Optimal Binary Search Tree

1 Équation de Bellman

Pour ne pas subir le calcul des sommes pendant la construction de l'arbre, on choisit d'utiliser un tableau auxiliaire sp de n cases, stockant les sommes partielles des p_i. Autrement dit, chaque case i de sp contient les sommes de $j = 0$ à i des p_j. Ce pré-travail coûte $O(n)$ opérations et $O(n)$ espace de stockage mémoire. Ainsi, notre somme se calcule en $O(1)$ opérations comme suit :

$$\sum_{l=i}^{j-1} p_l = \begin{cases} sp[j-1] - sp[i-1] & \text{si } i \neq 0 \\ sp[j-1] & \text{sinon.} \end{cases}$$

Pour notre programme, comme on a $\frac{n(n-1)}{2}$ possibilités pour les couples (i, j) avec $0 \leq i < j < n$, l'espace requis est en $\Theta(n^2)$.

Par ailleurs, on sait que tout sous-arbre d'un ABR optimal est un ABR optimal. Ainsi, on choisit de prendre comme racine tous les noeuds de l'ABR en faisant varier k de i à $j-1$. Lorsque l'on choisit que le k-ème noeud est la racine, on calcule récursivement le coût optimal de i à $k-1$ et de $k+1$ à j. Comme l'on choisit k parmi $j-i$ possibilités, la complexité temporelle est en $\Theta(n^3)$

Une autre façon de voir notre problème est de considérer le total de n^2 sous-problèmes (ou du moins $\Theta(n^2)$). Sachant que chaque sous-problème est de complexité $\Theta(n)$ si l'on suppose que tous les sous-problèmes sont déjà résolus, on obtient une complexité totale en $\Theta(n^3)$.

Maintenant, on peut restreindre le choix de k en sachant que $r(i, j-1) \leq r(i, j) \leq r(i+1, j)$, montrons-le.

L'équation de Bellman devient

$$C(i,j) = min_{r(i,j-1) \leq k \leq r(i+1,j)} C(i,k) + C(k+1,j) + \sum_{l=i}^{j-1} p_l$$

.

Grâce à cette restriction, $\sum_{i=1}^{n} \sum_{j=1}^{n} r(i+1,j) - r(i,j-1) + 1 \leq n^2 + \sum_{i=1}^{n} r(i,n) + \sum_{j=1}^{n} r(n+1,j) = O(n^2)$.

Le temps de calcul devient $O(n^2)$.

2 Analyse et Implémentation

2.1 Explication de l'algorithme

Sachant que les temps asymptotiques en itératif et récursif sont les mêmes, on choisit d'employer une méthode itérative, notamment pour éviter une surchage de pile en mémoire.

Le principe de notre algorithme se présente comme suit : on définit deux tableaux auxiliaires, notés c et r, de taille $\frac{n(n+1)}{2}$ pour stocker les coûts et racines des sous-arbres optimaux. Le calcul se fait en ascendant, en partant de la construction des plus petits sous-arbres pour construire les plus grands. La matrice des coûts – présentée ci-dessous – est triangulaire supérieure, contient sur sa diagonale pricipale les coûts des sous-arbres optimaux dont le nombre de sommets est égal à 1. On rappelle que notre programme ne stocke essentiellement que la partie triangulaire supérieure, on dispose d'un itérateur pour parcourir tous ces coefficients.

$$\begin{pmatrix} c_{0,1} & c_{0,2} & \cdots & \cdots & c_{0,n-1} \\ & c_{1,2} & \cdots & \cdots & c_{1,n-1} \\ & & \ddots & & \vdots \\ & 0 & & \ddots & \vdots \\ & & & & c_{n-2,n-1} \end{pmatrix}$$

Parallèlement, on dispose d'une matrice r construite de la même manière que c. L'idée de l'algorithme repose sur la construction pas à pas des diagonales de la matrice, débutant par la diagonale principale. Chaque diagonale représente des chaînes de sommets de longueur $j-i$ et dépend essentiellement des diagonales inférieures. Au fur et à mesure du calcul des coûts, on actualise r en déterminant le sommet du sous-arbre optimal qui minimise le coût, tout en appliquant bien évidemment la restriction de Knuth. Ainsi, la profondeur moyenne de l'arbre complet est $c_{0,n-1}$, soit la diagonale la plus haute. Ainsi, on dispose de r contenant l'intégralité des racines des sous-arbres minimisant le coût, on peut construire l'arbre en parcourant dans le sens inverse, de la diagonale la plus haute jusqu'à la diagonale principale.

Exemple : soit

2.2 Complexités

Sachant que la construction de l'arbre complet depuis r demande un espace mémoire de $2n$, notre algorithme utilise un espace mémoire de $2\frac{n(n+1)}{2} + 2n = O(n^2)$.

Le pire des cas se produit quand la profondeur de l'arbre est égale au nombre de sommets de l'arbre, autrement dit quand on a $r(i, j-1) - r(i+1, j) + 1$ est maximal.

Concernant le nombre de défauts de cache :

1. Si le cache est assez grand pour contenir tous les espaces de stockage, c'est-à-dire si $Z \gg n(n+1)+2n$ alors $Q(n, L, Z) = \frac{n(n+1)+2n}{L} = O(\frac{n^2}{L})$.
2. Si le cache est trop petit pour contenir un espace de taille n, c'est-à-dire si $Z \ll n$ alors on a déjà $\frac{n}{L}$ défauts de cache pour écrire la diagonale principale.