Rapport

I Description détaillée du programme

A) Les types

```
MAX: array(Character range 'a'..'z') of Natural
```

Ce tableau est utilisé afin de connaître le nombre maximal d'occurrences de chaque lettre, c'est-à-dire le nombre de fils de chaque nœud interne de l'arbre. En effet, le nombre maximal d'occurrence de la lettre 'a' peut être différent de celui de la lettre 'b'. Avoir un tableau plutôt qu'une simple constante nous permet ainsi de ne pas gaspiller de mémoire à chaque niveau de l'arbre. Par exemple, il est inutile que les noeuds "y" aient onze fils s'il n'existe pas de mots contenant plus de deux z.

```
type Tree
```

Notre arbre est un pointeur de nœud (la racine).

```
type Node
```

Un nœud est un type conditionnel qui ne possède qu'un attribut. Si le nœud est interne, il pointe sur un tableau de fils : c'est le cas pour les niveaux de 'a' à 'y'. Le noeud contient alors un pointeur de Arrnode. Sinon, notre nœud est une feuille et pointe vers le premier élément d'une liste de cellules (les mots rattachés à la feuille).

```
type Arrnode
```

Arrnode est un tableau de pointeur de nœud, à chaque case du tableau correspond un nombre d'occurrences de la lettre associé associée au niveau i+1 du nœud contenant le tableau. Pour le cas d'un nœud au niveau 'b', les cases de 0 à n de son attribut nodes, qui est de type Arrnode, sont des pointeurs des nœuds suivant (0c 1c 2c…).

```
type AccessArray
```

Pointeur de Arrnode.

```
type Cell
```

Lorsque vient le moment d'ajouter un mot dans notre arbre, le nœud z pointe vers le premier chaînon d'une liste dont les éléments sont de type Cell. Cell possède deux attributs, un champ word qui est une string non bornée et un attribut suivant qui pointe vers la prochaine cellule de la liste.

B) Les procédures et fonctions

Function New_Tree: Cette fonction se charge de créer un nœud racine et d'appeler la procédure SetMax qui va se charger de créer le tableau de MAX qui interviendra lors de l'insertion.

Procedure SetMax: Afin de conserver un programme assez générique, nous avons fait le choix de ne pas simplement fixer des constantes de max. SetMax va donc appeler SearchMax qui rempliera le tableau de MAX. Trouver le tableau de MAX nécessite d'effectuer un parcours du dictionnaire afin d'identifier les max d'occurrences de chaque lettres. Or, ces valeurs ne changent que si le dictionnaire est modifié.
La procédure SetMax va donc vérifier si le dictionnaire à été modifié grâce au fichier .dictionnaryDate qui contient sur la première ligne la dernière date de modification connue du dictionnaire et sur les 26 lignes suivantes, la liste des max pour chaque lettre.
Si la commande stat retourne une date différente de celle stocké dans .dictionnaryDate, SetMax appelle SearchMax et actualise .dictionnaryDate avec la nouvelle date connue et la nouvelle liste de max. Sinon, il se contente de lire la suite du fichier pour remplir le tableau MAX.

Procedure SearchMax: Cette procédure ne fait que lire chaque ligne du dictionnaire une par une en conservant les max d'occurrence pour chaque lettre. À la fin de l'exécution, le tableau MAX contient les bonnes valeurs.

Procedure Insertion: Cette procédure appelle GetPos afin de connaître la string caractéristique associée au mot rentré par l'utilisateur. Puis elle appelle AddWord qui se chargera de trouver où insérer le mot.

Function GetPos: Afin de savoir où insérer la chaîne dans notre arbre, il faut connaître le nombre d'occurrences de chaque lettre, pour faciliter l'insertion et la recherche, nous effectuons un parcours sur l'arbre à l'aide d'une chaîne caractéristique.
Par exemple, la chaîne caractéristique associée à "superceprojet" est a0b0c1d0e3f0g0h0j1k0l0m0n0o1p2r2s1t1u1v0w0x0y0z0. La procédure AddWord utilisera cette chaîne afin de se repérer dans l'arbre et pour déterminer le prochain nœud sur lequel aller.

Procedure AddWord: Cette procédure permet de parcourir l'arbre, le niveau actuel de l'arbre dans lequel se trouve la procédure est indiqué par l'état de la chaîne de caractéristique. À chaque appelle récursif, une partie de la chaîne caractéristique est "consommée" et le noeud transmis en argument est la ième case du tableau de noeud du noeud courant.
Exemple: La chaîne caractéristique associée au mot "cabb" est "a1b2c1d0...z0". On a donc le graphe d'appel suivant:
AddWord(T,"cabb","a1b2c1..z0") --->
AddWord(T.ALL.nodes.ALL(1),"cabb","b2c1..z0") --->
AddWord(T.ALL.nodes.ALL(1).nodes.ALL(2),"cabb",c1..z0") ---> ....
Lorsque le premier caractère de la chaîne caractéristique est un 'z', cela signifie que notre noeud est en fait une feuille, dans ce cas-là, il ne nous reste plus qu'à ajouter le mot "cabb" grâce à la procédure Add.

Procedure Add: Lorsque AddWord appelle cette fonction, on connaît déjà la feuille à laquelle ajouter le mot, il ne nous reste plus qu'à l'ajouter en tant que tête de son attribut head, qui est une liste.

Procedure Search_And_Display: Cette procédure se contente d'appeler la procédure Search en lui fournissant la chaîne caractéristique associée au mot recherché.

Procedure Search: Le comportement de Search est assez similaire à celui d'insertion, la différence majeure est qu'a la place de se déplacer de nœuds en nœuds récursivement, un appel récursif est effectué sur toutes les cases du tableau inférieur au nombre d'occurrence des lettre rentrées par l'utilisateur.
Exemple: On reprends le mot "cabb", la chaîne associée ne change pas.
Search(T,"a1b2c1..z0")-->
Search(T.ALL.nodes.ALL(0),"b2c1..z0") et Search(T.ALL.nodes.ALL(1),"b2c1..z0")
Puis chaque nouveau Search sera fait sur les cases de 0 à 2 et ainsi de suite.
Lorsque toute la chaîne caractéristique à été consommée, on appelle display sur la liste de mots associée à la feuille courante. Cela aura pour effet d'afficher tous les anagrammes du mot rentré, en prenant en compte l'éventualité qu'il n'est pas nécessaire d'utiliser exactement toutes les lettres, qu'il est possible d'en utiliser moins.

Procedure Display: Cette procédure va simplement afficher tous les mots de la liste transmise en arguments.

Procedure Free: Comportement similaire aux deux fonctions récursives déjà détaillées, il s'agit simplement de parcourir l'arbre afin de free toute la mémoire. Un simple run de Valgrind permet de s'assurer que la procédure n'oublie pas un seul octet.

II Justification des choix d'implémentation

Le sujet étant assez précis, voir trop pour permettre une vraie démarche de réflexion sur l'implémentation à choisir, nous nous somme assez vite décidé sur le choix d'un tableau pour représenter les fils d'un nœud.
Parmi les possibilités envisagées se trouvait une liste, intéressante car elle permet de ne pas utiliser de mémoire inutilement, ce qui est le cas avec le choix du tableau, mais rejetée car le coût de recherche est en moyenne de O(n/2).
La liste triée aurait permis de conserver l'avantage apporté par la liste simple en mémoire et permet d'avoir une recherche en O(1), cependant le coût en insertion s'en trouve grandement affecté avec une moyenne en O(n/2) et un pire cas en O(n).

L'autre solution envisagée était celle d'un AVL avec du log2(n) en insertion et en recherche et c'est certainement celle qui aurait été retenue si le sujet était plus vague ou alors s'il un programme générique et scalable. Il s'agit de la solution la plus rapide en recherche/insertion garantissant un usage optimal de la mémoire.

Cependant, puisque le problème étudié est assez précis, le choix du tableau est une évidence. Le max du max d'occurrence de chaque lettre de chaque mot du dictionnaire est de 6. Donc, avec un tableau de taille 6 ou plus, il ne sera jamais important d'avoir sur un grand nombre de nœud des cases mémoires réservés mais non utilisé.
De plus, cette solution permet de la recherche et de l'insertion en O(1) et c'est bien le temps qui est la première contrainte de ce projet.
Nous avons donc choisi de représenter les fils de nœuds comme un tableau de taille fixé égale au max d'occurrence de la lettre correspondante au niveau du nœud courant dans l'arbre.

Le second défaut du sujet, à mon sens, est celui d'imposer des feuilles sur le niveau Z de l'arbre uniquement. Il sera extrêmement courant d'avoir des nœuds au niveau Y ou plus qui aurait pu être des feuilles et ainsi économiser un niveau de parcours et un peu de mémoire.

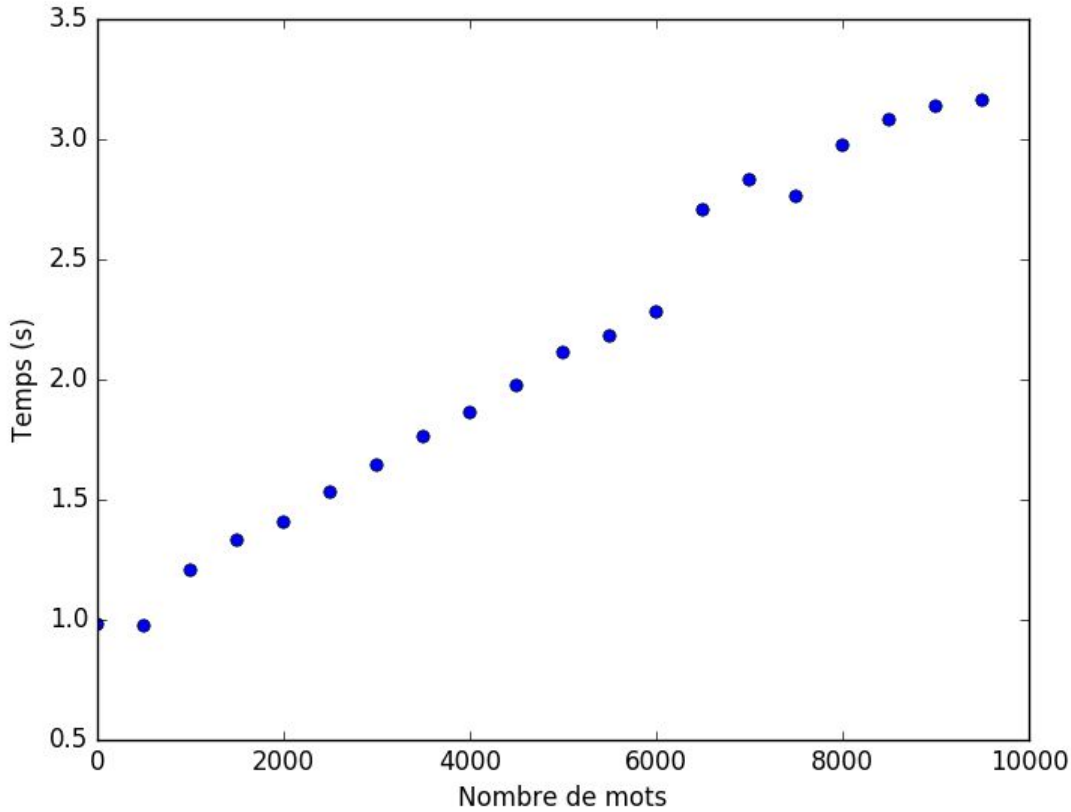
Nous avons fait le choix de respecter l'énoncer et de ne pas mettre en place cette amélioration.

III Performances

Les performances du programmes ont été testées selon deux critères : le nombre de mots (lignes), pour un nombre de lettres fixé, et le nombre de lettres, avec un seul mot. Pour ce faire, un programme Python, `benchmark.py`, a été développé, dans l'optique d'automatiser les appels au programme et la génération des graphiques.

A) Nombre de mots

Ici, le fichier d'entrée contient x lignes, avec dix lettres par ligne. La recherche d'un mot étant en temps constant, on obtient un coût linéaire, comme illustré sur la figure ci-dessous :



B) Nombre de lettres

Pour cette partie, nous n'avons qu'un seul mot, dont le nombre de lettres varie. Ce mot est généré aléatoirement par le script Python. Comme l'illustre la figure ci-dessous, les temps d'exécution sont assez aléatoires et ne semblent pas dépendre énormément de la taille de la chaîne.

Cela peut s'expliquer par le fait que l'entrée du programme est pré-traitée : même s'il y a cinq cent 'a', on n'en considère que n, n étant le nombre maximal de 'a' par mot.

Pour donner un ordre de grandeur, la pire chaîne possible, aaaaaaaaaabbbbbbbbbbbbccccccccccccccccddddddddddddddddeeeeeeeeeeeeeeeeeeffffffffffffffffgggggggggggghhhhhhhhhhhhhhhiiiiiiiiiiiiiiiiiiiijjjjjjjjjjjjjjjkkkkkkkkkkkkkkkllllllllllllllmmmmmmmmmmmmmnnnnnnnnnnnnoooooooooooooppppppppppppqqqqqqqqqqrrrrrrrrrrssssssssssstttttttttttttuuuuuuuuuuvvvvvvvvvwwwwwwwwwxxxxxxxxxxyyyyyyyyyyyyzzzzzzzzzzzzzz, donne un temps d'exécution de 1.231s. Ainsi, la variance entre les caractères n'influe pas énormément sur les performances temporelles.