Rapport de TP 4MMAOD : Génération d'ABR optimal

ARGENTO Adrien Groupe 1
VINCENT Nicolas Groupe 1

29 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.

Nous avons choisi de représenter la matrice c par un tableau 1d contenant les lignes de la matrice triangulaire supérieures ordonnées par ordre croissant. Cette représentation est optimale pour l'itération (i, j, k). Nous avons aussi testé de stocker les diagonales de la matrice triangulaire supérieure, ordonnées de la diagonale inférieure jusqu'à $c(0, n-1)$. On divise alors par deux le temps de calcul de l'algorithme (d, i, k). Les algorithmes (i, j, k) et (j, i, k) prennent le même temps de calcul, ils sont plus longs qu'avec la première méthode de stockage. Cette deuxième méthode de stockage pourrait être plus efficace si l'on parallélise le calcul des coefficients de chaque diagonale. Ce n'est pas celle qui a été retenue dans la suite du TP.

Pour observer les différences entre les deux méthodes de stockage, modifiez les macros de *src/bellman.h*.

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

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 CO.

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 $\forall [i,j] \subseteq [i',j'], w(i,j) \leq w(i',j')$. 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 par induction sur $j'-i$ que $\forall i \leq i' \leq j \leq j', c(i,j) + c(i',j') \leq c(i',j) + c(i,j')$.

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.* $r(i,j) = max\{k : c_k(i,j) = c(i,j)\} = max\{k : \forall k', c_{k'}(i,j) \leq c_k(i,j)\}$. 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,

$$\sum_{i=1}^{n}\sum_{j=1}^{n} r(i+1,j) - r(i,j-1) + 1 = n^2 + \sum_{i=1}^{n}\sum_{j=1}^{n} r(i+1,j) - \sum_{i=1}^{n}\sum_{j=1}^{n} r(i,j-1)$$

$$= n^2 + \sum_{i=2}^{n+1}\sum_{j=1}^{n} r(i,j) - \sum_{i=1}^{n}\sum_{j=0}^{n-1} r(i,j) \leq n^2 + \sum_{i=1}^{n+1}\sum_{j=1}^{n} r(i,j) - \sum_{i=1}^{n}\sum_{j=1}^{n-1} r(i,j)$$

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

.

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 :

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

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

$$= 2n + 45n\sum_{d=1}^{n-1}(n-d+2) = 2n + (45n^2(n-1) - 45n\frac{n(n-1)}{2} + 90n(n-1)) = O(n^3)$$

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 $2\frac{n(n+1)}{2} + n + 2n = O(n^2)$.

Un fois le calcul de l'arbre terminé, on peut libérer les arrays des coûts et des probabilités partielles.

2.3 Nombre de défauts de cache sur le modèle CO :

L'algorithme de Bellman (d,i,k) a des très mauvaises localités temporelle et spatiale. On mesure en effet 38.2% de miss-rate sur le cache L1. Prenons le calcul de la d-ième diagonale. Toutes les entrées précalculées à gauche et en dessous de la diagonale d sont accédées.

Les deux autres algorithmes calculent les sous-problèmes en suivant un ordre différent. Pour l'algorithme (i,j,k), les sous-problèmes de la forme $i \dots j$ sont résolus avant ceux de la forme $i-1 \dots j$ tandis que l'algorithme (j,i,k) résoud les sous-problèmes de la forme $i \dots j$ sont résolus avant ceux de la forme $i \dots j+1$. Ces deux solutions ont de meilleurs performances car elles ont une meilleur localité du cache. En effet, dans l'algorithme (i,j,k), lorsqu'une ligne est calculée, tous les éléments de cette ligne sont accédés à la suite.

Concernant le nombre de défauts de cache :

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 $Q(n,L,Z) = \frac{n(n+1)+2n}{L} = O(\frac{n^2}{L})$.

Prenons l'algorithme (i,j,k). On suppose que l'exécution de la boucle interne *for (int j = i + 1; j < n; ++j)* tient en cache (ie. $Z \gg \frac{n(n+1)}{2} + n + n$ pour c, $sommes_p$ et r). Cela revient à dire que pour le calcul d'une ligne de c, le cache peut contenir toutes les lignes en dessous ainsi que la ligne que l'on cherche à calculer en plus d'une ligne de r et de $sommes_p$. Le tableau *probabilities* est parcouru par ligne donc $\frac{n}{L}$ défauts de cache en lecture. Le tableau $sommes_p$ est parcouru par ligne donc $\frac{n}{L}$ défauts de cache en lecture. La matrice c (de taille $\frac{n(n+1)}{2}$) tient dans le cache et supposons que $Z \ll n(n+1) + n + n$. Les lignes de r s'accumulent dans le cache alors qu'on n'en a plus l'utilité. Pour faire de l'espace aux coefficients de c, les premières entrées de c présentes dans le cache vont être supprimées alors qu'elles sont nécessaires au calcul. On aura donc des défauts de cache supplémentaires sur c. On aura donc $\frac{n^2(n+1)}{2L}$ défauts de cache sur c en lecture et écriture. La matrice r est parcourue par ligne donc $\frac{n(n+1)}{2L}$ défauts de cache en écriture.

Pour l'algorithme (j,i,k), on intervertit lignes et colonnes. On a donc 1 défaut par élément pour r soit $\frac{n(n+1)}{L}$ défauts de cache. Pour c on obtient $\frac{n^2(n+1)}{2}$ défauts de cache.

Reprenons l'algorithme (i,j,k) et supposons que $Z \ll n$. La boucle interne sur k génère des défauts de cache capacitifs sur c mais pas sur r. $Q(n,L,Z) = \frac{n(n+1)}{2L} + \frac{n^2(n+1)}{2L} + 2\frac{n^2}{L} = O\left(\frac{n^3}{2L}\right)$.

L'algorithme (d,i,k) itérant diagonale par diagonale réalise $O(n^3)$ défauts de cache pour $Z \ll n$ (raisonner comme précédemment). On a probablement le même nombre de défauts de cache (en approximation O) si $Z \ll n(n+1) + n + n$.

3 Compte rendu d'expérimentation (2 points)

3.1 Conditions expérimentales

3.1.1 Description synthétique de la machine :

La machine sur laquelle ont été prises les mesures est d'architecture x86_64 Little Endian, possède 4 coeurs Intel(R) Core(TM) i5-2520M CPU @ 2.50GHz pour 2 threads par coeur et 2 coeurs par socket, une fréquence du CPU mesurée de 1253.326 MHz pour un minimum de 800 MHz et un maximum de 3200 MHz, les caches L1d et L1i de 64 KiB chacun, le cache L2 de 512 KiB et le cache L3 de 3 MiB. A cela s'ajoutent 6 GB de RAM DDR3-SODIMM, 1333 MT/s, répartis en 2 barrettes de 4 GB et 2 GB.

La machine n'était pas mobilisée pour un test, Firefox et Codium s'exécutaient en tâche de fond.

3.1.2 Méthode utilisée pour les mesures de temps :

Les mesures sont effectuées en exécutant le script *time_benchmarks.sh*. Les trois programmes Bellman ((d, i, k) non parallélisé) sont alors exécutés successivement 10 fois de suite sur le même fichier, dans l'ordre (d, i, k), (i, j, k) puis (j, i, k). Nous avons observé que l'ordre des fonctions n'influence pas le résultat. On utilise le module *timing.c* implémentant un décorateur autour des fonctions Bellman avec la fonction *clock()* de *time.h*. Les résultats sont donnés en secondes.

Nous avons réalisé 5 appels non consécutifs à ce script afin d'obtenir des mesures indépendantes de l'état du processeur.

L'exécution de *valgrind* sur *benchmark6* donne 150,115,592 bytes alloués, soit environ 150 MB. On obtient un nombre bien plus élevé si l'on choisit de stocker les matrices triangulaires inférieures de zéros.

3.2 Mesures expérimentales

	Bellman (d, i, k)			Bellman (i, j, k)			Bellman (j, i, k)		
	temps min	temps max	temps moyen	temps min	temps max	temps moyen	temps min	temps max	temps moyen
benchmark1	1e-06	6e-06	2.5e-06	1e-06	3e-06	1.62e-06	1e-06	3e-06	1.48e-06
benchmark2	2e-06	6e-06	3.28e-06	2e-06	4e-06	2.3e-06	2e-06	5e-06	2.54e-06
benchmark3	0.0110	0.0162	0.0123	0.0043	0.0066	0.0046	0.0080	0.0101	0.0086
benchmark4	0.037	0.100	0.062	0.019	0.031	0.022	0.030	0.067	0.046
benchmark5	0.088	0.215	0.161	0.045	0.082	0.056	0.072	0.192	0.126
benchmark6	0.489	0.867	0.641	0.166	0.247	0.180	0.370	0.646	0.484

FIGURE 1 – Mesures des temps minimum, maximum et moyen de 50 exécutions pour les 6 benchmarks et les 3 programmes Bellman.

3.3 Analyse des résultats expérimentaux

Sans surprise, l'algorithme de Bellman (d, i, k) (38.2% cache-misses) est le moins performant tandis que l'algorithme (i, j, k) (9.9% cache-misses), itérant ligne par ligne, est le plus performant. Le dernier algorithme, (j, i, k) (28.4% cache-misses), est légèrement plus performant que (d, i, k).

Nous avons parallélisé l'algorithme (d, i, k) sans obtenir de résultats visibles. Il est 2 fois plus lent ce qui est très étonnant. Le calcul en parallèle de chaque coefficient d'une même diagonale devrait accélérer l'exécution. L'optimisation des défauts de caches semble primer ici sur le parallélisme.