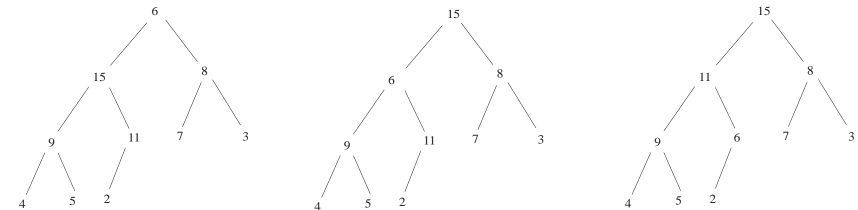


Figure 2.5: Suppression du maximum dans l'arbre précédent.

**Remarque 4** *Un arbre parfait partiellement ordonné, muni des opérations* `ajout` *et* `suppression` *décrites ci-dessus, ne se comporte pas comme une file d'attente à priorités égales. En effet, comme le montre la figure 2.6, deux éléments de même priorité peuvent ressortir dans n'importe quel ordre de la file d'attente, selon la configuration de l'arbre. Ceci illustre l'une des lacunes des spécifications algébriques; en effet, il n'est pas possible de trouver une formulation des axiomes qui permettent une spécification non déterministe de l'opération de suppression. Par contre il n'y a pas de difficulté pour exprimer une opération non déterministe avec une spécification ensembliste.*

### 2.2.2 Représentation par tas

En pratique, les arbres parfaits partiellement ordonnés ne sont pas implémentés par une structure d'arbre usuelle, mais à l'aide d'un tableau, appelé **tas**, qui permet un accès facile au père et aux fils d'un noeud. Pour cela, les éléments sont rangés par niveau, de haut en bas, comme l'illustre la figure 2.7.

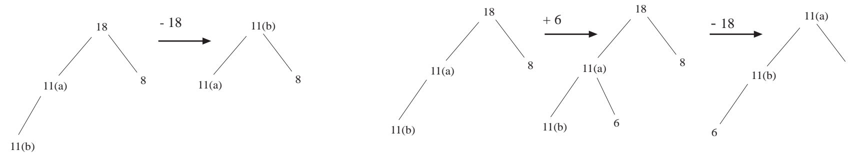


Figure 2.6: Un arbre parfait partiellement ordonné peut libérer dans n'importe quel ordre des éléments de même priorité (ici 11(a) et 11(b)).

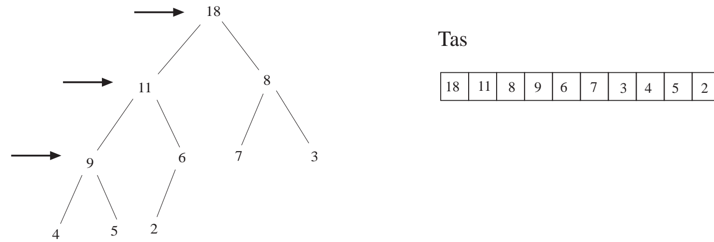


Figure 2.7: Représentation d'un arbre parfait à l'aide d'un tas.

**Propriétés d'un tas $T$ :**

- $T[1]$ désigne la racine de l'arbre parfait correspondant ;
- $T[i\,div\,2]$ désigne le père de $T[i]$ ;
- $T[2i]$ et $T[2i+1]$ sont les fils de $T[i]$ dans l'arbre parfait, s'ils existent ;
- si l'arbre a $p$ noeuds avec $p = 2i$, alors $T[i]$ n'a qu'un fils, $T[p]$ ;
- si $i > p\,div\,2$, alors $T[i]$ est une feuille.

**Exercice 2.1** *Implantation des opérations : Programmer, dans le cadre d'une représentation par tas, les opérations* `ajout` *et* `suppression` *d'une file de priorité. On utilisera les définitions suivantes :*

```
type Elt is record
  priorite:integer;
  valeur:V;
end record;
type Tas is array(integer range<>) of Elt;
function "<"(x,y:elt) return boolean is
  return x.priorite < y.priorite;
```

Solution :

```
procedure ajout(T:in out Tas; nb:in out integer; x:in Elt) is
  -- ajout de x dans T, dont le nombre d'elements est nb
  i:integer;
begin
  nb:=nb+1; T(nb):=x;     -- ajout de x dans la derniere feuille
```



```
  i:=nb;
  while(i>1) and T(i)>T(i div 2) loop
    T(i)<-->T(i div 2);   -- echange pere fils
    i:=i div 2;
  end loop
end ajout;

procedure suppression(T:in out Tas;nb:in out integer;val:out V) is
  -- suppression de l'elt prioritaire. val renvoie sa valeur
  i,j:integer;
begin
  val:=T[1].valeur;
  T[1]:=T[nb]; nb:= nb-1;  -- derniere feuille placee au sommet
  i:=1;
  while i<=nb div 2 loop   -- tant que T[i] n'est pas une feuille
    if (2i=nb) or T[2i]>T[2i+1]
      j:=2i; else j:=2i+1; -- T[j] est le fils a considerer
    end if;
    if T[i]<T[j] then
      T[i]<-->T[j]; i:=j;  -- echange pere fils
      else exit;           -- sinon, on sort de la boucle
    end if;
  end loop;
end supprimer;
```

**Remarque 5** *L'opération d'échange d'éléments peut être remplacée par une affectation puisque l'un des deux éléments echangés est toujours le même. De plus, lorsque les éléments sont de taille importante, on peut préfèrer une implémentation par "tas indirect"[2]. Des pointeurs vers les éléments sont alors stockés dans le tas plutôt que les éléments eux mêmes.*

**Exercice 2.2** *Comme nous l'avons montré plus haut, le tas implémente les opération usuelles d'une file de priorité en $O(\log_2(n))$ (hauteur de l'arbre parfait). Comment pourrait-on implémenter l'opération* `modif-pr`*, qui donne une nouvelle priorité à un élément déjà présent dans la file ? Quelle serait sont coût ?*

### 2.2.3 Application : le tri par tas

L'implémentation d'une file de priorité à l'aide d'un tas conduit à un algorithme de tri efficace, le **tri par tas**[3].

**Principe :** L'idée de base consiste à insérer les éléments du tableau à trier dans un tas, la valeur de priorité étant donnée par le champs selon lequel s'effectue le tri. Il reste alors à ressortir les éléments du plus grand au plus petit en vidant le tas progressivement. Etudions le coût de ce processus :

- La construction du tas par insertions successives se fait en :

$$\log(1) + \log(2) + \ldots + \log(n) = \log(n!) \leq \log(n^n) = n\log(n)$$

[2] "indirect heap" en anglais.
[3] "Heapsort" en anglais

- Un calcul analogue montre que les suppressions successives pour passer du tas au tableau trié se font également en $n\log(n)$ dans le cas le pire.

En pratique, l'implémentation peut être réalisée en évitant l'utilisation d'un tableau auxiliaire pour le tas : en effet, le tas peut être stocké dans la partie gauche du tableau initial, au fur et à mesure que des éléments en sont retirés. De même, les éléments sortis de la file, qui libèrent les dernières cases du tas, sont rangés sur la droite au fur et à mesure, comme le motre la figure 2.8.

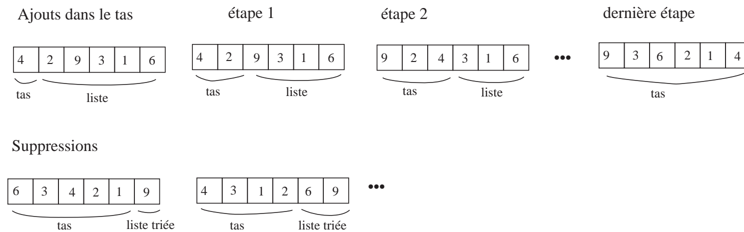


Figure 2.8: Tri en ordre croissant d'un tableau d'entiers :
(a) insertions sucessives dans le tas (partie gauche du tableau) ;
(b) suppressions successives des éléments les plus grands.

**Optimisation de la création du tas :** Une amélioration consiste à remplacer la première phase (insertions successives dans le tas) par une réorganisation directe du tableau de départ. Pour cela, considérons l'arbre parfait associé au tableau (et qui n'est pas, en général, partiellement ordonné). On appelle sur chaque noeud interne de cet arbre, du dernier jusqu'à la racine, une procédure `reorganiser` qui effectue des échanges père-fils en chaîne, jusqu'au bas de l'arbre si nécessaire. Ce procédé permet de donner une structure de tas au tableau en un temps $O(n)$ au lieu de $n\log(n)$. Il s'implémente de la manière suivante : première phase

```
for i in reverse 1..(n div 2) loop reorganiser(T,i);

procedure reorganiser(T: Tas; i:integer) is
  -- T(i) est compare a ses fils; echanges en chaine vers le bas
  j:integer;
begin
  if 2i+1>n or T(2i)>=T(2i+1)
    then j:= 2i; else j:= 2i+1; -- choix du fils
  end if;
  if T(j)>T(i) then
    T(j)<-->T(i);        -- echange pere fils
    if j<=n div 2 then   -- si j n'est pas une feuille
      reorganiser(T, j); -- on continue les echanges
    end if;
  end if;
end reorganiser;
```

**Exercice 2.3** *Faire fonctionner ce processus de réorganisation sur le tableau de la figure 2.8.*



Il nous reste à montrer que l'utilisation de cette procédure est plus efficace que des insertions successives dans un tas. Le coût est proportionnel au nombre total d'appels à réorganiser. Comptons le nombre d'appels engendrés par `reorganiser(p)` :

- si $p \in [\frac{n}{4}+1 \dots \frac{n}{2}]$, les fils de $p$ sont des feuilles. Il y a donc un seul appel (pas d'appel secondaire) ;
- si $p \in [\frac{n}{8}+1 \dots \frac{n}{4}]$, les petits fils de $p$ sont des feuilles. Le nombre d'appels est donc de deux (primaire plus secondaire) ;
- ...
- si $p \in [\frac{n}{2^{i+1}}+1 \dots \frac{n}{2^i}]$, le nombre d'appels est au plus égal à $i$. On remarque de plus qu'il y a un nombre d'éléments inférieur ou égal à $n/2^i$ à ce niveau.

Comptons maintenant le nombre total $C$ d'appels à `reorganiser`, en sommant les nombres d'appels effectués à chaque niveau de l'arbre, de bas en haut. Soit $k$ tel que $2^k \leq n < 2^{k+1}$. On a :

$$\begin{aligned} C &\leq \sum_{i=1}^{k} i\frac{n}{2^i} \quad \text{(nombre d'appels par elt} \times \text{nombre max d'elts à ce niveau)} \\ &\leq \frac{n}{2}\sum_{i=1}^{k}\frac{i}{2^{i-1}} \end{aligned}$$

Or on sait que la somme de la série $\sum ix^{i-1}$ est $1/(1-x)^2$, puisque c'est la dérivée de $\sum x^i = 1/(1-x)$. D'où $C \leq 2n$.

La réorganisation du tableau sous forme de tas se fait donc en $O(n)$ opérations. Notons que la complexité totale du tri par tas reste $n\log_2(n)$, du fait de la seconde phase.

**Exercice 2.4** *Déduire des calculs ci-dessus la distance moyenne d'un noeud à une feuille dans un arbre binaire complet.*

**Correction:** *En fait, on a calculé ci-dessus la somme des hauteurs des noeuds de l'arbre. Si l'arbre est complet, les inégalités deviennent des égalités. En divisant le résultat par $n$ (nombre de noeuds) on en déduit que la distance moyenne - comptée en nombre de noeuds - d'un noeud à une feuille dans un arbre binaire complet est de 2. Ce résultat peut se vérifier aisément sur un exemple.*

**Exercice 2.5** *Ecrire une procédure récursive pour la réorganisation d'un tableau $T[1..n]$ de $n$ éléments en tas. Analyser son coût en utilisant les techniques d'analyse d'équations aux récurrences (chap. 1 §1.3).*

**Correction:** *Le tableau implémente un arbre binaire complet. Pour réorganiser les feuilles de l'élément $i$ en tas, il suffit de réorganiser le sous arbre gauche (de racine $2i$), et le sous-arbre droit (de racine $2i+1$); ensuite, on range l'éléme nt $i$ en le faisant descendre à sa bonne place (ce qui revient à modifier sa priorité).*

*D'où le programme :*

`ReorganiserEnTas ( i, n ) ;` *// Reorganise l'arbre de racine i*

```
if (2*i <= n) {  // sinon i est une feuille et on a fini
    ReorganiserEnTas ( 2*i+1, n ) ; // dans ce sous arbre: (n-i)/2 éléments
    ReorganiserEnTas ( 2*i, n ) ;  // dans ce sous arbre: n-(n-i)/2 éléments
    RetablirTasDescente (i, n) ;  // en cout log n
}
```

*Le coût est* $T(n) = 2.T(n/2) + (\log n)$. *C'est de la forne* $f(n) = \alpha.if(n/K) + g(n)$ *avec* $\alpha = K = 2$ *et* $g(n) = \log n$. *On a donc* $g(n) = \log n$ *polynomialement négligeable devant* $n^{\log_K \alpha} = n$ *et donc* $T(n) = O(n^{\log_K \alpha}) = O(n)$.

# Chapitre 3

# Exemple : implantations de l'algorithme de Dijkstra

Ce chapitre présente la programmation de l'algorithme de Dijkstra pour caluler les plus courts chemins issus d'un sommet dans un graphe. Cet algorithme utilise une file de priorité; différentes variantes sont étudiées selon le choix d'implantation de cette file.

## 3.1 Introduction et rappels

Le but de cette section est de montrer comment, pour un problème spécifique on peut utiliser, voire adapter, les diverses représentations des files de priorités que nous venons de voir.

Dans tout ce qui suit on a un graphe $G = (V, E)$, orienté avec une fonction $\ell : E \to \mathbb{N}$ qui à chaque arc $e \in E$ associe une longueur **entière** $\ell_e$. On notera $L = \max_{e \in E} \ell_e$. On cherche un plus court chemin de $s \in V$ à tous les autres sommets.

**Rappel de l'algorithme**

**debut**
- $S = \emptyset$, $\overline{S} = \{s\}$
- $d(s) = 0$
- $d(i) = +\infty$ pour tout $i \in V \setminus \{s\}$
- **tant que** $\overline{S} \neq \emptyset$ **faire**
- **debut**
  - soit $i \in \overline{S}$ tel que $d(i) = \min\{d(j) : j \in \overline{S}\}$
  - $S = S \cup \{i\}$
  - $\overline{S} = \overline{S} \setminus \{i\}$
  - **pour tout** $j \in V$ tel que $(i, j) \in E$ **faire**
    - **si** $d(j) > d(i) + \ell_{ij}$ **alors**
    - **debut**
      - **si** $d(j) == +\infty$ $\overline{S} = \overline{S} \cup j$
      - $d(j) = d(i) + \ell_{ij}$ et pred($j$)=$i$.
    - **fin**
- **fin**

**fin**





**Proposition 1** *A la fin de l'algorithme les $d(i) < \infty$ représentent les plus courtes distances de $s$ à $i$. Le splus courts chemins se retrouvent en utilisant les pred(.).*

**Preuve**. Il suffit de montrer que, quand un sommet $i$ entre dans $S$, la valeur $d(i)$ dont il est affublé représente bien la plus courte distance de $s$ à $i$. Elle représente, par son calcul, la plus courte distance parmi tous les chemins qui n'utilisent que des sommets dans $S$, hormis $i$. S'il devait y avoir un chemin plus court, celui-ci devrait sortir de $S$. Appellons $j \neq i$ le premier sommet, en partant de $s$, hors de $S$ sur un tel chemin. Par le choix de $i$, à ce moment là $d(j) \geq d(i)$ et comme toutes les longueurs sont positives ou nulles, la longueur de ce chemin de $s$ à $i$ ne peut être inférieure à $d(j) \geq d(i)$ □

Les sommets $i$ qui à la fin de l'algorithme ont $d(i) = +\infty$ ne sont pas atteignables à partir de $s$.

Si on essaie de calculer la complexité de cet algorithme, on voit qu'il y a une boucle qui est parcourue $n = \mid V \mid$ fois. Donc il suffit de calculer la complexité d'une boucle. Plaçons nous dans la $k^{ème}$ itération de la boucle. Dans cette boucle plusieurs opérations sont effectuées:

1. On cherche le sommet $i$ de $\overline{S}$ tel que $d(i)$ soit minimum.
2. On fait des mises à jour des $d(j)$ pour les sommets $j$ successeurs de $i$ et non dans $S$.

Les mises à jour nécessitent au plus 3 opérations, deux additions et une comparaison, pour chaque successeur de $i$. En notant qu'une mise à jour ne se fait qu'une fois sur un arc donné, le total des mises à jour sur l'ensemble de l'algorithme est borné par $3m$ où $m = \mid E \mid$).

Reste la complexité de la recherche du sommet $i$. Celle-ci va dépendre de la recherche du minimum. Les éléments de $\overline{S}$ peuvent être considérés comme les éléments d'une file de priorité où entre deux éléments $j$ et $k$, la priorité dépend des valeurs de $d(j)$ et $d(k)$, la plus petite de ces valeurs donnant une priorité plus élevée.

## 3.2 Recherche séquentielle

Les éléments de $\overline{S}$ sont stockés dans un tableau ou une liste non triée. On parcourt donc la liste pour trouver l'élément de $d(i)$ minimum. Dans le pire des cas la taille de ces listes est de 1 au premier passage dans la boucle, puis de $n-1$, $n-2$, ..., 1. Donc on fera $(n-2) + (n-3) + \ldots 1$ comparaisons, et donc $O(n^2)$ comparaisons en tout dans l'algorithme.

Dans ce cas la complexité de Dijkstra est de $O(m+n^2) = O(n^2)$ puisque $m \leq n(n-1)/2$. Même si pour des graphes denses (beaucoup d'arcs) on ne peut pas faire mieux en terme de complexité, **asymptotique**; des optimisations sont possibles car ce qui nous intéresse est le temps réel.

## 3.3 Une remarque essentielle

**Proposition 2** *Les valeurs de $d(i)$ dans l'ordre d'entrée des sommets $i$ dans $S$ sont croissants (au sens large).*

Ceci est facile à voir. Soient deux sommets $i$ et $j$ entrés dans $S$ successivement. Quand on a choisi $i$ on avait $d(i) \leq d(j)$, $d(i)$ n'a pas changé, par contre $d(j)$ a pu changer dans la mise à jour; mais comme celle-ci implique $d(i)$ et que la longueur $\ell_{ij} \geq 0$, on ne peut pas avoir $d(j) < d(i)$.

## 3.4 L'implantation de Dial

La valeur maximum que peut prendre une valeur $d(i)$ est $(n-1)L$. On peut donc prendre un tableau de taille $(n-1)L+1$ de listes doublement chaînées. La liste correspondante à la case $p$ contient tous les sommets $i$ tels que $d(i) = p$. Au départ toutes les listes sont vides sauf celle correspondant à la case 0 qui contient $s$. La recherche du prochain sommet est alors très simple, si la liste correspondante à la case où l'on se trouve est vide, on cherche, à partir de là (on ne revient pas en arrière) la prochaine liste non vide et on choisit le premier sommet de cette liste, que l'on retire de la liste etc .... Toutes les opérations pour les mises à jour, insertion, retrait, se font en $O(1)$, donc au total les mises à jour se font en $O(m)$.

Il faut parcourir tout le tableau de listes, ce qui se fait en $O(nL)$, l'algorithme est donc en $O(m + nL)$. A noter (cf deuxième partie du cours d'algorithmique) que ce n'est pas un algorithme polynomial dans le sens de la théorie de la complexité, pour cela il faudrait que $L$ apparaisse sous forme de $logL$. Pour des valeurs de $L$ raisonnables, c'est un bon algorithme, cependant il peut nécessiter beaucoup de place. On va maintenant voir une manière astucieuse d'économiser de la place.

Pour cela il suffit de remarquer que la valeur maximum des $d(j) < \infty$ pour $j \in \overline{S}$ quand on vient de mettre $i$ dans $S$, est de $d(i) + L$, et d'autre part, d'après la proposition 1, $d(j) \geq d(i)$ pour tout $j \in \overline{S}$. Il n'y a donc que $L + 1$ valeurs différentes pour $d(j)$ pour $j \in \overline{S}$ (autre que $\infty$). Quand la liste 0 est vide on peut y stocker les sommets avec $d(i) = L + 1$, plus généralement on a un tableau de $L + 1$ listes, dont les cases sont numérotées $0, 1, \ldots L$, la liste $p$ contient dans un premier temps les sommets de $d(i) = p$, puis ceux avec $d(i) = p + L$, et ainsi de suite ceux de $d(i) = p + k(L+1)$, $k = 0, \ldots, n-2$. La recherche du prochain sommet $i$ traité consiste à soit prendre le prochain sommet de la liste que l'on est en train de traiter, soit, si celle-ci est vide de chercher dans la prochaine non vide. Quand on arrive à la liste $L$ on repart dans la recherche à la liste 0.

Ceci permet d'atteindre une complexité de $O(m+nL)$ avec une place mémoire en $O(L)$ pour la recherche du prochain sommet (il reste à considérer la place mémoire pour le graphe et les longueurs d'arcs).

**Remarque 6** *On peut voir cette implantation comme étant l'utilisation d'une table de haschage où la fonction de haschage est la valeur modulo $L + 1$. Les retraits et insertions se faisant de telle façon qu'un un instant donné une liste donnée ne contient que des éléments de même valeur.*

## 3.5 L'implantation par tas binaire

Ici on garde les valeurs $d(j) < \infty$, pour $i \in \overline{S}$ dans un tas; la racine contenant la plus petite valeur.



**Exercice 3.1** *1) Ecrire l'algorithme de Dijkstra en y incorporant l'appel au min, le retrait de cet élément, ainsi que les appels de mise à jour du tas lors de la mise à jour des $d(.)$. Attention les $j$ tels que $d(j) = \infty$ ne sont pas dans le tas, donc il y a des insertions à prévoir.*
*2) Quel est le coup de la gestion du tas? (aide montrer qu'elle en O(mlogn)*

**Proposition 3** *L'implantation de Dijkstra par tas binaire est en $O(mlog(n))$*

**Remarque 7** *Si le graphe est dense, cet algorithme est moins bon asymptotiquement que la recherche séquentielle. La plupart des graphes rencontrés dans la pratique sont peu denses, le nombre d'arc étant linéaire par rapport au nombre de sommets, par exemple des graphes planaires (les arcs ne se croisent pas) ou avec peu de croisements, dans ce cas cette implantation est bien meilleure.*

**Exercice 3.2** *Nous n'avons étudié que les tas binaires. Il existe d'autres structures de tas. Il y a le d-tas, $d \geq 2$, pour $d = 2$ bien sûre il s'agit du tas binaire.*

1. *Donner la complexité de l'opération d'insertion et de suppression du maximum dans un d-tas.*
2. *En déduire que la complexité de l'algorithme de Djikstra avec d-tas.*
3. *Quelle valeur de d donne la meilleure complexité ?*

**Correction:**

1. Insertion = remontée, donc $O(\log_d n)$ permutations avec le père au pire jusqu'à la racine de l'arbre.
   Suppression-Max = descente, donc $O(\log_d n)$ permutations avec le plus grand des d fils pour un coût de $O(d \log_d n)$.
2. D'après la question précédente, le coût est $O(m \log_d n + nd \log_d n)$.
3. Soit $\phi(d) = m \log_d n + nd \log_d n = \log n . \frac{m+nd}{\log d}$; pour $n$ et $m$ fixé, on cherche $d$ qui minimise $\phi$. L'annulation de $\phi'(d)$ donne comme choix possible (en supposant $m \gg n$) $d = max(2, \frac{m}{n \log \frac{m}{n}})$.

**Remarque 8** *Il y a aussi le tas de Fibonacci. Dans ce cas la complexité est de $O(m + nlogn)$. Et donc le pire cas est comparable au pire cas de la recherche séquentielle, ce qui n'est pas le cas des autres implantations par tas. Vous trouverez la description des d-tas et des tas de Fibonacci dans tous les bons livres d'algorithmique.*

**Remarque 9** *Les implantations par tas ne nécessitent pas que les longueurs soient entières.*

## 3.6 Les listes de Radix

Il s'agit d'une gestion très sophistiquée d'un tableau de listes. On va utiliser $O(log(nL))$ listes. La largeur du domaine des valeurs (valeur la plus haute - valeur la plus basse $+1$) des listes sera dans l'ordre de $1, 1, 2, 4, 8, 16, \ldots$.

L'idée est de modifier dynamiquement les bornes des valeurs affectées à chaque liste de sorte que l'élément minimal soit toujours dans une liste dont tous les éléments ont la même valeur (largeur du domaine des valeurs=1).

Dans ce qui suite les listes seront numérotées $0, 1, 2, \ldots, k = \lceil log(nL) \rceil$. Le domaine des valeurs de la liste $r$ sera noté $domaine(r)$.
Au départ:
$domaine(0) = [0]$
$domaine(1) = [1]$
$domaine(2) = [2, 3]$
$domaine(3) = [4, 7]$
$domaine(4) = [8, 15]$
$\vdots$
$domaine(K) = [2^{K-1}, 2^K - 1]$

Le domaine de chaque liste va changer au cours de l'algorithme, mais pas sa largeur.

Pour trouver le prochain sommet qui va entrer dans $S$, on parcourt, les listes jusqu'à trouver la première qui n'est pas vide. Supposons par exemple qu'il s'agit de la liste 4. On pourrait parcourir toute la liste pour trouver la plus petite valeur et ainsi de suite jusqu'à ce qu'elle soit vide. Une méthode plus judicieuse est de redistribuer ses éléments dans des listes, pour cela on va redistribuer les domaines des listes $0, 1, 2, 3, 4$, puisque les valeurs qu'ils contenaient ne seront plus rencontrées. Désormais:

- $domaine(0) = [8]$
- $domaine(1) = [9]$
- $domaine(2) = [10, 11]$
- $domaine(3) = [11, 15]$
- $domaine(4) = \emptyset$

et on répartit les éléments de la liste 4 dans ces listes. Si les listes 0 et 1 sont encore vides on recommence l'opération.

Ainsi comme annoncé les éléments de plus petite valeur seront dans une liste dont tous les éléments ont cette unique valeur.

**Exercice 3.3** *Ecrire l'algorithme de Dijkstra qui utilise les listes de Radix.*

**Proposition 4** *L'implantation de Dijkstra par listes de Radix est en* $O(m + nlog(nL))$

**Exercice 3.4** *Essayer de démontrer cette proposition.*

**Remarque 10** *Comme dans l'implantation de Dial, on peut n'utiliser que* $\lceil log(L) \rceil + 1$ *listes et dans ce cas l'algorithme est meilleur et sa complexité devient* $O(m + nlog(L))$*. En utilisant en plus des tas de Fibonacci à l'intérieur de chaque liste on peut encore réduire la complexité à* $O(m + n\sqrt{log(L)})$*.*



## 3.7 Conclusion

On vient de voir qu'il n'y a pas une meilleure manière de traiter une file de priorité. Pour un problème donné il n'y a pas une réponse unique à la question de savoir qu'elle est la meilleure implantation. Ici cela dépend de la densité du graphe et de la valeur de $L$.

D'autre part il y a un équilibre à trouver entre rapidité de l'implantation et l'effort de programmation nécessaire. Par exemple il est bien connu que les tas de Fibonnacci sont toujours gagnants, par contre leur implantation demande un certain effort.

# Chapitre 4

# Dictionnaires

Les dictionnaires sont des structures permettant de stocker puis de rechercher de l'information. Leur utilisation est fondamentale pour toutes les applications de l'informatique. Certains dictionnaires sont entièrement construits lors d'une phase d'initialisation, puis sont utilisés uniquement pour des opérations de recherche. D'autres permettent de réaliser conjointement des recherches, des ajouts, et des suppressions d'éléments. Ces deux modes de fonctionnement peuvent conduire à des choix différents pour l'implémentation.

Ce chapitre présente la structure dictionnaire, puis étudie les représentations les plus classiques en fonction du mode d'utilisation qui est désiré :

- tableau et liste d'éléments;
- table de hachage;
- arbre binaire de recherche;
- autres structures : arbres équilibrés, splay-tree.

**Définition :** Soit $V$ un type. On appelle **dictionnaire($V$)** une structure permettant de stocker un ensemble d'éléments de type $V$. Cette structure est munie d'une opération de recherche, ainsi que d'opérations d'ajout et de suppression d'éléments.

Notons qu'il n'y a pas véritablement de distinction entre le type dictionnaire et le type **ensemble**. Le terme "dictionnaire" est simplement préféré ici, dans la mesure où nous étudierons plus particulièrement les structures qui facilitent les opérations de recherche. Nous ne traiterons pas d'opérations classiques sur les ensembles comme l'union et l'intersection.

En pratique, l'opération de recherche est adaptée au cadre d'utilisation du dictionnaire, de manière à renvoyer des informations plus précises que la simple indication d'un succès ou d'un échec. Ces informations peuvent être la place de l'élément cherché dans la structure, ou bien un pointeur vers un enregistrement dont le champs de type $V$ correspond à la valeur recherchée.

## 4.1 Fonction caractéristique représentée par un tableau de booléens

Si le type $V$ est fini et de taille raisonnablement petite, un ensemble d'éléments de $V$ peut être représenté par un tableau de booléens indexé par le type $V$, et correspondant à sa





fonction caractéristique (vrai si l'élément est présent, faux s'il est absent).

Cette représentation, qui ne peut bien-sûr être utilisée que dans des cas très particuliers, présente l'avantage de permettre l'implantation des opérations usuelles du type dictionnaire (recherche, ajout, suppression) en temps constant. On peut aussi remarquer que c'est la seule représentation du type dictionnaire qui n'impose pas de redéfinition de la notion d'égalité. En revanche, les structures classiques que nous allons maintenant étudier (tableaux ou listes d'éléments, arbres de recherche) obligent à faire une distinction entre égalité des structures et égalité entre les ensembles qu'elles représentent.

## 4.2 Tableaux et listes d'éléments

Pour un type $V$ infini, ou comportant un nombre d'éléments beaucoup plus grand que celui qui sera utilisé en pratique dans les dictionnaires, on se tourne vers l'utilisation de structures où sont stockées les éléments présents. Les plus simples de ces structures sont les tableaux et les listes chaînées.

### 4.2.1 Tableaux : recherche séquentielle et dichotomique

Cette structure de taille fixe se justifie parfaitement lorsqu'il s'agit de dictionnaires initialisés une bonne fois pour toutes, et dans lesquels seules des recherches d'éléments sont effectuées.

**Exercice 4.1** *Tableau non trié. Donner le coût de l'initialisation et de la recherche séquentielle pour un tableau de taille $n$, dans le cas le meilleur, dans le cas le pire, et en moyenne.*

Solution : L'initialisation se fait par l'ajout successifs d'éléments (supposés distincts), et a donc pour coût $n$ en nombre d'affectations.

**Remarque sur la programmation de la recherche séquentielle.**
L'intérêt de la recherche séquentielle étant la simplicité, il est important de bien la programmer. Or, l'écriture "normale" (nous écrivons ici en C la recherche d'un élément dans un tableau $T$ de *last* éléments entiers) :

```
do(i=0;i<last&&T[i]!=val;i++);
```

effectue une comparaison sur l'indice en plus de la comparaison sur la valeur à chaque pas de la boucle. En utilisant une "sentinelle"[1] on peut sensiblement améliorer l'efficacité de la recherche (on suppose alors que la place d'indice *last* n'est pas utilisée) :

```
do(T[last]=val,i=-1;T[++i]!=val;);
```

Cette écriture donne un code très compact, qui peut être deux fois plus rapide que le précédent! Bien entendu, la différence s'estompe si la comparaison pour *val* est plus coûteuse que la comparaison d'indices, comme c'est le cas pour les chaînes de caractères. Dans cette version rapide, on effectue $n+1$ comparaisons si *val* ne figure pas dans $T$.

Évaluons le coût en nombre de comparaisons de la recherche séquentielle :

- cas le meilleur : 1

[1] L'introduction d'une sentinelle est une astuce de codage importante pour de nombreux algorithmes, notamment le tri par insertion et la recherche dans un ABR.

- cas le pire : $n+1$
- en moyenne : Soit $q_i$ la probabilité pour qu'un élément soit présent dans la i-ème case. La probabilité pour qu'un élément soit absent est $(1-q)$, où $q = \sum q_i$. D'après la formule du paragraphe 1.4.2, la complexité moyenne est alors :

$$C_{moyenne}(n) = (n+1)(1-q) + \sum iq_i$$

Si on suppose que les éléments sont équidistribués, alors $q_i = \frac{q}{n}$. On a donc :

$$\begin{aligned} C_{moyenne}(n) &= (n+1)(1-q) + \frac{q}{n}\sum i \\ &= (n+1)(1-q) + \frac{n+1}{2}q \end{aligned}$$

Le coût moyen d'une recherche est donc en $O(n)$. Si l'élément recherché est présent ($q=1$), on gagne un facteur 2 en moyenne par rapport à la complexité dans le cas le pire.

**Tableau trié et recherche dichotomique :** Le coût de l'initialisation comprend maintenant une phase de tri du tableau, et est donc de l'ordre de $n\log(n)$. Évaluons la complexité d'une recherche dichotomique, qui peut être réalisée par exemple comme suit :

```
procedure recherche(x, inf, sup)
  si inf <= sup
    entier m := E(inf + sup / 2);   -- E: partie entiere
    si x=T[m] alors renvoyer VRAI;
    sinon, si x>T[m] alors recherche(x, m+1, sup);
                  sinon recherche(x, inf, m-1);
  sinon renvoyer FAUX;
```

Dans le cas le meilleur (élément présent au milieu du tableau), une seule comparaison est nécessaire. Pour évaluer les complexités au pire et en moyenne, étudions **l'arbre de décision** associé à l'algorithme, représenté sur la figure 4.1.

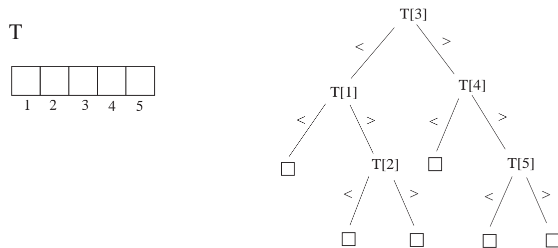


Figure 4.1: Arbre de décision d'une recherche dichotomique dans un tableau à 5 éléments.

- Le coût dans le cas le pire est égal à deux fois la hauteur de l'arbre de décision, puisque deux comparaisons sont effectuées pour chaque noeud.
- Le coût moyen d'une recherche positive est égal à deux fois la profondeur moyenne d'un noeud interne dans l'arbre de décision.



- Le coût moyen d'une recherche négative est égal à deux fois la profondeur moyenne d'une feuille.

Intuitivement, puisque le nombre d'éléments sur lequel s'effectue la recherche est divisé par deux à chaque étape, on se doute que la hauteur de l'arbre, ainsi que les moyennes évoquées ci-dessus, vont être de l'ordre de $\log_2(n)$.

Ceci peut être vérifié formellement, en se servant des mesures sur les arbres données dans l'annexe A:

- L'arbre de décision est un arbre binaire localement complet (0 ou 2 fils en chaque noeud). Il y a donc $n+1$ feuilles pour $n$ noeuds internes.
- On peut montrer que les feuilles sont sur deux niveaux seulement (par l'absurde : sinon une "'moitié" du tableau aurait deux éléments de plus que l'autre).
- Si on pose: $n = 2^p + k$, avec $0 \leq k < 2^p$, alors les feuilles sont aux niveaux $p$ et $p+1$ et il y a $k+1$ noeuds internes au niveau $p$.

  Démonstration : le nombre de feuilles sur les niveaux de 0 à $i$ avec $i$ complet est : $1+2+\ldots+2^i = 2^{i+1}-1$. Le niveau $p$ peut donc ne pas être complet, mais le niveau $p-1$ l'est. Il y a donc $2^p-1$ feuilles sur $0\ldots(p-1)$. Comme $n = 2^p+k$, il reste $k+1$ noeuds internes au niveau $p$, et les feuilles sont bien sur les niveaux $p$ et $p+1$.

On en déduit que la hauteur de l'arbre, qui donne le coût dans le cas le pire, est $p+1 = O(\log_2(n))$.

Le coût moyen d'une recherche négative est égal à deux fois la profondeur moyenne d'une feuille dans l'arbre de décision $A_n$, avec :

$$\begin{aligned} PE(A_n) &= \frac{1}{n+1}LCE(A_n) \quad (LCE(A_n) \text{ est la somme des hauteurs des feuilles}) \\ &= \frac{1}{n+1}\left(2(k+1)(p+1) + (n+1-2(k+1))p\right) \\ &= p+2-\frac{2^{p+1}}{n+1} \end{aligned}$$

puisque $k = n - 2^p$, $PE(A_n)$ est de l'ordre de $p = \log_2(n)$. Le coût moyen est donc logarithmique pour une recherche négative.

Le coût moyen d'une recherche positive est égal à deux fois la profondeur moyenne d'un noeud interne plus 1. Or,

$$\begin{aligned} PI(A_n) &= \frac{1}{n}LCI(A_n) \\ &= \frac{1}{n}(LCE(A_n) - 2n) \end{aligned}$$

D'où un ordre de grandeur de $\log_2(n)$ pour $PI(A_n)$, et pour le coût moyen d'une recherche positive.

**Conclusion :** Au pire comme en moyenne, le coût d'une recherche dans un tableau trié est en $\log_2(n)$. Il est donc beaucoup plus intéressant d'utiliser un tableau trié pour un dictionnaire avec initialisation puis recherches uniquement, dans le cas où le type $V$ est ordonné.

**Exercice 4.2** *Analyser le coût de la procédure de recherche par dichotomie donnée ci-dessus quand on supprime le test $x = T[m]$.*

### 4.2.2 Listes chaînées

On peut envisager la représentation d'un dictionnaire à l'aide d'une liste chaînée de ses éléments.

**Exercice 4.3** *1. Dans quel cas cette représentation vous semble-t-elle mieux adaptée qu'un tableau ?*

*2. Est-il intéressant d'utiliser une liste triée ?*

Réponse : Le tableau trié est mal adapté aux dictionnaires dans lesquels on désire pouvoir combiner recherches, ajouts et suppressions. En effet, les opérations d'ajout et de suppression demandent de coûteux décalages d'éléments à gauche ou à droite, en $O(n)$ affectations. Dans ce cas, une liste chaînée peut être un meilleur choix. Comme cette représentation ne permet pas les recherches dichotomiques, il est inutile de trier la liste. Le coût d'une recherche (séquentielle) est, comme on l'a vu plus haut, $O(n)$ en moyenne, celui d'un ajout en tête et d'une suppression (sans compter la phase de recherche de l'élément à supprimer) sont en $O(1)$.

## 4.3 Tables de hachage (hash code)

On a vu que la fonction caractéristique représentée par un tableau de booléens permet d'implémenter la recherche, l'ajout et la destruction dans le dictionnaire en temps $O(1)$. La table de hachage est une adaptation de cette implémentation lorsque le cardinal de l'ensemble des clefs $\mathcal{C}$ est grand (ce qui interdit de classer toutes les clefs possibles dans un tableau).

### 4.3.1 Représentation classique d'une table de hachage

Une table de hachage consiste à représenter le dictionnaire par tableau $T$ de taille $m$. Ce tableau est indicé par les valeurs d'une fonction de hachage $h$

$$h : \mathcal{C} \rightarrow \{0, \ldots, m-1\}$$

qui associe à chaque clef un indice dans la table. Un élément de clef $k$ est stocké à la case $T[h(k)]$.

Ainsi, la case $T[i]$ pointe vers l'ensemble des éléments du dictionnaire dont l'image de la clé par $h$ est $i$; ces éléments sont dits "en collision" (voir figure 4.2).

L'ensemble des éléments en collision peut être implémenté par un tableau de taille dynamique ou, le plus souvent, par une liste doublement chaînée pour permettre de détruire un élément en temps constant.



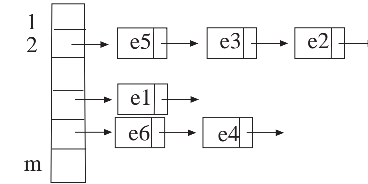


Figure 4.2: Hachage avec chaînage séparé.

D'autres structures dictionnaire peuvent aussi être utilisées à la place d'une liste pour gérer les éléments en collision: un arbre binaire de recherche ou un AVL par exemple. On peut aussi procéder à un "hachage dynamique", en réalisant un second hachage d'une liste de collision, dès que cette dernière devient trop longue.

### 4.3.2 Analyse du coût

Pour l'analyse, on suppose que les éléments en collision sont stockés dans une liste doublement chaînée. Ainsi, le coût de la destruction est constant.

Le coût de l'insertion d'un élément est égal au coût de l'évaluation de la fonction de hachage, plus $O(1)$; donc généralement constant en pratique.

Pour étudier le coût de la recherche, on considère le taux de remplissage $\frac{n}{m}$ de la table . En pire cas, le coût d'une recherche infructueuse est $O(n)$ (tous les éléments sont en collision sur une même case).

Pour évaluer le coût en moyenne, on fait l'hypothèse que le hachage est *uniforme*, c'est à dire que les valeurs de la fonction de hachage sont uniformément distribuées dans $\{0, \ldots, m-1\}$. Soit $P$ la distribution de probabilité sur l'ensemble $\mathcal{C}$ des clés ; l'hypothèse de hachage uniforme s'écrit :

$$\forall i \in \{0, \ldots, m-1\} \qquad \sum_{k \in \mathcal{C}\ mbox\ tel\ que\ h(k)=i} P(k) = \frac{1}{m}.$$

En supposant que l'évaluation de la fonction de hachage prend un coût constant, le coût en moyenne d'une recherche est alors :

- $\frac{\sum_{i=0}^{m-1} \text{coût recherche infructueuse liste }_i}{m} = \frac{n}{m} + O(1)$ lorsque la clé n'est pas dans la table;
- $\frac{\sum_{i=0}^{m-1} \text{coût recherche réussie liste }_i}{m} = \frac{\sum_{i=0}^{m-1} \frac{\#\text{éléments liste }_i}{2}}{m} = \frac{n}{2m} + O(1)$ lorsque la clé est dans la table.

Ainsi, en choisissant $\alpha = \frac{n}{m}$ constant, le coût en moyenne de la recherche est constant. En pratique, on choisit généralement un taux de remplissage de 80% environ, i.e. $m \simeq 1,25n$.

**Exercice.** On maintient maintenant les éléments en collision dans un tableau trié par ordre croissant des clés. Calculer en fonction de $n$ et $m$ les coûts en moyenne des opérations d'insertion, destruction et recherche. **Correction:**

- recherche : $\Theta(\log(n/m))$ par une recherche dichotomique.

- insertion et destruction : $\Theta(n/m)$ car il faut décaler le tableau à chaque fois.

### 4.3.3 Choix de la fonction de hachage

L'analyse précédente montre que, sous l'hypothèse de hachage uniforme, une table de hachage permet de réaliser toutes les opérations sur un dictionnaire en temps constant.

En pratique, il est difficile de trouver une bonne fonction de hachage qui fournit un hachage uniforme. On tombe parfois sur des fonctions $h$ se calculant en $O(n)$, ce qui diminue notablement l'intérêt de la structure.

Généralement, on choisit des fonctions de hachage qui peuvent être modifiées facilement, indépendamment de la taille $m$ de la table. Les fonctions le plus souvent utilisées pour réaliser un hachage sont :

- le reste euclidien : $h(e) = e \, mod \, m$, avec $m$ entier.
  Cas pratique : la clef $e = [e_0, \ldots, e_r]$ est une chaîne de $r+1$ caractères où $e_i$ est un caractère sur 8 bits qui peut être vu comme un entier ($0 \leq e_i < 256$). On choisit
  $$h(e) = \sum_{i=0}^{r} a_i.e_i \mod m$$
  où les $a_i$ sont $r+1$ entiers choisis arbitrairement dans $\{0, \ldots, m-1\}$.
  NB: On choisit généralement $m$ distinct d'une puissance de 2 (sinon, si $m = 2^p$, seuls les $p$ bits de poids faible de la clef interviennent pour le calcul du hachage). De même, on évite les puissances de 10 car les clefs peuvent être des nombres décimaux. Généralement, on choisit pour $m$ un nombre premier loin d'une puissance de 2 (pour éviter que 2 chaînes presque semblables ne hachent sur la même valeur).

- la multiplication lorsque les éléments sont des réels (une séquence de bits peut toujours être vue, de manière injective, comme la mantisse d'un réel). On prend pour cela une valeur $\theta \in [0,1]$ (souvent $\theta = \frac{\sqrt{5}-1}{2}$ ou $\theta = 1 - \frac{\sqrt{5}-1}{2}$), et on pose : $h(e) = \lfloor m((\theta e) \, mod \, 1) \rfloor$.
  L'intérêt de cette méthode par rapport à la précédente est que la valeur de $m$ n'est pas critique. En particulier, on peut changer de fonction de hachage en changeant seulement $\theta$, sans changer $m$.

- l'extraction de bits, où les élément sont vus comme des tableaux de bits. Par exemple, on peut prendre comme fonction de hachage les valeurs des $p$ premiers bits, ce qui crée une table de $m = 2^p - 1$ cases.

**Remarque 1.** On peut aussi utiliser comme fonction de hachage $h(e) = e \, mod \, P$, avec $P$ polynôme. Par exemple, on peut choisir pour $P$ un polynôme de degré $d$ dans $\mathbb{Z}/2\mathbb{Z}$. Le reste $h(e) \mod P$ étant alors de degré $d-1$, on a $m = 2^d$ valeurs de hachage possible.
Pour des raisons d'efficacité, comme en pratique on rencontre le plus souvent des chaînes de caractères, on choisit plutôt $P$ à coefficients dans $\mathbb{Z}/2^8\mathbb{Z}$, ou dans le corps de Galois $\mathbb{F}_{256}$ a $2^8$ éléments.



**Remarque 2.** Les empreintes de documents électroniques (i.e. un résumé du document qui permet de l'identifier) utilisent des fonctions de hachage qui ont la propriété d'être faciles à calculer mais pour lesquelles il est coûteux de trouver deux textes qui ont la même empreinte; de plus, deux texte voisins (i.e. la plupart des bits restent inchangés) ont deux valeurs de hachage distinctes avec une grande probabilité. Un exemple d'une telle fonction est SHA-1 (*Secure Hashing Algorithm*, cf [`http://www.itl.nist.gov/fipspubs/fip180-1.htm`]). qui fournit un résumé de 160 bits pour tout message de moins de $2^{64}$ bits,

### 4.3.4 Hachage universel

Dès qu'on fixe une fonction de hachage, on peut tomber dans le pire cas où tous les éléments ont la même valeur de hachage. Pour éviter cela, une solution est de changer dynamiquement de fonction de hachage lorsque trop de collisions se font sur une même entrée de la table.

Le hachage universel consiste à choisir la fonction de hachage au hasard parmi une collection $\mathcal{H}$ de fonctions de hachage. $\mathcal{H}$ est dite *collection universelle* ssi, pour tout couple de clés $k_1$ et $k_2$ distinctes,

$$\frac{\#\{h \in \mathcal{H} : h(k_1) = h(k_2)\}}{\#\mathcal{H}} = \frac{1}{m}.$$

Autrement dit, si on choisit une fonction de hachage $h$ au hasard dans $\mathcal{H}$, $\text{Prob}_h(h(k_1) = h(k_2)) = \frac{1}{m}$.

On peut montrer alors que le nombre moyen de collisions est inférieur à 1. En effet, pour tout couple de clés $k_1 \neq k_2$, soit $C_{k_1,k_2}$ la variable aléatoire qui vaut 1 si $h(k_1) = h(k_2)$ et 0 sinon. Comme la collection est universelle, $E(C_{k_1,k_2}) = 1/m$.
Le nombre total $C_k$ de collisions dans la table $T$ pour une clef $k$ est alors :

$$E(C_k) = \sum_{k' \in T; k' \neq k} E(C_{k,k'}) = \frac{n-1}{m}.$$

En choisissant $m \geq n$, le nombre moyen de collisions est inférieur à 1.

**Un exemple de collection universelle de fonctions de hachage.** Il est assez facile de construire des collections universelles de fonction de hachage.

Par exemple, supposons comme précédemment que la clef $e = [e_0, \ldots, e_r]$ est une chaîne de $r+1$ où $e_i$ est un caractère sur 8 bits ($0 \leq e_i < 256$).

On choisit alors $m$ premier et on note $h_{[a_0,\ldots,a_r]}(e) = \sum_{j=0}^{r} a_j.e_j \mod m$. On prend alors :

$$\mathcal{H} = \left\{ h_{[a_0,\ldots,a_r]} \text{ avec } (a_0, \ldots, a_r) \in \{0, \ldots, m-1\}^n \right\}.$$

On vérifie facilement que $\mathcal{H}$ est universelle. En effet, soit $k$ et $k'$ deux clefs différentes ; $\exists 0 \leq i < r : k_i \neq k'_i$. Fixons les $a_j$ pour $j \neq i$; on a alors $h_{[a_0,\ldots,a_r]}(k) = h_{[a_0,\ldots,a_r]}(k')$ si et seulement si

$$a_i \equiv -\frac{\sum_{j=0, j\neq i}^{r} a_j(k_j - k'_j)}{k_i - k'_i} \mod m$$

(comme $m$ est premier, $k_i - k'_i \neq 0$ est inversible modulo $m$).

Ainsi, pour chaque choix possible des $a_j$ pour $j \neq i$, il existe une unique valeur $a_i$ pour laquelle il y a collision entre $k$ et $k'$. Donc il existe exactement $m^r$ fonctions de hachage

$h_{[a_0,\dots,a_r]}$ (une fonction pour chaque choix possible des $(a_j)_{j\neq i}$) qui entraînent une collision entre $k$ et $k'$. Comme il y a $m^{r+1}$ choix possibles pour une fonction de hachage, on en déduit

$$\text{Prob}_{h\in\mathcal{H}}(h(k_1) = h(k_2)) = \frac{1}{m}$$

et donc $\mathcal{H}$ est bien une collection universelle de fonctions de hachage.

### 4.3.5 Table de hachage à adressage dispersé (open addressing)

Au lieu de stocker les éléments en collision dans un ensemble associé à chaque entrée de la table $T$, une autre solution est de les stocker directement dans des entrées encore libres de la table $T$: c'est ce que l'on appelle une table de hachage à adressage dispersé. L'intérêt est ici d'éviter le stockage de pointeurs intermédiaires.

Dans ce cas, on utilise une fonction de parcours qui donne, à partir d'une entrée donnée, l'entrée suivante à considérer dans le tableau. Le profil de la fonction de hachage $H$ est alors:

$$H : \mathcal{C} \times \{0, \dots, m-1\} \to \{0, \dots, m-1\}$$

et on suppose que la fonction $h$ est telle que $[H(k,0), H(k,1), \dots, H(k, m-1)]$ est une permutation de $[0, \dots, m-1]$.

**Exemples de fonction de hachage dispersé.** Le plus souvent, la fonction de hachage $H$ à adressage dispersé est construite à partir d'une fonction de hachage $h$ simple, comme celles vues précédemment (§4.3.3). Les fonctions $H$ suivantes sont généralement utilisées :

- **adressage linéaire** : $H(k,i) = (h(k) + i) \mod m$.

- adressage quadratique : $H(k,i) = (h(k) + ai + bi^2) \mod m$, avec $a, b \neq 0$ choisis tels que toutes les cases du tableau sont examinées dans le pire cas.
  Par exemple; $a =$

- **adressage double** (le plus courant en pratique) : il utilise deux fonctions de hachage simples différentes, $h_1$ et $h_2$, l'une pour la clef, l'autre pour l'adressage :

  $$H(k,i) = h_1(k) + ih_2(k) \mod m.$$

  On choisit $m$ entier premier: cela permet que quelles que soient les valeurs (non nulles) de $h_1(k)$ et $h_2(k)$, la fonction $H$ permet de parcourir tout le tableau. Contrairement aux adressages linéaires ou quadratique, deux clefs $k$ et $k'$ qui entrent en collision pour une case (par exemple $H(k,0) = H(k',0)$ ne rentrent pas en collision pour la case suivante.

**Algorithmes de gestion.** Chaque case de la table de hachage est marquée avec avec une marque booléenne qui indique si la case est libre ou non (i.e. contient un élément). A l'initialisation, toutes les marques sont libres.

Recherche. Lorsque l'on recherche une clef $k \in \mathcal{C}$, le principe est le suivant. On calcule d'abord $h_0 = H(k,0)$; si $T[h_0]$ ne contient pas un élément de clef $k$, on regarde ensuite $T[h_1]$



avec $h_1 = H(k,1)$, puis $T[h_2]$ avec $h_2 = H(k,2)$ et ainsi de suite. On s'arrête quand on a trouvé l'élément ou lorsqu'on a parcouru tout le tableau (échec de la recherche) ; pour des raisons d'efficacité, cela pré-suppose que l'on recherche presque toujours des éléments qui sont effectivement dans la table.

Insertion. On parcourt la table avec la fonction de hachage double, jusqu'à trouver la première case libre où on peut insérer un élément.

Suppression. Lorsqu'on supprime un élément, on met la marque à "libre" pour indiquer que cette case pourra être utilisée si elle est accédée pour une nouvelle insertion.

**Analyse du coût.** On suppose $m > n$. Sous l'hypothèse idéale de hachage uniforme, le nombre moyen de cases à examiner lors de la recherche d'un élément est majoré par :

- $\frac{m}{m-n}$ lorsque la clé n'est pas trouvée dans la table;
- $\frac{m}{n}\left(1 - \ln(\frac{n}{m-n})\right)$ lorsque la clé est trouvée dans la table..

Ainsi, dans une table de hachage remplie à 50% (resp. 90%), le nombre moyen de cases à examiner est de 3,4 (resp. 3,7) lorsque la clé est trouvée et de 2 (resp. 10) lorsqu'elle n'est pas trouvée.

**Exercice 4.4** *Programmer les procédures de recherche et d'insertion*

**Correction:**

Les deux procédures s'écrivent immédiatement; lors de la programmation, il faut optimiser l'appel à $H$ pour éviter de recalculer plusieurs fois le hachage de la clé.

```
int insertion_dispersee ( Table T, Elt k) {
   // retourne l'indice dans la table du nouvel element
   int kh2 = h2(k) ;
   int cour = h1(k) % m ; // m == T.size()
   for (int i=0 ; (i < m); i++) {
     cour = (cour + kh2) % m ;
     if (T[cour] == null) {
       T[cour] = k ;
       return cour ;
     }
   } ;
   throw Table_Hachage_Depassement() ;
} ;

bool  recherche_dispersee ( Table T, Elt k, int& indice ) {
   // retourne vrai si l'elt k est dans T et affecte dans
   // indice son indice dans T. Retourne faux sinon.
   int kh2 = h2(k);
   int cour =  h1(k) % m ; // m == T.size()
   for (int i=0 ; (i < m); i++) {
     cour = (cour + kh2) % m ;
     if (T[cour] == k) {
```

```
      indice = cour;
      return true ;
    }
    if (T[cour] == null) return false ;
  } ;
} ;
```

## 4.4 Arbres binaires de recherche

Nous allons maintenant étudier une structure qui, contrairement aux deux précédentes, offre simultanément des coûts moyens en $O(\log(n))$ pour toutes les opérations (ajouts, suppressions, recherches). Nous rappelons dans un premier temps la notion d'arbre binaire de recherche (ABR) et étudions la complexité des opérations qui lui sont associées. Nous montrons qu'il est possible d'obtenir des coûts logarithmiques même dans le cas le pire, grâce à l'utilisation d'AVL, arbres binaires de recherche équilibrés. Enfin, nous présentons une méthode de construction de l'arbre binaire de recherche optimal, dans le cas où le dictionnaire est initialisé au départ par des éléments dont on connaît la probabilité d'être recherchés.

### 4.4.1 Opérations sur un ABR

**Rappels :**

- Un **arbre binaire de recherche** est un arbre binaire tel que la valeur en tout noeud est inférieure ou égale aux valeurs des noeuds de son sous arbre droit et strictement supérieure à celles de son sous arbre gauche.
- L'ajout d'un élément se fait en insérant l'élément récursivement soit dans le sous arbre gauche, soit dans le sous arbre droit, en fonction de sa valeur comparée à celle de la racine (voir Figure 4.3).
- La recherche peut être de même programmée récursivement, en comparant l'élément cherché à la racine, puis en relançant si nécessaire une recherche soit dans le sous arbre gauche, soit dans le sous arbre droit.
- La suppression peut être effectuée en recherchant l'élément à supprimer, puis en remplaçant ce dernier par l'élément le plus à droite de son sous arbre gauche (ou le plus à gauche de son sous arbre droit), qui est lui même remplacé par son unique sous arbre. Voir Figure 4.3.

**Exercice 4.5** *1. Écrire la procédure de suppression dans un arbre binaire de recherche, ainsi que la procédure* `sup-max` *de suppression du maximum d'un sous arbre qu'elle utilise.*

*2. Donner la complexité en place mémoire, ainsi que les coûts dans le cas le meilleur et dans le cas le pire des opérations sur un ABR.*



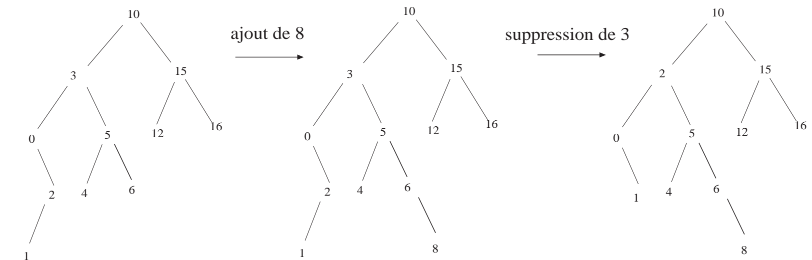


Figure 4.3: Ajout et suppression dans un arbre binaire de recherche.

**Solution :** 1. Algorithme de suppression.

```
procedure sup-max(max: out V; A: in out ABR(V)) is
debut
  si A.d = null
    max := A.val;
    A := A.g;
  sinon sup-max(max, A.d);
  fin si;
fin sup-max;

procedure supprimer(x: in V; A: in out ABR(V)) is
  max : V;
debut
  si A=null alors return; fin si;
  si x < A.val alors supprimer(x, A.g);
  sinon si x > A.val alors supprimer(x, A.d);
  sinon                  -- on a trouve la valeur x a supprimer
    si A.g = null alors A:= A.d;
    sinon si A.d = null alors A:= A.g;
    sinon                -- deux sous-arbres non nuls
      sup-max(max, A.g);
      A.val := max;
    fin si;
  fin si;
fin supprimer;
```

2. Complexités pour un ABR de taille $n$ :

- Place mémoire $O(n)$.
- Recherche : le cas le meilleur est la recherche de la racine, en $O(1)$, et le cas le pire, qui correspond à la recherche du dernier élément d'un ABR dégénéré (voir Figure 4.4), est en $O(n)$.
- Ajout : l'ajout comprend une phase de recherche de la place, puis une opération d'ajout en temps constant. Le cas le meilleur est donc en $O(1)$ (ex : ajout de 5 dans l'arbre de la Figure 4.4), et le pire en $O(n)$ (ex : ajout de 0).

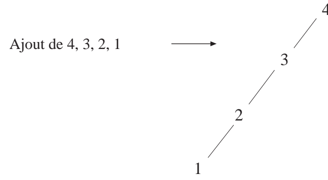


Figure 4.4: ABR dégénéré.

- Suppression : comme pour les tableaux et les listes, on ne compte pas ici le temps de recherche de l'élément à supprimer. La suppression passe cependant par la recherche de l'élément le plus à droite du sous arbre gauche, et les coûts min et max sont donc respectivement $O(1)$ et $O(n)$.

### 4.4.2 Complexités en moyenne des opérations

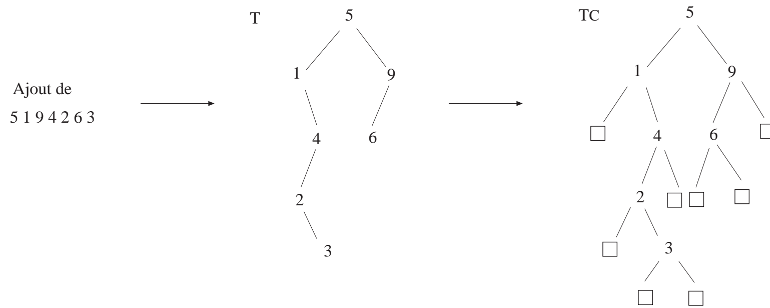


Figure 4.5: $T$ et son complètè $T_c$.

Soit $T$ un ABR à $n$ noeuds, et son **complété** $T_c$ crée en ajoutant $n+1$ feuilles à $T$ de manière à former un arbre localement complet. D'après les définitions données dans l'annexe :

- Le coût moyen d'un **ajout** ou d'une **recherche négative** dans $T$ est :

$$PE(T_c) = \frac{1}{n+1} LCE(T_c)$$

- Le coût moyen d'une **suppression** ou d'une **recherche positive** dans $T$ est :

$$PI(T_c) = \frac{1}{n} LCI(T_c)$$

Puisque l'on cherche à calculer une moyenne sur les $n!$ arbres binaires de recherche de taille



$n$ (supposés équiprobables), on désire évaluer :

$$\begin{aligned} PE_n &= \frac{1}{n!} \sum_{T \in ABR(n)} PE(T_c) \\ &= \frac{1}{n!(n+1)} \sum_{T \in ABR(n)} LCE(T_c) \end{aligned}$$

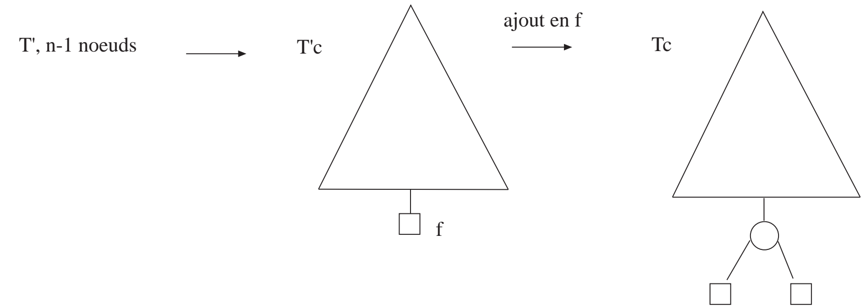


Figure 4.6: Calcul de $LCE(T_c)$, $T_c$ étant créé en ajoutant un noeud en $f$ sur $T'_c$.

Cherchons une relation de récurrence sur $PE_n$. D'après la Figure 4.6, si on suppose que $T$ a été créé à partir d'un arbre $T'$ à $n-1$ noeuds, en insérant un élément dans la feuille $f$ de $T'_c$, on a :

$$\begin{aligned} LCE(T_c) &= LCE(T'_c) - h(f) + 2(h(f)+1) \\ &= LCE(T'_c) + h(f) + 2 \end{aligned}$$

D'où :

$$\begin{aligned} \sum_{ABR(n)} LCE(T_c) &= \sum_{ABR(n-1)} \sum_{f \text{ feuille de } T'_c} (LCE(T'_c) + h(f) + 2) \\ &= \sum_{ABR(n-1)} \sum_{f} LCE(T'_c) + \sum_{ABR(n-1)} \sum_{f} h(f) + \sum_{ABR(n-1)} \sum_{f} 2 \\ &= n \sum_{ABR(n-1)} LCE(T'_c) + \sum_{ABR(n-1)} LCE(T'_c) + 2n(n-1)! \\ &= (n+1) \sum_{ABR(n-1)} LCE(T'_c) + 2n(n-1)! \end{aligned}$$

On a donc :

$$\begin{aligned} PE_n &= \frac{1}{n!(n+1)} \sum_{T \in ABR(n)} LCE(T_c) \\ &= \frac{1}{n!} \sum_{ABR(n-1)} LCE(T'_c) + \frac{2}{n+1} \end{aligned}$$

D'où la relation de récurrence :

$$\begin{aligned} PE_n &= PE_{n-1} + \frac{2}{n+1} \\ PE_1 &= 1 \end{aligned}$$

On en déduit :

$$PE_n = 2\sum_{1}^{n+1}\frac{1}{i} - 2 = 2H_{n+1} - 2$$

où le nombre harmonique $H_n$ est équivalent à $\log(n)$.

Puisque $T_c$ est localement complet, $LCI(T_c) = LCE(T_c) - 2n$, d'où :

$$\begin{aligned} PI_n &= \frac{1}{n!}\sum_{ABR(n)}\frac{1}{n}LCI(T_c) \\ &= (1+\frac{1}{n})PE_n - 2 \\ &= 2(1+\frac{1}{n})H_{n+1} - 4 \end{aligned}$$

**Conclusion :** Les coûts moyens des recherches positives ou négatives, des ajouts et des suppressions dans un arbre binaire de recherche sont de l'ordre de $\log(n)$.

**Remarque 11 ABR optimal** *Jusqu'à présent on a cherché à optimiser le temps de recherche dans un ABR dans le pire cas, ou encore lorsque tous les éléments ont la même probabilité d'être recherchés. Dans le cas où cette hypothèse est fausse, l'arbre doit être déséquilibré pour diminuer le temps de recherche des éléments les plus fréquemment recherchés (qui doivent être les moins profonds). Nous verrons dans le chapitre 13 un algorithme pour construire l'arbre optimal lorsque les probabilités d'accéder chaque élément sont connues.*

## 4.5 Autres arbres de recherche - Les AVL

Comme nous venons de le voir, si les coûts moyens sont intéressants dans un ABR, il n'en reste pas moins que les coûts dans le cas le pire restent en $O(n)$, l'arbre pouvant être dégénéré. Aussi différentes structures de données ont été proposés pour maintenir équilibré l'arbre tout au long de sa manipulation :

- les AVL : l'équilibre est maintenu par des rotations;
- les arbres 2-3 (et plus généralement les B-arbres) : l'équilibre est maintenu en adaptant le degré d'un noeud (2 ou 3 pour un arbre 2-3);
- les arbres bi-colorés, ou arbres rouge-noir, peuvent être vus comme une implémentation d'un arbre 2-3 avec un arbre binaire.

Nous étudions ici la structure AVL, introduite en 1960 par Adelson, Velskii et Landis, qui permet d'assurer un bon équilibrage d'un arbre de recherche.



**Définition :** On dit qu'un arbre est **équilibré** si en tout noeud de l'arbre, les hauteurs des sous-arbres gauche et droit - comptées en nombre de sommets - diffèrent au plus de 1.

Plus précisément, on définit le **déséquilibre** $d(A)$ **d'un arbre** par la différence des hauteurs de ses sous-arbres : $d(A) = h(A.g) - h(A.d)$. Un arbre $A$ est alors équilibré si et seulement si, pour tout sous arbre $S$ de $A$, $d(S) \in \{-1, 0, 1\}$. Voir figure 4.7.

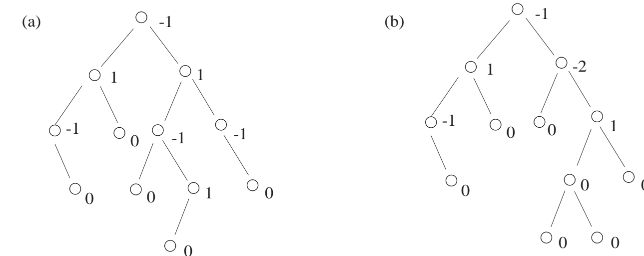


Figure 4.7: Un arbre équilibré (a), et un qui ne l'est pas (b). Les valeurs de déséquilibre pour chaque sous arbre sont indiquées près de leurs racines.

**Définition :** Un **AVL** est un arbre binaire de recherche équilibré.

**Propriété :** La hauteur d'un arbre équilibré est de l'ordre de $\log_2(n)$. En conséquence, le coût dans le cas le pire d'une recherche dans un AVL est un $O(\log_2(n))$.

La propriété précédente montre que la structure AVL est bien ce que nous recherchons, puisqu'elle permet de limiter le coût maximal d'une recherche. Il nous reste à vérifier que les opérations d'ajout et de suppression d'éléments, qui doivent maintenir le caractère équilibré, de l'arbre, peuvent également être réalisées en $O(\log_2(n))$ dans le cas le pire.

**Exercice 4.6** *Rééquilibrage lors de la construction d'un AVL.*

1. *Dessiner les étapes de la construction de l'AVL correspondant aux ajouts successifs des éléments : 12, 3, 2, 5, 4, 7, 9, 11, 14, 10.*
2. *Caractériser les différentes opérations de rééquilibrage que vous avez été amenés à utiliser.*

Solution : Les étapes 3 et 5 font appel à ce qu'on appelle une "rotation droite", et l'étape 6 à son symétrique, la "rotation gauche". Dans ces deux cas, le sous arbre sur lequel s'effectue la rotation est "suspendu" par l'un des fils de la racine. Des rotations plus complexes,appelées "rotations doubles" sont mises en évidence dans les dernières étapes. La nouvelle racine du sous arbre concerné est alors un petit fils de l'ancienne.

**Ajout dans un AVL : synthèse.** L'ajout d'un élément dans un AVL se fait comme dans un arbre binaire de recherche standard, mais est éventuellement suivi d'une opération de rééquilibrage sur le premier sous arbre sur lequel apparaît un déséquilibre, en partant du bas. Quatre opérations peuvent être utilisées pour le rééquilibrage, suivant la configuration

des sous-arbres. Nous en donnons deux sur la figure 4.8, les deux autres étant obtenus par symétries droite/gauche. Ces opérations demandant un nombre constant de comparaisons, le coût maximal d'un ajout est bien en $O(\log_2(n))$.

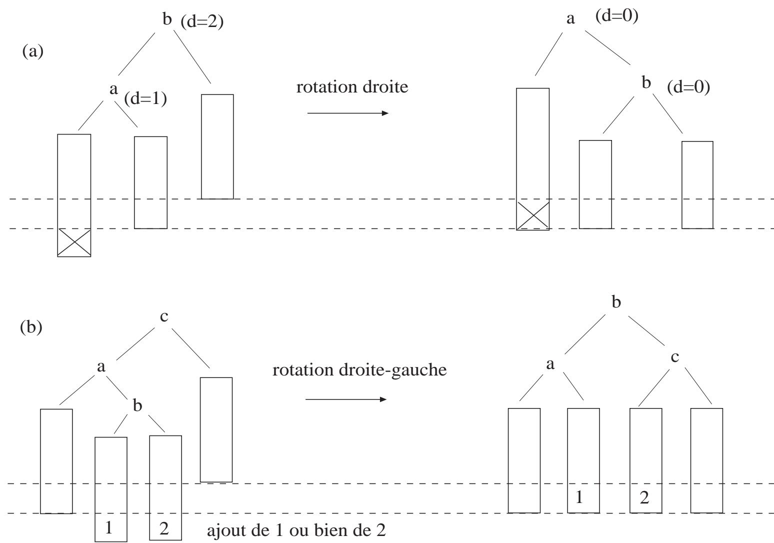


Figure 4.8: Rééquilibrage par rotation droite (a), et par rotation gauche-droite (b).

**Exercice 4.7** *Suppression dans un AVL. Nous allons travailler sur un exemple pour déterminer si la suppression dans un AVL se résout aussi simplement que l'ajout. Supposons que l'on désire supprimer l'élément 12 (la racine) sur l'arbre de la figure 4.9. Une première étape consiste à remplacer 12 par 11, qui est l'élément le plus à droite de son sous arbre gauche.*

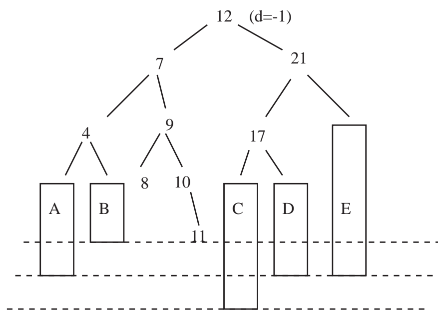


Figure 4.9: Suppression de la racine dans un AVL.

1. *Une seule rotation est-t-elle suffisante pour rétablir l'équilibre (comme dans le cas d'une ajout), ou faudra-t-il recourir à des rotations en chaîne ?*



2. *En déduire le coût dans le cas le pire d'une suppression dans un AVL (on ne demande pas une démonstration formelle).*

Réponse : 1. Cet exemple montre qu'après une suppression, il n'est pas suffisant de procéder à un rééquilibrage (ici une rotation gauche) du sous arbre déséquilibré le plus bas. En effet, le problème peut se propager plus haut dans l'arbre (une seconde rotation, droite-gauche, est nécessaire sur l'exemple).

2. Au pire, ces rééquilibrages pourraient être nécessaires à tous les niveaux de l'arbre, du premier sous arbre concerné jusqu'à la racine. Comme la hauteur de l'arbre est un $O(\log_2(n))$, et que chaque rotation s'effectue en temps constant, ces opérations ont une complexité maximale de $O(\log_2(n))$, qui vient s'ajouter à celle de la suppression standard dans un arbre binaire de recherche équilibré. Au total, on garantit donc bien une complexité de $O(\log_2(n))$ dans le cas le pire pour une suppression dans un AVL.

## 4.6 Les "splay trees"

Nous avons vu que pour un ABR le coût d'une opération pouvait être en $O(n)$, et que cela dépend de l'arbre. En moyenne sur TOUS les arbres elle est en $O(\log n)$, mais une fois que l'on a un mauvais arbre, on est mal parti pour toujours. Il existe des ABR équilibrés pour lesquels les opérations se font dans le pire cas et en moyenne en $O(\log n)$, mais leur programmation est malaisée. On a vu les AVL, il existe aussi les arbres bicolorés (ou arbres rouge-noirs) et les arbres 2-3-4, qui eux ne sont plus binaires.

L'idée des “splay trees” est de conserver une srtucture "simple" d'ABR, mais de le rééquilibrer partiellement pour assurer un coût *amorti* (cf §1.6) en $O(\log n)$. Autrement dit, on peut effectuer des opérations coûteuses, à condition qu'elles soient rares.

Un *splay tree* est un arbre binaire de recherche mais chaque fois que l'on fait une opération, on amène l'élément en question à la racine. Quand on ajoute ou recherche, l'élément en question devient la racine (pour la recherche, s'il existe, snon c'est le dernier élément trouvé qui le devient. Quand on supprime, on le met à la racine, puis on le supprime. L'idée derrière l'opération de suppression n'est pas bien claire, par contre pour les autres, les éléments souvent recherchés seront assez haut dans l'arbre. En plus chaque opération que l'on va voir, diminue la profondeur des nœuds du chemin d'accès à l'élément (en moyenne leur profondeur va être divisée par deux, alors que les autres sommets ne verront leur profondeur augmenter au plus de deux. Noter que la profondeur totale peut augmenter.

Pour une analyse (pas triviale) du coût amorti voir:
`http://www.eli.sdsu.edu/courses/fall95/cs660/notes/splay/Splay.html`. d'où proviennent les images et le texte ci-dessous. Si vous faites une recherche à splay tree sous Google, vous trouverez plusieurs sites qui vous proposent des animations Java.

Soit $p(x)$ le père du nœud $x$. Les trois opérations de base, qui s'appellent des *rotations*, dépendent de la position de $p(x)$ :

1. **zig** : $p(x)$ = racine de l'arbre (fig. 4.10) ;
2. **zig-zig** : $p(x)$ n'est pas racine de l'arbre et $x$ et $p(x)$ des fils gauches (respectivement droites) (fig. 4.11) ;
3. **zig-zag** : $p(x)$ n'est pas racine de l'arbre et $x$ est un fils gauche (resp. droite) et $p(x)$ est un fils droite (respectivement gauche). En d'autres termes $x$ et $p(x)$ sont des fils

de type opposés (fig. 4.12).

Pour “splayer” un nœud $x$, on répète ces opérations jusqu’à ce que $x$ devienne la racine; la figure 4.13 montre un exemple.

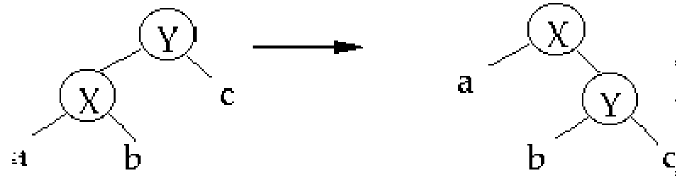


Figure 4.10: Cas 1 : opération zig

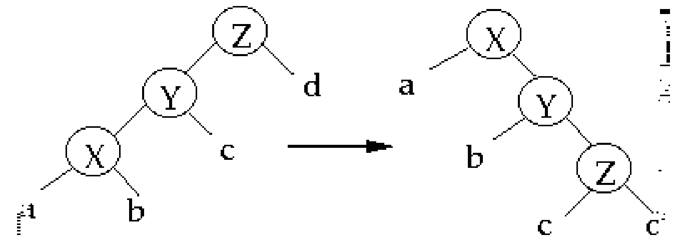


Figure 4.11: Cas 2 : opération zig-zig

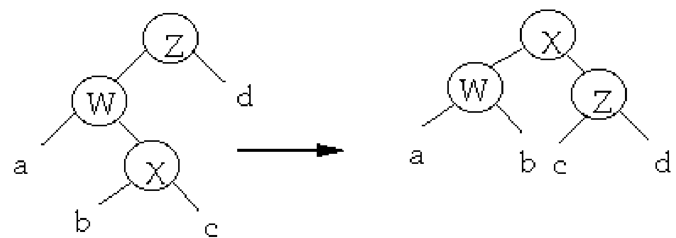


Figure 4.12: Cas 3 : opération zig-zag



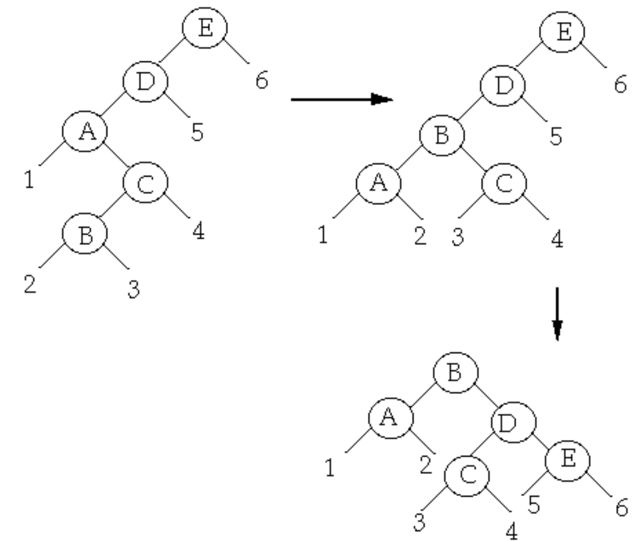


Figure 4.13: Exemple : “splay” de $B$

# Chapitre 5

# Bilan sur les structures de données

## 5.1 Utilisation des structures classiques

Nous avons passé en revue au cours des chapitres précédents les structures les plus classiques pour le stockage d'un ensemble d'éléments, la recherche, ou l'extraction d'un élément de priorité maximale. Rappelons qu'en pratique, le choix d'une implémentation dépend étroitement du problème à résoudre, des opérations qui seront les plus utilisées en pratique, et de la nature même des données. Par exemple, on ne fera pas le même choix pour la représentation d'un dictionnaire selon qu'il sera utilisé pour des recherches uniquement, ou pour des opérations successives d'ajout, de recherche et de suppression.

Le tableau du paragraphe 5.1.1 rappelle les résultats que nous avons obtenus. Résoudre un problème concret signifie aussi savoir combiner des structures entre elles si nécessaire, pour bénéficier simultanément des avantages qu'elles offrent. Nous en donnons un exemple au paragraphe 5.1.2.

### 5.1.1 Tableau récapitulatif

On utilise les structures de données pour la représentation d'ensembles de valeurs sur lesquelles on souhaite vouloir agir à travers une ou plusieurs des opération suivantes :

- appartient : `Ens x elt -> booleen`
- supprimer : `Ens x place -> Ens`
  Ici place indique la place ù la suppression se fait. Noter que la recherche de la place est souvent faite dans la fonction appartient, et qu'il faut donc moduler la complexité indiquée par la recherche de cette place si nécessaire.
- insérer : `Ens x elt -> Ens`
  On suppose que l'on recherche d'abord si l'élément y est, on insère ensuite.
- trouver-min : `Ens -> elt`
- parcourir : `Ens x fonction -> ensemble de résultats`
- union-multiple : `Ens x Ens -> Ens`
  Ici on autorise la duplication des éléments, précaution inutile si on sait que les ensembles sont disjoints a priori.





Les structures possibles sont :

- listes consécutives (tableaux) ou chaînées, triées ou non, pour au total $n$ éléments.
- table de hachage (hash-code). On suppose qu'elle est construite de manière à ce que les éléments ayant même code soient en un nombre réduit, $O(1)$.
- table en séquentiel indexé : cette structure est celle des fichiers de même nom et elle consiste en une table triée de clés telle que l'entrée $i$ fournit permet l'accès à la listes triée des éléments de clés comprises entre $table(i)$ et $table(i+1)$. La recherche se fera donc de façon dichotomique dans la table, puis en accès séquentiel dans la liste. La fonction de réorganisation du gestionnaire de fichier garantit que ces listes ne sont pas trop grandes et équilibrées. On suppose que l'on a $m$ sous listes de taille moyenne $n/m$.
- tas implantant une file de priorité.
- arbre de recherche que l'on supposera équilibré.
- tableau de témoins : une telle table suppose que le référentiel des éléments soit fini et permette l'indexation. Le référentiel est de taille $k(kn)$ et élément présent a son booléen positionné. On suppose que la machine est capable de faire des opérations booléennes sur des mots de $p$ bits (typiquement $p = 32$).

Ne figurent pas dans le tableau qui suit la complexité de la programmation de chaque structure, considération qui doit aussi être prise en compte lors du choix d'implantation.

On obtient alors le tableau suivant :

| | appart. | insere | suppress | trouver min | union | union multiple | parcours |
|---|---|---|---|---|---|---|---|
| consec. non trié | $n$ | $n$ | $n/2$ | $n$ | $n^2$ | $n$ | OK |
| consec. trié | $logn$ | $n/2$ | $n/2$ | 1 | $n$ | $n$ | OK trié |
| chaîné non trié | $n$ | $n/2$ | 1 | $n$ | $n^2$ | 1 | OK |
| chaîne trié | $n/2$ | $n/2$ | 1 | 1 | $n$ | $n$ (*) | OK trié |
| sequent. indexé | $\log m$ $+ n/m$ | $\log m$ $+ n/m$ | 1 | 1 | $n$ | $n$ (*) | OK trié |
| hash code | 1 | 1 | 1 | possible en $n$ | possible en $n$ | possible en $n$ | possible en $n$ |
| tas | $n$ | $n+$ $\log n$ | $\log n$ | 1 | $n^2$ | $n \log n$ | possible en $n$ |
| arbre recherche | $logn$ | $logn$ | $logn$ | $logn$ | $nlogn$ | $nlogn$ | OK trié |
| tableau témoins | 1 | 1 | 1 | n | $k/p$ | $k/p$ | p. en $k$ |

(*) : on suppose que le résultat doit être trié, et donc il faut interclasser les deux listes et non simplement les concaténer.

**Discussion :** On notera que le tableau des témoins est une structure intéressante, et que son emploi est donc recommandé chaque fois que ce sera possible.

L'intérêt des listes triées (et du séquentiel indexé) vient essentiellement de la possibilité de faire des intersections et des unions en $O(n)$. Ces opérations sont essentielles dans la manipulation des gros fichiers en informatique de gestion (fichier des factures à intersecter avec celui des clients par exemple).

Les constantes multiplicatives ne sont pas indiquées ici. La coût du calcul de la clé de hash code n'est pas négligeable et réduit un peu l'intérêt de cette approche qui semble ressortir comme l'une des plus performante. En outre, il faut pour cette approche que la table majeure soit bien surdimensionnée pour garantir un accès à des sous listes courtes (typiquement 10 à 20% de plus que le nombre max d'éléments).

Il faut moduler le coût de l'insertion pour les problèmes où l'on sait que l'élément n'est pas dans la table. Ainsi pour une liste non triée on passe alors à un coût en 0(1).

### 5.1.2 Structures hybrides (exercice)

L'exercice qui suit montre qu'il ne suffit pas toujours de connaître et d'utiliser les structures classiques. Il peut être très intéressant en pratique de les combiner entre elles sous forme de structures hybrides qui permettent de marier leurs avantages.

**Exercice :** On veut implanter un ensemble de valeurs qui possède les caractéristiques suivantes :

- le nombre d'éléments à manipuler est majoré par 100000,
- les éléments sont distingués par une chaîne de caractères,
- à cette chaîne est attachée une valeur numérique,
- les opérations que l'on a à effectuer sont : insertion, suppression, recherche du min sur la valeur, appartenance. Toutes ces opérations sont équiprobables à chaque instant.
- le temps de réponse moyen est critique.

Quel est votre choix d'implantation pour ce problème ?

**Solution :** Un arbre de recherche équilibré, du type AVL, est une bonne réponse.

On peut cependant optimiser en croisant une table de hash code dont la clé est la chaîne de caractères avec un tas dont la clé est la valeur numérique (il est dommage que l'on ne puisse pas le faire ici entre un tableau de témoins et un tas car le nombre d'entrée potentielles sur les chaînes de caractère est presque infini ; c'eût été plus facile). Des pointeurs d'une structure vers l'autre maintiennent la cohérence (voir figure 5.1). Il faut être conscient des problèmes que pose alors la mise à jour : l'insertion d'un élément implique une réorganisation du tas et donc il faut que les pointeurs des éléments vers leur place dans le tas soient mis à jour.

Cette structure correspond aux déclarations suivantes :

```
max_element: constant positive := 10**5;
type entree_table is natural range 0..integer(1.2 * max_element);
type acces_tas is integer range 1..max_element; -- indice dans tas

type element ;
type lien_hash is access element\,;  -- lien de chainage des elements
type element is record
cle: chaine;
valeur: integer;
suivant: lien_hash;
```



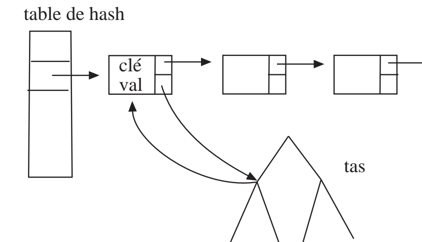


Figure 5.1: Structure hybride combinant deux structures classiques : table de hachage et tas.

```
dans_tas: acces_tas; end record;

ma_structure: record
nbre_element: positive := 0;
tete_liste_hash: array(entree_table)
      of lien_hash := (other => null);
tas: array(1..max_element) of lien_hash;
end record;
-- on notera la surdimensionnement de la table de hash, le tas
-- lui ne necessitant pas de precautions particulieres.
```

On vérifie directement que les recherches de minimum dans le tas et les suppressions se font en conjuguant les différents accès :

- l'appartenance se fait dans la table en hash-code ;
- la recherche du min se fait dans le tas ;
- l'insertion comprend une insertion dans la table en hash-code et une insertion dans le tas. Chaque modification de place dans le tas se répercute par une modification du lien `dans-tas` et une incrémentation du compteur `nbre-element` ;
- une suppression de clé donnée comprend une suppression dans la table de hash-code et une suppression dans le tas. Chaque modification de place dans le tas se répercute par une modification du lien `dans-tas` et une décrémentation du compteur `nbre-element` ;
- pour la suppression du min, après avoir accédé à la clé, on se ramène à la suppression précédente.

# Parcours de Graphes

L'objectif de cette partie est de fournir un ensemble d'algorithmes de recherche de chemins dans des graphes, de les comparer. Ces algorithmes sont des classiques à connaître en particulier ils sont à la base de nombreuses variantes qui ne seront pas exposées ici. Leurs applications sont variées ; ils sont en particulier utilisés pour la recherche de solutions pour des problèmes variés tels que les jeux ou l'optimisation de ressources.

Le plan de cette partie est le suivant :

- le chapitre 6 donne le vocabulaire de base sur les graphes ainsi que les représentations les plus classiques;
- les algorithmes d'exploration d'un arbre (i.e. graphe orienté sans circuit avec racine) sont rappelés dans le chapitre 7, puis étendus au cas de graphe quelconques;
- le chapitre 8 étudie le calcul de la fermeture transitive, i.e. de tous les sommets atteignables dans un graphe quelconque.
- les algorithmes de recherche du plus court chemin élémentaire issu d'un sommet sont étudiés dans le chapitre 9
- en conclusion, différents problémes applicatifs qui se ramènent au calcul de chemins dans un graphe sont présentés dans le chapitre 10.

# Chapitre 6

# Graphes : vocabulaire et représentations classiques

## 6.1 Vocabulaire

La terminologie des graphes peut varier d'une communauté à l'autre. Il sera donc toujours bon de s'assurer de l'exactitude des termes utilisés avec ses interlocuteurs.

Un graphe $\mathcal{G} = (\mathcal{X}, \mathcal{E})$ est donné par un ensemble de **sommets** $\mathcal{X}$ (appelé parfois aussi **points** ou encore **nœuds**), et d'une **relation** $\mathcal{E}$ contenue dans $\mathcal{X} \times \mathcal{X}$ ($\mathcal{E}$ like "edge"). Par la suite, sauf notation contraire, nous supposerons que $n = card(\mathcal{X})$ et $m = card(\mathcal{E})$. On a bien sûr $m \leq n^2$. Pour beaucoup de graphes le nombres d'arcs $m$ est bien plus petit que cette borne supérieure, il est souvent une fonction linéaire du nombre de sommets ; il faut donc bien faire attention que $m$ et $n^2$ peuvent avoir des ordres de grandeur très différents.

On notera $\mathcal{I}$ la relation identité qui ne correspond qu'aux **boucles** $(x, x)$ du graphe.

- Un graphe est dit **complet** si $\mathcal{E}$ est la relation pleine : $\mathcal{E} = \mathcal{X} \times \mathcal{X}$. On accepte aussi ce terme de complet si les boucles sont absentes.
- Un graphe est dit **non orienté** si la relation $\mathcal{E}$ est symétrique :

$$(x, y) \in \mathcal{E} \Longrightarrow (y, x) \in \mathcal{E}$$

  sinon le graphe est dit **orienté**.
- Le **symétrisé** d'un graphe est le plus petit graphe symétrique contenant le graphe initial. Il est donc défini par la relation $\mathcal{S} = \mathcal{E} \bigcup \mathcal{E}^{-1}$
- On appelle **arc** tout élément de $\mathcal{E}$. Si le graphe n'est pas orienté, on parle alors d'**arête** pour désigner indifféremment $(x, y)$ ou $(y, x)$.

  $x$ est l'**origine** ou **extrémité initiale** de l'arc $(x, y)$, et $y$ son **extrémité terminale**. Et dans le cas non orienté on ne parlera que d'extrémités.
- Le **demi-degré intérieur** (resp. extérieur) d'un sommet est le nombre d'arcs ayant ce sommet pour extrémité finale (resp. origine). Le **degré** est la somme de ces deux valeurs. Notez que l'arc $(x, x)$ sera compté deux fois dans le degré.





- Un **chemin** de longueur $k$, d'origine $x_0$ et d'extrémité $x_k$ est une suite $x_0, \ldots, x_i, \ldots, x_k$ telle que les $(x_i, x_{i+1})$ sont des arcs. Une **chaîne** est l'équivalent non orienté du chemin. Un chemin vide est un chemin de longueur 0.
- Un **circuit** est un chemin de longueur $\neq 0$ tel que $x_0 = x_n$. Un **cycle** est la version non orienté du chemin.
- Un chemin est dit **élémentaire** s'il ne contient pas deux fois le même sommet. Il sera dit simple s'il ne contient pas deux fois le même arc. Cette définition s'étend naturellement aux chaînes, circuits et cycles. De fait, en France l'habitude est de considérer que chaînes, circuits et cycles sont toujours simples.
- Un graphe est **connexe** si tout couple de sommets est relié par une chaîne du graphe symétrisé. On appelle **composante connexe** tout sous-ensemble maximal de sommets qui est connexe.
- Un chemin est **hamiltonien** si tout sommet de $\mathcal{G}$ n'est visité qu'une fois et une seule. Cette notion s'étend bien sûr aux chaîne, circuit et cycles. Un chemin est dit **euliérien** si tout arc apparait une fois et une seule dans le chemin.

**Exercice 6.1** *: Mémoriser toutes ces définitions en dessinant les graphes et les notions correspondantes.*

## 6.2 Calcul relationnel

Trois types de calculs sont faits habituellement sur les relations :

**inversion :** $\mathcal{E}^{-1} = \{(y, x) \mid (x, y) \in \mathcal{E}\}$
Intuitivement on inverse les flèches des arcs.

**produit :** Soient $\mathcal{E}$ et $\mathcal{S}$ deux relations, on définit $\mathcal{E} \circ \mathcal{S}$ par

$$\mathcal{E} \circ \mathcal{S} = \{(x, z) \mid \exists y (x, y) \in \mathcal{E} \text{ et } (y, z) \in \mathcal{S}\}$$

C'est donc le parcours que l'on peut faire en faisant d'abord un pas avec $\mathcal{E}$ suivi d'un pas avec $\mathcal{S}$. On notera naturellement

$$\mathcal{E}^k = \overbrace{\mathcal{E} \circ \mathcal{E} \ldots \circ \mathcal{E}}^{k}$$

**fermeture transitive :** c'est la plus petite relation transitive et réflexive qui contienne $\mathcal{E}$. Donc par définition c'est le plus petit point fixe du système d'équation :

$$\mathcal{E}^* = \mathcal{I} \bigcup \mathcal{E} \circ \mathcal{E}^*$$

Par définition $\mathcal{E}^*$ contient l'identité $\mathcal{I}$ et $\mathcal{E}$. Donc par transitivité $\mathcal{E}^2$, et donc $\mathcal{E}^3$, etc. On peut vérifier aisément que $\mathcal{I} \bigcup (\cup_{i=1}^{\infty} \mathcal{E}^i)$ est une relation transitive et réflexive, et par conséquent la relation minimale recherchée.

Pour tout couple de sommets $(x, y)$, s'il existe un chemin de $x$ à $y$, alors il existe un chemin de longueur au plus $n-1$ de $x$ à $y$. On en déduit donc que

$$\mathcal{E}^* = \mathcal{I} \bigcup \left( \bigcup_{i=1}^{n-1} \mathcal{E}^i \right)$$

Intuitivement, la fermeture transitive permet de déterminer l'ensemble des sommets que l'on peut atteindre en un nombre quelconque de pas dans le graphe depuis un sommet de départ. Ce calcul intervient fréquemment dans des problèmes très divers ; ceci justifie l'importance de la fermeture transitive et le fait que nous lui consacrerons un developpement particulier (cf la partie 8).

## 6.3 Représentation des graphes

Les graphes, à la différence des structures de données classiques, n'ont pas de représentation standard mais, au gré des problèmes qui sont traités, des représentations classiques. Chacune à des avantages et nous allons résumer les deux représentations principales pour les discuter :

### 6.3.1 Représentation matricielle

```
type sommet is natural range 1 .. N ;
type graphe is array (sommet, sommet) of boolean (ou encore of integer);

G : graphe;
```

$G(i, j)$ : sera soit un booléen pour indiquer l'existence d'un arc de $i$ vers $j$, soit une valeur numérique indiquant la valuation de l'arc.

**Occupation mémoire :** $0(n^2)$ ; soit beaucoup de place de perdue si $n$ est grand et si le graphe n'est pas dense, en revanche, c'est une représentation compacte si le graphe est presque complet.

**Avantages :** accès direct aux arcs depuis les sommets,
formulation classique pour produit de deux relations : c'est le produit matriciel (cf exercice 6.2).

### 6.3.2 Représentation par une table des successeurs

A chaque sommet on associe l'ensemble des sucesseurs (généralement ce dernier est représenté simplement par une liste)

```
type arc_elem ;
type arcs is access arc_elem;  -- les arcs sont une liste chainee ici
type arc_elem is
    record
        extremite : sommet;
        cout :  natural ;
```



```
        suivant : arcs;  end record;

type les_successeurs is record
    nbre : integer := 0; -- nbre de successeurs (optionnel)
    tete : arcs := null;   -- leur liste
        end record;

type table_successeurs is array (positive range /=) of arcs ;
 -- ce tableau donne pour chaque sommet la tete de liste de ses successeurs
```

**Avantages :** une représentation plus compacte pour les graphes creux (les plus fréquents), un accès itératif facile lorsqu'il faut traiter les sucesseurs dans une boucle.

**Variantes :** Selon la nature du problème il sera judicieux d'associer des informations complémentaires à ces représentations de base, par exemple le demi degré intérieur de chaque sommet, le graphe et le graphe inversé, les coûts des arcs ou des valuations de sommets, etc. Seule une analyse des structures nécessaires à l'algorithme permet de faire un choix optimal.

**Exercice 6.2** *: Ecrire les procédures d'inversion et de produit de relations pour les deux représentations ci-dessus, et fournir leur complexité en fonction de $n$ et $m$ (respectivement le nombre de sommets et le nombre d'arcs).*

# Chapitre 7

# Exploration arborescente

Un arbre peut être vu comme un graphe orienté sans circuit (DAG pour *Direct Acyclic Graph*) possédant un sommet racine (i.e. sans prédécesseur). Dans ce cas, l'exploration *arborescente* des sommets à partir de la racine est particulièrement simple. Les explorations arboresentes en largeur et profondeur sont rappelées ainsi que le tri topologique d'un graphe. Etendue au cas d'un graphe quelconque, ce type d'exploration est notamment utilisé dans le cas d'arbres modélisant un jeu.

## 7.1 Rappels sur l'exploration d'un arbre

### 7.1.1 Spécification d'un arbre

Le type `arbre` est spécifié par les générateurs suivants, pour un *arbre binaire* :

```
arbre_vide :         ⟶ arbre
enracine   : info × arbre × arbre ⟶ arbre
```

Pour un arbre à nombre quelconque de nœuds on rajoute le type `liste_arbre` :

```
enracine : info × liste_arbre ⟶ arbre
```

L'implantation du type `arbre` se fera selon les cas :

```
type arbre is access noeud ;

type noeud is record
  info : type_info ;
  fils_gauche, fils_droit : arbre;
end record ;
```

ou, si l'on manipule des listes d'arbres :

```
type noeud is record
  info : type_info ;
  fils_aine, frere_cadet : arbre;
end record
```

On utilise rarement un tableau de $n$ fils dans le cas de fils multiples, mais cela peut présenter des avantages pour accéder directement au k*ième* fils pour des arbres très larges. C'est la façon de transformer les arbres généraux en arbres binaires.





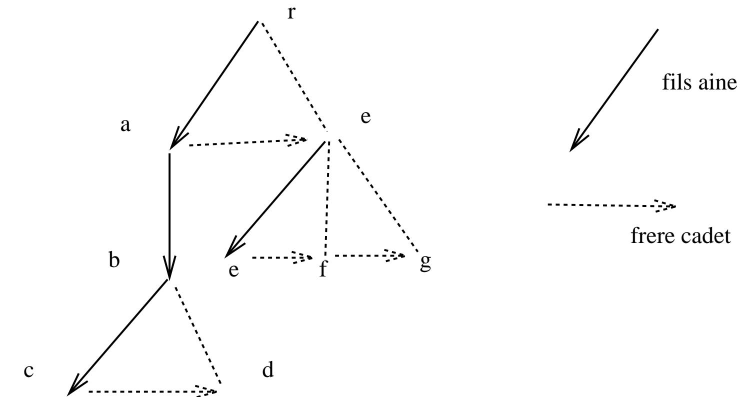


Figure 7.1: Utilisation des liens `fils_ainé` et `frère_cadet` dans le cas d'un arbre à nombre quelconque de fils

### 7.1.2 Exploration générique d'un arbre

#### Exploration en profondeur

Des constructeurs de ce type on dérive directement par récurrence les algorithmes classiques d'explorations et traitement en profondeur :

```
explore_et_traite (arbre_vide) = traitement_vide

explore_et_traite (enracine (inf, ag, ad)) =
                         combinaison (traitement(inf),
                                      explore_et_traite(ag),
                                      explore_et_traite(ad))
```

Selon l'ordre dans lequel on évalue les paramètres de la procédure `combinaison`, on parlera de traitement en préfixé, infixé ou postfixé. L'implantation sous forme de procédure récursive est immédiate.

La complexité de cette exploration est :

- en temps proportionnelle au nombre de nœuds de l'arbre, sous hypothèse d'avoir des temps de traitements qui soient $O(1)$.
- en place, proportionnelle à la taille de la pile de récursivité, qui atteint son maximum pour l'exploration de la branche la plus longue de l'arbre. S'il s'agit d'un arbre binaire équilibré à $n$ nœuds, cela fera donc $O(\log n)$, sinon au pire $O(n)$.

**Exercice 7.1** *: Montrer l'équivalence des deux algorithmes suivants (c'est vu en première année) :*

```
infixe(a) :
     si a /= arbre-vide alors
         infixe (fils-gauche (a));
```

```
        traitement (a) ;
        infixe (fils-droit (a))
    fsi

infixe2 (a) :
    en-cours := a ;
    tant que en-cours /= arbre-vide repeter
        infixe2 (fils-gauche (a))
        traitement (a)
        en-cours := fils-droit (en-cours)
    ftq
```

**Note :** Quelques commentaires sur la *complexité pratique* : les temps d'exécution sont comparables sur les deux versions de l'algorithme de parcours. Cette indications est à prendre avec précaution, car c'est dépendant du compilateur, mais aussi de l'art du programmeur.

#### Exploration en largeur

Cette exploration ne se dérive pas directement du type `arbre`. L'exploration se faisant par niveau, on gère les nœuds d'un même niveau sous forme de file.

L'algorithme devient alors :

```
-- initialisation
file := racine(arbre) ;
-- boucle d'exploration
tant que file /= file-vide repeter
        oter-en-tete (file, x) ;
        traiter (x) ;
        pour chaque fils y de x repeter
                inserer-en-queue (file, y)
        fin pour
ftq
```

Là encore la complexité en temps est proportionnelle au nombre de nœuds. En revanche la taille mémoire nécessaire est de l'ordre du plus grand niveau. Pour un arbre binaire équilibré de $n$ nœuds, cela fait un dernier niveau de $\frac{n+1}{2}$ nœuds !

A cause ce problème de place, cet algorithme, ou ses variantes, était peu utilisé. Les performances en taille mémoire des machines actuelles ont sensiblement changé la situation.

#### Contrôle du parcours

L'algorithme suivant ne diffère du précédent que dans la façon de gérer les nœuds ; un ensemble de nœuds est mis en attente dans une variable :

```
-- initialisation avec un ensemble a 1 element
Attente := singleton (racine (arbre)) ;

tant que Attente /= ensemble-vide repeter
        selectionner-et-retirer (Attente, x) ;
        traiter (x) ;
```



```
        pour chaque fils y de x repeter
                inserer-dans-ensemble  (Attente, y);
        fin pour;
fin tq;
```

Selon les implantations de la procédure `sélectionner_et_retirer`, nous aurons donc des parcours qui peuvent être très différents. Par exemple un critère de qualité nous permet de traiter le meilleur d'abord, par exemple pour espérer trouver le plus vite possible une bonne feuille.

**Exercice 7.2** *: Quel choix prendre pour la fonction* `sélectionner_et_retirer` *pour obtenir une recherche en profondeur (avec l'ordre préfixé) ?*

### 7.1.3 Application : exploration d'arbres combinatoires

Pour beaucoup de problèmes, la construction (ou la recherche) d'une solution passe par la construction d'une solution partielle que l'on peut ensuite prolonger de différentes façons.

Examinons le **problème des $n$ reines** à placer sur un échiquier de taille $n \times n$ sans que deux reines soient en prises. Essayer toutes les combinaisons de $n$ positions pour vérifier si elles répondent à la condition de non prise mutuelle est évidement prohibitif : complexité en $O((n^2)^n)$. On peut décider de placer les reines à raisons d'une seule par colonne, ce qui permet de définir la solution comme la ligne `ligne(i)` occupée par la reine de la ième colonne. On imagine alors que si $k$ reines sont placées, la solution partielle obtenue soient prolongée par le placement d'une reine supplémentaire. La solution est considérée comme partielle car la condition de "non prise" est satisfaite pour le sous ensemble considéré ; ainsi on aura $n$ choix pour la première reine, $n-1$ pour la seconde, etc. et donc la complexité sera majorée par $n!$.

D'où l'algorithme suivant qui considère les colonnes dans l'ordre (ce n'est pas la meilleure stratégie mais la plus simple) :

```
subtype pos_etendu is natural range 0 .. n;
subtype pos is pos_etendu range 1 .. n;
ligne : array(pos) of pos_etendu := (others => 0);

Reine (i : pos_etendu) :
-- entree : i reines sont placées, memorisées dans le tableau ligne
-- au retour, toutes les solutions complétant cette solution partielle
--            sont imprimées

si i=n alors
    -- toutes sont placées : la solution partielle est une solution complète
    -- on est donc a une feuille de l'arbre de recherche
    imprimer (ligne)
sinon
    -- exploration des différentes possibilités
    k := i+1;
    pour j de 1 a n loop
        si position_libre (k,j) alors
            ajouter_une_reine (k,j) ;
```

```
            reine(k) ;
            enlever_une_reine (k,j) ;
        fsi;
    fpour;
fsi;
fin reine;

procedure ajouter_une_reine (k,j : pos):
     ligne(k) := j ;
fin ajouter_une_reine;

preocedure retirer_une_reine (k,j : pos):
     ligne(k) := 0 ;
fin ajouter_une_reine;

-- programme principal :
reine (0);
```

On notera que la libération de lignes par affectation de "0" est inutile. Elle ne figure ici que pour être consistente avec le prédicat `ligne` *indique les lignes occupées par les reines des colonnes correspondantes et vaut 0 sinon.*

**Exercice 7.3** *: Comment implanter le test "position_ libre" en temps constant ?*

**Exercice 7.4** *: Quel problème pose l'implantation d'une recherche en largeur de l'algorithme des n reines*

**Exercice 7.5** *: Sur le même principe, écrire un algorithme qui énumère toutes les combinaisons de k lettres de l'alphabet choisies parmi les n premières ($k < n$).*

## 7.2 Exploration d'un DAG

### 7.2.1 Exploration des sommets d'un DAG

L'exploration d'un DAG est similaire à celle d'un arbre, en partant de (ou des) racines, et en allant vers les feuilles. Comme le graphe est sans cycle, l'algorithme se termine; cependant, pour éviter d'explorer plusieurs fois les fils d'un même noeud, les noeuds explorés sont marqués.

Selon la représentation du graphe, la marque peut être un attribut booléen ajouté à chaque noeud (par héritage par exemple). Si les noeuds ne sont pas stockés directement, le marquage peut aussi être réalisé par une table de hachage dans laquelle sont stockés les noeuds déja explorés.

### 7.2.2 Application: tri topologique d'un DAG

Soit $G = (V, E)$ un graphe orienté sans circuit. Un tri topologique est un ordre total $<$ sur tous les sommets de $G$ tel que :

$$(u, v) \in E \Longrightarrow v < u$$



Une adaptation de l'exploration arborescente en profondeur du DAG permet de calculer un tel tri topologique.

Pour l'algorithme, on suppose que le type sommet possède un attribut booléen `EstDejaAtteint`, initialisé à `false`. Le resultat est stocké dans une pile : le sommet en tête de pile est le plus grand par rapport à l'ordre topologique (i.e. un sommet sans prédécesseur).

```
stack< Sommet > TriTopologique ;

pour tout sommet x dans G.V() faire        (1)
  si  ( x.EstDejaAtteint == false ) alors ParcoursProfondeur ( x ) ;
fait ;


procedure ParcoursProfondeur( Sommet u ) {
  u.EstDejaAtteint = true ;
  pour tout sommet v fils de u faire {     (2)
    si ( v.EstDejaAtteint == false ) alors ParcoursProfondeur ( v ) ;
  }
  TriTopologique.push ( u ) ;
}
```

**Correction :** Chaque fois que ParcoursProfondeur est appelé sur un sommet, il est placé dans la pile. Ainsi, un sommet n'est placé dans la pile TriTopologique qu'après que tous ses successeurs aient été parcouru et déjà mis dans la pile. Par suite, la pile est bien ordonnée selon l'ordre topologique décroissant.

**Coût :** Du fait du marquage, ParcoursProfondeur n'est appelé qu'une fois au plus sur chaque sommet. Le nombre de passages dans la boucle (2) est proportionnel au nombre de fils du sommet, soit en $\Theta(m)$ pour tous les sommets. Avec la boucle (1), le coût est donc $\Theta(n + m)$.

**Exercice 7.6** *Tracer l'exécution de l'algorithme sur un DAG.*

**Exercice 7.7 Un autre algorithme de tri topologique.** *On suppose que tous les sommets ont un attribut* `degre_entrant` *(de type entier) qui donne le nombre d'arcs entrant dans le sommet. On propose l'algorithme suivant :*

```
stack< Sommet > TriTopologique ;
stack< Sommet > pile_tmp ;

pour tout sommet x dans G.V() faire        (1)
  si  ( x.degre_entrant == 0 ) alors  pile_tmp.push( x ) ;
fait ;


tantque ( pile_tmp.empty() == false ) {
  Sommet u = pile_tmp.pop() ;
  TriTopologique.push ( u ) ;
  pour tout sommet v fils de x faire  {
    v.degre_entrant -= 1 ;
    si (v.degre_entrant == 0) pile_tmp.push ( v ) ;
```

```
    }
  }
```

*Montrer que l'algorithme calcule un tri toplogique du graphe (supposé orienté et sans circuit). Donner son coût.*

## 7.3 Parcours arborescent de graphes

### 7.3.1 Exploration arborescente des sommets d'un graphe

L'exploration d'un arbre peut être étendue à un graphe quelconque : il suffit de considérer les successeurs d'un sommet comme les fils d'un nœud dans un arbre. On construit ainsi les algorithmes de parcours "arborescent" d'un graphe. Mais il faut prendre en compte deux changements :

- on peut accéder selon plusieurs chemins à un même sommet,
- l'existence de circuit peut amener l'algorithme à boucler.

Les modifications à apporter aux algorithmes dépendent des objectifs du parcours :

- si l'on souhaite simplement visiter une fois et une seule tous les sommets, il suffit de marquer ceux déjà rencontrés,
- si l'on souhaite construire tous les chemins élémentaires (à partir d'un sommet source par exemple), il faut éviter de boucler et donc être capable de tester si un sommet appartient déjà au chemin que l'on considère.

Ceci conduit à une implantation simple dans le cas de recherche en profondeur qui ne considère qu'un chemin à la fois, mais impose une gestion complexe pour la recherche en largeur car il faut gérer un ensemble de chemins.

### 7.3.2 Enumération de tous les chemins dans un graphe

Si par fait extraordinaire, on s'intéresse à des chemins non élémentaires, alors il faudra prendre des précautions supplémentaires pour éviter de boucler, et cela en fonction du problème traité.

Algorithme d'exploration arborescente de tous les chemins élémentaires :

```
type sommet is natural range 1..N;

chemin is record
    les_points     : array (sommet) of sommet;
    indice_dernier : (0..N) := 0;
    dans_chemin    : array(sommet) of boolean := (others => false) ;
end record;

ch : chemin;

procedure Explore (extremite: sommet) is
----------------
    -- entree : ch contient un chemin partiel dont le dernier
```



```
    --           sommet est extremite
    -- sortie :  Explore construit tous les chemins elementaires
    --           prolongeant ce chemin

pour chaque y sucesseur de extremite repeter
    si non (ch.dans_chemin(y)) alors
           mettre_dans_ch (y) ;
           Explore (y);
           retirer_de_ch (y);
    fsi;
fin pour;
fin Explore;

-- mettre et retirer dans chemin sont evidents

-- appel initial
ch.indice_dernier := 1;
ch.les_sommets(1) := depart ;
Explore(depart);
```

**Exercice 7.8** *: la procédure ci-dessus ne fait que parcourir les chemins ; comment la modifier pour qu'elle compte le nombre de chemins élémentaires maximaux (c.à.d. qui ne peuvent être prolongés) à partir d'un sommet de départ a ?*

**Exercice 7.9** *: Montrer la complexité de l'algorithme d'exploration peut être $O(2^n)$ dans un cas de graphe que l'on construira. Quelle est la complexité en place ?*

*Montrer qu'en programmant attentivement, la complexité d'une telle recherche passe à $O(m)$ si l'on se contente simplement de visiter les sommets acessibles depuis un sommet de départ donné en marquant les sommets visités.*

De fait, l'exploration de tous les chemins possibles est seulement théorique à cause du nombre exponentiel de chemins possibles dans la plupart des graphes. On verra dans la suite comment limiter cette recherche.

### 7.3.3 Recherche de chemins optimaux par exploration

C'est un sous problème courant dans de nombreuses applications, par exemple lorsqu'il faut déterminer le chemin le plus court (le moins cher, le plus probable, etc) dans un graphe. Nous supposerons dans la suite que les coûts sont additifs. Cette hypothèse simple n'est pas toujours vraie. Ainsi certains coûts peuvent dépendre de couple d'arcs consécutifs dans un graphe et non seulement d'un arc. Pour le cas de probabilité, les coûts sont multiplicatifs, mais un simple passage aux logarithmes permet de revenir au cas de coûts additifs. Notez que dans ce cas le coût devient négatif (les probabilités sont inférieures à 1) mais les logarithmes restant monotones, on continuera à rechercher un maximum.

Notons la proposition évidente ci-dessous dont nous ferons un large usage par la suite :

**Proposition :** *Tout sous-chemin d'un chemin optimal est optimal.*

*Preuve :* soit $x, \ldots, u, \ldots, v, \ldots y$ un chemin optimal de $x$ à $y$. Si le sous-chemin de $u$ à $v$ n'était pas optimal, il existerait un sous chemin de $u$ à $v$ qui, substitué au sous-chemin actuel permettrait d'obtenir un chemin de $x$ à $y$ qui soit de coût moindre, ce qui n'est pas possible.

### 7.3.4 Exploration des chemins optimaux en profondeur d'abord

La construction de l'algorithme d'exploration en profondeur d'abord est simplement de la transcription de ce que nous avons vu dans la section précédente. Mais cette fois ci l'objectif n'est plus d'énumérer tous les chemins, mais de trouver le meilleur. On notera que ce problème n'a pas de solution si on a des cycles absorbants : cycles dont le coût total est négatif. Dans ce dernier cas il faut nécessairement rajouter une contrainte de chemin élémentaire. Cette contrainte n'est évidemment pas nécessaire si les *coûts sont strictement positifs*, hypothèse que nous considérerons ici. Nous supposerons de plus que nous avons un sommet `arrivée` que nous devons atteindre.

L'idée élémentaire exploitée est qu'il est inutile de prolonger les chemins dont le coût sera supérieur à celui de la meilleure solution actuellement trouvée ; on associe donc un champ `cout` au type chemin, et dans la procedure d'exploration une variable `alpha` indiquera le coût de chemin actuellement en cours de construction.

On a donc une solution provisoire `optimal` ; c'est le meilleur chemin trouvé jusque là. `ch` est toujours la solution partielle en cours de construction.

la procédure de recherche devient alors :

```
procedure meilleur (x : sommet; alpha : integer) is
-----------
-- entree : comme pour explore, avec en plus
--       alpha : le cout du chemin en construction et alpha < optimal.cout
-- resultat : la variable globale optimal contiendra le meilleur chemin
--          aboutissant a arrivee et qui ameliore la solution "optimal"
--          existant avant l'appel.

y : sommet;

si x = arrivée alors
    optimal := ch ;  -- on a améliore la solution existante
  sinon
    pour y parcourant les successeurs de x repeter
        si non (ch.dans_chemin(y)) et cout(x,y)+ alpha < optimal.cout
        alors
            mettre_dans_ch (y) ;
            meilleur (y, alpha+cout(x,y));
            retirer_de_ch (y);
        fsi
    fin pour
fsi
fin meilleur
```

Ensuite si l'on désire chercher le plus court chemin d'un point de départ à l'arrivée, l'initialisation des variables globales `ch` et `optimal` ainsi que l'appel initial de `meilleur` ne présentent aucune difficulté.

Cette procédure fait cependant des calculs manifestement inutiles : si un sommet est atteint par un chemin plus long qu'un autre chemin antérieur qui l'avait déjà atteint, il est inutile de prolonger cette solution partielle. Mémoriser cette longueur et faire le test qui évite ces recherches inutiles s'implante facilement et réduit notablement la complexité ; c'est l'objet de l'exercice suivant.



**Exercice 7.10** *Ecrire l'algorithme de recherche en profondeur du chemin optimal jusqu'à* `arrivée` *et qui réalise l'optimisation indiquée ci-dessus.*

### 7.3.5 Exploration des chemins optimaux en largeur d'abord

Le fait que l'on ne recherche qu'un seul chemin fait tomber la difficulté de la gestion d'un ensemble de chemin pour la recherche en largeur dès lors que l'on cherche les plus courts chemins depuis une source donnée $a$. D'après le principe d'optimalité, tout chemin optimal de $a$ à $x$ passant par $y$ a comme chemin préfixe un chemin optimal de $a$ à $y$.

Pour mémoriser les chemins, il suffit donc de retenir pour chaque $y$ son antécédent dans le chemin optimal ; c'est le principe du lien inverse. Le parcours de ces liens permettra ensuite de reconstruire à l'envers les chemins optimaux au départ de $a$. La figure 7.2 illustre ce chaînage.

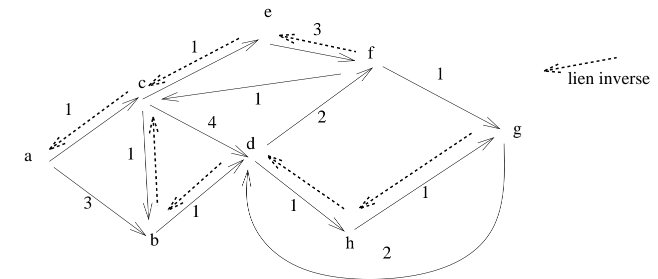


Figure 7.2: Liens inverses pour la recherche du plus court chemin depuis $a$.

En adaptant le schéma de l'exploration en largeur d'arbres (cf 7.1.2) on obtient :

```
distance : array(sommet) of integer := (others => last'integer);
lien_inverse : array(sommet) of sommet ;

- initialisation
lien_inverse(depart) := depart;
distance(depart) := 0;
file := file_vide;
insere_en_queue(file, depart);
tantque file ≠ file_vide répéter
       - invariant : lien_inverse indique le lien du meilleur chemin
       -           trouve jusqu'a chaque sommet et
       -           distance la longueur du ce chemin
       -           file contient les sommets dont les
       -           successeurs sont a examiner
       oter_en_tête(file, x);
       pour y dans successeur de x répéter
                   si distance(x)+ cout(x,y) < distance(y) alors
                               distance(y) := distance(x) + cout(x,y);
                               lien_inverse(y) := x;
                               insere_en_queue(file, y);
                   fsi
       fin pour
```

```
ftq
```

Par rapport à tous les chemins possibles qui sont explorés par la recherche en largeur simple, on note qu'on ne prolonge pas les chemins qui ne sont manifestement pas optimaux `distance(x) + cout(x,y) > distance(y)`. Donc tous les chemins optimaux sont explorés et donc l'algorithme fournit bien les chemins de distance minimale.

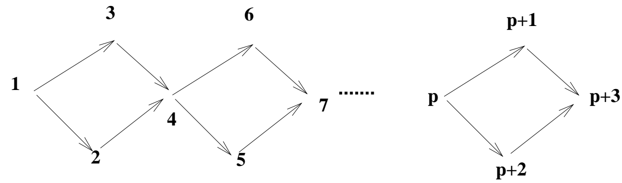


Figure 7.3: Graphe dont tous les arcs ont un coût unitaire.

**Exercice 7.11** *Comment se comporte l'algorithme de recherche en largeur sur le graphe de la figure 7.3 ?*

**Exercice 7.12** *Construire un graphe pour lequel cet algorithme à un comportement exponentiel en le nombre de sommets. Comment éviter ce type de comportement ?*

# Chapitre 8

# Fermeture transitive d'un graphe

Rappelons la définition de la fermeture transitive ; c'est la relation $\mathcal{R}^*$ qui peut être définie à partir de $\mathcal{R}$ par le plus petit point fixe de l'équation

$$\mathcal{R}^* = \mathcal{I} \bigcup \mathcal{R} \circ \mathcal{R}^* \tag{8.1}$$

et dont la solution est (cf 6.2) :

$$\mathcal{R}^* = \mathcal{I} \bigcup (\cup_{i=1}^{n-1} \mathcal{R}^i) \tag{8.2}$$

## 8.1 Algorithme de produits sucessifs

L'équation (8.2) suggère de construire les produits successifs de $\mathcal{R}$ et d'en faire l'union. Si nous retenons la représentation matricielle et un schéma de calcul dérivé du schéma de Horner :

$$\mathcal{R}^* = \mathcal{R} \circ (\mathcal{R} \circ \ldots \circ (\mathcal{R} \cup \mathcal{I}) \ldots \cup \mathcal{I}) \cup \mathcal{I}$$

la complexité sera $O(n^4)$ en prenant une complexité en $O(n^3)$ pour le produit de matrices booléennes.

On peut cependant noter que

$$(\mathcal{I} \cup \mathcal{R})^2 = \mathcal{I} \cup \mathcal{R} \cup \mathcal{R}^2$$

et plus généralement

$$(\mathcal{I} \cup \mathcal{R})^k = \mathcal{I} \cup \mathcal{R} \cup \ldots \cup \mathcal{R}^k$$

La puissance n-*ième* peut être obtenu en $k = \lceil \log n - 1 \rceil$ élévations à la puissance successives (on a bien le droit de dépasser $n$) et donc ce schéma de calcul a une complexité de $O(n^3 \log n)$.

## 8.2 Construction par exploration arborescente

La définition implicite de la fermeture transitive par l'équation à point fixe (8.1) conduit à la définition récursive suivante des points accessibles depuis un sommet $x$ :

$$\texttt{accessible}(x) = \{x\} \bigcup ( \bigcup_{y \in \texttt{succ}(x)} \texttt{accessible}(y))$$





Ceci correspond à une exploration récursive arborescente comme nous l'avons déjà vu. Sa programmation directe conduit en revanche à des bouclages infinis dès qu'il y a un circuit dans le graphe. Pour éviter cela, il suffit donc de mémoriser les points déjà rencontrés, mais cela conduit à ne rechercher que les points accessibles depuis un seul point. Ce qui nous donne l'algorithme suivant, où pour la simplicité de l'écriture, on a préféré représenter $\mathcal{R}^*$ par la matrice de booléens `fermeture`:

```
atteint : array(sommet) of boolean;
fermeture : array(sommet,sommet) of boolean;

procedure Explore(y : sommet);
--- en entree : tous les sommets deja visites sont consignes dans atteint
--- en sortie : tous les sommets atteignables depuis y et non
---             encore  visites sont  ajoutes a atteint
pour z dans successeur (y) repeter
    si non atteint(z)  alors
        atteint(z) := true;
        Explore(z)
    fsi
fin pour
fin Explore

----- programme de construction de la fermeture transitive
pour x dans sommet repeter
        atteint := (others => false);
        atteint(x) := true;
        Explore(x);           -- calcul des sommets accessibles depuis x
        fermeture(x,1..N) := atteint(1..N);
fin pour
```

On notera que cet algorithme se prête facilement à la recherche des sommets atteignables depuis un point source seulement, à la différence de l'algorithme précédent.

### 8.2.1 Complexité de la fermeture transitive par exploration

L'appel de `explore(x)` ne peut conduire à visiter chacun des sommets qu'une seule fois au plus. Pour chaque sommet visité, on examine tous ses successeurs `z`. Donc pour l'ensemble, on aura au plus $m$ examens, et donc cet appel a un nombre d'étapes en $O(m)$.

`explore(x)` est appelée pour chaque sommet x, ce qui nous fait une complexité totale en $O(mn)$, ce qui se compare favorablement avec l'algorithme précédent.

**Exercice 8.1** *Fournir un graphe où le nombre d'étapes de l'algorithme est $O(n^3)$.*

*Fournir deux graphes connexes avec $O(n)$ arcs pour lesquels la complexité de la fermeture transitive soit respectivement en $O(n)$ et $O(n^2)$, hors l'initialisation de* `atteint` *et sa recopie dans* `fermeture`.

### 8.2.2 Etude de la complexité en moyenne

L'exercice 8.1 montre que cet algorithme peut avoir des comportements très différents. Pour étudier sa complexité en moyenne, nous supposerons que les successeurs d'un sommet `y`

sont distribués aléatoirement et de façon uniforme. L'algorithme se terminera au plus tard quand les $n$ sommets seront atteints. Par le rajout d'un compteur `cpt` qui totalise les points atteint nous modifions légèrement `Explore` pour tester ce cas :

```
procedure Explore2(y : sommet);
--- en entree : tous les sommets deja visites sont consignes dans atteint
--              et cpt compte les sommets deja atteints.
--- en sortie : tous les sommets atteignables depuis y et non encore
--              visites sont ajoutes a atteint
pour z dans successeur de x repeter
    si non atteint(z) alors
        atteint(z):= true;
        cpt:= cpt +1;
        si cpt=N alors raise fin_explore fsi
        explore(z)
    fsi
fin pour
fin Explore2

-- la nouvelle version de l'appel de Explore2 r\'ecup\`ere fin_explore \`a la Ada
pour x dans sommet r\'ep\'eter
        atteint := (others => false);
        atteint(x) := true;
        begin
           Explore2(x);            -- calcul des sommets accessibles depuis x
        when fin_explore => null; --   il n'y a rien \`a faire :
        end ;                      -- atteint contient les points accessibles
        fermeture(x,1..N) := atteint(1..N);
fin pour;
```

On voit que la procédure Explore2 est, dans le pire des cas, amenée à compléter totalement la liste des points atteints. Si on suppose que les successeurs arrivent dans un ordre aléatoire uniforme, on peut appliquer le résultat établi pour l'algorithme de la collection complète (IV- 1.5.2). L'algorithme se termine donc en un nombre moyen d'étapes au pire égal à $O(n \log n)$ étapes. De fait, dans la réalité, les successeurs ne sont pas tirés aléatoirement, car pour un même sommet `y`, ils sont tous différents. Donc l'algorithme réel aura un comportement en moyenne plus performant que celui de la collection complète. Au total pour tous les points, cet algorithme sera donc majoré en moyenne par $O(n^2 \log n)$.

**Exercice 8.2** *Construire un graphe avec $m = O(n^2)$ arcs et pour lequel le test d'arret* `cpt=n` *n'apporte rien quant aux performances.*
*Quel est l'intérêt de mentionner la condition $m = O(n^2)$ dans la question précédente ?*

**Exercice 8.3** *Pour éviter d'implanter une procédure récursive avec exception (l'échappement* `fin_explore`*) on préfère implanter l'exploration en profondeur de façon itérative selon le schéma de la recherche en largeur (cf exercice 7.2). Ecrire la partie itérative correspondant à l'appel de* `explore(x)` *avec le test d'arret* `cpt = n`*.*



## 8.3 Algorithme de Warshall

Les algorithmes précédents reposaient respectivement sur les définitions de l'ensemble des chemins suivantes :

- les chemins sont définis comme réunion de chemin de longueur $k$ (algorithme par produit).
- les chemins sont définis par les constructeurs de l'arbre d'exploration sous-jacent (algorithme d'exploration).

L'idée de l'algorithme de Warshall est de définir les chemins en faisant une récurrence sur la nature des sommets intermédiaires rencontrés. On définit ainsi $W_{i,j,k}$ comme l'ensemble des chemins de $i$ à $j$ en passant par des sommets intermédiaires de numéro inférieur ou égaux à $k$. La définition par récurrence s'écrit facilement :

$$W_{i,j,k} = W_{i,j,k-1} \bigcup W_{i,k,k-1} \oplus (W_{k,k,k-1})^* \oplus W_{k,j,k-1} \quad (8.3)$$

$$W_{i,j,0} = \begin{cases} \{(i,j)\} & \text{si } \mathcal{R}(i,j) \\ \{(i)\} & \text{si } i = j \text{ (cas du chemin de 1 sommet)} \\ \emptyset & \text{sinon} \end{cases} \quad (8.4)$$

L'opérateur $\oplus$ réalise la concaténation d'ensemble de chemins. Cette définition se traduit aisément en langue naturelle : les chemins pour aller de $i$ à $j$ en passant des sommets de numéro inférieurs ou égaux à $k$ sont composés de ceux passant par les sommets de numéro strictement inférieurs à $k$, et ceux passant par $k$. Pour passer par $k$, on va d'abord de $i$ à $k$, puis on boucle un certain nombre de fois sur $k$ avec $(W_{k,k,k-1})^*$, pour enfin se rendre en $j$.

Il est clair que les boucles exprimées par $(W_{k,k,k-1})^*$ peuvent être supprimées si on ne s'y intéresse pas. Noter que cette simplification ne garantit pas que les chemins construits sont élémentaires.

De cette définition de chemin découle directement l'algorithme de Warshall qui remplace la construction de l'ensemble des chemins simplement par le calcul de l'existence de chemins :

```
-- intialisation de W(i,j,0)
pour (i,j) dans  1..N x 1..N repeter
    W(i,j) := R(i,j);
fin pour

pour k dans 1..N repeter
-- construction de W(i,j,k)
    pour (i,j) dans  1..N x 1..N repeter
        W(i,j) := W(i,j) ou ( W(i,k) et W(k,j) );
    fin pour;
fin pour
```

De façon évidente la complexité de cet algorithme est $O(n^3)$. Cet algorithme est régulier et donc, comme pour les algorithmes de produits successifs, se prètera donc à une bonne optimisation par les compilateurs. Cet avantage, couplé à sa complexité, le fera préférer au premier dans le cas de graphes presque complets.

**Exercice 8.4** *Dans l'algorithme de Warshall on ne respecte pas strictement la définition récursive, car la variable* `W` *qui mémorise les différents* $\mathtt{W_k}$ *est écrasée en cours de boucle et, donc parfois le calcul réalise*

$$W(i,j,k) = W(i,j,k-1) \ \text{ ou } \ (W(i,k,k) \text{ et } W(k,j,k))$$

*au lieu de*

$$W(i,j,k) = W(i,j,k-1) \ \text{ ou } \ (W(i,k,k-1) \text{ et } W(k,j,k-1))$$

*Justifier que ceci ne change pas le résultat.*

## 8.4 Extension : calcul de tous les plus courts chemins par Floyd-Warshall

Si le graphe est pondéré par des distances, la définition des chemins vue en 8.3 peut être adaptée pour calculer les plus courts chemins entre tout couple de points :

$$\begin{aligned} F_{i,j,0} &= 0 \\ F_{i,j,k} &= \min(F_{i,j,k-1}, F_{i,k,k-1} + F_{k,j,k-1}) \end{aligned}$$

L'implantation est directe et laissée à titre d'exercice. Sa complexité est, comme pour la fermeture transitive, en $O(n^3)$.

**Exercice 8.5** *Implanter l'algorithme de Floyd-Warshall. Comment se transpose l'optimisation proposée pour Warshall à ce cas (cf 8.3) ?*
*Modifier l'algorithme pour retenir en même temps les chemins optimaux découverts.*

## 8.5 Bilan pour la fermeture transitive

La fermeture transitive par exploration est asymptotiquement le plus performant des trois algorithmes présentés. De plus il travaille sur une représentation du graphe par ensemble de successeurs. C'est donc la solution pour les gros graphes dans la mesure où leurs arcs ne sont pas trop dense.

En revanche, si les graphes sont de tailles raisonnables, ou presque complet, l'algorithme de Warshall devient intéressant. D'abord il est très simple. Ensuite la régularité de ses calculs autorise une utilisation optimale des ressources des machines (mémoire cache) tout comme l'optimisation du code par les compilateurs.

Dans tous les cas, si le temps reste un problème prépondérant, il faudra veiller à un codage fin de l'algorithme, voire même procéder à des tests pour vérifier quel codage est optimal.

A titre d'exemple d'optimisation de code, voici une variante du codage de la boucle de Warshall :

```
pour k dans 1..N repeter
    pour i dans 1..N repeter
        si  W(i,k) alors -- W(i,1..N):= W(i,1..N) or W(k,1..N)
            pour j dans 1..N repeter
```



```
                W(i,j) := W(i,j) ou W(k,j)
            fin pour
        fsi
    fin pour
fin pour
```

Avec un tel codage, la vectorisation de la boucle la plus interne est à la portée de n'importe quel optimiseur ; et sans vectorisation, la gestion de la boucle interne est facile à optimiser pour un compilateur. Ainsi il faudra atteindre des valeurs de l'ordre de 1000 sommets (cela dépend de la machine, du compilateur, du nombre d'arcs, ...) pour que Warshall soit battu par l'algorithme d'exploration.

# Chapitre 9

# Recherche du plus court chemin

De nombreux problèmes se ramènent au calcul du plus court chemin dans un graphe. Dans le chapitre précédent, l'algorithme de Floyd-Warshall permet de calculer tous les plus courts chemins élémentaires dans un graphe. En pratique, on rencontre aussi les sous-problèmes suivants :

- calcul du plus court chemin élémentaire entre un sommet source et un sommet destination;
- calcul de tous les plus courts chemins issus d'un sommet source (i.e. vers tous les autres sommets); ou encore, de manière duale en retournant les arètes du graphe, calcul de tous les plus courts chemins aboutissant à un même sommet destination.

On ne connaît pas d'algorithmes asymptotiquement plus efficaces pour résoudre le premier problème (plus court chemin entre deux sommets) que pour le deuxiéme. Aussi, les algorithmes de calcul de tous les plus courts chemins issus d'un même sommet sont très souvent rencontrés :

- par exploration arborescente pour un graphe orienté sans cycle.
- l'algorithme de Bellman pour le cas d'un graphe orienté sans circuit;
- l'algorithme de Dijkstra pour un graphe dont les arètes sont à poids positif;

## 9.1 Recherche par exploration arborescente

Pour cette façon de rechercher, nous nous limitons aux chemins dont l'origine est fixée : c'est le sommet `départ`.

Lors de la recherche du meilleur chemin par exploration (largeur ou profondeur d'abord), nous avons constaté que cette recherche systématique conduisait à revisiter plusieurs fois des sommets et pouvait conduire à des solutions exponentielles en temps.

Une heuristique directe pour l'amélioration de la recherche en profondeur est de parcourir les successeurs dans l'ordre croissant des coûts. Il va sans dire que ce tri ne doit être fait qu'une seule fois. Expérimentalement, cette heuristique réduit le temps d'exécution d'un facteur très dépendant de la nature du graphe ; typiquement une division par 2 est observée pour des graphes de faible degré, mais ce facteur est nettement plus fort pour des





graphes de fort degrés et dont la variance des coûts est élevée. Comme toute heuristique, elle peut bien sûr être mise en défaut.

**Exercice 9.1** *Construire un graphe pour lequel l'heuristique précédente ralentit l'exécution.*

**Exercice 9.2** *Que faut-il modifier si le problème n'est pas de chercher les plus courts chemins au départ de $x$, mais les plus courts chemins aboutissant à $x$ ? Et si le problème est la recherche du plus court chemin de $x$ à $y$ ?*

## 9.2 Graphe orienté sans circuit: algorithme de Bellman

Comme le graphe est sans circuit, il n'existe pas de circuit absorbant même lorsque les poids sont négatifs. De plus un chemin optimal suit la relation pére-fils de l'arbre; tous les sommets sur le chemin apparaissent donc selon l'ordre topologique.
Il suffit donc d'un passage sur les sommets dans l'ordre topologique pour calculer les plus courts chemins issus d'un sommet.
C'est l'idée de l'algorithme de Bellman présenté ici.

La définition `dist`, longueur du plus court chemin de `depart` à `x` peut être exprimée ainsi :

$$\texttt{dist}(x) = \begin{cases} 0 \quad si \quad x = \texttt{depart} \\ \min_{y \in \text{pred}(x)}(\texttt{dist}(y) + \texttt{cout}(y,x)) \end{cases} \tag{9.1}$$

Une telle définition conduit bien sûr directement à un programme. Mais ce dernier bouclera si le graphe comporte des circuits ; si le graphe ne comporte pas de circuits en revanche, l'appel de la fonction `dist(a)` se termine et la réponse est correcte, conséquence de la correction de la définition.

Il faut de plus éviter des appels redondants à `dist` pour un même paramètre. Pour ce faire, on mémorise simplement les distances déjà calculées dans un tableau `distance` :

```
fonction dist( x: sommet) return longueur is
   si est_calculé(x) alors return distance(x) ;
   sinon si x = depart alors
      distance(x) := 0;
      est_calculé(x) := true;
      return(0)
   sinon
      res:= integer'last;
      pour y dans  predecesseurs de x repeter
          d := dist(y) + cout (y,x);
          res := min (res, d));
      fin pour
      distance(x) := res;
      est_calculé(x) := true;
      return(res)
   fsi
fin dist

-- appel initial
est_calculé(1..N):= (others => false);
d:=dist(a) ;
```

On notera que cet algorithme ne visite au plus chaque arc qu'une seule fois, et donc sa complexité en temps est $O(m)$.

Dans le cas de circuits, un tel algorithme conduirait à un nombre infini d'appels récursifs. Un tel effet est à éviter. Pour ce faire, constatons d'abord que l'algorithme ne considère qu'un seul chemin à la fois, car il est du type "profondeur d'abord". Il faut donc marquer les sommets déjà dans le chemin en cours d'exploration comme cela a déjà été fait auparavant. Sans reprendre les initialisations, on obtient donc la procédure suivante :

```
fonction dist( x: sommet) return longueur is
   si est_calcule(x) alors return distance(x) ;
   sinon si x = depart alors distance(x):= 0;
      est_calcule(x) := true;
      return(0) ;
   sinon
      res := integer'last;
      pour y dans predecesseurs de x repeter
         si dans_chemin(y) alors null;
         sinon
            mettre_dans_chemin(y)
            d := dist(y) + cout (y,x);
            res := min (res, d));
            retirer_de_chemin(y) ;
         fsi ;
      fin pour
   distance(x):= res;
   est_calcule(x):=true;
   return(res)
   fsi
fin dist
```

**Exercice 9.3** `dist(a)` *ne calcule que la distance de* `depart` *à* `a`. *Comment modifier la prodédure pour aussi obtenir le chemin correspondant ?*

**Exercice 9.4** *Pour le graphe de la figure 9.1, simuler l'exécution de l'appel intial de* `dist(a)` *suivi par l'appel de* `dist(b)`. *Quel est le résultat de l'appel de* `dist(c)` *?*

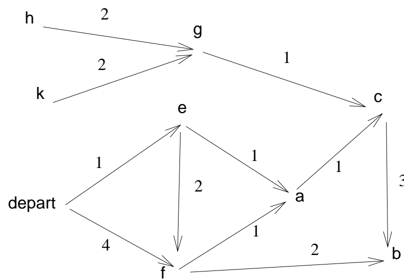


Figure 9.1: Exemple de graphe.



Cet algorithme est connu sur le nom d'algorithme de Bellman. De fait l'algorithme de Bellman est cet algorithme dans sa version itérative. Pour ce faire, on suppose que non seulement le graphe est sans circuit, mais que de plus pour tout sommet `x`, il existe un chemin depuis `départ` à `x`. On explore alors itérativement depuis le départ :

```
distance(1..N) := (others => integer'last);
distance(depart) := 0;
Attente := { depart };
marquer(depart);   -- des qu'un noeud a sa bonne distance, il est marque
                   --   et mis dans Attente

tant que Attente /= ensemble_vide repeter
    -- invariant: Attente contient les points dont la distance est
    --          correcte mais dont les successeurs  sont a examiner.
    --          Tous les points passes par Attente ont ete marques.
    choix de x dans Attente;
    retirer x de Attente;
    pour y dans successeurs de x repeter
        distance(y):= min( distance(y), distance(x)+cout(x,y) )
        si tous les predecesseurs  de y ont ete marques
            alors marque(y); Attente := Union(Attente,{y})
    fin pour
fin tq
```

Cet algorithme s'implante sans difficulté en utilisant les mêmes structure de données que l'algorithme du tri topologique. Là encore toutes les opérations s'implantent en $O(1)$ et donc l'algorithme s'exécute en $O(m)$ étapes.

**Exercice 9.5** *Implanter l'algorithme itératif de façon que les manipulations de chaque étape de l'algorithme se fassent en $O(1)$.*

**Exercice 9.6** *Construire un graphe pour lequel le calcul de la distance de* `départ` *à* `a` *se fasse nettement plus rapidement dans le cas de la version récursive que itérative. Peut on aussi avoir le cas contraire ?*

**Exercice 9.7** *En vous inspirant de la recherche des chemins optimaux en profondeur, donner un algorithme de coût $\Theta(n+m)$ qui calcule tous les plus courts chemins issus d'un sommet s dans un DAG, avec comme pré-calcul un tri-topologique du graphe.*
*Chaque sommet* `u` *a deux attributs supplémentaires pour mémoriser le chemin optimal :*

- `u.lien_inverse` *(de type Sommet) pour calculer le prédécesseur de* `u` *sur un chemin optimal issu de* `s`*;*
- `u.distance` *(de type entier) pour calculer la distance optimale de* `s` *à* `u`.

### Correction

```
Pour tout sommet u de G {
  u.lien_inverse = null ;
  u.distance = INFINI ;
}
```

```
tritopo = TriTopologique ( G ) ;

while ( ( ! tritopo.empty() ) && ( tritopo.top() != s) )
  tritopo.pop() ;

s.distance = 0 ;
while (! tritopo.empty() ) {
  u = tritopo.pop() ;
  Pour tout sommet v fils de u {
    if (v.distance > u.distance + poids(u,v) )
      v.distance = u.distance + poids(u, v);
      v.lien_inverse = u ;
  }
}
```

Les arêtes étant parcourues au plus une fois, le coût est $\Theta(n+m)$.

## 9.3 Graphe orienté, coûts positifs : algorithme de Dijkstra

Si l'on considère l'exploration en largeur d'abord (cf 7.3.5), les sommets atteints doivent être réexaminés dès qu'ils sont à nouveau atteints par un chemin de coût plus faible. Si les **coûts** sont **positifs**, les chemins se prolongent avec des coûts croissants. En conséquence le sommet $x$ atteint avec le coût minimum ne sera plus remis en cause. On peut donc prolonger le chemin optimal d'extrémité $x$ sans crainte de révision.

Cette idée est à la base de l'algorithme de Dijkstra. Au lieu de simplement gérer les sommets en largeur, on choisit donc le sommet non encore prolongé de coût optimal pour lequel on traite les successeurs. Les sommets en attente sont donc traités selon une stratégie du "meilleur d'abord". Cela donne l'algorithme suivant :

```
-- initialisation
Attente := {depart};
lien_inverse(depart) :=  lien_vide;
      -- c'est pour arreter le chainage inverse des chemins
distance(depart) := 0;

tant que Attente non vide repeter
    oter-minimum (Attente, x);
    pour y dans successeur de x repeter
        si distance(x) + cout(x,y) < distance(y) alors
            distance(y) := distance(x) + cout(x,y);
            lien_inverse(y) := x;
            insérer (Attente, y);
        fsi
    fin pour
ftq
```

On notera que pour chaque sommet, l'ensemble des successeurs est examiné une fois et une seule. La complexité de l'algorithme sera donc $O(m \times \alpha + n \times \beta)$ où $\alpha$ et $\beta$ sont respectivement le coût de l'insertion dans `Attente` et celui de l'extraction du minimum. L'implantation à l'aide d'une file de priorité implantée par un tas permet de majorer $\alpha$ et $\beta$ par $\log n$.



## 9.4 Bilan de la recherche du plus court chemin

Tous les algorithmes que nous avons vus ne sont pas équivalents dans leurs hypothèses d'application. Pour tous nous avons supposés que les coûts sont positifs. Le cas de coûts négatifs peut cependant être pris en compte par l'algorithme de Floyd-Warshall à la condition qu'il n'y ait pas de cirduit absorbant ; on pourra consulter [Sak84] au chapitre 2.3 pour la détection de tels circuits.

Rappelons ici les caractéristiques de chaque algorithme :

- recherche en profondeur : le meilleur sur un graphe orienté sans circuit. Dans le cas général, exponentiel mais nettement amélioré par le tri des arcs si les coûts sont différents.
- Bellman : le plus rapide de tous sous cette hypothèse. Espace mémoire minimal.
- Dijkstra : comme la recherche en profondeur, cet algorithme limite sa recherche aux sommets issus du sommet, ce qui peut être un avantage pour certains problèmes ; si le nombre d'arcs est grand, il faut réfléchir à la gestion des sommets en attente.
- Floyd–Warshall : cubique en le nombre de sommet, mais performant dans ses boucles d'itérations ce qui fait que le facteur de proportionnalité est faible. Nécessite un espace de travail en $O(n^3)$ (matrice) pour mémoriser les longueurs et les chemins. Considère tous les sommets comme origines possibles à la différence des autres algorithmes.

Une étude expérimentale de la complexité sur des graphes aléatoire (TP de Flébus et Visconti, 1993) montre que :
– Bellman est presque toujours vainqueur quand il est applicable ;
– la recherche en profondeur n'est intéressante que sur des graphes dont le demi degré extérieur est très faible (entre 1 et 3 le gain est net sur les autres) ;
– Floyd et Dijskstra sont comparables et battent nettement la recherche en profondeur dès que le degré du graphe s'élève, Dijskstra barrant Floyd pour des degrés moyens, étant battu de justesse pour des graphes complets.

Grossièrement, la situation pourrait se résumer en :

- dans le cas de graphe sans circuit : Bellman
- sinon
  - graphe à très faible degré : recherche en profondeur en triant les arcs (stratégie du meilleur d'abord)
  - graphe à degré faible ou moyen : Dijkstra
  - graphe presque complet : Floyd–Warshall.

Cette réflexion n'a été menée que pour des graphes finis et de taille raisonnable. Dans le cas de jeux où le graphe des états peu être considéré comme infini, les algorithmes matriciels ne peuvent bien sûr pas s'appliquer, et même Bellman ne peut pas être mis en oeuvre dans le cas général car on peut pas connaître l'ensemble des prédécesseurs accessibles. La recherche en profondeur appliquée directement conduit à des explorations infinies: il faut en borner la profondeur, ou faire appel à un algorithme de recherche en largeur.

# Chapitre 10

# Applications à des problèmes de recherche opérationnelle

Nous présentons ici quelques problèmes de rechreche opérationnelle se ramenant au calcul d'un plus court chemin dans un graphe, ou à un problème lié.

## 10.1 Approximation linéaire par morceau d'une courbe

De nombreuses applications recueillent des données comme une suite de points dans un espace, par exemple la trajectoire d'une souris d'ordinateur, ou la suite des coordonnées d'un bus dans une ville transmise par GPS. Il s'agit d'approcher la succession de coordonnées par une courbe polygonale : cela simplifie le stockage, et cela simplifie les traitements ultérieurs, mais bien sûr au prix d'une perte de précision. On aimerait bien trouver un compromis entre perte de précision et réduction des données, deux facteurs antinomiques.

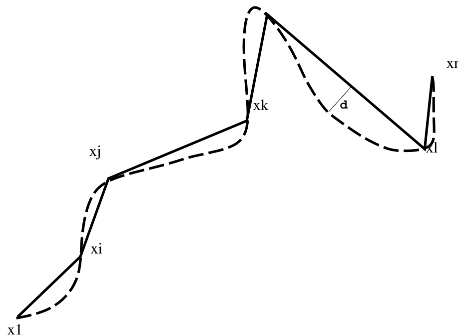


Figure 10.1: Exemple d'approximation polygonale

Soient $P_i = (x_i, y_i), i = 1, ..., n$ les $n$ points successifs que l'on considère. Si l'on retient comme points successifs $P_i$ et $P_j$, le coût de l'approximation pour ce segment est

$$\alpha + \beta \sum_{k=i+1}^{j-1} d(P_k, [P_i, P_j])$$





où $\alpha$ est le coût unitaire de stockage, $\beta$ un facteur de pondération pour l'approximation, et $d(P_k, [P_i, P_j])$ la distance de $P_k$ au segment $[P_i, P_j]$). Le coût de l'approximation totale est la somme des coûts des approximations de chacun des segments.

**Exercice 10.1** *Montrer que trouver l'approximation qui est minimale correspond à trouver un meilleur chemin dans un graphe que l'on spécifiera.*

## 10.2 Répartition des moyens de contrôle sur une chaîne

Une chaîne de production est une liste ordonnée de $n$ étapes de production. Chaque étape consiste en une phase de fabrication éventuellement suivie d'une séance d'inspection. Au cours de la fabrication, les différentes unités peuvent induire des défauts, et bien sûr ces défauts sont irrémédiables et les objets devront être jetés. Une inspection terminale sera toujours nécessaire pour ne pas livrer des objets défectueux.

Le problème se pose d'optimiser la position des postes de contrôle sachant : - $\alpha_i$ la probabilité que la phase $i$ induise un défaut sur une pièce,
- que le coût unitaire de la phase $i$ est $p_i$,
- que le coût d'inspection au poste $j$ dépend du poste $i$ de la dernière inspection, car il faudra inspecter plus de choses, interrompre une chaîne plus longue avec probablement une erreur en série entre $i$ et $j$ due à une panne d'outillage. On note ce coût $I(i, j)$.

Pour ce faire on considérera le graphe sur $n + 1$ sommets numérotés de 0 à $n$ avec un arc entre $i$ et $j$ dés lors que $i < j$,

**Exercice 10.2** *1) Comment interpréter un chemin de 0 à n dans ce graphe de façon qu'à toute décision de placement d'équipe de contrôle correspond un tel chemin.*

*2) Quel coût mettre sur les arcs de façon à ce que la longueur d'un tel chemin corresponde au coût de fabrication d'une unité entrée dans la chaîne, contrôles et risques de perte inclus ? En déduire que le chemin le plus court correspond à la stratégie optimale.*

*3) Si on avait dû prendre en compte le coût de contruction (coût unique) des postes de contrôles, quelles autres données aurait-il fallu prendre en compte ?*

## 10.3 Le problème du caboteur

Les côtes de Norvège abritent dans leurs fjords de nombreux ports naturels et les caboteurs offrent des transports réguliers entre les différentes villes de la côte. Pour le patron d'un de ces navires, le problèmes est de déterminer une tournée régulière (un circuit). On estime que le voyage de $i$ en $j$ coûte $t_{ij}$ en temps, et rapporte $g_{ij}$ en gain. La tournée la plus profitable est celle qui rapporte le plus en moyenne, c'est à dire que c'est le circuit $\mathcal{C}$ qui maximise

$$\frac{\sum_{(i,j)\in\mathcal{C}} g_{ij}}{\sum_{(i,j)\in\mathcal{C}} t_{ij}}$$

On a bien sûr $t_{ij} > 0$ pour tout arc $(i, j)$.

**Exercice 10.3** *1) Pourquoi le problème directe de recherche de circuit optimal n'entre-t-il pas dans la catégorie des problèmes de recherche de circuit ?*

*2) L'armateur veut savoir simplement s'il existe un circuit de gain moyen supérieur à une limite $\mu$ qu'il s'est fixé. Ceci revient à rechercher un circuit $\mathcal{C}$ tel que*

$$\sum_{(i,j)\in\mathcal{C}} (\mu t_{ij} - g_{ij}) < 0$$

*c.à.d. un circuit de longueur négative. Montrez que l'algorithme de Floyd-Warshall peut permettre de trouver un tel circuit.*

## 10.4 Système de contraintes formées de différences

Dans certaines applications de la programmation linéaire, les contraintes sont de la forme $Ax \leq b$ où la matrice $A$, de taille $m \times n$, contient exactement deux éléments non nuls par ligne, l'un qui vaut -1 et l'autre +1. Donc toute inégalité est de la forme

$$x_{j_k} - x_{i_k} \leq b_k$$

(Le cours d'optimisation combinatoire II donnera des exemples pratiques de contraintes de ce type)

On souhaite savoir si un tel système a une solution réalisable, et dans la négative, on souhaite identifier les contraintes incompatibles. On associe un tel système un graphe $\mathcal{G}$ appelé graphe des contraintes. Ce graphe a $n$ sommets correspondants aux $n$ variables, et $m$ arcs correspondants aux $m$ contraintes. A la contrainte $x_{j_k} - x_{i_k} \leq b_k$ on associe l'arc $(i_k, j_k)$ de "longueur" $b_k$.

**Exercice 10.4** *1) Construire le graphe des contraintes associé au système suivant:*

$$\begin{aligned} x_3 - x_4 &\leq 5 \\ x_4 - x_1 &\leq -10 \\ x_1 - x_3 &\leq 8 \\ x_2 - x_1 &\leq -11 \\ x_3 - x_2 &\leq 2 \end{aligned}$$

*Trouver par tâtonnement un circuit de longueur totale négative. Montrer, à partir de ce circuit, que ce système n'admet pas de solutions.*

*2) Montrer que l'existence d'un circuit de longueur négative dans $\mathcal{C}$ implique l'impossibilité de construire une solution.*

*3) Pour montrer que cette condition est suffisante, ajoutons un sommet $s$ à $\mathcal{G}$ ainsi que les arcs $(s, x_i)$ de coût 0. Considérons les plus courts chemins de $s$ à chacun des $x_i$. S'il n'existe pas de circuits de coût négatif (circuit absorbant), ces plus courts chemins existent. Soit $\pi(x_i)$ les longueurs minimales ainsi construites.*

*Montrer que $\pi(x_i) - \pi(x_j) \leq cot(x_i, x_j)$ pour tout arc $(x_i, x_j)$ de $\mathcal{G}$. En déduire que le système initial possède bien une solution.*

### 10.4.1 Arborescence des plus courts chemins

Soit un sommet $s$ (source) dans un graphe et nous considérons le problème de trouver tous les chemins les plus courts de ce sommets à tous les autres sommets atteignables. Nous



avons vu dans ce chapitre (7.3.5) que l'on peut retenir l'ensemble des chemins par un lien inverse vers le prédécesseur. Ces liens inverses déterminent *l'arborescence des plus courts chemins*, arborescence bien utile comme l'a montré l'exercice précédent 10.4.

On note souvent $\pi$ la fonction qui à tout sommet $x$ donne la longueur du plus court chemin d'origine $s$. On a bien sûr : $\pi(s) = 0$.

**Exercice 10.5** *Montrer que l'arbre des plus courts chemins n'est pas nécessairement unique, bien que la fonction $\pi$ l'est.*

**Exercice 10.6** *1) Soit $\mathcal{P}$ un plus court chemin de $s$ à $x$. Montrez que pour tout arc $(i, j)$ de $\mathcal{P}$, alors $\pi(j) - \pi(i) = c(i, j)$*

*2) Réciproquement montrer que si un chemin $\mathcal{P}$ vérifie la propriété ci-dessus pour tous ses couples de sommets consécutifs $(i, j)$, alors $\mathcal{P}$ est un plus court chemin.*

De l'exercice précédent, on détermine un algorithme de vérification particulièrement simple (linéaire) pour déterminer si une fonction $\pi$ est une mesure de distance des plus courts chemins :
- d'une part on doit avoir $\pi(s) = 0$,
- d'autre part si on construit le graphe ayant les arcs entre $i$ et $j$, si $\pi(j) - \pi(i) = c(i, j)$, alors l'algorithme de fermeture transitive doit pouvoir atteindre tous les sommets depuis $s$.

# Programmation récursive

# Chapitre 12

# Diviser pour résoudre et applications

La stratégie "diviser pour résoudre" [1] que nous avons déjà rencontrée dans des algorithmes tels que la recherche dichotomique, le tri fusion et le tri rapide, est l'une des plus répandues pour l'élaboration d'algorithmes. Elle consiste à diviser un problème de taille $n$ en sous-problèmes plus petits, dont la résolution permet de construire la solution du problème entier. Cela conduit le plus souvent à l'écriture d'algorithmes compacts et efficaces.

Il y a deux manières d'appliquer cette stratégie, que nous illustrerons par plusieurs exemples :

- La récursivité sur les données, comme dans le cas du tri-fusion :
  1. On sépare les données en deux parties quelconques ($O(1)$),
  2. on résout les sous-problèmes,
  3. il y a un certain travail à faire pour combiner les résultats (typiquement en $O(n)$).
- La récursivité sur le résultat, comme dans le cas du tri-rapide :
  1. On effectue un prétraitement, en $O(n)$ par exemple, pour trouver un bon découpage des données,
  2. on résout les sous-problèmes,
  3. les sous-résultats se combinent d'eux mêmes, grâce à l'effort fait en 1.

## 12.1 Complexité des algorithmes "diviser pour régner"

Notations: $n$ taille du problème. $T(n)$ coût en nombre d'opérations fondamentales pour un problème de taille $n$.

### 12.1.1 Cas où la résolution d'un seul sous problème suffit

**Théorème :** Si $T(n) = T(n/2) + g(n)$, avec $T(1) = C$, alors:

$$T(n) = C + \sum_{1}^{E(\log_2(n))} g(2^i)$$

[1]on dit aussi "diviser pour régner", "divide and conquer", traductions plus ou moins fidèles du latin “divide et impera”.





**Démonstration :** Supposons que $n = 2^k$ (ou encadrer $n$ ....). On a:

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

**Exemple :** Recherche dichotomique. En deux comparaisons, on sait dans quel sous-tableau effectuer les recherches.
$T(n) = T(n/2) + 2 = \sum_1^{\log_2(n)} 2 = O(\log_2(n))$.

### 12.1.2 Cas général : division des données puis regroupement

En général, on divise les données du problème en sous ensembles, on résout chaque sous-problème, puis on collecte les résultats. C'est ce qui se passe par exemple pour les algorithmes de tri.

**Equilibrage des sous-problèmes:** Les algorithmes de type diviser pour règner sont d'autant plus efficaces que les sous ensembles construits sur les données sont de même taille. Par exemple, il est possible de considérer le tri par insertion comme la division d'un problème en deux sous problèmes de tailles 1 et $n-1$ respectivement. La collecte des résultats a pour complexité $n$ au maximum. On aboutit à un algorithme en $O(n^2)$. Le tri fusion, pour qui la collecte a également une complexité en $n$ mais qui divise les données en deux sous ensembles de taille $n/2$ donne par contre un coût en $nlogn$.

Le théorème qui suit s'applique à tous les algorithmes pour lesquel on divise les données en sous ensembles de taille égale, et où le temps nécessaire au regroupement des résultats à chaque étape est proportionnel à $n$.

**Théorème :** Si $T(n) = aT(n/b) + cn$, avec $T(1) = C$, $a > 1$, $b > 1$, alors :

$$\begin{array}{lcl} a < b & \Rightarrow & T(n) = O(n) \\ a = b & \Rightarrow & T(n) = O(nlog(n)) \\ a > b & \Rightarrow & T(n) = O(n^{log_b(a)}) \end{array}$$

**Démonstration :**

$$\begin{array}{rcl} T(n) & = & aT(\frac{n}{b}) + cn \\ aT(\frac{n}{b}) & = & a^2T(\frac{n}{b^2}) + a\frac{cn}{b} \quad \text{(ligne précédente multipliée par } a\text{, avec } n \text{ remplacé par } n/b) \\ ..... & = & ...... \\ a^iT(\frac{n}{b^i}) & = & a^{i+1}T(\frac{n}{b^{i+1}}) + a^i\frac{cn}{b^i} \\ ...... & = & ...... \\ a^{log_b(n)}T(1) & = & a^{log_b(n)}.C \end{array}$$

On peut donc éliminer les termes en $T$ à droite :

$$T(n) = cn \sum_{i=0}^{log_b(n)-1} (\frac{a}{b})^i + a^{log_b(n)}.C$$

- Si $a < b$, la série $\sum_{i=0}^{\infty}(\frac{a}{b})^i$ converge:

$$\sum_{i=0}^{log_b(n)-1}(\frac{a}{b})^i < \sum_{i=0}^{\infty}(\frac{a}{b})^i = \text{constante}$$

Le premier terme de $T(n)$ est donc en $0(n)$. De plus, $a^{log_b(n)} = n^{log_b(a)}$ donc le second terme est négligeable.

- Si $a = b$, le premier terme est en $nlog(n)$, et le second, en $O(n)$, est négligeable.

- Si $a > b$, on a:

$$\sum_{i=0}^{log_b(n)-1}(\frac{a}{b})^i = \frac{(\frac{a}{b})^{log_b(n)} - 1}{\frac{a}{b} - 1} \simeq (\frac{a}{b})^{log_b(n)} = \frac{a^{log_b(n)}}{n}$$

Les deux termes sont donc de même ordre:

$$T(n) \simeq a^{log_b(n)} = n^{log_b(a)}$$

**Exemple :** Tri fusion.
$T(n) = 2T(n/2) + n$, donc $a = b = 2$. $T(n) \simeq n\log(n)$.

Le reste de ce cours présente une série d'algorithmes qui illustrent l'intérêt des stratégies diviser pour régner.

## 12.2 Exemples classiques

### 12.2.1 La multiplication de grands entiers

Il s'agit de multiplier deux entiers X et Y qui ont $n$ chiffres en base 2. La méthode que l'on utilise habituellement dans les classes primaires consiste à fabriquer n produits partiels de taille n, et a donc une compexité de $O(n^2)$ (en nombre d'additions et de multiplications entre chiffres).

Une aproche diviser pour résoudre consiste à scinder X et Y en deux entiers de $n/2$ chiffres (pour simplifier, on suppose ici que n est une puissance de 2) :

$$\begin{aligned} X &= A2^{\frac{n}{2}} + B \\ Y &= C2^{\frac{n}{2}} + D \end{aligned}$$

Le produit de X et Y s'écrit maintenant :

$$XY = AC2^n + (AD + BC)2^{\frac{n}{2}} + BD \qquad (12.1)$$

Si on évalue XY de cette manière, on a : quatre multiplications à $\frac{n}{2}$ chiffres, trois additions avec au maximum $2n$ chiffres, et deux décalages (multiplications par $2^n$ et $2^{\frac{n}{2}}$). Ces deux derniers types d'opérations sont en $O(n)$, d'où :

$$\begin{aligned} T(1) &= 1 \\ T(n) &= 4T(\frac{n}{2}) + cn \end{aligned}$$



Nous sommes donc dans les conditions d'application du théorème de la section 12.1, avec $a = 4$ et $b = 2$. Ce qui nous donne $T(n) = O(n^{log_2(4)}) = O(n^2)$. Il n'y a donc pour l'instant pas d'amélioration de la complexité asymptotique.

Par contre, on peut reformuler l'équation 12.1 de manière à *diminuer le nombre de sous-problèmes*, c'est à dire le nombre de multiplications entre entiers à $n/2$ chiffres :

$$XY = AC2^n + [(A - B)(D - C) + AC + BD]2^{\frac{n}{2}} + BD$$

Il n'y a plus que trois sous-produits à effectuer. On a donc :

$$\begin{aligned} T(1) &= 1 \\ T(n) &= 3T(\frac{n}{2}) + c'n \end{aligned}$$

d'où $T(n) = O(n^{log_2(3)}) = O(n^{1.59})$. En pratique, cette solution n'est plus efficace que l'algorithme naïf que si $n > 25$, du fait de la valeur élévée des constantes. (En codant un nombre d'environ 10 chiffres décimaux sur un mot machine de 32 bits, cela correspond à 250 chiffres décimaux.)

### 12.2.2 La multiplication de polynomes (exercice)

Proposer un algorithme analogue pour effectuer la multiplication de deux polynômes de degrés $n - 1$ avec une complexité inférieure à $n^2$.

Solution : On sépare les polynomes $P$ et $Q$ à $n$ coefficients en polynomes gauches et droits, qui ont chacun $n/2$ coefficients (on suppose que $n$ est une puissance de 2) :

$$\begin{aligned} P(X) &= P_g(X) + X^{n/2}P_d(X) \\ Q(X) &= Q_g(X) + X^{n/2}Q_d(X) \end{aligned}$$

On reformule ensuite le produit de manière à ce qu'il n'y ait que trois multiplications de polynomes de taille $n/2$, par exemple :

$$PQ = P_gQ_g + [(P_g + P_d)(Q_g + Q_d) - P_gQ_g - P_dQ_d]X^{\frac{n}{2}} + P_dQ_dX^n$$

D'où une complexité $T(n) = 3T(n/2) + cn = O(n^{1,59})$, comme dans l'exemple précédent.

### 12.2.3 Multiplication de matrices: algorithme de Strassen

La multiplication de grosses matrices est un autre exemple de l'application de cette stratégie. Jusqu'en 1968, on pensait qu'il était impossible de réaliser cette opération en moins de $O(n^3)$ multiplications ($n$ multiplications pour chacun des $n^2$ termes). Mais Strassen a proposé un algorithme de type diviser pour règner qui fonctionne en $O(n^{2,81})$ multiplications.

Supposons pour simplifier qu'il s'agit de multiplier deux matrices $n, n$ avec $n = 2^k$. Pour cet exemple, nous allons évaluer les coûts en nombre de multiplications entre termes.

L'application de la stratégie diviser pour règner consiste à couper la matrice en quatre sous-matrices carrées :

$$P = M.N = \begin{pmatrix} M_{11} & M_{12} \\ M_{21} & M_{22} \end{pmatrix} . \begin{pmatrix} N_{11} & N_{12} \\ N_{21} & N_{22} \end{pmatrix}$$

Si on multipliait simplement les sous matrices, on aurait :

$$T(n) = 8\, T(\frac{n}{2}) = O(n^3)$$

(ce n'est pas une application directe du théorème vu plus haut, car le terme en $n$ ne figure pas, mais la résolution est analogue).

Strassen a proposé de remplacer cette multiplication directe des sous-matrices par les calculs suivants :

$$\begin{aligned} P_{11} &= X_1 + X_4 - X_5 + X_7 \\ P_{12} &= X_3 + X_5 \\ P_{21} &= X_2 + X_4 \\ P_{22} &= X_1 + X_3 - X_2 + X_6 \end{aligned}$$

avec:

$$\begin{aligned} X_1 &= (M_{11} + M_{22})(N_{11} + N_{22}) \\ X_2 &= (M_{21} + M_{22})N_{11} \\ X_3 &= M_{11}(N_{12} - N_{22}) \\ X_4 &= M_{22}(N_{21} - N_{11}) \\ X_5 &= (M_{11} + M_{12})N_{22} \\ X_6 &= (M_{21} - M_{11})(N_{11} + N_{12}) \\ X_7 &= (M_{12} - M_{22})(N_{21} + N_{22}) \end{aligned}$$

Il y a donc 7 multiplications de sous matrices à réaliser au lieu de 8, ce qui conduit à :

$$T(n) = 7\, T(\frac{n}{2}) = O(n^{log_2(7)}) = O(n^{2,81})$$

Comme pour les exemples précédents, les constantes sont très importantes du fait des nombreuses additions et soustractions supplémentaires.

$$Strassen - Winograd : T(n) = 7\, T(\frac{n}{2}) + 15 Add(\frac{n}{2})$$
$$Standard : T(n) = 8\, T(\frac{n}{2}) + 4 Add(\frac{n}{2})$$

Le gain en exposant est si faible que l'algorithme est intéressant seulement si $n > 1000$.

## 12.3 Transformée de Fourier rapide discrète (FFT)

Nous allons étudier en détail une application du précepte "diviser pour résoudre" qui a révolutionné récemment le monde du traitement du signal : la transformée de Fourier rapide discrète, ou FFT[2]. Cette technique a rendu possible en pratique l'utilisation des techniques de filtrage par transformée de Fourier, en faisant passer d'un coût quadratique à un coût en $n \log(n)$. L'algorithme FFT permet également, comme nous le verrons, de calculer le produit de deux polynômes de degrés $n$ en $O(n \log(n))$ opérations (nous avions obtenu une complexité de $O(n^{1.59})$ en appliquant directement diviser pour résoudre).

[2] Fast Fourier Transform.



### 12.3.1 Transformée de Fourier et transformée discrète

**Définitions**

- La **transformée de Fourier** $\hat{f}$ d'une fonction $f$ est définie par :

$$\hat{f}(x) = \int f(y) e^{-2\pi ixy} dy$$

- Le **produit de convolution** $f * g$ de deux fonctions est :

$$(f * g)(x) = \int f(x-u) g(u) du$$

**Propriété :** Transformée de fourier et produit de convolution sont reliés par la relation : $\widehat{f * g} = \hat{f}.\hat{g}$

Cette propriété a un grand intéret en pratique. En effet, en traitement du signal, le signal original $s$ est souvent observé à travers un capteur qui le transforme par une réponse impulsionnelle $h$ connue. Le signal recueilli est alors $s * h$, et il s'agit de restaurer $s$. En passant $s * h$ dans l'espace transformé, il suffit de diviser par $\hat{h}$ pour obtenir $\hat{s}$, puis une transformée de Fourier inverse nous ramène à $s$.

La transformée de Fourier discrète se définit de manière analogue et a la même propriété vis à vis du produit de convolution :

**Définition :** Soit $X = (x_0 \dots x_{n-1})$ un vecteur dont les coefficients sont des nombres complexes. Sa **transformée de Fourier discrète à l'ordre** $n$ est un vecteur complexe noté $\hat{X} = (a_k)_{k \in \{0 \dots n-1\}}$, défini par :

$$a_k = \sum_{j=0}^{n-1} x_j \omega^{kj}$$

où $\omega = e^{-\frac{2i\pi}{n}}$ est une racine $n^{\text{ième}}$ de l'unité. En notation matricielle, on peut écrire :

$$\hat{X} = \begin{pmatrix} 1 & \cdots & 1 & \cdots & 1 \\ \vdots & & \vdots & & \vdots \\ 1 & \cdots & \omega^{kj} & \cdots & \omega^{k(n-1)} \\ \vdots & & \vdots & & \vdots \\ 1 & \cdots & \omega^{(n-1)j} & \cdots & \omega^{(n-1)^2} \end{pmatrix} X = \Omega X$$

La transformation a un inverse défini par $(\Omega^{-1})_{jk} = \frac{1}{n}\omega^{-jk}$.

Le calcul direct d'une transformée de Fourier en utilisant cette définition se fait en $O(n^2)$ opérations ($n$ additions et multiplications pour chacun des $n$ termes de $\hat{X}$). Pour des problèmes réels (signal de la parole, images, etc), $n$ est autour de valeurs comme 256 ou 1024, voire 512x512. Il est donc particulièrement important de réduire le coût de cette transformation.

### 12.3.2 Algorithme de FFT : diviser pour résoudre

Supposons pour simplifier que $n$ est une puissance de 2 ($n = 2^N$), et posons $n = 2m$. Coupons chacun des termes $a_k$ à calculer en deux moitiés :

$$a_k = \sum_{j=0}^{m-1} x_j \omega^{kj} + \sum_{j=m}^{n-1} x_j \omega^{kj}$$

En posant $j' = j - m$ on a :

$$\begin{aligned} a_k &= \sum_{j=0}^{m-1} x_j \omega^{kj} + \sum_{j'=0}^{n-m-1} x_{j'+m} \omega^{k(j'+m)} \\ &= \sum_{j=0}^{m-1} x_j \omega^{kj} + \sum_{j'=0}^{m-1} x_{j'+m} \omega^{kj'} \omega^{km} \end{aligned}$$

Or, on peut remarquer que $\omega^m = e^{\frac{-2i\pi m}{n}} = e^{-i\pi} = -1$.

- Si $k$ est pair, $k = 2r$, et $\omega^{km} = (-1)^{2r} = 1$. D'où :

$$a_{2r} = \sum_{j=0}^{m-1} (x_j + x_{j+m}) \omega^{2rj}$$

  Comme $\omega^2 = e^{\frac{-2i\pi}{m}}$, les nombres $a_{2r}$ constituent la transformée de Fourier à l'ordre $m = n/2$ du vecteur de composantes :

$$\alpha_j = x_j + x_{j+m} \quad (0 \le j \le m-1)$$

- Si $k$ est impair, $k = 2r + 1$, $\omega^{km} = -1$. et donc On a donc : $\omega^{(2r+1)(j+m)} = \omega^{(2r+1)j} \omega^{km} = -\omega^{(2r+1)j}$. D'où :

$$a_{2r+1} = \sum_{j=0}^{m-1} (x_j - x_{j+m}) \omega^j \omega^{2rj}$$

  On reconnaît la transformée de Fourier à l'ordre $m = n/2$ du vecteur de composantes :

$$\beta_j = (x_j - x_{j+m}) \omega^j \quad (0 \le j \le m-1)$$

**Algorithme :** Ce qui précède conduit à l'algorithme :
Si la taille de $X$ est supérieure à 1 – sinon $X$ est son propre transformé

- calculer $\alpha$ et $\beta$, vecteurs de taille $m = n/2$,
- appeler récursivement le calcul de FFT sur $\alpha$ et $\beta$,
- *entrelacer* les termes des résultats pour former la FFT de $X$.



**Complexité en temps :** On s'est ramenés du calcul d'une transformée de Fourier d'ordre $n$ à deux calculs de transformées de Fourier d'ordre $n/2$, dont il suffit d'entrelacer les termes (ce qui se fait en $O(n)$) pour obtenir le résultat. Le théorème de la section 12.1, appliqué pour $a = b = 2$, nous dit que $T(n) = n\log(n)$.

### 12.3.3 Vers une bonne implantation

Si on implante l'algorithme précédent directement, on risque de se heurter à des problèmes de *place mémoire*, du fait du stockage de vecteurs intermédiaires $\alpha$ et $\beta$. Notez que tous les caculs sont faits avant les appels récursifs et que donc la place des paramètres d'entré est libre après ces calculs. En pratique, on utilise directement l'espace de $X$ pour stocker les valeurs calculées en les rangeant dans les deux moitiés inférieures et supérieures de $X$, et le même espace servira pour y construire le résultat. D'où le programme :

```
fonction FFT(X:vecteur) : vecteur
  var X1: vecteur;
  procedure FFT1(k,q:entier)   -- transforme la partie X1(k..q-1)
    taille := q-k;
    si taille > 1 alors
      omega = racine_primitive (1, taille) ;
      m := taille div 2;
      pour i de k a k+m-1 faire
        a := X1(i); b:= X1(i+m);
        X1(i) := a+b; X1(i+m) := (a-b) * omega^(i-k);
      fin pour;
      FFT1(k, k+m-1);
      FFT1(k+m,q);
      entrelacer(k,q);         -- entrelace les deux moities de X1(k..q)
    fin si;
  fin FFT1;
debut
   X1:=X;
   FFT1(0, taille(X));
   return X1;
fin;
```

**Exercice 12.1** *Combien d'appels à* `racine_primitive` *opère-t-on si l'appel initial porte sur un vecteur de taille* $2^n$ *? Combien d´élévation à puissance opère-t-on sur* `omega` *?*

*Pour éviter cette complexité inutile, on peut remarquer que si l'appel initial porte sur un vecteur de taille initiale* $n$*, alors seule* $n$ *différentes valeurs peuvent être prises pour les puissances de* `omega` *au cours des différents appels. Ces différentes valeurs peuvent donc être précalculées et stockées dans un tableau, où elles seront prises selon les besoins. Modifiez le programme pour implanter cette optimisation.*

Entrelacer les deux moitiés du vecteur $X$ consiste à effectuer la permutation $p$ définie par :

- si $i < m$ , $p(i) = 2i$
- si $i \ge m$, $p(i) = 2(i - m) + 1$

Ce qui se regroupe en : $p(i) = 2(i \bmod m) + i \operatorname{div} m$. Or, nous avons suposé que n est la puissance de deux $n = 2^N$. Alors $m = 2^{N-1}$. Si on peut manipuler la représentation binaire de $i$, notée $(i_N i_{N-1} \ldots i_0)$, alors $i \operatorname{div} m = i_N$ et $i \bmod m = (i_{N-1} \ldots i_0)$. Par conséquent, $p(i)$, représenté par $(i_{N-1} \ldots i_0 i_N)$ est obtenu par décalage circulaire. Au cours de l'algorithme, cette opération n'est plus effectuée sur $X$ complet mais sur une partie du vecteur.

Si on analyse de manière globale le comportement de la fonction récursive FFT1, on voit qu'elle effectue une phase de descente (appels récursifs sur des tableaux de plus en plus petits), lors de laquelle les valeurs de $X$ sont modifiées, puis une phase de remontée, avec entrelacements successifs des résultats partiels.

En fait, comme chaque entrelacement est une permutation, il est possible, et considérablement plus efficace, d'effectuer directement la permutation composée, plutôt que chacune successivement. Etudions cette composée : soit un indice $j$ de représentation binaire $(j_N \ldots j_0)$. A l'entrelacement des deux moitiés d'un tableau de taille $2^k$, son rang dans le tableau est donné par les derniers bits $j_k \ldots j_0$. Nous avons vu qu'il est transformé par décalage circulaire en $j_{k-1} \ldots j_0 j_k$. Comme $k$ prend, pendant la phase de remontée, les valeurs successives 1, 2, .., N, l'indice $j$ est finalement remplacé par l'indice de repésentation $j_0 j_1 \ldots j_{N-1} j_N$, correspondant à son image miroir en binaire ! La permutation composée que nous cherchons est donc tout simplement constituée de $m = n/2$ échanges, d'un indice à son image miroir en binaire. En séparant la partie récursive de la FFT et la phase d'entrelacements (itérative) on obtient :

```
fonction FFT(X:vecteur) : vecteur  -- indice allant de 0 à n-1
  var X1: vecteur;
  var n := taille (X);
  var omega_puissance (0..n-1) de complexe;
-- ce tableau stocke les puissances successives de la racine
-- primitive de l'unité

  procedure FFT2(k,q:entier)          -- comme FFT1 sauf les
      -- entrelacements et les
      -- puissances de omega précalculées
    taille := q-k;
    pas_omega :=  n / taille;   -- progression dans le tableau des puissances
    si taille > 1 alors
      m := taille div 2;
      pour i de k a k+m-1 faire
        a := X1(i); b:= X1(i+m);
        X1(i) := a+b; X1(i+m) := (a-b) * omega_puissance ((i-k)* pas_omega);
      fin pour;
      FFT2(k, k+m-1);
      FFT2(k+m,q);
  fin FFT2;

debut
   X1:=X;
   -- précalcul des puissances de la racine primitive
   omega := racine_primitive(1, n);
   omega_puissance(0) := 1;
   pour i de 1 à n-1 répéter
       omega_puissance (i) := omega_puissance(i-1) * omega;
```



```
   FFT2(0, n);
   pour j de 1 a taille(X)-1          -- identité pour les deux extrêmes
     j1:= miroir(j);                  -- image miroir en notation binaire
     si j1>j alors X1(j)<->X1(j1);    -- sinon échange deja fait
   return X1;
fin;
```

### 12.3.4 Version itérative

Si on dessine l'arbre des appels de `FFT`, on se retrouve dans la même situation que dans celle du tri-fusion, à la différence que tous les calculs sont faits avant les appels récursifs ici, alors que dans le cas du tri, ils sont fait après (la fusion).

La même technique de prise en compte des calculs peut être appliquée à cette modification près: on traite d'abord tout le tableau des données, puis les deux moitiées, puis les quatre quarts, etc. Lorsque la taille dess blocs atteint 1 tout est terminé, il ne reste plus que l'interclassement de l'algorithme précédent à effectuer.

On aboutit alors à l'algorithme suivant:

```
procedure FFT( X: in out vecteur)  -- X vecteur d'indice de 0 à n-1
  var n := taille (X) ;
  var omega_puissance (0..n-1) de complexe;

debut
   -- précalcul des puissances de la racine primitive
   omega := racine_primitive(1, n);
   omega_puissance(0) := 1;
   pour i de 1 à n-1 répéter
       omega_puissance (i) := omega_puissance(i-1) * omega;
   fin pour;

  -- itération sur les niveaux
  taille := n; pas_omega:= 1;
  tant que taille > 1 répéter
    m := taille /2;
    k := 0;
    -- itération sur chaque bloc de grandeur ‘‘taille"
tant que k<n répéter
   pour i de k a k+m-1 répéter
               a := X(i); b:= X(i+m);
                X(i) := a+b;
                X(i+m) := (a-b) * omega_puissance ((i-k)* pas_omega);
          fin pour;
          k := k + taille;
      fin tant que;
     taille:= m; pas_omega := pas_omega *2;
   fin tant que;

-- il ne reste plus qu'à interclasser comme dans la procédure précédente
```

```
 ....
fin FFT;
```

### 12.3.5 Application : produit de polynômes en $n\log(n)$

Nous avons vu plus haut que la transformée de Fourier discrète est inversible, et sa propriété vis à vis du produit de convolution, défini par :

$$(X * Y)_k = \sum_{j=0}^{n-1} x_j y_{k-j}$$

s'écrit : $\widehat{X * Y} = \hat{X}.\hat{Y}$.

En remarquant que le produit de deux polynomes correspond au produit de convolution des vecteurs qui les représentent, ces propriétés permettent de calculer le produit en $O(n\log(n))$ :

1. On étend les polynômes par des zéros jusqu'au degré $2n$ (degré du produit).
2. On calcule $\hat{P}$ et $\hat{Q}$ en $n\log(n)$ étapes.
3. On calcule le produit scalaire de vecteurs $\hat{P}.\hat{Q} = P \hat{*} Q$ en $2n$ étapes.
4. On effectue la transformation de Fourier inverse pour avoir $X * Y$ ($n\log(n)$ étapes).

**Remarque 12** *A moins d'effectuer une implémentation spécifique dans un corps fini comportant des racines n-ièmes de l'unité, le résultat obtenu par l'algorithme précédent est un résultat numérique, dont la précision est donc forcément limitée.*

## 12.4 Algorithmes d'enveloppe convexe

Les calculs d'enveloppe convexe sont à la base de nombreuses applications en reconnaissance des formes, vision et infographie. Ils vont nous permettre de comparer un algorithme de type diviser pour résoudre avec une méthode de construction purement itérative.

**Définition :** L'**enveloppe convexe** d'un ensemble de $n$ points du plan est le plus petit polygone convexe qui les contient (voir figure 12.1).

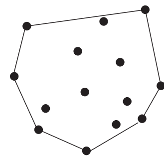

Figure 12.1: Enveloppe convexe de $n$ points du plan.



**Propriétés :**

1. Les sommets de l'enveloppe convexe se trouvent parmi les $n$ points.
2. Une droite quelconque partage le plan en deux parties. Dans chacune de ces parties, celui parmi les points donnés qui est le plus éloigné de la droite est un sommet de l'enveloppe convexe.
3. Une droite qui joint deux des $n$ points est le support d'une arête de l'enveloppe convexe *ssi* tous les autres points sont du même côté.

Calculer l'enveloppe convexe consiste à trouver les points qui forment son contour dans l'ordre. On peut pour cela envisager plusieurs méthodes, par exemple :

- Chercher à appliquer une stratégie diviser pour résoudre.
- Déterminer itérativement l'enveloppe convexe, en y ajoutant successivement des sommets.

### 12.4.1 Algorithme de "l'enveloppe rapide" (diviser pour résoudre)

Comme indiqué en introduction de ce chapitre, il y a deux manières d'appliquer la stratégie diviser pour résoudre : diviser les données telles qu'elles se présentent (ici, les points donnés dans le désordre), ou effectuer un partage "intelligent" des données en prétraitement, de manière à ce que les résultats partiels se combinent bien. Il est clair que dans le cas qui nous intéresse, la seconde stratégie semble la meilleure. En effet, il semble délicat de fusionner efficacement les enveloppes convexes de deux sous ensemble quelconques des points.

Pour construire deux moitiés du résultat, on a besoin de connaître deux sommets $P$ et $Q$ de l'enveloppe convexe. Prenons par exemple les deux points d'abscisse minimale et maximale (si elles sont égales, le problème est dégénéré, et l'enveloppe convexe est constitué des points d'ordonnée minimale et maximale).

La droite $PQ$ partage les données $E$ en deux parties $E'$ est $E$" contenant toutes deux $P$ et $Q$. L'enveloppe convexe de chacune de ces parties est un polygone d'arête $PQ$ : pour l'une $PP'_1 \dots P'_{r'}Q$, et pour l'autre $QP"_1 \dots P"_{r"}P$. Le résultat cherché est la concaténation de ces deux résultats partiels, dont on retire l'arête $PQ$ de manière à ce qu'elle ne figure pas en double.

A l'étape suivante de l'algorithme (construction des enveloppes des deux sous ensembles), il faut de nouveau selectionner un point de l'enveloppe pour construire deux sous résultats. La propriété 2. nous dit que le point $S$ qui est le plus éloigné de la droite $PQ$ fait partie de l'enveloppe (voir figure 12.2). On se sert alors des ensembles $E_1$ et $E_2$ pour continuer le calcul (les points intérieurs au triangle $PSQ$ ne sont pas considérés dans la suite, puisqu'ils ne peuvent pas faire partie de l'enveloppe convexe).

L'algorithme complet est donc :

```
procedure enveloppe(E)
  procedure env(E', P, Q)
    S:= le point de E' le plus eloigné de PQ;
    si S est sur PQ alors
      renvoyer (P,Q);
    sinon
      E1:= les points de E' délimités par PS ne contenant pas Q;
```

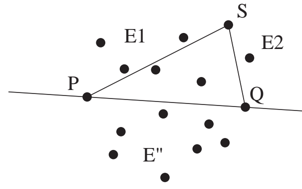


Figure 12.2: Partage des données pour l'algorithme d'enveloppe rapide.

```
      E2:= les points de E' délimités par QS ne contenant pas P;
      renvoyer conc(env(E1,P,S),reste(env(E2,S,Q));
      -- les enveloppes sont données par la liste ordonnée des sommets,
      -- et l'opération reste enleve la seconde occurence de S.
  fin env
debut
  calculer P et Q;
  calculer E' et E";
  renvoyer conc(enveloppe(E',P,Q),reste(enveloppe(E",Q,P));
fin;
```

**Complexité :** Soient $n_1$ et $n_2$ les nombres respectifs de points de $E_1$ et $E_2$. Le temps mis à déterminer le point $S$ et à trier les points entre E1 et E2 est linéaire. On a donc :

$$T(n) = cn + T(n_1) + T(n_2) \quad \text{avec} \quad n_1 + n_2 \leq n - 1$$

Cette formule ressemble à celle de la complexité du tri rapide, pour laquelle on avait $n_1 + n_2 = n - 1$. On reconnait donc un algorithme en $O(n \log(n))$ en moyenne.

**Exercice 12.2** *Dessiner la configuration qui constitue le cas le pire pour cet algorithme. Quelle est la complexité dans ce cas ?*

**Exercice 12.3** *Si l'on suppose que la moitié des points en considération sont dans le triangle PSQ de l'algorithme, et que un quart est au delà du coté PS et de même un quart au delà du coté SQ, quelle est la complexité de cet algorithme ?*

*Ces hypothèses de position sont elles réalistes ? Discutez.*

### 12.4.2 Algorithme itératif de Jarvis

Une idée totalement différente pour calculer une enveloppe convexe consiste à partir d'un premier sommet $P_1$ (par exemple le point d'abscisse minimale), et à trouver successivement les autres sommets de l'enveloppe en "tournant" autour des données. Plus précisément, si on a trouvé les sommets jusqu'à $P_i$, la troisième propriété nous dit que $P_{i+1}$ est le point tel que la droite $P_iP_{i+1}$ laisse tous les autres points du même coté. Cela peut être déterminé en prenant pour $P_{i+1}$ le point $Q$ qui minimise l'angle $a(P_{i-1}P_i, P_iQ)$. Voir figure 12.3.

**Complexité :** On remarque qu'à chaque étape, le nouveau sommet est calculé en $O(n)$ (calcul de l'angle minimal). Le nombre $h$ d'étapes est égal au nombre de sommets de



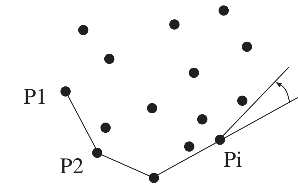


Figure 12.3: Calcul itératif d'une enveloppe convexe par l'algorithme de Jarvis.

l'enveloppe convexe. Dans le cas le pire, $h = O(n)$, et la complexité est donc de $O(n^2)$. On peut en déduire la complexité dans trois cas typiques :

- Si la distribution des points est uniforme dans un polygone convexe, le nombre moyen de points de l'enveloppe convexe est $O(\log(n))$. Le coût est alors un $O(n \log(n))$, comme pour l'enveloppe rapide.
- Si la distribution est uniforme dans un cercle, le nombre moyen de points de l'enveloppe convexe est un $O(n^{\frac{1}{3}})$. Cela donne un coût moyen de $O(n^{\frac{4}{3}})$, plus élevé que celui de l'enveloppe rapide.
- Si la distribution est gaussienne dans le plan, le nombre moyen de points de l'enveloppe convexe est un $O(\log(n)^{\frac{1}{2}})$. Cela donne un coût moyen de $O(n \log(n)^{\frac{1}{2}})$, meilleur que celui de l'enveloppe rapide.

Ceci illustre le fait que le principe "diviser pour résoudre" n'est pas une recette miracle, et que d'autres formes de raisonnement conduisent parfois à une meilleure solution. Le chapitre suivant présente une telle forme de raisonnement en algorithmique.

# Chapitre 13

# Programmation efficace des formules récursives

## 13.1 Introduction

La décomposition récursive est un principe fondamental en algorithmique. Dans le chapitre précédent, sous la dévise “diviser pour règner”, nous avons vu qu'une telle décomposition peut conduire à des programmes d'une qualité remarquable. Dans le chapitre présent, nous étudions des décompositions récursives pour lesquelles une programmation directe, trop naïve, conduit à des inefficacités grossières. Nous introduisons alors plusieurs transformations de programmes, à caractère général, permettant d'aboutir à des algorithmes efficaces.

**Exemples :**

- Suite de Fibonacci. Si on programme directement la relation de récurrence ci dessous, on effectue un grand nombre de calculs redondants, comme le montre l'arbre des appels de la figure 13.1 :

$$fib(n) = fib(n-1) + fib(n-2)$$

Et on obtient au bout du compte une complexité exponentielle...

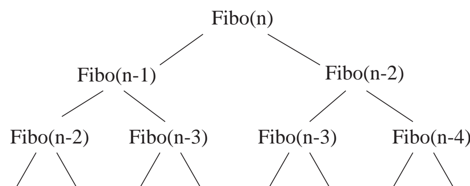


Figure 13.1: Arbre des appels d'une procédure récursive calculant la suite de Fibonacci.

- Le même phénomène se produit pour le calcul récursif des combinaisons $C_n^p$, définies par :

$$C_n^p = C_{n-1}^p + C_{n-1}^{p-1}$$





**L'élimination des calculs redondants** est une technique de résolution qui peut être utilisée chaque fois que l'écriture récursive d'un algorithme conduit à effectuer plusieurs fois le même calcul. L'approche repose sur une *évaluation ascendante* [1] des résultats intermédiaires. La fin de ce chapitre décrira l'application de cette technique à la **programmation dynamique**, c'est à dire à la résolution de problèmes d'optimisation s'exprimant par une équation de récurrence.

**Remarque 13** *L'approche "diviser pour résoudre" que nous avons étudiée dans le chapitre précedent échappe à ces problèmes de redondance, dans la mesure où il s'agissait de lancer le processus de résolution sur deux moitiés dijointes des données. Notons que c'est une approche “descendante" (top-down), par opposition aux méthodes ascendantes que nous allons introduire ci-dessous. Diviser pour résoudre est ainsi une méthode naturellement récursive, alors que l'élimination des calculs redondants conduira généralement à l'écriture d'algorithmes itératifs.*

## 13.2 Elimination des calculs redondants

Au lieu de représenter les appels d'une procédure récursive comme un arbre, on peut utiliser un graphe, en confondant les sommets qui portent le même étiquette. On obtient ainsi le **graphe des appels**, sur lequel la présence de calculs redondants se visualise très bien. Des techniques de marquage ou d'exécution dans un "ordre topologique" peuvent alors être utilisées pour éliminer les redondances.

### 13.2.1 Exemple : Fibonacci

(à traiter sous forme d'exercice)

1. Dessiner le graphe des appels pour la suite de Fibonacci. Quelle technique, vue dans le cours sur les graphes, peut-on utiliser pour éviter de traiter deux fois le même noeud ?

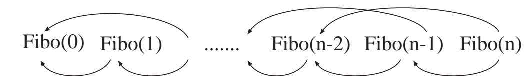


Figure 13.2: Graphe des appels pour Fibonacci

Réponse: on peut utiliser les techniques de "marquage". Au premier passage par chaque noeud, on le marque. On évite ainsi de refaire le calcul ultérieurement. Marquer ne suffit pas : il faut aussi stocker le résultat. Comme la procédure Fibonacci ne renvoie que des valeurs $\geq 0$, cela peut être fait dans le même tableau. On obtient alors :

```
integer fonction fib(n)
{
  si (tab(n) = -1) alors  -- indique que fib(n) est non calculé
                 {
```

[1] En anglais "bottom-up". C'est ascendant par rapport à notre habitude de dessiner les arbres avec les feuilles en bas.

```
        tab(n) := si n=0 ou n=1 alors n
                                sinon fib(n-1) + fib(n-2);
                 }
  return tab(n);
}
```

2. L'inconvénient de cette méthode, c'est qu'elle prend beaucoup de place mémoire (ici, un tableau de taille $n$), ce qui la rend inutilisable pour des données de grande taille. Proposer une autre méthode pour limiter le coût mémoire (s'inspirer encore du cours sur les graphes).

Réponse: Analysons le graphe d'appel de fib : Un arc $n \to n'$ signifie "pour calculer l'argument $n$, on a besoin de la valeur de l'argument $n'$". En regardant le graphe, on peut trouver un ordre de traitement des sommets (correspondant à un tri topologique des tâches à accomplir) qui limite le nombre de données à garder en mémoire : pour caluler $fib(n)$, seuls sont utiles $fib(n-1)$ est $fib(n-2)$. On en déduit la procédure :

```
integer fonction fib(n)
{
  a, b, i: integer;

  a := 1;
  b := 1;
  pour i variant de 2 a n faire
  {
    aux:= b;
    b := a+b;     -- b contient fib(i)
    a := aux;     -- a contient fib(i-1)
  }
  return b;
}
```

Sur cet exemple, on utilise seulement deux variables auxiliaires, et le coût est $O(n)$ au lieu d'être exponentiel ! On remarque que l'écriture de cette procédure nous a amené à passer (naturellement) d'une programmation récursive à une programmation itérative.

### 13.2.2 Synthèse

La méthodologie à utiliser en programmation dynamique est la suivante :

- On détecte le calculs redondants en dessinant le graphe des appels d'une procédure.
- Pour les éliminer, on a le choix entre deux types d'algorithmes :
  1. Calcul avec marquage et stockage des résultats partiels, de manière à éviter de faire plusieurs fois le même calcul.
  2. Calcul selon un ordre topologique sur les noeuds du graphe, ce qui a souvent l'avantage d'économiser la place mémoire. Ce procédé conduit à des algorithmes itératifs, qui, dans certains cas, peuvent être plus coûteux en temps.

Alors que la première méthode peut être utilisée de manière systématique, la seconde correspond à une analyse fine du problème à résoudre de la part du programmeur.



**Remarque 14** *Utiliser un ordre topologique correspond bien à une construction ascendante (bottom-up) des solutions, des problèmes les plus simples aux plus compliqués. Nous avons déjà vu des exemples d'application de cette stratégie :*

- *pour la construction d'un arbre binaire de recherche optimal dans le chapitre "structures de données" ;*
- *avec les algorithmes de Warshall et de Floyd dans le cours sur les graphes.*

*Il s'agit dans chaque cas de problèmes d'optimisation, résolus par construction de sous-solutions optimales.*

## 13.3 Exemples d'applications

**Exercice 13.1** *Pour chacun des exemples ci-dessous,*

1. *Dessiner le graphe des appels correspondant à une solution récursive naïve.*
2. *Donner une solution qui évite les calculs redondants en utilisant une méthode de marquage.*
3. *Trouver un autre algorithme, basé sur un ordonnancement "intelligent" des tâches, qui permette d'éviter les redondances tout en limitant si possible la place mémoire utilisée.*

### 13.3.1 Combinaisons $C_n^p$

Les nombres $C_n^p$ sont définis par la formule de récurrence suivante :

$$\begin{aligned} C_n^p &= C_{n-1}^p + C_{n-1}^{p-1} \quad \text{pour } 0 < p < n \\ C_n^0 &= C_n^n = 1 \end{aligned}$$

1) Solution récursive naïve :

```
procedure combinaison1(n,p:entiers; resultat r:entier)
  -- precondition: 0 <= p <= n
{
  si p=0 ou p=n alors r:=1;
  sinon
  {
    variables r1, r2: entier;
    combinaison1(n-1, p, r1);
    combinaison1(n-1, p-1, r2);
    r := r1 + r2;
  }
}
```

Cet algorithme corespond au graphe des appels de la figure 13.3.

2. Algorithme avec marquage :
Les éléments à marquer sont les sommets du graphes des appels. Il faut donc trouver une représentation du graphe d'appels permettant de réaliser le marquage et si possible

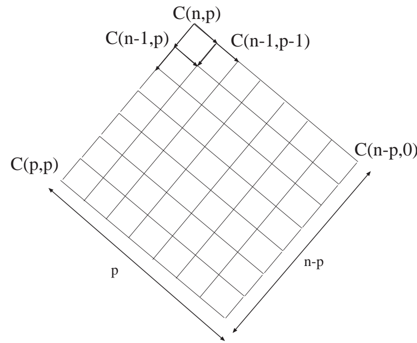


Figure 13.3: Graphe des appels d'un calcul récursif des $C_n^p$

de stocker simultanément les résultats (pour les nombres de Fibonacci, on utilisait un tableau d'entiers de taille $n$). Ici, une matrice M(p..n,1..p) d'entiers convient. On initialise le contenu à 0 par exemple, ce qui signifie non marqué, puis on y stocke les résultats au fur et à mesure qu'ils sont trouvés.

```
procedure combinaison1(n,p:entiers; resultat r:entier)
  -- precondition: 0 <= p <= n
{
  si p=0 ou p=n alors r:=1;
  sinon
  {
    si C(n,p)=0      -- (n,p) non marque
    {
      variables r1, r2: entier;
      combinaison1(n-1, p, r1);
      combinaison1(n-1, p-1, r2);
      r := r1 + r2;
      C(n,p) := r;
    }
    sinon r:=C(n,p);
  }
}
```

La place mémoire utilisée est $O(np - p^2) = O(np)$, mais le gain en temps d'execution est énorme ($O(np)$ au lieu d'exponentiel !).

3) Cherchons un bon ordonnancement des tâches sur le graphe des appels :

L'idée consiste à faire les calculs par ligne, en remontant dans le graphe des appels, ce qui fait qu'on a seulement besoin de stocker une ligne (tableau 1..p) avant de calculer la ligne au dessus. En d'autres termes, pour calculer $C_n^p$

1. Calculer la ligne d'en dessous jusqu'à $C_{n-1}^p$, stocker les résultats ;
2. Calculer la queue de la ligne courante, de $C_{n-p}^0$ en remontant jusqu'à $C_{n-1}^{p-1}$ (on peut écraser au fur et à mesure les valeurs stockées dans le début du tableau).



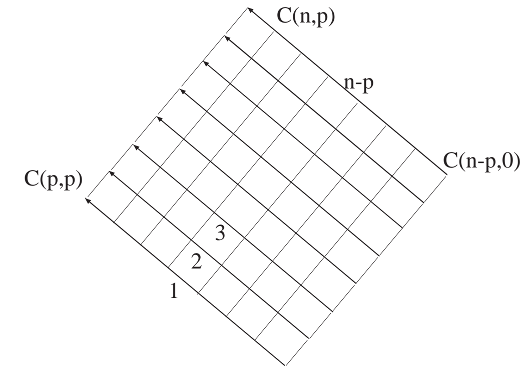


Figure 13.4: Un ordonnancement des tâches dans le graphe des appels des $C_n^p$

3. Sommer les résultats obtenus pour $C_{n-1}^p$ (case $p$ du tableau) et pour $C_{n-1}^{p-1}$ (case $p-1$ du tableau).

Cet algorithme s'écrit simplement de manière itérative :

```
-- Calcul de C(n,p)
pour i dans 1..p faire tab(i):= 1; -- initialisation tab(i)=C(i,i)=1

pour k dans 1..n-p faire        -- Calcul de la ligne debutant a C(k,p),
  pour i dans 1..p faire        -- de C(k,1) a C(k,p)
    tab(i) := tab(i) + tab(i-1);-- Utilise tab(0)=C(k,0)=1, et le fait que
                                -- tab contenait la ligne du dessous.
return tab(p);
```

**Exercice 13.2** *La suite de Fibonacci est généralement définie sous la forme :*

$$\begin{aligned} f_0 &= f_1 = 1 \\ f_n &= f_{n-1} + f_{n-2} \quad n \geq 2 \end{aligned}$$

*Le but est détudier une autre formulation récursive, qui conduit à de meilleurs temps de calcul.*

1. *Montrer que, pour tout $n \geq 2$,*

$$f_n = \begin{array}{ll} si & n = 2k \text{ alors } (f_k)^2 + (f_{k-1})^2 \\ si & n = 2k+1 \text{ alors } (f_k)^2 + 2f_k f_{k-1} \end{array}$$

2. *Montrer que pour tout $n \geq 2$, le triplet $f_{n-2}$, $f_{n-1}$, $f_n$ peut s'exprimer en fonction du triplet $f_{(ndiv2)-2}$, $f_{(ndiv2)-1}$, $f_{ndiv2}$.*
3. *En utilisant les propriétés précédentes, écrire une procédure d'en-tête* `Fib(n : entier ; var p, q, r : entier)` *qui calcule la suite de Fibonacci (pour n2, p sera $f_n$, q sera $f_{n-1}$ et r sera $f_{n-2}$ ).*

   *Analyser le temps et la taille de la mémoire pour un appel* `Fib(n, p, q, r)`*. Comparer avec les performances du programme vu au début de ce chapitre, fondé sur la première définition récursive de la suite.*

4. *Transformer la procédure 3) en un programme non récursif. Justifier votre transformation.*

# Chapitre 14

# Application : programmation dynamique

Les paragraphes suivants présentent des exemples de problèmes d'optimisation s'exprimant par une relation de récurrence. Ces problèmes constituent un champ d'application privilégié pour les techniques d'élimination de redondances que nous venons d'étudier.

## 14.1 Reconnaissance de chaînes de caractères bruitées

Le problème très académique suivant connait des applications importantes dans l'analyse de chaînes de DNS.

On considère des chaînes sur un alphabet fini $V$. Une chaîne de référence $A = a_1..a_n$ est comparée avec une chaîne $B = b_1..b_m$. La seconde est une version bruitée de la première, et les perturbations possibles sont de trois types :

1. disparition d'un élément de A
2. insertion d'un élément parasite
3. modification d'un élément.

On attribue à ces perturbations des coûts entiers positifs. Pour tout $a$ de $V$, on note $D(a)$ le coût de la disparition de $a$, $P(a)$ le coût de l'insertion de $a$ ; pour tous $a$ et $b$ éléments de $V$ on note $S(a, b)$ le coût de la substitution de $b$ à $a$. Le coût d'une transformation de $A$ en $B$ est défini comme étant la somme des coûts des perturbations de caractères qui la composent.

On cherche à calculer la "distance" de $B$ à $A$, c'est à dire le coût minimal d'une transformation de $A$ en $B$ (on supposera que les chaînes de caractères sont stockées dans des tableaux).

1. Ecrivons une solution récursive naïve, et montrons qu'elle donne lieu à des calculs redondants.





On a une relation de récurrence sur les coûts ($\epsilon$ est la chaîne vide) :

$$\begin{aligned}
d(aX, bY) &= Min\,(S(a,b) + d(X,Y)\ ,\ D(a) + d(X, bY)\ ,\ P(b) + d(aX, Y)) \\
d(aX, \epsilon) &= D(a) + d(X, \epsilon) \\
d(\epsilon, bY) &= P(b) + d(\epsilon, Y) \\
d(\epsilon, \epsilon) &= 0
\end{aligned}$$

On en déduit une procédure récursive donnant la distance entre les parties $(i..n)$ et $(j..m)$ des deux tableaux :

```
fonction distance(A: tableau(1..n), B: tableau(1..m), i,j: entiers) :
entier
{
  si i=n+1          -- La partie qui nous interesse de A est vide
  {
    si j=m+1
      alors return 0;  -- Comparaison de deux chaines vides
      sinon  return P(B(j)) + distance(A,B,i,j+1);
  }
  sinon
  {
    si j=m+1
      alors return D(A(i)) + distance(A,B,i+1,j); -- B vide, A non
      sinon return Min( S(A(i), B(j)) + distance(A,B,i+1,j+1) ,
                        D(A(i)) + distance(A,B,i+1,j) ,
                        P(B(j)) + distance(A,B,i,j+1) );
  }
}
```

Preuve d'arrêt : définissant la "taille" du problème par $t = (n + 1 - i) + (m + 1 - j)$. $t$ décroit strictement d'un appel récursif aux appels secondaires qu'il engendre. D'autre part, une solution directe est donnée pour un problème de taille 0 (ie. si $i = n+1$ et $j = m+1$). Cela prouve que la procédure termine.

Coût de la procédure (en temps d'exécution) : Soit $C(p, q)$ le coût de l'appel comparant des sous tableaux de tailles respectives $p$ et $q$. On a :

$$\begin{aligned}
C(p,q) &= cte + C(p-1, q-1) + C(p-1, q) + C(p, q-1) \\
C(p,0) &= cte + C(p-1, 0) \\
C(0,q) &= cte + C(0, q-1) \\
C(0,0) &= 0
\end{aligned}$$

Si on exprime ce coût en fonction de la taille $t = p + q$ on a :

$$\begin{aligned}
C(t) &= cte + C(t-2) + 2\,C(t-1) \text{ dans le cas général} \\
C(t) &= cte + C(t-1) \text{ dans certains cas dégénérés} \\
C(0) &= 0
\end{aligned}$$

On associe l'équation homogène $x^2 - 2x - 1 = 0$ à la première des équations ci-dessus. Cela donne une solution exponentielle pour $C(t)$ dans le cas le pire (cas le pire = deux tableaux de même taille) :

$$C(t) \simeq O(\phi^t)$$

où $\phi$ est une solution de l'équation homogène.

Pour montrer que des calculs redondants sont effectués, on dessine le graphe des appels. Ce graphe comporte des cycles : plusieurs chemins mènent au même sommet.

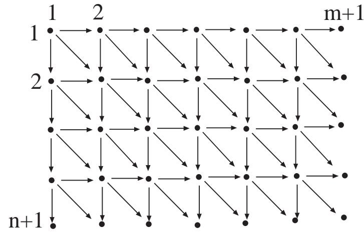


Figure 14.1: Graphe des appels pour le calcul d'une distance entre chaînes

2) Elimination des redondances grâce à une technique de marquage. On utilise un tableau auxiliaire $dis(i,j)$ de taille $(n+1)(m+1)$ (égale au nombre de sommets du graphe des appels). On prend la convention de l'initialiser à $-1$ pour indiquer qu'un sommet n'est pas marqué.

```
-- matrice dis(i,j) initialisee a -1...
fonction distance(A: tableau(1..n), B: tableau(1..m), i,j: entiers) :
entier
{ si dis(i,j)=-1 alors {
    -- si ce n'est deja fait, on calcule dis(j,i)
    si i=n+1 { -- La partie qui nous interesse de A est vide
      si j=m+1  alors dis(i,j):=0;
      -- Comparaison de deux chaines vides
      sinon dis(i,j):= P(B(j))+distance(A,B,i,j+1);
    }
    sinon {
      si j=m+1 alors dis(i,j):= D(A(i))+distance(A,B,i+1,j);
      -- B vide, A non
      sinon dis(i,j):=Min( S(A(i),B(j)) + distance(A,B,i+1,j+1) ,
                           D(A(i)) + distance(A,B,i+1,j) ,
                           P(B(j)) + distance(A,B,i,j+1) );
    }
  }
  return dis(i,j);
  -- soit il etait deja calcule, soit on vient de le faire...
}
```

Complexité : mémoire en $O(nm)$. Temps d'éxécution en $O(nm)$ (au lieu d'un coût exponentiel) puisqu'on n'évalue qu'une seule fois chaque sommet du graphe, et qu'il y a un nombre constant d'opérations par appel.

3) Solution itérative : on cherche un ordonnancement des tâches dans le graphe des appels qui permette à la fois d'évaluer une seule fois chaque sommet, et d'économiser de la place mémoire.

On remarque du point de vue du graphe des appels, les rôles de n et m sont symétriques. Une idée consiste à remonter le long des colonnes de bas en haut et de droite à gauche



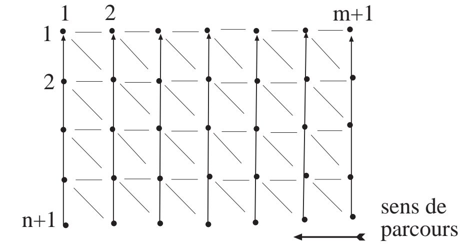


Figure 14.2: Choix d'un ordre de parcours sur le graphe des appels

si $n \leq m$, ou bien le long des lignes de droite à gauche et de bas en haut si $m < n$. Le stockage mémoire nécessaire est donc limité à $min(n+1, m+1)$ (dernière colonne ou ligne calculée).

Supposons par exemple que $n \leq m$, et soit $dis(i)$ un tableau de taille indexé sur $1..n+1$. L'algorithme itératif de calcul de la distance est :

```
dis(n+1):=0;
pour i dans n .. 1
  dis(i):=dis(i+1)+D(A(i));      -- calcul de la derniere colonne
pour j dans m .. 1  faire        -- calcul de la j-ieme colonne
{
  b:=0;                          -- init variable auxiliaire
  dis(n+1):= dis(n+1)+P(B(j));   -- valeur bas j-eme colonne
  pour i dans n .. 1 faire {     -- calcul de l'element i,j
    a := dis(i);
    dis(i) = min( a + P(B(j)),            -- a contient d(i,j+1)
                  b + s(A(i),B(j)),       -- b contient d(i+1,j+1)
                  dis(i+1) + D(A(i)) ); -- dis(i+1) contient d(i+1,j)
    b := a;
  }
}
return dis(1);
```

## 14.2 Calcul d'un ABR optimal

Nous avons vu la structure d'arbre de binaire de recherche (ABR) dans le chapitre 4; le but alors était d'optimiser le temps de recherche d'un élément dans le pire cas.

Dans ce paragraphe on cherche à optimiser le temps moyen de recherche lorsque le dictionnaire est initialisé, puis utilisé uniquement pour des recherches. Plus précisemment, on cherche à calculer la forme que doit prendre un ABR pour minimiser le temps moyen de recherche en fonction des probabilités (que l'on supposera connues) pour que chaque élément soit recherché.

**Définition :** L'arbre binaire de recherche **optimal** est l'ABR qui minimise le nombre de moyen de comparaisons lors d'une opération de recherche.

L'exercice suivant montre qu'optimal ne signifie pas forcément bien équilibré.

**Exercice 14.1** *Soit un dictionnaire contenant trois éléments, 1, 2 et 3, dont les probabilités d'être recherchés, $p_1$, $p_2$, et $p_3$ sont connues. Le dictionnaire est stocké dans un arbre binaire de recherche $T$. On note $C(T)$ le nombre moyen de comparaisons pour une recherche dans $T$.*

1. *Dessiner les 5 configurations possibles du dictionnaire.*
2. *Calculer les valeurs de $C(T)$, et en déduire quel arbre est l'ABR optimal, dans les cas suivant :*
   - *équiprobabilité de recherche des éléments : $p_i = \frac{1}{3}$, pour $i = 1, 2, 3$ ;*
   - $p_1 = \frac{1}{7}$, $p_2 = \frac{2}{7}$, $p_3 = \frac{4}{7}$.

   *Que peut-on en conclure ?*

Solution : pour chacun des 5 arbres, on écrit $C(T)$ comme la somme des $p_i$ pondérés du coût de recherche de l'élémént $i$ dans l'arbre (ce qui correspond à la hauteur de l'élément). On trouve naturellement que dans le cas d'équiprobabilité, l'arbre équilibré est le meilleur. Par contre, dans le second cas, l'arbre optimal est celui construit par les ajouts successifs de 3, puis 2, puis 1 (arbre dégénéré). Ceci montre qu'utiliser un ABR équilibré n'est pas le meilleur choix en général (les cas d'équiprobabilité sont d'ailleurs peu fréquents en pratique).

**Notations :** Nous allons maintenant présenter un algorithme permettant la construction de l'ABR optimal. Nous utiliserons les notations suivantes :

- $\{A_1, A_2, \ldots, A_n\}$ désignent les éléments du dictionnaire.
- $a_i$ est la probabilité (supposée connue) pour que $A_i$ soit recherché.
- $b_i$ est la probabilité de la recherche d'un élément absent appartenant à l'intervalle $]A_i, A_{i+1}[$. Par extension, $b_0$ est la probabilité de recherche d'un élément strictement inférieur à $A_1$, et $b_n$ celle d'une élément strictement supérieur à $A_n$.

  Les probabilités vérifient : $\sum a_i + \sum b_i = 1$ En fait, nous ne tiendrons pas compte de cette propriété, et on travaillera avec des poids entiers plutôt qu'avec des réels.
- $h_i$, $1 \leq i \leq n$, est le nombre de comparaisons pour la recherche de $A_i$ ;
- $h'_i$, $0 \leq i \leq n$, est le nombre de comparaisons pour la recherche d'un élément absent dans $]A_i, A_{i+1}[$ (et par extension dans $]-\infty, A_1[$ et dans $]A_n, \infty[$ pour $h'_0$ et $h'_n$ respectivement).

  On peut remarquer que ces nombres de comparaison correspondent à la hauteur de l'élément recherché dans l'arbre $T$ (où à celle de la feuille de l'arbre complèté $T_C$ sur laquelle on aboutit en cas d'échec).

Avec ces notations, le coût moyen d'un arbre (en nombre de comparaisons) est donné par :

$$C(T) = \sum_{i=1}^{n} a_i h_i + \sum_{i=0}^{n} b_i h'_i$$



**Théorème :** Tout sous arbre d'un ABR optimal est un ABR optimal.

Démonstration : Soit un sous arbre $T'$ de $T$, construit sur les éléments $A_{i+1}, \ldots A_j$ (tout sous arbre d'un ABR contient forcément des éléments consécutifs). Si $T'$ n'était pas optimal, on pourrait le remplacer par l'ABR optimal construit sur les, mêmes éléments. Le coût de recherche dans $T$ serait amélioré, ce qui contredirait notre hypothèse : $T$ optimal.

En conséquence, nous allons adopter une approche **programmation dynamique** pour la construction de l'ABR optimal. Cette approche, sur laquelle on reviendra lors d'un chapitre spécifique, consiste à résoudre des sous problèmes dont on combine peu à peu les résultats pour obtenir la solution globale.

Dans le cas qui nous intéresse, il s'agit de trouver comment l'ABR optimal peut être déduit des ABR optimaux de ses sous arbres gauche et droit. Nous allons établir pour cela une relation de récurrence sur $C(T)$.

Supposons que la racine de $T$ soit $A_k$, et que les deux sous arbres de $T$ soient des ABR optimaux. Notons $C(i,j)$ le coût optimal pour un arbre construit sur les éléments $[A_{i+1}, \ldots A_j]$. On a alors :

$$C(T) = C(i, k-1) + C(k, j) + \sum_{l=i+1}^{j} a_l + \sum_{l=i}^{j} b_l$$

Les deux derniers termes correspondent au surcoût d'une recherche sur $T$ par rapport à une recherche dans l'un de ses sous arbres, la hauteur de chaque élément étant augmentée de 1.

On en déduit la formule de récurrence :

$$C(i,j) = min_{i+1 \leq k \leq j} \left(C(i, k-1) + C(k, j)\right) + \sum_{l=i+1}^{j} a_l + \sum_{l=i}^{j} b_l$$

avec la condition aux limites

$$C(i,i) = 0$$

L'élément $k$ qu'il vaut mieux choisir pour racine à chaque étape de la construction est donc celui qui permet de minimiser $C(i,j)$. Il nous reste à écrire l'algorithme qui calcule ces valeurs.

**Exercice 14.2** *On suppose que les valeurs $\sum_{l=i+1}^{j} a_l + \sum_{l=i}^{j} b_l$ sont déjà calculées et stockées dans un tableau $W(i,j)$. Programmer directement le calcul des $C(i,j)$ à partir de l'équation précédente conduirait à de nombreux calculs redondants.*

1. *Proposer une solution simple pour éviter de calculer plusieur fois les mêmes valeurs. Ecrire la procédure correspondante. Evaluer le coût mémoire et le coût en temps.*
2. *Modifier la procédure pour permettre la construction de l'arbre optimal après la phase de calcul des $C(i,j)$.*

Solution : 1. Il s'agit d'utiliser un tableau de marques $C$ de taille $n \times n$ pour stocker les valeurs déja calculées. Ce tableau est initialisé à $-1$, sauf la diagonale principale qui est initialisée à 0. La procédure de calcul est alors :

```
procedure poids-opt(i,j:entier)
  si C(i,j) = -1 alors    -- si la valeur n'a pas encore ete calculee
    min := + INFINI;
    pour k de i+1 a j faire :
      poids-opt(i, k-1);
      poids-opt(k,j);
      si C(i,k-1) + C(k,j) < min alors
        min := C(i, k-1) + C(k, j);
        ---- rac := k ----
    C(i,j) := min + W(i,j);
    ---- r(i,j) := rac; ----
```

Le coût mémoire est $O(n^2)$. Chaque $C(i,j)$ n'est calculé qu'une fois, et le coût d'un calcul de minimum est de l'ordre de $n$. Le coût en temps est donc un $O(n^3)$.

2. Pour la construction de l'arbre, il suffit d'ajouter les lignes entre commentaires ci-dessus. Le tableau $r(i,j)$ donne alors l'indice à choisir pour la racine du sous arbre construit sur $[A_{i+1}A_j]$. L'arbre optimal peut alors être crée en appelant :

```
procedure arbre-opt(i,j: entier; var p:sommet)
  si i=j alors p:= nil;
  sinon
    new p;
    p.val := A_{r(i,j)};
    arbre-opt(i, r(i,j)-1, p.g);
    arbre-opt(r(i,j), j, p.d);
```

**Passage à une procédure itérative :** Plutôt que d'utiliser un tableau de marques et une procédure récursive, on peut programmer le calcul des $C(i,j)$ de manière itérative, en choisissant un bon ordre de parcours du tableau de valeurs à calculer. On voit sur la figure 14.3 que cela ne permet malheureusement pas d'économiser de la place mémoire, chaque $C(i,j)$ dépendant d'une ligne et d'une colonne de valeurs.

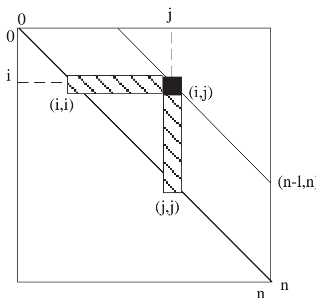


Figure 14.3: Calcul itératif du coût optimal.

La procédure itérative correspondant à cet ordre de calcul s'écrit :

```
procedure poids-opt2 (i,j:entier)
  pour l de 1 a n faire      -- numero de la diagonale
    pour i de 0 a n-l faire  -- chaque element de la diagonale
      j:= i+l; min:=C(i+1, j); rac:=i+1;
      pour k de i+2 a j faire
        si min > C(i, k-1) + C(k,j) alors
          min := C(i, k-1) + C(k,j);
          rac:= k;
      C(i,j) := min + W(i,j);
      r(i,j) := rac;
```



Le temps de calcul est du même ordre que celui de la version récursive. Plus précisément, il est proportionnel au nombre de passages dans la boucle la plus interne :

$$\begin{aligned}\sum_{l=1}^{n}\sum_{i=0}^{n-l}\sum_{k=i+2}^{i+l} 1 &= \sum_{l=1}^{n}\sum_{i=0}^{n-l}(i+l-i-1)\\ &= \sum_{l=1}^{n}(n-l+1)(l-1)\\ &= \sum_{k=1}^{n-1}(n-k+2)k = (n+2)\sum_{k=1}^{n-1}k - \sum_{k=1}^{n-1}k^2\\ &= (n+2)\frac{n(n-1)}{2} - \frac{n(n-1)(2n-1)}{6}\end{aligned}$$

Le coût est donc bien un $O(n^3)$.

**Exercice 14.3** *Définir l'ordre de calcul des $C(i,j)$ par la procédure récursive poids-opt. Ecrire une procédure itérative évaluant les $C(i,j)$ dans le même ordre que la procédure récursive.*

**Réduction du coût de l'algorithme :** La procédure itérative que nous venons de donner peut être améliorée en remarquant que les racines $r(i,j)$ vérifient :

$$r(i,j-1) \leq r(i,j) \leq r(i+1,j)$$

On peut alors réduire la longueur des boucles dans `poids-opt2` en écrivant :

```
min:=C(r(i,j-1), j) + C(i, r(i,j-1) -1);
rac:=r(i,j-1);
pour k de (r(i,j-1)+1) a r(i+1,j) faire
 ...
```

Cela permet de gagner un ordre de grandeur sur le coût, et de rendre ainsi le processus d'initialisation d'un dictionnaire plus utilisable :

$$\begin{aligned}\sum_{l=2}^{n}\sum_{i=0}^{n-l}\sum_{k=r(i,i+l-1)+1}^{r(i+1,i+l)} 1 & \\ &= \sum_{l=2}^{n}\sum_{i=0}^{n-l}(r(i+1,i+l)-r(i,i+l-1)) \\ &= \left(\sum_{l=2}^{n} r(n-l+1,n)-r(0,l-1)\right) \\ &\leq \sum_{l=2}^{n}(n-1)\end{aligned}$$

L'initialisation du dictionnaire se fait donc en $O(n^2)$.

## 14.3 Sous-suites ascendantes (sujet d'examen corrigé)

On considère un tableau $T(1..n)$ dont les éléments appartiennent à un ensemble muni d'une relation d'ordre notée $\leq$. Le tableau $T$ n'est pas trié. On appelle sous-suite d'éléments de $T$ toute suite $T(j_1), T(j_2), \ldots, T(j_k)$ avec $1 \leq j_1 < j_2 \ldots < j_k \leq n$. Une sous-suite $T(j_1), T(j_2), \ldots, T(j_k)$ est dite ascendante si $T(j_1) \leq T(j_2) \ldots \leq T(j_k)$. Le problème à résoudre est le suivant : étant donné $T$, calculer une sous-suite ascendante de $T$ de longueur maximale. On traite d'abord le problème du calcul de la longueur maximale d'une sous-suite ascendante de $T$.

Soit la fonction `LSSA(m,i)` définie par :

```
si i > n alors 0
si i <= n  alors
         si T(i) < m alors LSSA(m,i+1)
         si T(i) >= m alors Max (LSSA(m,i+1), LSSA(T(i),i+1) + 1)
```

1. Prouver que `LSSA(m,i)` est égal à la longueur maximale d'une sous-suite ascendante de $T(i..n)$ dont tous les éléments sont supérieurs ou égaux à $m$.

   La longueur maximale d'une sous-suite ascendante de $T$ est alors égale à `LSSA(`$-\infty$`,1)`, où $-\infty$ note un objet strictement inférieur à tout élément du tableau $T$.

2. Procédure récursive naïve :

   (a) Ecrire une procédure récursive avec un paramètre variable, utilisant les variable globales $T$ et $n$ , calculant la fonction `LSSA` selon la définition donnée ci-dessus.

   (b) Montrer que cette procédure peut exécuter des calculs redondants.

   (c) Evaluer, dans le cas le plus favorable et dans le cas le plus défavorable, le temps d'exécution et la taille de la mémoire nécessaire pour calculer la longueur maximale d'une sous-suite ascendante du tableau $T(1..n)$. Préciser pour quelles valeurs du tableau T se réalisent les cas les plus favorables et défavorables.



3. Procédure récursive avec marquage :

   (a) Modifier la procédure de la question 2 pour obtenir une procédure récursive n'exécutant pas de calculs redondants.

   (b) Quel sont alors le temps d'exécution et la taille de la mémoire nécessaire pour calculer la longueur maximale d'une sous-suite ascendante de $T(1..n)$, dans le cas le plus favorable et dans le cas le plus défavorable ?

4. Calcul des sous suites ascendantes :

   (a) Modifier la procédure de la question 2 de façon à calculer une sous-suite ascendante de longueur maximale.

   (b) Modifier cette procédure de façon à calculer toutes les sous-suites ascendantes de longueur maximale.

5. Ecrire une procédure itérative calculant la fonction `LSSA`. Justifier votre procédure. Quels sont les coûts en temps et place mémoire ?

**Corrigé :**

1. On note *long(m,i)* la longueur maximale d'une sous-suite ascendante de $T(i..n)$ dont tous les éléments sont supérieurs ou égaux à $m$. On prouve que *LSSA(m,i)* = *long(m,i)* par récurrence sur $n-i$.

- Si $i > n \quad long(m,i) = 0 = LSSA(m,i)$.
- Si $i \leq n$ on distingue deux cas.

  1. Si $T(i) < m$, $T(i)$ n'appartient à aucune sous-suite dont tous les éléments sont supérieurs ou égaux à $m$. Donc

     $$long(m,i) = long(m,i+1)$$

     Or $LSSA(m,i+1) = long(m,i+1)$ par hypothèse de récurrence.

  2. Si $T(i) \geq m$, l'ensemble des sous-suites ascendantes de $T(i..n)$ dont tous les éléments sont supérieurs ou égaux à $m$ est l'union des deux ensembles disjoints suivants :

     A, l'ensemble des sous-suites contenant $T(i)$ : elles sont composées de T(i) et d'une sous-suite ascendante de $T(i+1..n)$, dont tous les éléments sont supérieurs ou égaux à $T(i)$.

     B, l'ensemble des sous-suites ne contenant pas $T(i)$ : ce sont des sous-suites ascendantes de $T(i+1..n)$, dont tous les éléments sont supérieurs ou égaux à $m$.

     La longueur maximale d'une sous-suite de A est $1+long(T(i),i+1)$, qui est égal, par hypothèse de récurrence, à $1+LSSA(T(i),i+1)$. La longueur maximale d'une sous-suite de B est $long(m,i+1)$, qui est égal, par hypothèse de récurrence, à $LSSA(m,i+1)$. Par conséquent, la longueur maximale d'une sous-suite ascendante de $T(i..n)$ dont tous les éléments sont supérieurs ou égaux à $m$, est le maximum de ces deux valeurs :

     $$long(m,i) = Max(1+LSSA(T(i),i+1), LSSA(m,i+1))$$

On en conclut que, dans tous les cas, on a $LSSA(m,i) = long(m,i)$.

2. Procédure récursive naïve.
(a) De la définition de la fonction `LSSA` on déduit la procédure suivante :

```
procedure LSSA(m, i : entier ; var r : entier) ;
  var r1, r2 : entier ;
  début
    si  i > n  alors r := 0
    sinon
      si  T(i) < m  alors  LSSA(m, i+1, r)
      sinon debut
        LSSA(m, i+1, r1) ;
        LSSA(T(i), i+1, r2) ;
        r := Max(r1, r2 + 1)
      fin
  fin
```

b. Soit $n = 2$, $T(1) = 1$, $T(2) = 2$. Comme on peut le vérifier sur l'arbre des appels de la figure 14.4, l'appel `LSSA(-∞,1,)` provoque deux appels `LSSA(2,3,)`.

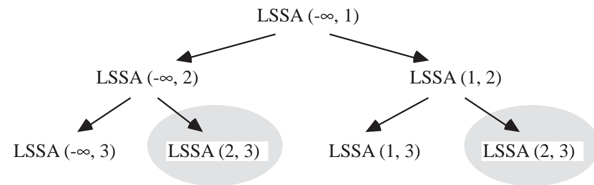


Figure 14.4: Arbre des appels de LSSA

(c) Temps d'exécution : Le corps de la procédure ne comportant pas de boucle, son temps d'exécution (appels récursifs exclus) est borné par une constante. Le temps d'exécution d'un appel de la procédure est donc du même ordre que le nombre d'appels récursifs provoqués (le nombre de sommets de l'arbre des appels). Dans tous les cas où $i \leq n$, un appel de `LSSA(m, i, )` provoque un appel de `LSSA(m,i+1,)`. Si, de plus, on a $m = -\infty$, alors $T(i) > m$, et il y a deux appels récursifs. Ainsi, les cas favorables sont les cas (s'ils existent) tels que, chaque fois que $m \neq -\infty$, on a un seul appel récursif, c'est-à-dire $T(i) < m$. Les cas défavorables, ceux (s'ils existent) où l'on a chaque fois deux appels récursifs, c'est-à-dire $T(i) \geq m$.

*Cas le plus favorable :*
Si le tableau $T$ est trié dans l'ordre décroissant, ses éléments étant tous différents, on a toujours $T(i) < m$, si $m \neq -\infty$. Cela conduit à l'arbre des appels de la figure 14.5, où les embranchements se trouvent seulement sur la branche de gauche.

Le nombre de sommets de cet arbre, à la profondeur $k$, est $k+1$. Le nombre total de sommets est donc :

$$\sum_{k=0}^{n}(k+1) = \frac{(n+1)(n+2)}{2}$$



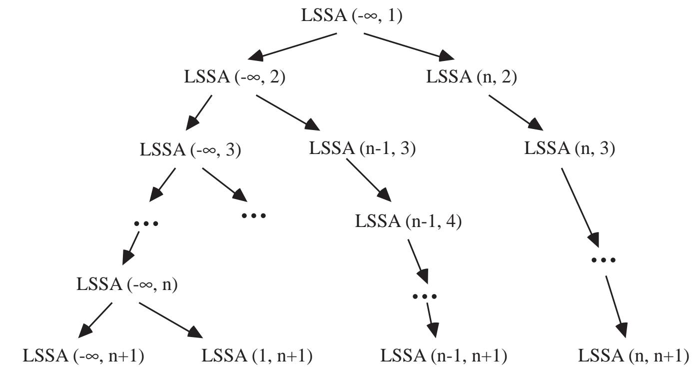


Figure 14.5: Arbre des appels de LSSA si le tableau est trié en ordre décroissant.

**Remarque 15** *On peut obtenir aussi ce résultat de la manière suivante. Soit $N1(i)$ le nombre d'appels récursifs provoqués par un appel $LSSA(-\infty, i)$ et $N2(i)$ le nombre d'appels récursifs provoqués par un appel $LSSA(m,i)$, avec $m \neq -\infty$. On a :*

$$\begin{aligned} N_1(i) &= N_1(i+1) + N_2(i+1) + 2 \\ N_2(i) &= N_2(i+1) + 1 \\ N_1(n+1) &= N_2(n+1) = 0. \end{aligned}$$

*Donc $N_2(i) = n - i + 1$ et $N_1(i) = (n-i+2)(n-i+3)/2 - 1$. Le nombre total d'appels récursifs provoqués par un appel `LSSA( - ∞,1)` est $N_1(1)$, soit $(n+1)(n+2)/2 - 1$.*

Le temps d'exécution est donc de l'ordre de $n^2$ dans les cas favorables. La hauteur de l'arbre est $n$. La taille de la mémoire est donc de l'ordre de $n$.

*Cas le plus défavorable :*
$T$ est un tableau trié en ordre croissant. Le nombre de sommets de l'arbre des appels de `LSSA(-∞,1)` est : $1 + 2 + 2^2 + +2^n = 2^{n+1} - 1$. Le temps d'exécution est donc de l'ordre de $2^n$. La hauteur de l'arbre est $n$. La taille de la mémoire est donc de l'ordre de $n$.

3. Procédure récursive avec marquage.
(a) A l'exception de l'appel principal, tous les appels sont de la forme `LSSA(T(j),i)`.Il suffit donc d'un tableau d'entiers $L(1..n, 1..n)$, initialisé à -1, pour mémoriser les résultats intermédiaires. On aura alors :

$$L(j,i) \neq -1 \;\Rightarrow\; L(j,i) = LSSA(T(j), i)$$

Pour éviter les cas particuliers, on utilise un tableau d'entiers $L(0..n, 1..n+1)$ et on pose $T(0) = -\infty$.

```
procedure LSSA(j,i : entier) ;
si L(j,i) <> -1 alors
debut
  si i > n alors L(j,n+1) := 0
```

```
  sinon
    si T(i) < T(j) alors
      debut LSSA(j,i+1) ; L(j,i) := L(j,i+1) fin
      sinon debut
        LSSA(j,i+1) ;
        LSSA(i,i+1) ;
        L(j,i) := Max(L(j,i+1), L(i,i+1) + 1)
       fin
fin

Appel principal : LSSA(0,1);     -- Le resultat est dans L(0,1)
```

(b) Le temps d'exécution et la mémoire sont de l'ordre de $n^2$ dans tous les cas.

4. Calcul des sous-suites ascendantes.
(a) Les sous-suites sont représentés avec des enregistrements définis par :

```
enregistrement elem (indice:entier; suiv:^elem)

procedure LSSA (m,i : entier ; var r : entier ; var tete : ^elem) ;
var r1, r2 : entier ; var p1, p2 : ^elem ;
debut
  si i > n alors debut r := 0 ; tete := nil fin
  sinon si T(i) < m alors LSSA (m, i+1, r, tete)
    sinon debut
        LSSA(m,i+1,r1,p1) ; LSSA(T(i),i+1,r2,p2) ;
        si r1 < r2+1 alors debut
           r := r2 + 1 ;
           nouveau(tete) ;
           tete^.indice := i ;
           tete^.suiv := p2
        fin
        sinon debut r := r1 ; tete := p1 fin
    fin
fin
```

(b) Pour représenter un ensemble de sous-suites on modifie la définition des enregistrements élém :

```
enregistrement elem (indice : entier ; alt : ^elem ; suiv : ^elem)

procedure LSSA (m,i : entier ; var r : entier  ; var tete : ^alt) ;
var r1, r2 : entier ; var p1, p2 : ^alt ;
debut
  si i > n alors debut r := 0 ; tete := nil fin
  sinon si T(i) < m alors LSSA (m, i+1, r, tete)
    sinon debut
      LSSA(m,i+1,r1,p1) ; LSSA(T(i),i+1,r2,p2) ;
      si r1 <= r2+1 alors
        debut
          r := r2 + 1 ;  nouveau(t\^ete) ;
          t\^ete^.indice := i ; tete^.suiv := p2 ;
          si r1 = r2+1 alors tete^.alt := p1 sinon t\^ete^.alt := nil
        fin
```



```
      sinon debut r := r1 ; tete := p1 fin
    fin
fin
```

Il est plus intéressant de partir de la procédure sans calculs redondants. Il faut alors utiliser un tableau $S(0..n, 1..n+1)$ de pointeurs vers des `elem` pour mémoriser les sous-suites partielles.

5. Procédure itérative.

```
pour j := 0 jusqu'a n faire L(j,n+1) := 0 ;
pour i := n pas -1 jusqu'a 1 faire
    pour j := 0 jusqu'a i - 1 faire
        si T(i) < T(j) alors L(j,i) := L(j,i+1)
        sinon L(j,i) := Max(L(j,i), L(i,i+1) + 1)
```

On modifiera ce programme pour n'utiliser qu'une taille mémoire de l'ordre de $n$, en ne conservant que deux colonnes.

## 14.4 Produit de $n$ matrices

**Exercice 14.4** *1. On considère quatre matrices d'entiers $M_1, , M_4$, de dimensions respectives $100 \times 1$, $1 \times 50$, $50 \times 20$ et $20 \times 1$.*

*(a) Le produit de matrices étant associatif, de combien de façons peut-on associer les opérateurs $\times$ pour évaluer le produit $M_1 \times \ldots \times M_4$ ?*
*Soit $M$ et $N$ deux matrices $n \times p$ et $p \times q$. On définit le coût du calcul du produit $M \times N$ comme étant $npq$.*

*(b) Quel est le coût minimal du calcul de $M_1 \times \ldots \times M_4$ ?*

*2. Soit $M1, \ldots, Mn$ $n$ matrices de dimensions respectives $d_1 \times d_2, d_2 \times d_3, \ldots, d_n \times d_{n+1}$. Soit $c(i,j)$ la fonction définie par :*

$$
\begin{aligned}
c(i,i) &= 0 \\
c(i,j) &= min_{i<k\leq j}\ (c(i,k-1) + c(k,j) + d_i d_k d_{j+1}) \quad \text{pour } i < j
\end{aligned}
$$

*Montrer que $c(1,n)$ est le coût minimal pour calculer $M_1 \times \ldots \times M_n$.*

*3. Ecrire une procédure récursive calculant $c(i,j)$ sans exécuter de calcul redondant.*

*4. Quels sont le temps et la taille de la mémoire nécessaires pour exécuter la procédure de la question 3 ?*

*5. (a) Transformer la procédure de la question 3 en une procédure sans paramètre ni variable locale.*

*(b) Ecrire l'organigramme d'une procédure itérative équivalente.*

6. *On considère l'ensemble des valeurs $c(i,j)$ nécessaires pour évaluer $c(1,n)$ en utilisant la définition de la question 2. Définir un ordre de calcul de ces valeurs ne nécessitant pas l'utilisation d'une pile. Ecrire la procédure itérative correspondante de calcul de $c(1,n)$.*

7. *Modifier la procédure de 3 ou celle de 6 pour calculer une suite de produit de matrices correspondant à une évaluation de coût minimal de $M_1 \times \times M_n$.*

## 14.5 Plus longue sous-séquence commune

**Exercice 14.5** *On appelle sous-séquence d'une chaîne $x$ sur un vocabulaire $V$ toute chaîne $a1 \dots ak$ telle que $x = x_1 a_1 x_2 a_2 \dots x_k a_k x_{k+1}$ avec $x1, x_2, , x_{k+1} \in V^*$, et $a_1, a_2, \dots, a_k \in V$. Soit deux chaînes $x = a_1 a_2 \dots a_n$ et $y = b_1 b_2 b_m$, où les $a_i$ et les bi sont des éléments du vocabulaire $V$ et la fonction $f$ est définie comme suit :*

```
f(j, k) = si j > n ou k > m alors 0
          sinon si aj = bk alors 1 + f(j + 1, k + 1)
                sinon Max {f(j, k + 1), f(j + 1, k)}
```

1. *Quelle est la valeur de $f(j,k)$ ? De $f(1,1)$ ?*

2. *Déduire de la définition de $f$ une procédure récursive calculant, sans appels récursifs redondants, une plus longue sous-séquence commune à deux chaînes données.*

3. *Evaluer le temps d'exécution et la taille de la mémoire nécessaires pour exécuter votre procédure. Quel serait le temps d'exécution dans le pire cas d'une procédure exécutant des appels récursifs redondants ?*

4. *Peut-on améliorer le temps d'exécution de votre procédure en utilisant une technique de "branch and bound" ?*

5. *Modifier votre procédure pour évaluer toutes les sous-séquences communes à deux chaînes données, de longueur inférieure ou égale à une constante c donnée.*

## 14.6 Modélisation et programmation dynamique

### 14.6.1 Problèmes des cageots de fraises

**Exercice 14.6** *$n$ cageots de fraises doivent être distribués dans $k$ succursales ; connaissant les bénéfices que l'on peut retirer de chaque succursale en fonction du nombre de cageots fourni (pas linéaire) on veut trouver la répartition qui optimise notre gain.*

**Remarque 16** *Si les rapports marginaux sont décroissants, un algorithme glouton donne toujours la solution optimale.*

### 14.6.2 Sac à dos

Il s'agit de résoudre le problème du sac à dos par une méthode style "programmation dynamique".



**Exercice 14.7** *Soit un sac de volume $V$, et $n$ articles différents dont on dispose à volonté. On note $a_k$ le bénéfice du k-ième article, et $v_k$ le volume qu'il occupe. Proposer et comparer deux méthodes de calcul du bénéfice maximal, basées respectivement sur une récurrence sur le nombre d'articles utilisés et sur une récurrence sur le volume du sac.*

### 14.6.3 Jeu télévisé

**Exercice 14.8** *Dans un jeu télévisé après avoir répondu à un certain nombre de questions, le montant du gain va être décidé de la façon suivante : On va vous fournir à la suite 10 nombres tirés au hasard de façon uniforme entre 0 et 1. A chaque nombre vous avez soit la possibilité de dire STOP, et le gain sera de 1 000 000 de fois le nombre retenu, soit de continuer mais le nombre est définitivement perdu. Si vous arrivez au 10 ème vous devrez garder ce nombre. On cherche une stratégie de jeu qui maximise l'espérance de gain.*

1. *Comment se présente une telle stratégie ?*

2. *Montrer que vous connaissez un élément de cette stratégie*

3. *Par la programmation dynamique trouver la stratégie optimale.*

### 14.6.4 Fonction d'Ackermann

```
Ackermann (m, n : entier naturel) =
  si m=0 alors  n+1
  sinon si n=0 alors   Ackermann(m-1, 1)
        sinon   Ackermann(m-1, Ackermann(m, n-1))
```

**Exercice 14.9** *Donner un algorithme de calcul de* `Ackermann(m, n)` *utilisant une taille mémoire de l'ordre de $m$. Utiliser deux tableaux* `val, ind [0..m]` *tels que :*
`val(i) = Ackermann(i, ind(i))`.

## 14.7 Programmation du parcours d'un chariot automatisé

Dans une grande agglomération les diverses pharmacies passent plusieurs fois (en général 2) par jour des commandes à un grossiste. Celui-ci a tous les médicaments rangés dans un entrepôt automatisé. Les médicaments sont rangés dans des tiroirs superposés et alignés de part et d'autre d'allées parallèles (appelées par la suite allées de rangement). On ne peut passer d'une allée à une autre qu'en empruntant une des 2 allées transversales situées aux extrémités de celles-ci. La distance entre 2 allées de rangement consécutives, le long d'une allée transversale, est en générale la même (mais peut ê tre différente à chaque extrémité), mais ce n'est pas nécessairement le cas pour des entrepôts autres que ceux relatifs à des petits objets (les zones de rangement peuvent différer de largeur en fonction de la taille des objets à entreposer). L'entrepôt a donc l'une des formes illustrées par la figure 14.6 ou toute autre forme qui respecte les conditions sur le passage d'une allée à l'autre (pouvez vous en dessiner d'autres ?).

Un chariot automatisé peut parcourir l'ensemble des allées, s'arrêter automatiquement devant les tiroirs concernés, un mécanisme poussant dans celui-ci le nombre de boîtes nécessaires à la satisfaction de la commande. Une fois la commande remplie, le chariot

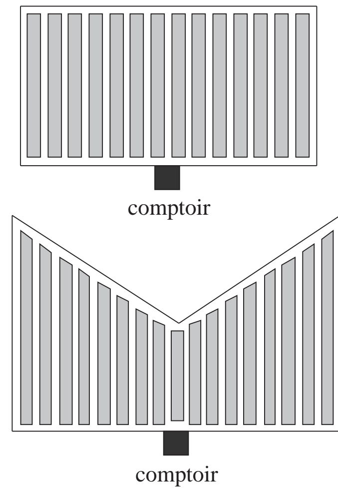


Figure 14.6: Exemples de configurations d'entrepots

revient au comptoir. On suppose la capacité du chariot telle qu'il puisse toujours prendre en totalité une commande avant de revenir au comptoir.

La question est le parcours du chariot. Une possibilité est de lui faire parcourir systématiquement chaque allée pour collecter chaque commande. Sachant que le nombre de commandes traitées par jour est très grand, on pourrait se trouver limités par les temps de parcours du chariot.

La question est donc de savoir si on peut optimiser le parcours du chariot. Le temps de calcul de cette optimisation doit être inférieure au temps qu'il faut au chariot pour ramasser une commande et ceci sur un micro-ordinateur de relativement faible puissance. Si c'est possible, pendant la collecte d'une commande, les calculs seront faits pour la suivante.

Ce problème peut se résoudre par programmation dynamique.

### 14.7.1 Modélisation

On va représenter le problème par un *parcours de graphe*. Les sommets du graphe représentent les intersections d'allées (sommets "blancs") et les point de ramassage d'objets (sommets "noir") (cf figure 14.7). Pour que toutes les allées soient similaires on choisit de représenter le comptoir par un sommet "noir" sur une allée et à distance 0 de l'extrémité de l'allée.

Le problème est donc en partant du comptoir visiter au moins une fois tous les sommets "noirs" et revenir au comptoir, et ceci en parcourant le moins de distance possible. Pour cela on sera amené à visiter certains sommets "blancs". Une solution est donnée par la figure 14.8. En fait cette figure nous dit pour chaque arête du graphe combien de fois (0-1-2) elle est utilisée dans le parcours. Il y a plusieurs parcours possibles sur ces arêtes, tous de même longueur, on y reviendra plus tard. Le problème est donc de chercher une famille d'arêtes (dans une famille le même élément peut être répété plusieurs fois) telle que chaque sommet noir soit incident à un nombre pair d'arêtes et que le graphe partiel sur ces



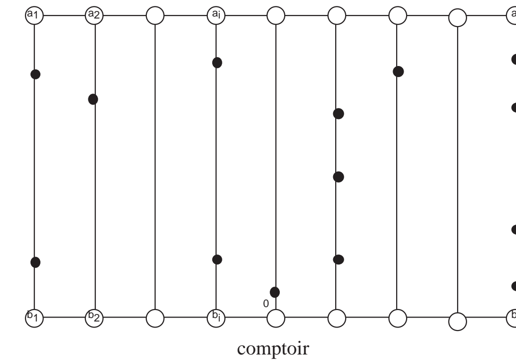


Figure 14.7: Représentation de l'entrepot et des produits à ramasser sous forme d'un graphe bicolore.

arêtes soit connexe, le tout de longueur minimum. Tout parcours du chariot correspond à un tel ensemble d'arêtes et réciproquement à un tel ensemble d'arêtes on peut associer un parcours (en général pas unique) du chariot puisque le graphe partiel sur ces arêtes est eulérien (cf cours de graphe de première année).

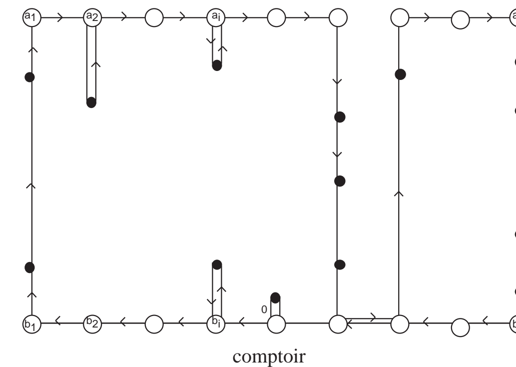


Figure 14.8: Un parcours de l'entrepot.

### 14.7.2 Résolution par la programmation dynamique

**Notations:** On notera $a_1, a_2, \ldots, a_i, \ldots, a_n$ et $b_1, b_2, \ldots, b_i, \ldots, b_n$ les sommets qui représentent les extrémités des allées de rangements avec les allées transversales haut et bas respectivement (l'ensemble des sommets blancs). On notera $L_{+i}$ (resp. $L_{-i}$) les arêtes du graphe qui sont à gauche de l'allée $i$ y compris l'allée $i$ (resp.sans l'allée $i$). On notera $L_i$ indifféremment $L_{+i}$ et $L_{-i}$.

a) Les arêtes d'une tournée qui sont dans $L_i$ (c. à d. les familles d'arêtes de $L_i$ qui peuvent se compléter en une tournée par des arêtes du complémentaire de $L_i$) peuvent se caractériser

par un vecteur à 3 composantes : (parité de $a_i$, parité de $b_i$, nombre de composantes connexes). Montrer que si on différencie 3 possibilités pour la parité : 0, Pair et Impair, alors il n'y a que 7 possibilités pour le vecteur ci dessus. Les lister. Considérons toutes les familles d'arêtes de $L_i$ qui peuvent se compléter en une tournée par des arêtes du complémentaire de $L_i$, et définissons sur cet ensemble la relation d'équivalence suivante : 2 familles sont équivalentes si et seulement si on peut les compléter en une tournée par les même familles d'arêtes du complémentaire de $L_i$. Montrer que les classes d'équivalence sont données par les 7 vecteurs ci-dessus.

**Remarque 17** *Si on prend la précaution de limiter le problème aux allées comprises entre la plus à gauche et celle la plus a droite ayant quelque chose a y être ramassé, on n'a que 5 classes à considérer.*

b) Montrer que seul le meilleur des éléments de chacune des sept classes ci-dessus peut appartenir à une solution optimale. Si $i = n$, montrer que dans $L + n$ le choix est plus limité et donne la solution optimale recherchée.

c) Montrer que l'on peut calculer le meilleur de chaque classe dans $L + 1$.

d) Supposant connu le meilleur élément de chaque classe de $L + i$, montrer que l'on peut faire de même pour chaque classe de $L - i + 1$

e) Supposant connu le meilleur élément de chaque classe de $L - i + 1$ montrer que l'on peut faire de même pour chaque classe de $L + i + 1$.

f) décrire l'algorithme de résolution. N'oubliez pas qu'il faut pouvoir retrouver la solution à la fin des calculs.

### 14.7.3 Parcours du chariot

Le parcours d'un graphe eulérien peut s'obtenir à partir de plusieurs algorithmes, les plus simples étant souvent très difficiles à implémenter, par exemple un algorithme peut s'énoncer: "parcourir le graphe en effaçant les arêtes, ne prendre un isthme que s'il n'y a pas d'autre choix". La difficulté est de tester une arête pour savoir si c'est un isthme. L'algorithme classique est un algorithme assez complexe de marquage des sommets avec fusion de cycles. Dans le cas présent, à cause de la structure du graphe l'algorithme suivant est correct : "parcourir le graphe en effaçant les arêtes, si on a le choix entre une arête simple et une double, toujours prendre la première occurrence de la double, ne prendre la deuxième occurence d'une arête double que s'il n'y a pas d'autre choix".

### 14.7.4 Remarques sur l'efficacité de cet algorithme

Montrer que ce n'est qu'un plus court chemin sur un graphe à $7n$ sommets (ou $5n$). Quel est un majorant du nombre de ses arêtes? Il faut ajouter le calcul des longueurs. Quelle est sa complexité ? La pratique, qui corrobore l'étude théorique, montre que l'algo est très rapide et le temps dépend plus du nombre d'allées que du nombre de produits à ramasser.

# Chapitre 15

# Résolution par séparation et évaluation

Ce chapitre explore une méthode classique pour rechercher un élément optimum dans un ensemble combinatoire de solutions possibles. D'abord on présente le principe d'exploration de l'ensemble des solutions par *séparation*, puis on aborde l'exploration de ces solutions avec un principe de majoration qui évite de considérer toutes les solutions possibles et rendant ainsi (parfois) l'exploration possible pour des problèmes de forte combinatoire.

## 15.1 Arborescence des solutions possibles

Soit $E$ un ensemble discret que l'on supposera fini et très grand. On peut essayer d'énumérer tous les éléments de $E$ en séparant l'ensemble en $n_1$ sous-ensembles non vides contenant chacun une partie, de taille variable, des éléments de $E$ (on n'impose pas toujours que ces sous ensembles soient disjoints, mais c'est en général le cas). On peut recommencer avec chaque sous ensemble qui contient plus d'un élément et ainsi de suite jusqu'à ce que tous les ensembles ne contiennent plus qu'un seul élément. Cette énumération peut se représenter par un arbre de la façon suivante (cf. figure 15.1) : la racine de l'arbre représente $E$, ses $n_1$ fils représentent les $n_1$ sous ensembles créés dans la première partition de $E$, et ainsi de suite.

Les feuilles de l'arbre représentent alors les divers éléments. Si on a véritablement effectué une partition des ensembles à chaque fois, tous les éléments ne sont représentés qu'une fois. S'il y a eu séparation en sous ensembles non nécessairement disjoints on obtient des répétitions d'éléments. Il y a bien entendu un très grand nombre de façons différentes de procéder pour un ensemble donné.

**Exemple 1 :** Soit $E$ l'ensemble des $n!$ permutations de $n$ éléments. La façon la plus classique consiste à d'abord diviser sur le premier (resp. dernier) élément de la permutation, puis sur le suivant (resp. précédent) et ainsi de suite. La figure 15.2 illustre un tel arbre dans le cas où $n = 4$ et où l'on divise sur le premier élément.

On aurait pu aussi séparer l'ensemble des permutations en deux sous-ensembles, celles dont le premier élément est entre 1 et $k$ et celles dont cet élément est entre $k+1$ et $n$ (en général on prend $k = n/2$), chaque sous ensemble étant ensuite divisé en deux de la même manière, et ainsi de suite. Ceci a l'inconvénient de créer des arbres profonds.





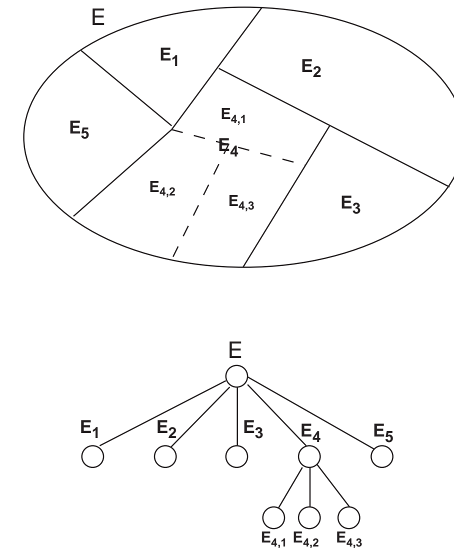


Figure 15.1: Représentation par un arbre d'une énumération totale des éléments d'un ensemble.

**Exemple 2 :** Problème du sac à dos. Le problème du sac à dos consiste à remplir au mieux un sac à dos de capacité $C$ avec des produits $P_1, P_2, \ldots, P_n$, qui prennent une place $v_1, v_2, ...., v_n$ et rapportent $a_1, a_2, ...., a_n$ par unité. On suppose que l'on a autant d'unités de chaque produit que l'on veut. Borner le nombre de chaque produit ne peut que simplifier l'énumération, le cas où l'on n'a qu'une unité de chaque produit s'appelle le problème du sac à dos (0,1).

Ce problème (le plus général) peut s'écrire comme un programme linéaire en nombres entiers : notons $x_i$ le nombre d'unités du produit $P_i$ que l'on choisit de prendre. Il s'agit de calculer:

$$\max_{\{x_i\}} \sum_{i=1}^{n} a_i x_i$$

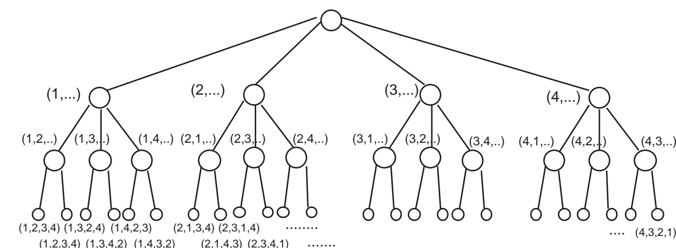


Figure 15.2: Arbre énumérant l'ensemble des permutations de quatre éléments.

avec $x_i \geq 0$ et sous la contrainte:

$$\sum_{i=1}^{n} v_i x_i \leq C$$

**Exercice 15.1** *Montrer que l'on peut supposer tous les $a_i$ différents. De même pour les $v_i$.*

Prenons l'exemple numérique suivant avec 4 produits :

$$max(4x_1 + 5x_2 + 6x_3 + 2x_4)$$

sous les contraintes

$$33x_1 + 49x_2 + 60x_3 + 32x_4 \leq 130$$

$$x_1, x_2, x_3, x_4 \geq 0 \text{ et entiers}$$

L'ensemble de toutes les solutions maximales, c'est à dire celles auxquelles on ne peut pas rajouter de produit, peut se décrire par l'arbre suivant de la figure 15.3. On verra plus loin dans ce chapitre que lors de l'énumération des solutions, on a intérêt à classer les objets par intérêt décroissant. L'ordre des variables dans l'exemple numérique tient compte de cette remarque, le produit le plus intéressant étant, dans ce cas du sac à dos, celui qui rapporte le plus par unité de volume.

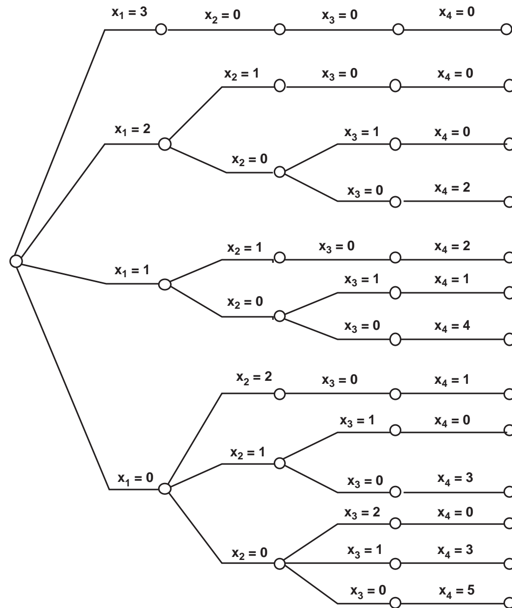


Figure 15.3: Solutions maximales pour l'exemple du sac à dos.



## 15.2 La méthode du "Branch and Bound"

On se propose de résoudre le problème suivant : Trouver dans un ensemble $E$ donné un élément de valeur optimale. Pour fixer les idées nous supposerons dans ce qui suit que l'on cherche un élément de valeur maximum. Le cas "minimum" s'en déduit aisément.

**Remarque 18** *Comme dans tous les problèmes d'optimisation on cherche un élément de valeur optimale et non tous, pour la bonne raison qu'il se pourrait qu'il y ait un nombre considérable de tels élément et leur écriture nécessiterait à elle seule un temps considérable. Par exemple dans certains problèmes d'ordonnancement de tâches, le temps d'achèvement de l'ensemble des $n$ tâches peut être indépendant de l'ordre de passage de celles-ci, et par conséquent les $n!$ solutions ont la même valeur. Même pour $n$ de valeur très modeste, l'énumération des $n!$ permutations serait fastidieuse.*

### 15.2.1 Principe de la méthode

La dénomination *séparation et évaluation* (Branch and Bound) recouvre deux concepts :

- Branch : qui consiste à séparer un ensemble de solutions en sous-ensembles.
- Bound: qui consiste à évaluer les solutions d'un sous-ensemble par une évaluation optimiste, c.à.d. qui majore la valeur de la meilleure solution de ce sous-ensemble.

Pour effectuer l'opération de division il suffit de dire comment l'on divise un ensemble de solutions en sous ensembles. En d'autres termes comment on construirait l'arbre permettant éventuellement de lister toutes les solutions. Bien entendu notre but est d'éviter autant que possible de construire en entier cet arbre.

Pour illustrer le concept d'évaluation optimiste (ici une majoration), reprenons l'énumération partielle de $E$ comme décrite dans la figure 15.2. Supposons que l'on ait pu évaluer pour chaque feuille de cet arbre un majorant de la valeur de toutes les solutions qui sont contenues dans le sous ensemble de solutions qu'elle représente. Ce majorant s'appelle l'évaluation du nœud de l'arbre. Alors il est inutile de diviser un ensemble (nœud) dans les cas suivants :

1. L'évaluation a permis de détecter en même temps un élément qui a cette valeur. Cet élément est nécessairement optimum dans ce sous-ensemble de solutions. Si cette solution est la meilleure trouvée jusque là, elle devient, temporairement peut-être, la meilleure solution courante. Ce cas est plutôt rare.
2. L'évaluation est inférieure ou égale à la valeur de la meilleure solution trouvée jusque là. On n'a aucune chance de trouver mieux dans ce sous-ensemble. Ceci peut permettre des gains importants, car on élimine une partie de l'arbre de recherche.
3. le sous ensemble est vide ou réduit à un seul élément.

Dans les cas 1 et 2, on gagne dans l'exploration de l'arbre puisque la branche suivant le nœud considéré n'a pas besoin d'être explorée. On dit que cette branche est élaguée (on parle de pruning en anglais). Les nœuds de l'arbre qu'il est inutile de diviser sont dits élagués (d'autres disent stérilisés, tués...). A noter que dans le cas 1, si la meilleure solution courante a changé, il convient de parcourir tous les nœuds actifs pour voir s'ils le restent.

**Principe de Branch and Bound :** à chaque étape, on prend un nœud actif, on le divise, on évalue ses fils qui deviennent des nœuds actifs et éventuellement on élague certains d'entre eux. L'algorithme s'arrête quand on n'a plus de nœuds actifs. Le nœud initialement actif est la racine de l'arbre de séparation.

### 15.2.2 Choix du nœud actif à diviser

Le choix de ce nœud est crucial au succès de l'algorithme. Plusieurs stratégies s'affrontent :

1. une stratégie de recherche en profondeur: on s'occupe toujours d'un des fils du nœud que l'on vient de diviser. Si aucun de ces nœuds n'est actif on revient en arrière (backtrack) dans l'arbre.
2. une stratégie en largeur: on divise les nœuds dans l'ordre de leur création.
3. stratégie de la meilleure évaluation: on divise le nœud de meilleure évaluation (donc la plus grande pour un problème de maximisation).
4. stratégie mixte : On va en profondeur tant qu'on le peut, mais quand on ne peut plus on saute au nœud de meilleure évaluation. On explorera les fils dans l'ordre d'évaluation : le meilleur d'abord.

La stratégie (2) est abandonnée. La stratégie (1) qui était presque toujours utilisée est maintenant souvent remplacée par la (4). La (3) est utilisée dans certains problèmes seulement.

La recherche en profondeur a plusieurs avantages:

- le nombre de nœuds actifs reste relativement faible et nécessite donc moins de mémoire. Avec la disponibilité actuelle de la mémoire, cet avantage n'est plus aussi crucial ; il faut cependant considérer que des recherches en largeur du type (b) et (c) peuvent conduire à un nombre exponentiel de nœuds actifs.
- l'évaluation d'un fils peut profiter de l'évaluation du père. En d'autres termes, si on vient d'évaluer le père, ce qu'il y a en mémoire permet parfois d'évaluer un fils beaucoup plus rapidement que si on le faisait plus tard. Par exemple supposons que l'on veuille résoudre un programme linéaire en nombres entiers par Branch and Bound. Une manière consiste à évaluer un nœud en résolvant le programme linéaire obtenu en oubliant la condition d'intégralité des variables. On sépare ensuite en choisissant une variable non entière dans la solution optimale du problème résolu, supposons que $x_8 = 5,59$. On va diviser toutes les solutions contenues dans ce nœud en 2, d'un côté celles pour qui $x_8 \leq 5$ et de l'autre celles pour qui $x_8 \geq 6$. Pour résoudre le programme linéaire d'un des fils, on pourra partir de la base obtenue lors de la résolution précédente et par la méthode duale simplexe obtenir très rapidement la solution optimale de ce nouveau programme linéaire.
- c'est profond dans l'arbre que l'on a le plus de chances de tomber sur une solution lors de l'évaluation et donc d'obtenir une valeur qui permet d'élaguer des sommets. Cette dernière raison est moins valable qu'elle ne l'était auparavant, car désormais on fait toujours précéder l'appel de la résolution par Branch and Bound d'une résolution heuristique. Tous les bons code de Branch and Bound permettent aujourd'hui de passer la valeur de la solution trouvée en paramètre.



**Remarque 19** *Normalement tout nœud de l'arbre a au moins 2 fils, mais il arrive que dans la description utilisée certains nœuds n'ont qu'un fils (cf. l'exemple précédent du sac à dos). Si on sait qu'un nœud n'a qu'un fils, il est inutile de l'évaluer, mieux vaut évaluer son fils. Dans une recherche en profondeur, puisque l'on va de toutes façons créer les fils, cela ne coûte rien d'inverser l'ordre. On verra dans le cas du problème du sac à dos que cela peut accélérer l'algorithme.*

En résumé, chaque fois que l'on peut tirer profit de la relation père fils on optera pour la stratégie mixte, sinon mieux vaut utiliser la meilleure évaluation.

Le succès de la méthode dépend essentiellement de l'acuité de l'évaluation. Il faut être conscient que très souvent, faute de bonne évaluation, l'algorithme ne se termine pas dans un temps raisonnable. On peut accélérer en se contentant d'une solution approchée avec garantie de qualité. On peut décider d'élaguer tout nœud dont l'évaluation est inférieure à $(1+\alpha)$ fois la valeur de la meilleure solution courante. Par exemple si $\alpha = 0,05$, alors quand on s'arrêtera la valeur de la solution trouvée sera à moins de 5% de l'optimum. La plupart des codes vous permet de passer cette valeur $\alpha$.

## 15.3 Exemple du problème du sac à dos

Reprenons l'exemple numérique de la section 1 :

$$max(4x_1 + 5x_2 + 6x_3 + 2x_4)$$

sous les contraintes

$$33x_1 + 49x_2 + 60x_3 + 32x_4 \leq 130$$

$$x_1, x_2, x_3, x_4 \geq 0 \text{ et entiers}$$

Si on a fixé un sous-ensemble de variables, un majorant de toutes les solutions restantes s'obtient très facilement : on remplit la place restante avec le produit qui rapporte le plus par unité de volume (on considère donc que c'est un liquide). La valeur trouvée est de toute évidence un majorant. D'autres évaluations plus sophistiquées sont possibles et souhaitables, on peut y réfléchir. Pour l'illustration de cet exemple nous utiliserons celle que l'on vient de décrire en raison de sa très grande simplicité.

La division est celle décrite dans la figure 15.3. Donc on divise E en 4 sous-ensembles. Si on utilise la remarque de la section précédente, le premier sommet évalué est la feuille du haut et c'est une évaluation exacte (correspond à une solution), sa valeur est de 12, ce qui donne aussi la valeur de la meilleure solution courante. Ce nœud est élagué. Le suivant est le nœud $x_1 = 2$, rien d'autre de fixé. Son évaluation est : 8 + 5/49 (130 - 66) = 14.53. Puisque $14,53 \geq 12+1$, on divise ce nœud et on part sur un de ses fils.

Remarquons que le fait que toutes les données sont entières, si on veut faire mieux que 12 on doit faire au moins 13. Du fait que les produits les plus intéressants a priori ont un indice petit, on choisira toujours le fils "haut" d'abord dans la recherche en profondeur.

Le nœud suivant est donc la deuxième feuille en partant du haut. Son évaluation est exacte et vaut 13. La solution correspondante devient la nouvelle meilleure solution courante et la meilleure valeur est désormais 13. On revient en arrière jusqu'au nœud d'évaluation

14,53. On a toujours espoir de trouver une solution de valeur 14 dans ce sous arbre et on part donc sur l'autre fils. L'évaluation de celui-ci est de : $8 + 0 + 6/60(130 - 66) = 14.4$.

On continue donc et on arrive sur la troisième feuille. Son évaluation est exacte et vaut 14. La solution correspondante devient la nouvelle meilleure solution courante et la meilleure valeur est désormais 14. Ceci élague la partie inférieure du sous arbre dont l'évaluation de la racine était de 14,4 puisque l'on n'a aucune chance de trouver mieux que 14 dedans. On backtrack jusqu'à la racine de l'arbre et on regarde son troisième fils. Son évaluation est de $4 + 5/49(130 - 33) = 13,89$, on élague donc tous le sous arbre dont il est la racine. On passe au dernier fils de la racine, son évaluation est de $130 * 5/49 = 13,26$, on l'élague et la recherche est terminée, la valeur de la solution optimale est de 14 et elle consiste à prendre 2 unités du produit 1 et une du produit 3.

La figure 15.4 montre la partie de l'arbre qu'il a fallu développer pour obtenir le résultat, en noir figurent les sommets qu'il a fallu évaluer.

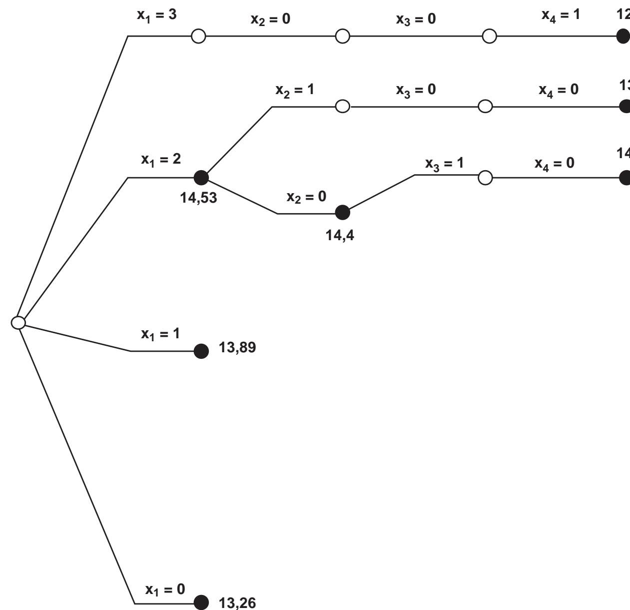


Figure 15.4: Méthode de Branch and Bound appliquée à l'exemple du sac à dos.

S'il y a des bornes aux nombre d'unités de chaque produit que l'on peut transporter la méthode n'est pas affectée (au contraire), il suffit de limiter le nombre de fils de chaque nœud de manière à respecter ces bornes. Par contre dans ce cas la remarque sur le fait que l'on peut supposer tous les profits unitaires et tous les volumes différents ne tient plus.

On verra dans le cours de programmation dynamique d'autres méthodes pour résoudre ce problème. Même si les performances du Branch and Bound sont moins bonnes dans le cas le plus défavorable, c'est cette méthode qui a pour le moment les faveurs en raison de sa performance en moyenne sur les cas pratiques.



## 15.4 Relaxation de programmes linéaires en nombres entiers

Un programme linéaire en nombre entiers, PLE, peut être décrit par :

$$z_{IP} = \max cx : Ax \leq b, x \geq 0 \text{ et à composantes entières}$$

où $A$ est une matrice $m \times n$, $c$ et $b$ des vecteurs ligne et colonne de dimensions adéquates. La *relaxation linéaire*, RL, du problème précédent consiste à oublier la condition d'intégralité sur les variables : $z_{LP} = \max cx : Ax \leq b, x \geq 0$.

**Remarque 20** *On a toujours $z_{IP} \leq z_{LP}$ car toute solution de PLE en est une de RL. Par conséquent si la solution optimale de RL est entière, c'est une solution optimale de PLE.*

L'*évaluation* d'un nœud se fait en résolvant la relaxation linéaire du PLE attaché à celui-ci. La *division* d'un nœud se fait en choisissant une variable $x_i$ qui a une valeur $x_i^*$ non entière dans la solution optimale du programme linéaire que l'on a résolu à ce nœud, l'ensemble des solutions se divise en deux, celles pour lesquelles $x_i \leq \lfloor x_i^* \rfloor$ et celles pour lesquelles $x_i \geq \lfloor x_i^* \rfloor + 1$ ( $\lfloor a \rfloor$ représente le plus grand entier inférieur ou égal à $a$). Les deux PLE associés aux deux fils s'obtiennent donc en modifiant une des bornes d'une variable. Ceci a l'énorme intérêt de ne pas augmenter la taille de la base des programme linéaires relaxés correspondant et donc de ne pas rendre ceux-ci plus difficiles à résoudre (on utilise une version du simplexe dite à variables bornées). D'autre part à partir de la résolution du programme linéaire du père, avec très peu d'efforts, on obtient à l'aide de l'algorithme dual du simplexe la solution optimale d'un des fils. Ce qui milite pour l'emploi de la stratégie mixte décrite plus haut.

Un nœud est élagué dans les cas décrits plus haut à savoir :

- PL irréalisable
- valeur inférieure à la meilleure valeur réalisable trouvée
- solution entière.

Il reste beaucoup à faire pour trouver un bon critère de choix de la variable sur laquelle faire la séparation. Un critère assez grossier est de choisir une variable loin d'être entière, par exemple préférer 8,54 à 4,12 ou à 4,88. Un autre critère est de tenir compte de la hiérarchie des variables. Par exemple une variable 0-1 peut signifier que l'on prend ou non un projet et une autre variable être relative à une tâche de ce projet, on préférera diviser sur la première variable. L'expérience montre que seul un petit nombre de variables a besoin d'être fixé, les autres étant alors automatiquement entières, malheureusement jusqu'à présent on ne sait pas très bien les trouver. Si un PLE possède, comme c'est très souvent le cas, des contraintes du type :

$$\sum_{i \in Q} x_i = 1$$

alors la division précédente serait trop dissymétrique, puisque fixer à 1 une des variables $x_i$ pour $i \in Q$ est un choix fort qui implique que toutes les autres variables soient alors à 0, alors que le choix de 0 pour une de ces variables n'implique rien sur les autres. On préfère

dans ce cas poser $Q = Q_1 \cup Q_2$ avec, avec, si $x^*$ est la solution optimale du PL attaché au nœud considéré,

$$0 < \sum_{i \in Q_1} x_i^* < 1$$

Dans ce cas on divise d'un côté sur les solutions telles que :

$$\sum_{i \in Q_1} x_i^* = 1$$

et de l'autre sur celles telles que :

$$\sum_{i \in Q - Q_1} x_i^* = 1$$

De cette façon il y a plus d'équilibre dans la division des solutions.

Même pour les problèmes de taille moyenne la procédure peut prendre un temps prohibitif. Tout dépend d'une bonne formulation du problème. En gros, une formulation est bonne si les valeurs de $z_{IP}$ et $z_{LP}$ sont assez proches. Comment obtenir une telle formulation, quand c'est possible, dépasse le cadre de ce cours et est l'objet de nombreuses recherches. L'étudiant intéressé pourra lire les premiers chapitres de "Integer and Combinatorial Optimization" de G.L. Nemhauser et L.A. Wolsey.

**Exercice 15.2** *Le problème du sac à dos binaire est celui du sac à dos où chaque objet n'existe qu'en un exemplaire. On se convainc aisément que la stratégie de recherche qui consiste à tenter de remplir le sac avec les objets de valeur volumique la plus grande est une bonne heuristique, et que donc on considérera, comme pour le sac à dos vu dans cette section, les objets triés dans l'ordre décroissant des valeurs volumique.*

*a) Comment étendre la fonction de calcul de l'estimation optimiste lorsqu'un volume $V$ est rempli et que l'on considère les objets de rang supérieur à $i$ ? Écrire cette fonction. Comment la calculer en un temps plus faible que $O(n)$ en considérant un précalcul initial ?*

*b) Écrire l'algorithme de recherche du meilleur chargement et utilisant la fonction heuristique précédente.*

*c) Construire un jeu de données pour lequel l'évaluation heuristique et l'élagage associé ralentit l'algorithme par rapport à une recherche directe. (Pour chaque heuristique, on peut toujours trouver des données qui contrarient l'heuristique ; on espère juste que cette dernière l'accélère le plus souvent, ce qui est le cas pour le problème du sac à dos)*

## 15.5 Un exemple d'ordonnancement de tâches

Le but de cet exemple n'est pas de faire de vous des spécialistes de l'ordonnancement de tâches sur des machines, mais d'illustrer l'effort qui peut être fait pour faire marcher de mieux en mieux la méthode du Branch and Bound. Comme il a été déjà dit, le succès de la méthode dépend essentiellement de l'évaluation.

Le problème qui nous intéresse est celui d'ordonnancer le passage de $p$ pièces sur 3 machines en série. On notera $A$, $B$ et $C$ les 3 machines, chaque pièce doit passer d'abord sur $A$, puis sur $B$ et enfin terminer par $C$ et chaque machine ne peut s'occuper que d'une pièce à la fois. La durée de passage de la pièce i sur les machines sera noté par $d^i_A$, $d^i_B$ et $d^i_C$ respectivement. On veut terminer le plus tôt possible.



**Pourquoi 3 machines?**

- S'il n'y a qu'une machine, quel que soit l'ordre de passage on termine dans le même temps.
- S'il y a 2 machines, l'algorithme polynomial suivant du à Johnson donne un ordre optimum :

  L'ordre de passage sur les 2 machines A et B est le même et donné par :

```
tant que toutes les pieces ne sont pas placees repeter
      soit t(iM) la duree minimum de passage sur une des machines M
         parmi les pieces restantes.
      si M=A, placer la piece en premier parmi les pieces restantes
 sinon la placer en dernier parmi les pieces restantes
      fin si;
     retirer i des pieces a placer ;
fin boucle ;
```

- s'il y a 4 ou plus machines, alors, comme le montre l'exemple suivant, il se peut que dans la solution optimale l'ordre de passage ne soit pas le même sur toutes les machines.

  **Exemple :** On a 4 machines A, B, C et D et deux pièces, les durée de passages sont donnés par le tableau suivant :

| | machine A | machine B | machine C | machine D |
|---|---|---|---|---|
| pièce 1 | 1 | 4 | 4 | 1 |
| pièce 2 | 4 | 1 | 1 | 4 |

  Si on fait passer la pièce 1 en premier sur toutes les machines alors on ne peut pas terminer en moins de 14. De même si c'est la pièce 2 qui passe en premier sur toutes les machines. Par contre si la pièce 1 en premier sur les deux premières machines et en dernier sur les autres on peut terminer en 12. Donc dans une solution optimale certaines pièces ont intérêt à attendre, alors qu'elles sont déjà engagées dans le processus, et laisser passer d'autres pièces avant elles. Ceci rend le problème très difficile. Il est d'ailleurs NP-difficile.

- le cas de 3 machines est aussi NP-difficile, mais suite au théorème suivant, l'ordre de passage, dans une solution optimale, est le même sur les 3 machines.

**Théorème :** *Dans le problème d'ordonnancement de $p$ pièces sur $m$ machines qui minimise l'heure de complétion de toutes les tâches, dans une solution optimale il n'y a jamais d'inversion d'ordre entre les deux premières machines ni entre les deux dernières.*

Dans le cas de 3 machines puisque l'ordre de passage est le même pour toutes les machines, l'on cherche l'une des $p!$ permutations qui permette de terminer le plus rapidement possible. La *division* peut être n'importe laquelle de celles que l'on a vu pour les permutations. C'est l'évaluation qui va nous guider vers l'une ou l'autre de celles-ci.

L'algorithme suivant est dû à Ignall et Schrage.

**Notations :** Soit $\pi$ un ensemble ordonné de pièces qui passera avant toutes les autres et dans l'ordre de $\pi$. Notons $\pi'$ l'ensemble des autres pièces. $t_A^\pi$, $t_B^\pi$ et $t_C^\pi$ sont les temps qu'il faut pour compléter les pièces de $\pi$ sur la première, les deux premières et sur les trois machines respectivement. C'est à dire qu'aucune autre pièce ne peut passer sur ces machines avant ces dates (la date de départ est fixée à 0).

Supposons donc que les pièces de $\pi$ passent avant toutes les autres et dans l'ordre de $\pi$. Toutes les autres pièces doivent encore passer sur la première machine, ce qui prend un temps de $\sum_{i\in\pi'} d_A^i$. Ceci, ajouté à $t_A^\pi$, est un minorant du temps total nécessaire mais on peut faire encore mieux. La dernière des pièces de $\pi'$ qui passera sur la première machine devra encore passer sur les deux autres et au mieux passer sans attendre. Pour obtenir un minorant du temps total il faut supposer que cette dernière pièce est celle pour laquelle la somme des durées de passage sur les deux dernières machine est minimum. On obtient donc un premier minorant $b_A^\pi$ sur le temps total quand les pièces de $\pi$ passent avant toutes les autres et dans l'ordre de $\pi$ (la somme des 2 premiers termes est la somme des durées de passage de toutes les pièces sur la première machine :

$$b_A^\pi = t_A^\pi + \sum_{i\in\pi'} d_A^i + min_{i\in\pi'}\{d_B^i + d_C^i\}$$

Ceci fournit une évaluation du nœud de l'arbre qui correspond à toutes les permutations qui commencent par $\pi$ dans l'ordre de $\pi$. On va essayer de faire mieux en cherchant d'autre minorants et en prenant le max de tous.

Le même raisonnement peut être fait au sortir de la machine B. Toutes les autres pièces doivent encore passer sur la cette machine, ce qui prend un temps de $\sum_{i\in\pi'} d_B^i$. Ensuite la dernière pièce doit encore passer sur la dernière machine, on a un minorant en supposant que c'est celle de $\pi'$ qui nécessite le moins de temps sur cette machine. On obtient donc un deuxième minorant $b_B^\pi$ (à noter que $t_B^\pi$ n'est pas nécessairement égal à la somme des durées de passage sur B).

$$b_B^\pi = t_B^\pi + \sum_{i\in\pi'} d_B^i + min_{i\in\pi'}\{d_C^i\}$$

Enfin en se plaçant après la machine C on obtient :

$$b_C^\pi = t_C^\pi + \sum_{i\in\pi'} d_C^i$$

Alors $b = \max(b_A^\pi, b_B^\pi, b_C^\pi)$ est un minorant du temps total nécessaire et est donc une évaluation du nœud de l'arbre qui correspond à toutes les permutations qui commencent par $\pi$ dans l'ordre de $\pi$. A noter qu'aucune des trois valeurs précédentes ne domine les autres.

**Exercice 15.3** *Résoudre par Branch and Bound l'exemple numérique suivant, en utilisant l'évaluation précédente. Les valeurs $t_B^\pi$ et $t_C^\pi$ se trouvent en faisant un diagramme de Gant (à barres) de l'occupation des machines.*

| | *machine A* | *machine B* | *machine C* |
|---|---|---|---|
| *pièce 1* | *3* | *4* | *10* |
| *pièce 2* | *11* | *1* | *5* |
| *pièce 3* | *7* | *9* | *12* |
| *pièce 4* | *10* | *12* | *2* |



On va voir qu'avec un peu d'effort on peut encore améliorer l'évaluation. Bien entendu, il faut trouver un compromis entre le temps passé à trouver une meilleure évaluation et celui que l'on gagne en développant un arbre plus petit.

Les bornes $b_B^\pi$ et $b_C^\pi$ peuvent être légèrement renforcées par la remarque suivante : le calcul de $t_B^\pi$ ignore le fait que, parfois, le moment où l'on peut commencer de s'occuper d'une pièce de $\pi'$ dépend de l'occupation de la machine A. Ce serait le cas par exemple si tous les temps de passage sur B pour les pièces de $\pi$ sont inférieurs ou égaux à ceux sur A et que toutes les pièces de $\pi'$ ont un temps de passage sur A supérieurs à celles de $\pi$ sur cette même machine. On peut remplacer $t_B^\pi$ par

$$t'^\pi_B = \max\{t_B^\pi, t_A^\pi + \min_{i\in\pi'} d_A^i\}$$

dans le calcul de $b_B^\pi$. De même on peut remplacer $t_C^\pi$ par :

$$t'^\pi_C = \max\{t_C^\pi, t_B^\pi + \min_{i\in\pi'} d_B^i, t_A^\pi + \min_{i\in\pi'} d_A^i + d_B^i\}$$

Pour aller plus loin il faut un d'effort de réflexion (et de calcul) plus conséquent. Les bornes précédentes peuvent être considérées comme des bornes "machine". On va développer des bornes "pièces". Avec $\pi$ et $\pi'$ ayant la même signification que précédemment, soit K une pièce de $\pi'$ et $J_1$ l'ensemble des éléments $i \neq K$ de $\pi'$ tels que $d_A^i \leq d_C^i$ et $J_2$ l'ensemble des éléments $i \neq K$ de $\pi'$ tels que $d_A^i > d_C^i$. Maintenant supposons que toutes les pièces de $J_1$ précèdent K ainsi que toutes celles de $J_2$. On a la situation de la figure 15.5. La date de complétion est supérieure ou égale à :

$$t_A^\pi + (d_A^K + d_B^K + d_C^K) + \sum_{i\in J_1} d_A^i + \sum_{i\in J_2} d_C^i$$

quelque soit l'ordre de passage de pièces de $\pi'$ (pourquoi?) et comme ceci est vrai pour toute pièce de $\pi'$ on obtient un nouveau minorant du temps total :

$$b_4 = t_A^\pi + \max_{K\in\pi'}\left\{ d_A^K + d_B^K + d_C^K + \sum_{i\in\pi', i\neq K} \min(d_A^i + d_B^i) \right\}$$

On peut faire un raisonnement analogue en raisonnant sur les machines B et C (cf figure 15.6) et on obtient :

$$b_5 = t_B^\pi + \max_{K\in\pi'}\left\{ d_B^K + d_C^K + \sum_{i\in\pi', i\neq K} \min(d_B^i + d_C^i) \right\}$$

Une meilleure évaluation s'obtient donc en prenant :

$$b' = \max\left\{b_\pi^A, b_\pi^B, b_\pi^C b_4, b_5\right\}$$

Le tableau suivant donne les résultats d'une expérience en utilisant la borne $b$ et la borne $b'$, la moyenne étant prise sur 50 problèmes de chaque taille. On ne dit pas comment les problèmes tests ont été générés. Les temps sont en secondes sur un CDC 3600 (entre 100 et 1000 fois plus lent qu'une SUN Sparc 10).

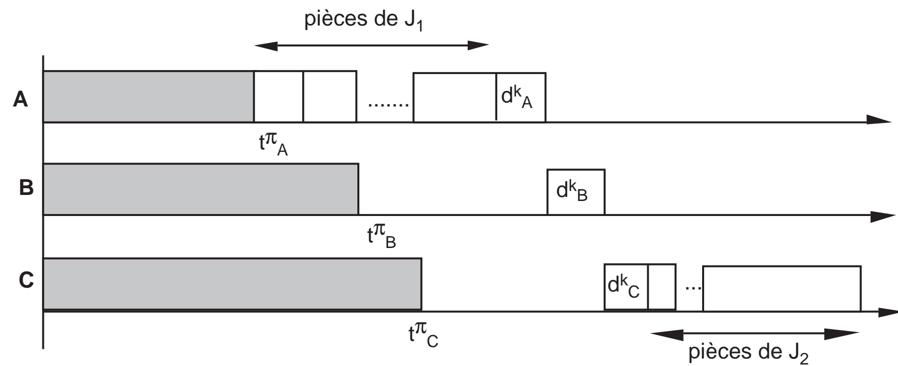


Figure 15.5: Passage de pièces sur trois machines.

Figure 15.6: Passage des pièces sur B et C.

| | Nombre moyen de nœuds | | Temps moyen d'exécution | |
|---|---|---|---|---|
| p | borne b | borne b' | borne b | borne b' |
| 4 | 12,1 | 9,4 | 0,030 | 0,024 |
| 5 | 22,2 | 16,0 | 0,065 | 0,045 |
| 6 | 43,5 | 26,6 | 0,197 | 0,089 |
| 7 | 71 | 38,7 | 0,898 | 0,160 |
| 8 | 137,1 | 71,8 | 1,744 | 0,409 |
| 9 | 148,8 | 89,7 | 2,449 | 0,635 |
| 10 | 143,9 | 90,9 | 3,313 | 0,631 |

Ce tableau suscite les commentaires suivants: La multiplication du temps de calcul par un facteur supérieur à 4 entre $p = 6$ et $p = 7$ dans le cas de la borne $b$, s'explique très certainement par le fait que du fait des mémoires très petites de l'époque, le problème ne tenait, à cause de la taille des arbres générés, à ce point plus en mémoire et qu'il fallait swapper très souvent. Ce phénomène se produirait aujourd'hui pour un $n$ beaucoup lus élevé. Par contre le doublement des nœuds entre $p = 7$ et $p = 8$ est difficile à expliquer faute de savoir comment ont été générés les problèmes. A noter l'augmentation des temps de calcul en fonction de $p$.

On peut encore essayer d'accélérer le temps de calcul par des remarques provenant d'une étude approfondie des performances de la méthode que l'on vient de développer. A chaque problème on peut associer le problème inverse qui consiste à imposer que les pièces passent sur les machines dans l'ordre C, B et A. Le temps total minimum nécessaire est le même et il suffit d'inverser l'ordre des pièces pour obtenir un ordre optimal pour l'autre problème. On peut alors se demander quand est-il plus facile de résoudre le problème inverse que l'original ? Un élément de réponse est donné par le concept de machine dominante. On dit que la machine C est dominante si $\sum_i d^i_C > \sum_i d^i_A$ et réciproquement. Quand la machine C est dominante, le temps pendant lequel elle est inactive est largement défini par les décisions en amont. On peut donc penser que quand cette machine est dominante,



les évaluations que l'on a faites seront bonnes. Par contre quand c'est la machine A qui est dominante, il vaudra mieux résoudre le problème inverse. Le tableau suivant donne les résultats d'une expérience en fonction de la machine dominante :

| | Nombre moyen de nœuds | | Temps moyen d'exécution | |
|---|---|---|---|---|
| p | A dominante | C dominante | A dominante | C dominante |
| 4 | 10,6 | 09,4 | 0,028 | 0,024 |
| 5 | 18,4 | 15,9 | 0,054 | 0,045 |
| 6 | 35,1 | 26,5 | 0,141 | 0,088 |
| 7 | 114,4 | 37,7 | 1,898 | 0,155 |
| 8 | 345.1* | 63,9 | 4.586* | 0,335 |
| 9 | 632.5* | 82,7 | 30.301* | 0,575 |
| 10 | 1036.8* | 81,3 | 382.750* | 0,571 |

* certains problèmes de cette classe n'ont pas été résolus dans la limite de 3000 nœuds fixés à l'algorithme.

On voit que la méthode a toujours réussi à trouver dans un temps raisonnable la solution quand la machine C était dominante et que ce n'est pas le cas si c'était la machine A dès que $p \geq 8$.

Pour conclure, ce que nous avons vu pour cet exemple est assez caractéristique de toute résolution de problème, à savoir qu'il ne suffit pas de choisir une méthode de résolution, un effort de réflexion sur les particularités du problème s'impose et peut faire la différence entre un problème que l'on peut résoudre et un que l'on ne peut pas.

**Exercice 15.4** *Considérons le problème d'ordonnancer 4 pièces sur 3 machines en série décrit par le tableau ci-dessous.*

1. *Résoudre avec la borne b.*

2. *Résoudre avec la borne b'.*

| | *machine A* | *machine B* | *machine C* |
|---|---|---|---|
| *pièce 1* | *13* | *3* | *12* |
| *pièce 2* | *7* | *12* | *16* |
| *pièce 3* | *26* | *9* | *6* |
| *pièce 4* | *2* | *6* | *1* |

# Annexe A

# Mesures sur les arbres

Soit $A$ un arbre comportant $n$ noeuds internes et $f$ feuilles.

**Taille de $A$ :** nombre de noeuds de $A$

$$\begin{cases} T(\emptyset) = 0 \\ T(\circ, G, D) = 1 + T(G) + T(D) = n + f \end{cases}$$

**Profondeur d'un noeud** (ou hauteur d'un noeud)

$$\begin{cases} h(x) = 0 & \text{si } x \text{ est la racine} \\ h(x) = 1 + h(y) & \text{si } y \text{ est le père de } x \end{cases}$$

**Profondeur d'un arbre** (ou hauteur)

$$h(A) = \max\{h(x)\ ,\ \ x \text{ noeud de } A\ \}$$

**Longueur de cheminement**

$$LC(A) = \sum_{x \in A} h(x)$$

**Longueur de cheminement interne**

$$LCI(A) = \sum_{\text{noeuds internes}} h(x)$$

**Longueur de cheminement externe**

$$LCE(A) = \sum_{\text{feuilles}} h(F)$$

**Profondeur moyenne interne :** profondeur moyenne d'un noeud interne

$$PI(A) = \frac{1}{n}\, LCI(A)$$

**Profondeur moyenne externe :** profondeur moyenne d'une feuille

$$PE(A) = \frac{1}{f}\, LCE(A)$$

**Arbre binaire localement complet :** arbre binaire dont tous les noeuds internes ont exactement 2 fils.





**Propriété :** dans un arbre binaire localement complet comportant $n$ noeuds internes :

$$\begin{cases} f = n + 1 \\ LCE(A) = LCI(A) + 2n \end{cases}$$

démonstration : par récurrence sur $n$.

# Annexe B

# Petit lexique d'algo-R.O.

Voici un petit lexique du vocabulaire le plus fréquent utilisé.
Suggestions bienvenues à Roger.Mohr@imag.fr.





| Français | Anglais | Allemand |
|---|---|---|
| absorbant (circuit) | dominating | äußerlich stabil |
| adjacent | adjacent | adjazent |
| appel (de proc.) | call | Aufruf |
| arbre | tree | Baum |
| arbre de décision | decision tree | Entscheidungsbaum |
| arbre équilibré | balanced tree | balancierter Baum |
| arbre maximal | spanning tree | Gerüst, Spannbaum |
| axiome | axiom | Axiom |
| arc | directed edge | Bogen |
| arête | edge, line | Kante |
| chaîne | chain | Kantenfolge |
| chaîne élémentaire | elementary chain | Weg |
| chemin | path | Bogenfolge, Pfad |
| chemin élémentaire | elementary path | elementare Bahn oder Pfad |
| circuit | circuit | geschlossene Bogenfolge |
| circuit élémentaire | elementary circuit | Elementarzirkuit |
| complet | complete | vollständig |
| complexité | complexity | Komplexität |
| composante connexe | connected component | Komponente |
| connecté | connected | zusammenhängend |
| creux (graphe –) | sparse | dünn besetzt |
| cycle | cycle | Zyklus |
| cycle élémentaire | elementary cycle | Kreis, Elementarzyklus |
| degré | degree, valency | Grad |
| demi degré intérieur | id-degree | Eingangsgrad |
| dense (graphe –) | dense | dicht |
| disjoint | disjoint | disjunkt |
| diviser pour règner | divide and conquer | Teile-und-herrsche-Verfahren |
| équidistribué | evenly distributed | gleichmäßig verteilt |
| espérance | expectation | Erwartung |
| file d'attente | queue | Warteschlange |
| flot | flow | Strom |
| génie logiciel | software engineering | Software Engineering |
| graphe | graph (directed –) | Graph (gerichtet, ungerichtet) |
| implantation | implementation | Implementierung |
| itératif | iterative | wiederholt, iterativ |
| | | .../... |

| Français | Anglais | Allemand |
|---|---|---|
| jeu | game | Spiel |
| liste | list | Liste |
| matrice d'adjacence | adjacency matrix | Adjazenzmatrix |
| matrice d'incidence | incidence matrix | Inzidenzmatrix |
| noeud | node | Knoten(punkt), Verzweigung |
| optimal | optimal | optimal |
| ordonné, trié | sorted | geordnet |
| ordonnancement | scheduling | Zuteilung |
| pile | stack | Stapel |
| pb d'ordonnancement | scheduling pb | Zuteilungsverfahren |
| programmation dynamique | dynamic programming | dynamische Programmierung |
| racine | root | Wurzel |
| récursif | recursive | rekursiv |
| réseau de tansfert | network | Netz, Netzwerk |
| S.E.P. | branch and bound | Verzweig-und-Begrenze-Suche |
| signature (profil) | profile | Signatur |
| sommet, point | vertex | Knotenpunkt |
| spécification | specification | Spezifikation, Spezifierung |
| successeur | successor | Nachfolger |
| structure de données | data structure | Datenstruktur |
| tableau | array | Feld |
| tas | heap | Haufen |
| transitif | transitive | transitiv |
| trier | to sort | sortieren |
| type abstrait | abstract data type | abstrakter Datentyp |
| voisin | neighbour | Nachbar |

# Annexe C

# Formulaire de calcul de coûts

Les formules de coût indiquées dans ce formulaire sont utiles non seulement pour analyser le coût d'un algorithme mais aussi pour guider la construction d'algorithmes de type Diviser pour Régner performants (séquentiels ou parallèles)

## C.1 Rappels sur les notations asymptotiques

Soient $f(n)$ et $g(n)$ deux fonctions à valeurs positives.

- $f = o(g) \iff \forall C > 0, \exists N_0 > 0 : \forall n > N_0 \quad f(n) \leq Cg(n)$
- $f = O(g) \iff \exists A > 0, \exists N_0 > 0 : \forall n > N_0 \quad f(n) \leq Ag(n)$
- $f = \Omega(g) \iff \exists A > 0, \exists N_0 > 0 : \forall n > N_0 \quad f(n) \geq Ag(n)$
- $f = \Theta(g) \iff \exists A, B > 0, \exists N_0 > 0 : \forall n > N_0 \quad Ag(n) \leq f(n) \leq Bg(n)$

Exemple: $12.345n^2 + 3.14n \log n + O(n) = \Theta(n^2)$

## C.2 Théorème magique

Soit $T(n)$ une fonction à valeur positive définie par la récurrence:

$$T(n) = \alpha T\left(\frac{n}{K}\right) + f(n)$$

où $\alpha \geq 1$ et $K > 1$ sont deux constantes et $f$ une fonction.
Alors, $T(n)$ peut être bornée asymptotiquement comme suit :

1. si $f(n) = O\left(n^{\log_K \alpha - \epsilon}\right)$ avec $\epsilon > 0$ alors $T(n) = \Theta\left(n^{\log_K \alpha}\right)$.
2. si $f(n) = \Theta\left(n^{\log_K \alpha}\right)$ alors $T(n) = \Theta\left(n^{\log_K \alpha} \log n\right)$.
3. si $f(n) = \Omega\left(n^{\log_K \alpha + \epsilon}\right)$ avec $\epsilon > 0$ et si $\exists c < 1$ tel que $\alpha f\left(\frac{n}{K}\right) \leq cf(n)$ pour $n$ suffisament grand alors $T(n) = \Theta\left(f(n)\right)$.
4. si $\alpha = 1$ et $f(n) = O\left(\log^k n\right)$ avec $k \geq 1$ alors $T(n) = O(\log^{k+1} n)$.

N.B.: Les résultats sont valides même lorsque $n/K$ est pris en partie entière inférieure ou supérieure.





**Exemples:**

- $T(n) = T(n/K) + O(1) \Longrightarrow$ (cas 2) $T(n) = \Theta(\log n)$
- $g(n) = g(n/K) + O(\log n) \Longrightarrow$ (cas 2 en posant $T = \frac{g}{\log n}$) $g(n) = O(\log^2 n)$
- $T(n) = KT(n/K) + \log^{O(1)} n \Longrightarrow$ (cas 1) $T(n) = \Theta(n)$
- $T(n) = 2T(n/3) + \Theta(n) \Longrightarrow$ (cas 3) $T(n) = \Theta(n)$

exercice 1: Expliciter l'utilisation du théorème sur ces 4 exemples.
exercice 2: Retrouvez le résultat de ces 4 exemples directement par déroulement de la récurrence.
exercice 3: (plus fastidieux) démontrer le théorème magique: cf [CLR90] pages 65–72)

## C.3 Découpe récursive à pas dynamique

Soit $T(n)$ une fonction à valeur positive définie par la récurrence:

$$T(n) = \alpha T\left(n^{1/K}\right) + f(n)$$

où $\alpha \geq 1$ et $K > 1$ sont deux constantes et $f$ une fonction telle que $f(n) \geq \alpha f\left(n^{1/K}\right)$.
Alors, $T(n)$ peut être bornée asymptotiquement par :

$$T(n) = \Theta\left[(\log n)^{log_K \alpha}\right] + O(f(n) \log \log n).$$

Si de plus $f(n) = \Omega(\log^c n)$ avec $c \geq 1$ alors $T(n) = \Theta\left[f(n) + \log^{log_K \alpha} n\right]$.

**Exemples:**

- $T(n) = T\left(\sqrt{n}\right) + O(1) \Longrightarrow T(n) = \Theta(\log \log n)$
- $T(n) = 2T\left(\sqrt{n}\right) + O(\log n) \Longrightarrow T(n) = \Theta(\log n)$

exercice 4: Retrouvez ces 2 exemples en déroulant la récurrence (le deuxième exemple est plus fin que la formule du théorème général).
exercice 5: En déduire la formule générale.

# Bibliographie


[AHU87] Alfred Aho, John Hopcroft, and Jeffrey Ullman. *Structures de données et algorithmes.* InterEditions, 1987.

[BB89] Pierre Berlioux and Philippe Bizard. *Algorithmique.* Dunod informatique, 1989.

[BBC92] Danièle Beauquier, Jean Berstel, and Philippe Chrétienne. *Eléments d'algorithmique.* Masson, 1992.

[CLR90] Thomas Cormen, Charles Leiserson, and Ronald Rivest. *Introduction to Algorithms.* MIT Press, 1990.

[FGS90] Christine Froidevaux, Marie-Claude Gaudel, and Michèle Soria. *Types de Données et Algorithmes.* Mc Graw-Hill, 1990.

[PMS88] Claude Pair, Roger Mohr, and René Schott. *Construire les algorithmes.* Dunod informatique, 1988.

[Sak84] M. Sakarovitch. *Optimisation Combinatoire, Programmation Discrète.* Herman, Paris, 1984.

[Sed91] Robert Sedgewick. *Algorithmes en langage C.* InterEditions, 1991.