Sorbonne Université

Automate - Clone de egrep avec support partiel des ERE

Binôme :
Floria LIM
Julien FANG

Enseignants :
Binh-Minh Bui-Xuan
Alfred Deivassagayame
Arthur Escriou
Guillaume Hivert
Suxue Li

6 octobre 2024

Table des matières

1 Introduction

La recherche de motifs dans des fichiers textuels est un problème fondamental en informatique, avec de nombreuses applications allant de la recherche d'information à l'analyse de données. Pour répondre à ce besoin, des outils comme la commande `egrep` permettent d'utiliser des expressions régulières pour détecter des motifs dans de grandes quantités de texte. Cependant, la complexité et la diversité des expressions régulières rendent cette tâche particulièrement délicate, notamment en ce qui concerne la norme Extended Regular Expressions (ERE).

Dans le cadre de ce projet, nous visons à développer un clone de la commande `egrep`, avec un support partiel des Extended Regular Expressions (ERE). En effet, nous avons limité l'ensemble des opérations aux parenthèses, à l'alternative, à la concaténation, à l'étoile de Kleene, au caractère universel, et aux lettres ASCII.

2 Définition du problème

Le problème traité dans ce projet est la recherche efficace de motifs à l'aide d'expressions régulières conformes à une partie de la norme Extended Regular Expressions (ERE). Cette recherche doit être réalisée dans des fichiers textuels, composés de plusieurs lignes, en appliquant le motif sur chaque ligne de manière indépendante.
La principale difficulté réside dans la gestion de la diversité des motifs décrits par les expressions régulières. Le traitement de ces motifs nécessite l'utilisation de techniques algorithmiques spécifiques pour assurer une détection précise et rapide des motifs dans de grands volumes de texte.

Dans ce contexte, deux stratégies principales sont envisagées :

1. **Utilisation d'automates déterministes (DFA) minimisés :** Lorsque le motif est complexe et ne se réduit pas à une simple concaténation de caractères, il peut être efficace de convertir l'expression régulière en un automate fini. Cette conversion se fait en plusieurs étapes, passant d'abord par la création d'un automate non-déterministe, suivi de sa transformation en automate déterministe, puis se termine par sa minimisation, réduisant le nombre d'états et optimisant ainsi le processus de reconnaissance du motif dans les lignes de texte.
2. **Utilisation de l'algorithme de Knuth-Morris-Pratt (KMP) :** Si le motif est une simple suite de concaténations, l'algorithme KMP peut être utilisé. Il s'agit d'une méthode plus efficace pour ce type de motifs, car elle permet de réaliser la recherche en temps linéaire par rapport à la longueur du texte.

L'objectif du projet est de comparer ces deux approches en termes de performance et d'efficacité, afin d'identifier la méthode la plus appropriée selon le type de motif et la taille du fichier textuel.

3 Structure de données utilisée

Dans le cadre de la recherche de motifs textuels, la sélection et l'utilisation des structures de données adéquates jouent un rôle fondamental pour garantir l'efficacité des algorithmes implémentés. En effet, les performances en termes de temps et de mémoire dépendent largement des structures choisies.

Dans ce projet, plusieurs structures de données ont été employées en fonction des algorithmes étudiés :

- **Arbre RegEx** :
  - **root** : Un entier représentant le code ASCII du caractère contenu dans le noeud.
  - **subTrees** : Une liste d'arbres RegEx représentant les fils du noeud.

- **Automate fini non-déterministe** :
  - **counterState** : Un compteur static pour donner un identifiant unique à chaque nouvel état.
  - **startState** : Un entier représentant l'identifiant de l'état initial.
  - **finalState** : Un entier représentant l'identifiant de l'état final.
  - **transitions** : Un tableau à deux dimensions d'entier, les lignes correspondent aux différents états de l'automate et les colonnes correspondent aux différents symboles. Le contenu indique l'état suivant auquel l'automate doit se déplacer lorsqu'il se trouve dans l'état correspondant à la ligne et qu'il lit le caractère correspondant à la colonne. Si une transition n'est pas possible pour une combinaison donnée d'état et de caractère, la case contient -1 pour indiquer l'absence de transition.
  - **epsilonTransitions** : Un tableau de listes d'entier, où chaque liste contient les états accessibles par une transition epsilon à partir d'un état donné.
  - **symbols** : Une liste d'entiers représentant les différents symboles utilisés dans les transitions de l'automate.

- **Automate fini déterministe** :
  - **stateId** : Un compteur static pour donner un identifiant unique à chaque nouvel état.
  - **startState** : Un entier représentant l'identifiant de l'état initial.
  - **finalStates** : Une liste d'entier correspondant à l'ensemble des identifiants des états finaux.
  - **transitions** : Une HashMap où la clé est un entier représentant un état source et la valeur est une autre HashMap. Dans cette seconde HashMap, la clé est un entier représentant un symbole d'entrée, et la valeur est un entier représentant l'état cible vers lequel l'automate se déplace lorsqu'il lit ce symbole depuis l'état source.
  - **symbols** : Une liste d'entiers, correspondant aux symboles utilisés dans les transitions de l'automate.

- **Algorithme KMP** :
  - **retenue** : Un tableau d'entier représentant les retenues.

Note : Pour traiter le caractère universel (dot), nous avons créé un nouveau symbole spécial attribué à l'identifiant 256, car la plage des caractères ASCII s'étend de 0 à 255. Ce symbole représente l'ensemble des caractères ASCII possibles.

4 Algorithmes de recherche de motifs connus dans la littérature

La recherche de motifs dans des fichiers textuels a donné lieu à plusieurs algorithmes bien établis, chacun ayant ses propres caractéristiques, avantages et inconvénients. Dans cette section, nous allons fournir une présentation et une analyse détaillés des structures et algorithmes que nous avons utilisés pour le recherche de motifs, ainsi que d'autres algorithmes significatifs qui jouent un rôle crucial dans ce contexte.

4.1 Aho-Ullman

Aho-Ullman propose une méthode fondamentale pour transformer une expression régulière d'abord en un automate fini non déterministe (NDFA) avec transitions epsilon, puis en un automate fini déterministe (DFA). Cette méthode est décrite en détail dans le chapitre 10 du livre de Aho-Ullman [1].

4.1.1 Transformation en automate avec epsilon transition

L'algorithme de Aho-Ullman pour la construction d'un automate fini non déterministe (NDFA) avec transitions epsilon (ϵ) passe par les étapes suivantes :

- **Décomposition en arbre de syntaxe** : L'expression régulière est décomposée en une structure d'arbre, où chaque noeud représente soit une opération (la concaténation, l'union, ou l'étoile de Kleene), soit une lettre ou un symbole du motif.
- **Transformation de l'Arbre en NDFA** : À chaque noeud de l'arbre, un NDFA est créé selon le type d'opération qu'il représente :
  - **Concaténation (AB)** : On crée un nouvel état initial qui a une transition ϵ vers l'état initial de A, puis on construit de l'état final de A une transition ϵ qui mène vers l'état initial de B. L'état final de B devient l'état final du NDFA résultant.
  - **Alternation (A|B)** : Pour l'alternation, on crée un nouvel état initial avec des transitions ϵ vers les états initiaux de A et de B, ainsi qu'un nouvel état final qui reçoit des transitions ϵ des états finaux de A et de B.
  - **Etoile (A*)** : Pour l'opérateur étoile, on crée un nouvel état initial et un nouvel état final. L'état initial a une transition ϵ vers l'état initial de A et une autre transition ϵ vers le nouvel état final. De plus, on ajoute deux transitions ϵ à l'état final de A : l'une vers son état initial et l'autre vers le nouvel état final.

Cet automate permet plusieurs transitions possibles pour un même symbole depuis un état donné, ainsi que des transitions via des transitions epsilon (ϵ) qui permettent de passer d'un état à un autre sans consommer de symbole.

1. http ://infolab.stanford.edu/ ullman/focs/ch10.pdf

Cet algorithme présente une complexité de $O(n)$, où n est la longueur de l'expression régulière. En effet, chaque symbole et opération de l'expression est traité une seule fois lors de la construction de l'arbre syntaxique.

4.1.2 Transformation en automate fini déterministe

La prochaine étape de l'algorithme de Aho-Ullman est la conversion de l'automate fini non déterministe (NDFA) en un automate fini déterministe (DFA). Cette conversion permet d'éliminer l'ambiguïté des transitions multiples pour un même symbole dans un état donné, garantissant ainsi un parcours déterministe à travers l'automate. Le processus repose sur la méthode des sous-ensembles d'états : chaque état du DFA correspond à un ensemble d'états du NDFA, représentant ainsi toutes les configurations possibles du NDFA après la lecture d'un symbole donné.

Cette conversion se fait par les étapes principales suivantes :

- **États initiaux et transitions** : La conversion d'un NDFA en DFA consiste à regrouper les états du NDFA en sous-ensembles. Chaque sous-ensemble d'états dans le NDFA correspond à un état unique dans le DFA. Ainsi, pour chaque symbole, le DFA détermine un seul état de destination à partir de chaque sous-ensemble.
- **Gestion des transitions epsilon** : Les transitions epsilon, qui permettent de passer d'un état à un autre sans consommer de symbole dans le DFA, sont éliminées lors de la conversion. Chaque étape consomme un symbole, garantissant ainsi qu'un seul état est atteint à chaque étape du parcours.
- **Table de transition déterministe** : Après avoir déterminé tous les sous-ensembles d'états, une table de transition est construite. Cette table définit une transition unique pour chaque symbole lu à partir d'un état donné. Cela garantit que le DFA est entièrement déterministe et qu'il suit un seul chemin pour chaque mot d'entrée.

La complexité de la conversion d'un NDFA en DFA est généralement de $O(2^n)$, où n est le nombre d'états dans le NDFA. Cette complexité exponentielle résulte du fait que chaque sous-ensemble d'états du NDFA peut potentiellement être un état unique dans le DFA. Ainsi, dans le pire des cas, le nombre d'états dans le DFA peut croître de manière exponentielle par rapport à ceux du NDFA.

4.2 Hopcroft : minimisation de l'automate

L'automate fini minimal contient le moins d'états possibles tout en conservant sa capacité à reconnaître le même motif que l'automate d'origine. La minimisation consiste à identifier et fusionner les états équivalents, c'est-à-dire ceux qui réagissent de manière identique pour toutes les chaînes d'entrée possibles. L'algorithme de Hopcroft permet d'obtenir celà en raffinant efficacement les partitions d'états à l'aide d'une approche de partitionnement. En itérant sur les états et leurs transitions, il divise les états non équivalents jusqu'à ce que tous les états soient correctement regroupés.

La méthode de partitionnement se compose principalement des étapes suivantes :

- **Initialisation des partitions** : Les états de l'automate sont d'abord divisés en deux groupes principaux : les états finaux, qui acceptent un motif, et les états non finaux, qui ne l'acceptent pas. Ces deux groupes forment la partition initiale.

— **Raffinement des partitions** : À partir de cette partition initiale, chaque partition est ensuite subdivisée en fonction des transitions des états pour chaque symbole de l'alphabet. Si deux états d'une même partition ont des comportements différents (c'est-à-dire, s'ils passent vers des partitions distinctes pour un même symbole), ils sont placés dans des sous-partitions différentes. On s'arrete lorsque les partitions ne changent plus.
— **Fusion des états équivalents** : Une fois le processus de partitionnement terminé, les états appartenant à une même partition finale sont fusionnés. Chaque partition correspond à un unique état dans l'automate fini minimal, réduisant ainsi le nombre total d'états tout en préservant les mêmes capacités de reconnaissance de motifs.

La complexité de cet algortihme est en $O(nlog(n))$, où n est le nombre d'états du DFA, ce qui le rend plutôt efficace pour les automates de grande taille.

4.3 Knuth-Morris-Pratt (KMP)

L'algorithme Knuth-Morris-Pratt (KMP) est une méthode efficace pour la recherche d'un motif dans une chaîne de caractères. Elle exploite les répétitions à l'interieur du motif pour éviter des comparaisons inutiles.
L'algorithme KMP utilise une table de retenue, qui indique pour chaque position du motif la longueur du plus long préfixe correspondant aussi à un suffixe. En cas de non-correspondance, cette table permet de reprendre la recherche sans revérifier les caractères déjà comparés, ce qui accélère la recherche par rapport à une méthode brute-force.

Cette table est obtenue en utilisant les formules suivantes :
Soit P un tableau contenant le motif.
— $\forall i \in [0, n], n = len(P) : LTS[i] =$ size of the longest prefix which is suffix
— $\forall i \in [1 \dots n-1] : P[i] = P[LTS[i]] \wedge LTS[LTS[i]] = -1 \Rightarrow LTS[i] = -1$
— $\forall i \in [1 \dots n-1] : P[i] = P[LTS[i]] \wedge LTS[LTS[i]] \neq -1 \Rightarrow LTS[i] = LTS[LTS[i]]$

4.4 Comparaison entre les méthodes de recherche

La complexité temporelle pour reconnaître un motif à l'aide d'un automate fini déterministe minimisé est $O(n)$, où n est la longueur du texte, ce qui en fait une méthode très efficace capable de traiter des motifs en temps linéaire. Cependant, son inconvénient réside dans le fait que la construction d'un DFA minimisé peut entraîner une explosion exponentielle du nombre d'états dans certains cas, rendant leur utilisation impraticable pour des motifs complexes. En effet, la transformation d'une expression régulière en un DFA minimisé se fait au total en $O(n + 2^e + f \log f)$, avec n la taille de l'expression régulière, e le nombre d'états dans le NDFA et f le nombre d'états dans le DFA.

En comparaison, l'algorithme de Knuth-Morris-Pratt (KMP) présente une complexité temporelle de $O(n+m)$, où n est la longueur du texte et m la longueur du motif. Sa véritable force réside dans sa capacité à gérer des motifs particulièrement longs et des textes importants, car KMP réduit le nombre de comparaisons nécessaires grâce à sa table de retenue qui lui permet de sauter des comparaisons inutiles. De plus, la construction de cette table dans KMP se fait efficacement avec une complexité de $O(m)$.

Bien que les deux méthodes offrent des performances linéaires dans la recherche de motifs, elles se distinguent par leur complexité de construction et leur efficacité.
Le DFA nécessite une étape de construction qui peut être coûteuse en mémoire et en temps, notamment lorsque l'expression régulière est complexe. À l'inverse, KMP n'exige pas de construire un automate et se concentre uniquement sur le motif et le texte à traiter, ce qui peut lui permettre d'afficher une complexité plus favorable malgré sa complexité en $O(n+m)$.
Enfin, en termes de simplicité d'implémentation, KMP est généralement plus facile à mettre en oeuvre par rapport à la construction d'un DFA où l'implémentation exige une compréhension plus approfondie des automates, ce qui peut représenter un obstacle pour certains développeurs.

4.5 Autres Algorithmes

Il existe plusieurs autres algorithmes bien connus pour la recherche de motifs, chacun ayant ses propres avantages et inconvénients en termes de complexité, de performance, et d'efficacité pour différents types de motifs et de textes.

Parmi ces algorithmes, nous retrouvons :

— **Minimisation de Aho-Ullman** : L'algorithme de minimisation proposé par Aho-Ullman, bien qu'instructif, a été décidé de ne pas être implémenté dans ce projet en raison de son inefficacité. Cette méthode repose sur des étapes de partitionnement qui peuvent devenir coûteuses en temps et en mémoire lorsque le nombre d'états est élevé, ralentissant considérablement le processus de minimisation. En effet, sa complexité temporelle est de $O(n^2)$, où n est le nombre d'états dans l'automate.

— **Boyer-Moore** [2] : L'algorithme Boyer-Moore est l'une des méthodes les plus efficaces pour la recherche de motifs, en particulier lorsque le motif est long et que l'alphabet est large. Cet algorithme utilise des heuristiques pour sauter des sections du texte en fonction des symboles non correspondants, ce qui le rend très efficace pour les longues chaînes de caractères.
Sa complexité temporelle dans le pire des cas est $O(nm)$, où n est la longueur du texte et m la longueur du motif, ce qui est similaire à la méthode naïve. Cependant, dans la majorité des cas, il se comporte beaucoup mieux avec une complexité moyenne proche de $O(n/m)$. Comparé à KMP et aux DFA, Boyer-Moore excelle donc pour des motifs longs et dans des contextes où de nombreuses comparaisons peuvent être évitées grâce à ses heuristiques, mais dans le pire des cas, il peut être plus lent que ces derniers où la linéarité est garantie.

— **Brzozowski** : L'algorithme de Brzozowski repose sur un processus d'inversion double : d'abord, il inverse le DFA initial pour obtenir un NDFA, qu'il minimise, puis il inverse à nouveau le NDFA minimisé pour obtenir un DFA minimal. Sa complexité est de $O(n^2)$, où n est le nombre d'états dans l'automate.
Cet algorithme est simple à comprendre et à mettre en oeuvre, mais il s'avère souvent moins efficace que d'autres méthodes de minimisation plus directes comme

2. https ://www.cs.utexas.edu/ moore/publications/fstrpos.pdf

celle de Hopcroft. En effet, l'inversion répétée des automates et la minimisation des NDFAs peuvent entraîner des coûts de calcul élevés, particulièrement lorsque le nombre d'états devient important. Comparé aux DFA minimisés directement ou à l'algorithme KMP, cet algorithme est donc moins performant pour la recherche de motifs.

5 Tests et Résultats

5.1 Testbeds

Pour évaluer les performances des algorithmes de recherche de motifs, nous avons sélectionné les 41 livres les plus téléchargés de la base de Gutenberg [3] comme testbeds. En effet, cela nous offre une grande variété de contenus et de tailles de fichiers, permettant ainsi de réaliser des tests complets et diversifiés.

5.2 Tests de performance

5.2.1 Temps d'exécution

Pour évaluer et comparer les performances en temps d'exécution des différentes méthodes de recherche de motifs (automates finis, KMP et `egrep`), nous avons mesuré le temps d'exécution de chaque méthode en fonction du nombre de lignes présentes dans les fichiers de notre testbeds. Chaque recherche a été exécutée 50 fois sur chaque fichier, et une moyenne sur ces 50 exécutions a été calculé pour assurer la fiabilité des résultats.

Nous avons réalisé deux séries de tests distinctes : la première avec un motif court, représentant un cas optimal pour les algorithmes, et la seconde avec un motif plus long pour KMP et un motif spécifiquement conçu pour l'automate, chacun servant de cas défavorable pour chaque algorithme. Cette approche nous permet d'observer comment chaque méthode se comporte face à des motifs variés, tant en termes de longueur qu'en termes de complexité.

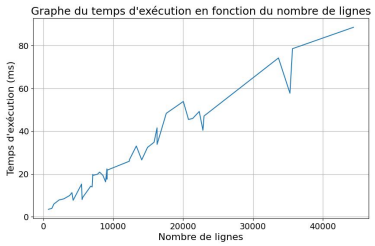

FIGURE 1 – Temps d'exécution avec automate

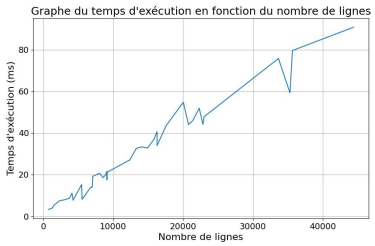

FIGURE 2 – Temps d'exécution avec KMP

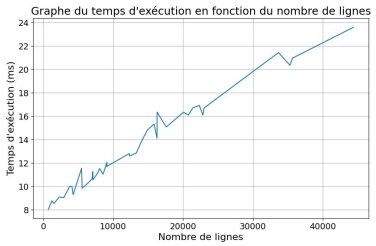

FIGURE 3 – Temps d'exécution avec `egrep`

Les courbes obtenues montrent une progression linéaire du temps d'exécution en fonction du nombre de lignes pour les trois méthodes. Chacune d'elles présente ainsi une complexité temporelle linéaire par rapport à la taille de l'entrée, ce qui est en accord avec leurs complexités théoriques.

3. http ://www.gutenberg.org/

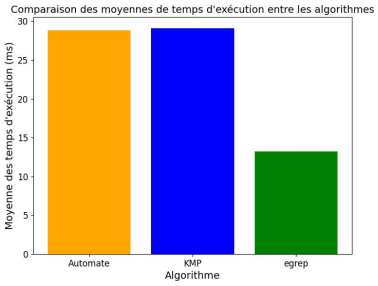

FIGURE 4 – Cas optimal

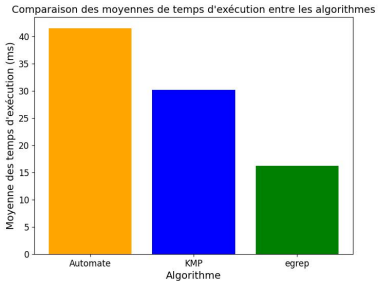

FIGURE 5 – Cas défavorable

Dans la comparaison de leurs moyennes, les temps d'exécution des automates finis et de l'algorithme KMP sont identiques pour le cas optimal, atteignant 28 ms en moyenne pour tous les fichiers. Cela est logique car dans ce scénario où le motif est court, les deux méthodes traitent le texte de manière similaire, permettant une recherche efficace. En effet, KMP a une complexité temporelle de O(n+m), où n est la longueur du texte et m celle du motif. Ainsi, dans ce cas où m est court, la complexité de KMP se rapproche de celle de l'automate qui est en O(n).
Cependant, dans le cas défavorable, où le motif inclut des éléments complexes pour l'automate tels que l'opérateur étoile (*), le temps d'exécution finit par être significativement plus long que celui de KMP, atteignant 41 ms en moyenne. En effet, l'utilisation de l'étoile entraîne des vérifications supplémentaires à chaque itération, car elle permet de matcher zéro ou plusieurs occurrences du caractère précédent. Cela peut entraîner un grand nombre de transitions et un parcours d'états plus long dans l'automate, ce qui explique ainsi ce temps d'exécution plus long.
Enfin, il est à noter que dans les deux cas, la commande `egrep` s'avère être deux à trois fois plus rapide que les autres méthodes, atteignant seulement 13 à 15 ms en moyenne. Cette rapidité peut être attribuée à son optimisation intrinsèque et à son implémentation robuste, qui exploitent des algorithmes avancés pour la recherche de motifs dans des textes, lui permettant ainsi de surpasser KMP et les automates finis, quel que soit le contexte.

5.2.2 Consommation mémoire

Pour la consommation mémoire, nous avons mesuré celle de chaque méthode sur un seul fichier, mais en utilisant plusieurs patterns de tailles différentes. Comme pour les mesures de temps d'exécution, chaque méthode a été exécutée 50 fois, et une moyenne a été calculée à partir de ces 50 exécutions. Cela nous a permis de générer des graphes représentant la consommation mémoire en fonction de la longueur du motif.

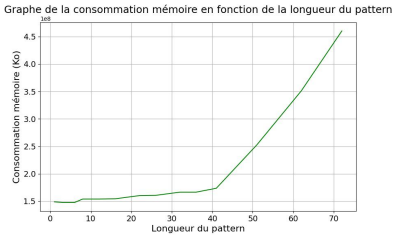

FIGURE 6 – Consommation mémoire avec automate

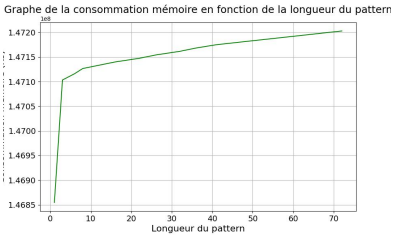

FIGURE 7 – Consommation mémoire avec KMP

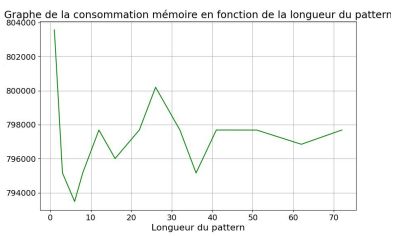

FIGURE 8 – Consommation mémoire avec `egrep`

Les courbes obtenues pour la consommation mémoire des différentes méthodes révèlent des comportements distincts. Pour l'automate fini, nous observons une courbe à tendance exponentielle. Cela peut s'expliquer par la nécessité de stocker toutes les transitions possibles entre les états, qui augmentent rapidement avec la longueur et la complexité du motif. Chaque nouveau symbole ou combinaison introduit de nouveaux états, conduisant à une augmentation de l'utilisation mémoire.
Pour KMP, la courbe suit une tendance logarithmique. Cela est attendu, car KMP se base sur une table de prétraitement (tableau de retenue), qui reste relativement compacte en comparaison à la structure de l'automate, même pour des motifs plus longs. La consommation mémoire augmente donc de manière plus lente en fonction de la taille du motif.
Enfin, la courbe de consommation mémoire de `egrep` montre des fluctuations irrégulières, pouvant s'expliquer par des optimisations internes spécifiques à son implémentation. En effet, certaines optimisations pourraient être plus coûteuses en mémoire pour certains motifs, expliquant ainsi ces variations.

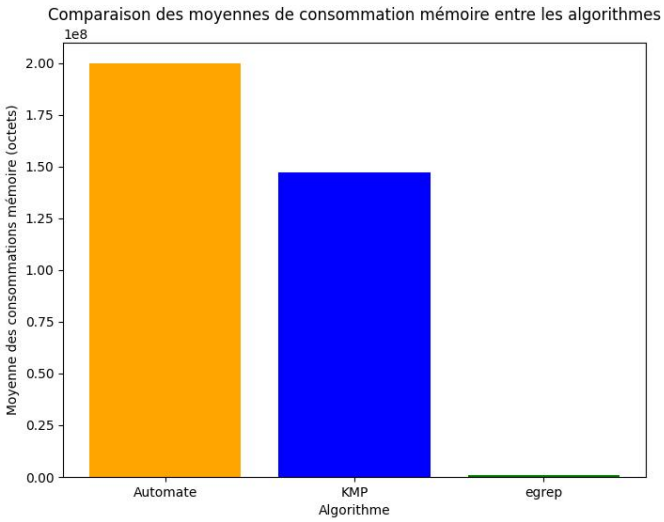

FIGURE 9 – Moyenne des consommations mémoire

Nous avons observé des moyennes de consommations mémoire présentant des différences nettes entre les trois méthodes. L'automate fini affiche une consommation mémoire moyenne atteignant quasiment 200 000 Ko, ce qui reflète la surcharge en mémoire liée à la création d'un grand nombre d'états et de transitions. La consommation moyenne de KMP, quant à elle, est plus économe, atteignant 147 000 Ko, démontrant qu'elle ne nécessite pas autant de ressources que la représentation d'un automate. Enfin, avec seulement 800 Ko en moyenne, `egrep` est de loin la méthode la plus légère en mémoire. Cela reflète les optimisations internes poussées de `egrep`, qui, en plus d'offrir une rapidité supérieure, minimise également la consommation de ressources mémoire.

6 Conclusion

La recherche de motifs textuels est un domaine clé en informatique, avec des applications vastes allant de l'analyse de données à la bioinformatique. Ce projet a permis d'explorer différentes techniques, en montrant leurs forces et faiblesses dans divers contextes. Chacune des méthodes testées a montré des performances différentes selon les contextes, soulignant l'importance d'un choix d'algorithme adapté.
Cependant, les résultats révèlent également des limitations, notamment en termes de gestion des motifs longs et complexes, qui impactent à la fois les temps d'exécution et la consommation mémoire. Les automates minimisés se révèlent très efficaces pour des motifs complexes, mais leur coût en mémoire peut devenir prohibitif. De son côté, l'algorithme Knuth-Morris-Pratt s'avère performant pour des motifs simples, offrant une solution rapide et économe en ressources. Les résultats montrent ainsi qu'il existe un compromis entre la rapidité d'exécution et l'utilisation des ressources.

Pour répondre aux défis de la recherche de motifs textuels dans un environnement offline, ce projet a montré que, bien que chaque algorithme présente des avantages spécifiques, les limitations en termes de rapidité et de consommation mémoire restent importantes. Pour améliorer ces performances, l'introduction de techniques d'indexation locale et de prétraitement des données pourrait considérablement réduire les temps d'exécution en évitant de parcourir systématiquement tout le fichier pour chaque recherche. De plus, une approche hybride combinant l'algorithme KMP et les automates minimisés permettrait de maximiser l'efficacité en adaptant dynamiquement le choix de l'algorithme selon la nature du motif et la taille des fichiers. Aussi, le recours à des techniques de parallélisation ou de calcul distribué local permettrait d'exploiter pleinement les ressources disponibles pour traiter de grands volumes de données textuelles tout en restant dans un cadre offline.
Ces perspectives ouvrent la voie à des moteurs de recherche offline plus rapides, plus économes en ressources, et capables de traiter des motifs complexes dans des contextes de données massives.