Rapport de TP 4MMAOD : Génération d'ABR optimal

ARGENTO Adrien Groupe 1
VINCENT Nicolas Groupe 1

13 octobre 2020

1 Principe de notre programme

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 :

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

Sachant que les temps asymptotiques en itératif et récursif sont les mêmes, on choisit d'employer une méthode itérative (dynamic-programming), 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 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.

L'utilisation du cache peut être améliorée. En effet, au lieu de boucler en commençant par la diagonale, (d, i, k), on peut itérer en (i, j, k) ou (j, i, k). La boucle commençant par i conduit naturellement à moins de défauts de caches, le language C ayant une structure mémoire en *row-major*. Nous avons implémenté et comparé ces trois algorithmes.

Nous proposerons en ouverture un algorithme plus performant en cache.

2 Analyse du coût théorique

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 $r(i,j-1) \leq r(i,j) \leq r(i+1,j)$.

En effet, posons $w(i,j) = p_i + ... + p_{j-1}$. On a $\forall i \leq i' \leq j \leq j', \quad w(i,j) + w(i',j') \leq w(i',j) + w(i,j')$. Remarquons maintenant que $w(i,j) = w(0,j) - w(0,i)$. Ainsi $w(i,j) + w(i',j') = w(i',j) + w(i,j')$. De ces deux propriétés on en déduit que $\forall i \leq i' \leq j \leq j', \quad c(i,j) + c(i',j') \leq c(i',j) + c(i,j')$.

Ayant établi cette propriété, démontrons que $r(i,j-1) \leq r(i,j) \leq r(i+1,j)$. Supposons $i < j$. Montrons que $r(i,j-1) \leq r(i,j)$. Posons $c_k(i,j) = w(i,j) + c(i,k) + c(k+1,j)$. Remarquons tout d'abord que $r(i,j)$ est le plus grand indice k auquel l'on trouve le minimum, *ie.* $r(i,j) = max\{k : c_k(i,j) = c(i,j)\}$. Il suffit de prouver que pour $i < k \leq k' \leq j$ on a $c_{k'}(i,j-1) \leq c_k(i,j-1) \Rightarrow c_{k'}(i,j) \leq c_k(i,j)$. Or $c(k+1,j-1) + c(k'+1,j) \leq c(k'+1,j-1) + c(k+1,j)$. On ajoute à chaque membre $w(i,j-1) + w(i,j) + c(i,k) + c(i,k')$ pour obtenir $c_k(i,j-1) + c_{k'}(i,j) \leq c_{k'}(i,j-1) + c_k(i,j)$. L'implication est donc bien vérifiée. Ainsi $r(i,j-1) \leq r(i,j)$.

La preuve pour $r(i,j) \leq r(i+1,j)$ est similaire.

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 $O(n^2)$.

2.1 Nombre d'opérations en pire cas :

Le pire des cas se produit quand la profondeur de l'arbre est égal au nombre de sommets, c'est-à-dire dans le cas d'un *unbalanced tree*. Le coût de la recherche d'un élément dans l'arbre est en $O(n)$ et la structure de l'arbre se résume à une *linked list*. Déterminons le nombre d'opérations en pire cas en prenant comme indicateurs le nombre de comparaisons, le nombre d'affectations, le nombre d'additions et le nombre de soutractions pour l'algorithme de Bellman (d,i,k) :

1. Initialisation de la diagonale principale : 2 affectations n fois.
2. Boucle $1...n-1$: pas d'opérations.
3. Boucle $0...n-d+1$: 6 affectactions (3 dans la boucle, 3 dans *get_min*), 2 additions et 1 instruction conditionnelle.
4. Boucle $r(i+1,j)...r(i,j-1)$: 2 affectactions et 3 instructions conditionnelles.

Le programme itératif contient la boucle $r(i+1,j)...r(i,j-1)$ imbriquée dans $0...n-d+1$ imbriquée dans $1...n-1$ correspondant à la somme :

$$T(n) = \sum_{i=0}^{n-1} 2 + \sum_{d=1}^{n-1} \sum_{i=0}^{n-d+1} 9 \sum_{k=r(i+1,i+d)}^{r(i,i+d-1)} 5 = 2n + \sum_{d=1}^{n-1} \sum_{i=0}^{n-d+1} 45(r(i,i+d-1) - r(i+1,i+d) + 1)$$

$$\leq 2n + \sum_{d=1}^{n-1} \sum_{i=0}^{n-d+1} 45n$$

$$= 2n + \sum_{d=1}^{n-1} (n-d+2)n = 2n + n^2(n-1) - n\frac{n(n-1)}{2} + 2n(n-1) = O(n^3)$$

On en déduit un nombre d'opérations de $O(n^3)$. C'est cohérent. En effet, la distance entre $r(i+1,j)$ et $r(i,j-1)$ n'est pas constante dans le pire des cas mais l'est en moyenne.

Notons qu'avec un algorithme récursif la complexité est exponentielle. la relation de récurrence est de la forme $T(i,j) = \sum_k T(i,k-1) + \sum_k T(k+1,j) + 1$ et $T(i,i) = 1$. On trouve par substitution que $T(i,j) = \Theta(3^{j-i})$. Le calcul de $C(0,n-1)$ prend donc un temps en $\Theta(3^n)$.

2.2 Place mémoire requise :

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

2.3 Nombre de défauts de cache sur le modèle CO :

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.

Justification :

3 Compte rendu d'expérimentation (2 points)

3.1 Conditions expérimentales

Décrire les conditions permettant la reproductibilité des mesures : on demande la description de la machine et la méthode utilisée pour mesurer le temps.

3.1.1 Description synthétique de la machine :

indiquer ici le processeur et sa fréquence, la mémoire, le système d'exploitation. Préciser aussi si la machine était monopolisée pour un test, ou notamment si d'autres processus ou utilisateurs étaient en cours d'exécution.

3.1.2 Méthode utilisée pour les mesures de temps :

préciser ici comment les mesures de temps ont été effectuées (fonction appelée) et l'unité de temps ; en particulier, préciser comment les 5 exécutions pour chaque test ont été faites (par exemple si le même test est fait 5 fois de suite, ou si les tests sont alternés entre les mesures, ou exécutés en concurrence etc).

3.2 Mesures expérimentales

Compléter le tableau suivant par les temps d'exécution mesurés pour chacun des 6 benchmarks imposés (temps minimum, maximum et moyen sur 5 exécutions)

	coût du patch	temps min	temps max	temps moyen
benchmark1				
benchmark2				
benchmark3				
benchmark4				
benchmark5				
benchmark6				

FIGURE 1 – Mesures des temps minimum, maximum et moyen de 5 exécutions pour les 6 benchmarks.

3.3 Analyse des résultats expérimentaux

Donner une réponse justifiée à la question : les temps mesurés correspondent ils à votre analyse théorique (nombre d'opérations et défauts de cache) ?