Jeux de mots

Algo2 - Ensimag 1A

1 Introduction

Nous considérons ici la programmation d'un algorithme permettant de jouer à différents jeux de sociétés basés sur la formation de mots.

Dans différents jeux, comme le Scrabble, les joueurs disposent d'un ensemble de lettres et ils cherchent à former à partir de celles-ci un mot qui sera ensuite utilisé dans le jeu. Bien entendu, certains mots nécessitent de posséder plusieurs fois une même lettre, le mot "foo" par exemple requiert deux fois la lettre "o".

Nous cherchons ici à écrire un algorithme qui renvoit, à partir d'un ensemble de lettres, la liste de tous les mots qu'il est possible d'obtenir. Chaque lettre ne peut être utilisée qu'une seule fois, mais peut également rester inutilisée. Nous utiliserons pour nos tests une liste de mots du dictionnaire en langue anglaise.

Par exemple, étant donné les lettres a, b et c, il est possible d'obtenir les mots "a", "b", "c" et "cab".

2 Algorithme

L'idée principale est de réaliser un algorithme rapide en stockant l'ensemble des mots du dictionnaire dans un arbre. Chaque feuille de l'arbre contient une liste de mots qui sont tous des anagrammes. Pour réaliser ce stockage, on utilise au sein de chaque feuille une structure de liste chainée. L'arbre contient 27 niveaux. Au niveau du noeud racine, aucune lettre n'est définie. Nous notons par M une constante définissant le nombre maximal d'apparition d'une même lettre dans un mot. Le noeud racine possède jusqu'à $M + 1$ noeuds fils. Le premier d'entre eux va mener vers l'ensemble des mots ne possédant aucune fois la lettre 'a'. Le second d'entre eux mène vers l'ensemble des mots contenant exactement une fois la lettre 'a', et ainsi de suite. Chacun de ces sommets possède à son tour jusqu'à $M + 1$ noeuds fils, qui concernent cette fois-ci la lettre 'b'.

En partant de la racine, et au bout de 26 arètes traversées, on arrive sur un noeud feuille qui contient tous les mots contenant exactement pour chaque lettre un nombre de lettres correspondant aux différents noeuds du chemin.

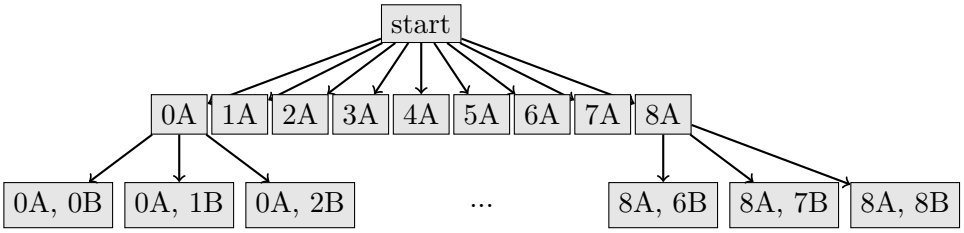

Figure 1: Racine

La figure 1 illustre le haut de l'abre pour $M = 9$ tandis que la figure 2 illustre un noeud feuille.

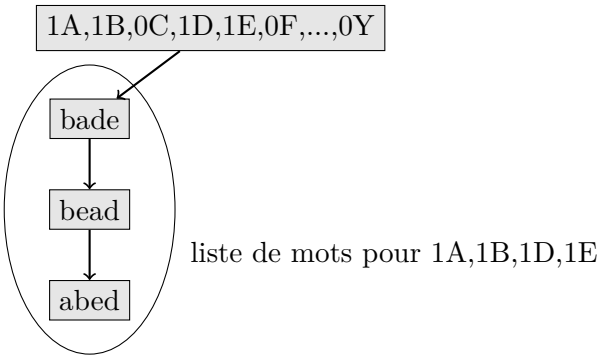

Figure 2: Feuille

Deux algorithmes différents doivent être programmés. D'une part, un algorithme d'insertion qui insère un mot dans notre structure d'arbre et d'autre part, un algorithme de recherche qui affiche à l'écran l'ensemble des mots qu'il est possible de réaliser à partir d'un ensemble de lettres.

Chacun de ces algorithmes fonctionnera de manière récursive.

3 Programmation

On vous fourni un fichier principal (words.adb) complet, qui lit un dictionnaire anglais et construit l'arbre de recherche en insérant les mots un par un. Il attend alors des lettres sur l'entrée standard et pour chaque entrée affiche la liste des mots possibles.

Le programme fourni fait appel aux fonctions d'insertions et de recherche dont les prototypes sont définis dans *tree.ads*. Le fichier *tree.adb* correspondant n'est pas fourni et vous avez donc à l'écrire en totalité.

Les trois fonctions suivantes sont donc à écrire (n'hésitez pas à définir également des sous-fonctions).

```
function New_Tree return Tree;
```

Cette fonction crée un arbre contenant uniquement un noeud racine.

```
procedure Insertion(T : in out Tree ; Word : in String);
```

La procédure d'insertion prend en entrée un arbre et un mot, et insère ce dernier dans l'arbre. N'hésitez pas à utiliser des sous-fonctions pour rendre votre code le plus clair possible.

```
procedure Search_And_Display(T : in Tree ; Letters : in String);
```

La procédure de recherche prend un arbre ainsi qu'un ensemble de lettres codé sous forme de chaine de caractères et affiche à l'écran tous les mots qu'il est possible de réaliser à partir de ces lettres. La procédure ne *renvoie* donc rien.

En plus de toutes ces fonctions, c'est également à vous de choisir la meilleure manière de coder le type *Node*.

Notez enfin, qu'il n'est pas requis que vous codiez les listes (pour les mots d'une feuille) vous-même. Vous pouvez utiliser les paquets ada de type Ada.Containers.Doubly_Linked_Lists par exemple.

La note prendra en compte la clarté du code, décomposé en fonctions claires.

4 Expérimentation

On se propose maintenant d'étudier la vitesse du programme réalisé.

Il est possible de mesurer le temps pris pour un ensemble de recherche de la manière suivante:

- créez un fichier *req.txt* contenant sur chaque ligne les lettres à chercher
- exécutez time ./words < req.txt > /dev/null

On propose ici de regarder le temps d'exécution moyen des recherches pour les mots formés à partir de n lettres choisies aléatoirement (incluant donc possiblement plusieurs fois la même lettre).

Comment effectuer ce genre d'expérience ? Quelles valeurs de n tester ? Pouvez vous afficher une courbe ? La variance a-t'elle une importance ? Quelles conclusions pouvez-vous en tirer ? Comment comparez-vous les résultats obtenus avec ce qu'indiquerait une analyse de coût au pire cas ?

5 Rapport

Le rapport du projet devra avoir une taille de quelques pages (4 ou 5). Il vous permettra d'expliquer quels choix vous avez réalisés et pourquoi (quelles structures de données ? quelle organisation du code ?...).

Il contiendra une section "expérimentation" où vous présenterez les résultats de vos expériences ainsi que les conclusions que vous en tirez.

Les consignes de rendu[1] valables pour algo1 le sont également pour algo2.

[1] `https://intranet.ensimag.fr/KIOSK/Ensimag/1A/Logiciel/Algo1/CR_TPL1.txt`