# Clone de egrep avec support partiel des ERE.

Pablito Bello, Pierre Gomez

Novembre 2020

**Résumé :** Étant donné un fichier textuel comprenant $l$ lignes et un motif $m$ représenté par une expression régulière, on propose une implantation de la recherche du motif $m$ sur chacune des $l$ lignes du fichier, afin d'afficher uniquement les lignes contenant le motif $m$. On propose ici la construction d'un automate fini déterministe minimal en plusieurs étapes. La solution se comportera comme un clone de la commande egrep avec un ensemble de motifs réduits ; seuls les parenthèses, l'alternative, la concaténation, l'opération étoile, le point (caractère universel, mais seules les lettres sont acceptées) et la lettre ASCII seront autorisés dans un motif.

**Mots-clés :** Egrep, Expressions régulières, Automates, Automate fini non déterministe avec epsilon transitions, Automate fini déterministe, Automate fini déterministe minimal.

# Table des matières

## Première partie

# Introduction

Afin de réaliser un clone de la commande `egrep` nous allons nous intéresser à la construction d'un automate fini déterministe de taille minimale (`Min-DFA`), reconnaissant un motif donné.

La reconnaissance d'expressions régulières est utilisé dans de nombreux domaines de l'informatique moderne, notamment pour les moteurs de recherche ou encore les éditeurs de texte (remplacement de mots, etc).

Notre objectif sera de discuter des différents algorithmes permettant la création d'un automate fini déterministe de taille minimale tout en présentant notre implantation.

Nous réaliserons également des tests de performance en faisant un comparatif avec la commande `egrep`.

## Deuxième partie

# Notions utiles

**Expressions régulières :** Les expressions régulières sont des chaînes de caractères qui permettent de désigner un ensemble de chaînes de caractères. Elles utilisent pour cela des caractères spéciaux, nous détaillons ici uniquement ceux utilisés dans le cadre de ce projet :

1. le "." correspond au caractère universel, il peut être remplacé par n'importe quel caractère de l'alphabet (attention : un espace par exemple ne sera PAS accepté.).
2. l'"*" (e.g. `X*`) correspond à la répétition, le motif `X` peut alors apparaître un nombre quelconque de fois ou ne pas apparaître.
3. le "+" (e.g. `X+`) correspond à la répétition mais avec pour contrainte que le motif `X` doit apparaître au moins une fois.
4. le "|" (e.g. `X|Y`) correspond à l'opérateur "ou exclusif", le motif peut être soit `X`, soit `Y`.
5. les parenthèses permettent d'appliquer un caractère spécial sur la chaîne de caractères contenue entre parenthèses.

**Automate fini :** Un automate fini est une machine abstraite permettant de reconnaître un langage donné ; dans notre cas une expression régulière. Il est constitué d'états et de transition. Lorsqu'on veut tester un mot, l'automate passe d'états en états en suivant les transitions à la lecture de chaque caractère du mot donné. Si le mot peut être entièrement lu et qu'une fois tous les caractères lu l'automate se trouvent dans un état dit "final", le mot est accepté, sinon, il est refusé.

## Troisième partie

# Création d'un automate fini non déterministe avec epsilon transitions

**Automate fini non déterministre** Un automate fini non déterministe (`nfa`) est un automate fini tel que dans un état donné, il peut y avoir de multiples transitions pour un même caractère.

**Automate fini non déterministe avec epsilon transitions** Un automate fini non déterministe avec epsilon transitions ($\epsilon$-`nfa`) est un `nfa` dont les transitions peuvent être étiquetées par le mot vide (epsilon), si une transition entre deux états est étiquetée par epsilon, on peut alors passer d'un état à l'autre sans consommer de caractère.

Construire un `nfa` ou un $\epsilon$-`nfa` est le moyen le plus simple pour créer un automate à partir d'une expression régulière, plusieurs algorithmes permettent ceci :

1. L'algorithme de `Thompson` (utilisé pour la commande `grep` permet de construire un $\epsilon$-`nfa`. Soit $n$ le nombre de symboles de l'expression régulière, l'automate aura alors au plus $2n$ états. La complexité en temps de construction de l'automate est de $\theta(n)$.
2. L'algorithme de `Glushkov` permet de construire un `nfa`. Soit $n$ le nombre de symboles de l'expression régulière, la complexité en temps de construction de l'automate est de $O(n)$, mais l'automate ne contient pas d'epsilon transitions.

Pour faciliter l'écriture de l'algorithme, nous allons ici suivre une méthode plus simple, décrite au chapitre 10 du livre `Aho-Ullman`.

Une expression régulière est représentée en mémoire comme un arbre syntaxique, les noeuds correspondent aux caractères spéciaux et les feuilles aux caractères alphabétiques.

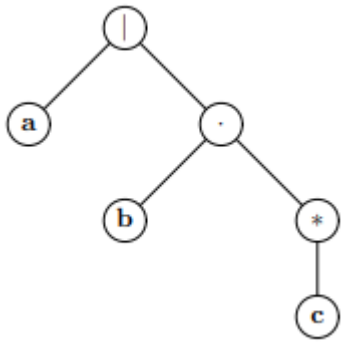


FIGURE 1 – Arbre syntaxique pour l'expression $a|bc*$

L'automate est représenté en mémoire par sa table de transition (une matrice). Les lignes correspondent aux états de l'automate, les colonnes correspondant aux caractères `ASCII` des symboles du langage. Elles contiennent une liste d'état qu'on peut atteindre en lisant le symbole donné. Trois autres colonnes sont ajoutés : une pour les epsilon transitions, une pour marquer l'état comme final, et une pour marquer l'état comme initiale.

L'algorithme de construction de l'$\epsilon$-`nfa` est le suivant :

1. On parcourt l'arbre syntaxique en partant de la racine.
2. Si le noeud courant est un caractère alphabétique, on renvoie l'automate suivant :


FIGURE 2 – Automate pour le caractère $x$

3. Si le noeud courant est le symbole spécial universel ".", on renvoie le même automate qu'au point précédent, mais x est remplacé par tous les caractères alphabétiques. (Le premier état admet donc une transition vers le deuxième et cela pour n'importe quel caractère alphabétique).
4. Si le noeud courant est le symbole spécial $*$ ou $+$, on appelle récursivement l'algorithme sur son premier fils, on obtient ainsi l'automate $R1$. On renvoie ensuite un des automates suivants selon le symbole spécial sur lequel on se trouve :

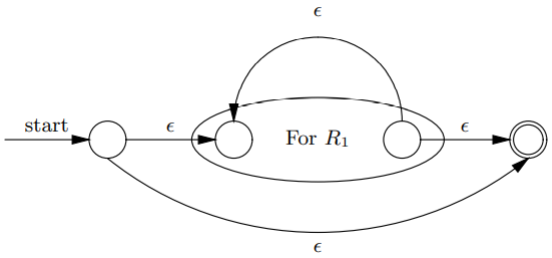


FIGURE 3 – Automate pour le symbole $*$

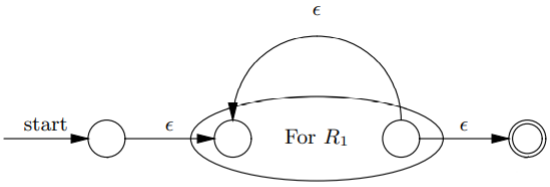


FIGURE 4 – Automate pour le symbole $+$

5. Si le noeud courant, le symbole spécial de la concaténation ou le symbole $|$, on appelle récursivement l'algorithme sur les deux fils du noeud courant, on obtient ainsi les automates $R1$ et $R2$. On renvoie ensuite un des automates suivants selon le symbole spécial sur lequel on se trouve :

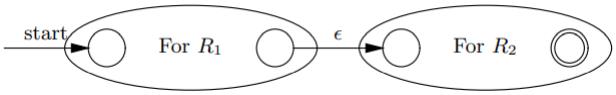


FIGURE 5 – Automate pour le symbole de concaténation

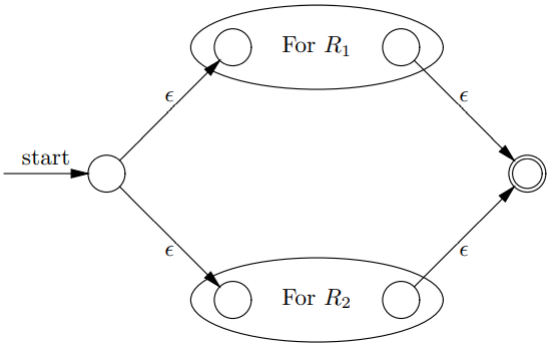


FIGURE 6 – Automate pour le symbole |

On obtient récursivement $\epsilon$-`nfa` correspondant à l'expression régulière donnée. L'automate est construit en un temps linéaire à la taille de l'arbre et donc de l'expression régulière.

On pourrait savoir si un motif est accepté ou non par cet automate en le parcourant et donc répondre à notre problème de recherche du motif. Néanmoins, la recherche serait moins efficace car l'automate est non déterministe, ainsi dans le pire des cas la recherche d'un motif de longueur $n$ sur un `nfa` comportant $m$ états se fait en $O(2^n)$.

Afin d'améliorer le temps de recherche, on va donc rendre notre automate déterministe.

## Quatrième partie

# Création d'un automate fini déterministe

Un automate fini déterministe (`dfa`) est un automate contenant un seul état initial, chaque état de l'automate doit aussi admettre une unique transition pour chaque symbole du langage.

Tout $\epsilon$-`nfa` peut-être convertis en `dfa`. Nous allons présenter l'algorithme utilisé pour la construction d'un tel automate à partir d'un $\epsilon$-`nfa` obtenu dans l'étape précédente.

On définit l'$\epsilon$-`closure` d'un état $X$ comme l'ensemble des états atteignables en lisant seulement $\epsilon$ depuis $X$, autrement dit, c'est l'ensemble des états atteignables en ne suivant que des epsilon transitions. $X$ fait toujours partie de son $\epsilon$-`closure`.

L'ensemble des états atteignables depuis un état $X$ pour un symbole $c$ se construit à partir de l'$\epsilon$-`closure` de l'état $X$, on regarde pour tous les états $Y$ de l'$\epsilon$-`closure` les états $Z$ qu'on peut atteindre en lisant $c$. L'ensemble des états atteignables est l'ensemble des états $Z$, autrement dit l'ensemble des états qu'on peut atteindre depuis $X$ en lisant autant de fois $\epsilon$ qu'on veut et en lisant une fois $c$.

Voici les étapes de l'algorithme de déterminisation :

1. On calcul l'$\epsilon$-`closure` de l'état initial de l'$\epsilon$-`nfa`. On crée le premier état du `dfa` qui est identifié par l'ensemble obtenu, cet état est l'état initial du `dfa`.
2. On marque l'état initial du `dfa` comme état courant.
3. On cherche l'ensemble des états atteignables à partir de l'état courant pour chaque symbole du langage.
4. Si cet ensemble n'identifie aucun état déjà présent dans le dfa, ont créé un nouvel état du `dfa`, identifié par cet ensemble et on marque ce nouvel état comme étant l'état courant.
5. On répète les étapes 3 et 4 tant qu'on trouve des nouveaux états du `dfa`.
6. On déclare final tout état du `dfa` dont l'identificateur contient un état de l' $\epsilon$-`nfa` qui est final dans l'$\epsilon$-`nfa`.
7. On renomme les états du `dfa`, ils ne sont plus identifiés par un ensemble d'états de l'$\epsilon$-`nfa`, mais par un entier correspondant à leur index dans le `dfa`.

## Cinquième partie

# Création d'un automate fini déterministe de taille minimale

On a obtenu dans la partie précédente un `dfa` reconnaissant le même langage que notre $\epsilon$-`nfa`. La recherche d'un motif sur le `dfa` se fait en temps linéaire par rapport au nombre de symboles du motif.

En revanche, si l'$\epsilon$-`nfa` contenait $m$ états, alors le `dfa` construit peut en contenir $2m$ dans le pire des cas. Sur de gros automates, cette augmentation du nombre d'états peut être problématique de par l'espace mémoire utilisé.

Nous allons donc construire l'unique `dfa` de taille minimale (`min-dfa`) existant pour le `dfa` obtenu dans la partie précédente.

Plusieurs algorithmes permettent d'obtenir un `min-dfa` :

1. L'algorithme de `Moore` utilise un système de partition d'états, il permet d'obtenir un `min-dfa` en $O(sm^2)$ dans le pire cas et $O(m \log(m))$ en moyenne, avec $s$ la taille de l'alphabet et $m$ le nombre d'états du `dfa`.
2. L'algorithme de `Brzozowski` fonctionne de la façon suivante : pour un $dfa$ $A$, on calcule son automate transposé, qu'on nomme $B$. On déterminise l'automate $B$, on appelle l'automate résultant $C$. On calcul l'automate $D$ qui est l'automate transposé de $C$. Enfin, on obtient le `min-dfa` en déterminisant l'automate $D$.

   La complexité moyenne de cet algorithme est donnée comme étant `génériquement super-polynomial`.

Nous allons utiliser un algorithme élémentaire pour trouver le `min-dfa` :

1. On construit une matrice de taille $m^2$ avec $m$ le nombre d'états du `dfa` qu'on appelle `matrice d'équivalences`. Dans cette matrice, si on marque la case de coordonnées $(X, Y)$, cela veut dire que les états $X$ et $Y$ ne sont pas équivalents.
2. On marque tous les $(X, Y)$ tel que X est un état initial et Y est un état non initial ou bien Y est un état initial et X est un état non initial, comme non équivalents.
3. On marque tous les $(X, Y)$ tel que X est un état final et Y est un état non final ou bien Y est un état final et X est un état non final, comme non équivalents.
4. On parcourt tous les $(X, Y)$ de la matrice qui ne sont pas encore marqués comme non équivalents. Si $X$ à une transition allant vers un état $Z$ pour un symbole $c$ et que $Y$ ne possède pas cette même transition (ou réciproquement), on marque la case comme non équivalente.

5. À ce stade, tous les couples d'états non marqués dans la matrice sont des états équivalents, on peut donc les fusionnés. Ainsi, pour deux états équivalents $X$ et $Y$, on ne garde que $X$ dans le `min-dfa` et on ajuste toutes les transitions pointant vers $Y$ comme pointant désormais vers $X$.

On obtient ainsi le `min-dfa`, qui est l'unique automate minimal correspondant à l'expression régulière qu'il reconnaît. C'est donc sur cet automate que nos recherches de motifs seront faites.

## Sixième partie

# Tests de performance

Les tests de performances sont réalisés sur un livre tiré de la base `Gutenberg`. On varie le type d'expressions cherché (symboles spéciaux contenus dans les expressions). Le texte sur lequel on réalise les recherches compte 761995 caractères, 124304 mots et 13309 lignes.

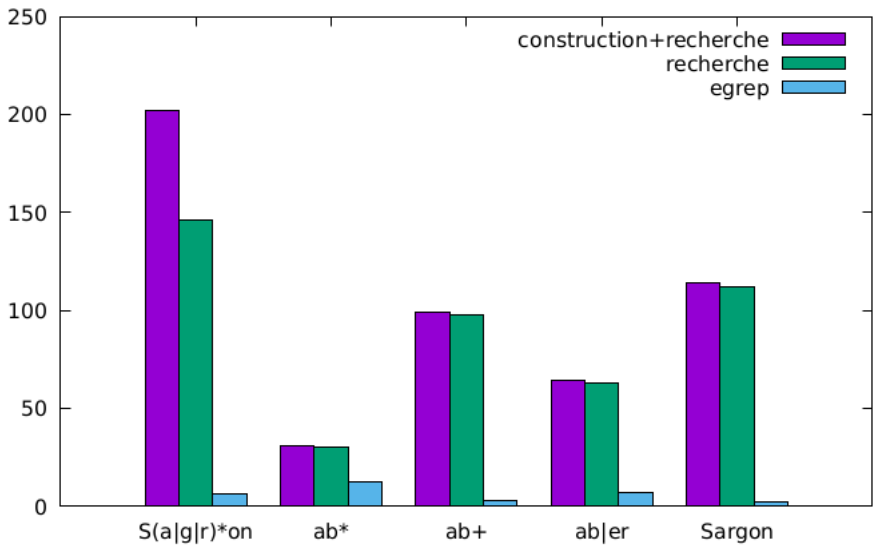


FIGURE 7 – Résultats des tests de performance.

Sur l'axe des ordonnées on retrouve le temps d'exécution totale du programme en millisecondes. Sur l'axe des abscisses on trouve les différents motifs recherchés. Pour notre programme, on affiche deux données pour chaque motif recherché : le temps total (construction du `min-afd` et recherche), mais aussi le temps de recherche uniquement (sans compté la construction du `min-afd`). En

effet, on se rend compte que la recherche occupe presque tout le temps d'exécution du programme, la construction du `min-afd` étant de l'ordre de quelques millisecondes sur nos exemples.

Cela peut s'expliquer par le fait que l'algorithme de recherche est dans le pire des cas de complexité $O(n)$ avec $n$ le nombre de caractères du fichier sur lequel on exécute la recherche.

## Septième partie
# Conclusion

À partir des travaux réalisés, on peut voir que les automates se prêtent bien à l'exercice de reconnaissance d'expressions régulières. Nous n'atteignons pas la rapidité de la commande `egrep` et de ses nombreuses optimisations, mais la recherche de motifs reste très rapide, même sur un texte de bonne taille.

L'optimisation des algorithmes de recherches trouve tout son intérêt pour les moteurs de recherche modernes, qui doivent proposer les meilleurs résultats à leur utilisateur le plus rapidement possible. Le problème est alors plus complexe qu'un simple clone de la commande `egrep` et il faut bien souvent faire des choix entre pertinences des résultats trouvés et efficacité de la recherche.