Université de Rouen Normandie - Campus Madrillet

Compte-rendu

Compte-rendu
Projet Algorithmique du texte

Membre : Etienne BINGINOT
Membre : Axelle MAZUY

Novembre 2023 — Décembre 2023

Table des matières

1 Présentation de l'archive

Ce projet est composé d'un ensemble de documents spécifiques servant à l'implémentation de l'algorithme d'Aho-Corasick.

1.1 Dossier Trie

On peut ainsi retrouver un dossier nommé Trie, contenant les fichiers de gestion d'un trie par table de hachage et par matrice de transition.

1.2 La gestion par file et par pile

On peut ensuite retrouver un certain nombre de fichiers comme les fichiers queue qui correspondent à la gestion des files et stackTransition s'occupant de la gestion de pile des transitions d'un noeud vers un autre par une lettre.

1.3 Les fichiers ac

Les fichiers ac constituent l'essentiel du projet puisqu'ils correspondent aux fonctions d'implémentation de l'algorithme Aho-Corasick (les fonctions de pré-traitement, complétion, l'algorithme lui-même) mais aussi, étant donné que c'est le point crucial du projet, ce sont dans ces fichiers que l'on retrouve la lecture et le traitement des fichiers donnés en paramètres (le fichier de mots et le fichie de texte).

1.4 Les scripts

Enfin, on peut retrouver des scripts bash (.sh), un script python (.py) qui sont présents pour exécuter l'algorithme et évaluer les performances pour des fichiers de grande taille, des alphabets de toute taille et en faire des courbes.

Le makefile est présent pour compiler le projet et le rendre exécutable. Il s'agit donc d'un point essentiel.

2 Notice d'utilisation

Pour utiliser l'archive de ce projet, il faut tout d'abord se positionner dans le dossier principal et ouvrir un terminal. Il faut ensuite compiler l'ensemble du projet avec la commande **make**.

On trouve ensuite 2 exécutables dans l'archive : ac-matrice et ac-hachage. Pour utiliser l'un ou l'autre de ces exécutables, on utilisera une commmande de la forme suivante :

./ac-matrice mots.txt texte.txt ou **./ac-hachage mots.txt texte.txt**

Ces commandes permettent de lancer la recherche des mots présents dans le fichier mots.txt dans le texte texte.txt, que ce soit en utilisant le trie par table de hachage ou par matrice de transition.

Dans le but de réaliser des statistiques concernant la performance de l'une ou l'autre version des tries, il est également possible de générer des fichiers de mots et texte de longueurs spécifiques et de les appliquer aux deux algorithmes. Pour créer ces fichiers, on utilisera la commande **./executeAlgo.sh**.

Enfin pour obtenir des graphiques à partir des fichiers créés précédemment, on peut utiliser la commande **python3 stat.py**. On aura alors un certain nombre de graphiques similaires à ceux présents dans la suite du document qui s'afficheront et seront présents également dans le dossier graph.

3 Choix d'implémentation

Etant donné que l'on a utilisé comme base les TP1 et 2, nous avons repris entièrement l'implantation des trie (par matrice de transition et table de hachage) en ajoutant seulement quelques fonctions qui nous ont permis une certaine factorisation et que nous avons réutilisé par la suite dans nos fonctions concernant l'algorithme d'Aho-Corasick. Parmi ces fonctions, il s'agit notamment de simplement récupérer le noeud de trie correspondant à la transition du noeud de départ vers ce noeud en une lettre donnée, une fonction créant une transition dans le trie, ou encore des méthodes requêtes (pour consulter le dernier noeud ajouté dans le trie, vérifier si un noeud est final ou non).

Une autre prise de partie a été le choix d'implémenter une pile pour la gestion des transitions. Nous avons donc créer un type transition qui est géré en pile, et chaque fois qu'une transition est consommée, elle est sortie de la pile. Etant donné que dans l'algorithme, la dernière transition ajoutée est la première utilisée, cela nous semblait correspondre davantage à une pile qu'à une liste.

Pour le calcul du nombre de mots reconnu par un état final pour l'algorithme d'Aho-Corasick, nous avons choisi de nous éloigner légèrement de l'algorithme de base. En effet, nous ne pouvions pas de façon efficace utiliser une gestion ensembliste pour connaître les mots reconnus à chaque état. Nous avons donc choisi d'utiliser un tableau pour cette partie là. Ce tableau contient pour chaque état le nombre de mots qu'il reconnaît (un état final a donc au moins une valeur de 1).

Pour réaliser une factorisation de notre code, nous avons choisi d'exploiter les possibilités du makefile. En effet, une de nos premières implantations était d'avoir deux fichiers c Aho-Corasick identiques mais avec des include différents. Cette solution n'étant pas satisfaisante, nous avons modifié notre makefile en faisant en sorte de compiler nos executables Aho-Corasick avec le même fichier d'implantation de l'algorithme mais des Trie différents, ce qui permet ainsi d'avoir une seule implantation pour l'algorithme.

4 Difficultés rencontrées

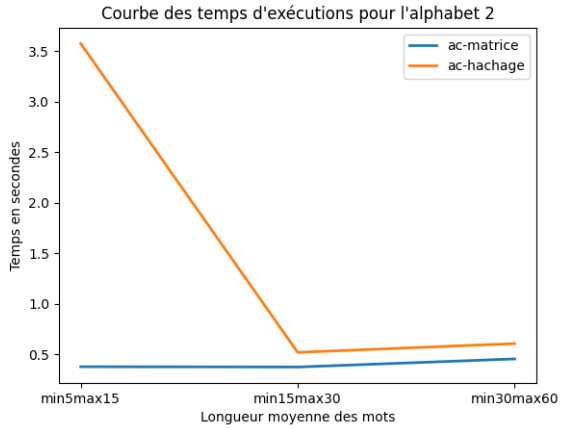
L'une des difficultés principales rencontrées a été la gestion de la génération aléatoire. En effet, lorsque l'on exécutait le programme sur des textes et mots générés, nous pouvions être confrontés à la présence de chacun des mots dans le texte, autrement dit le texte n'était qu'une concaténation de l'ensemble des mots générés. Cela était dû à une mauvaise initialisation de la graine random, ainsi le programme s'exécutant trop vite, le random n'avait pas le temps de changer.

Il était également important de porter une attention particulière à la libération de la mémoire. En effet, que ce soit par le biais d'un trie par une table de hachage ou par une matrice de transition, il y avait déjà beaucoup d'allocations mémoires. Mais on retrouvait également des allocations pour les piles et les files, ou encore les chaînes de caractères notamment pour le texte. Valgrind a ainsi été d'une grande aide pour contrôler la mémoire libérée.

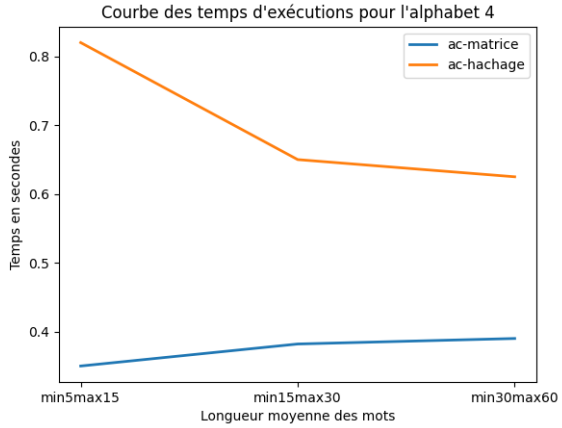
5 Analyse des résultats courbes

5.1 Les temps d'exécution en fonction de la longueur moyenne des mots

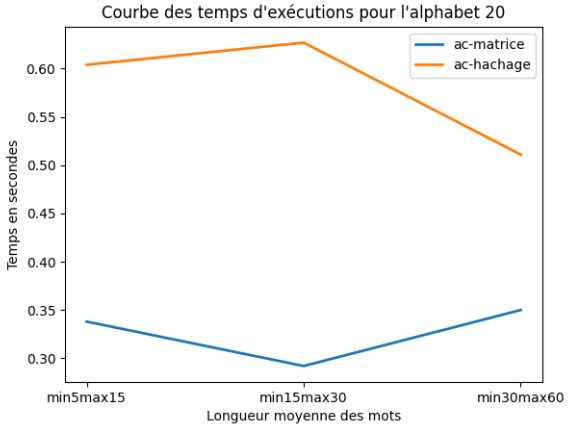
Graphique pour un alphabet de taille 2



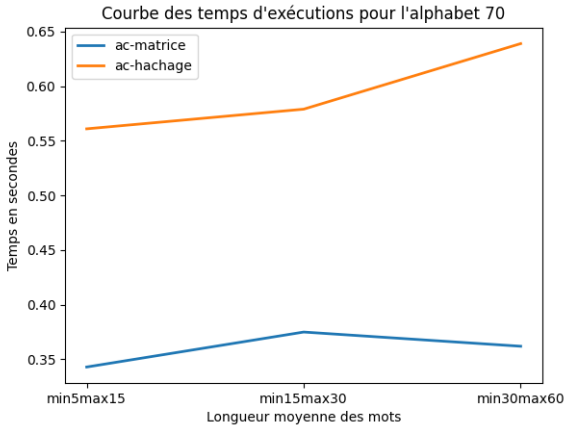
Graphique pour un alphabet de taille 4



Graphique pour un alphabet de taille 20



Graphique pour un alphabet de taille 70



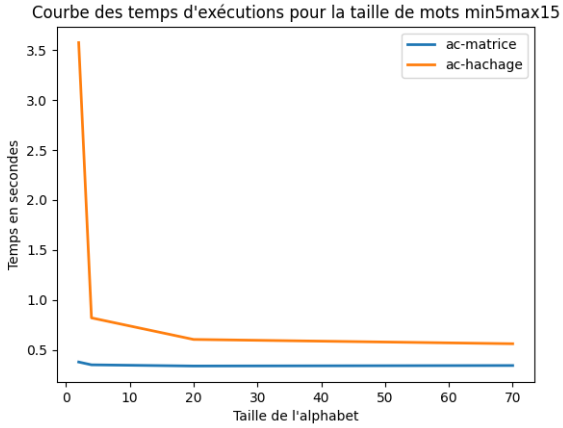
Contrairement à ac-matrice où le temps d'exécution augmente avec la taille du mot, on peut voir qu'ac-hachage a tendance à être plus rapide ou tout du moins ne pas augmenter aussi vite qu'ac-matrice.

En effet la table de hachage est optimisée pour des mots de longueur plus importante car son temps d'exécution n'augmente pas avec la taille des mots. Celui-ci utilise une table de hachage basée sur son nombre de noeuds et va donc allouer en temps réel des cellules pour chaque mot.

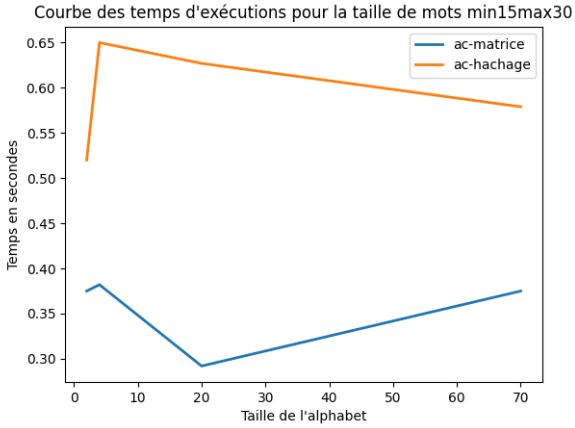
ac-matrice lui doit utiliser une matrice et doit donc allouer un immense tableau ce qui lui fait prendre beaucoup d'espace mémoire. Cependant, ce tableau lui permet d'avoir accès aux cellules en temps constant, lui permettant une efficacité importante.

5.2 Les temps d'exécution en fonction de la longueur moyenne des alphabets

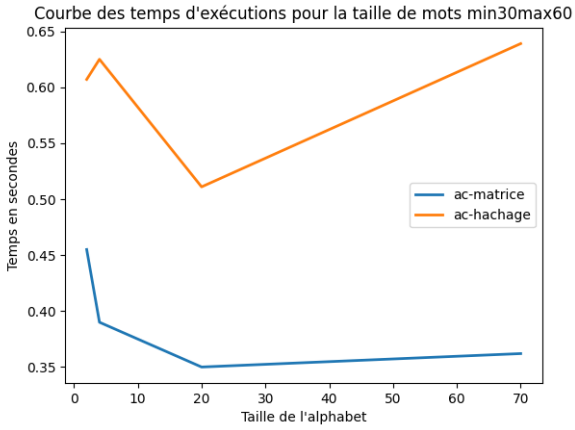
Graphique pour un mot de taille moyenne entre 5 et 15



Graphique pour un alphabet de taille moyenne entre 15 et 30



Graphique pour un alphabet de taille moyenne entre 30 et 60



On peut voir que peu importe la taille du mot, l'algorithme semble garder la même efficacité, bien que ac-matrice garde son avance sur ac-hachage avec l'utilisation des matrices. Les valeurs extrêmes que l'on peut parfois observées sont causées par l'ordonnanceur du processeur, qui peut mettre du temps à réagir.