Rapport de TP 4MMAOD : Génération d'ABR optimal

BAYARD Guillaume (IF2)
BERTHON Christophe (IF2)

10 novembre 2016

1 Principe de notre programme

L'affichage des noeuds de l'arbre par la fonction *ABR* se fait récursivement mais le calcul même de ses noeuds se fait pars la fonction itérative *ABRopt*. La troisième boucle impliquée $s = i + 1...j$ détermine, grâce à la formule de récurrence déterminée grâce à l'équation de Bellman la racine de l'ABR optimal contenant les éléments numérotés i à j. La boucle for stocke le numéro de la racine courante de l'indice s dans $racine[ij]$ et ceci chaque fois qu'elle trouve une racine permettant un coût de recherche plus faible.
Les sommes partielles de probabilités, nécessaires pour calculer le minimum des coûts, ont au préalable été calculées puis stockées dans une matrice n^2n dans la fonction *somme.*

2 Analyse du coût théorique

2.1 Nombre d'opérations en pire cas :

Justification : *Le programme* lecture_fichier qui stocke dans un tableau de float les fréquences d'apparition des éléments dans le fichier entré en argument contient deux boucles $i = 0..n-1$, correspondant à la somme $2\sum_{k_1=1}^{n-1} = 2(n-1)$
Le programme *somme*, calculant les sommes partielles des probabilités, contient les boucles $i = 0..n-1$ et $j = i+1..n-1$ correspondant donc à la somme

$$\begin{aligned} &= \sum_{i=0}^{n-1}\left(1 + \sum_{j=i+1}^{n-1} 2\right) \\ &= \sum_{i=0}^{n-1}(1 + 2(n-1-i)) \\ &= n + 2n^2 - 2n - \frac{n(n-1)}{2} \\ &= \frac{2n^2 - n}{2} \end{aligned}$$

Le programme itératif *ABRopt* contient les boucles imbriquées $k = 1..n-1$, $i = 0 = n-k-1$ et $s = i+1..j$. De plus, dans le pire des cas, la conditionn $t < cout[i][j]$ dans la boucle $s = 1..i+k+1$ est systématiquement rempli et les deux affections qu'elle contient sont donc effectuées. On obtient donc la somme que l'on calcule

directement

$$
\begin{aligned}
&= \sum_{k_1=1}^{n-1} \sum_{i=0}^{n-k-1} \left(1 + \sum_{s=i+1}^{j} 3\right) \\
&= \sum_{k_1=1}^{n-1} \sum_{i=0}^{n-k-1} (1 + 3(j-i)) \\
&= \sum_{k_1=1}^{n-1} \sum_{i=0}^{n-k-1} (1 + 3(i+k+1-i)) \\
&= \sum_{k_1=1}^{n-1} \sum_{i=0}^{n-k-1} (3k+4) \\
&= \sum_{k_1=1}^{n-1} (n-k)\,(3k+4) \\
&= \sum_{k_1=1}^{n-1} -3k^2 - 4k + 3nk + 4n \\
&= 4n(n-1) + (3n-4)\frac{n(n-1)}{2} - \frac{n(n-1)(2n-1)}{2} \\
&= 4n(n-1) + (3n-4-2n+1)\frac{n(n-1)}{2} \\
&= (8+3n-4-2n+1)\frac{n(n-1)}{2} \\
&= (n+5)\frac{n(n-1)}{2} \\
&= \frac{n(2n-1)(n+5)}{2} \\
&= \frac{n(2n-1)(n+5)}{2}
\end{aligned}
$$

Le coût total, au pire cas, est donc de $2(n-1) + \frac{2n^2-n}{2} + \frac{n(2n-1)(n+5)}{2}$ opérations. L'ordre du coup est donc $\Theta(n^3)$.

2.2 Place mémoire requise :

Justification : Les principales allocations mémoire ont lieu
- dans la fonction *somme*, où un pointeur double de flottants est alloué afin de stocker les sommes partielles des probabilités dans une matrice de taille $n.n$
- dans la fonction *ABRopt* où un pointeur double de flottants est alloué afin de stocker les coûts de recherche dans une matrice $n + 1.n + 1$
- dans la fonction /em ABRopt où un pointeur double d'entiers est alloué afin de stocker les racines des sous-arbres optimaux dans une matrice de taille $n.n$.

Toutes les autres allocations de pointeurs du programmes sont négligeables.

2.3 Nombre de défauts de cache sur le modèle CO :

Justification :

3 Compte rendu d'expérimentation

3.1 Conditions expérimentaless

3.1.1 Description synthétique de la machine :

La machine utilisée pour les tests était l'ordinateur ENSIPC369 de la salle 212. Son processeur était un i5-4590, de fréquence 3,3 GhZ de mémoire cache 6144kB, de mémoire RAM 16 Giga, de système d'exploitation

Linux-GNU. Une utilisation légère de firefox a été faite en parallèle des tests.

3.1.2 Méthode utilisée pour les mesures de temps :

fonction appelée : *time -p ./bin/computeABROpt n benchmarks/benchmark.in* . Unité de temps : seconde. Chaque test a été effectué 5 fois de suite.

3.2 Mesures expérimentales

	temps min	temps max	temps moyen
benchmark1	0,001	0,004	0,0022
benchmark2	0,001	0,003	0,0022
benchmark3	1,095	1,14	1,109
benchmark4	9,86	10,33	10,002
benchmark5	49,03	50,82	50,2
benchmark6	267,20	278,28	274,59

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

3.3 Analyse des résultats expérimentaux

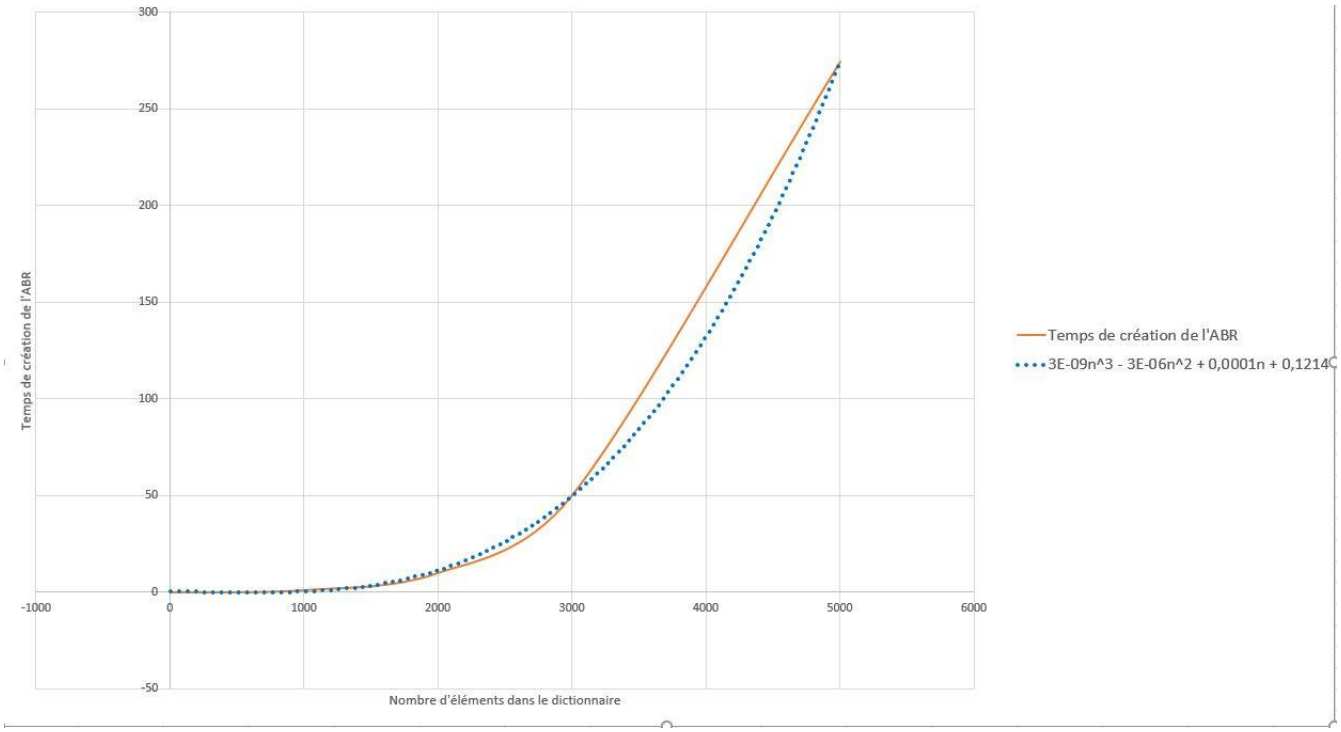

FIGURE 2 – Temps de création de l'arbre en fonction du nombre d'éléments présents dans le dictionnaire

On voit que la courbe des temps obtenus lors des tests sur les benchmark est très proche de la courbe d'équation $n \mapsto 3.10^{-9}n^3 - 3.10^{-6}n^2 + 10^{-4}n + 0,1214$, ce qui tend à valider l'hypothèse que le nombre d'opérations en pire cas est en $O(n^3)$