Roos Bastien
Bouton Paul

RAPPORT PROJET ALOGORITHMIQUE 2 : JEUX DE MOTS

I-Organisation du code et choix des structures de données

1-Organisation du code

Au sein de notre code, nous avons utilisé plusieurs fonctions auxiliaires permettant une meilleure lisibilité de l'ensemble.

Tout d'abord, la fonction compter lettre permet d'alléger les deux procédures principales (*insertion* & *search_and_display*) en factorisant le code. La procédure *free* libère la structure de donnée (l'arbre de recherche et les listes de mots). Cette dernière est récursive et est utilisée lorsque l'utilisateur décide de quitter le programme (Ctrl + D).

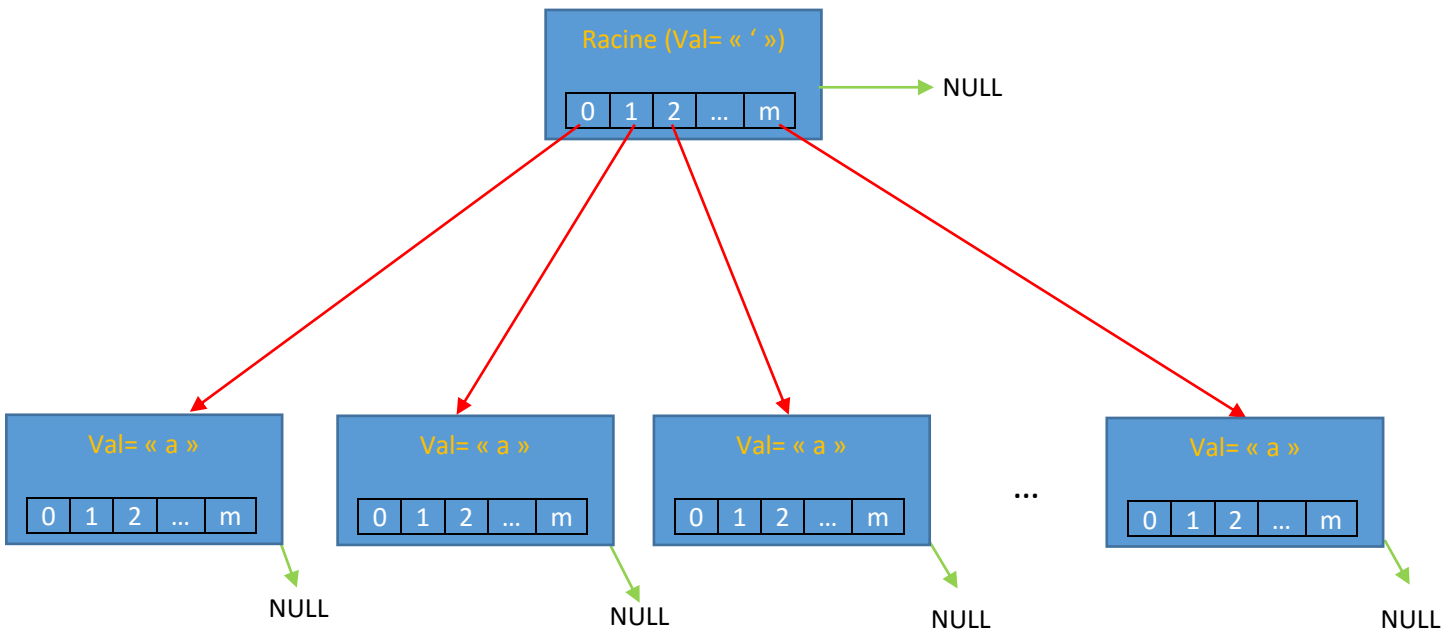
En ce qui concerne les fonctions d'insertion et de recherche, elles sont, comme demandé par le sujet, récursives et utilisent le package *Ada.Containers.Doubly_Linked_Lists.* Ce package facilite les opérations d'insertion et de recherche dans les listes pointées par les feuilles de l'arbre, ces fonctions étant préprogrammées et optimisées dans le module. Cependant, ce package permet de ne créer que des listes d'éléments de type non contraint, c'est pourquoi nous avons utilisé la package *Ada.Strings.Unbounded* permettant la conversion de chaines de caractère (type contraint en ADA) en chaines de caractères non contraintes (*To_unbounded_strings(words)*).

Enfin, nous avons eu recours aux exceptions afin de gérer les cas où un caractère est présent trop de fois au sein d'un même mot (CONSTRAINT_ERROR). En effet, nous avons utilisé des tableaux de pointeurs (type contraint en ADA) dans notre structure de données.

2-Structures de données

Au sein de notre arbre, nous avons décidé d'utiliser un seul type de nœud, un *record* composé de:

- Val : La valeur du caractère en ce nœud.
- Liste : Un pointeur vers une liste de mot.
- Fils : Un tableau de pointeurs vers les nœuds fils.



En mettant dans chaque cellule un pointeur vers une liste de mot (initialisée à NULL pour chaque nœud créé) nous conservons une unique structure d'arbre. En effet, tous nos nœuds étant du même type, chaque sous arbre de notre structure initiale reste un arbre. Il est ainsi plus facile de gérer les appels récursifs dans nos procédures en ne différenciant pas le type des cellules pointant vers des listes de mots non vide (Feuilles de l'arbre (Val= « z »)) de celui des nœuds de l'arbre.

De plus, le choix d'un tableau de pointeurs indicé par le nombre de lettres permet de clarifier notre code : Nous utilisons une boucle for pour les appels récursifs au lieu d'un case.

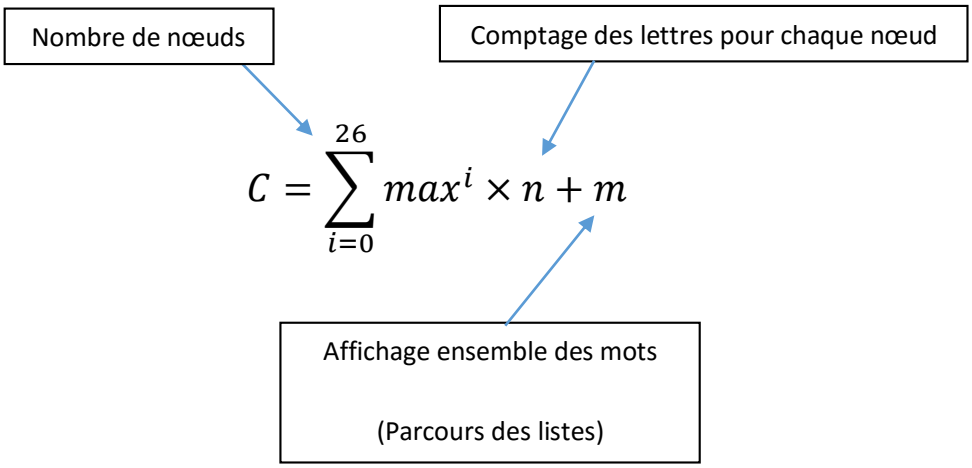
Enfin, pour éviter un surcout mémoire, la procédure insertion crée l'arbre en fonction des mots à insérer. Par conséquent, les feuilles dont le chemin ne correspond à aucun mot ne sont pas générées. Ce fonctionnement permet également de détecter plus vite les combinaisons de lettres ne menant à aucun mot : Le passage par un pointeur NULLlors de la recherche suffit à s'assurer que la combinaison ne mène à aucun mot (Il n'est pas nécessaire de continuer le parcours de l'arbre jusqu'aux cellules contenant la valeur « z »).

II-Tests et résultats

1-Calcul théorique du coût

Calculons le coût dans le pire cas de notre algorithme (on considère ici la recherche de mots) :

Le pire cas correspond à rechercher tous les mots d'un arbre complet (toutes les listes de mots des cellules où val= « z » sont non vides). On considère m mots de n lettres et on note *max* le nombre de lettres identiques maximales permises dans un mot (cela correspond à la taille des tableaux de pointeurs)



Cette formule nécessite que n soit au moins égal à $26 \times \max$ pour inclure les feuilles correspondant au nombre maximum de chaque lettres.

$$\sum_{i=0}^{26} max^i = \frac{1 - max^{27}}{1 - max} \approx max^{26}$$

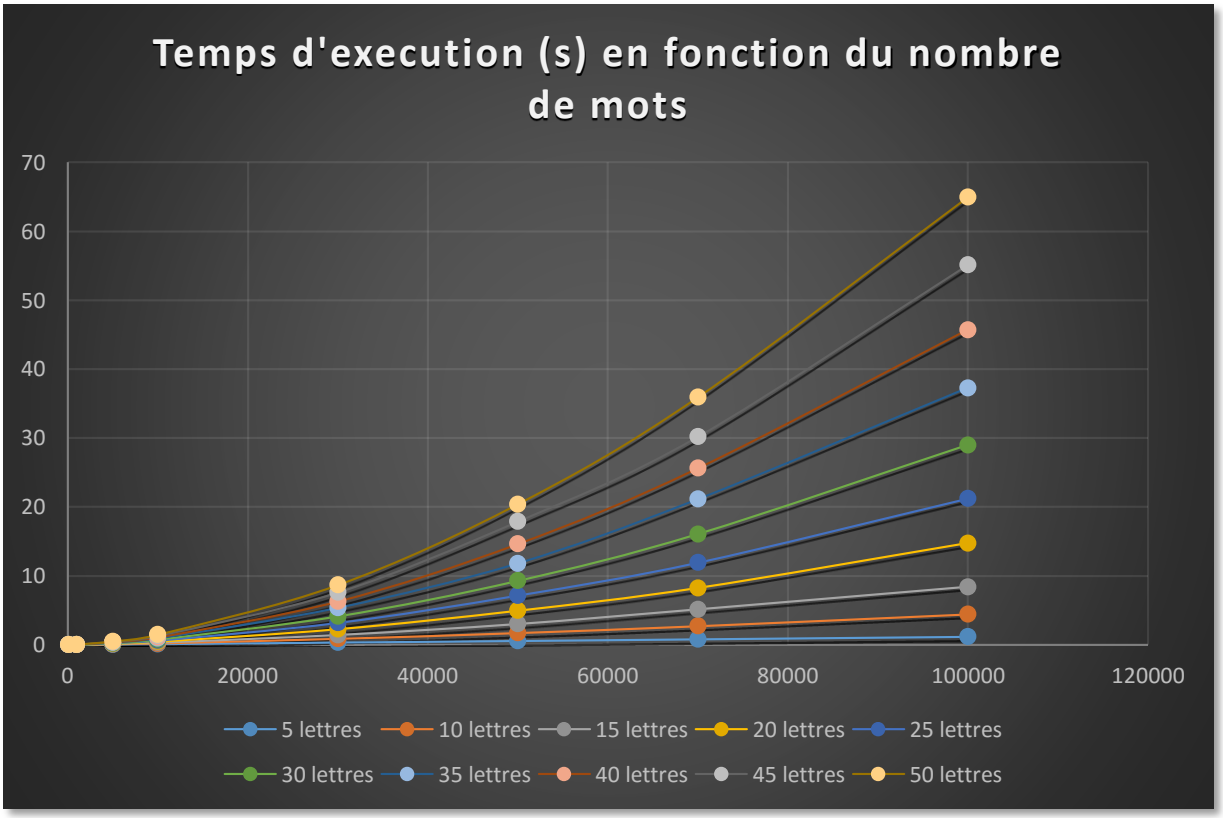
On a donc :

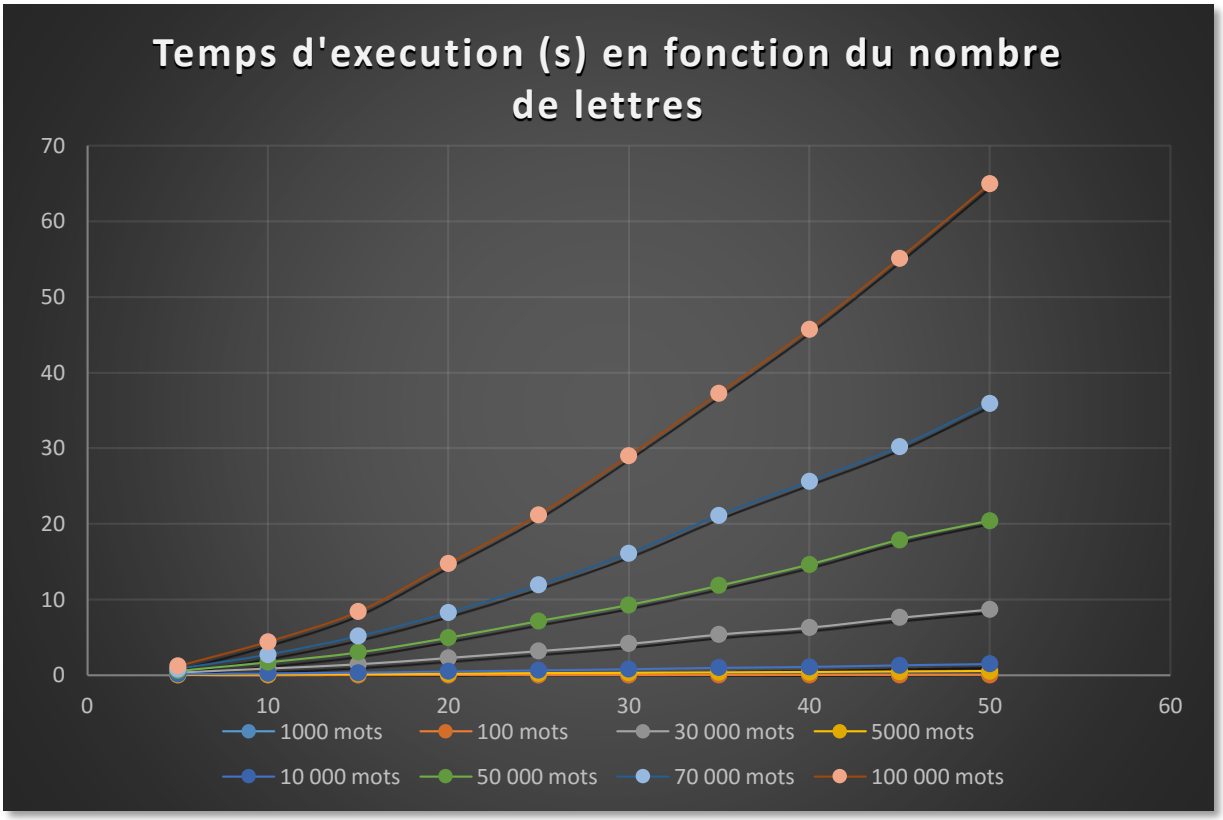
$$C = O(max^{26}) \times O(max) + m = O(max^{27})$$

Il est important de préciser que ce cout en pire cas n'est en pratique quasiment jamais atteint, ce qui a été vérifié par nos différents tests (coût en réalité très inférieur au coût théorique).

2-Tests

Lors de nos tests, nous avons étudié deux relations : Le temps d'exécution en fonction du nombre de mots et le temps d'exécution en fonction du nombre de lettres. Pour cela, nous avons modifié le fichier random_input.pl afin de générer des fichiers contenant des listes de n mots de p lettres. Pour ne pas fausser nos tests, nous avons pris soin de remplacer également le fichier d'entrée pour l'insertion des mots afin de s'assurer que tous les mots recherchés soient bien présents. Enfin, nous avons également relevé notre limite de lettre maximale pour éviter tout rejet de mots lors de l'insertion. Voici les résultats obtenus (les points sur les courbes correspondent à des moyennes de 3 tests) :





Nous observons que lorsque le nombre de lettres augmente (courbe 2), l'écart entre les courbes s'accroit de manière importante. En effet, l'augmentation des appels récursifs (les lettres sont présentes un plus grand nombre de fois) entraine une exploration beaucoup plus grande de l'arbre. Ainsi, on voit que la courbe correspondant à 100 000 mots explose : il y a énormément d'appels récursifs ($50 \approx 2 \times 26$ il y a donc de manière presque certaine de la redondance sur chacun des caractères) sur un grand nombre de mots. En revanche, lorsque le nombre de mots augmente (courbe 1), il n'y a pas plus d'appels récursifs pour chaque mot (nombre de lettre fixé). Ainsi la fonction *compter_lettre* influe majoritairement sur le temps d'exécution : son coût étant linéaire, cela explique l'écart relativement constant entre les courbes du premier graphe.

On est cependant bien loin des courbes correspondant au pire cas : max^{27}. En effet, nous avions fixé max à 20 lettres identiques et nous n'avons pas dépassé les 50 lettres dans un seul mot. Or il nous aurait fallu un mot d'au moins 520 lettres (20 fois chaque lettre) pour parcourir tout l'arbre lors de la recherche.

En ce qui concerne la variance du nombre moyen de lettre identique dans un même mot, elle n'est pas importante pour nos tests. En effet, le nombre de mot minimum testé étant de 100, une variance élevée impliquant un grand nombre d'appels récursif pour un certains mot (Les plus écarté de la moyenne avec un écart positif) est compensé par des mots ayant un faible nombre de lettres identiques (Les plus écarté de la moyenne avec un écart négatif). De plus une faible variance permet

de ne pas avoir une explosion du cout dans certains cas (nombre de lettre identiques important). Ainsi, la variance n'influe que très peu sur nos tests.

III-Pistes d'amélioration

Tout d'abord, notre structure de donnée impose une grande perte mémoire au niveau des feuilles où val = « z ». En effet, nous allouons un tableau de pointeur (même si les pointeurs valent NULL, le tableau occupe une place mémoire) qui ne sera jamais utilisé En effet, les cellules contenant la valeur « z » pointent seulement vers des listes mais n'ont jamais de fils.

De plus, chaque cellule pointe vers une liste ce qui inutile hormis pour la dernière cellule (correspondant à la lettre z). Cependant, les listes étant initialisées à NULL, le surcout mémoire est négligeable.

Nous aurions également pu réaliser une étude statistique afin de fixer le nombre limite de lettre identique dans chaque mot. Il suffirait de relever le nombre d'apparition en moyenne de chaque lettre dans les mots de la langue considérée et fixé max à une valeur légèrement supérieure au maximum mesuré. Ainsi, nous éviterions d'avoir beaucoup de fils inutilisés tout en rejetant un faible nombre de mots lors de l'insertion.

IV-Conclusion

Après la réalisation de nos tests, nous sommes satisfaits de l'efficacité de notre structure. En effet, elle permet une exécution relativement rapide tout en conservant un code clair.

Enfin, notre fonction free (également récursive) permet de libérer la mémoire allouée dans son intégralité (nœud + listes) et donc d'éviter toute fuite mémoire (résultat confirmé par Valgrind).