Projet 1

Clone de la commande UNIX egrep

Alessandro RINAUDO
Willyan LIN

SOMMAIRE

I. Introduction

Effectuer une recherche devient un usage très courant dans notre quotidien. De nos jours il y a plusieurs outils qui nous permettent d'effectuer une recherche comme par exemple : la commande *egrep* des systèmes d'exploitation Unix, ou bien la recherche Spotlight intégrée dans tous les dispositifs les plus récentes Apple. Les deux outils permettent respectivement d'effectuer une recherche de motif dans un texte, ou de pouvoir chercher des applications ou fichiers présents dans un ordinateur.

Le but de ce projet est d'effectuer un clone de la commande *egrep* en implémentant différentes stratégies de recherche de motif à l'aide des algorithmes vu dans les séances de l'UE et à travers des solutions naïves proposées par nous-mêmes.

Le projet nous permettra d'effectuer une recherche dans l'intégralité d'un fichier de texte

- Par chaîne ou sous chaîne de caractères,
- Par mot,
- Par une expression régulière.

Lorsqu' un match avec le mot recherché a eu lieu, l'algorithme retourne uniquement la ligne du match. L'utilisateur aura donc un accès plus rapide à un document textuel par recherche de mot-clé.

II. Algorithmes utilisées

Il existe de nombreux algorithmes centrés sur la recherche par motif, nous allons en particulier étudier celui d'Aho-Ullman.

A) Aho-Ullman – RegEx

1) Présentation

Pour analyser une expression régulière, il est important de bien la modéliser en transformant l'expression en automate. Un automate est un modèle de mécanisme mathématique qui formalise les méthodes de calcul. Grâce aux automates nous pouvons comprendre assez facilement la structure de la « RegEx ».

Les deux chercheurs Aho et Ullman, réalisateurs du célèbre livre « Foundation of computer science » ont proposés un algorithme, permettant de rejoindre cet objectif.

L'algorithme se base sur plusieurs étapes dont :

- La transformation d'une RegEx en arbre syntaxique
- La transformation de l'arbre en automate avec epsilon transition
- La déterminisation de l'automate
- La minimisation de l'automate

2) Implémentation

Afin de pouvoir implémenter les différentes étapes des algorithmes pour nous permettre d'effectuer une recherche de motif dans un texte, nous avons utilisés plusieurs structures différentes.

Comme l'algorithme est divisé en quatre étapes distinctes en termes de traitement des données, nous avons décidé d'utiliser des structures différentes en les adaptant aux besoins, pour que la résolution de calcul soit la plus rapide possible.

La première étape consiste à transformer la RegEx en arbre syntaxique, nous avons utilisé une liste d'objets que nous avons appelée RegexTree contenant une racine et une sous liste du même objet (des feuilles).

```
>> Please enter a regEx: a|bc*
>> Parsing regEx "a|bc*".
>> ...
>> ASCII codes: [,97,124,98,99,42].
>> String result: a|bc*
>> Tree result: |(a,.(b,*(c)))
```

*Figure 1 : Parsing de l'expression a|bc**

Ensuite la deuxième étape a pour objectif de créer un automate NDFA avec les états de transition epsilon (ε), nous avons utilisé deux structures de données :

- Une composée par une matrice (en Java représentées par un tableau à deux dimensions) contenante respectivement les valeurs correspondantes :
  - Les lignes (états de départ),
  - Les colonnes (lettres en ASCII)
  - Les valeurs (états finaux), lorsque la transition est inexistante sa valeur est -1.
- Une autre composée par une liste (en Java ArrayList) contenant pour chaque transition les epsilons nécessaires.

```
BEGIN NDFA
Initial state: 0
Final state: 9
Transition list:
  0 -- epsilon --> 1
  0 -- epsilon --> 3
  2 -- epsilon --> 9
  4 -- epsilon --> 5
  5 -- epsilon --> 6
  5 -- epsilon --> 8
  7 -- epsilon --> 8
  7 -- epsilon --> 6
  8 -- epsilon --> 9
  1 -- a --> 2
  3 -- b --> 4
  6 -- c --> 7
END NDFA.
```

*Figure 2 : NDFA de l'expression a|bc**

Puis grâce au NDFA précédemment obtenu, nous pourrions construire la déterminisation qui consiste à éliminer les transitions epsilon.
Nous retrouvons toujours l'état initial qui est à la position 0 puis la liste des états finaux.

De ce fait nous pouvons construire la table des transitions DFA.

```
BEGIN DETERMINISATION :

line=[0, 1, 3], column=a, valeur=[2, 9]
line=[0, 1, 3], column=b, valeur=[4, 5, 6, 8, 9]
line=[4, 5, 6, 8, 9], column=c, valeur=[7, 8, 6, 9]
line=[7, 8, 6, 9], column=c, valeur=[7, 8, 6, 9]
first state : 0
final state : 2, 4, 7

END DETERMINISATION
```

*Figure 3 : DFA de l'expression a|bc**

Enfin, à l'aide de la précédente étape, nous pouvons créer un nouvel automate avec les transitions. Cette partie permettra également de commencer la minimisation que nous n'avons pas eu assez de temps pour l'implémenter.

En effet, avec la méthode de la minimisation, le temps de calcul diminue énormément puisque l'algorithme consiste à réduire le nombre d'itération des transitions.

```
AUTOMATE PROPRE :

0 --> a --> 2
0 --> b --> 4
4 --> c --> 7
7 --> c --> 7
 initial states : 0
 final states : [2, 4, 7]

END AUTOMATE PROPRE
```

*Figure 4 : DFA simplifié de l'expression a|bc**

Pour effectuer la recherche, le programme appelle cet algorithme puis il est répété autant de fois qu'il y a de ligne et de mot dans un texte.

B) Algorithme naïve - validation de la recherche

La fonction récursive et naïve crée directement par nous-mêmes, prend le nom de « regexValidator » et permet la vérification d'une RegEx ou bien d'une chaine de caractère entière dans un mot passé en paramètre. Cette fonction sera la même utilisée dans la partie finale de notre algorithme afin de pouvoir filtrer des lignes d'un texte (ou un livre) contenant les expressions régulières choisies par l'utilisateur.
La fonction se base sur l'analyse de l'automate déterministe, généré à travers les étapes précédentes.

Dans l'algorithme il y a trois phases principales :

- Premièrement, la vérification des caractères contenu dans le mot (si le mot à rechercher ne contient aucune lettre appartenant à l'expression régulière, alors l'algorithme s'arrête et nous donne la réponse)
- Puis, nous vérifions si le caractère analysé appartient à la fois à l'état initial et à l'état final de l'automate
- Pour finir, nous faisons appel à une récursion dans le cas contraire, qui commence l'analyse en comparant chaque caractère de l'état initial avec l'état final du précédent, et ainsi de suite pour le reste des caractères en lisant l'automate généré précédemment.

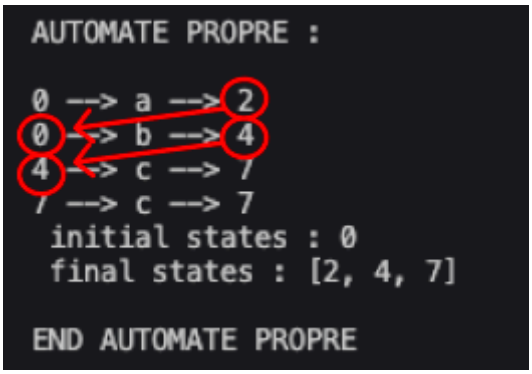

Figure 5 : Exemple d'exécution de l'algorithme regexValidator

III. Test unitaire

Les tests unitaires sont des moyens permettant de s'assurer du bon fonctionnement des fonctionnalités, indépendante des unes des autres au sein d'une application. De ce fait l'application sera plus durable et surtout plus résistante aux évolutions.

Pour cela, nous avons implémenté des tests pour chaque partie de l'algorithme : le parsing, la NDFA et enfin le DFA.

Nous avons choisi d'utiliser un célèbre framework Java en open source **JUnit** pour réaliser nos tests unitaires. Des séries de jeu de test sont lancées avec la concaténation, l'altern et l'étoile.

Grâce à la librairie JUnit et à Visual Studio Code, nous obtenons un affichage très indicatif sur la liste des tests, leur état et leur temps d'exécution Illustré dans la figure ci-dessous.

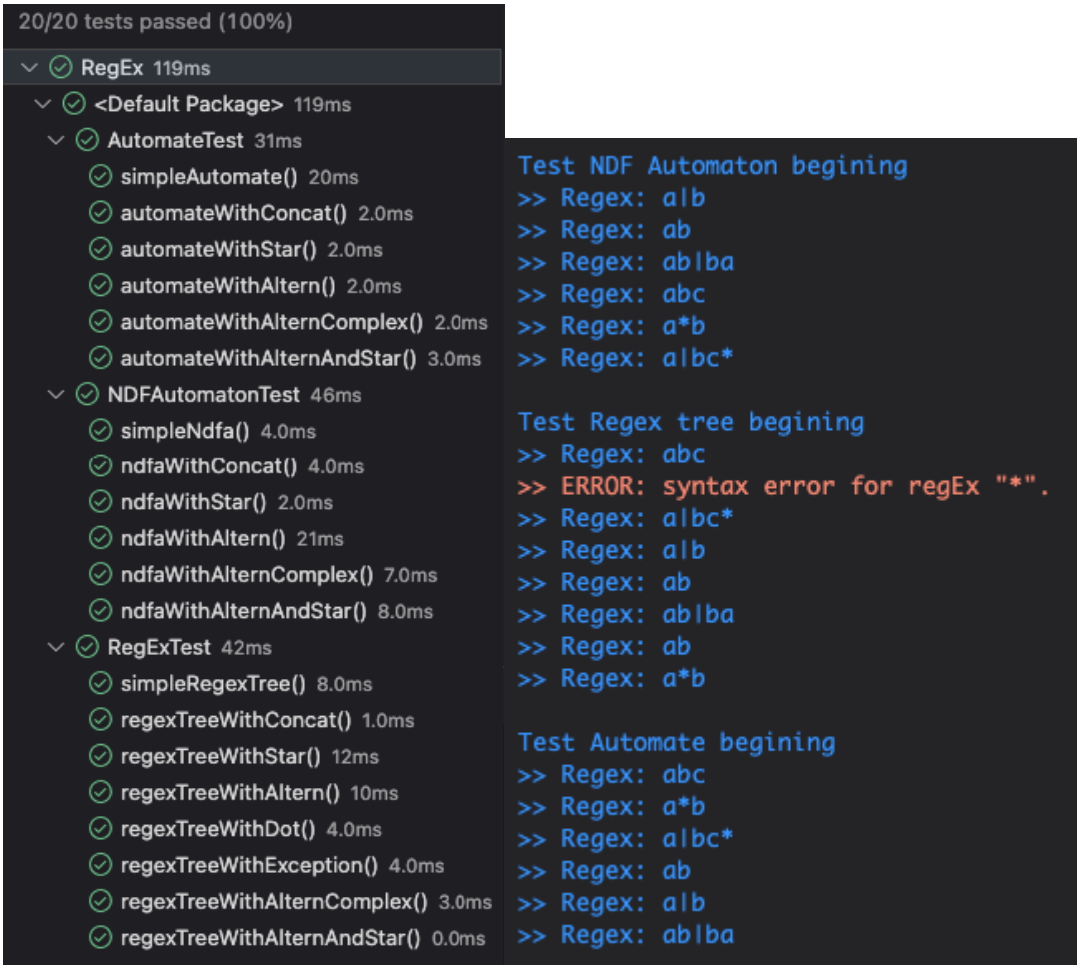

Figure 6 : Liste et résultat des tests unitaires

La ligne en rouge avec « error » est faite exprès, nous avons pour but d'englober le plus de cas possible notamment ceux des erreurs et des exceptions.
Dès que nous rencontrons un exemple de test intéressant, nous l'ajoutons aussi tôt dans nos jeux de test.

IV. Test de performance

A) Comparaison entre les symboles

À l'aide des résultats obtenu précédemment, nous pouvons tirer un graphe du temps d'exécution par opérateur de notre clone pour les expressions suivantes : a.b a*b a|b

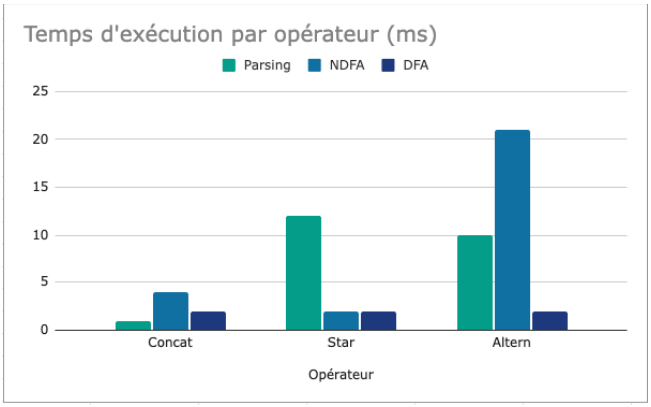

Figure 7 : Graphe de comparaison du temps d'exécution par opérateur

Nous constatons que l'expression avec une concaténation est plus rapide à exécuter par rapport à celle de l'altern qui nécessite un peu plus de traitement. Les résultats sont à titre indicatif, pour obtenir un résultat proche du réel il sera nécessaire d'avoir un jeu de données massives.

B) Comparaison entre egrep et le clone

Afin d'avoir une approche plus significative, nous avons comparé le temps d'exécution de la commande egrep avec notre clone, nous avons testés ces différentes expressions : J*Q J|Q Sargon S(a|g|r) S(a|g|r)*on

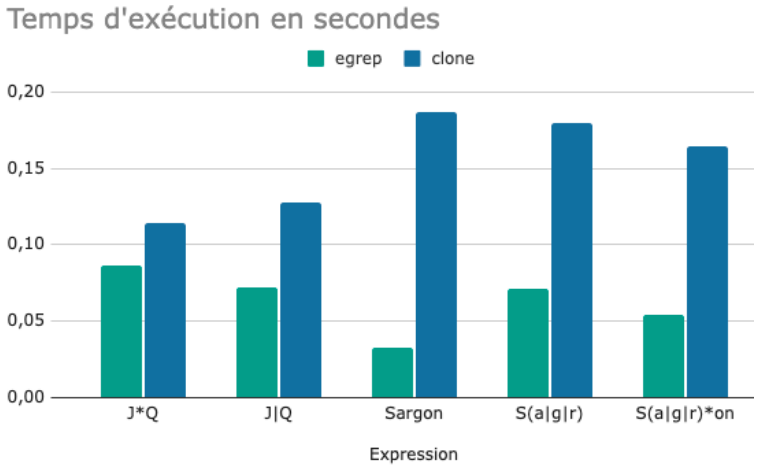

Figure 8 : Graphe de comparaison du temps d'exécution par expression

Nous constatons que notre clone a un temps d'exécution stable, pour n'importe quelle expression le temps est d'environ de 0.15 secondes.
Pour certains cas, le temps d'exécution se rapproche d'egrep mais egrep reste tout de même plus rapide.

V. Résultat

La figure ci-dessous représente le fruit de notre commande, avec le nombre total de match, le temps d'exécution et ainsi que les lignes avec le mot matché en couleur.

*Figure 9 : Résultat du clone pour le mot "S(a|g|r)*on"*

VI. Conclusion

Par suite de cette expérimentation nous avons non seulement compris la partie théorique des algorithmes mais en plus nous avons réussi à les implémenter en code afin de l'utiliser pour des fin pratiques comme la recherche d'un motif dans un texte.

Cependant notre code n'inclue pas la totalité des fonctionnalité d'egrep, avec un peu plus de temps nous aurions aimé d'implémenter la minimisation et ainsi que l'expression "+" qui était optionnel.

En conclusion ce projet nous a permis d'apprendre le fonctionnement de base des moteurs de recherches ainsi que d'apprendre des nouvelles compétences en algorithmique ce qui nous pourra servir un jour pour le développement d'application réticulaires toujours plus complexes.

VII. Annexes

Voici quelques réalisations que nous avons effectué au tableau pour mieux comprendre la partie théorique du livre Aho-Ullman.