# Rapport du projet de SDA2

## **Détection automatique de langue**

L2S4 Informatique

Omar El-Chamaa & Kseniia Kaledina

Pour ce projet, il nous a été demandé d'implémenter un programme de détection automatique de langue. Pour cela, nous avons procédé avec deux structures de données proposées : le *Trie* et le *DAWG* (Graphe orienté acyclique). Notre but est d'essayer d'analyser la situation, pour déduire laquelle de ces 2 structures est la plus adaptée dans notre cas.

# Le Trie

*Le trie* (ou arbre préfixe), est une structure de données sous forme d'arbre qui permet le stockage de chaînes de caractères.

Dans notre cas spécifique, on utilise une structure de données permettant de désigner un nœud dans l'arbre, celle-ci sera composée d'un tableau de pointeurs vers ses 26 fils possibles et un attribut qui nous permet de savoir si ce sommet est la fin d'un mot ou pas (`bool`).

L'implémentation en C de cette structure est assez directe et simple, sans vraiment de subtilités. Pour insérer un élément, il suffit d'ajouter un nœud dans le tableau à l'indice respectif de la lettre s'il n'existe pas, et sinon de se déplacer vers ce nœud s'il est présent. On obtient alors la complexité de $\Theta(n)$ où n est la longueur du mot à insérer.

Pour la recherche, on fait le même parcours que pour l'insertion sauf qu'on vérifie si les nœuds respectifs existent ou pas. On retrouve alors la même complexité que pour l'insertion qui est de $\Theta(n)$.

On évalue alors le temps d'insertion moyen des 3 dictionnaires, et le temps de recherche moyen d'un mot pour cette structure:

Pour le chargement des dictionnaires :

| Dictionnaire: | Temps Moyen: |
|---|---|
| Français: | ~0.148255sec |
| Anglais: | ~0.112639sec |
| Allemand: | ~0.245464sec |

Le temps pour le dictionnaire Allemand est beaucoup plus élevé que les deux autres, ce qui est normal comme ce dictionnaire contient le plus de mots.( 685620 mots pour le dictionnaire allemand, contre 336528 pour le français et 274411 pour l'anglais).

Quant à la recherche, on retrouve un temps moyen uniforme pour tous les dictionnaires, qui est d'environ 0.0000015 secondes.

De plus, comme l'implémentation est assez intuitive, le temps d'exécution n'est pas très important. Le chargement des 3 dictionnaires, la recherche de 7 mots dans chacun et la libération de mémoire se fait en 0.744618 secondes. Par contre la place mémoire prise est très importante, 531,373,576 bytes.

# DAWG

Immédiatement, on voit que l'implémentation d'un *Dawg* n'est pas aussi simple que la précédente.

Premièrement, on retrouve plusieurs structures demandées :

- La structure du DAWG, identique à celle du *trie* avec un id unique en plus .
- La structure d'une *Arête*, qui contient un label (`char`), et deux pointeurs, un vers le sommet gauche et un autre vers le sommet droit.

De plus, il nous a été demandé de manipuler deux autres structures pour nous permettre d'implémenter le DAWG : une *pile* et une *hashmap*.

Il était intéressant de se familiariser avec le fonctionnement de la hashmap, et de l'association clé/valeur, qui nous permet de stocker des sommets de type DAWG.

La première étape était de décomposer l'insertion d'un mot en plusieurs étapes. Le pseudo-code fourni fut très utile quant à la compréhension de ces étapes, mais elles restaient tout de même assez complexes à implémenter. Il fallait tout d'abord commencer par comprendre comment, où, et à quel moment minimiser le DAWG. Pour cela on utilise la structure de pile (`stack`) fournie.

Pour minimiser, il était important de comprendre comment générer une clé, qui dans le cas de cette structure était une chaîne de caractères. Pour cela, on a décidé d'utiliser la forme suivante : 1er bit, 1 si sommet final et 0 sinon, puis on sépare avec ! les arêtes sortantes avec le label et l'id du sommet droit. On obtient alors une clé de la forme : **0!e6/g9**

Ce qui signifie que ce sommet n'est pas final, et dont deux arêtes sortent, un de label 6 vers le sommet d'id 6 et l'autre de label **g** vers le sommet d'id 9.

L'insertion dans le *dawg* étant plus complexe que celle du Trie, sa complexité sera plus importante. Ici, nous suivons les étapes du pseudo-code :

<u>Étape 1</u> : en pire cas, la taille $n$ du plus grand préfixe commun sera de longueur(mot_a_inserer)-1, donc une complexité de $\Theta(n)$ .

<u>Étape 2</u> : On dépile tant que la taille de la pile est supérieure à $p$, donc on a une boucle que l'on répète en pire cas $\Theta(m)$ où m est la taille de pile et p=0.

Les vérifications, l'enregistrement et la liaison se font dans un temps constants $\Theta(1)$ . En meilleur cas on on une complexité $\Theta(1)$ quand la taille de la pile est inférieure ou égale à p.

Étape 3 : On retrouve encore une boucle de taille $p$ (taille du suffixe) dont on va ajouter au graphe. Donc la complexité de $\Theta(p)$ .

Étape 4 : $\Theta(1)$

En conclusion, on se retrouve avec une complexité en pire cas de de $\Theta(n)+\Theta(m)+\Theta(p)$ donc de $\Theta(n+m+p)$ , donc $\Theta(n)$.

Quant à la recherche dans un dawg, on procède de la même façon que pour le Trie, donc avec une complexité de $\Theta(n)$.

On a cependant rencontré un problème dont on ne connaît toujours pas la solution pour l'implémentation en C. En effet, quand on rajoute des mots manuellement au DAWG, tout semble fonctionner normalement et on arrive à insérer des éléments dans la hashmap. Sauf que quand on cherche à charger le dictionnaire, le nombre d'éléments dans cette même table est de 0. La source de cette erreur nous échappe, car on suit les étapes d'insertion et de minimisation à la lettre. On ne prend pas en compte la valeur des temps d'insertions obtenues dans ce rapport car elles sont erronées.

Il nous est donc impossible de comparer les valeurs concrètes d'insertion, de recherche et de la place mémoire prise par cette structure.

On va donc supposer : le temps de recherche est le même que celui des *Tries,* sauf que pour l'insertion, elle aurait pris un peu plus de temps. En revanche, la place mémoire prise par le DAWG serait largement moins importante, ce qui est l'objectif de cette structure.

# Conclusion

En conclusion, on peut dire, que pour notre sujet, il est plus intéressant d'utiliser un DAWG. Cette structure est beaucoup plus adaptée car notre volume de données étant très large, la structure qui utilise moins de place mémoire est privilégiée. Surtout que le temps de recherche pour les DAWG est le même que celui des *Tries*. Il n'y a donc pas vraiment d'avantage à utiliser les *Tries* à part la simplicité d'implémentation.

Ce projet était très enrichissant, car il nous a permis d'appliquer de différentes structures de données apprises en cours dans un cas de la vie réelle. De plus, à cause de la situation sanitaire actuelle, nous avons appris à mieux manipuler les outils de collaboration (Tel que : GIT, Live Share, Discord, etc ...).

Nous aimerions remercier Mr. Haas pour sa réactivité et sa patience envers nos barrages de questions.