SORBONNE UNIVERSITÉ
CRÉATEURS DE FUTURS DEPUIS 1257

Projet 1: eGrep clone

Oussama Bouibker, Alexandre Em

October 10, 2021

1 Introduction

Ce projet consiste à reproduire le comportement simplifié de la commande egrep. Cette commande est une dérivé de la commande grep ou *Global Regular Expression Print* qui permet rechercher dans un ou plusieurs fichiers les lignes dont l'un de ses mots correspondrait au pattern saisie par l'utilisateur. Un Regex (Regular expression) est un moyen de spécifier un ensemble particulier de lettres ou de mots à partir d'un texte. Lors de la saisie, le pattern doit être entre guillemet. `egrep` quand à lui, est utilisé pour les recherches nécessitant une syntaxe d'expressions rationnelles plus sophistiquée. Il existe plusieurs manières d'effectuer une recherche, mais nous nous sommes inspiré dans ce projet du livre *Aho - Compilers - Principles, Techniques, and Tools*.

2 Algorithme utilisé

2.1 Analyse et présentation théorique

L'algorithme du livre permet de transformer l'argument qui est une chaîne de caractère (Regex) en un arbre syntaxique, où l'on sépare opérateur et lettre tout en prenant en compte l'ordre de profondeur de ceux-ci. À partir de cet arbre, nous construisons un automate fini non déterministe qui nous permettra ici d'avoir le chemin complet de l'automate comprenant les noeuds, les lettres et les symboles Regex, pour ensuite avoir l'automate déterministe fini sur lequel nous vérifierons qu'un mot match avec le Regex ou non. Nous avons fait le choix de cet algorithme, car il est assez comprehensible et representatif du comportement du grep.
Afin de faire la déterminisation de l'automate, nous parcourons le NFA à partir de l'état initial généralement noté 0 et nous ajoutons chaque état étant sur une ϵ transition jusqu'à rencontrer une transition contenant un caractère, mais aussi les états suivant comprenant des ϵ transitions. Tout ces états formeront un état dans le nouvel automate DFA. Le principe étant de fusionner les états contenant des ϵ transitions tout en conservant le comportement souhaité du Regex.

2.2 Implémentation

Lors de notre implémentation des différents tests, nous avons utilisé le test-driven development (TDD) qui est une méthode de développement qui consiste à écrire des tests unitaires avant toute implémentation de fonctionnalité, ce qui permet d'avoir une idée claire de ce que nous voulions implémenter en proposant des exemples (et en quelque sorte une sorte de documentation), mais aussi de vérifier que la méthode reste valide à chaque nouvelle fonctionnalité implémenté.

Pour générer l'automate déterministe fini décrit dans la sous partie précédente, nous implémentons en premier lieu le passage de la chaîne de caractère Regex en arbre, celui-ci va parcourir cette chaîne et pour chaque symbole rencontré, va placer en son noeud le symbole, et dans sa liste, le reste de la chaîne Regex avec comme objet en tête de liste l'élément droit de l'arbre syntaxique et le 2-ème objet comme élément gauche de cet arbre. Ceci sera représenté dans l'affichage console avec des parenthèses comprenant le corps de l'arbre et une virgule séparant les éléments droit et gauche.
Le passage de l'arbre syntaxique en automate permet le parcours de chaque noeud de l'arbre pour récupérer toutes les transitions de l'automate. Pour chaque noeud parcouru, nous notons les numéros des transitions correspondant aux lettres dans le tableau. Si aucune transition n'a été détecté pour une lettre donné sur un noeud, la valeur -1 sera stocké. Les ϵ transitions correspondent a des transitions neutre permettant de concaténer les arbres des différents symboles Regex.
Pour la transformer le NFA en DFA implémenter dans la classe `DFA`, nous commençons par récupérer toutes les lettres disponibles dans le regex, qui nous serviront par la suite de générer les états du DFA. Nous avons choisi de l'implémenté sous forme de HashMap qui à partir d'une Arraylist en clé, qui correspond au noeud du DFA d'obtenir ses transitions qui sont représentées par une HashMap de Character qui sont les lettres possibles dans le regex que l'on a récupérer au début du code en clé et en valeur une ArrayList contenant les états qui sont connexes et contenant une ϵ transition et qui représente un autre état du DFA qui est exploré si la HashMap principale contient cette ArrayList dans sa liste de clé sinon elle ne l'est pas et on l'explore récursivement. (voir example *figure 1*). La recherche des états contenant des ϵ transition et connexe à l'ensemble des états courant à été implémenté dans la méthode `epsilonClosure` et la recherche des états contenant le caractère c et connexes aux états dans l'ensemble d'états courant.
À partir de cet automate, il est beaucoup plus facile de traiter la recherche de pattern, car il nous a suffit de parcourir le fichier texte de n lignes, qui comporte moins de m mots pour chacune (avec $m < 100$). Lors du parcours, on split la ligne afin d'obtenir une liste de mot, puis nous parcourons pour chaque mot, nous parcourons l'automate avec chacune de ses lettres jusqu'à ce que l'automate ne trouve pas la lettre en question. Si la lettre fait partie d'un état qui est dit finale, alors le regex est valide et on imprime la ligne, sinon on passe au mot ou à la ligne suivante, selon la position du pointer.

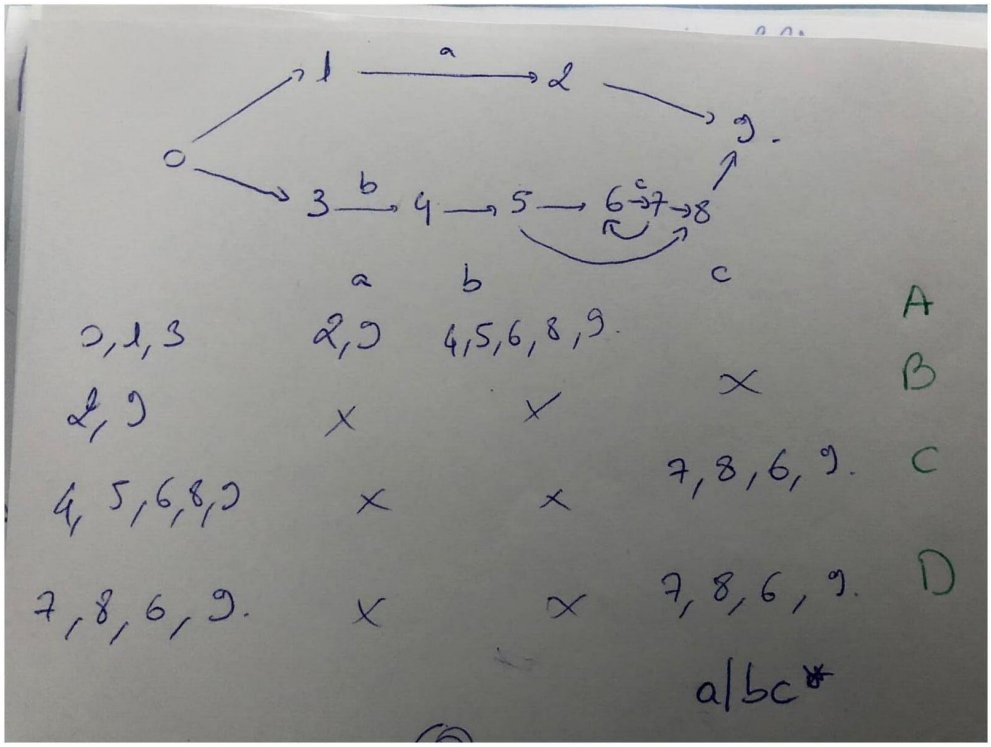

Figure 1: Tableau généré avec comme regex $a—bc*$

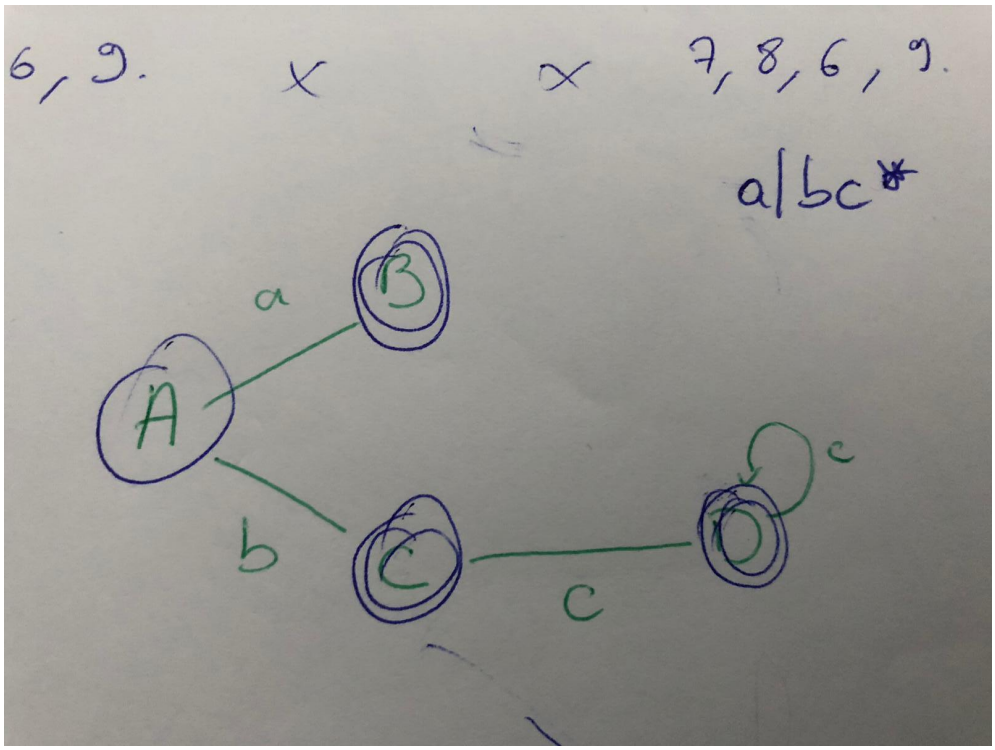

Figure 2: Automate déterministe fini obtenu avec le tableau généré

3 Phase de test

3.1 Test de performance

Pour les instances de test, nous avons récupéré des eBook sur la base de Gutenberg dont le nombre de ligne varie fortement, afin de pouvoir évaluer l'impact de ce paramètre avec le nombre de mots matché et du temps d'exécution de notre programme en milli-secondes. Pour chaque test, nous avons effectué une recherche sur des Regex de niveau de "difficultés" différents, qui sont les suivant: *to*, *the*, *the|or*, *a*, *or** et *t|a|o|r*. On compare ici notre programme avec la véritable commande *egrep*:

Instance de test: *The omnipotent self, a study in self-deception and self-cure by Paul Bousfield*, contenant 3591 lignes

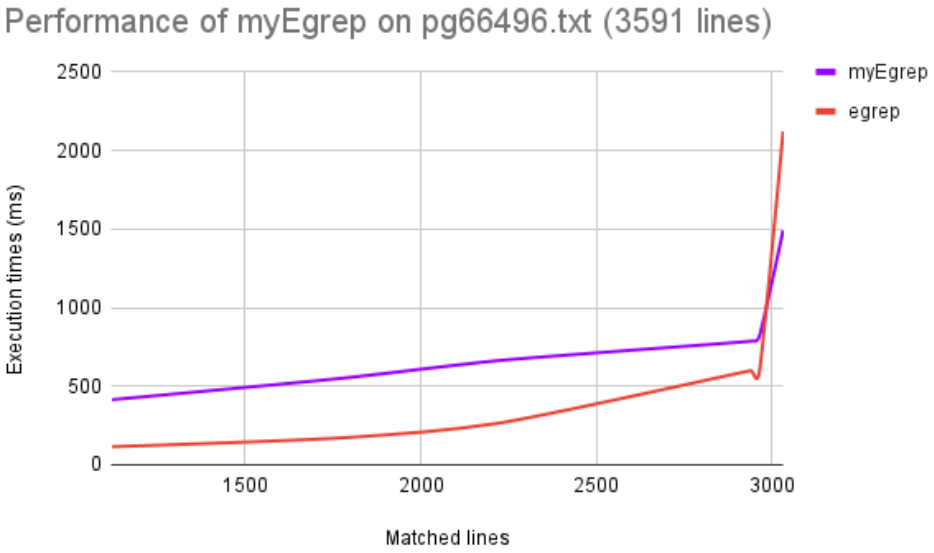

Figure 3: Diagramme de performance pour un fichier de 3591 lignes

Instance de test: *The Boy and the Baron by Adeline Knapp*, contenant 3786 lignes

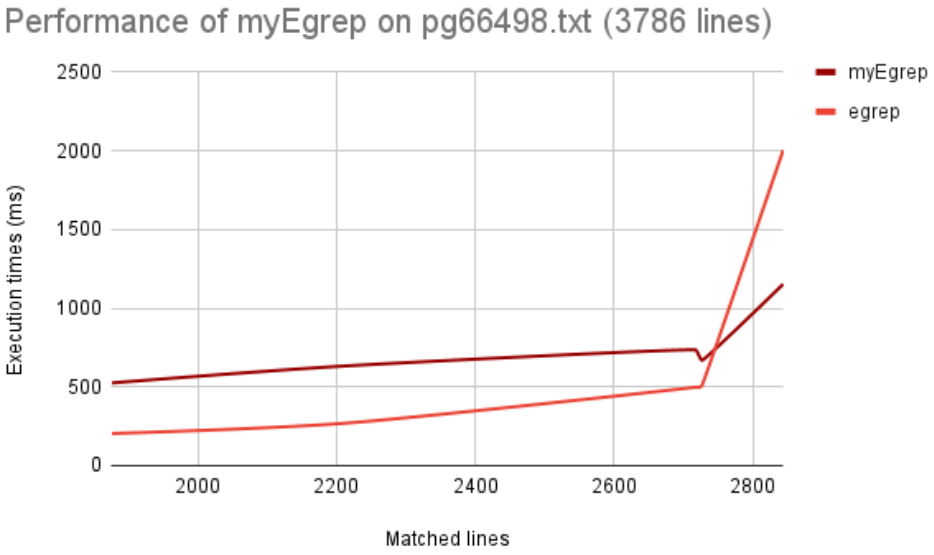

Figure 4: Diagramme de performance pour un fichier de 3786 lignes

Instance de test: *La Chèvre d'Or by Paul Arène* contenant 5832 lignes

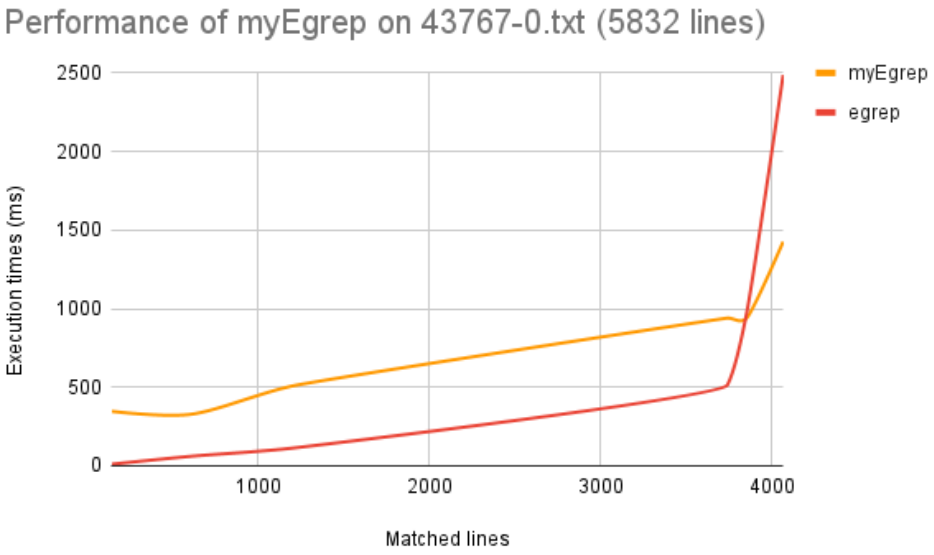

Figure 5: Diagramme de performance pour un fichier de 5832 lignes

Instance de test: *With Sword and Crucifix; Being an Account of the Strange Adventures of Count* contenant 7450 lignes

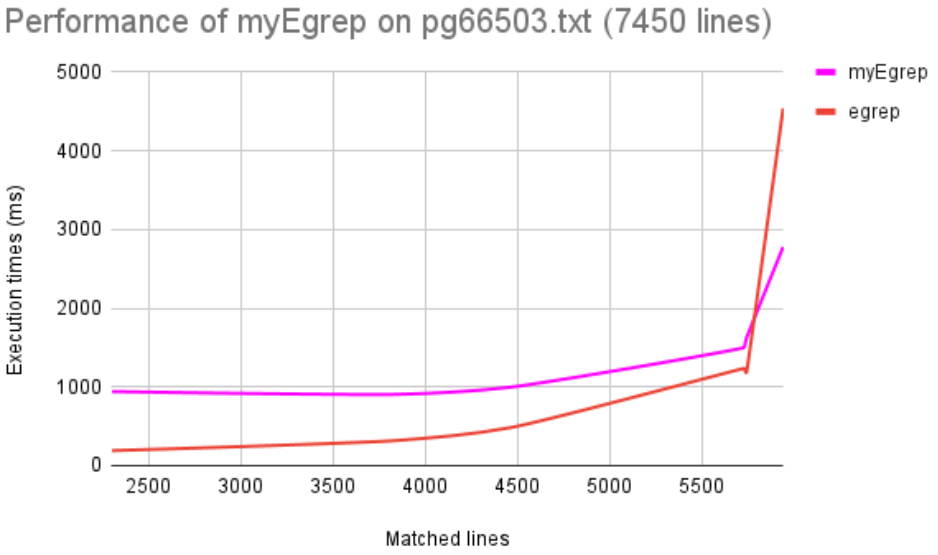

Figure 6: Diagramme de performance pour un fichier de 7450 lignes

Instance de test: *Ravachol oder die Pariser Anarchisten by Arthur Holitscher* contenant 10962 lignes

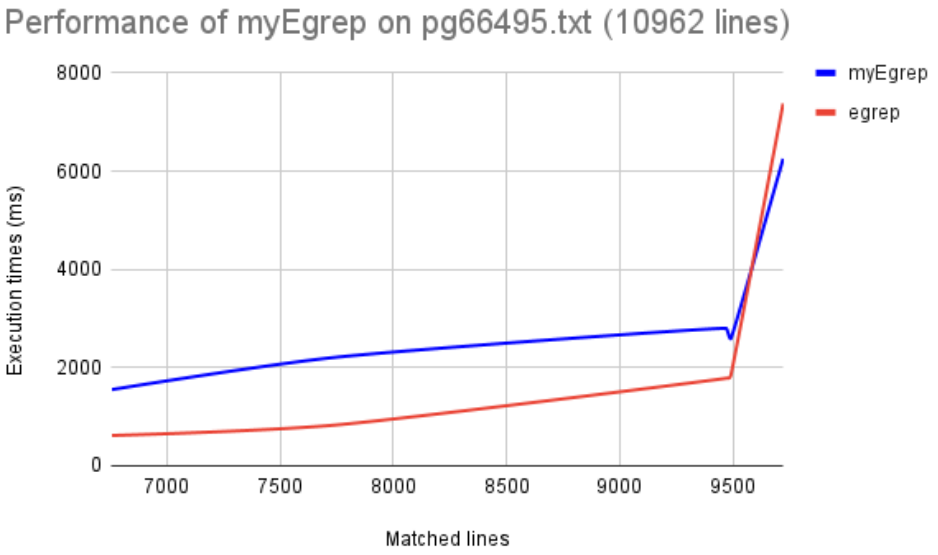

Figure 7: Diagramme de performance pour un fichier de 10962 lignes

Instance de test: *A History of Babylon, From the Foundation of the Monarchy to the Persian Conquest History of Babylonia vol. 2* contenant 13308 lignes

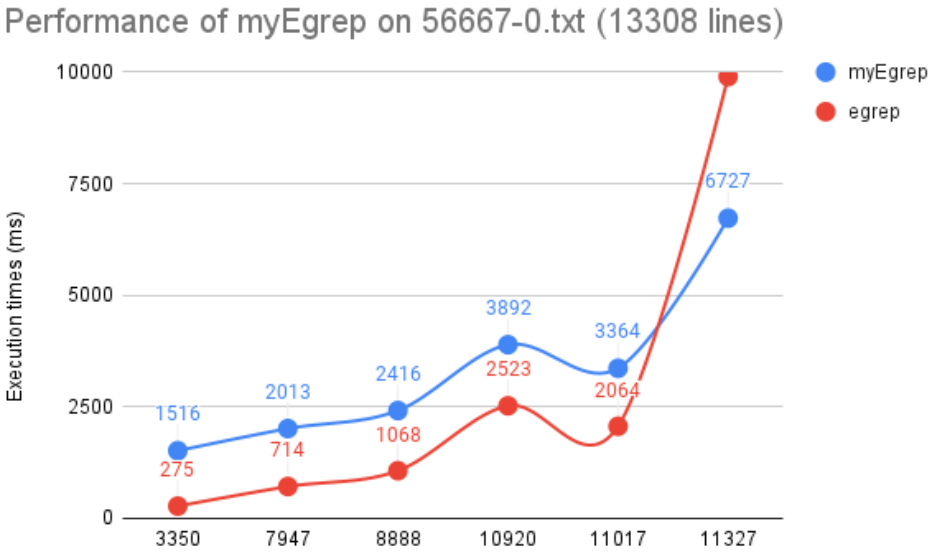

Figure 8: Diagramme de performance pour un fichier de 13308 lignes

3.2 Observation

On peut observer sur les différentes courbes, que globalement la commande *egrep* est plus performante en temps d'exécution que notre programme, mais que les temps sont relativement proches et ont la même allure lorsque le Regex est de "faible" niveau. On peut aussi voir que le temps d'exécution de chacun commence à tendre vers une allure exponentielle lorsque la difficulté devient importante, mais que notre programme gagne en temps d'exécution. On peut aussi observer en comparant chaque courbe que le nombre de lignes total du fichier n'impacte peu le temps d'exécution contrairement au nombre de ligne contenant un mot matchant le pattern et donc au Regex saisie par l'utilisateur.

4 Conclusion

`egrep` est un outil offline qui permet d'effectuer des recherches de pattern dans un ou plusieurs fichiers. Notre implémentation de egrep, nous a permis d'avoir un aperçu du fonctionnement de egrep en profondeur pour des cas simple. Nous avons pu voir que le temps d'exécution de notre programme est un peu plus long que la commande unix, pour des cas de Regex très simple. On pourrait par la suite minimiser l'automate (DFA) et réduire au maximum le nombre d'état dans l'automate afin d'observer le nouveau temps d'exécution afin de les comparer. Mais aussi compléter les pattern et corriger ceux qui ne fonctionnent pas correctement tel que la répétition "+".