Chapitre 6

Programmation dynamique

6.1 Programmation dynamique et problèmes d'optimisation

La programmation dynamique est une méthodologie générale pour concevoir des algorithmes permettant de résoudre efficacement certains problèmes d'optimisation. Un problème d'optimisation consiste à rechercher, parmi un ensemble de solutions d'un problème, celles qui optimisent un certain critère. Par exemple, trouver un plus court chemin pour aller d'un point à un autre dans un réseau de transport est un problème d'optimisation

La conception d'un algorithme de programmation dynamique se décompose en quatre étapes.

1. Caractérisation de la structure d'une solution optimale.
2. Définition récursive de la valeur de la solution optimale.
3. Calcul *ascendant* de la valeur de la solution optimale.
4. Construction de la solution optimale à partir des informations obtenues à l'étape précédente.

L'étape 4 peut être omise si on a seulement besoin de la valeur de la solution optimale et non de la solution elle même.

Les sections suivantes décrivent l'utilisation de la programmation dynamique pour résoudre en temps polynomial trois problèmes d'optimisation : le calcul d'un parcours optimal dans un atelier de montage, le calcul d'une chaîne de mulplications matricielles avec un nombre minimum d'opérations scalaires, le calcul d'un arbre binaire de recherche optimal. Ensuite, nous

mettrons en évidence deux caractéristiques que doit posséder un problème d'optimisation pour que la programmation dynamique soit applicable.

6.2 Ordonnancement optimal d'une chaîne de montage

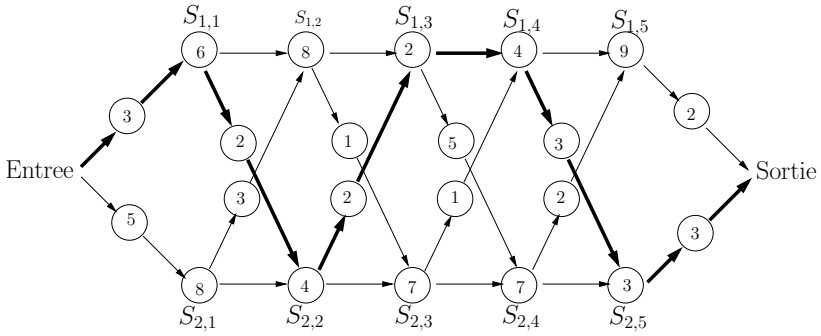
Un constructeur automobile possède un atelier avec deux chaînes de montage comportant chacune n postes de montages. Chaque véhicule doit passer par les n postes dans l'ordre. Le constructeur cherche à déterminer quels sont les postes à sélectionner sur la chaîne 1 et sur la chaîne 2 pour minimiser le délai de transit d'une voiture à travers l'atelier. Les données du problème d'optimisation qu'il doit résoudre sont les suivantes. Pour $i = 1, 2$ et $j = 1, \ldots, n$, on note $S_{i,j}$ le j-ème poste de la chaîne i, e_i le *temps d'entrée* d'un véhicule sur la chaîne i, $a_{i,j}$ le *temps de montage* pour le poste j sur la chaîne i, $t_{i,j}$ le *temps de transfert* d'un véhicule de la chaîne i vers l'autre chaîne après le poste $S_{i,j}$ et finalement x_i le temps de sortie d'un véhicule de la chaîne i (voir Figure 6.1a).

Chaque solution de ce problème d'optimisation est définie par le sous-ensemble de postes de la chaîne 1 utilisés (les postes restant sont choisis dans la chaîne 2). Il y a donc 2^n solutions possibles, i.e. le nombre de sous-ensembles d'un ensemble à n éléments. Par conséquent, l'approche naïve consistant à considérer tous les chemins possibles est inefficace. La programmation dynamique permet de résoudre ce problème efficacement.

La *première étape* consiste à identifier des sous-problèmes dont les solutions optimales vont nous permettre de reconstituer une solution optimale du problème initial. Les sous-problèmes à considérer ici consistent à calculer un itinéraire optimal jusqu'au poste $S_{i,j}$ pour $i = 1, 2$ et $j = 1, \ldots, n$. Par exemple, considérons un itinéraire optimal jusqu'au poste $S_{1,j}$. Si $j = 1$, il n'y a qu'un seul chemin possible. Pour $j = 2, \ldots, n$, il y a deux possibilités. Un itinéraire optimal jusqu'à $S_{1,j}$ est, ou bien un itinéraire optimal jusqu'à $S_{1,j-1}$ suivi du poste $S_{1,j}$, ou bien, un itinéraire optimal jusqu'à $S_{2,j-1}$ suivi d'un changement de chaîne et du poste $S_{1,j}$.

La *deuxième étape* consiste à définir la valeur optimale de manière récursive à partir des valeurs des solutions optimales des sous-problèmes. Soit $f_i[j]$ le délai optimal jusqu'à $S_{i,j}$ et f^* le délai optimal total. Pour traverser l'atelier, il faut atteindre ou bien $S_{1,n}$ ou bien $S_{2,n}$ et sortir de l'atelier. Par conséquent, on a

$$f^* = \min(f_1[n] + x_1, f_2[n] + x_2).$$



(a)

j	1	2	3	4	5
$f_1[j]$	9	17	19	23	32
$f_2[j]$	13	15	22	29	29

$f^* = 32$

j	2	3	4	5
$l_1[j]$	1	2	1	1
$l_2[j]$	1	2	2	1

$l^* = 2$

(b)

FIGURE 6.1 – Exemple d'instance du problème d'ordonnancement optimal d'une chaîne de montage : (a) les données du problème et (b) les tables de programmation dynamique.

Pour le poste 1 de chaque chaîne, il n'y a qu'un itinéraire possible :

$$f_1[1] = e_1 + a_{1,1}$$
$$f_2[1] = e_2 + a_{2,1}$$

Pour le poste $j = 2, \ldots, n$ de la chaîne 1, il y a deux possibilités :

$$f_1[j] = \min(f_1[j-1] + a_{1,j} \ , \ f_2[j-1] + t_{2,j-1} + a_{1,j}), \qquad (6.1)$$

et symétriquement

$$f_2[j] = \min(f_2[j-1] + a_{2,j} \ , \ f_1[j-1] + t_{1,j-1} + a_{2,j}). \qquad (6.2)$$

Les $f_i[j]$ sont les valeurs des solutions optimales des sous-problèmes. Pour pouvoir reconstruire les solutions optimales elles-mêmes, on définit $l_i[j]$ le numéro de la chaîne (1 ou 2) dont le poste $j-1$ est utilisé par un chemin optimal jusqu'au poste $S_{i,j}$ pour $j = 2, \ldots, n$, ($l_i[1]$ n'est pas défini car aucun poste ne vient avant le poste 1).

La *troisième étape* consiste à concevoir un algorithme qui calcule en temps polynomial les valeurs $f_i[j]$ des solutions optimales et les informations $l_i[j]$ nécessaires à la construction de ces solutions. L'algorithme suivant fait ce travail en $O(n)$ en utilisant les formules récursives (6.1) et (6.2). Il revient à remplir les tables de la Figure 6.1b de la gauche vers la droite.

Algorithme 19: CheminLePlusRapide

entrée : Les tableaux a, t, e, x et le nombre de postes n.
résultat : les tableaux f et l, les variables f^* et l^*.
début
  pour $i = 1, 2$ **faire**
    $f_i[1] := e_i + a_{i,1}$
  pour $j = 2$ **à** n **faire**
    si $f_1[j-1] \leq f_2[j-1] + t_{2,j-1}$ **alors**
      $f_1[j] := f_1[j-1] + a_{1,j}$
      $l_1[j] := 1$
    sinon
      $f_1[j] := f_2[j-1] + t_{2,j-1} + a_{1,j}$
      $l_1[j] := 2$
    si $f_2[j-1] \leq f_1[j-1] + t_{1,j-1}$ **alors**
      $f_2[j] := f_2[j-1] + a_{2,j}$
      $l_2[j] := 2$
    sinon
      $f_2[j] := f_1[j-1] + t_{1,j-1} + a_{2,j}$
      $l_2[j] := 1$
  si $f_1[n] + x_1 \leq f_2[n] + x_2$ **alors**
    $f^* = f_1[n] + x_1$
    $l^* = 1$
  sinon
    $f^* = f_2[n] + x_2$
    $l^* = 2$

Finalement, la *quatrième et dernière étape* consiste à reconstruire une solution optimale en utilisant les informations sauvegardées à l'étape 3. La procédure suivante affiche les postes utilisés par une solution optimale par ordre décroissant de numéro de poste.

Algorithme 20: AfficherPostes

entrée : Le tableau l et le nombre de postes n.
résultat : Les chaînes utilisées dans une solution optimale.
début
  $i := l^*$
  afficher "chaîne" i, "poste" n
  pour $j := n$ **à** *2* **faire**
    $i := l_i[j]$
    afficher "chaîne" i, "poste" $j - 1$

6.3 Chaîne de multiplications matricielles

On se donne une suite de n matrices $A_1, \ldots, A_n$ et on veut calculer le produit

$$A_1 A_2 \ldots A_n \tag{6.3}$$

Une fois que l'on a parenthésé cette expression de manière à supprimer l'ambiguïté liée à l'ordre dans lequel les multiplications doivent être effectuées, on peut évaluer l'expression (6.3) en utilisant comme routine l'algorithme standard de multiplication de deux matrices.

On dit d'un produit de matrices qu'il est *complètement parenthésé* dans les deux cas suivants :

— c'est une matrice isolée,
— c'est le produit de deux matrices complètement parenthésées.

Le produit de matrices est associatif par conséquent quelque soit le parenthésage on obtient le même résultat. Par exemple, si la chaîne de matrices est A_1, A_2, A_3, A_4, le produit $A_1 A_2 A_3 A_4$ peut être parenthésé de 5 façons différentes :

$$(A_1(A_2(A_3A_4))),$$
$$(A_1((A_2A_3)A_4)),$$
$$((A_1A_2)(A_3A_4)),$$
$$((A_1(A_2A_3))A_4),$$
$$(((A_1A_2)A_3)A_4).$$

La façon dont on parenthèse une telle chaîne de matrices peut avoir une grande importance sur le nombre d'opérations nécessaires pour effectuer le produit.

Si A est une matrice $p \times q$ et B une matrice $q \times r$, la matrice produit est une matrice $q \times r$. La complexité du calcul du produit est dominé par le nombre de multiplications scalaires qui est égal à pqr.

Pour illustrer les différences de coûts obtenus en adoptant des parenthésages différents, considérons un produit de 3 matrices $A_1,\ A_2,\ A_3$ de dimensions respectives 10×100, 100×5 et 5×50. Si on effectue les multiplications dans l'ordre donné par le parenthésage $((A_1A_2)A_3)$, le nombre d'opérations nécessaires est $10 \times 100 \times 5 = 5000$ pour obtenir le produit A_1A_2, qui est une matrice 10×5, plus $10 \times 5 \times 50 = 2500$ pour obtenir le produit $((A_1A_2)A_3)$, soit 7500 opérations en tout. Si on adopte le parenthésage $(A_1(A_2A_3))$, le nombre d'opérations nécessaires est $100 \times 5 \times 50 = 25000$ pour obtenir le produit A_2A_3, qui est une matrice 100×50, plus $10 \times 100 \times 50 = 50000$ pour obtenir le produit $(A_1(A_2A_3))$, soit 75000 opérations en tout. Par conséquent, calculer le produit en accord avec le premier parenthésage est 10 fois plus rapide.

Notre problème se formule de la façon suivante : étant donné une suite de n matrices $A_1, \ldots, A_n$, avec $p_{i-1} \times p_i$ la dimension de la matrice A_i, pour $i = 1, \ldots, n$, trouver le parenthésage du produit $A_1A_2 \ldots A_n$ qui minimise le nombre de multiplications scalaires à effectuer.

Remarque : le nombre α_n de parenthésages différents pour un produit de n termes est connu sous le nom de *nombre de Catalan*. On peut montrer les relations suivantes

1. $\alpha_1 = 1$ et $\alpha_n = \sum_{p=1}^{n-1} \alpha_p \alpha_{n-p}$.
2. $\alpha_{n+1} = \frac{(2n)!}{n!(n+1)!}$.
3. Et en appliquant la formule de Stirling,
$$\alpha_{n+1} \sim \frac{4^n}{n^{3/2}\sqrt{\pi}}.$$

C'est une fonction exponentielle de n. Par conséquent, la stratégie qui consiste à énumérer tous les parenthésages est exclue pour de grandes valeurs de n.

6.3.1 Structure d'un parenthésage optimal

La première étape dans une approche de type programmation dynamique consiste à identifier la structure des solutions optimales.

Notons $A_{i..j}$ la matrice qui résulte de l'évaluation du produit $A_i A_{i+1} \dots A_j$. Un parenthésage optimal de $A_1 \dots A_n$ découpe le produit entre les matrices A_k et A_{k+1} pour un certain k compris entre 1 et $n-1$. C'est-à-dire que pour un certain k, on calcule d'abord le produit $A_{1..k}$ et $A_{k+1..n}$, ensuite on multiplie ces produits pour obtenir le produit final $A_{1..n}$. Le coût de ce parenthésage est la somme des coûts du calcul de $A_{1..k}$ et $A_{k+1..n}$ et du produit de ces deux matrices.

Remarquons que le parenthésage de $A_{1..k}$ doit être lui-même un parenthésage optimal de $A_1 \dots A_k$, de même le parenthésage de $A_{k+1..n}$ doit être optimal. Par conséquent, une solution optimal d'un problème de parenthésage contient elle-même des solutions optimales de sous-problèmes de parenthésage. La présence de sous-structures optimales dans une solution optimale est l'une des caractéristiques des problèmes pour lesquels la programmation dynamique est applicable.

6.3.2 Une solution récursive

La seconde étape dans une approche de type programmation dynamique consiste à définir la valeur de la solution en fonction des solutions optimales de sous-problèmes. Dans notre cas, un sous-problème consiste à déterminer le coût minimal d'un parenthésage de $A_i \dots A_j$ pour $1 \leq i \leq j \leq n$. Soit $m[i,j]$ le nombre minimum de multiplications scalaires nécessaires pour obtenir le produit $A_{i..j}$. Le nombre minimum de multiplications scalaires nécessaires pour obtenir $A_{1..n}$ sera $m[1,n]$.

On peut calculer $m[i,j]$ récursivement de la façon suivante. Si $i = j$, la chaîne de matrices se réduit à une seule matrice $A_{i..i} = A_i$, aucune opération n'est nécessaire, et donc $m[i,i] = 0$ pour $i = 1, \dots, n$. Pour calculer $m[i,j]$ quand $i < j$, on se sert de la structure des solutions optimales que nous avons précédemment mise en évidence. Supposons que la solution optimale du sous-problème découpe le produit $A_i \dots A_j$ entre A_k et A_{k+1} avec $i \leq k < j$. Alors, $m[i,j]$ est égal au nombre minimum de multiplications scalaires pour obtenir $A_{i..k}$ et $A_{k+1..j}$ plus le nombre de multiplications nécessaires pour effectuer le produit matriciel $A_{i..k} A_{k+1..j}$, c'est-à-dire $p_{i-1} p_k p_j$ multiplications scalaires, on obtient

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

Cette équation récursive suppose que nous connaissions la valeur de k, ce qui n'est pas le cas. Il y a seulement $j - i$ valeurs possibles pour k :

les valeurs $i, i+1, \ldots, j-1$. Puisque la solution optimale correspond à l'une de ces valeurs, il suffit de les essayer toutes et de garder la meilleure. Par conséquent, notre définition récursive pour le coût minimum d'un parenthèsage de $A_i \ldots A_j$ devient

$$m[i,j] = \begin{cases} 0 & \text{si } i = j, \\ \min_{i \le k < j}\{m[i,k] + m[k+1,j] + p_{i-1}p_k p_j\} & \text{si } i < j. \end{cases} \tag{6.4}$$

Les valeurs $m[i,j]$ donnent les coûts des solutions optimales des sous-problèmes. Pour reconstituer une solution optimale a posteriori, nous allons stocker dans $s[i,j]$ une valeur de k qui minimise $m[i,k] + m[k+1,j] + p_{i-1}p_k p_j$, i.e. telle que $m[i,j] = m[i,k] + m[k+1,j] + p_{i-1}p_k p_j$.

6.3.3 Calcul du coût optimal

Au point où nous en sommes, on peut écrire facilement un programme récursif basé sur la relation de récurrence (6.4) pour calculer le coût de la solution optimale. Cependant, cet algorithme n'a pas une meilleure complexité qu'un algorithme énumératif brutal.

En fait, on peut remarquer qu'il existe relativement peu de sous-problèmes : un pour chaque choix de i et j qui satisfasse $1 \le i \le j \le n$, c'est-à-dire à peu près n^2 (en fait, $n(n+1)/2+n$ exactement). Un algorithme récursif peut rencontrer plusieurs fois chacun de ces sous-problèmes dans l'arbre des appels récursifs. Cette propriété est la deuxième caractéristique des problèmes pour lesquels la programmation dynamique est applicable.

Au lieu de calculer la solution de la récurrence (6.4) récursivement, nous allons effectuer la troisième étape la troisième étape d'une approche de type programmation dynamique en calculant le coût optimal de façon *ascendante*.

L'algorithme suivant remplit la table en commençant par résoudre le problème de parenthésage sur les chaînes de matrices les plus courtes et en continuant par ordre croissant de longueur de chaînes. L'équation de récurrence (6.4) montre que l'on peut calculer le coût optimal pour une chaîne en connaissant les coûts optimaux pour les chaînes de longueurs inférieures.

Algorithme 21: OrdreChaîneMatrices

entrée : Le tableau p des dimensions.
résultat : m et s.
début
  $n :=$ longueur(p)-1
  pour $i := 1$ **à** n **faire**
    $m[i, i] := 0$
  pour $l := 2$ **à** n **faire**
    # longueur des intervalles à traiter
    pour $i := 1$ **à** $n - l + 1$ **faire**
      # lignes à traiter
      $j := i + l - 1$ # colonne correspondante
      $m[i, j] := +\infty$ # calcul d'un minimum
      pour $k := i$ **à** $j - 1$ **faire**
        $q := m[i, k] + m[k + 1, j] + p_{i-1}p_kp_j$
        si $q < m[i, j]$ **alors**
          $m[i, j] := q$ et $s[i, j] := k$

Cet algorithme est en $O(n^3)$. En effet, il contient trois boucles imbriquées dans lesquelles chaque index peut prendre au plus n valeurs. De plus, le traitement est effectué à l'intérieur de ces boucles se fait en temps constant. Par conséquent, cet algorithme est beaucoup plus efficace que la méthode exponentielle qui consiste à examiner chaque parenthésage.

6.3.4 Construction d'une solution optimale

L'algorithme que nous avons décrit précédemment donne le nombre minimum de multiplications scalaires nécessaires pour calculer la chaîne de produits matriciels mais ne montre pas directement comment multiplier ces matrices en utilisant ce nombre minimum de multiplications. La dernière étape dans notre approche de type programmation dynamique consiste à reconstruire une solution optimale à partir des informations que nous avons sauvegardées à l'étape précédente.

Nous allons utiliser la table $s[\ ,\ ]$ pour déterminer la meilleure façon de multiplier les matrices. Chaque entrée $s[i, j]$ de cette table contient la valeur k telle qu'un parenthésage optimal sépare le produit $A_iA_{i+1} \dots A_j$ entre A_k et A_{k+1}. Par conséquent, nous savons que la dernière multiplication matricielle dans la solution optimale que nous voulons reconstruire

est $A_{1..s[1,n]}A_{s[1,n]+1..n}$. Les multiplications précédentes peuvent être reconstituées récursivement de la même façon. La procédure suivante affiche le parenthésage optimal du produit matriciel $A_{i..j}$ étant donnée la table $s[\ ,\ ]$ calculée à l'étape précédente et les indices i et j. Pour calculer le parenthésage complet, le premier appel de cette procédure sera AFFICHAGE-PARENTHÉSAGE-OPTIMAL$(s, 1, n)$.

Algorithme 22: AffichageParenthésageOptimal

```
entrée : Tableau s et indices i ≤ j.
résultat : Affichage du parenthésage optimal.
début
    si i = j alors
        Afficher "A_i"
    sinon
        Afficher "("
        AffichageParenthésageOptimal(s, i, s[i, j])
        Affichage-Parenthésage-Optimal(s, s[i, j] + 1, j)
        Afficher ")"
```

Exemple On peut vérifier qu'avec 6 matrices $M_0, M_2, \ldots, M_5$, de dimensions respectives : (2,1), (1,4), (4,2), (2,5), (5,1) et (1,3), on obtient des matrices m et s égales à

$$m = \begin{pmatrix} 0 & 8 & 12 & 28 & 22 & 28 \\ & 0 & 8 & 18 & 20 & 23 \\ & & 0 & 40 & 18 & 30 \\ & & & 0 & 10 & 16 \\ & & & & 0 & 15 \\ & & & & & 0 \end{pmatrix} \text{ et } s = \begin{pmatrix} 0 & 0 & 0 & 0 & 4 \\ & 1 & 2 & 2 & 4 \\ & & 2 & 2 & 4 \\ & & & 3 & 4 \\ & & & & 4 \\ & & & & \end{pmatrix}$$

conduisant au parenthésage optimal $((M_0((M_1M_2)(M_3M_4)))M_5)$ qui nécessite 28 multiplications. Le parenthésage $((M_0M_1)((M_2M_3)(M_4M_5)))$ en nécessite 147.

6.4 Arbre binaire de recherche optimal

Le problème d'optimisation que nous allons considérer dans cette section consiste étant donné un ensemble ordonné de clefs $\{a_1, a_2, \ldots, a_n\}$ et $p_1, p_2, \ldots, p_n$ les fréquences de recherche de ces clefs, à calculer un arbre

binaire de recherche qui permettent de minimiser le temps de recherche de ces clefs. En remarquant que le temps nécessaire pour rechercher une clef est proportionnel à sa profondeur dans l'arbre binaire de recherche, on déduit que l'arbre binaire de recherche que nous souhaitons identifier doit minimiser la quantité suivante

$$\text{coût}(T) = \sum_{i=1}^{n} p_i \times (\text{prof}_T(a_i) + 1)$$

où $prof_T(a_i)$ est la profondeur de la clef a_i dans l'arbre T.

clefs	a_1	a_2	a_3	a_4	a_5
fréquences	20	40	12	16	12

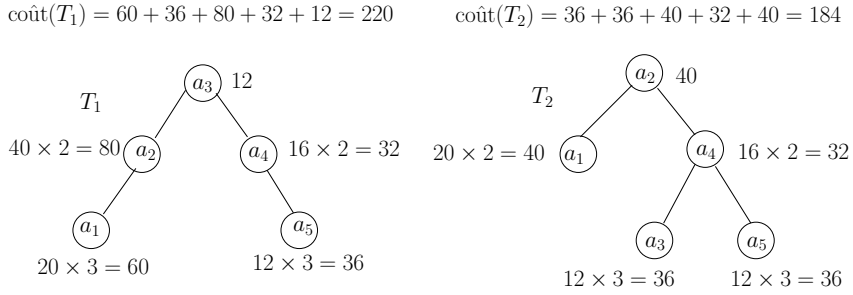


FIGURE 6.2 – Un exemple d'instance du problème de l'arbre binaire de recherche optimal avec deux arbres et leurs coûts respectifs.

Pour commencer, nous allons mettre en évidence la présence de sous-structures optimales dans un arbre binaire de recherche (ABR) optimal. Pour cela, nous allons introduire les notations suivantes :

— $T_{i,j}$ un ABR optimal pour les clefs $a_{i+1}, \ldots, a_j$;
— $w_{i,j} = p_{i+1} + \ldots + p_j$ la somme des fréquences des clefs de l'arbre $T_{i,j}$;
— $r_{i,j}$ la racine de l'arbre $T_{i,j}$;
— $c_{i,j}$ le coût de l'arbre $T_{i,j}$;
— $T_{i,i}$ l'arbre vide ($c_{i,i} = 0$).

Considérons un ABR optimal $T_{i,j}$ et notons k l'entier compris entre $i+1$ et j tel que la racine $r_{i,j}$ de l'arbre $T_{i,j}$ soit a_k. Le sous-arbre gauche de la racine a_k est constitué des clefs $a_{i+1}, \ldots, a_{k-1}$. De plus, ce sous-arbre T

doit nécessairement être un ABR optimal sur cet ensemble de clefs car s'il existait un meilleur ABR T' sur cet ensemble de clefs alors en remplaçant T par T' dans $T_{i,j}$ on obtiendrait un arbre meilleur que $T_{i,j}$, ce qui contredirait l'optimalité de $T_{i,j}$. Le même raisonnement s'applique au sous-arbre droit et montre que l'arbre optimal $T_{i,j}$ est constitué d'une racine a_k dont le fils gauche est la racine d'un ABR optimal $T_{i,k-1}$ et le fils droit est la racine d'un ABR optimal $T_{k,j}$ (voir Figure 6.3).

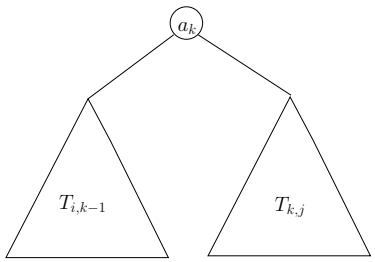


FIGURE 6.3 – Sous-structures optimales dans un ABR optimal

Voyons maintenant comment donner une définition récursive du coût d'un ABR optimal. Si a_k est la racine de T_{ij}, on a :

$$\begin{aligned} c_{i,j} &= (c_{i,k-1} + w_{i,k-1}) + p_k + (c_{k,j} + w_{k,j}) \\ &= (w_{i,k-1} + p_k + w_{k,j}) + c_{i,k-1} + c_{k,j} \\ &= w_{i,j} + c_{i,k-1} + c_{k,j} \end{aligned}$$

La première ligne s'explique par le fait que lorsque l'on place l'arbre $T_{i,k-1}$ sous la racine a_k, la profondeur de chacun de ces noeuds augmente de un, donc globalement le coût $c_{i,k-1}$ de ce sous-arbre augmente de la somme des fréquences des clefs qu'il contient, c'est-à-dire $w_{i,k-1}$. Idem pour la contribution du sous-arbre droit $T_{k,j}$ qui augmente de $w_{k,j}$. Finalement, il reste à compter la contribution de a_k qui est égale à p_k. La deuxième ligne est juste une réécriture de la première et la troisième utilise simplement la définition de $w_{i,j}$.

Pour finir, on remarque que la valeur de k à utiliser est celle qui minimise le coût $c_{i,k-1} + c_{k,j}$, ce qui donne la définition récursive suivante :

$$c_{i,j} = w_{i,j} + \min_{k \in \{i+1,\ldots,j\}} (c_{i,k-1} + c_{k,j})$$

L'algorithme suivant consiste à résoudre les sous-problèmes sur des sous-ensembles de clefs de longueurs l croissantes, en utilisant la formule récursive

ci-dessus :

Algorithme 23: ABR_Optimal

entrée : Tableau des fréquences p et nombre de clés n.
résultat : Les tableaux des coûts et des racines, c et r.
début
  pour $i := 0$ **à** n **faire**
    $w_{i,i} := 0$
    $c_{i,i} := 0$
  pour $l := 1$ **à** n **faire**
    pour $i := 0$ **à** $n - l$ **faire**
      $j := i + l$
      $w_{i,j} := w_{i,j-1} + p_j$
      Soit m la valeur de k telle que $c_{i,k-1} + c_{k,j}$ soit minimal
      $c_{i,j} = w_{i,j} + c_{i,m-1} + c_{m,j}$
      $r_{i,j} := m$

L'algorithme précédent permet d'obtenir le coût d'un ABR optimal et sauvegarde les informations nécessaires pour le construire grâce à l'algorithme suivant.

Algorithme 24: ConstABR_Opt

entrée : Le tableau des racines r et deux indices $i \leq j$.
résultat : L'arbre binaire de recherche correspondant correspondant.
début
  p:= **CréerNoeud**($r_{i,j}$, NULL, NULL)
  si $i < r_{i,j} - 1$ **alors**
    `p->fg` := **Const-ABR-opt**$(i, r_{i,j} - 1)$
  si $r_{i,j} < j$ **alors**
    `p->fd` := **ConstABR_Opt**$(r_{i,j}, j)$

6.5 Applicabilité de la programmation dynamique

Dans cette section, nous présentons deux caractéristiques que doit avoir un problème d'optimisation pour que la programmation dynamique soit applicable : la présence de sous-structures optimales et le recouvrement des sous-problèmes.

6.5.1 Sous-structures optimales

La première étape dans une approche de type programmation dynamique consiste à identifier la structure des solutions optimales. On dit que le problème présente une *sous-structure optimale* si une solution optimale contient des solutions optimales de sous-problèmes.

C'était le cas dans les trois problèmes précédents. Par exemple, chaque parenthésage optimal de $A_1 \dots A_n$ était obtenu comme le produit du parenthésage optimal de $A_1 \dots A_k$ et de $A_{k+1} \dots A_n$ pour un certain k et que chaque ABR optimal peut être obtenu en attachant à une racine bien choisie deux ABR optimaux pour des sous-problèmes.

6.5.2 Recouvrement des sous-problèmes

Pour que la programmation dynamique soit applicable, il faut également que l'espace des sous-problèmes soit suffisamment "petit". Dans le sens où un algorithme récursif qui résoudrait le problème rencontrerait les mêmes sous-problèmes plusieurs fois, au lieu de générer toujours de nouveaux sous-problèmes. Pour que la programmation dynamique permette de concevoir un algorithme polynomial, il faut bien sûr que le nombre de sous-problèmes à considérer soit polynomial.