SCIENCES SORBONNE UNIVERSITÉ

DAAR

Clone de egrep avec support partiel des ERE

Malek BOUZARKOUNA 28706508
Yacine KESSAL 21311739
06 octobre 2025

Superviseur académique :
Binh-Minh BUI-XUAN

Master STL 2024/2025

Table des matières

Résumé

Ce projet implémente une version simplifiée de la commande UNIX egrep pour reconnaître des motifs dans des fichiers texte à l'aide d'expressions régulières étendues (ERE). Nous allons également y intégrer des algorithmes, et les performances seront comparées à celles de egrep.

1 Introduction

Le traitement et la reconnaissance des motifs sont des domaines essentiels en informatique, notamment dans la manipulation de chaînes de caractères. Dans ce cadre, les expressions régulières (RegEx) jouent un rôle fondamental en permettant de définir et de reconnaître des motifs au sein de fichiers texte. Ce projet s'inscrit dans cette dynamique, en cherchant à reproduire le comportement de la commande UNIX `egrep`, tout en se limitant à une version simplifiée des expressions régulières étendues (ERE). L'objectif est de fournir une implémentation performante capable de reconnaître des motifs spécifiés par une ERE au sein de fichiers texte.

Le problème peut être abordé à travers plusieurs stratégies, allant de la transformation de l'expression régulière en un automate fini non déterministe (NFA) puis en un automate fini déterministe (DFA), à des méthodes d'optimisation basées sur l'indexation des sous-chaînes. Notre implémentation suivra les grandes lignes des algorithmes enseignés lors des cours de DAAR, tout en cherchant à améliorer la performance par l'exploration de techniques alternatives comme l'algorithme de Knuth-Morris-Pratt (KMP).

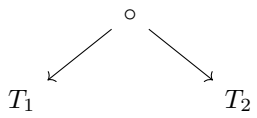
Dans les sections suivantes, nous détaillerons la méthodologie employée pour transformer et traiter les expressions régulières, ainsi que les structures de données associées. Nous comparerons également les performances de notre implémentation avec celles de `egrep`, afin de mettre en lumière les gains et les défis liés à notre approche.

2 Définitions et notations

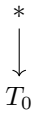
Pour commencer, nous proposons de définir quelques notions utiles à notre sujet et qui seront utilisées dans les algorithmes.

Definition 2.1 (Arbre abstrait) *Soit r une expression régulière. L'arbre abstrait de r est construit par induction sur la structure de r de la manière suivante :*

- *Si $r = a$: l'arbre est une simple feuille étiquetée par a.*
- *Si $r = r_1 \circ r_2$:*



  où T_i est l'arbre abstrait de r_i, $i \in \{1, 2\}$, et $\circ \in \{|, \cdot\}$, désignant l'opérateur de concaténation.
- *Si $r = r_0^*$:*



  où T_0 est l'arbre abstrait de r_0.
- *Si $r = (r_0)$: l'arbre abstrait de r est le même que celui de r_0. [1]*

Definition 2.2 (Expressions régulières (regexps)) *Une expression régulière est un modèle qui permet de rechercher ou manipuler des chaînes de caractères. Elle inclut des lettres simples, la concaténation (ex. : ab), le choix (ex. : $a \mid b$), et la répétition (ex. : a^*).[1]*

Definition 2.3 (Automate fini déterministe (DFA)) *Un **automate fini déterministe** (DFA) est un quintuplet $A = (Q, \Sigma, \delta, q_0, F)$ où :*

- *Q est un ensemble fini d'états.*
- *Σ est un ensemble fini de symboles (l'alphabet).*
- *$\delta : Q \times \Sigma \to Q$ est une fonction de transition qui, pour chaque paire (q, a), où $q \in Q$ et $a \in \Sigma$, définit un unique état $q' \in Q$.*
- *$q_0 \in Q$ est l'état initial.*

— $F \subseteq Q$ *est l'ensemble des états acceptants.*

Un DFA accepte une chaîne si, après avoir consommé tous les symboles de la chaîne, il termine dans un état appartenant à F*.[3]*

Definition 2.4 (NFA (Automate fini non déterministe)) *Un NFA est similaire à un DFA, mais il peut avoir plusieurs transitions possibles ou aucune pour un même symbole et état. Il accepte une chaîne si au moins une suite d'états mène à un état acceptant.[3]*

Definition 2.5 (DFA minimisé) *Un DFA minimisé est une version simplifiée d'un DFA, avec moins d'états, mais qui reconnaît le même langage. Les états redondants sont fusionnés ou supprimés.[3]*

3 Analyse et Présentation Théorique des Algorithmes Connus

Dans la littérature, plusieurs approches ont été abordées pour résoudre le problème du pattern matching avec des expressions régulières (regex), en utilisant des théorèmes de l'informatique théorique, ainsi que des approches déterministes et non déterministes. Dans cette section, nous allons brièvement analyser des algorithmes connus.

3.1 Algorithmes Basés sur les Automates

3.1.1 Algorithme de Thompson

L'algorithme de Thompson [9], conçu dans le cadre de la théorie des automates, est une méthode efficace pour construire un automate fini non déterministe (NFA) à partir d'une expression régulière. L'algorithme fonctionne en créant des états et des transitions qui représentent les différents symboles de l'expression régulière, en utilisant une approche récursive.

Dans un premier temps, Thompson décompose l'expression régulière en éléments de base : symboles simples, unions, concaténations et étoiles de Kleene. Pour chaque élément, il crée un sous-automate. Ensuite, ces sous-automates sont combinés pour former un automate global qui accepte le langage décrit par l'expression régulière initiale. Ainsi, l'algorithme construit des états et des transitions qui permettent de représenter de manière graphique la structure du langage.

En termes de complexité, l'algorithme de Thompson présente un temps de complexité de $O(n)$, où n représente la longueur de l'expression régulière. De plus, l'espace utilisé est également proportionnel à $O(n)$, car chaque symbole et opération peut créer de nouveaux états dans l'automate résultant.

3.2 Algorithmes Non Basés sur les Automates

3.2.1 Algorithme de Backtracking

L'algorithme de backtracking [8] est utilisé pour résoudre des problèmes de décision et d'optimisation, notamment dans les domaines de l'intelligence artificielle et de la théorie des graphes. Ce procédé consiste à explorer les différentes possibilités d'un espace de solutions, en construisant progressivement des candidats pour une solution et en abandonnant (ou "backtrack") ceux qui ne répondent pas aux critères souhaités.

Le fonctionnement du backtracking repose sur une approche récursive. À chaque étape, il choisit un élément d'une solution partielle, puis explore les conséquences de ce choix. Si cette solution partielle ne mène pas à une solution complète valide, l'algorithme revient en arrière, annule ce choix, et essaie une autre option. Ce processus continue jusqu'à ce que toutes les possibilités aient été explorées ou qu'une solution satisfaisante ait été trouvée.

En termes de complexité, celle de l'algorithme de backtracking peut varier considérablement selon le problème spécifique qu'il traite. En général, la complexité temporelle est souvent exponentielle, notée $O(b^d)$, où b représente le nombre de choix possibles à chaque niveau de la recherche et d la profondeur de la recherche. Toutefois, des optimisations, comme le choix de branches prometteuses, peuvent améliorer cette complexité dans certains cas. En dépit de sa puissance, l'algorithme de backtracking peut s'avérer coûteux en temps pour des problèmes de grande envergure, ce qui incite souvent à rechercher des méthodes heuristiques ou des approches approximatives pour des solutions plus rapides.

3.2.2 Algorithme de Boyer-Moore

L'algorithme de Boyer-Moore [2], publié en 1977, est une méthode de recherche de sous-chaînes dans un texte, particulièrement efficace lorsque le texte à analyser est long. Il se distingue par une approche innovante qui permet d'éviter d'examiner chaque caractère du texte dans l'ordre, comme le ferait une méthode naïve. Au lieu de cela, Boyer-Moore utilise des heuristiques pour sauter plusieurs caractères à la fois, réduisant ainsi le nombre de comparaisons nécessaires.

Son fonctionnement repose sur deux principales heuristiques. La première, appelée "heuristique du mauvais caractère", déplace la sous-chaîne vers la droite lorsqu'un caractère du texte ne correspond pas à celui attendu. La seconde, "heuristique de la bonne suffixe", utilise les motifs déjà comparés avec succès pour ajuster la position de la sous-chaîne, sautant ainsi plusieurs caractères du texte.

En termes de complexité, Boyer-Moore présente un meilleur cas très performant de $O(n/m)$, où n est la longueur du texte et m celle de la sous-chaîne recherchée. Dans le pire des cas, la complexité reste $O(n \cdot m)$, bien que ce scénario soit rare. Grâce à ses optimisations, l'algorithme est souvent le choix privilégié pour la recherche dans des textes volumineux, en particulier lorsqu'il y a peu de correspondances à trouver.

Les algorithmes pour le pattern matching avec des expressions régulières sont variés et offrent différents compromis entre flexibilité, efficacité et complexité. Les algorithmes basés sur les automates, comme celui de Thompson et la conversion NFA à DFA, sont puissants mais peuvent être coûteux en mémoire. Les algorithmes non basés sur les automates, comme le backtracking et Boyer-Moore, offrent des approches alternatives avec leurs propres avantages et inconvénients.

4 Clone de la commande Egrep

Pour simuler la commande `grep`, nous employons deux implémentations : la première repose sur l'algorithme de Knuth-Morris-Pratt pour les suites de concaténation, tandis que la seconde utilise la méthode d'Aho-Ullman pour les autres cas. Pour déterminer quelle méthode appliquer, nous recourons à la fonction `istreeconcat()`, qui parcourt l'arbre syntaxique de l'expression régulière afin de vérifier si les racines correspondent à des concaténations.

4.1 Méthode de Aho-Ullman

La méthode de Aho-Ullman [4]transforme une expression régulière (RegEx) en un automate fini non déterministe (NFA), puis en un automate fini déterministe (DFA). Ce processus permet de tester si une chaîne correspond à une expression régulière donnée. L'exemple utilisé dans cette section, ainsi que les illustrations, proviennent du livre classique de Aho et Ullman sur les compilateurs, et sont basés sur l'expression régulière `a|bc*`.

4.1.1 Structure de données utilisée

L'algorithme repose sur deux structures de données principales pour représenter les automates, qui sont manipulées dans le code.

— **Transitions** : Dans le NFA et le DFA, les transitions sont modélisées sous forme de liste d'adjacence. Chaque état est associé à une liste de transitions, où chaque transition est représentée par un couple `(symbole, état suivant)`. Le symbole peut être un caractère de l'expression régulière ou ε pour les transitions spontanées dans le NFA.

La structure `transitions` est donc une liste de listes, où chaque sous-liste contient les transitions sortantes d'un état spécifique, chacune stockée sous la forme `(symbole, état)`.

— **États finaux** : Les états finaux du NFA et du DFA sont stockés dans un ensemble, permettant de représenter plusieurs états finaux. Un état final désigne celui où l'automate accepte l'entrée. L'ensemble `finalStates` regroupe tous les états représentant des états de terminaison acceptants dans l'automate.

4.1.2 Explication des opérations sur la structure

— **Ajout des transitions** : Chaque transition est créée via la méthode `addTransitionToAutomate()`, qui prend en paramètre l'état source, l'état cible et le symbole associé (ou ε pour les transitions non déterministes dans le NFA). Elle est représentée par une liste contenant le `symbole` et l'état cible (`fils`), puis ajoutée aux transitions sortantes de l'état source (`parent`).

— **Ajout d'un état final** : Les états particulier sont ajoutés à l'ensemble des états finaux à l'aide de la méthode `addFinalState()`.

— **Fusion des sous-automates** : La méthode `Fusion()` permet de combiner plusieurs sous-NFAs pour créer un automate plus grand, comme dans le cas de la concaténation ou de l'alternance. Cela est utilisé pour combiner les parties de l'expression régulière dans un NFA global.

4.1.3 Fonctionnement de l'algorithme

Le fonctionnement de l'algorithme sera illustré à travers l'exemple de l'expression `a|bc*`, que nous détaillons en quatre étapes principales :

1. **Construction de l'arbre de RegEx** : L'expression régulière est représentée sous forme d'arbre syntaxique. Pour `a|bc*`, l'alternance (`|`) constitue le nœud racine, avec `a` en tant que sous-arbre gauche et `bc*` à droite, où `b` est concaténé avec `c*`.

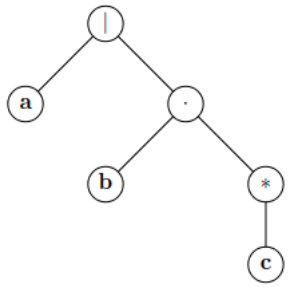
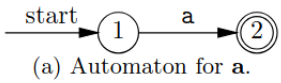


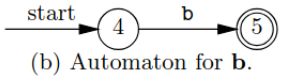
FIGURE 1 – Arbre de syntaxe pour `a|bc*`.

Dans la Figure 1, l'opérateur `|` est le nœud racine, illustrant la structure hiérarchique de l'expression. Des appels récursifs permettent de construire les sous-automates non déterministes (NFA) pour chaque segment de l'expression.

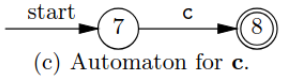
2. **Transformation en automate non déterministe (NFA)** : Chaque feuille de l'arbre syntaxique d'une expression régulière est convertie en un sous-automate NFA. Ces sous-automates sont ensuite combinés pour former l'automate global, conformément aux règles de construction des automates décrites dans la Figure 3a. Par exemple, le symbole `a` génère un NFA simple, comme illustré dans la Figure 2. De même, l'expression `bc*` produit un sous-automate NFA distinct (voir Figure 2), où l'étoile (*) indique que le symbole `c` peut apparaître zéro ou plusieurs fois. Il est important de noter que, même lorsque le même opérande se répète dans l'expression, des états distincts sont utilisés pour chaque occurrence, afin d'assurer la cohérence de l'automate.



(a) Automaton for **a**.



(b) Automaton for **b**.

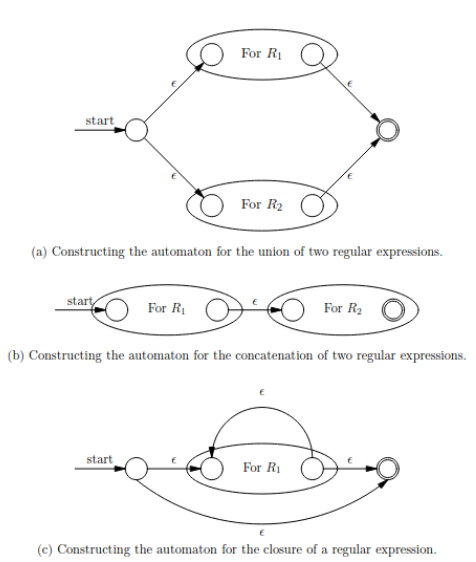


(c) Automaton for **c**.

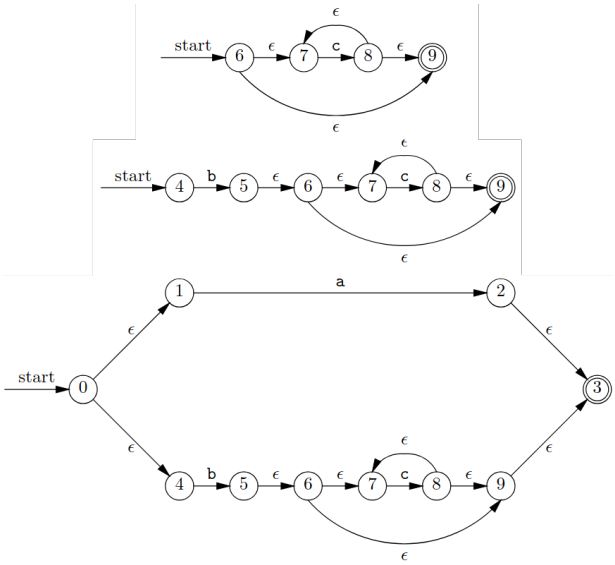
FIGURE 2 – Automate pour a, b, et c

Nous construisons le NFA complet en suivant l'arbre de la Figure 1. L'opérateur de fermeture est appliqué à `c` en utilisant la règle c de la Figure 3a, comme illustré dans la Figure 3b pour `c*`. Ensuite, la concaténation de `b` et `c*` est réalisée avec la régle b de la Figure 3a pour `bc*`. Enfin, l'union de `a` et `bc*` utilise la règle a de la Figure 3a, conduisant à l'automate final de la Figure 3b.

3. **Transformation en automate déterministe (DFA)** : Le NFA est ensuite converti en un automate sans transitions ϵ en regroupant les transitions ϵ en transitions directes étiquetées avec les symboles correspondants. Par exemple, les transitions ϵ entre l'état 0, 1, et 4 de la Figure 3b sont combinées avec les transitions sur `a` et `b`, créant des transitions directes de l'état 0 à l'état 2 (étiquetée `a`) et de l'état 0



(a) règles de construction des automates



(b) construction NFA pour `a|bc*`

FIGURE 3 – Etape NFA pour `a|bc*`

à l'état 5 (étiquetée `b`). Ce processus, basé sur la méthode de subset construction, simplifie les chemins ϵ-transitions en un automate plus compact, comme illustré dans la Figure 4 tout en conservant le même langage accepté.

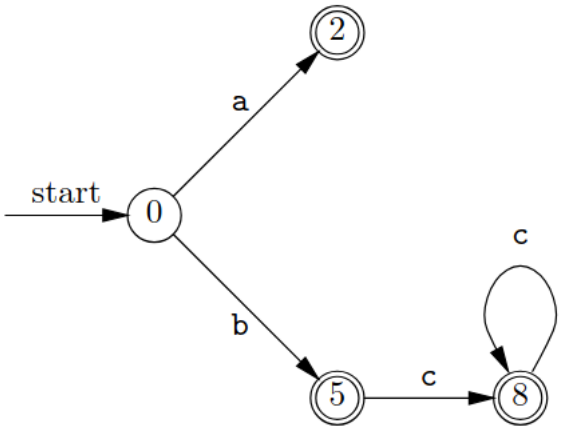


FIGURE 4 – construction DFA pour `a|bc*` avec subset construction

Il existe un automate déterministe plus simple qui accepte le même langage que celui illustré dans la Figure 4 . Cet automate, présenté dans la Figure 5 , résulte de la reconnaissance que les états 5 et 8 sont équivalents et peuvent donc être fusionnés.

4. **Minimisation du DFA** : Nous avons utilisé la méthode de minimisation d'automate proposée par un utilisateur sur GitHub (kashaf12 [6]) pour optimiser notre automate. Cette approche permet de réduire le nombre d'états du DFA, améliorant ainsi ses performances lors de l'exécution.Comme illustré dans la Figure 5, le DFA final après minimisation pour l'expression régulière `a|bc*` montre une réduction du nombre d'états grâce à la méthode de minimisation.

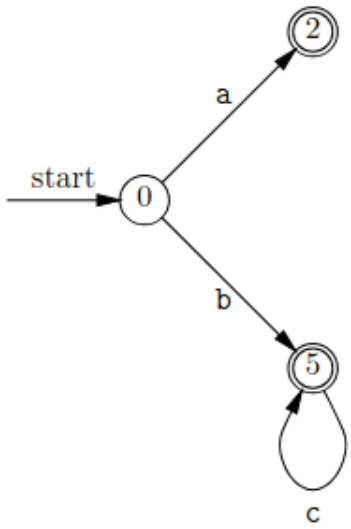


FIGURE 5 – DFA final après minimisation pour `a|bc*`.

4.1.4 Conclusion

La méthode de Aho-Ullman démontre une efficacité remarquable pour transformer des expressions régulières, telles que `a|bc*`, en automates. En exploitant des structures comme l'arbre syntaxique et en optimisant les transitions via des NFAs et DFAs, l'algorithme assure une exécution rapide et précise. Les illustrations fournies clarifient chaque étape du processus, de la construction de l'arbre à la minimisation de l'automate.

4.2 Algorithme de Knuth-Morris-Pratt

L'algorithme de Knuth-Morris-Pratt (KMP) [7] est une méthode efficace pour rechercher une sous-chaîne dans une chaîne. Conçu en 1970 par Donald Knuth, James H. Morris Jr. et Vaughan Pratt, cet algorithme améliore l'approche naïve de recherche de sous-chaîne en utilisant un prétraitement pour éviter des comparaisons inutiles, ce qui réduit le temps de recherche global.

4.2.1 Principe de l'Algorithme KMP

L'approche naïve de recherche d'une chaîne P dans un texte T consiste à décaler la chaîne P le long du texte T, de gauche à droite, d'une lettre à chaque étape. À chaque fois, on compare la chaîne P et la portion $T[s+1..s+|P|]$ du texte T. Cette méthode a une complexité temporelle de $O(|P| \cdot |T|)$ dans le pire des cas, où $|P|$ est la longueur du motif P et $|T|$ la longueur du texte T.

L'algorithme KMP [5] améliore l'approche naïve en utilisant l'information acquise lors des comparaisons lettre à lettre. Ainsi, on peut décaler le motif P de plus d'une lettre à chaque étape, ce qui permet de réduire le nombre de comparaisons nécessaires.

4.2.2 Fonction Préfixe

La fin du texte déjà lu correspond à la fin de la portion du motif qui coïncide. On se ramène donc à l'étude du motif. En particulier, pour tout préfixe du motif $P[1..k]$, on cherche la longueur $\pi[k]$ du plus long préfixe de $P[1..k]$ qui est aussi suffixe de $P[1..k]$. Plus ce plus long préfixe/suffixe est court, plus on décale. Dans l'exemple ci-dessus, $\pi[5]$ vaut 3 car le mot "ABA" est le plus long qui soit à la fois un préfixe propre et un suffixe propre de $P[1..k]$. La fonction π s'appelle fonction préfixe. Voici la table qui donne les valeurs de $\pi[i]$ pour le motif "ABABACA" :

i	1	2	3	4	5	6	7
$P[i]$	A	B	A	B	A	C	A
$\pi[i]$	0	0	1	2	3	0	1

TABLE 1 – Table de préfixes pour le motif "ABABACA"

4.2.3 Structure de l'Algorithme

L'algorithme de Knuth-Morris-Pratt se compose de deux phases :

1. **Calcul de la fonction préfixe** : L'algorithme construit la fonction préfixe π.
2. **Phase de recherche** : On recherche le motif P dans le texte T en utilisant la fonction préfixe π pour déterminer le décalage.

4.2.4 Complexité Temporelle

L'algorithme de Knuth-Morris-Pratt (KMP) se décompose en deux phases : le calcul de la table de préfixes et la recherche du motif dans le texte. Comme le calcul de la fonction préfixe est similaire à la phase de recherche, l'analyse de la complexité temporelle de ces deux phases est identique.

Théorème 1 *L'algorithme KMP recherche un motif P de longueur $|P|$ dans une chaîne de texte T de longueur $|T|$ avec une complexité temporelle de $O(|T| + |P|)$.*

Le calcul de la table de préfixes se fait en un temps $O(|P|)$, car chaque caractère du motif est visité au plus une fois. De même, lors de la phase de recherche, chaque caractère du texte est comparé avec le motif au plus une fois grâce à l'utilisation de la table de préfixes pour éviter des comparaisons inutiles. Ainsi, la recherche s'effectue également en un temps $O(|T|)$.

Par conséquent, la complexité globale de l'algorithme KMP est en $O(|T| + |P|)$.

4.2.5 Phase de Recherche

Algorithm 1: Algorithme KMP (Knuth-Morris-Pratt)

Input : T (texte), P (motif)
Output: Index de la première occurrence de P dans T, ou -1 si non trouvée

```
1  Calculer la table de préfixes π pour P;
2  i ← 0, j ← 0;
3                                              ▷ Initialisation des index pour T et P
4  while i < longueur(T) do
5      if T[i] = P[j] then
6          if j = longueur(P) − 1 then
7              return i − j;
8                                                          ▷ Correspondance trouvée
9          else
10             i ← i + 1;
11             j ← j + 1;
12         end
13     else
14         if j > 0 then
15             j ← π[j − 1];
16                                          ▷ Utiliser la table de préfixes pour décaler P
17         else
18             i ← i + 1;
19         end
20     end
21 end
22 return-1;
23                                                    ▷ Aucune correspondance trouvée
```

4.2.6 Calcul de la Fonction Préfixe

Algorithm 2: Calcul de la Table de Préfixes

Input : P (motif)
Output: Table de préfixes π pour P

```
Soit π[1..|P|] un tableau;
π[1] ← 0;
k ← 0;
for q = 2 to |P| do
    while k > 0 et P[k + 1] ≠ P[q] do
        k ← π[k];
    end
    if P[k + 1] = P[q] then
        k ← k + 1;
    else
        k ← 0;
    end
    π[q] ← k;
end
return π;
```

4.2.7 Conclusion

L'algorithme de Knuth-Morris-Pratt est une méthode efficace pour la recherche de sous-chaînes dans une chaîne, grâce à son prétraitement qui permet de réduire le nombre de comparaisons nécessaires. Sa complexité temporelle linéaire en fait un outil précieux pour les applications nécessitant des recherches rapides et efficaces.

5 Résultats Expérimentaux

Dans cette section, nous présentons les résultats de nos expérimentations. Cette étape est cruciale, car elle nous permettra de mettre à l'épreuve les algorithmes étudiés dans les sections 4.1 et 4.2. Les critères pour évaluer sont le temps d'exécution moyen des algorithmes et leurs correction. Ces tests, seront réalisés avec l'ebook "The Project Gutenberg EBook of A History of Babylon" qui est un livre contenant 13308 lignes et, pour ce qui est des conditions des expérimentations, les tests ont été effectués sur un processeur 12th Gen Intel(R) Core(TM) i5-12450H, 2000 MHz, 8 cœur(s), 12 processeur(s) logique(s).

Ils ont été exécutés dans un environnement Java, Les motifs de recherche ont été extraits d'un fichier nommé `most_used_words.txt`, qui contient des mots couramment utilisés dans la langue anglaise, servant de base pour les recherches des algorithmes. En parallèle, les résultats des algorithmes KMP et DFA ont été comparés à ceux obtenus via le `grep` de linux, utilisé comme référence pour la recherche de motifs.

5.1 Correction

La validation des algorithmes a été réalisée en comparant les résultats des algorithmes KMP et DFA à ceux de `grep`. Chaque algorithme a été exécuté sur le même ensemble de données, en appliquant successivement les motifs de recherche tirés du fichier `most_used_words.txt`. Le nombre d'occurrences trouvées pour chaque motif a été enregistré pour chaque algorithme.

Les résultats obtenus ont été collectés dans un fichier CSV (`Grep_vs_DFA_vs_KMP_Comparison.csv`) afin de pouvoir visualiser la concordance ces derniers. Pour qu'un test soit validé, les trois algorithmes devaient retourner un nombre identique d'occurrences pour chaque motif. Cela a permis d'évaluer la précision et la conformité des algorithmes KMP et DFA vis-à-vis de `grep` ce qui nous permettra de conclure que nos implémentations renvoient bien les résultats escomptés et qu'ils sont donc correctes.

5.2 Évaluation des Performances

Pour mesurer l'efficacité des algorithmes implémentés, une série de tests de performance a été réalisée. L'objectif de ces tests était de comparer les temps d'exécution des algorithmes KMP et DFA en fonction de la taille des données et de les confronter à ceux de `grep`.

Les données d'entrée ont été divisées en sous-ensembles de différentes tailles : 100, 500, 1000, 5000, 10000, 12000, et 15000 lignes. Pour chaque sous-ensemble, les algorithmes ont été exécutés à plusieurs reprises (10 répétitions) afin d'obtenir des temps d'exécution moyens plus représentatifs. les durées sont en nanosecondes pour chaque motif et chaque algorithme.

Les résultats des tests de performance ont été agrégés dans un fichier CSV (`Algorithms-Time-Execution.csv`). Ce fichier contient, pour chaque taille de sous-ensemble, le temps moyen d'exécution des algorithmes KMP, DFA et `grep`. Ces résultats ont ensuite été analysés afin de déterminer la complexité temporelle des algorithmes et d'identifier lequel présente les meilleures performances sur de grands volumes de données.

5.3 Critères d'évaluation

Les tests ont été validés selon deux critères principaux : l'exactitude des résultats et la performance. Pour l'exactitude, les résultats des algorithmes KMP et DFA devaient être conformes à ceux obtenus avec `grep`, garantissant ainsi la validité des implémentations. Du point de vue des performances, les temps d'exécution des algorithmes ont été comparés afin d'évaluer leur efficacité selon la taille des données d'entrée.

5.4 Résultats

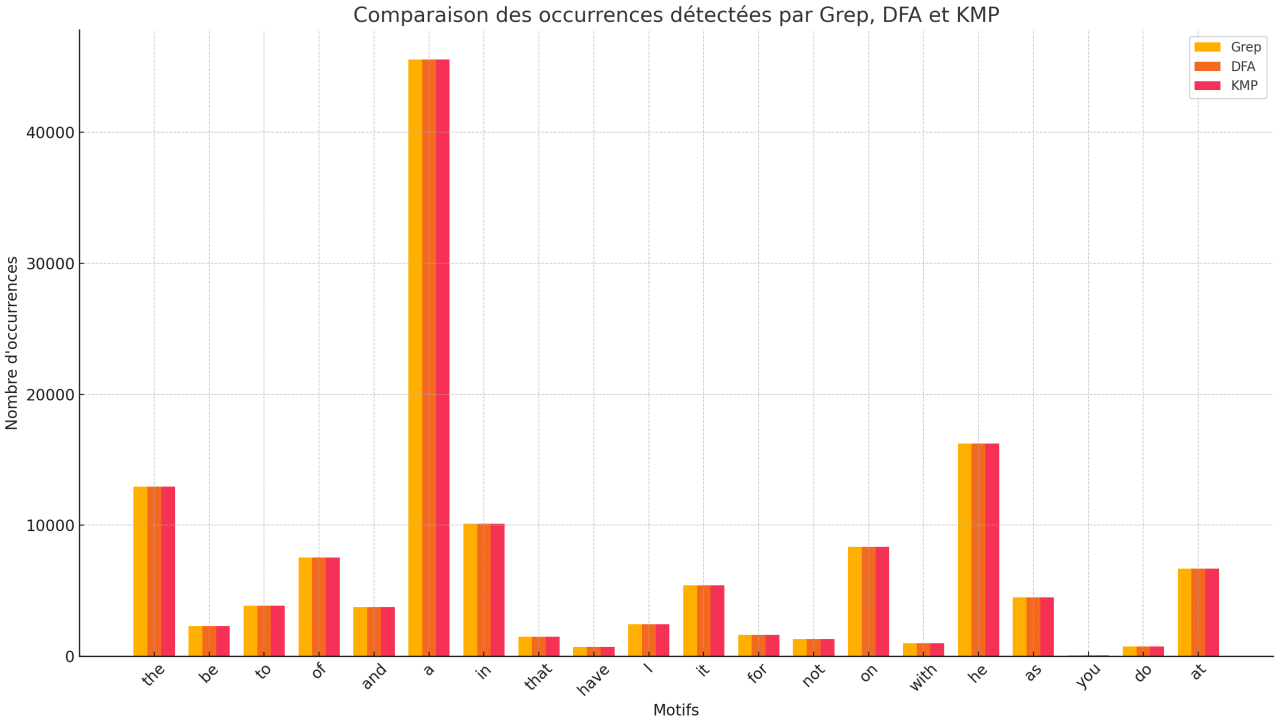


FIGURE 6 – Comparaison des occurrences détectées par Grep, DFA et KMP pour différents motifs.

La figure 6 présente une comparaison des occurrences de motifs détectées par les trois algorithmes : Grep, DFA et KMP. L'axe des abscisses représente les différents motifs (par exemple, "the", "be", "to", etc.), tandis que l'axe des ordonnées indique le nombre d'occurrences détectées pour chaque motif. Les barres pour chaque motif montrent les résultats obtenus par les trois algorithmes. Dans tous les cas, on remarque des occurrences similaires pour les trois algorithmes. Cela suggère que nos implémentations sont fiables et offrent des résultats similaires pour la recherche de motifs dans le texte analysé.

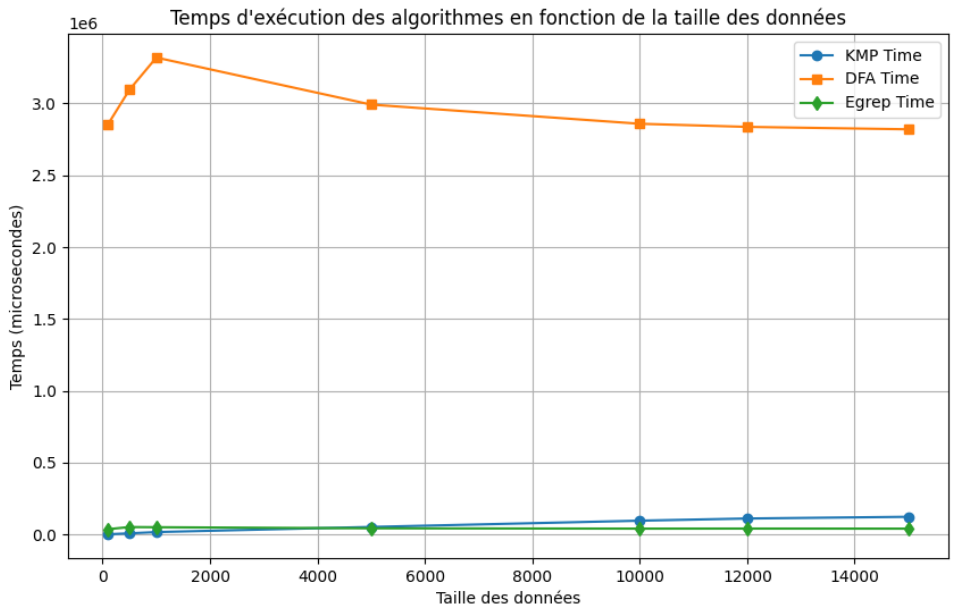


FIGURE 7 – Comparaison des temps d'exécution de KMP, DFA et Grep en millisecondes en fonction du nombre de ligne.

La figure 7 montre que L'algorithme DFA affiche des temps d'exécution significativement plus élevés, atteignant environ 2.85 seconds pour des nombres de lignes allant de 100 à 15 000, ce qui indique une relative inefficacité pour des nombres de lignes variables. Cette constante, très haute, reflète probablement une charge élevée associée à la construction de l'automate fini, particulièrement lors de l'initialisation. Les temps d'exécution de DFA tendent à rester élevés et stables à mesure que la taille des données augmente, montrant que l'algorithme n'est pas particulièrement bien adapté aux grandes tailles de données dans un contexte où la rapidité est cruciale.

En revanche, l'algorithme KMP présente des temps d'exécution bien inférieurs, oscillant entre 1 660 microsecondes pour un nombre de ligne de 100 et environ 123 000 microsecondes pour 15 000 ligne. Sa courbe de performance montre une croissance modérée avec l'augmentation de la taille des données, dû à sa complexité linéaire en $O(n+m)$ de l'algorithme, où n est la taille du texte et m celle du motif. Cela fait de KMP un choix approprié pour les applications où la performance doit rester stable face à des volumes de données faibles.

Egrep, quant à lui, se distingue par sa rapidité d'exécution et sa scalabilité, avec des temps variant entre 37 027 microsecondes pour les petites tailles de données et environ 52 000 microsecondes pour les grandes tailles. Egrep maintient une courbe constante à travers les différentes tailles de données, suggérant l'efficacité de ses optimisations pour des opérations de recherche sur des données volumineuses.

En résumé, cette analyse met en lumière l'efficacité des algorithmes KMP et Egrep par rapport à DFA. Tandis que DFA présente des temps d'exécution excessifs pour toutes les tailles de données, KMP et Egrep se révèlent être des choix plus judicieux pour les applications nécessitant des temps d'exécution rapides et une bonne adaptabilité à la croissance des données. En particulier, pour des applications de grande envergure où la performance est cruciale, KMP et Egrep offrent des solutions plus performantes et plus fiables.

6 Discussion

Dans ce rapport, nous avons mis en œuvre la méthode Aho-Ullman ainsi que l'algorithme de Knuth-Morris-Pratt (KMP) pour aborder la problématique de la recherche de motifs dans des chaînes de caractères.

Les tests effectués montrent que notre implémentation de la méthode Aho-Ullman, bien qu'elle soit plus lente que l'outil egrep, offre une stabilité certaine, même face à des fichiers texte de grande taille, maintenant un temps de traitement linéaire. Toutefois, cette méthode présente des défis liés à la consommation de mémoire, en raison de la taille potentiellement importante des automates finis déterministes (DFA), particulièrement pour des expressions régulières complexes. De plus, la transformation d'un automate non déterministe (NFA) en un DFA peut s'avérer extrêmement coûteuse en ressources.

Quant à l'algorithme KMP, il se montre particulièrement efficace pour traiter des chaînes formées de concaténations, avec un temps d'exécution linéaire, se rapprochant ainsi des performances de egrep.

Les perspectives dans ce domaine incluent l'exploration de techniques d'optimisation et de compression des automates afin de réduire l'utilisation mémoire. L'introduction de méthodes d'apprentissage automatique pourrait également améliorer la précision et l'efficacité de la reconnaissance des motifs. En outre, l'adoption de structures de données avancées, telles que les arbres et tableaux de suffixes, pourrait rendre la recherche de motifs plus rapide en offrant une indexation performante des sous-chaînes.

Enfin, l'adaptation des algorithmes actuels pour traiter des données en temps réel, en particulier dans le cadre des moteurs de recherche, apparaît comme une piste prometteuse. Ces systèmes doivent être capables de répondre rapidement à des requêtes dynamiques, exigeant des algorithmes hautement optimisés.

En conclusion, la recherche de motifs dans les fichiers textuels reste un domaine riche en défis et en opportunités. Les progrès futurs pourraient transformer les technologies de traitement de texte et de recherche d'information, en apportant des solutions plus rapides, plus précises et plus efficaces.

Références

[1] *AFD depuis RegEx*. Accessed : 2024-09-30. URL : https://agreg-maths.fr/uploads/versions/1502/AFDdepuisRegEx.pdf.

[2] *Algorithme de Boyer-Moore*. Accessed : 2024-10-06. URL : https://fr.wikipedia.org/wiki/Algorithme_de_Boyer-Moore.

[3] *Automata theory*. Accessed : 2024-10-06. URL : https://en.wikipedia.org/wiki/Automata_theory.

[4] Maxime Crochemore et Wojciech Ryller. *Text Algorithms*. Chapter 10 : Patterns, Automata, and Regular Expressions. Oxford University Press, 1990, p. 34-42. ISBN : 0-19-508609-0. URL : http://infolab.stanford.edu/~ullman/focs/ch10.pdf.

[5] GeeksforGeeks. "Java Program for KMP Algorithm for Pattern Searching". In : *GeeksforGeeks* (). Accessed : 2024-09-29. URL : https://www.geeksforgeeks.org/java-program-for-kmp-algorithm-for-pattern-searching/.

[6] Kashaf12. *DFA Minimization in Java*. GitHub. Accessed : 2024-09-29. URL : https://github.com/kashaf12/DFA-Minimization-Java/blob/master/src/com/minimizer/DFA_Minimizer.java.

[7] Donald Knuth, James H. Morris Jr. et Vaughan Pratt. "Fast pattern matching in strings". In : *SIAM Journal on Computing* 6.2 (1977), p. 323-350. URL : https://www.cs.jhu.edu/~misha/ReadingSeminar/Papers/Knuth77.pdf.

[8] *Retour sur trace*. Accessed : 2024-09-30. URL : https://fr.wikipedia.org/wiki/Retour_sur_trace.

[9] *Thompson's Construction*. Accessed : 2024-09-30. URL : https://en.wikipedia.org/wiki/Thompson%27s_construction.