Rapport de TP 4MMAOD : Génération d'ABR optimal

Maxime Deloche
Ludovic Carré
Equipe Teide 4
2ème partie du rendu

10 novembre 2016

1 Principe de notre programme (1 point)

Nous avons implémenté les 2 méthodes (récursive et itérative) pour le calcul du nombre moyen optimal de comparaisons (fonction `avg_comp_rec et avg_comp_iter` de `averagedepth.c`, appelée par `get_avg`). La version itérative est largement plus rapide, et le code la version récursive est commenté.

C'est l'implémentation directe de notre première version de la formule de Bellman, à laquelle nous avons ajouté des optimisations détaillées dans la partie suivante (nous nous sommes rendus compte d'erreurs de calcul dans la 2e version de l'équation après le rendu de la 1e partie du rapport).

Une fois la fonction `avg_comp_iter` appelée, la fonction `build_tree` construit, à partir du tableau de mémoïsation, l'arbre optimal, et le renvoit.

Enfin, la fonction `treetoarray` écrit, à partir de cet arbre, le code C attendu sur la sortie standard (et peut au passage afficher la profondeur moyenne de cet arbre optimal, stockée dans l'objet `Tree`).

Nous avions commencé par une implémentation en Python. Nous nous sommes cependant vite rendus compte de la difficulté à effectuer le moindre calcul d'optimisation et de la lenteur du langage, et avons donc décidé de passer à du C. Nous avons tout d'abord fait la version récursive, qui était elle aussi très lente. Nous avons fini par réaliser la version itérative qui surpasse de loin la version récursive (cf. partie Performances ci-dessous).

2 Analyse du coût théorique (2 points)

2.1 Choix d'implémentations

— Le tableau `comp_array` est un tableau de mémoïsation : à la position `(i, j)`, il stocke l'indice de la racine choisie pour le sous-arbre formé des noeuds d'indices `i` à `j` (les deux inclus), et le nombre de comparaisons moyen de cet arbre optimal.
Nous n'avions pas besoin de stocker l'arbre en entier, seulement la racine : en effet, si on sait que la racine est d'indice `k`, on trouve son fils gauche (resp. droit) dans la case `(i, k)` (resp. `(k+1, j)`).
C'est ce tableau qui est parcouru par la fonction `build_tree` pour reconstituer l'arbre optimal.

— Nous avons à chaque appel besoin de sommer les probabilités de `i` à `j`. A la lecture du fichier de données, nous remplissons un tableau `proba_sums` stockant les sommes des probas des indices 0 à `i`. Nous accédons donc ensuite à la somme de probabilités de `i` à `j` en calculant `proba_sums[j] - proba_sums[i]` (fonction `proba_sum`), donc en coût constant.

— Nous avons ensuite réduit le nombre de racines à tester en utilisant l'"optimisation de Knuth" (nous avons trouvé cette information sur la page Wikipedia des Binary Search Trees).

2.2 Nombre d'opérations en pire cas

Justification On a les coûts suivants pour les différentes boucles de notre fonction :

— $T_1(n) = 2n$: Première boucle qui initialise la diagonale.

— $T_2(n) = \sum_{l=2}^{n} T_3(n, l)$: Seconde boucle qui indique les diagonales.

— $T_3(n, l) = \sum_{i=0}^{n-l} O(1) + T_4(n, l, i)$: Troisième boucle pour parcourir les cases.

— $T_4(n,l,i) = \sum_{r=i+l-1}^{n} O(1)$: Dernière boucle optimisée avec Knuth.

On obtient le résultat suivant pour $T(n)$ le nombre d'opérations pour n valeurs :

$$T(n) = 2n + \sum_{l=2}^{n} \left(\sum_{i=0}^{n-l} \left(\theta(1) + \sum_{r=i+l-1}^{n} \theta(1) \right) \right)$$

$$T(n) = 2n + \sum_{l=2}^{n} \left(\sum_{i=0}^{n-l} \left(\theta(1) + (n-i-l+1) * \theta(1) \right) \right)$$

$$T(n) = 2n + \sum_{l=2}^{n} \left(\sum_{i=0}^{n-l} \theta(n-i-l) \right)$$

$$T(n) = 2n + \sum_{l=2}^{n} \theta((n-l)^2)$$

$$T(n) = 2n + \sum_{l=2}^{n} \theta(n^2) + \sum_{l=2}^{n} \theta(l^2) - \sum_{l=2}^{n} \theta(2nl)$$

$$T(n) = 2n + \theta((n-2+1)n^2) + \theta(\frac{n(n+1)(2n+1)}{6}) - \theta(2n(\frac{(n-1)(2+n)}{2}))$$

$$T(n) = \theta(n^3)$$

Notre fonction opère donc en $\theta(n^3)$ en pire cas et grace à l'amélioration de Knuth, elle est en $\theta(n^2)$ en moyenne.

2.3 Place mémoire requise

La place mémoire requise est en $\theta(n^2)$, avec n le nombre d'éléments de l'arbre.

Justification La plus grande place mémoire nécessaire vient du tableau de mémoïsation, qui stocke deux valeurs pour chaque couple d'indices, et a donc un coût de l'ordre de n^2 (en pratique 2 fois moins, puisque l'on ne stocke quelque chose que lorsque $low\_index < high\_index$). Les tableaux de probabilités coûtent une taille n en mémoire, de même que l'arbre, ce qui est donc négligeable devant n^2.

2.4 Nombre de défauts de cache sur le modèle CO

Mesure des défauts de cache sur le benchmark 6 : 3.2% sur le cache D1 (le plus petit).

Nous avons manqué de temps pour effectuer une optimisation des défauts de cache, mais on pourrait optimiser le tableau de mémoïsation. En effet, il est parcouru en diagonale : en utilisant des indices différents, on peut sûrement effectuer des parcours en ligne et diminuer ainsi les cache miss.

3 Compte rendu d'expérimentation (2 points)

3.1 Conditions expérimentales

3.1.1 Description synthétique de la machine

Les tests ont été effectués sur une machine du 3e étage (bâtiment E) :
— **Processeur** : Intel(R) Core(TM) i5-6500 CPU @ 3.20GHz
— **RAM** : 32 GB
— **Système d'exploitation** : CentOS Linux release 7.2.1511
— Machine monopolisée pour le test

3.1.2 Méthode utilisée pour les mesures de temps

Les mesures de temps peuvent être effectuées en lançant le script `time_test.sh`, qui affiche le résultat des appels à `time` sur la sortie standard. Ce script exécute pour chaque benchmark le même test 5 fois de suite. Nous avons ensuite entré ces valeurs dans un tableur et tracé le graphe ci-dessous. Le fichier `execution_times.txt` contient un résultat de `time_test.sh`, qui nous a servi à remplir le tableau ci-dessous.

3.2 Mesures expérimentales

Les mesures ont été effectuées sur la fonction itérative `avg_comp_iter`.

	Taille du test	temps min	temps max	temps moyen
benchmark1	5	0.002	0.005	0.015
benchmark2	10	0.002	0.003	0.002
benchmark3	1000	0.029	0.061	0.044
benchmark4	2000	0.128	0.155	0.136
benchmark5	3000	0.279	0.321	0.302
benchmark6	5000	0.890	1.011	0.937

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

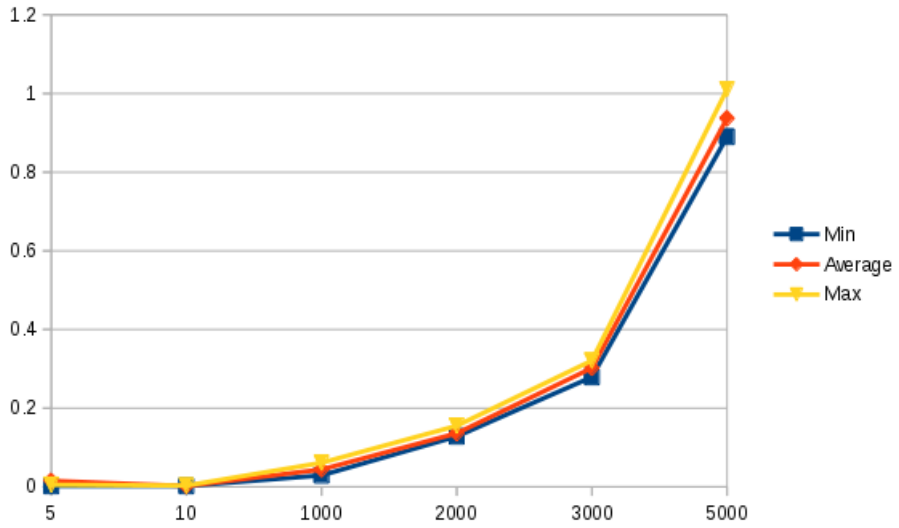

FIGURE 2 – Temps d'exécution minimum, moyen et maximum en fonction du nombre de noeud

3.3 Analyse des résultats expérimentaux

On obtient des résultats cohérents avec notre analyse théorique lors des test expérimentaux.
Malgré notre implémentation peu optimale en terme d'utilisation du cache, le pourcentage de cache miss est assez faible.