Clone de Egrep avec support partiel des ERE.

Le : Vendredi 13 novembre 2020

Auteure : Mylène Balech

Table des matières

Présentation

La commande EGREP d'Unix permet de faire une recherche sur un motif spécifique et d'en ressortir les résultats correspondants.
Cette fonctionnalité permet de rechercher aussi bien des expressions régulières que des chaines de caractères.
Pour rechercher un motif dans un texte, la recherche s'effectue autant de fois que le fichier possède de lignes.

Le but de ce devoir est d'implémenter un clone de la commande Egrep en excluant quelques formes d'expressions régulières car elles sont assez fastidieuses à implémenter.

1. Recherche de motif par RegEx

Qu'est-ce qu'une expression régulière ? Une expression régulière est une représentation d'un langage algébrique reconnaissable par un automate fini déterministe.

Qu'est-ce qu'un automate fini ? Un automate fini est une construction abstraite, susceptible d'être dans un nombre fini d'états. Le passage d'un état à un autre se fait par un événement et est appelé transition.

Qu'est-ce qu'un automate fini déterministe ? C'est un automate fini dont les transitions à partir de chaque état sont déterminées de façon unique par son point d'entrée.

Qu'est-ce qu'un automate fini non déterministe ? C'est un automate fini dont les transitions entre les états ne sont pas uniques.

Afin de faire une recherche de motif par RegEx, plusieurs étapes de constructions sont nécessaires afin d'obtenir l'automate fini déterministe.
Nous devons suivre les étapes de constructions proposées par Aho-Ullman, qui sont la création d'un automate non déterministe avec des ε-transitions (NDFA), puis la création à partir du NDFA de l'automate fini déterministe (DFA) puis créer un DFA avec un minimum d'état.

2. Structures de données utilisées

a. RegEx par automates selon Aho Ullman

Afin de procéder à la construction de l'automate, le motif recherché est transformé en un arbre de syntaxe.
Une classe nommée RegEx, crée un arbre unique possédant, une racine et des branches.
Une branche est aussi une RegEx qui contient une racine et des branches, etc.

i. NDFA

Pour générer le NDFA, une classe nommée Epsilon est créée qui permet de stocker la source de départ, sa destination, l'évènement (ε ou un caractère), sa racine (si elle appartient à un ou, une étoile ou une concaténation) et si cette transition est finale ou pas.

Pour stocker toutes les transitions, un tableau est créé et à chaque nouvelle transition détectée celle-ci est ajoutée dans le tableau.

ii. DFA

Une classe nommée EtatDFA, permet de créer deux types d'état différents utilisé à divers endroits. Dans un premier temps, elle permet de créer un état possédant une source sous forme d'entier, sa liste de destination qui est un tableau d'entier, le chemin emprunté par un caractère et si cette transition est finale en booléen.
Dans un second temps, cette classe permet aussi de créer un état possédant, un tableau d'entier de sources, un tableau d'entier de destinations, un chemin sous forme de chaine de caractère et un booléen qui permet de spécifier si la transition est finale.

Un tableau d'EtatDFA est créé et permet de stocker toutes les transitions.

iii. DFA avec minimum d'états

Afin de créer un automate avec un minimum d'états et qu'il soit compréhensible, une classe nommée EtatRenamed, qui créée une source sous forme d'entier, une destination sous forme d'entier, un chemin stocké sous forme d'entier, et un booléen qui permet de dire si ce chemin est final.

Afin de créer le tableau des équivalences à l'aide du puit, une nouvelle classe est nécessaire, c'est la classe TableauEquivalence. Pour chaque équivalence, il y a, un tableau d'entiers de départ, un tableau d'entiers des arrivées, un entier qui représente le chemin ainsi que le booléen représentant si les deux chemins sont équivalents est créé.

Afin d'obtenir un tableau d'automate, la classe automate est utilisée de deux façons, une fois lorsque les états sont encore des listes d'états et une autre fois lorsque la liste des états est transformée en un identifiant unique.

Chaque ligne d'automate final est représentée par une source, une destination, un chemin et s'il est final.

iv. Lecture du fichier

Chaque ligne du fichier texte est convertie en un tableau de string.
Il permet de convertir chaque caractère en ascii et de statuer s'il fait partie du motif.

b. KMP

i. Création des états du préfixes

Le préfixe doit être de type chaine de caractère.
Afin de créer le tableau le tableau des états du motif recherché, un tableau d'entier est créé.

ii. Recherche le facteur dans le texte

Afin de chercher le motif à l'aide du tableau du préfixe, le texte est transformé en tableau de chaine de caractères.

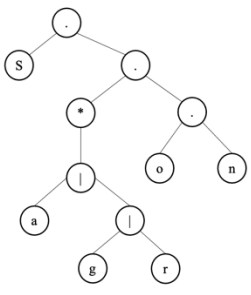
Pour récupérer les résultats trouvés correspondant au motif, un tableau de chaine de caractères est créé.

3. Analyse et présentation des algorithmes connus

a. RegEx d'Aho-Ullman

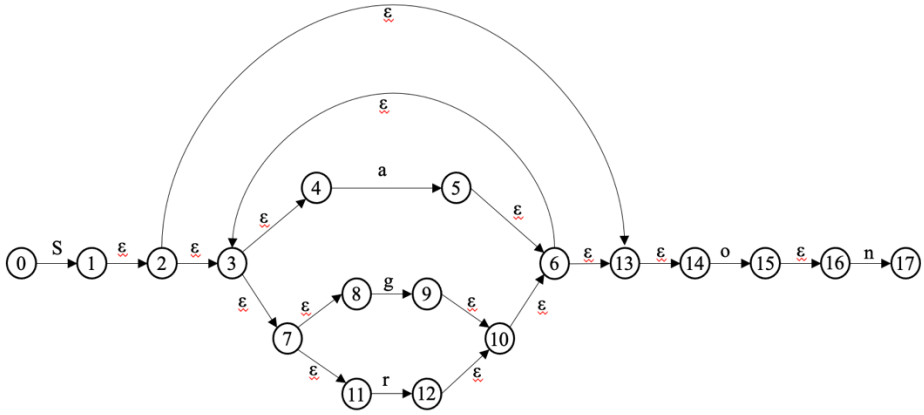
Afin d'avoir un automate fini déterministe, Aho-Ullman proposent un ensemble d'étapes.
Tout d'abord, il faut créer un arbre syntaxique pour le motif recherché.

Exemple pour S(a|r|g)*on : S(a|r|g)*on → .(.(.(S,*(|(|(a,r),g))),o),n)



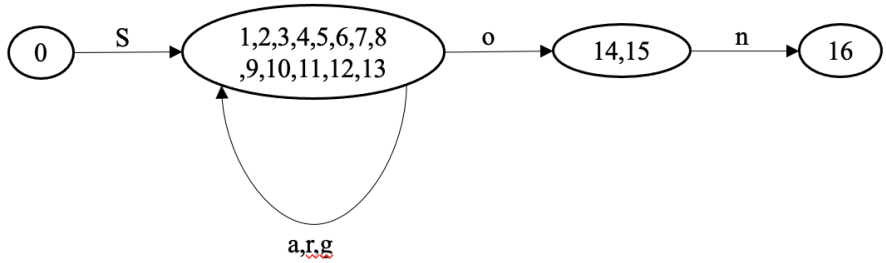
Lorsque nous avons l'arbre syntaxique représenté, il faut créer l'automate fini non déterministe avec les ε-transitions, en suivant les règles décrites si c'est un caractère seul, une étoile, un ou et une concaténation.

Exemple pour S(a|r|g)*on :



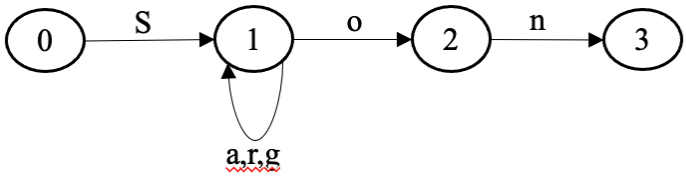
Avec l'automate fini avec ε-transitions, le tableau d'équivalence est créé (il ne sera pas représenté ici). Puis, nous devons créer l'automate fini déterministe.

Exemple pour S(a|r|g)*on :



Cet automate n'est pas final car il faut ensuite renommer les différents états.

Exemple pour S(a|r|g)*on :



Ceci est l'automate déterministe avec le minimum qui sera utilisé pour rechercher le motif dans le texte.

b. KMP

Pour créer le tableau des états du préfixe, on considère que celui-ci à n caractères.

On définit que le premier élément -1 du tableau est égal à -1. On déclare deux variables, une j qui est égale – 1 et i qui est égale à 0.

Tant que i n'est pas égal à la taille du préfixe, alors on compare le caractère à l'index i et celui à l'index j du préfixe.
Si les deux sont égaux alors l'enregistrement du tableau des états à l'index i prend la valeur j+1, et i et j sont incrémentés de 1.
Sinon si j est supérieur à 0, j prend la valeur du tableau des états à l'index j-1.
Sinon le tableau à l'index i prend la valeur 0 et i est incrémenté alors que j prend la valeur 0.

Si i à la même valeur que la taille du préfixe, la boucle se termine.

T[-1] = -1
i = 0
j = -1
Tant i < taille P faire
    Si P[i] = P[j] alors T[i] = j+1 et i = i+1 et j = j+1
    Sinon si j > 0 alors j = T[j-1]
    Sinon T[i] = 0 et i = i+1 et j = 0
Fin tant que

Cet algorithme permet de rechercher seulement des chaines de caractères, il n'est pas adapté pour la recherche de motif RegEx.

4. Argumentation sur les différents algorithmes

a. L'automate fini déterministe d'Aho-Ullman

Facile à comprendre mais difficile à implémenter, lorsqu'il est fonctionnel, il permet d'effectuer des recherches sous formes de RegEx dans un texte.

Créer un automate fini non déterministe avec des ε-transitions, permet par la suite de regrouper en fonction des différentes règles les états qui sont équivalents.

Lorsque les états équivalents ont été réunis, ils sont renommés en un nouvel état unique. Chaque nouvel état est donc relié à un autre par une transition qui n'est autre qu'un caractère du motif à rechercher.

Suite à cela, un automate fini déterministe avec un minimum d'états peut être utilisé afin de rechercher les correspondances dans le texte.

Ces diverses créations d'automate, permettent d'assurer la récupération des résultats tout comme la recherche Egrep.

La complexité de l'algorithme se calcule d'abord avec une recherche du prochain état accessible par une ε-transitions d'un état, c'est-à-dire que cette recherche à une complexité de O(m), puis il faut refaire cette recherche pour chaque état présent donc la complexité devient O(mn). Cependant dans l'automate, il y a au plus 2 arcs sur n'importe quel état, donc m est inférieur ou égal à 2n, donc la complexité est donc de $O(n^2)$ minimum.

b. KMP

Efficace pour les chaines de caractères mais inutilisable pour la recherche de motif spéciaux comme le ou et l'étoile dans un motif, KMP est plus rapide pour la recherche de chaine de caractères.

Son implémentation est beaucoup plus rapide que l'implémentation des automates.

KMP permet de rechercher une occurrence d'une chaine P, dans un texte. Cet algorithme ne réexamine pas les caractères qui ont été vu auparavant, il limite donc le nombre de comparaisons.

Cet algorithme a une complexité de O(|P|+|Taille du Texte|), ce qui réduit considérablement son temps d'exécution.

5. Tests

a. L'obtention des fichiers de tests

Les différents fichiers de tests ont été récupérés sur http://www.gutenberg.org/.
Deux textes ont été sélectionnés, un sur l'histoire de Babylone et un second sur Sherlock Holmes.

b. Tests

Le temps d'exécution de la commande est récupéré par la commande suivante : « time egrep "Sherlock" sherlock.txt ».

Afin de comparer la performance des différents algorithmes avec la commande Egrep, plusieurs motifs ont été recherchés dans différents texte.

Dans le texte sur l'histoire de Babylone, nous avons cherché les motifs suivants : Sargon, S(a|r|g)*on et S(a|r|g)*on|A(s|y)*rian.

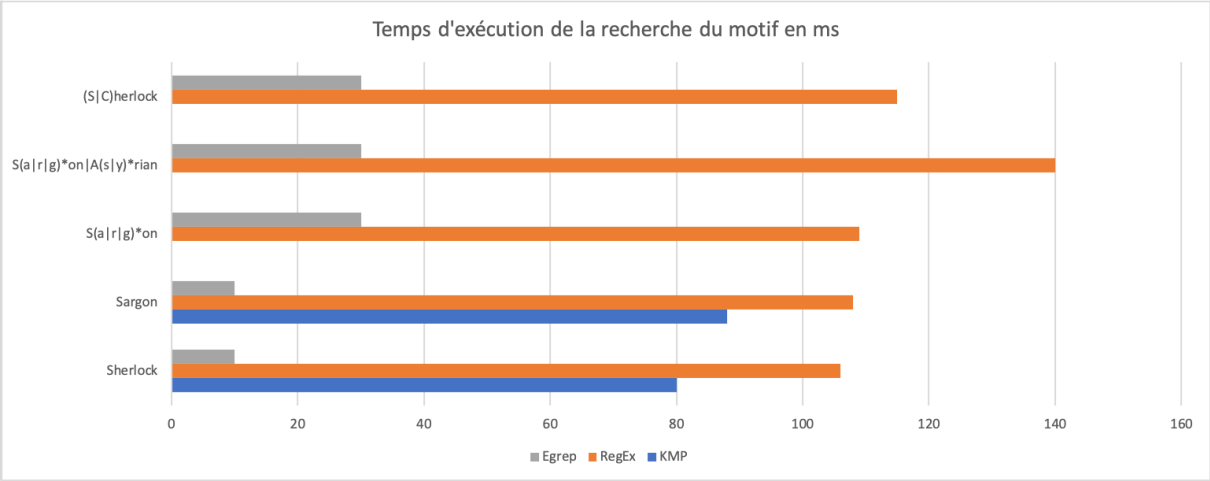
Dans le texte de Sherlock Holmes, nous avons cherché ceux-ci : Sherlock et (S|C)herlock.

Tous les résultats de la recherche quelques soit la méthode utilisée, sont présents dans le dossier résultats du programme.

Pour chaque motif recherché, les résultats obtenus sont identiques aux résultats obtenus grâce à la commande egrep.

c. Performance des algorithmes face à Egrep

KMP étant spécifique à la recherche d'un facteur, il n'a donc pas de résultat pour (S|C)herlock, S(a|r|g)*on et S(a|r|g)*on|A(s|y)*rian.



Pour chaque test et chaque méthode, les résultats et le temps d'exécution ont été récupérés.

Nous pouvons voir que pour la recherche d'un facteur dans un texte, KMP est beaucoup plus performant que la méthode d'Aho-Ullman, il y a presque 20ms de différences en temps d'exécution ce qui est conséquent.
Si nous comparons, ce temps d'exécution avec la commande egrep, celle-ci est beaucoup plus performante que KMP ou les automates, elle a presque 100 ms de différences avec les automates, ce qui est non négligeable pour la recherche dans un texte de plusieurs milliers de lignes.

Pour la recherche de motifs plus complexe, KMP n'étant pas créé pour ce genre de recherche, il n'est pas utilisé.
Cependant si on compare chaque test, la méthode d'Aho-Ullman met beaucoup plus de temps que la commande egrep, elles ont entre 80 et 100 ms d'écart en fonction du motif recherché.

Aux vues des résultats obtenus pour un même motif en fonction de la méthode utilisée, si nous n'avons pas accès à la fonctionnalité egrep, il est préférable de combiner les deux algorithmes en un même programme. Si le programme détecte un facteur, alors KMP sera appelé, alors que si c'est un motif plus complexe, les Automates seront appelés.

Conclusion

Suite à l'implémentation des différents algorithmes et de la comparaison des différents résultats obtenus, nous pouvons dire que la création de plusieurs automates successifs permet de créer un clone de la commande Egrep d'Unix.

Afin d'obtenir le clone de la commande Egrep, nous avons dû créer l'arbre syntaxique du motif, puis nous avons dû créer l'automate non déterministe avec les ε-transitions, et ensuite à la création de l'automate déterministe avec le minimum d'états.
Si nous avons respecté les règles et réussi l'implémentation de ces différents automates, si on utilise le dernier automate obtenu pour rechercher un motif, nous devrions tomber sur les mêmes résultats que la commande Egrep.

Pour conclure, nous pouvons donc dire que la méthode Aho-Ullman permet de cloner la commande Egrep.