Clone de egrep avec support partiel des ERE

Ecrit par:

SADI Aksil 21427293
SADI Massin 21416091

Encadré par les enseignants :

Bùi Xuân Bình Minh

Promotion 2025 - 2026

Table des matières

1 Partie introductive

1.1 Introduction

Dans un monde où les volumes de données textuelles ne cessent de croître, la capacité à rechercher efficacement des motifs précis dans une masse d'informations est devenue essentielle. Qu'il s'agisse de filtrer des journaux système, d'analyser du code source ou d'extraire des fragments pertinents dans un texte, les outils de recherche reposent sur une même idée : décrire ce que l'on cherche plutôt que de le parcourir manuellement.

Les expressions régulières permettent de formuler, à l'aide d'une syntaxe compacte, des règles capables d'identifier un ensemble potentiellement infini de chaînes de caractères. Derrière cette simplicité apparente se cache une base théorique solide issue des langages formels et des automates finis.

Le projet présenté dans ce rapport propose de reconstruire, à une échelle pédagogique, le comportement du programme `egrep`, outil emblématique des systèmes UNIX, en se concentrant sur son cœur algorithmique : la reconnaissance des motifs à partir d'expressions régulières. L'enjeu n'est pas seulement de reproduire un outil existant, mais de comprendre les mécanismes qui le rendent possible: analyse syntaxique, construction d'automates et reconnaissance de chaînes.

2 Définition du problème et structure de données utilisée

2.1 Définition du problème

Le problème des expressions régulières consiste à déterminer comment un motif textuel, défini par une expression régulière, peut être interprété et traité de manière à identifier efficacement toutes les chaînes de caractères qui lui correspondent.

Dans le contexte actuel, où les volumes de données textuelles sont très importants, il est crucial de disposer d'outils capables de rechercher et de reconnaître rapidement des motifs spécifiques au sein de textes longs ou nombreux.

Le défi principal réside donc dans la conception d'un mécanisme qui, pour une expression régulière donnée et un ensemble de chaînes de caractères, puisse décider efficacement pour chaque chaîne si elle satisfait le motif ou non.

2.2 Structure de données utilisée

La mise en œuvre du projet repose sur trois structures de données principales : l'arbre syntaxique des expressions régulières, l'automate non déterministe à transitions ϵ (NDFA) et l'automate déterministe (DFA). Ces structures sont étroitement liées et interviennent successivement dans le processus

de reconnaissance des motifs.

1. Arbre syntaxique des expressions régulières

L'arbre `RegexArbre` représente la structure hiérarchique d'une expression régulière. Chaque nœud correspond soit à un opérateur (`*`, `+`, `|`, `.`), soit à un symbole terminal (caractère). Les sous-arbres décrivent la composition des sous-expressions. Cette représentation facilite la traduction systématique d'une expression régulière vers un automate NDFA.

2. Automate non déterministe (NDFA)

La classe `Ndfa` modélise un automate non déterministe avec transitions ϵ. Elle se compose d'un *état initial* et d'un *état final*. Chaque état, représenté par la classe `Etat`, maintient :

- un ensemble de transitions étiquetées par des symboles (caractères ASCII) ;
- un ensemble de transitions ϵ (sans consommation de symbole).

Cette structure permet de représenter fidèlement la sémantique d'une expression régulière selon la construction de Thompson.

3. Automate déterministe (DFA)

La classe `Dfa` représente un automate déterministe obtenu à partir du NDFA par la méthode de la *déterminisation*. Chaque état du DFA correspond à un ensemble d'états du NDFA. La classe interne `Etat` contient :

- un identifiant unique ;
- une table de transitions (symbole $\rightarrow$ état) ;
- une méthode pour accéder à l'état suivant selon un symbole donné.

Cette structure permet une reconnaissance efficace des chaînes, car chaque symbole conduit vers un seul état suivant.

Ainsi, l'ensemble de ces structures permet de passer progressivement :

$$\text{Expression régulière} \Rightarrow \text{NDFA} \Rightarrow \text{DFA}$$

pour obtenir un mécanisme complet de reconnaissance de motifs.

3 Analyse et presentation théorique des algorithmes

3.1 Méthode d'Aho-Ullman

Cette section présente les principaux algorithmes utilisés dans la traduction et l'optimisation des expressions régulières, selon le processus défini par Aho et Ullman [6]. Ce processus se décompose en

4 étapes fondamentales : Transformation de l'expression régulière en arbre syntaxique,la construction de l'automate non déterministe (algorithme de Thompson), sa déterminisation, puis la minimisation de l'automate obtenu (algorithme de Hopcroft).

3.1.1 Transformation en arbre syntaxique

Principe: Avant toute construction d'automate, l'expression régulière est d'abord convertie en un **arbre syntaxique** (ou arbre d'expression). Chaque nœud interne de l'arbre représente un opérateur (comme la concaténation, l'alternance ou l'étoile), tandis que chaque feuille correspond à un symbole terminal de l'expression. Cette représentation arborescente permet de traiter la regex de manière récursive, ce qui facilite la construction systématique de l'automate selon l'algorithme de Thompson.

Exemple: Considérons l'expression régulière suivante : $a|bc^*$.

Son arbre syntaxique aura comme racine l'opérateur d'alternance (`|`). Le sous-arbre gauche correspond à la feuille `a`, et le sous-arbre droit représente la concaténation (`.`) entre `b` et `c*`, où `c*` indique une étoile de Kleene appliquée au symbole `c`.

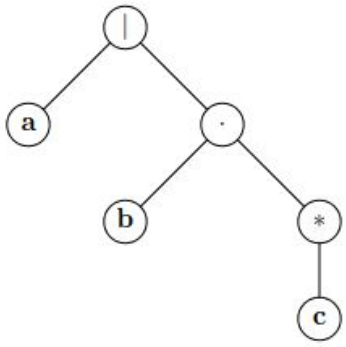


Figure 1: Arbre de syntaxe pour a|bc*

3.1.2 Algorithme de Thompson

Principe : L'algorithme de Thompson [1] permet de construire un automate non déterministe à transitions ϵ (NDFA) à partir d'une expression régulière. Le principe repose sur la composition récursive d'automates élémentaires correspondant à chaque opérateur :

- **Concaténation** : relie la fin du premier automate au début du second par une transition ϵ.
- **Alternance (|)** : crée un nouvel état initial et final, chacun relié par des transitions ϵ aux deux sous-automates.
- **Étoile de Kleene (*)** : ajoute une boucle ϵ permettant de répéter ou d'ignorer le sous-automate.

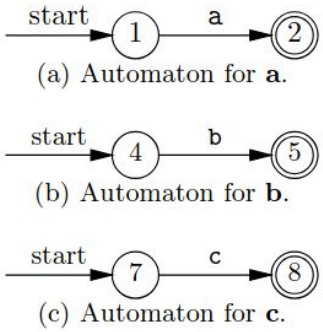
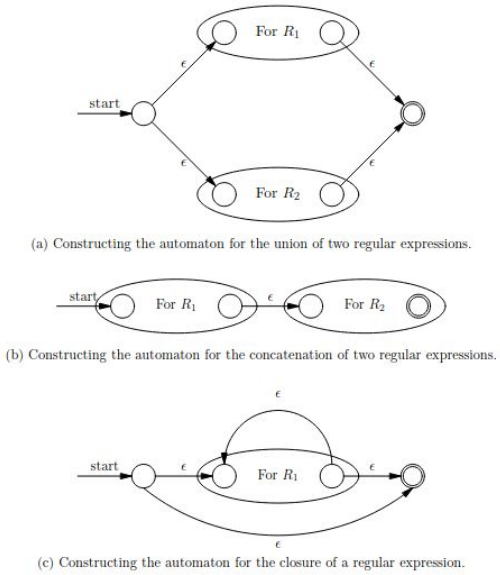
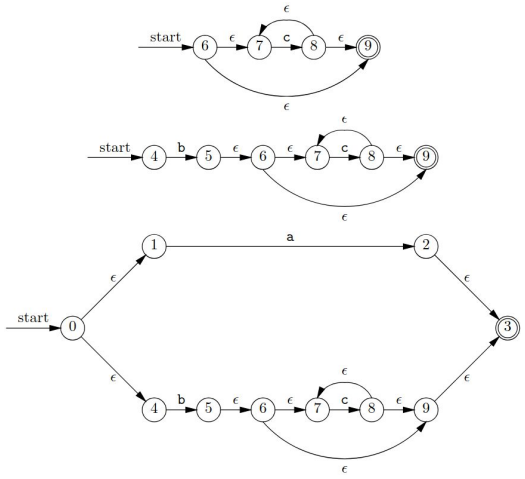


Figure 2: Automates de base pour les symboles a, b, et c.



(a) Opérations de base (union, concaténation, fermeture).



(b) Construction complète pour $a|bc^*$.

Figure 3: Illustration du processus de construction selon la méthode de Thompson.

Les figures 2, 3a et 3b illustrent respectivement les étapes de la construction selon la méthode de Thompson.

Pseudo-code :

Algorithm 1 Construction NDFA par Thompson

Input: Expression régulière $expr$
Output: NDFA N correspondant à $expr$
if $expr$ *est un symbole* **then**
  retourner NDFA simple avec une transition
else if $expr = (A \cdot B)$ **then**
  construire $NDFA\_A$ et $NDFA\_B$ relier fin$(A) \rightarrow$ début(B) par ϵ
else if $expr = (A \mid B)$ **then**
  créer un nouvel état initial et final relier initial $\rightarrow A$ et initial $\rightarrow B$ par ϵ relier fin(A), fin$(B) \rightarrow$ final par ϵ
else if $expr = (A^*)$ **then**
  créer un nouvel état initial et final relier initial $\rightarrow A$ et fin$(A) \rightarrow$ final par ϵ relier fin$(A) \rightarrow$ début(A) et initial $\rightarrow$ final par ϵ

Complexité : $O(n)$ en nombre d'opérateurs et symboles de l'expression régulière.

3.1.3 Algorithme de Déterminisation (construction des sous-ensembles)

Principe : Cet algorithme [2] transforme un automate non déterministe (NDFA) en un automate déterministe (DFA) équivalent. Chaque état du DFA correspond à un ensemble d'états du NDFA. Pour chaque symbole, on calcule l'ensemble des états atteignables via les transitions, en incluant les transitions ϵ.

La figure 4 illustre le résultat de cette transformation : toutes les transitions ϵ sont éliminées, et chaque état du DFA représente désormais un ensemble bien défini d'états accessibles du NDFA.

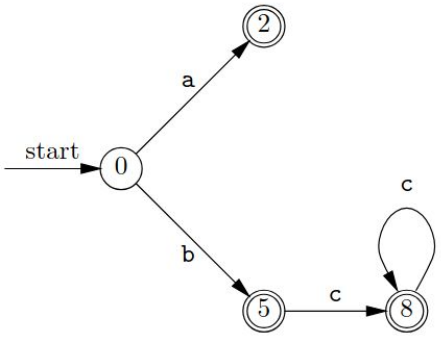


Figure 4: Résultat de la déterminisation d'un automate non déterministe.

Pseudo-code :

Algorithm 2 Déterminisation d'un NDFA

Input: NDFA $N = (Q, \Sigma, \delta, q_0, F)$
Output: DFA équivalent D
état_initial_DFA $\leftarrow$ ϵ-fermeture$(N.q_0)$ ajouter état_initial_DFA à la file des états à traiter **while** *file non vide* **do**
  état_DFA $\leftarrow$ retirer un état de la file **foreach** *symbole* $a \in \Sigma$ **do**
    ensemble $\leftarrow$ ϵ-fermeture(états atteignables depuis état_DFA avec a) **if** *ensemble est nouveau* **then**
      créer un nouvel état dans le DFA ajouter cet état à la file
    ajouter la transition (état_DFA, a, ensemble)

Complexité : $O(2^n)$ dans le pire des cas (chaque sous-ensemble d'états du NDFA peut devenir un état du DFA).

3.1.4 Algorithme de Minimisation de Hopcroft

Principe: Cet algorithme [3] [4] [5] réduit le nombre d'états d'un DFA sans modifier le langage reconnu. Il regroupe les états équivalents (qui mènent aux mêmes classes d'états finaux pour toute entrée possible). La Figure 5 illustre le résultat obtenu après la minimisation : plusieurs états équivalents du DFA initial sont fusionnés, produisant un automate plus compact tout en conservant le même langage.

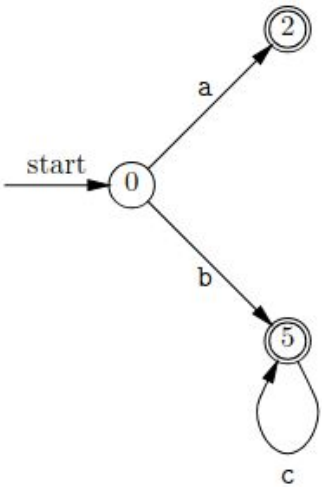


Figure 5: Résultat de la minimisation du DFA selon l'algorithme de Hopcroft.

Pseudo-code:

Algorithm 3 Minimisation d'un DFA

Input: DFA $M = (Q, \Sigma, \delta, q_0, F)$
Output: DFA minimal équivalent
partition $\leftarrow \{F, Q \setminus F\}$ file $\leftarrow \{F, Q \setminus F\}$ **while** *file non vide* **do**
  retirer un ensemble A de la file **foreach** *symbole* $a \in \Sigma$ **do**
    $X \leftarrow \{q \in Q \mid \delta(q, a) \in A\}$ **foreach** *sous-ensemble* Y *dans partition* **do**
      if $X \cap Y \neq \emptyset$ ***et*** $Y \setminus X \neq \emptyset$ **then**
        remplacer Y par $\{X \cap Y, Y \setminus X\}$ dans partition ajouter $X \cap Y$ et $Y \setminus X$ à la file
construire le DFA minimal à partir de la partition finale

Complexité: $O(n \log n)$ — plus rapide que les autres méthodes de minimisation.

3.1.5 Avantages et inconvénients

Avantages :

- Permet de rechercher plusieurs motifs simultanément en une seule passe.
- La recherche se fait en temps linéaire par rapport à la longueur du texte.

Inconvénients :

- La construction de l'automate peut consommer beaucoup de mémoire.
- Un prétraitement est nécessaire avant la recherche.

3.2 Méthode de Knuth-Morris-Pratt

3.2.1 Principe

L'algorithme de Knuth–Morris–Pratt (KMP) [7] permet de rechercher efficacement un motif P dans un texte T sans revenir en arrière dans le texte. Il repose sur la construction d'une *table de décalage* (appelée aussi table de *carryover* ou *préfixe-suffixe*) qui indique, lors d'un échec de comparaison, la plus longue correspondance déjà trouvée entre le début du motif et la partie courante.

Exemple illustratif : Considérons le motif $P =$ `abababc` et le texte $T =$ `abababcabababcababc`. L'algorithme construit d'abord la table de *carryover* (ou `next`) permettant d'indiquer à quelle position du motif reprendre la comparaison après une erreur.

La Table 1 montre la table de `carryover` calculée pour le motif choisi. Chaque case indique, pour une position donnée du motif, la longueur du plus long préfixe-propre qui est aussi un suffixe.

Table 1: Table `carryover` (ou `next`) pour le motif `abababc`.

Position (i)	0	1	2	3	4	5	6
Caractère ($mot[i]$)	a	b	a	b	a	b	c
Valeur `carryover[i]`	-1	0	0	1	2	3	0

3.2.2 Pseudo-code

Algorithm 4 Recherche KMP

Input: texte T, motif P
Output: Positions où P apparaît dans T

$n \leftarrow$ longueur(T) $m \leftarrow$ longueur(P) $CO \leftarrow$ ConstruireCarryover(P) $i \leftarrow 0, j \leftarrow 0$ **while** $i < n$ **do**
  if $j == -1$ ***ou*** $T[i] == P[j]$ **then**
    $i \leftarrow i + 1, j \leftarrow j + 1$ **if** $j == m$ **then**
      afficher("Motif trouvé à", $i - m$) $j \leftarrow CO[j]$
  else
    $j \leftarrow CO[j]$

Algorithm 5 Construction de la table carryover

Input: motif P
Output: table $next$

$next[0] \leftarrow -1$ $i \leftarrow 0, j \leftarrow -1$ **while** $i <$ *longueur(P)* **do**
  if $j == -1$ ***ou*** $P[i] == P[j]$ **then**
    $i \leftarrow i + 1, j \leftarrow j + 1$ $next[i] \leftarrow j$
  else
    $j \leftarrow next[j]$

return $next$

3.2.3 Complexité

La complexité de l'algorithme est $O(n + m)$, soit $O(m)$ pour la construction de la table de prétraitement et $O(n)$ pour la recherche dans le texte (au pire des cas $O(m \times n)$ en implémentation naïve).

3.2.4 Avantages

- Complexité linéaire garantie.
- Applicable sur de grands textes ou flux de données.

3.2.5 Inconvénients

- Moins intuitif que la recherche naïve.
- Ne gère pas les expressions régulières.

4 Partie test : Méthode d'obtention des testbeds

4.1 Source des données et description

Les données utilisées proviennent de la bibliothèque numérique Project Gutenberg (`http://www.gutenberg.org/`), qui offre de nombreux livres au format texte brut. Le fichier principal exploité est le suivant :

- `http://www.gutenberg.org/files/56667/56667-0.txt`

D'autres ouvrages issus de la même base ont également été utilisés afin de varier la taille et la structure des corpus (de quelques centaines de Ko à plusieurs Mo). Cette diversité permet d'évaluer plus équitablement les performances des approches (`egrep, automate et KMP`) sur des textes de complexité différente.

Avant de lancer les expérimentations sur les fichiers, chaque composant du programme (classes Arbre, NDFA, DFA et minimisation) a été soumis à des tests unitaires, situés dans le dossier `src/test`, afin de vérifier leur bon fonctionnement et garantir la fiabilité des résultats.

4.2 Processus d'expérimentation

Pour chaque fichier de test, le processus d'expérimentation s'effectue selon les étapes suivantes :

1. Lecture du fichier texte ligne par ligne.
2. Application de l'algorithme choisi (`egrep`, automate ou KMP) sur chaque ligne du texte, selon le motif recherché.
3. Mesure du temps d'exécution global ou du nombre total de correspondances détectées, selon la nature de l'expérimentation.
4. Stockage des résultats obtenus (temps, nombre de correspondances, nom du fichier, motif, etc.) dans un répertoire dédié nommé `results/`.

Ce processus garantit la reproductibilité des expériences et permet une comparaison équitable entre les différentes méthodes de recherche de motifs.

5 Tests de performance

5.1 Temps d'exécution des différentes méthodes de recherche

Afin d'évaluer les performances des trois approches de recherche: egrep, KMP et automate fini déterministe .Nous avons mené deux séries de tests sur des textes issus du Project Gutenberg. La première série porte sur un seul fichier de grande taille afin d'observer le comportement des méthodes selon la complexité des motifs. La seconde série étudie l'impact de la taille du fichier sur le temps d'exécution pour deux motifs de nature différente (`"Sargon"` et `"and"`).

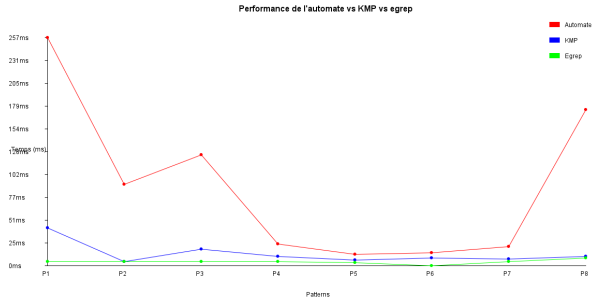


Figure 6: Comparaison des performances pour plusieurs motifs dans un même fichier.

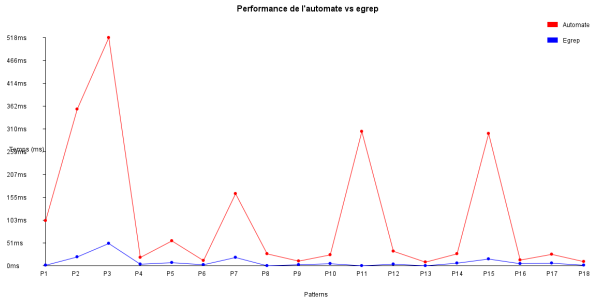


Figure 7: Comparaison egrep–automate sur des motifs complexes

Les figures 6 et 7 comparent les performances sur des motifs simples et complexes (expressions régulières). La figure 6 montre l'évolution du temps d'exécution pour huit motifs simples (`P1` à `P8`) dans un même fichier texte. Egrep reste la méthode la plus rapide grâce à son implémentation optimisée, tandis que KMP offre des performances stables et régulières. La méthode par automate est plus lente, en raison du coût de construction et du parcours des états.
La figure 7 illustre le cas des motifs complexes, où seul egrep est comparable à la méthode par automate, KMP ne s'appliquant pas aux expressions régulières. Les résultats confirment l'avantage d'egrep, tandis que l'automate voit son temps d'exécution croître avec la complexité des motifs et la taille de l'automate généré.
Pour évaluer l'influence de la taille du corpus, nous avons répété les expériences sur trois fichiers de tailles différentes (*petit*, *moyen* et *grand*) et sur deux motifs contrastés : un motif rare (`"Sargon"`) et un motif fréquent (`"and"`).

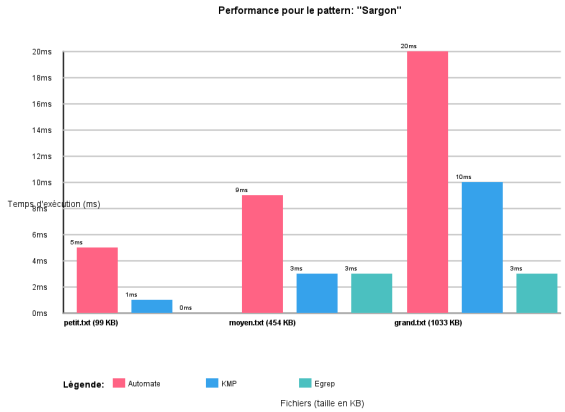


Figure 8: Temps d'exécution pour le motif `"Sargon"`.

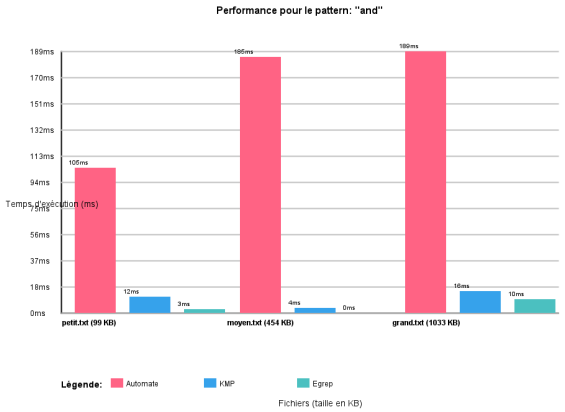


Figure 9: Temps d'exécution pour le motif `"and"`.

Les graphiques des figures 8 et 9 montrent que le temps d'exécution augmente proportionnellement à la taille du fichier pour toutes les méthodes. Cependant, les écarts deviennent plus marqués selon la fréquence du motif : pour le motif rare `"Sargon"`, les trois approches restent relativement proches, tandis que pour le motif fréquent `"and"`, la méthode par automate devient nettement plus lente, à cause du nombre élevé de transitions effectuées. À l'inverse, egrep et KMP conservent des temps faibles et stables, confirmant leur efficacité pour des recherches répétées dans de grands corpus.

5.2 Validation de la fiabilité des résultats

Afin de garantir la fiabilité de nos mesures de temps d'exécution, la première étape a consisté à valider la justesse des résultats retournés par les trois approches de recherche — l'algorithme KMP, l'Automate Fini Déterministe et l'utilitaire de référence Egrep.

Nous avons comparé le nombre d'occurrences trouvées par chaque méthode pour dix motifs différents au sein du fichier de test du Project Gutenberg (`56667-0.txt`).

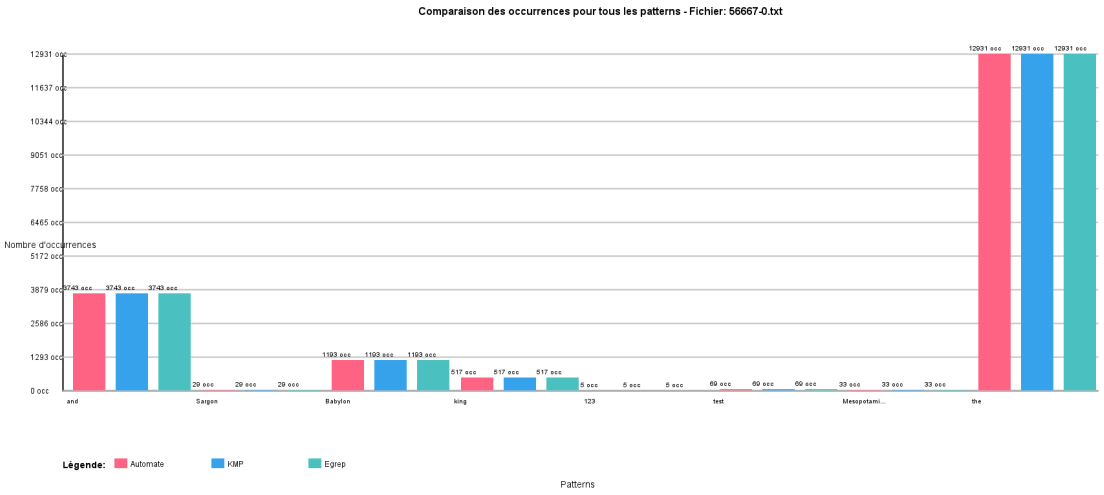


Figure 10: Comparaison du nombre d'occurrences trouvées pour différents motifs par les trois méthodes dans le fichier 56667-0.txt.

L'observation de la figure 10 est formelle : pour l'ensemble des motifs testés, des mots très fréquents comme `"the"` (avec 12931 occurrences) aux mots rares (`Mesopotamia` ou `Babylon`), les trois méthodes retournent un nombre d'occurrences strictement identique.

Cette parfaite cohérence des résultats confirme la fidélité d'implémentation des algorithmes KMP et Automate Fini par rapport à l'outil Egrep. Ayant validé la justesse et la robustesse de nos algorithmes.

6 Discussion sur les résultats des tests de performance

Les expériences illustrées par les figures 6, 7, 8, 9 et 10 évaluent les performances des trois approches (egrep, KMP et Automate Fini Déterministe) selon la complexité des motifs, la taille des fichiers et la fréquence d'occurrence.

Les figures 6 et 7 montrent que egrep est la plus rapide,KMP reste stable, et l'Automate plus coûteux à cause de la construction et du parcours des états. Les figures 8 et 9 confirment que la taille du fichier et la fréquence du motif influencent fortement les performances : pour le motif rare `"Sargon"`, les écarts sont faibles, tandis que pour `"and"`, l'Automate devient nettement plus lent. La figure 10 valide la justesse des implémentations : toutes les méthodes renvoient le même nombre d'occurrences, qu'il s'agisse de mots fréquents ou rares.

En résumé, egrep demeure la plus rapide pour les recherches simples sur de grands textes, KMP offre des performances stables et équilibrées, tandis que l'Automate, plus flexible, reste plus coûteux pour les motifs complexes ou fréquents.

7 Conclusion et perspectives

Ce travail a porté sur l'étude et la comparaison de différentes méthodes de recherche de motifs textuels : egrep, l'algorithme Knuth-Morris-Pratt (KMP) et la méthode par automate fini déterministe. L'objectif était de comprendre leurs principes, leurs performances et leurs domaines d'application.

Les expérimentations ont montré que egrep reste la plus rapide grâce à son implémentation optimisée, tandis que KMP offre un bon compromis entre efficacité et simplicité. La méthode par automate, bien que plus lente, se distingue par sa flexibilité et sa capacité à traiter des expressions régulières complexes.

Dans l'ensemble, cette étude met en évidence le compromis entre vitesse, généricité et coût de calcul, et souligne l'importance du choix de l'approche selon le contexte d'utilisation et la nature du motif recherché.

7.1 Perspectives

Optimiser la construction des automates pour réduire le coût en temps et mémoire, adapter les méthodes à des fichiers volumineux ou des flux de données en temps réel, étendre la recherche à des motifs multiples ou imbriqués, et explorer la parallélisation pour tirer parti des architectures multi-cœurs. Ces axes permettront de concilier performance et flexibilité pour les moteurs de recherche hors ligne.

References

[1] Thompson, K. (1968). *Regular expression search algorithm*. Communications of the ACM, 11(6), 419–422.
https://dl.acm.org/doi/10.1145/363347.363387

[2] Baburin, I., Cotterell, R. (2024). *A Close Analysis of the Subset Construction*. arXiv preprint arXiv:2407.09891.
https://arxiv.org/abs/2407.09891

[3] Hopcroft, J. E. (1971). *An n log n algorithm for minimizing states in a finite automaton*. Technical Report, Stanford University.
https://dl.acm.org/doi/10.1145/321250.321253

[4] Knuutila, T. (2001). *Re-describing an algorithm by Hopcroft*. Department of Computer Science, University of Turku.
https://www.cs.cmu.edu/~cdm/resources/Knuutila2001.pdf

[5] Berstel, J., Boasson, L., Carton, O., Fagnot, I. (2010). *Minimization of Automata*. arXiv preprint arXiv:1010.5318.
https://arxiv.org/abs/1010.5318

[6] 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*.
http://infolab.stanford.edu/~ullman/focs/ch10.pdf

[7] Knuth, D. E., Morris, J. H., Pratt, V. R. (1977). *Fast Pattern Matching in Strings*. SIAM Journal on Computing, 6(2), 323–350.
https://doi.org/10.1137/0206024