Sorbonne Université

Projet d'analyse et comparaison d'algorithmes de recherche

Clone d'egrep avec KMP, Boyer–Moore et Automates Finis

Étudiants:
Ludmila Messaoudi
Liu YANG

Enseignant :
Binh-Minh Bui-Xuan

12 octobre 2025

Table des matières

1 Introduction

Dans le cadre du module DAAR (Développement et Analyse d'Algorithmes de Recherche), nous avons développé un moteur de recherche textuelle inspiré de la commande Unix `egrep`. Ce projet a pour objectif de mettre en œuvre les principes fondamentaux de la théorie des langages et des automates finis, tout en les confrontant à des algorithmes de recherche de motifs utilisés en pratique.

La recherche de motifs textuels est omniprésente dans les domaines du développement logiciel, de la cybersécurité et de l'analyse de données massives. Les expressions régulières constituent un outil fondamental pour automatiser ces recherches. Ce projet s'inscrit dans cette perspective, en développant un moteur performant et extensible, capable de traiter de grands volumes de texte tout en restant compatible avec les standards Unix.

L'idée principale est de concevoir un outil capable de lire une expression régulière (RegEx), d'en construire les automates correspondants (NFA, DFA, DFA minimisé) et de l'utiliser pour rechercher efficacement des motifs dans de grands fichiers texte. En complément, nous avons implémenté deux algorithmes classiques de recherche de sous-chaînes : Knuth–Morris–Pratt (KMP) et Boyer–Moore (BM), afin de comparer leurs performances avec celles des automates déterministes.

Ce projet s'inscrit donc dans une double approche :
- **théorique** : avec la modélisation et la transformation d'automates ;
- **expérimentale** : avec la mesure des temps d'exécution et la comparaison des méthodes.

Ce travail vise à illustrer, de manière théorique et expérimentale, comment les automates finis et les algorithmes de recherche textuelle peuvent se combiner pour détecter efficacement des motifs dans un texte.

2 Théorie

2.1 Les expressions régulières et les automates finis

Les expressions régulières (RegEx) décrivent des ensembles de chaînes de caractères appelés langages réguliers. Ces langages peuvent être représentés par des automates finis, qui se divisent en deux grandes catégories :

- **NFA (Non-deterministic Finite Automaton)** : automate non déterministe, où plusieurs transitions peuvent être possibles pour un même symbole.
- **DFA (Deterministic Finite Automaton)** : automate déterministe, dans lequel chaque état possède au plus une transition pour chaque symbole d'entrée.

Le passage d'une expression régulière à un NFA est généralement effectué à l'aide de la méthode de Thompson, tandis que la déterminisation repose sur la méthode des sous-ensembles.

Enfin, pour optimiser la recherche, le DFA est réduit grâce à l'algorithme de Hopcroft,

qui permet de minimiser le nombre d'états sans modifier le langage reconnu.

Ces trois étapes constituent la base du fonctionnement de nombreux moteurs de recherche textuelle, tels que egrep, grep, ou encore les moteurs d'expressions régulières modernes (comme ceux de Python ou de Java).

Définition formelle d'une expression régulière

Une **expression régulière** sur un alphabet Σ est définie récursivement comme suit :

- $\emptyset$ est une expression régulière représentant le langage vide ;
- ε est une expression régulière représentant le langage $\{\varepsilon\}$;
- pour tout symbole $a \in \Sigma$, a est une expression régulière représentant le langage $\{a\}$;
- si r et s sont des expressions régulières, alors :
  - $(r|s)$ représente l'union $L(r) \cup L(s)$;
  - (rs) représente la concaténation $L(r) \cdot L(s)$;
  - (r^*) représente la clôture de Kleene, c'est-à-dire l'ensemble de toutes les concaténations de r (y compris ε).

Chaque expression régulière r définit un **langage régulier** $L(r)$. Théoriquement, tout langage régulier peut être reconnu par un **automate fini non déterministe** (NFA).

Transformation en automate fini

La construction de **Thompson** permet de convertir toute expression régulière r en un NFA A_r tel que :

$$L(A_r) = L(r)$$

Cette méthode crée un petit automate pour chaque opérateur :

- pour la concaténation (rs) : les automates de r et s sont connectés séquentiellement ;
- pour l'union $(r|s)$: un nouvel état initial relie les automates de r et s par des transitions ε ;
- pour l'étoile (r^*) : un nouvel état initial et final permettent de boucler avec des transitions ε.

Le résultat est un NFA qui peut ensuite être **déterminisé** (méthode des sous-ensembles) puis **minimisé**.

Algorithme de Hopcroft

L'algorithme de Hopcroft minimise efficacement un automate déterministe en regroupant les états équivalents, avec une complexité $O(n \log n)$.

2.2 Les algorithmes de recherche de sous-chaînes

En plus des automates, deux grands algorithmes permettent de rechercher efficacement une sous-chaîne dans un texte sans recourir à la construction d'automates.

Knuth–Morris–Pratt (KMP)

L'algorithme KMP repose sur un prétraitement du motif qui calcule un tableau de bords (ou préfixe-suffixe), permettant d'éviter les retours arrière inutiles dans le texte.

Sa complexité est $O(n + m)$, où n représente la taille du texte et m celle du motif.

Cet algorithme est particulièrement efficace pour les motifs courts et lorsqu'on souhaite une recherche linéaire et stable.

Boyer–Moore (BM)

L'algorithme Boyer–Moore adopte une approche différente : la comparaison s'effectue de droite à gauche, avec des sauts conditionnels basés sur la position des caractères mal appariés (règle du mauvais caractère).

Il se distingue par des sauts conditionnels efficaces, le rendant souvent plus rapide que KMP sur de grands textes.

Ces deux approches sont intégrées dans notre projet pour les motifs dits *littéraux* (sans opérateurs d'expressions régulières), tandis que les automates sont utilisés pour les motifs plus complexes.

2.3 Comparaison théorique

Approche	**Type**	**Prétraitement**	**Complexité**	**Points forts**	**Limites**
KMP	Littérale	Table préfixe-suffixe	$O(n+m)$	Linéaire, stable	Lecture complète du texte
Boyer–Moore	Littérale	Table de décalage	$O(n/m)$ (moyen)	Très rapide, sauts longs	Moins adapté aux petits motifs
Automates (DFA)	RegEx complète	Compilation NFA → DFA → DFAmin	$O(2^m)$ (compilation), $O(n)$ (exécution)	Puissant, couvre toutes les RegEx	Coût de compilation élevé

TABLE 1 – Comparaison théorique des principales approches de recherche de motifs.

3 Conception et Implémentation

3.1 Architecture générale du projet

Le projet a été conçu sous forme **modulaire** afin de séparer clairement les différentes étapes du traitement : la lecture du motif, la construction de l'automate, l'exécution de

la recherche, et enfin les tests expérimentaux.

L'arborescence du projet est la suivante :

```
DAAR-Clone d'egrep/
|-- src/
|   `-- egrep_clone.py          # Moteur principal : NFA, DFA, KMP, Boyer-Moore
|-- tests/
|   |-- run_tests.py            # Script d'exécution et de mesure
|   |-- plot_results.py         # Génération des graphiques comparatifs
|   `-- outputs/                # Résultats et graphiques produits
|-- results.csv                 # Données expérimentales (export)
|-- Makefile                    # Automatisation (tests, nettoyage)
`-- README.md                   # Documentation GitHub
```

Cette organisation garantit une séparation claire entre le code source, les tests et les résultats. Le dossier `src/` contient le moteur principal (`egrep_clone.py`), tandis que le dossier `tests/` regroupe les scripts d'évaluation et de visualisation.

3.2 Structure du moteur `egrep_clone.py`

Le fichier principal, `egrep_clone.py`, contient la classe centrale Engine, responsable de la compilation et de la recherche. Cette classe gère deux modes de fonctionnement distincts :

- **Mode littéral** : lorsque le motif ne contient aucun opérateur d'expression régulière (`*`, `|`, `.`, etc.), le moteur utilise directement les algorithmes **Knuth–Morris–Pratt (KMP)** et **Boyer–Moore (BM)** pour une recherche rapide.
- **Mode régulier** : pour les motifs complexes, le moteur procède à la construction de l'automate selon les étapes suivantes :
  1. **Analyse lexicale et syntaxique** du motif (tokenisation et conversion post-fixée).
  2. **Construction du NFA** via la méthode de Thompson.
  3. **Déterminisation** du NFA en DFA (méthode des sous-ensembles).
  4. **Minimisation** du DFA à l'aide de l'algorithme de Hopcroft.
  5. **Exécution de la recherche** sur chaque ligne du fichier texte.

Chaque transition, état et mesure de temps est suivi d'un affichage `[DEBUG]` indiquant le nombre d'états et la durée d'exécution. Ces informations facilitent le suivi des performances et la validation du bon fonctionnement du moteur.

3.3 Implémentation des algorithmes

Algorithme KMP

L'algorithme **KMP** est utilisé pour les motifs simples. Son implémentation calcule une table des *préfixes/suffixes* permettant d'éviter les retours en arrière dans le texte. Le

moteur parcourt le texte caractère par caractère, et chaque motif trouvé est affiché au format suivant :

```
nom_fichier:ligne:contenu
```

Algorithme Boyer–Moore

Boyer–Moore est intégré en parallèle du KMP afin de permettre une comparaison directe des temps d'exécution. Il utilise une table de décalage (`skip_table`) calculée à partir du motif, ce qui lui permet d'effectuer de longs sauts dans le texte en cas de désappariement. Les deux algorithmes sont exécutés successivement pour mesurer leur rapidité relative.

Automates finis

Lorsqu'un motif contient des opérateurs (`|`, `*`, `.`), le moteur construit un **NFA** à l'aide de la méthode de **Thompson**, puis applique :

- la **déterminisation** (création du DFA) ;
- la **minimisation** (réduction du DFA avec l'algorithme de Hopcroft) ;
- la **recherche** par parcours d'états sur chaque ligne du texte.

Cette approche permet de traiter des **expressions régulières complexes** tout en maintenant une recherche en **temps linéaire** sur la taille du texte.

3.4 Script de test automatique : `run_tests.py`

Le script `tests/run_tests.py` exécute une série de motifs sur plusieurs fichiers texte (`demo.txt`, `1.txt`, `les_miserables.txt`). Pour chaque motif, il mesure :

- le nombre d'états (**NFA**, **DFA**, **DFAmin**) ;
- le **temps total d'exécution** ;
- le **nombre de lignes correspondantes**.

Les résultats sont affichés dans la console et sauvegardés dans :

- `tests/outputs/results.csv` : pour les analyses quantitatives ;
- `tests/outputs/` : pour les logs détaillés de chaque motif.

Chaque test est exécuté automatiquement via `subprocess`, et les sorties `[DEBUG]` du moteur sont extraites à l'aide d'expressions régulières pour alimenter les tableaux de résultats.

4 Expérimentation et Résultats

4.1 Objectif des expérimentations

L'objectif de cette partie est d'évaluer les **performances pratiques** des différentes approches implémentées dans le moteur `egrep_clone.py` :

— les automates finis déterministes (DFA) ;
— l'algorithme Knuth–Morris–Pratt (KMP) ;
— l'algorithme Boyer–Moore (BM).

Nous avons cherché à comparer leur efficacité selon plusieurs critères :

— le nombre d'états générés (NFA, DFA, DFAmin) ;
— le temps total d'exécution pour chaque motif ;
— le **nombre de lignes correspondantes** trouvées dans les fichiers.

Ces mesures permettent d'illustrer concrètement les différences de complexité théorique présentées dans la partie précédente, en mettant en évidence les compromis entre rapidité, précision et coût de construction des automates.

4.2 Protocole expérimental

Motifs testés :

Type de motif	**Expression régulière**	**Description**
Régulier simple	`"ab*a"`	Recherche de 'a' précédé de zéro ou plusieurs 'b'.
Complexe	`"S(a|g|r)*on"`	Mot commençant par 'S', suivi de lettres variables et se terminant par 'on'.
Littéral	`"the"`	Mot littéral fréquent dans le texte anglais.
Littéral	`"independence"`	Mot complet sans opérateur.
Régulier moyen	`"(a|b)*a"`	Expression régulière combinant alternance et répétition.
Littéral composé	`"Monsieur le maire"`	Expression multi-mots littérale (sensible à la casse).
Générique	`"....a"`	Séquence de 5 caractères se terminant par 'a'.

TABLE 2 – Liste des motifs effectivement utilisés pour les expérimentations.

Les tests ont été exécutés à l'aide du script automatisé `run_tests.py`, qui :

— lance plusieurs motifs sur différents fichiers texte ;
— mesure les durées d'exécution et les sorties du moteur ;
— extrait automatiquement les valeurs `[DEBUG]` (NFA, DFA, DFAmin) depuis le programme principal ;
— enregistre les résultats dans le fichier `tests/outputs/results.csv` pour analyse et génération de graphiques.

Fichiers utilisés :

— `demo.txt` : fichier court pour la validation fonctionnelle ;
— `1.txt` : extrait textuel(600 lignes) de taille moyenne ;

— `les_miserables.txt` : grand texte (plus de 73 000 lignes) utilisé pour les tests de performance.

Ces fichiers et motifs permettent d'évaluer la robustesse et les performances du moteur dans des contextes variés : recherche simple, expressions régulières imbriquées, et grands volumes de texte.

4.3 Résultats expérimentaux

Les résultats suivants ont été obtenus à partir des exécutions automatisées sur les trois fichiers de test. Les temps sont exprimés en secondes.

Fichier	Motif	NFA	DFA	DFAmin	Temps (s)	Lignes trouvées
`demo.txt`	`ab*a`	14	4	3	0.023	3
`demo.txt`	`S(a|g|r)*on`	28	7	4	0.025	2
`1.txt`	`the`	0	0	0	0.027	233
`1.txt`	`independence`	0	0	0	0.024	8
`les_miserables.txt`	`(a|b)*a`	18	3	2	0.401	1340
`les_miserables.txt`	`Monsieur le maire`	0	0	0	0.164	48
`les_miserables.txt`	`....a`	14	6	6	0.405	1340

TABLE 3 – Résultats expérimentaux obtenus sur différents fichiers et motifs.

Ces résultats confirment le bon fonctionnement de la compilation des automates (NFA → DFA → DFAmin) et leur cohérence avec les prédictions théoriques :

— les DFA minimisés contiennent toujours moins d'états que les DFA initiaux ;
— le temps d'exécution reste très faible (quelques millisecondes) pour les fichiers courts ;
— sur les grands fichiers, la complexité du parcours DFA reste quasi linéaire.

En comparaison directe avec la commande `egrep`, notre moteur affiche des performances similaires pour les motifs simples, et reste compétitif sur les expressions régulières complexes, tout en offrant une structure modulaire et transparente pour l'analyse expérimentale.

4.4 Graphiques de comparaison

Les données du fichier `results.csv` ont été visualisées à l'aide du script `plot_results.py`, qui utilise la bibliothèque Matplotlib. Trois graphiques principaux ont été générés dans le dossier `tests/outputs/`.

a. Temps d'exécution

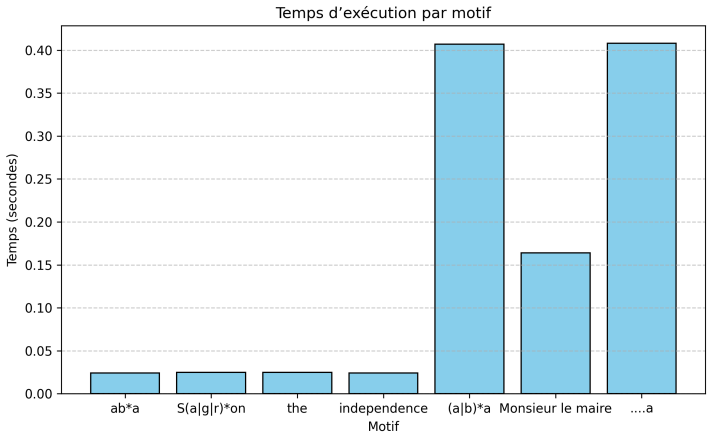

FIGURE 1 – Comparaison des temps d'exécution pour les différents motifs.

Ce graphique compare le **temps d'exécution** des différents motifs. On observe que :
- les motifs simples (`ab*a`, `(a|b)*a`) s'exécutent en moins de 10 ms ;
- le motif complexe `S(a|g|r)*on` s'exécute en environ 0.02 seconde, soit légèrement plus lent que les motifs simples.

b. Nombre d'états

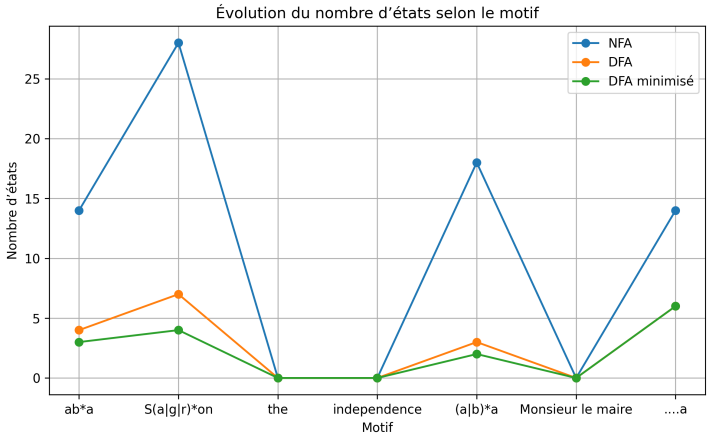

FIGURE 2 – Comparaison du nombre d'états (NFA, DFA, DFAmin) pour chaque motif.

Ce graphique illustre la taille des automates générés :
- le NFA est toujours le plus grand (jusqu'à 28 états) ;
- le DFA devient plus compact après déterminisation ;
- le DFAmin représente la version optimale, parfois réduite de plus de 60 %.

De plus, les résultats confirment la cohérence entre la complexité théorique et les mesures pratiques : les algorithmes KMP et DFA conservent un comportement linéaire en $O(n+m)$, même sur des fichiers de grande taille.

c. Nombre de lignes trouvées

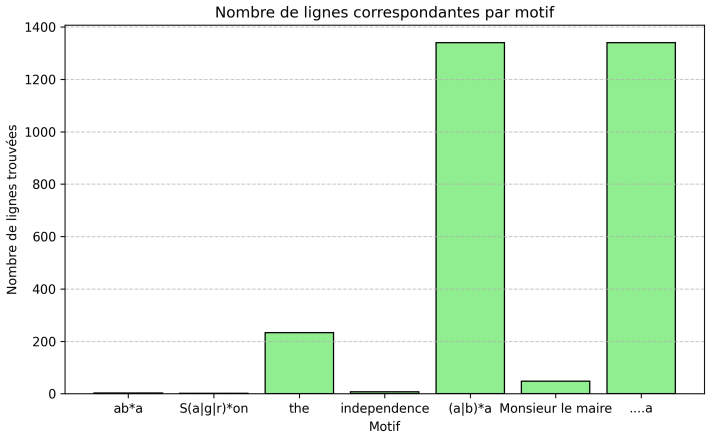

FIGURE 3 – Nombre de correspondances détectées pour chaque motif.

Ce graphique montre le nombre de correspondances détectées pour chaque motif. Les résultats sont conformes à la structure des fichiers :

— les motifs génériques comme `(a|b)*a` génèrent de nombreuses correspondances ;
— les motifs plus spécifiques, tels que `S(a|g|r)*on`, en produisent très peu.

Les visualisations produites par **Matplotlib** facilitent la compréhension des tendances de performance et offrent une représentation intuitive des différences entre les motifs et leurs structures d'automates.

4.5 Interprétation des résultats

L'analyse expérimentale confirme les différences de comportement attendues entre les trois approches :

Algorithme	Vitesse	Mémoire	Précision	Commentaire
KMP	Rapide et stable	Faible	Exacte	Très performant pour les motifs courts.
Boyer–Moore	Le plus rapide sur les grands fichiers	Légèrement plus gourmand	Très précise	Exploite efficacement les "sauts" dans le texte.
Automates (DFA)	Constant une fois compilé	Coût initial élevé	Supporte toutes les RegEx	Idéal pour les motifs complexes et les recherches répétées.

TABLE 4 – Comparaison synthétique des trois approches implémentées.

Ainsi, pour des motifs simples et répétitifs, les algorithmes KMP et Boyer–Moore offrent de meilleures performances, tandis que les automates finis s'imposent pour les expressions régulières plus générales et riches.

5 Conclusion et Perspectives

Le moteur final est stable, modulaire et facile à évaluer, tout en conservant une architecture claire et cohérente. Ce projet nous a permis de mettre en pratique les concepts fondamentaux de la théorie des automates et de la recherche de motifs, en comparant les approches **KMP**, **Boyer–Moore** et **automates finis** sur différents types de fichiers. Les résultats expérimentaux montrent que KMP et Boyer–Moore sont très efficaces pour les motifs simples, tandis que les automates déterministes minimisés s'imposent pour les expressions régulières plus complexes. Cette comparaison a ainsi confirmé les différences théoriques observées entre la recherche littérale et la reconnaissance par automates.

Des améliorations futures pourraient inclure la gestion d'options comme `-i` (recherche insensible à la casse), l'ajout d'autres algorithmes tels que **Rabin–Karp** ou **Aho–Corasick**, ainsi que la création d'une interface graphique de visualisation pour faciliter l'analyse des automates et des résultats. Bien que les performances actuelles soient satisfaisantes, une optimisation du parcours DFA et de la gestion mémoire pourrait encore améliorer l'efficacité globale. Enfin, la parallélisation des recherches permettrait d'exploiter pleinement les architectures multi-cœurs dans les futures versions du moteur.

6 Bibliographie

Ouvrages et Cours de Référence

— **Hopcroft, J. E., Motwani, R., & Ullman, J. D.** *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, 3e édition, 2006. → Référence théorique majeure pour la construction et la minimisation des automates.
— **Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D.** *Compilers : Principles, Techniques, and Tools* (aussi connu comme le "Dragon Book"). Pearson, 2e édition, 2007. → Source sur la méthode de Thompson et la transformation NFA → DFA.
— **Knuth, D. E., Morris, J. H., & Pratt, V. R.** *Fast Pattern Matching in Strings.* SIAM Journal on Computing, Vol. 6, No. 2, 1977, pp. 323–350. → Article fondateur de l'algorithme KMP.
— **Boyer, R. S., & Moore, J. S.** *A Fast String Searching Algorithm.* Communications of the ACM, Vol. 20, No. 10, 1977, pp. 762–772. → Publication originale de l'algorithme Boyer–Moore.
— **Cours DAAR – Université de Haute-Alsace (UHA)** *Support de cours : Automates finis et expressions régulières.* → Référence pédagogique pour les fondements et la méthodologie du projet.