Analyse théorique du projet 2018

CHRISTOFFEL Quentin NASSABAIN Marco

Lundi, 7 Mai 2018

1 Introduction

Ce rapport est une analyse théorique du programme donné pour le projet 2018. Le programme doit prendre en entrée un fichier. Il lit le fichier ligne par ligne et il insère les mots trouvés et leurs occurences dans un arbre binaire de recherche équilibré.

Le programme est testé sur `Turing` et compile sans problèmes. Il n'a pas d'erreur de mémoire. Le programme ne traite pas les sauts de ligne comme fin de phrase. En effet, il est capable de différencier les mots des séparateurs et donc traiter un point comme la fin d'une phrase. Le programme gère aussi les lettres en majuscules et les accents. Vous trouverez plus de détails dans les fichiers source. Le Makefile comporte un jeu de tests pour tester le programme.

2 Manuel d'utilisation

Pour lancer le programme il faut appeler le fichier executable avec un argument: le fichier depuis vous voulez lire le texte.
`./analyser fichier`

Voici une liste des options que vous pouvez utiliser:

- `-a`: Affiche une aide pour l'utilisation.
- `-h`: Affiche la hauteur de l'arbre binaire de recherche.
- `-p`: Affiche la profondeur moyenne des noeuds de l'arbre.
- `-e`: Teste si l'arbre est bien équilibré.
- `-u`: Passage en mode utf-8 pour le traîtement des accents.

Le programme va lire depuis le fichier et afficher l'arbre binaire équilibré de recherche associé au texte.

3 Ensemble ordonné

3.1 Implémentation

Nous avons choisi de représenter la structure de ensemble ordonné par une table. Cette table contient un tableau, qui stocke les élements triés par ordre croissant. La structure contient aussi la

capacité du tableau ainsi que le nombre d'élements dans le tableau. Il est important de remarquer que la capacité est le nombre maximum d'éléments qu'il peut stocker avant de faire une réallocation de mémoire, s'il n'y a plus assez de place dans le tableau.

```
typedef struct s_set
{
    int * elements;
    int max_elt;
    int n_elt;

} * OrderedSet;
```

Nous avons décidé de représenter cette structure sous cette forme car l'insertion nécessite de faire une recherche pour voir si l'ensemble contient déjà l'élément, ce qui est plus efficace avec une recherche dichotomique dans un tableau trié.

3.2 Fonctionnement de la fonction intersection

Au debut de l'algorithme nous choisisons l'ensemble qui contient moins d'éléments que l'autre. Nous faisons un parcours linéaire sur celui-ci et nous testons si les éléments sont présents dans l'autre ensemble. Nous créeons un nouvel ensemble qui contient l'intersection.

3.3 Complexité

3.3.1 Insertion dans l'ensemble

Pour insérer un élément dans l'ensemble nous faisons d'abord une recherche dichotomique pour voir si l'élément est présent, si pendant cette recherche on trouve l'élement alors on s'arrête et on retourne l'ensemble d'entrée. Si l'élément n'est pas présent, on effectue un décalage des éléments du tableau vers la droite, à partir de la position trouvée. Si ce décalage entraîne un dépassement de la capacité du tableau on effectue une réallocation. Ensuite on insère l'élement à cette position.

La complexité de la recherche dichotomique est en $\Theta(\log n)$ pour un ensemble de n éléments. Le décalage du tableau a une complexité en $\Theta(n)$ dans le pire cas. Ce cas correspond à une insertion en première position dans l'ensemble ordonné.

3.3.2 Appartenance d'un élément

Quand il faut tester si un élément appartient à un ensemble on fait une recherche dichotomique. Si on trouve l'élément à la position donnée par la recherche, on renvoie `true` et `false` sinon.

La complexité de cet algorithme est $\Theta(\log n)$. Le pire cas est quand l'élément est en première ou dernière position.

3.3.3 Intersection de 2 ensembles

Pour réaliser l'intersection de deux ensembles nous faisons un parcours linéaire de l'ensemble qui contient moins d'éléments que l'autre. Ensuite on teste si cet élément appartient au deuxième

ensemble avec une recherche dichotomique.

La complexité de cet algorithme est $\Theta(\log m)$.

4 Arbre binaire de recherche

4.1 Choix d'implémentations pour l'arbre binaire de recherche

Pour représenter un arbre binaire de recherche nous avons choisi de créer une structure qui contient les données (le mot et ses occurences), les fils du noeud et le facteur d'équilibrage du noeud.

```
typedef struct s_arbre
{
    char * mot;
    OrderedSet positions;

    int eq;

    struct s_arbre *fg, *fd;

} Noeud, *SearchTree;
```

On a choisi cette structure car elle permet de simplifier les opérations sur les arbres. On stocke le facteur d'équilibrage `eq` dans cette structure pour améliorer la complexité lors de l'équilibrage de l'arbre.

4.2 Fonctionnement des fonctions getAverageDepth et isBalanced

La fonction `getAverageDepth()` utilise la fonction `lci()` qui permet de calculer la longueur de cheminement interne et divise cette valeur par le nombre de noeuds de l'arbre obtenus grâce à la fonction `getNumberString()`. Ce calcul nous donne la hauteur moyenne de tous les noeuds dans l'arbre.

La fonction `isBalanced()` commence par la racine. Si la racine est déséquilibrée elle renvoie `false`. Si elle est équilibrée, on appelle cette fonction sur le fils gauche. Si le fils gauche est déséquilibré, on renvoie `false` et sinon on appelle la fonction sur le fils droit. Si lui aussi est équilibré, on renvoie `true` et sinon `false`. Or cette fonction est récursive donc on va pouvoir répéter ces opérations pour chaque noeud dans l'arbre pour tester son équilibrage.

4.3 Complexité de FindCooccurrences

4.3.1 Hauteur moyenne d'un noeud

Prenons l'exemple de deux arbres avec le même nombre de noeuds dont un est équilibré et l'autre non. Le facteur d'équilibrage d'un noeud correspond à la difference des hauteurs de ses fils. On peut remarquer que cette difference va être plus grande si l'arbre est déséquilibré, car rien n'oblige l'arbre à avoir le même nombre de noeuds de chaque côté. Cependant, dans un arbre équilibré, cette difference est minimale. L'arbre impose que la difference en valeur absolue entre les hauteurs

du fils gauche et du fils droit soit nulle ou un. Donc un arbre non équilibré aura plus de noeuds avec une hauteur importante. Cela résulte en une hauteur moyenne supérieure à la hauteur moyenne d'un arbre équilibré.

Cela peut avoir un grand impact sur les performances d'un programme. Lors du parcours de l'arbre non équilibré il va falloir parcourir beaucoup plus de noeuds pour trouver ce qu'on cherche. Une recherche dans un arbre équilibré est très similaire à une recherche dichotomique dans un ensemble.

4.3.2 Complexité dans le pire cas dans un arbre non équilibré

La première phase de cet algorithme est de trouver les mots donnés dans l'arbre binaire. Dans le cas d'un arbre non équilibré, le parcours va être plus lent. Effectivement, dans le pire cas, les mot recherchés sont des feuilles ayant une profondeur égale à la hauteur de l'arbre. Pour les trouver il va falloir parcourir l'arbre deux fois. La complexité de cette recherche est de $\Theta(n)$, où n est le nombre de noeuds dans l'arbre.

Après avoir fait cela on fait appel à la fonction `intersect()` sur les ensembles des occurences des mots cherchés, dont on a déjà calculé la complexité, qui est donc de $\Theta(k \log k)$ dans le pire cas, k étant le nombre maximal de phrases dans lesquelles un mot apparaît.

4.3.3 Complexité dans le pire cas dans un arbre équilibré

Dans le cas d'un arbre équilibré, la recherche est beaucoup plus rapide. En effet, dans un arbre de `n` éléments, dans le pire cas il va falloir parcourir $\log_2(n+1)$ noeuds. Donc la complexité de la recherche est $\Theta(\log n)$ et pas de $\Theta(n)$ comme dans le cas d'un arbre binaire non équilibré.

La complexité de l'intersection est la même.

5 Exemple

```
$ ./analyser toto
   /--waldo: [ 3 ]
   |
/--qux: [ 2 ]
|  |
|  \--grault: [ 1, 3 ]
|
foo: [ 1, 2, 3 ]
|
|  /--corge: [ 2, 3 ]
|  |
\--baz: [ 1 ]
   |
   \--bar: [ 1, 2 ]
```