Rapport de TP 4MMAOD : Génération d'ABR optimal

GARAT Cassius
HOURLIER Simon

April 14, 2017

1 Principe de notre programme

Nous avons décidé d'utiliser une méthode itérative pour calculer les différents coûts et racines de l'arbre optimal, puis une méthode récursive pour construire l'arbre optimal au format souhaité, à partir des valeurs calculées.

Pour calculer le nombre moyen de comparaisons, nous avons utilisé l'équation de Bellman suivante :

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

dans cette équation, p_i représente la fréquence d'apparition du symbole e_i. Nous avons :

$$C(i,i) = p_i$$

Nous avons utilisé une matrice pour mémoiser les valeurs calculées grâce à cette équation. La case d'indices i et j de cette matrice représente l'arbre optimal contenant les éléments e_i à e_j. Cette matrice contient des structures : le premier élément de cette structure représente le coût de l'arbre optimal et le deuxième est l'indice de la racine de cet arbre.
Ainsi, la case située au coin supérieur droit de cette matrice contient notamment le coût total de l'arbre (par coût, nous entendons nombre moyen de comparaisons).

Pour la deuxième partie, le principe de la fonction récursive est le suivant : imaginons que la racine optimale entre les éléments e_i et e_j soit d'indice k. On cherche alors la racine optimale pour les éléments e_i à e_{k-1} d'une part, et pour les éléments e_{k+1} à e_j d'autre part. On applique le même principe récursivement.

2 Analyse du coût théorique

2.1 Nombre d'opérations en pire cas :

Le nombre d'opérations en pire cas est en $O(n^2)$.

Justification : Le programme itératif contient une première boucle $q = 1...n-1$, une deuxième $i = 0...n-q$ et une dernière boucle interne $tmp = r(i,i+q-1)...r(i+1,i+q)$ où $k = i+q$. Une seule addition est opérée dans cette dernière boucle. Il y a donc $r(i+1,j) - r(i,j-1) + 1$ opérations pour chaque i. Ainsi le nombre d'opérations dans la boucle i est :

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

En observant que $r(i+1,i+q) = r(i+1,(i+q-1)+1)$ on simplifie très facilement la somme :

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

L'écart entre les deux racines optimales ne dépassera pas n, et $q \geq 1$ donc la somme peut être majorée par $2n$. La boucle q tourne n fois, donc le nombre d'opérations est bien en $O(n^2)$.

La deuxième fonction, permettant de construire l'arbre pour l'afficher sur le terminal, ne fait pas d'opération et est appelée n fois. Nous pouvons compter les comparaisons : il y en a au plus 3 par appel, soit $3n$ au total.

2.2 Place mémoire requise :

La place mémoire requise est en $\Theta(n^2)$.

Justification : Pour stocker les coûts des arbres optimaux intermédiaires, nous utilisons une matrice de taille n^2 (nous y stockons une structure contenant deux éléments). Nous stockons les fréquences des symboles dans un tableau de taille n. Nous utilisons également un tableau de taille n pour stocker les sommes des fréquences des symboles (la case d'indice i contient la somme des fréquences des éléments jusqu'à e_i).

2.3 Nombre de défauts de cache sur le modèle CO :

Le nombre de défauts de cache devrait être proche de n^2 lorsque $Z < nL$.

Justification : Imaginons que nous souhaitions calculer le coût de l'arbre optimal entre les éléments e_i et e_j. On fixe alors une racine d'indice k telle que

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

Pour chacune de ces racines, en appliquant l'équation de Bellman, on récupère le résultat dans la case de la matrice d'indices $(i, k-1)$ et celui de la case d'indices $(k+1, j)$. On opère donc deux lectures. On peut observer que pour i et j donnés, les lectures auront exclusivement lieu dans les cases situées dans la ligne d'indice i de la matrice, et dans la colonne j. Le nombre de défauts de cache dans la ligne i sera alors

$$(r(i+1, j) - r(i, j-1) + 1)/L$$

et

$$r(i+1, j) - r(i, j-1) + 1$$

dans la colonne j.
De façon analogue à la section `2.1`, nous arrivons à calculer une somme :

$$\sum_{i=0}^{n-q}((r(i+1, i+q) - r(i, i+q-1) + 1)(1 + 1/L))$$

Soit :

$$\sum_{i=0}^{n-q}(r(i+1, i+q) - r(i, i+q-1) + 1) + \sum_{i=0}^{n-q}(r(i+1, i+q) - r(i, i+q-1) + 1)/L$$

Ce qui, de la même manière qu'à la section `2.1`, nous amène à une majoration par $2n + 2n/L$. Ceci étant répété n fois, nous arrivons à environ $2n^2 + 2n^2/L$ défauts.

3 Compte rendu d'expérimentation

3.1 Conditions expérimentaless

3.1.1 Description synthétique de la machine :

Processeur et fréquence : Intel(R) Celeron(R) CPU N2840 @ 2.16GHz
Système d'exploitation : Ubuntu 16.04
Mémoire vive : 4 gigabits

Pour les tests, nous avons fermé toute application susceptible de monopoliser des ressources.

3.1.2 Méthode utilisée pour les mesures de temps :

Pour les mesures de temps, nous avons utilisé la fonction `clock()` de la bibliothèque time. Nous avons mesuré un intervalle de temps : le premier déclencheur était placé au début du main, juste après l'ouverture du fichier contenant les fréquences d'apparition (avant la lecture de celui-ci, pour insérer les fréquences dans un tableau). Le deuxième était placé à la fin du main, après les désallocations mémoire.
L'unité choisie était la seconde et la fonction utilisée nous garantissait une précision à la microseconde.
Concernant l'exécution des tests, ceux-ci étaient lancés 5 fois de suite. Les tests sur les benchmarks suivants étaient lancés directement après à l'aide d'un script.

3.2 Mesures expérimentales

	coût du patch	temps min	temps max	temps moyen
benchmark1	5	0.000046	0.000048	0.000047
benchmark2	10	0.000050	0.000054	0.000052
benchmark3	1000	0.070408	0.071777	0.071183
benchmark4	2000	0.501098	0.509637	0.504611
benchmark5	3000	1.709092	1.719507	1.716065
benchmark6	5000	5.883850	5.913964	5.897850

Figure 1: Mesures des temps minimum, maximum et moyen de 5 exécutions pour les 6 benchmarks. Temps mesuré en secondes.

NB : nous avons testé notre programme sur une machine Windows à processeur i5 cadencé à 2,6 GHz (TurboBoost à 3,2 GHz). Le temps moyen mesuré pour le dernier benchmark était d'environ 1,3 secondes.

3.3 Analyse des résultats expérimentaux

Concernant les défauts de cache, nous avons bien observé que des éléments contigus provoquent moins de défauts de cache. Pour stocker les coûts et les racines des arbres optimaux intermediaires, nous avons choisi d'utliser une structure, pour ainsi n'utiliser qu'une matrice. Cette matrice stocke alors des données de taille double par rapport à l'implémentation naturelle (une matrice pour les coûts et une matrice pour les racines). Nous avons constaté qu'il y avait moins de défauts de cache de cette façon, puisque les données sont contiguës. Le programme tournait donc plus rapidement grâce à cette structure. Par exemple, le dernier benchmark durait 1 seconde de plus sans la structure sur le système d'exploitation explicité plus haut (Ubuntu).
Au delà de cette observation, nous avons pu vérifier que le nombre de défauts de cache semblait effectivement être proche de n^2, aussi bien au premier qu'au dernier niveau de cache, grâce à l'outil `cachegrind` de valgrind.

	D1 misses	LL misses
benchmark1	3,175	3,797
benchmark2	3,195	3,816
benchmark3	2,739,769	445,055
benchmark4	11,305,871	1,867,060
benchmark5	25,636,781	5,144,110
benchmark6	71,406,606	43,438,544

Figure 2: Nombre de défauts de cache observés au premier et dernier niveau de cache, d'après l'outil cachegrind.

Expérimentalement, nous pouvons constater en reprenant les valeurs mesurées dans le tableau de la section précédente, que le temps mesuré moyen croît de façon quadratique, plus proche de $2n^2$ que de n^2. Les résultats de l'expérimentation sont donc assez proche de ce à quoi nous nous attendions en théorie, puisque le nombre de défauts de cache et le nombre d'opérations étaient en $O(n^2)$ en théorie.
Pour aller plus loin dans l'optimisation, il serait intéressant d'implémenter notre programme en Cache Oblivious.