
# Projet 1 - Clone de egrep avec support partiel des ERE

Claire Bardoux - Cyann Donnot

Master 2 - Spécialité STL

https://github.com/zahrof/EGREP et
https://github.com/zahrof/Search_feature_by_RegEx

2020 - 2021

# Table des Matières



# 1 Automate d'une expression régulière

La base de la fonction `egrep` consiste à tout d'abord créer l'automate déterministe minimal de l'expression régulière donnée par l'utilisateur. Nous allons donc voir dans cette section comment y arriver.

## 1.1 De l'expression régulière à un arbre syntaxique

Dans un premier temps nous devons transformer l'expression régulière demandée en un arbre syntaxique. Au sein de notre projet nous avons utilisé le code suggère dans le TME 1.

## 1.2 De l'arbre syntaxique à l'automate avec $\epsilon$ - transitions

Maintenant que nous avons un arbre de syntaxe représentant l'expression régulière de l'utilisateur, nous devons en déduire un automate avec des $\epsilon$-transitions qui accepte ce dernier. Pour cela, nous nous basons sur le chapitre 10 les pages 571-582 du livre Allo-Hullman.[1]

**Structure de Données** Dans un premier temps, il nous paraissait intéressant de transformer notre arbre syntaxique en une matrice d'adjacence (du type `ArrayList<State>[][]`) de l'automate $\epsilon$-transition correspondant. Cependant, nous nous sommes assez vite rendu compte qu'il était plus utile d'avoir un automate pour réaliser l'algorithme de détermination. Nous avons donc décidé de partir sur une structure récursive (Cf. Figure 2) que nous construisons récursivement en parcourant l'arbre syntaxique. Chaque noeud, possède alors une Table de Hachage *sons* qui lui permet pour un code ASCCI donné de pointer sur d'autres états de l'automate. Un booléen *terminal* permet de reconnaître l'état final de l'automate. On notera aussi que chaque noeud possède un identifiant *id* unique utile pour l'algorithme tout au long du processus de minimisation.

## 1.3 Suppression des $\epsilon$ - transitions

Pour transformer notre automate en un `DFA` nous avions utilisé l'algorithme de construction de sous-ensemble décrit sur cette page [3]. Tout d'abord nous allons fusionner les états de notre `NFA` dans les cas suivants:

- Si nous passons d'un état à un autre à partir d'une $\epsilon$ transition. Nous pouvons les fusionner car pour passer d'un état à l'autre nous n'utilisons aucun caractère.
- Si nous avons plusieurs transitions avec le même symbole $\alpha$, alors on peut considérer que nous avons un état qui à partir d'une transition avec le symbole $\alpha$ il va aller vers un ensemble d'états. Cet ensemble d'états sera considéré comme un seul état pour la création du `DFA`.

Dans notre algorithme nous devons définir deux fonctions complémentaires:

- La fonction $\epsilon$-CLOSURE prend un état et retourne l'ensemble des états accessibles à partir de celui-ci avec une ou plusieurs $\epsilon$-transitions. Nous pouvons remarquer que cela inclura toujours l'état passé en paramètre.
- La fonction MOVE prend en entrée un état et un caractère et renvoie l'ensemble des états accessibles par *une* transition avec ce symbole.

**Algorithm 1** L'algorithme de construction de sous-ensemble

Créez l'état initial du nouveau DFA en le résultat de la fonction l'$\epsilon$- closure ayant comme argument l'état initial du NFA
Procédez comme suit pour le nouvel état DFA:
    Pour chaque symbole d'entrée possible:
        1. Appliquer la fonction MOVE avec l'état nouvellement créé et le symbole qu'on est en train de traiter, cela renverra un ensemble d'état.
        2. Appliquez la fonction $\epsilon$-closure en prenant en entrée cet ensemble d'état, cela pourra entraîner un nouvel ensemble.
Cet ensemble d'états NFA sera un unique état dans DFA.
Chaque fois que nous générons un novuel état DFA, nous devons lui appliquer l'étape 2. Le processus est terminé lorsque l'application de l'étape 2 ne produit aucun nouvel état.
Les états terminaux du DFA sont ceux qui contiennent l'un des états de fin du NFA.

Cependant cet algorithme avait une très mauvaise complexité temporelle à cause de l'utilisation de `Set<>` dont nous n'arrivions pas à nous passer. Nous avons donc opté pour un autre algorithme, directement sur les Automates $\epsilon$-transition, beaucoup plus efficace.

**Algorithm 2** Algorithme de détermination d'automate

**function** DETERMINISATION(()A : Automate $\epsilon$-transition)
    T $\leftarrow$ tableau ou chaque case i contient le état d'id i
    **for all** Automate a de T **do**
        ABSORB(a)
    **for all** Automate a de T **do**
        Recréer chaque liaison aux fils en pointant la bonne case de T
    Renommer Tout les noeuds atteignable depuis T[0]
    **retourner** T[0]
**function** ABSORB(()A : Automate $\epsilon$-transition)
    **if** A ne possède pas d'$\epsilon$-transition **then**
        **retourner** A
    L $\leftarrow$ fils d'$\epsilon$-transition de A supprimer les $\epsilon$-transition de A
    **for all** Automate a de L **do**
        fusionner les tables de fils de a avec celle de A
    **retourner** ABSORB(A)

Son fonctionnement est très simple, pour chaque noeud de l'arbre, on va absorber ses fils d'$\epsilon$-transition jusqu'à ce qu'il n'y en ai plus. Comme cela a pour effet de déconstruire l'arbre, on recrée les liens puis on renomme chaque noeud avant de simplement retourner le noeud initial.

Remarque : Cet algorithme n'est pas valide, dans le sens où dans certains cas il ne marche pas car il n'implémente pas un équivalent de la fonction MOVE, ce qui l'empêche de générer correctement de l'automate déterministe.

### 1.4 Minimisation de l'automate

Afin de minimiser notre automate nous avons crée l'objet `MinimalizedAutomaton` 2.

Pour minimiser notre automate avons utilisé l'algorithme de minimisation décrit sur cette page [2].

A différence de `EAutomaton` nous allons garder en mémoire le père de l'état, cela nous sera utile lors de la fusion de deux états équivalents.

Le principe de cet algorithme est de prendre les états deux à deux et de leur appliquer la fonction `merge` dans la classe `MinimalizedAutomaton` qui renvoie la fusion des deux automates ou `null` si la fusion ne peut avoir lieu car les deux états ne sont pas équivalents.

**Algorithm 3** L'algorithme de minimisation

```
1: function MINIMIZE(dfa : Automate))
2:     sizeAutomata ← La taille de l'automate
3:     i ← 0
4:     while i < sizeAutomata do
5:         j ← 1
6:         ms = null
7:         while j <sizeAutomata do
8:             ms = merge de la valeur de la i^eme et j^eme clé de dfa.
9:             if (ms != null) & & (i!=j) then On ajoute ms à l'automate
10:                if i < j then
11:                    Supprimer j de dfa
12:                    Supprimer i de dfa
13:                else
14:                    Supprimer i de dfa
15:                    Supprimer j de dfa
16:                sizeAutomata = sizeAutomata - 1
17:                i = 0
18:                j = 0
19:            j = j + 1
20:        i = i + 1
21:    min = null
22:    for all Automate ms dans dfa do
23:        if l'identifiant de ms == 0 then
24:            min = ms
25:            break
26:    retourner min
```

## 2 Clone de egrep

Maintenant que nous avons réussi à créer l'automate déterministe minimal correspondant à l'expression régulière donnée par l'utilisateur, nous devons l'appliquer pour chercher les motifs qu'il accepte dans le livre.

## 2.1 Application d'un Automate dans la recherche de motifs

Nous allons d'abord commencer par créer l'algorithme d'application et comme son nom l'indique lance une fois la reconnaissance de l'automate dans le livre à partir d'un caractère donné. (Cf Algorithme 4)

**Algorithm 4** Recherches de motifs acceptés par un automate

```
1: function Application(A : Automate, B : Livre, p : position)
2:     c ← le caractère à la position p dans B
3:     if exist un chemin vers un fils a avec c depuis A then
4:         retourner Application(a, B, p + 1)
5:     if A est un état final then
6:         retourner p
7:     retourner null
```

Il trois cas dans l'algorithme d'application.

1. L'automate $A$ accepte le caractère à la position $p$ comme étant un chemin valide vers un autre état. Dans ce cas, on peut continuer l'application et retourner le résultat d'un appel récursif sur ce nouvel état sur la lettre suivante.

2. L'automate n'admet pas de chemin mais est état final. Dans ce cas, nous ne pouvons plus agrandir le mot mais ce dernier est déjà accepté par l'automate. nous pouvons donc le renvoyer. (à travers la position de son dernier caractère)

3. L'automate n'admet pas de chemin et n'est pas un état final. On en conclut que le mot n'est pas accepté, aucun mot n'est renvoyé. (à travers un pointeur nul)

Il ne reste donc plus qu'à l'exécuter sur l'intégralité du livre, c'est a dire sur chaque caractère tel que le fait l'algorithme egrep (Cf. Algorithm 5)

**Algorithm 5** Application de la recherche de motif sur un livre

```
1: function egrep(A : Automate, B : Livre)
2:     P ← Une liste vide de mots
3:     p ← la position du premier caractère de B
4:     while p est dans le livre do
5:         if exist Application(A, B, p) as p' then
6:             P apprend le mot allant de p à p'
7:         p ← p + 1
8:     retourner P
```

## 2.2 Comparaison entre egrep et son clone

Du fait que nous n'ayons pas de système de mesures sur les expressions régulières. Nous ne pouvons pas comparer autre chose que les temps d'exécution entre eux. Nous avons donc utilisé python pour réaliser des tests de temps sur les deux `egrep`. On peut donc juste voir sur la Figure 3 de l'annexe est bien moins efficace que celui présent de base sur Linux mais qu'il reste tout de même exploitable avec un temps d'exécution correcte

# References


[1] Jeff Ullman Al Aho. "Foundations of Computer Science". In: (1992), pp. 571–582. URL: http://infolab.stanford.edu/~ullman/focs/ch10.pdf.

[2] *CDFA minimisation*. http://www.cs.nuim.ie/~jpower/Courses/Previous/parsing/node10.html.

[3] *Constructing a DFA from an NFA*. http://www.cs.nuim.ie/~jpower/Courses/Previous/parsing/node9.html.


# 3 Annexe

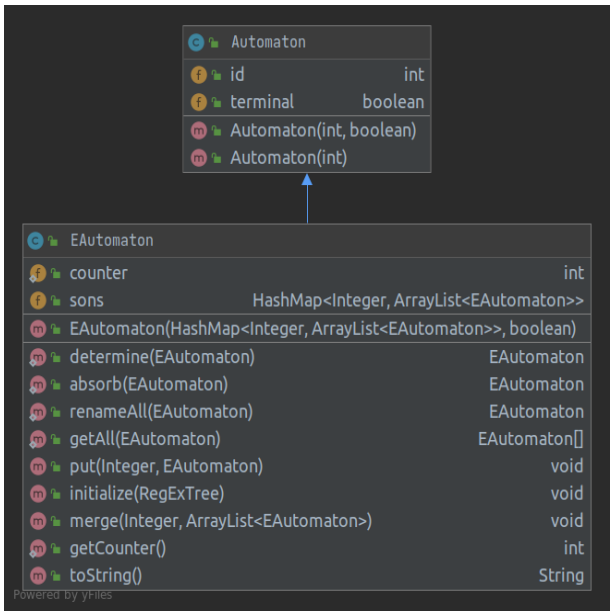


Figure 1: UML : Automate $\epsilon$-transition

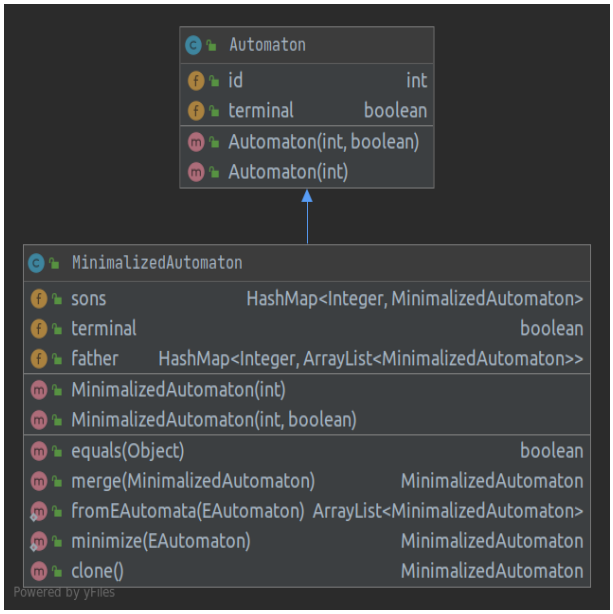


Figure 2: UML : MinimalizedAutomaton

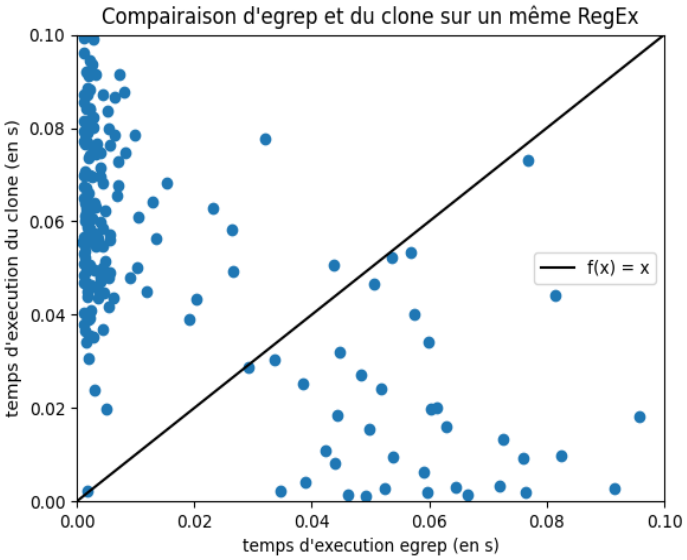


Figure 3: Comparaison d'egrep et du clone)