TP 4MMAOD : ABR Optimal

De Bollivier Enea
Duc Melchior

14 avril 2017

1 Equation de Bellman

Pour obtenir l'équation de Bellman correspondant à notre problème il faut, dans un premier temps, justifier qu'un arbre binaire de recherche optimal est composé de sous arbres optimaux. En effet si on peut améliorer un sous-arbre, on peut diminuer son poids. En réinsérant cet arbre dans l'arbre initial on obtient un arbre de poids plus faible, l'arbre initial n'était donc pas optimal. Ainsi les sous arbre d'un arbre optimal sont optimaux.

On note :

- $T_{i,j}$: ABR optimal pour les éléments e_{i+1},...,e_j
- p_i : Probabilité de rechercher l'élément e_i
- $r_{i,j}$: racine de l'ABR T_i, j
- $c_{i,j}$: coût (nombre moyen de comparaisons) de l'ABR T_i, j
- $w_{i,j} = \sum_{k=i+1}^{j} p_k$

Si e_k la racine de T_i, j, $T_{i,k-1}$ son sous arbre gauche et $T_{k,j}$ son sous arbre droit on a :

$$c_{i,j} = (c_{i,k-1} + w_{i,k-1}) + p_k + (c_{k,j} + w_{k,j})$$

En effet les sous arbres droit et gauche sont placés un étage plus bas, il faut ajouter pour chaque arbre la somme ($w_{i,k-1}$ et $w_{k,j}$) des fréquences de leurs éléments.

$$c_{i,j} = (w_{i,k-1} + p_k + w_{k,j}) + c_{i,k-1} + c_{k,j}$$

D'où l'équation de Bellman :

$$c_{i,j} = w_{i,j} + c_{i,k-1} + c_{k,j}$$

Il faut donc choisir l'élément e_k qui minimise cette égalité.

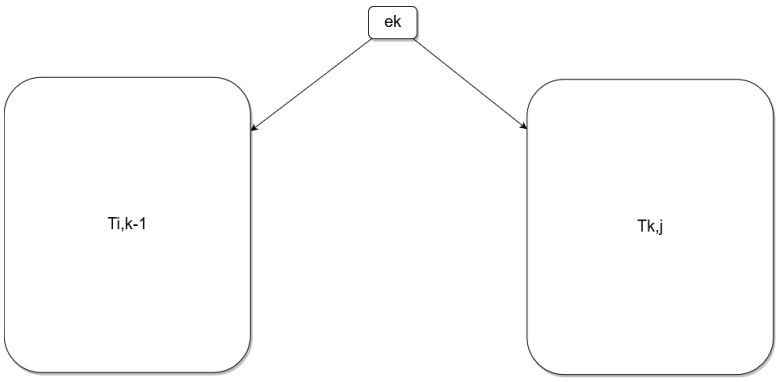

Schéma explicatif de l'arbre T_i, j

2 Principe de notre programme

Mettre ici une explication brève du principe de votre programme en précisant la méthode implantée (récursive, itérative) et les choix effectués (notamment pour l'ordonnancement des instructions).

Pour développer notre TP nous avons choisit de coder en Ada afin de pouvoir récupérer un package de gestion d'arbre que nous avions déjà développé. Notre programme est divisé en plusieurs parties.

Mise en place du programme : le but est d'appliquer la formule de Bellman précédente afin de construire une matrice $R_{i,j}$ contenant pour tout couple (i,j) tel que i<= j une racine optimale pour l'arbre $T_{i,j}$. La construction de cette matrice repose sur les matrice $C_{i,j}$ (des coûts) et la matrice $W_{i,j}$ (contenant les sommes contigues de probabilités).

```
procedure Mise_En_Place_Optimal(Nom_Fichier : in String; n : in Integer;
                                R : out T_Int_access;NBELEM : out Integer) is
  Fichier : Ada.Streams.Stream_IO.File_Type;
  Flux : Stream_Access;
  data_read : Character;
  data_value : Integer:=0;
  add : Boolean;
  data_sum : Integer:=0;
  S : Float; --somme de tous les entiers contenus dans le fichier (sommes des proba)
  P : array(0..n) of Float; -- Proba
  C : T_Float_access:= new T_Float(0..n,0..n) ;
            -- C(i,j) = cout de l arbre T(i,j)
  W : T_Float_access:= new T_Float(0..n,0..n) ;
            -- W(i,j) = somme des p(k) pour k allant de i a j-1
  j : Integer;
  Cmin : Float; -- Cmin
  m : Integer; -- valeur de k minimisant C(i,k-1)+C(k,j)
```

Construction de l'arbre optimal : Une fois que la matrice R est construite on se sert du fait qu'un arbre optimal a pour sous arbre des arbres optimaux, ainsi par appel récursif on vient chercher dans R la racine optimal de l'ensemble des sous arbres pour obtenir notre arbre optimal.

```
function Construit_Abr_Optimal(i : Integer; j : Integer; R : in T_Int_access)
                                                        return Arbre is
  A : Arbre ;
begin
  A := creer_arbre(R(i,j)-1);
  if(i < R(i,j)-1) then
    A.filsgauche:= Construit_Abr_Optimal(i, R(i,j)-1,R);
  end if;
  if(R(i,j)<j) then
    A.filsdroit:= Construit_Abr_Optimal(R(i,j),j,R);
  end if;
  return A;
end Construit_Abr_Optimal;
```

Parcours de l'arbre : Nous obtenons un arbre optimal, pour pouvoir l'afficher comme demandé il faut parcourir l'arbre afin d'obtenir le tableau souhaité qui est ensuite affiché. Ce parcours s'effectue de façons récursive, on place dans le tableau les éléments correspondants.

Affichage : Il suffit alors d'afficher le tableau précédent avec la mise en page souhaitée.

3 Analyse du coût théorique

Nous allons donner ici l'analyse du coût théorique de notre programme en fonction du nombre n d'éléments dans le dictionnaire. Pour chaque coût, donner la formule qui le caractérise (en justifiant brièvement pourquoi cette formule correspond à votre programme), puis l'ordre du coût en fonction de n en notation Θ de préférence, sinon O.

3.1 Nombre d'opérations en pire cas :

Dans le pire cas notre programme possède un complexité en $oollO(n^3)$.

Justification : La mise en place itérative contient une première double boucles d'initialisation comportant n^2 opérations.

Il y a ensuite une boucle correspondant à la somme

$$T(n) = \sum_{i=0}^{n} i(n-i) = \frac{n(n+1)}{2}(n - \frac{2n+1}{3}) = O(n^3)$$

somme que nous avons calculée (ou majorée) par la technique de ... "

La lecture du fichier se fait par une boucle while sur l'ensemble des éléments à ajouter à l'arbre soit dans le pire cas $O(n)$.

Pour les appels récursifs de la construction de l'arbre la complexité est en $O(n)$. En effet il faut créer n noeuds de l'arbre. Par contre cet appel utilisant le tableau C(i,j) sans le parcourir dans le sens normal. Il risque d'y avoir beaucoup de miss de cache

De même pour les appels récursifs du parcours l'arbre, en notant d le temps constant.

$$T(n) = T(k) + T(n-k-1) + d \text{ avec les conditions initiales T(0) = c}$$

Le coût indiqué est obtenu en résolvant ce système.

$$T(n) = [(d * k + d] + [(d * (n-k-1) + d] + d$$

$$T(n) = d * n - d + d$$

$$T(n) = d * n = O(n)$$

3.2 Place mémoire requise :

Justification : Pour trouver l'abre optimal nous utilisons différents tableaux

- P : tableau de taille n
- C : Matrice de taille (n+1)(n+1)
- W : Matrice de taille (n+1)(n+1)
- R : Matrice de taille (n+1)(n+1)
- T : Matrice de taille (n)(2)

3.3 Nombre de défauts de cache sur le modèle CO :

Justification :

4 Compte rendu d'expérimentation

4.1 Conditions expérimentales

Pour mesurer le temps d'execution de notre programme nous l'executons avec la commande "time" avant. Nous avons choisi cette méthode car nous avons eu des problèmes avec la librairie "Ada.Real_Time". De plus cette méthode a l'avantage de nous renvoyer le temps réel d'éxecution et non pas le temps processeur qui n'est pas forcément justifié sur une machine multi-coeurs.

4.1.1 Description synthétique de la machine :

Les tests des benchmarks ont étés effectués sur un machine Dell Inspiron 3542. Voici les caractéristiques de la machine :

- processeur Intel Core i3-4005U (1.70 GHz 4 coeurs) architecture Haswell

-memoire vive : 4 Gb

-systeme Ubuntu 64 bit

Lors des Tests seul le terminal nécessaire au lancement du programme était ouvert. Aucun autre processus n'était en cours.

4.1.2 Méthode utilisée pour les mesures de temps :

Comme expliqué précédemment la fonction utilisée pour mesurer le temps d'exécution du programme est la fonction "time" du système. L'unité de retour est la seconde avec une approximation à 10^-3 secondes. Lors de l'exécution des tests, nous exécutions les tests séparément et non pas en concurrence. De plus deux tests consécutifs portaient sur des fichiers benchmark différents afin de devoir recharger le fichier dans le cache à chaque nouvelle exécution du test. (Si jamais celui-ci etait conservé à la fin de l'exécution). Lors de l'exécution des tests nous avons toujours choisi de prendre le nombre d'éléments du fichier comme "n" d'entrée pour avoir des tests le plus long possibles.

4.2 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)

	temps min	temps max	temps moyen
benchmark1	0.004s	0.005s	0.0043s
benchmark2	0.004s	0.005s	0.0047s
benchmark3	7.001s	7.171s	7.074s
benchmark4	59.831s	60.403s	60.124s
benchmark5	203.872s	204.939s	204.393s
benchmark6	956.741s	970.615s	961.618s

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

4.3 Analyse des résultats expérimentaux

Les temps observés correspondent bien à nos estimations. En effet le temps de calcul augmente très rapidement. (multiplication par 4.6 du temps de calcul entre le test 5 et le 6 pour une augmentation du nombre d'élément de 1.6) Ce qui correspond à peu près à nos calculs de complexité.(1.6^3=4.096). Comme on peut le voir notre algorithme codé sous cette forme n'est pas du tout optimisé et très vite le temps de calcul devient trop grand.