Sorbonne Université

Projet DAAR

Clone de egrep

Rapport de Projet

Auteurs :
Ning GUO

14 Octobre 2019

Table des matières

1 Introduction

1.1 La commande egrep d'Unix

La commande Unix egrep permet de chercher un motif dans un texte. Chaque ligne tapée est
— soit recopiée par egrep, si le motif s'y trouve
— soit ignorée
Le motif peut être défini par la suite de ses caractères, mais il peut être défini aussi par une expression rationnelle.

egrep <motif> <nom-fichier>

1.2 Expressions régulières

une expression régulière est une chaîne de caractères, qui décrit, selon une syntaxe précise, un ensemble de chaînes de caractères possibles. Ces ensembles peuvent contenir :
— des suites de caractères définies explicitement : abc
— des répétitions de suites de caractères : *
— des choix entre plusieurs suites de caractères : |
— Un seul caractère indéfini : .
Exemple : a(b|c)*d

2 Transformation d'automates

2.1 Définition du problème et la structure de données utilisée

— **Définition du problème**
Soit une expression régulière définissant un langage
rechercher les sous-chaînes appartenant au langage ?

— **Structures de données**
Automates finis

— **Algorithmes**
Transformation des expression régulières aux automates

2.2 Analyse et présentation théorique des algorithmes

Théorème : tout langage regulier est accepte par un automate fini deterministe. –> Possibilité de transformer une expression régulière en automate.

Dans cette partie, nous allons voir comment construire un automate fini deteministe mininal à partie d'expression regulieres.
On appliquera les 5 étapes suivants :

1. Etape 1
   nous allons d'abord transformer le motif en un arbre de syntaxe (voir automate.RegEx.java)

2. Etape 2
   puis en un automate fini non déterministe avec ϵ-transitions selon la méthode Aho-Ullman
3. Etape 3
   puis en un automate fini déterministe avec la méthode des sous-ensembles
4. Etape 4 : Minimisation d'un automate fini
   L'objectif de cette étape est de trouver une representation minimale, unique, (canonique) de l'automate reconnaissant un langage regulier.

   L'idée de l'algorithme va être de partitionner l'ensemble des états d'un automate fini déterministe. Pour trouver cette partition, nous allons procéder par raffinements successif. Au départ, il n'y aura que 2 classes, les états appartenant à F (les états finaux) et les états appartenant à Q-F (les états non finaux). À chaque étape, on vérifie que pour tout couple d'états $q, q\prime$ appartenant à la même classe, on a la propriété $\forall a \in \Sigma, (q, a), (q\prime, a)$ sont dans la même classe, si ce n'est pas le cas $q, q\prime$ ne sont pas dans la même classe et on rejette $(q, q\prime)$.
   Le processus s'arrête quand il n'y a plus aucun retrait.
5. Etape 5
   On utilise l'automate minimal obtenu par étape précedent pour tester si un suffixe d'une ligne du fichier textuel est reconnaissable par cet automate.

3 Algorithme de KMP

3.1 Définition du problème et la structure de données utilisée

Définition du problème

— Facteur = expression régulière avec uniquement concaténation
— Recherche de facteur = Trouver une chaîne de caractères (le motif C) à l'intérieur d'une autre chaîne (le texte T).

3.2 Analyse et présentation théorique des algorithmes

3.2.1 Présentation de KMP

L'algorithme de Knuth-Morris-Pratt (KMP) est un algorithme de recherche de sous-chaîne de caractères, permettant de trouver les occurrences d'une chaîne C dans un texte S avec une complexité linéaire $O(|C|+|S|)$ dans le pire cas. Sa particularité réside en un pré-traitement de la chaîne, qui fournit une information suffisante pour déterminer où continuer la recherche en cas de non-correspondance. Ainsi l'algorithme ne ré-examine pas les caractères qui ont été vus précédemment, et donc limite le nombre de comparaisons nécessaires.

3.2.2 Phases

1. La première phase de l'algorithme construit un tableau, indiquant pour chaque position un « décalage », c'est-à-dire la prochaine position où peut se trouver une occurrence potentielle de la chaîne.
2. La deuxième phase effectue la recherche à proprement parler, en comparant les caractères de la chaîne et ceux du texte. En cas de différence, il utilise le tableau pour connaître le décalage à prendre en compte pour continuer la recherche sans retour en arrière.

3.3 Complexité

3.3.1 Complexité de l'algorithme de construction du tableau

La complexité de l'algorithme de construction du tableau est O(n), où n désigne la longueur de la chaîne C.

3.3.2 Complexité de l'algorithme de recherche

En supposant l'existence préalable du tableau T, la phase « recherche » de l'algorithme de Knuth-Morris-Pratt est de complexité O(l), où l désigne la longueur du texte S.

3.3.3 Complexité de l'algorithme de Knuth-Morris-Pratt

Comme les deux parties de l'algorithme ont respectivement des complexités de O(n) et O(l), la complexité de l'algorithme dans sa totalité est O(n+l).

4 Radix Tree

4.1 Définition du problème et la structure de données utilisée

— **Structures de données**
un arbre radix

— **Algorithmes**
Transformation des texts en arbre radix

4.2 Analyse et présentation théorique des algorithmes

Un arbre radix est une structure de données compacte permettant de représenter un ensemble de mots adaptée pour la recherche. Il est obtenu à partir d'un arbre préfixe en fusionnant chaque nœud n'ayant qu'un seul fils avec celui-ci.
Pour construire l'arbre radix, on appliquera 2 étapes suivants :

1. Construction de map
   la clé est un mot, et la valeur correspondant sont tous les index de ce mot dans le texte
2. Construction de l'arbre radix à partir de ce map de l'étape 1

4.3 Complécité

On supposerque les mots sont de longueur k et que le nombre de mot est n. Les opérations de recherche et d'insertion d'un mot ont des complexités en O(k). Par conséquent, la complexité temporelle de ces opérations ne dépend pas du nombre de données contenues par l'arbre radix.

5 Test de performance

5.1 Outil utilisé : gnuplot

Gnuplot est un bon outil à utiliser en ligne de commande pour visualiser des données.

5.2 Résultat du test

5.2.1 Comparaison de performance entre Automate, KMP et Radix Tree

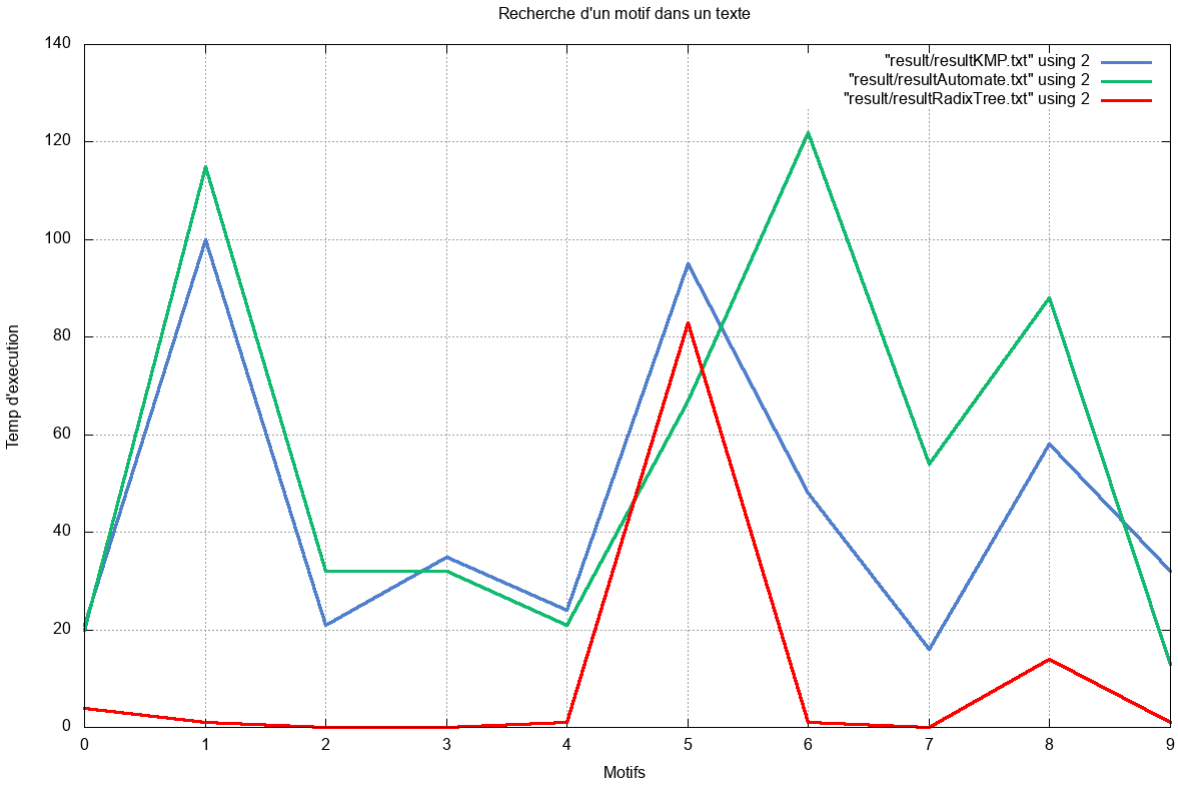

FIGURE 1 – Temps d'exécution pour les 3 algorithmes implementés

Dans ce graphe :

— x représente les differentes mots à rechercher.

— y représente le temps d'exécution correspondant à ce motif.

Par le résultat dessus, on remarque que si on compte pas le temps de construction d'un arbre radix, juste pour chercher un mot, c'est arbre radix qui est le plus vite.

5.2.2 Comparaison de performance entre egrep et KMP

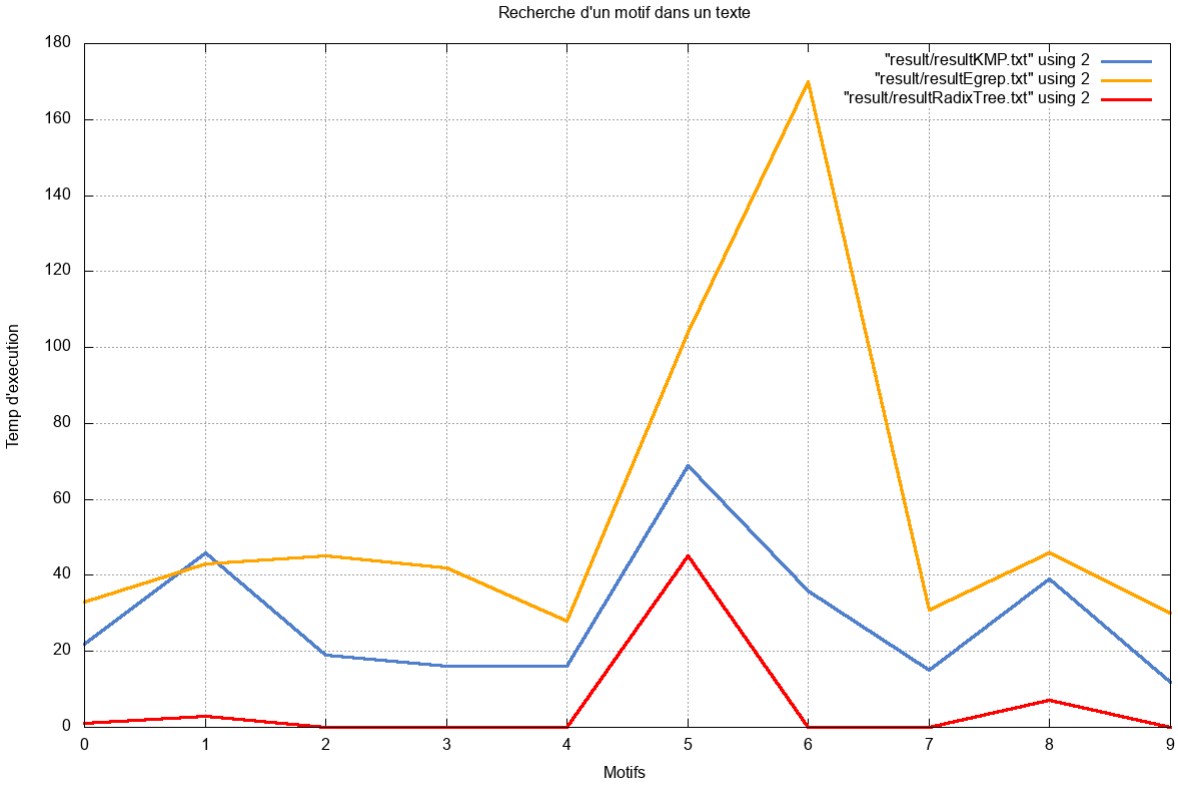

FIGURE 2 – Temps d'exécution pour KMP, egrep et Radix Tree

Dans ce graphe :
La courbe orange représente le temps d'exécution de commande egrep, la courbe bleue est pour KMP, et la rouge est pour Radix Tree.
On peut conclure que : Si le motif est réduit à une suite de concaténations, il s'agit de la recherche d'un facteur dans une chaine de caractères -> c'est plus efficace d'utiliser l'algorithme Knuth-Morris-Pratt (KMP) ou l'algorithme de l'arbre radix à la place des automate.

5.2.3 Pour les expressions régulières

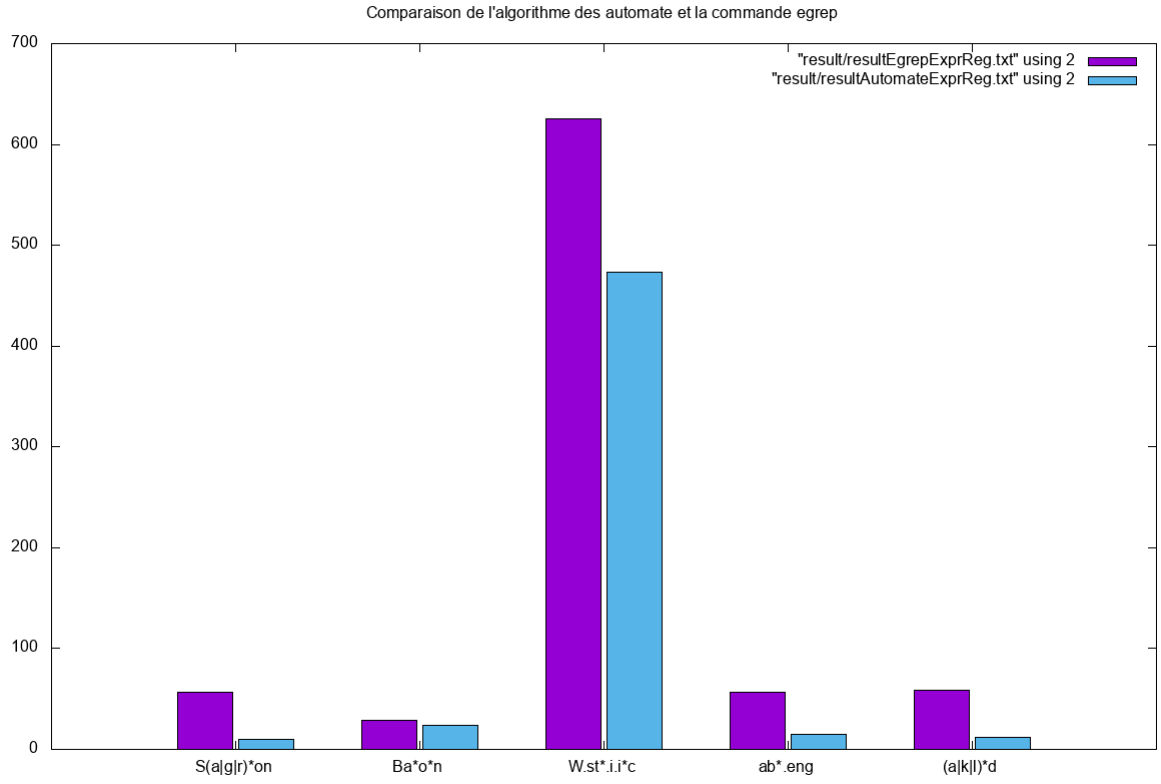

FIGURE 3 – Comparaison entre egrep et Automates

le violet représente l'algorithme des automate, le bleu représente commande egrep.
On trouve que pour les expressions régulières, l'algorithme des automates implémenté est plus efficace que egrep en temps d'exécution.