Sorbonne Université

Faculté des Sciences et Ingénierie

Département d'Informatique

Rapport sur le clonage de la commande UNIX "egrep"

Dans le cadre de l'UE : Développement des Algorithmes d'Application Réticulaire

Par :
Amine BENSLIMANE
Walid SADAT

Binh-Minh BUI-XUAN

Master 2 STL 2021/ 2022
10 octobre 2021

Table des matières

0.1 Introduction et mise en contexte

De nos jours, la quantité d'informations stockée dans le monde est en croissance exponentielle.

Selon le Digital Economy Compass [1], le volume annuel des données numériques créées était de 50 *zétaoctets* en 2020 et avoisinera les 600 *zétaoctets* d'ici 2030.

Cette expansion explosive et inévitable nécessite le développement de nouvelles technologies de fouilles de données massives pour rendre leur manipulation possible et pouvoir en extraire de nouvelles connaissances.

D'où l'importance des algorithmes de recherches par mot-clé ou par expression régulière, que l'on détaillera dans le chapitre 1.

On s'intéresse dans ce travail au clonage de la commande UNIX *egrep*. Cette dernière permet de retrouver et retourner dans un fichier donné en entrée les lignes qui *matchent* un motif précis.

Pour détailler cela, nous avons organisé ce document en trois chapitres :

— Chapitre 1 "État de l'art" : nous y exposons l'essentiel de l'état des avancées (théoriques et pratiques) de la thématique étudiée. Nous y explicitons les notations que l'on utilise ainsi que l'ensemble des définitions nécessaires pour entamer le coeur de notre travail. Bien sûr, nous ne couvrons pas l'ensemble de la théorie, ceci reviendrait à épuiser le nombre de pages qui nous ait autorisé.

— Chapitre 2 : "Algorithmes de recherche par motif" : nous proposons notre algorithme en détaillant chaque étape essentielle, à savoir la lecture (parsing) du motif, la construction de l'automate non déterministe, sa déterminisation, sa minimisation ainsi que la recherche dans le fichier textuel, nous illustrons bien évidemment chaque étape par des exemples.

— Chapitre 3 : "Mesure de performances" : Nous consacrons ce chapitre à l'étude des performances de notre algorithme comparé à la commande *egrep* où l'on s'assure d'abord de la validité des résultats et ensuite la comparaison des performances obtenus et ce par des batteries de tests quasi-exhaustives.

Chapitre 1

État de l'art

1.1 Automate à états fini

Definition 1.1.1 (Automate à états fini)**.** Un automate à états fini est un quintuplet (Σ, Q , δ, q_0, F) tel que Σ un ensemble d'alphabet, Q un ensemble fini d'états, δ un ensemble de règles de transitions tel que $\delta \subseteq (Q \times (\Sigma \cup (\epsilon)) \times Q)$, $q_0 \in Q$ l'état initial et $F \subseteq Q$ représente l'ensemble des états finaux. *Voir figure 2.6 page 7*

1.2 Langage reconnu par un automate à états fini

Definition 1.2.1 (Langage reconnu par un automate à états fini)**.** Un mot w de longueur n est dit *reconnu* par un automate si et seulement s'il existe un chemin menant de l'état q_0 à $q_m \in F$ étiqueté par une suite de lettres du mot w. L'ensemble de tous les mots acceptés par l'automate fini A forme le langage reconnu noté $\mathscr{L}(A)$, c-à-d :

$$\mathscr{L}(A) = \{w = (w_1.w_2...w_m) \in \Sigma^* \mid q_0 \xrightarrow{w1} q_1 \dot{\rightarrow} .. \xrightarrow{w_m} q_m \,^{1}\}$$

avec $q_m \in F$, $\quad m \leq n-1$ et «.» l'opérateur de concaténation.

On s'intéresse à l'étude des langages dits *réguliers*. En pratique, on formalise ces derniers par des expressions régulières (*RegEx*) que l'on suppose connues du lecteur *(voir [2])*. Par exemple, le langage reconnu par l'automate de la figure 2.6 est : $\mathscr{L}(A) = \{a \mid bc^*\}$ [2]

1.3 Automate à états fini déterministe

Definition 1.3.1 (Automate à états fini déterministe)**.** Un automate à états fini $A = (\Sigma, Q, \delta, q_0, F)$ est dit déterministe *(DFA)* si et seulement si $\delta : \Sigma \setminus \{\epsilon\} \times Q \rightarrow Q$ est une fonction de transition tel que d'un état donné, il existe au plus un seul arc sur une lettre donnée *(injectivité de δ)*, c-à-d :

$$\forall q, q1, q2 \in Q \quad \delta(q, q1) = \delta(q, q2) \implies q1 = q2$$

Voir figure 2.9 page 8

1. Une notation équivalente à : $q_0 \xrightarrow{w} q_{m-1}$
2. $c^* \equiv c^n, n \geq 0$

1.4 Équivalence des automates

Definition 1.4.1 (Équivalence des automates)**.** Soient A et B deux automates,
On dit que A et B sont équivalents si et seulement si le langage accepté par A équivaut au langage accepté par B [2] et on écrit :

$$A \sim B \iff \mathscr{L}(A) = \mathscr{L}(B)$$

1.5 Automate à états fini déterministe et minimisé

Definition 1.5.1 (Automate à états fini déterministe et minimisé)**.** Soient A et B deux automates,
On dit que B est la minimisation de A si et seulement si A et B sont équivalents et que B admet moins d'états que A et on écrit :

$$B = min_A \iff B \sim A \quad \text{et} \quad |B| < |A|$$

1.6 Algorithmes de déterminisation

Plusieurs algorithmes de déterminisation des automates sont connus de la littérature, nous ne présentons que celui que nous avons utilisé, à savoir l'algorithme de Rabin-Scott pour la construction des sous-ensembles.

1.6.1 Agorithme de Rabin-Scott (construction de sous-ensembles

---

Algorithm: Rabin-Scott subset construction

Data: NDFA
Result: DFA

1. Créer l'état initial du DFA à partir de l'ϵ-closure de l'état initial du NDFA
2. Dans le nouvel état effectuer : pour chaque symbole d'entrée possible
   (a) Appliquer le déplacement à l'état nouvellement créé et au symbole d'entrée ; cela renverra un ensemble d'états.
   (b) Appliquer la fermeture ϵ à cet ensemble d'états, résultant éventuellement en un nouvel ensemble.

   Cet ensemble d'états NDFA sera un seul état dans le DFA.
3. Chaque fois que nous générons un nouvel état DFA, nous devons lui appliquer l'étape 2. Le processus est terminé lorsque l'application de l'étape 2 ne génère aucun nouvel état.
4. Les états d'arrivée du DFA sont ceux qui contiennent l'un des états d'arrivée du NDFA.

L'algorithme admet une complexité au pire cas de $\Theta(2^n)$ [3]

1.7 Algorithme de minimisation

La littérature regorge d'algorithmes pour la minimisation des automates déterministes. A savoir,

- L'algorithme de **Hopcroft** qui, au pire cas, admet une compléxité de $O(ns \times log(n))$ tel que n le nombre d'états et s la taille de l'alphabet.[4]
- L'algorithme de **Brozowski** qui, au pire cas, admet une compléxité exponentielle (mais se comporte très rarement au pire cas) [5]

Nous choisissons l'algorithme de **Moore** que nous détaillons ci-dessous.

1.7.1 Algorithme de Moore

Algorithm: Algorithme de Moore
Data: $DFA = (\Sigma, Q, \delta, q_0, F)$
Result: $DFA_{min} = (\Sigma', Q', \delta', q'_0, F')$

1. $P \leftarrow \{F, Q \setminus F\}$
2. Tant que ($P' \neq P$) faire
   (a) $P' \leftarrow P$
   (b) Pour tout $a \in \Sigma$ faire
      i. Calculer le raffinement de p induit par a
   (c) Remplacer p par son intersection avec ces raffinements

---

A la fin de l'algorithme, l'ensemble P contiendra les nouveaux états (certains du DFA deviennent jumelés).

L'algorithme se comporte, au pire cas en $O(n^2 \times s)$ pour n nombre d'états et s la taille de l'alphabet. Quoiqu'en moyenne et pour un s fixé, l'algorithme admet une complexité de $O(n \times log(n))$ voire même $O(n \times log(log(n)))$. [6]

Chapitre 2

Algorithme de recherche par motif

L'algorithme qu'on a implémenté peut être divisé en deux *sous-algorithmes* pour différencier deux cas de *motif* à rechercher dans un fichier :

1. Le cas où le motif est une *expression régulière* : On transforme la *RegEx* en un arbre de syntaxe, puis en automate non-déterministe en utilisant la méthode décrite dans le livre **Aho-Ullman**, chapitre 10, pages 571 - 582 [2], puis en automate déterministe, puis on le minimise, et enfin, on l'utilise pour tester si un suffixe d'une ligne du texte en entrée est reconnaissable par cet automate.
2. Le cas où le motif est une *suite de concaténations* : on utilise l'algorithme **Knuth-Morris-Pratt** (KMP) [7].

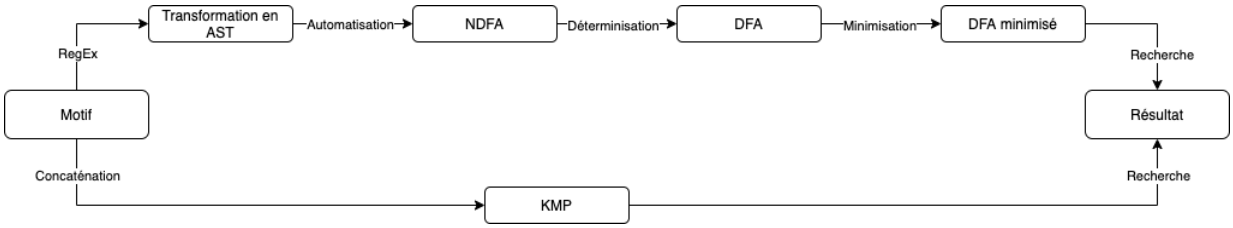


FIGURE 2.1 – Schéma suivi par l'algorithme implémenté

Nous décrivons ci-dessous les deux sous-algorithmes cités.

2.1 Motif à expression régulière

Dans le cas où le motif recherché dans le texte est une expression régulière (*i.e.* contient des *parenthèses*, un *ou* logique | ou une étoile *), on transforme la RegEx en automate qu'on utilisera pour chercher tous les mots qui matchent dans le texte en passant par cinq étapes :

1. Parsing de la *RegEx*.
2. Création de l'automate fini non-déterministe avec ϵ-transitions selon la méthode *Aho-Ullman*.
3. Passage à un automate déterministe avec la méthode des *sous-ensembles*.
4. Minimisation de l'automate pour avoir un automate équivalent avec un nombre minimum d'états.

5. Utilisation de l'automate pour tester si un suffixe d'une ligne du fichier textuel donné initialement est reconnaissable par cet automate.

Pour les deux premières étapes, on a utilisé le code fourni en cours [8].

2.1.1 Parsing et création du NDFA avec ϵ-transit

Lorsque la *RegEx* est récupérée, la première chose faite par l'algorithme est sa transformation en un arbre syntaxique

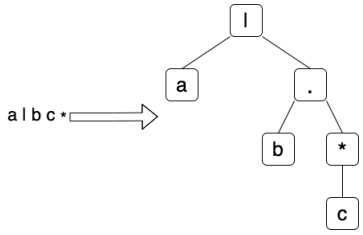


FIGURE 2.2 – Parsing de la RegEx **a|bc***

Transformation de la RegEx en NDFA

Lors du parsing de la *RegEx*, on distingue deux cas :

1. Le noeud représente un opérateur (unaire ou binaire) $\iff$ noeud interne
2. Il représente une lettre $\iff$ feuille

Lorsque l'on rencontre un noeud *feuille*, on construit l'automate de base. Ces feuilles seront transformées en noeuds *internes* en tant que fils gauche et fils droit (connectés par ϵ-transition) en fonction de l'opérateur actuellement rencontré.
On distingue deux formes de connexion : Concaténation *(figure 2.3)*, ϵ-closure *(figure 2.4)*.

FIGURE 2.3 – Connexion par concaténation

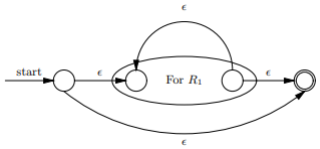


FIGURE 2.4 – Connexion par ϵ-closure

Enfin, on obtient plusieurs *RegEx* qu'il suffit de concaténer avec des ϵ-transitions pour pouvoir créer l'opération *OU "|" (voir 2.5)*

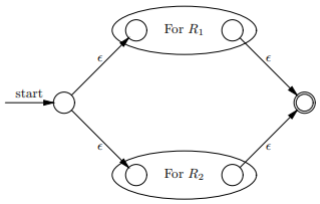


FIGURE 2.5 – Union de deux expressions régulières pour **a|bc***

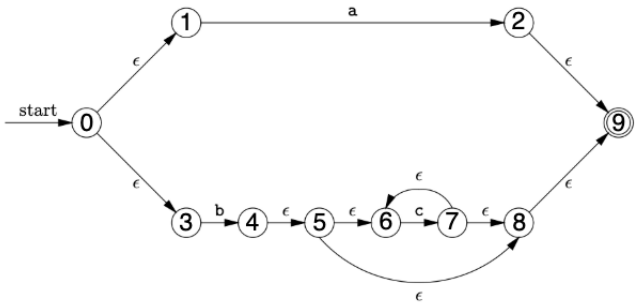


FIGURE 2.6 – Automate avec les ϵ-transitions pour **a|bc***

2.1.2 DFA avec la méthode des sous-ensembles

Le DFA est représenté par un ensemble de structures de données qui sont :

1. Une *Map* (*transitionsMap*) qui stocke les transitions avec leurs index dans la table des transitions comme clés.
2. Une matrice de listes d'entiers qui représente les transitions possibles d'une liste de transitions vers une autre par un caractère (*transitionsTable*).
3. Une liste des états finaux (*finals*).

La déterminisation de l'automate se fait en 3 étapes :

Étape 1 :

L'algorithme récupère d'abord toutes les ϵ-transitions de l'état initial et des états récupérés et crée la nouvelle transition **0** qui est l'ensemble des états récupérés.

Puis, pour chaque état, ajoute ses transitions par caractère à la transition 0 dans la table des transitions du *DFA* ainsi que leurs ϵ-transitions.

Puis chaque nouvel ensemble de transitions dans la table des transitions est ajouté à la map comme nouvel état.

Étape 2 :

Pour chaque nouvel ensemble d'états, comme fait en l'étape 1, l'algorithme récupère les ϵ-transitions de tous ses états ainsi que leurs ϵ-transitions.

Puis, pour chaque état, ajoute ses transitions par caractère à la transition en cours dans la table des transitions du *DFA* ainsi que leurs ϵ-transitions.

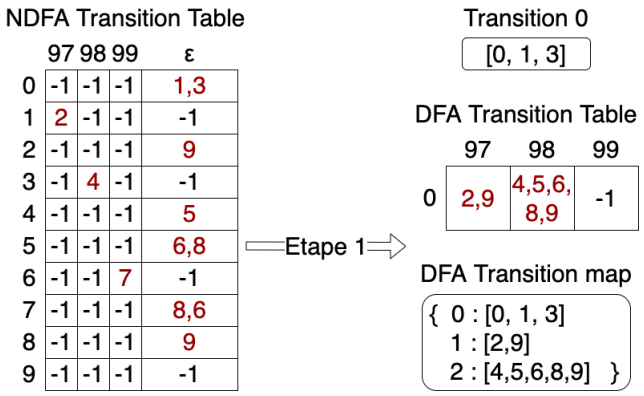


FIGURE 2.7 – Résultat de l'*étape 1* pour **a|bc***

Puis chaque nouvel ensemble de transitions dans la table des transitions est ajouté à la map comme nouvel état.

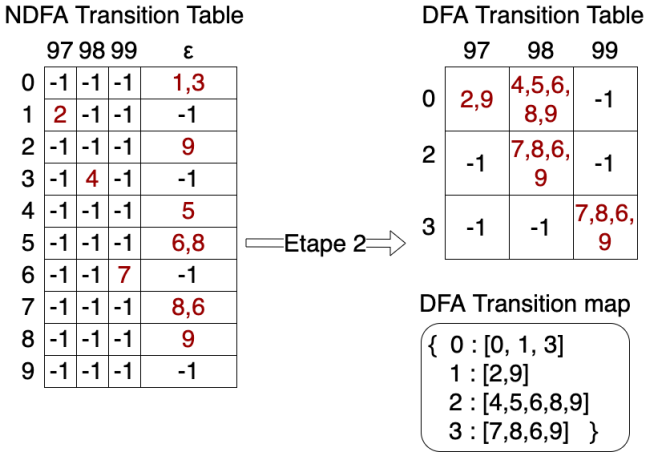


FIGURE 2.8 – Résultat de l'*étape 2* pour **a|bc***

Étape 3 :

Le but de cette étape est de définir les ensembles d'états finaux du *DFA*.

Donc, pour chaque ensemble d'états, si il contient l'état final du NDFA alors son index dans la liste *finals* est mis à 1, sinon à 0.

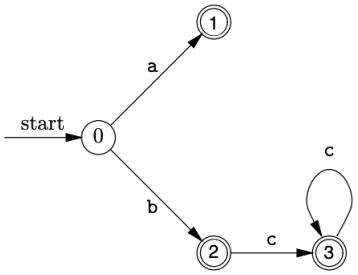


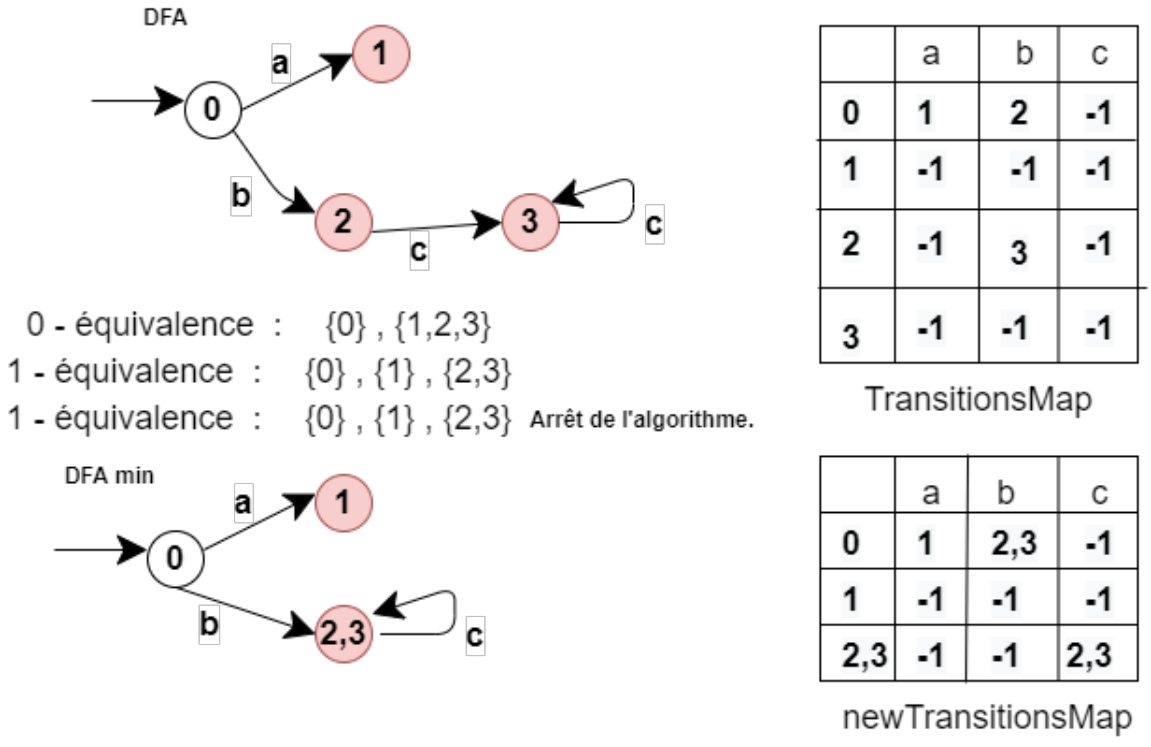
FIGURE 2.9 – Automate déterministe pour **a|bc***

2.1.3 Minimisation du DFA

On partitionne l'ensemble des états du DFA en deux sous-ensembles $P = (P_0, P_1)$ (états initiaux et finaux), dans le jargon, cette étape est appelée *0-équivalence*. Pour chaque élément des sous-ensembles, on teste l'équivalence de ces éléments entre eux, on entend par «équivalence» le fait d'avoir les mêmes éléments dans le vecteur des transitions *transitionsMap* ou bien que les transitions ne sortent jamais du sous-ensemble actuel. Sinon, l'état actuel n'est pas équivalent aux autres et il faut l'isoler (créer un nouvel sous-ensemble et l'ajouter à l'ensemble P.

On ré-itère *(1-équivalence, 2-équivalence...)* jusqu'à ne plus pouvoir avoir de nouveaux sous-ensembles. Le vecteur *(ensemble)* $P = (P_0, ..., P_n)$ contient alors les n nouveaux états (qui contiennent potentiellement des états jumelés du DFA).

Il ne reste plus qu'à *refactorer* la table des transitions *transitionsMap* en ne tenant compte que d'un seul état par sous-ensemble P_i de P. Si un sous ensemble P_i contient un état qui était initial au DFA, alors P_i est initial au DFA_{min}, de manière analogue pour les états finaux.
La figure 2.10 représente le déroulement de l'algorithme de *Moore* qui prend en entrée le DFA de la *RegEx* **a|bc*** (les états finaux sont représentés en rouge)



	a	b	c
0	1	2	-1
1	-1	-1	-1
2	-1	3	-1
3	-1	-1	-1

	a	b	c
0	1	2,3	-1
1	-1	-1	-1
2,3	-1	-1	2,3

FIGURE 2.10 – Automate déterministe minimisé pour **a|bc***

2.1.4 Utilisation de l'automate sur un texte

Pour trouver les *matchs* de la *RegEx* dans un texte, on doit parcourir chaque lettre de chaque mot de chaque ligne de ce dernier.

Si l'état initial de l'automate a une transition par le caractère qu'on lit, l'algorithme vérifie si l'état suivant a une transition par le caractère suivant, si c'est le cas, il continue d'avancer dans l'automate caractère par caractère, jusqu'à ce qu'il tombe sur un état sans transitions par le caractère lu. Dans ce cas là deux cas sont possibles :

1. L'état est final : C'est un *match*, alors la ligne est ajoutée à la liste des lignes qui matchent la *RegEx*.
2. L'état n'est pas final : Ce n'est pas un *match*, alors il vérifie si l'état initial de l'automate a une transition à partir de ce caractère, si ce n'est pas le cas, il passe au caractère suivant.

Pour gagner en performance, dès qu'on trouve un *match*, on ajoute la ligne à la liste résultante, sans la parcourir en entier.

2.2 Motif à suite de concaténations

Dans le cas où le motif recherché dans le texte est une suite de concaténations, on cherche les bouts du texte qui *matchent* en utilisant l'algorithme de **Knuth-Morris-Pratt**.

Pour l'implémentation on s'est inspirés de l'algorithme décrit sur le site web **geeksforgeeks** (cf. https://www.geeksforgeeks.org/kmp-algorithm-for-pattern-searching/)

Le recherche avec KMP se fait en plusieurs étapes :

— L'algorithme construit la table des plus longs préfixes qu'on appellera *carryOver* en cherchant les sous-chaînes du motif qui sont à la fois préfixe et suffixe.

— La comparaison commence par le caractère *motif[j]* (avec $\mathbf{j = 0}$) et le premier caractère de la première ligne du texte *ligne[i]* avec ($\mathbf{i = 0}$).

— Tant que le caractère *ligne[i] matche* avec *motif[j]*, **i** et **j** sont incrémentés.

— S'il ne matche pas, on sait déjà que *ligne[i-j...i-1]* matche avec *motif[0...j-1]* et que *carryOver[j-1]* est le nombre de caractères du *motif[0...j-1]* qui sont à la fois un préfixe et un suffixe, donc plus besoin de *matcher* les *carryOver[j-1]* avec *ligne[i-j...i-1]* parce qu'on sait d'avance qu'ils vont correspondre.

La complexité du **KMP** dans le pire cas est en $O(n)$. [7]

Chapitre 3

Mesure de performances

3.1 Comparaison des résultats

Avant de tester la vitesse de calcul et de recherche de l'algorithme décrit ci-dessus, des tests de vérification de la véracité des résultats retournés ont été fait.

Ci-dessous une comparaison des résultats retournés par l'algorithme implementé *(à gauche)* et la commande UNIX *egrep (à droite)* et sur la recherche du motif **S(a|g|r)*on** dans le fichier textuel https://www.gutenberg.org/files/56667/56667-0.txt.

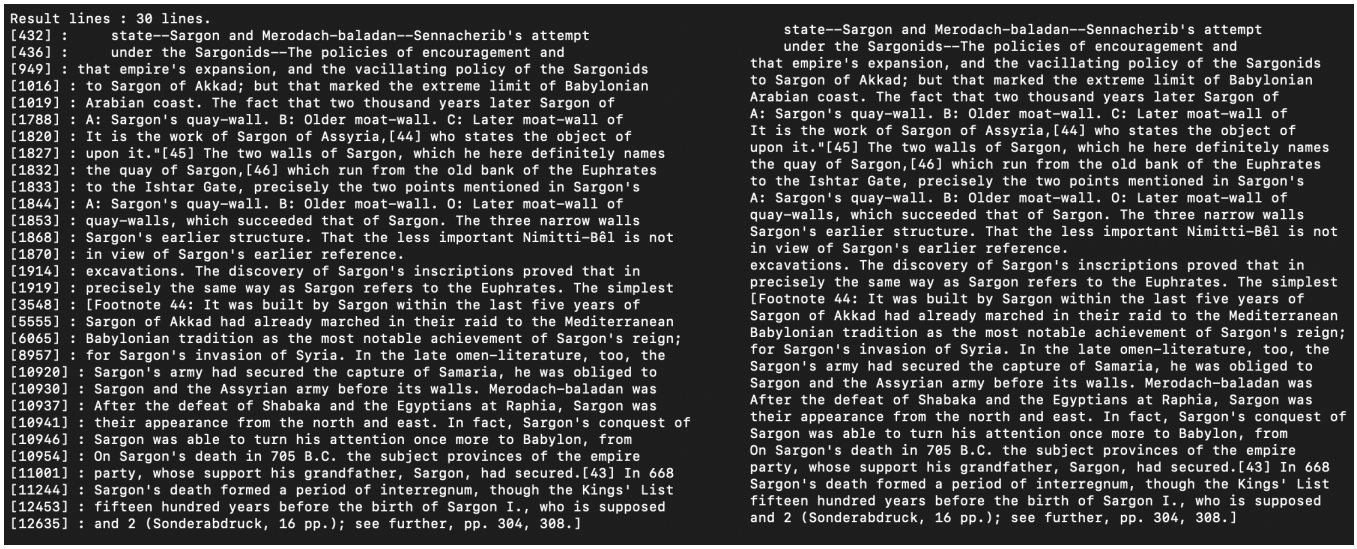


FIGURE 3.1 – Résultats de l'algorithme implementé vs la commande UNIX *egrep*

On remarque la similarité des résultats retournés par les deux commandes.

3.2 Temps de calcul

Ce qui est des temps de calcul, on remarque que même si l'algorithme de recherche implementé est assez rapide (*i.e.* < 1s), il reste beaucoup moins rapide que la commande *egrep* (*i.e.* < 0.05s).

La figure 3.2 ci-dessous résume les performances des trois technologies (commande egrep, DFA et KMP) sur plusieurs volumes du fichier cité plus haut (on entend par volume le nombre de lignes du fichier).

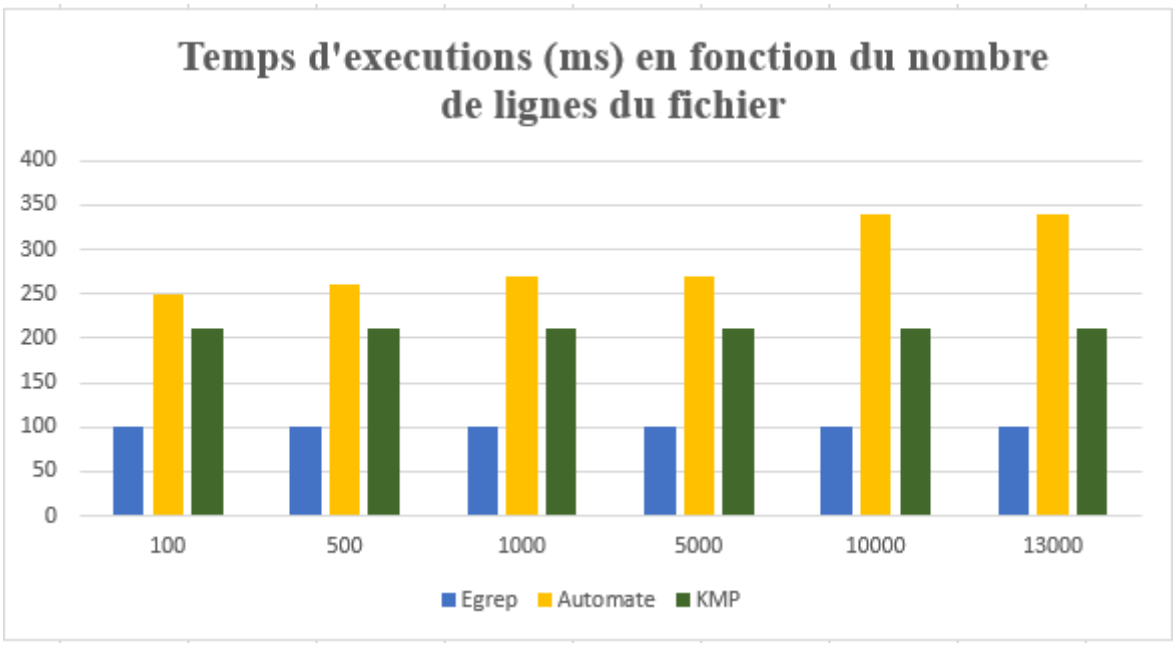


FIGURE 3.2 – Mesure du temps d'exécution en ms

On remarque que *egrep* et *KMP* démontrent une constance assez optimisée quant au temps de calcul (100 ms et 210 ms respectivement).
Notre algorithme de recherche par automate montre lui-aussi d'excellents résultats ($< 350ms$ pour 13k lignes) et permet donc un passage à l'échelle honorable.

3.3 Conclusions et perspectives

Par le biais de ce projet, nous avons implémenté une commande de recherche par mot-clé / RegEx qui présente les résultats équivalents à la commande UNIX de référence «*egrep*» et qui permet un passage à l'échelle conséquent pour pouvoir manipuler des données massives et ainsi en extraire de nouvelles connaissances.

Une certaine optimisation est aussi envisageable, notamment sur les structures de données utilisés ainsi que sur les nombreuses boucles imbriquées qui construisent l'automate.

Bibliographie

[1] « Digital Economy Compass ». In : (2019) (page 1).

[2] Alfred Aho et Jeffrey Ullman. « Foundations of Computer Science ». In : (1992) (pages 2, 3, 5).

[3] Oleg B. Lupanov. *A comparison of two types of finite sources*. Problemy Kibernetiki, 1966 (page 3).

[4] John E. Hopcroft, Rajeev Motwani et Jeffrey D. Ullman. *Introduction to automata theory, languages, and computation, 2nd Edition*. Addison-Wesley series in computer science. Addison-Wesley-Longman, 2001. isbn : 978-0-201-44124-6 (page 4).

[5] J. A Brzozowski. *Canonical regular expressions and minimal state graphs for definite events*. 1963 Proc. Sympos. Math. Theory of Automata. Polytechnic Press of Polytechnic Inst. of Brooklyn, Brooklyn, N.Y, 1963 (page 4).

[6] Edward F Moore. *Gedanken-experiments on sequential machines*. Automata studies. Princeton University Press, Princeton, N.J, 1956, p. 129-153 (page 4).

[7] Donald Knuth, James Morris et Vaughan Pratt. « Fast pattern matching in strings ». In : (1977) (pages 5, 10).

[8] Arthur Escriou. https ://slack-files.com/T02EWD10BC0-F02EZAUFV50-2002f5dbce (page 6).