Rapport de projet de programmation

Conception d'un correcteur orthographique

Équipe 81

23 novembre 2021

Sommaire

I) Introduction

L'objet de ce projet était l'implémentation d'un algorithme visant à identifier des mots appartenant à un dictionnaire dans un texte, et ainsi de déterminer les mots existants selon le lexique établi par ce même dictionnaire. Nous rappelons que l'explication pour utiliser chaque exécutable est décrit dans le fichier README du projet.

Pour mettre en oeuvre cet algorithme, nous devions implémenter trois approches différentes, afin d'en comparer les performances respectives. Nous avons alors décidé d'implémenter un arbre préfixé, un arbre radix et une liste. Nous nous attendons à priori à ce que la liste soit bien moins adaptée à ce genre de problème que les deux autres structures.

Dans la première partie de ce rapport, nous expliciterons les raisons qui nous ont poussé à choisir chaque structure en commençant par rappeler leurs principes fondamentaux, les choix techniques que nous avons dû faire, la complexité théorique des fonctions appelées puis une description précise des résultats obtenus.

Enfin, nous conclurons en comparant les performances de chaque implémentation et nous discuterons du choix de l'implémentation à réaliser en fonction de plusieurs cas d'usages.

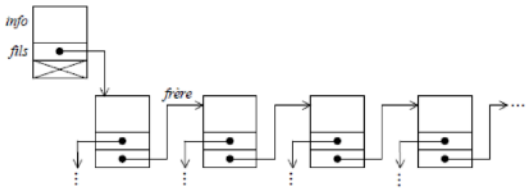
II) L'implémentation de l'algorithme

a) L'arbre préfixe

1) Choix technique de l'implémentation

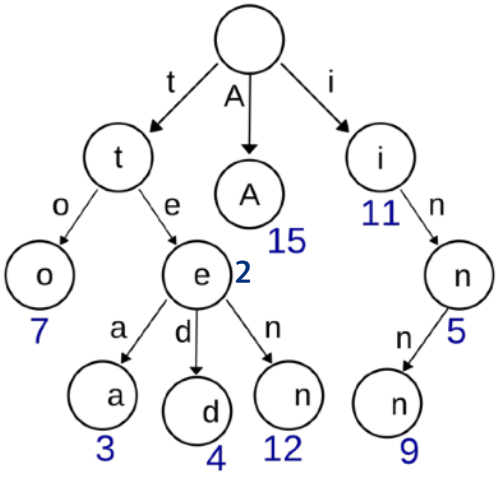
L'arbre préfixe est une structure arborescente qui à chaque noeud associe 1 caractère, 1 noeud fils et 1 noeud frère.

Pour commencer nous avons choisi d'utiliser des listes pour relier chaque frère et chaque fils car cela permet d'utiliser peu de place mémoire et d'ajouter facilement de nouveaux noeuds lorsqu'on souhaite ajouter un mot à l'arbre. La recherche dans une liste est d'une complexité temporelle en O(n) mais ce ne sera pas un problème puisque chaque liste ne peut pas être plus longue que 26 cellules (nombre de lettre dans l'alphabet) pour une liste de frères et 25 cellules (longueur du mot le plus long) pour une liste de fils (soit un parcours maximal de 25*26=650 cellules par mot dans le cas ou le graphe serait rempli de toutes les branches possibles de 25 lettres soit $26^{25}=2.10^{35}$ mots alors que la langue française n'en comporte que 3.10^{4}!).



La recherche d'un mot dans cet arbre consiste donc à comparer chaque lettre composant le mot (en le lisant de gauche à droite) avec un noeud de l'arbre, si celui -ci correspond on refait la même chose avec le caractère suivant du mot sur le noeud fils jusqu'à faire toutes les lettres, s'il ne correspond pas on réessaie avec le frère du noeud tant qu'il existe un frère. Un mot est donc présent si on a parcouru l'arbre en consommant tous les caractères du mots et ne l'est pas si on a parcouru tous les frères sans trouver le caractère du mot.

La structure correspond à ce schéma :



Cependant, on peut constater que si le mot "te" existe et est représenté dans l'arbre au noeud 5, ce n'est pas forcément le cas du mot "in" qui ne veut rien dire en français mais qui est pourtant bien présent dans l'arbre au noeud 2. Nous avons donc fait le choix d'ajouter un noeud fils contenant le caractère '\0' à chaque fin de mot dans l'arbre. Ainsi le noeud "te" comportera un noeud fils "\0" pour signifier qu'il peut représenter la fin d'un mot mais pas le noeud "in" ce qui signifiera que le mot "in" n'est pas un mot. Cela imposera donc, lors de la recherche d'un mot, de bien vérifier la présence de chaque lettre du mot dans l'ordre puis à la fin de vérifier aussi la présence de '\0' en tant que noeud fils de là où on se situe dans l'arbre.
Le but est de vérifier l'existence d'un mot issu d'un texte dans une dictionnaire mais le mot peut être entouré de guillemets, de parenthèses etc.. Il convient donc de retirer toutes les ponctuations avant de rechercher le mot dans l'arbre. Nous avons décidé de retirer les '(' et les ' " ' pouvant être présents au début du mot et les ')' ' " ' ',' '-' à la fin du mot. Si une fois cela fait on remarque que le mot est vide (le mot lu n'étais qu'une parenthèse par exemple), alors on le considère comme français). Si le mot contient une majuscule ou un ponctuation (ex : grand-mère) alors on le considère aussi comme français. Sinon, dans le cas général, on vérifie la présence du mot dans l'arbre

Fonctions implémentées : création d'un noeud, test si l'arbre est vide, ajout d'un noeud fils, descente dans le noeud fils correspondant à un caractère c, affichage d'e l'arbre, insertion d'un mot dans l'arbre, test si un mot est présent dans l'arbre, destruction d'un noeud, destruction d'un arbre, création d'un arbre selon les données d'un fichier.

2) Complexité théorique

Pour l'insertion :

L'insertion est décrite dans la fonction insererMot_ArbrePrefixe, elle se fait en appelant successivement n fois (n=longueur d'un mot) la fonction add_fils décrite dans ArbrePrefixe.h.
Dans le pire des cas pour add_fils (ou le noeud n'existait pas et qu'il faut l'ajouter), on doit parcourir toute la liste de frère (complexité en O(k) en temps avec k la longueur d'une liste de frère borné par 27 le nombre de lettre de l'alphabet+caractère '\0') et ajouter le noeud à la fin(complexité en O(1) en temps).
L'insertion est donc d'une complexité en O(n).
Pour la création d'un graphe à x mots, la complexité est donc en O(n.x)=O(x) car il faut insérer x fois un mot et n borné(la longueur max d'un mot français avec le caractère '\0' à la fin) . Et l'insertion consiste à créer ou parcourir autant de noeuds qu'il y a de lettres dans le mot.

Pour la recherche :

La recherche est décrite dans la fonction est_present_arbrePrefixe qui retire toutes les ponctuations au début puis descend dans le graph tant qu'un noeud contient le caractère numéro i du mot.

Avec un arbre composé de n mots, on aura au mieux (insertion de n mots répartis uniformément dans l'arbre) une complexité en log(n) car, en moyenne, multiplier par 2 le nombres de mot dans l'arbre revient à faire +1 au nombre d'embranchement par branches existant et donc faire 1 comparaison supplémentaire.

Pour la destruction :

La destruction consiste à libérer chaque noeud 1 à 1, on peut donc estimer sa complexité à O(n).

3) résultats obtenus

Pour tester nos algorithmes et leur fonctionnement réel, nous avons mené 4 types de tests consécutifs : un test de temps de remplissage de la structure étudiée avec les mots du dictionnaire, un test de recherche des mots du texte dans la structure, un test de destruction de la structure et un ultime test de mesure du temps pris par le process complet. Nous mesurons à chaque fois le temps pris par l'exécution du programme, en fonction du pourcentage de mots dans le dictionnaire. Le processus de test sera le même pour les autres structures (arbre radix et liste).

L'implémentation fonctionne bien, le temps d'exécution du test verif_ortho prend environ 0,28 secondes(résultats avec une autre machine que celle pour faire les graphiques suivants) en prenant tout en compte (lecture des fichiers, creation, recherche & destruction de l'arbre). Nous avons réalisé des mesures de performance temporelle de l'implémentation en fonction du nombre de mot ajoutés (en utilisant un certain pourcentage du dictionnaire FR.txt), les résultats des tests sont présentés en page suivante.

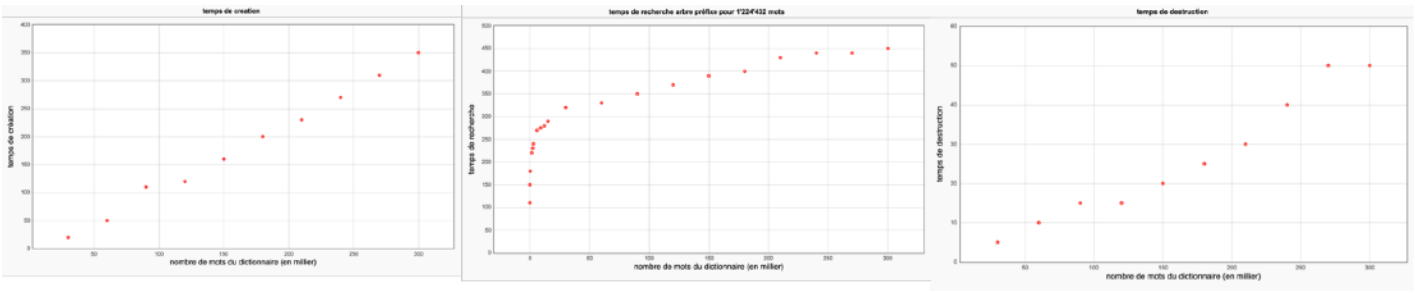


Figure 1 : Résultats des tests menés sur l'arbre préfixe

Courbe de gauche : temps de création de la structure en fonction du nombre de Mots présents dans le dictionnaire
Courbe du milieu : temps de recherche dans la structure en fonction du nombre de Mots présents dans le dictionnaire
Courbe de droite : temps de destruction de la structure en fonction du nombre de mots présents dans le dictionnaire

La courbe d'insertion est linéaire ce qui s'explique par une complexité théorique en O(n) comme justifié précédemment.
La courbe de destruction est linéaire ce qui s'explique par une complexité théorique en O(n) comme justifié précédemment.
La courbe de recherche est logarithmique ce qui s'explique par une complexité en O(log(n)) comme justifié précédemment.

On comprend donc que cette structure est très efficace pour rechercher des mot permis une grande base de donnée de mot. Plus le dictionnaire de la base de donnée sera grand, plus l'avantage d'utiliser un arbre Préfixe pour rechercher des mots sera grand.

b) L'arbre radix

1)Choix technique de l'implémentation

L'arbre radix est une structure de donnée dérivée de l'arbre préfixe. Sa spécificité est la suivante : si un noeud n'a qu'un fils, le noeud père et fils sont fusionnés. Cela permet de réduire théoriquement le nombre de comparaisons lors de la recherche d'un mot, car toute la fin du mot sera contenue dans le noeud. Toutefois, on anticipe que ce gain de temps pour la recherche de mot se répercutera sur le temps de création du dictionnaire, plus important à priori.

2) Complexité théorique

Théoriquement, la complexité temporelle de construction d'un arbre radix est supérieure à celle de la construction d'un arbre préfixe. En effet, à chaque ajout d'une chaine de caractère dans le graphe (opération que réalise la fonction rec_insertion_arbreRadix), il est nécessaire de considérer le cas où le noeud racine du graphe que l'on considère a un préfixe en commun avec la chaine que l'on veut rajouter. Pour ce faire, il faut modifier le noeud racine en y attribuant la valeur du préfixe, et lui attribuer un nouveau fils contenant le suite du mot. La complexité temporelle de construction reste cependant en O(n).

De plus, la complexité en recherche sur aussi en O(log(n)) mais elle devrait être néanmoins plus rapide qu'avec un arbre préfixe car il nécessite moins de comparaison puisque certains noeuds ont été fusionnés (cette affirmation est de moins en moins véridique quand l'arbre est de plus en plus dense car moins de fusions sont possibles).

Enfin, la destruction de l'arbre consiste à libérer chaque noeuds créé, on estime donc sa complexité en O(n).

3) Résultats obtenus à l'issue des tests

L'implémentation fonctionne bien, le temps d'exécution du test verif_ortho prend environ 0,89 secondes(résultats avec une autre machine que celle pour faire les graphiques suivants) en prenant tout en compte (lecture des fichiers, creation, recherche & destruction de l'arbre). Nous avons réalisé des mesures de performance temporelle de l'implémentation en fonction du nombre de mot ajoutés (en utilisant un certain pourcentage du dictionnaire FR.txt), voici les résultats obtenus :

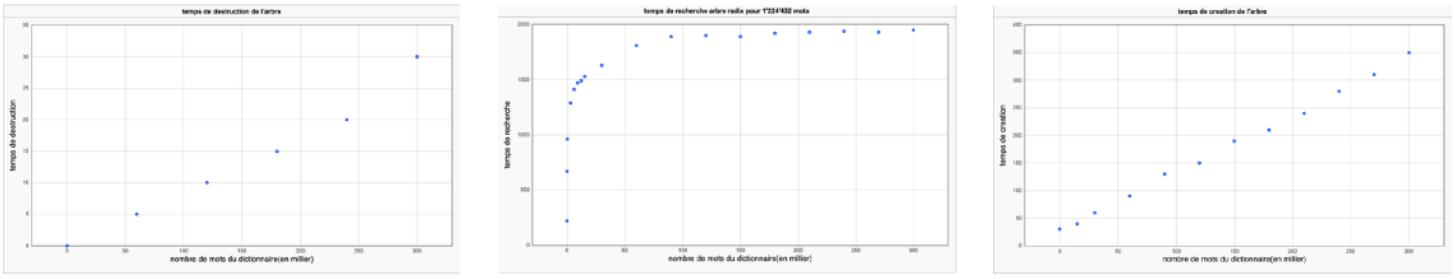


Figure 2 : Résultats des tests menés sur la structure radix

Courbe de gauche : temps de création de la structure en fonction du nombre de mots présents dans le dictionnaire
Courbe du milieu : temps de recherche dans la structure en fonction du nombre de mots présents dans le dictionnaire
Courbe de droite : temps de destruction de la structure en fonction du nombre de mots présents dans le dictionnaire

On remarque donc, comme nous l'avions dit précédemment que le temps de construction et de destruction est linéaire (complexité en 0(n)) et que le temps de recherche est logarithmique (complexité en 0(log(n)).

b) La liste

1) Choix technique de l'implémentation

La structure de liste est très limitée dans l'utilisation que l'on veut en faire ici. Il est impossible de faire apparaitre la notion de noeud fils et de noeud frère, hormis en leur attribuant des indices spéciaux mais ce cas là correspondrait davantage à une implémentation de tas. Ici, la constitution du dictionnaire est primaire : cela consiste seulement à ajouter toutes les chaines de caractères du dictionnaire à la liste. La recherche fonctionne aussi de façon triviale : pour statuer sur la présence d'un mot du texte dans le dictionnaire, il suffit de comparer ce mot avec tous les autres éléments de la liste. Si l'élément n'est pas présent, le compteur des mots non français s'incrémente. Le choix de cette structure relève plutôt d'un désir de mettre en évidence l'inefficacité de cette structure pour ce type de problème, plutôt que d'un choix d'optimisation.

2) Complexité théorique

La complexité de l'ajout d'un mot dans le dictionnaire est O(1) car on ajoute seulement le nouveau mot en tête de liste. Ainsi, la complexité de création du dictionnaire est linéaire par rapport au nombre de mots présents dans ce dernier. (O(n))

Pour ce qui est de la recherche, dans le pire des cas le mot n'est pas présent dans le dictionnaire, et l'on aura eu à comparer les n mots du dictionnaire avec le mot recherché. Ainsi, la complexité temporelle de la recherche est en O(n).

Concernant la destruction, elle est aussi de complexité temporelle linéaire par rapport au nombre de mots, car on libère l'espace mémoire de chaque mot jusqu'à la fin de la liste.

3) Résultats obtenus à l'issue des tests

L'implémentation fonctionne bien, le temps d'exécution du test verif_ortho prend environ 35 minutes en prenant tout en compte (lecture des fichiers, creation, recherche & destruction de l'arbre). Nous avons réalisé des mesures de performance temporelle de l'implémentation en fonction du nombre de mot ajoutés (en utilisant un certain pourcentage du dictionnaire FR.txt), voici les résultats obtenus :

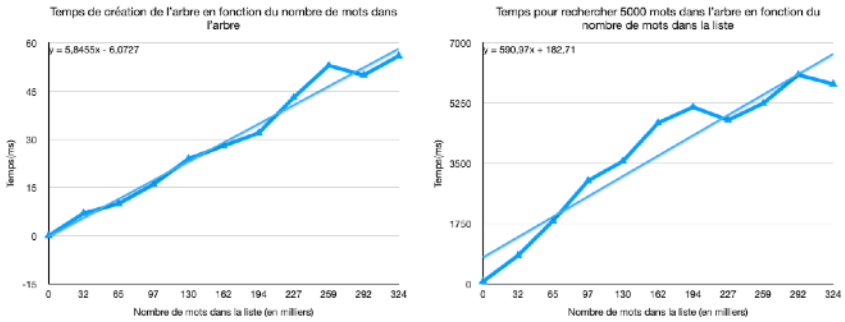


Figure 3 : Résultats des tests menés sur l'arbre préfixe
Courbe de gauche : temps de création de la structure en fonction du nombre de mots présents dans le dictionnaire
Courbe du milieu : temps de recherche dans la structure en fonction du nombre de mots présents dans le dictionnaire

Les résultats obtenus sont cohérents avec notre analyse de la complexité théorique. Plus le nombre de mots du dictionnaire augmente, plus le temps de creation augmente (relation linéaire), de même pour le nombre de mots à rechercher avec le temps de recherche.

III) Conclusions sur l'optimisation

a) Performances comparées

Performances temporelles :

Nous vous remarqué que la structure la plus efficace en temps est l'arbre préfixé, suivi de l'arbre radix puis la liste. Le liste est largement moins efficace que les 2 autres et ceci s'explique par le fait qu'il faut parcourir la liste jusqu'à trouver le mot recherché, ce qui est très long puisque la liste fait plusieurs milliers de cellules.

En revanche il est étonnant de voir que l'arbre suffixe est plus rapide que l'arbre radix, ceci s'explique sans doute par le manque d'optimisation de l'arbre radix qui doit faire de nombreux free et faire des comparaisons avec des chaînes de caractères alors que l'arbre préfixe semble lui bien optimisé et ne compare que des caractères uniques.

Performance en mémoire :

Pour mesurer expérimentalement l'empreinte mémoire de chaque structure. Nous avons utilisé l'outil massif-vizualizer qui enregistre à certains instants la place mémoire qu'utilise le programme. ?Nous avons obtenu les résultats ci-dessous : (Comme la liste est bien trop lente pour pouvoir être exploitable, nous avons effectué les mesures pour un dictionnaire avec uniquement les 100 premiers mots, nous approcherons le résultat en multipliant par environ 3000 l'espace mémoire nécéssaire car la mémoire qu'occupe une liste est proportionnel à sont nombre de cellule)

Nous avons utilisé les commandes ci dessous pour obtenir les graphes suivants :

```
valgrind --tool=massif ./bin/verif_ortho text/FR.txt text/a_la_recherche_du_temps_perdu.txt
ms_print massif.out.PID
```

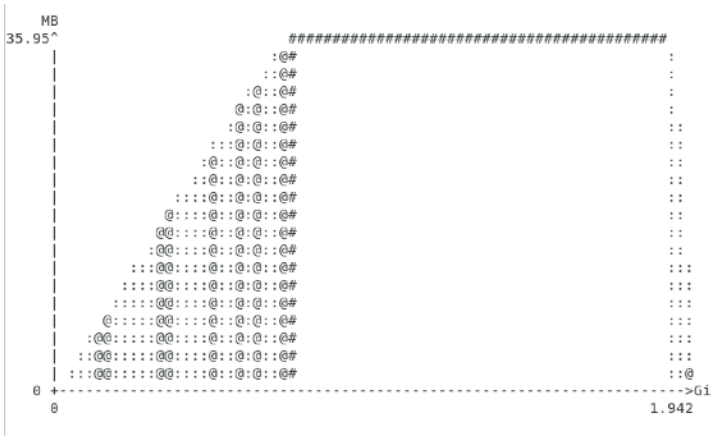


Figure 4 : *Mesure de l'empreinte mémoire de l'arbre préfixe*

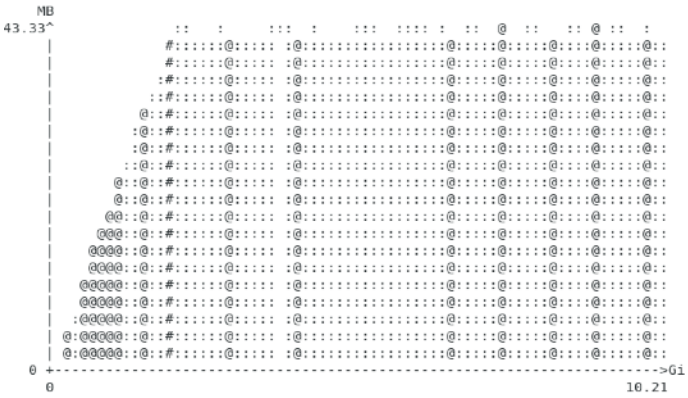


Figure 5 : *Mesure de l'empreinte mémoire de l'arbre radix*

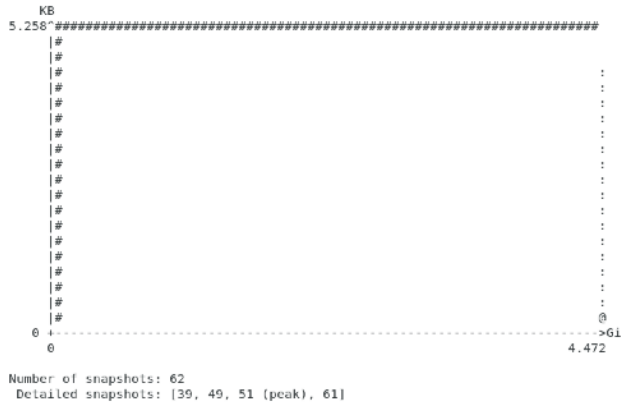


Figure 6 : *Mesure de l'empreinte mémoire de la liste*

Sans surprise la liste à une empreinte mémoire d'environ 3000*5kB=15MB, ce qui est la structure utilisant le moins d'espace mémoire. Vient ensuite l'arbre Prefixe (36MB) puis l'arbre radix (43MB), le fait que l'arbre radix prenne plus de place est surprenant mais cela vient du fait que cette structure n'est pas assez optimisée pour être compétitive par rapport à l'arbre préfix qui elle est optimisée.

b) Implémentation à privilégier selon l'usage

Nous pouvons donc en conclure qu'avec les structures que nous avons implémentés, l'arbre radix ne semble avoir aucun avantage vis à vis de l'arbre préfixe (plus long avec une empreinte mémoire plus important). Ceci est sans doute dût à l'optimisation de cette dernière. En théorie cette structure aurait été la plus intéressante pour construire un arbre contenant un grand nombre d'élément mais peu dense (donc avec des mots très long ce qui ne sera jamais le cas avec des mot de la langue française).

En revanche, grâce à l'optimisation de l'arbre préfixe, celui ci est très intéressant pour contenir une grande base de donnée et ainsiAavoir un arbre dense. Ceci permet de gagner beaucoup de temps par rapport à une structure de liste malgré une empreinte mémoire plus grande mais qui reste dérisoire vis à vis du gain de temps obtenu.

Enfin, la liste reste très avantageuse pour une petite base de donnée et lorsqu'il faut rechercher un petit nombre de mot car cette structure est facile à mettre en place et possède une empreinte mémoire très faible mais elle montre très rapidement ses limites à mesure que la taille de la base de donnée augmente.

Dans notre cas, c'est à dire l'utilisation pour un correcteur orthographique, l'utilisation de l'arbre préfixe est en théorie la plus adapté (grande base de donnée, nombre de mot recherché important et arbre très dense) ce qui ce confirme expérimentalement par la mesure de performance temporelle et en mémoire comme nous l'avons vu précédemment.