Sorbonne Université

Master Science et technologie du logiciel

Rapport PSTL

Moteur de recherche

rédigé par

Samira FAWAZ
Kheireddine OUNNOUGHI

Table des matières

1 Présentation du projet

1.1 Contexte

La recherche rapide d'informations dans de grandes quantités de texte est un enjeu fondamental en informatique car elle intervient dans un grand nombre d'applications.

Pour chercher des motifs dans des fichiers texte, on utilise des outils de recherche textuelle comme grep et egrep, qui permettent de parcourir rapidement de grandes quantités de données pour trouver des correspondances précises.

Notre projet consiste à développer un clone de `egrep`, c'est-à-dire un moteur de recherche textuel capable de localiser des motifs dans des fichiers textuels.

La recherche de motifs est un problème central en informatique et trouve des applications dans de nombreux domaines, tels que les moteurs de recherche, l'analyse de fichiers et de logs. La gestion efficace de grands volumes de texte et la rapidité de recherche sont des enjeux majeurs dans ces contextes. contextes.

Ce travail s'appuie sur les algorithmes de recherche de motifs étudiés durant le cours et a pour but de mettre en œuvre plusieurs approches afin de les comparer dans un cadre expérimental cohérent. Nous avons ainsi choisi d'implémenter des algorithmes déterministes comme Knuth–Morris–Pratt (KMP), des méthodes heuristiques telles que Boyer–Moore, ainsi qu'une approche plus générale basée sur les expressions régulières et leur implémentation via des automates finis (AST $\rightarrow$ NFA $\rightarrow$ DFA).

L'objectif de ce rapport est de présenter les structures de données et les mécanismes internes de chaque algorithme, d'analyser leur complexité théorique, puis d'évaluer leurs performances expérimentales sur des ensembles de textes de référence, notamment issus du projet Gutenberg [1]. Nous discuterons enfin des résultats obtenus.

Nous avons également pris soin de mesurer les performances sur un grand nombre de lignes (plusieurs centaines de milliers) et de traiter les données de manière à limiter la volatilité dans les graphiques, afin de fournir une analyse claire et représentative.

La suite du rapport présente les détails des algorithmes étudiés, la méthodologie expérimentale, puis l'analyse des résultats obtenus.

1.2 Source

Notre code est disponible à l'adresse suivante : SearchEngine.

2 Explication des Algorithmes utilisés

2.1 RegEx

L'algorithme se compose de trois étapes : la construction de l'arbre syntaxique (AST), la construction de l'automate fini non déterministe (NFA), puis la minimisation de l'automate pour le rendre déterministe et accélérer son accès. Pour l'implémentation des expressions régulières, nous nous sommes inspirés du livre d'Aho-Ullman [2].

2.1.1 AST

Pour les expressions régulières utilisées dans notre arbre syntaxique, nous avons suivi les Extended Regular Expressions définies dans la version 2 de Single Unix [3].

2.1.2 Structure de données

Nous avons défini les types d'opérations à l'aide d'un **énumérateur** (‘enum‘), afin de représenter de manière claire les différentes opérations possibles dans les expressions régulières.

Algorithm 1 Définition de la classe `Operation`

```
1: class Operation(Enum) :
2:     DOT ← 1
3:     ETOILE ← 2
4:     PLUS ← 3
5:     ALTERN ← 4
6:     PARENTHESE_L ← 5
7:     PARENTHESE_R ← 6
8:     CONCAT ← 7
9:     PROTECTION ← 8
```

2.1.3 Classes pour l'arbre syntaxique

Pour la représentation de l'arbre syntaxique, nous avons défini deux classes principales : `RegExTree` et `RegEx`.

- `RegExTree` : Cette classe représente un nœud de l'arbre syntaxique. Chaque nœud contient :
  - un attribut `root`, correspondant à l'opération ou caractère associé au nœud (par exemple `CONCAT`, `ETOILE`, etc.)
  - une liste `subTrees` contenant les enfants (sous-nœuds) de ce nœud.
  - un identifiant `id` optionnel, utile pour le débogage.

  La classe fournit également une fonction `rootToString` qui retourne une représentation textuelle de l'opération du nœud, et une méthode `__str__` pour afficher l'arbre sous forme de chaîne.
- `RegEx` : Cette classe représente une expression régulière entière. Elle contient :
  - l'expression régulière sous forme de chaîne de caractères ;
  - plusieurs fonctions de vérification (`contain`) pour déterminer si un certain type d'opération est présent dans l'arbre ;
  - des fonctions de traitement (`process`) pour transformer les nœuds selon leur type d'opération ;
  - la fonction principale `parseList` qui prend une liste de nœuds et construit l'arbre syntaxique complet en appliquant successivement les opérations.

Algorithm 2 Classes pour l'analyse d'AST

```
1: class RegExTree :
2:     attributs : root, subTrees, id
3:     méthode : rootToString()          ▷ Retourne une représentation en chaîne du nœud racine
4:
5: classe RegEx :
6:     attribut : regex
7:     méthodes :
8:         chartoRoot(trees)                                  ▷ Transforme un caractère en opération
9:         ContainOperation(trees)          ▷ Vérifie si une opération existe dans les sous-arbres
10:        ProcessOperation(trees)                ▷ Traite les nœuds selon leur type d'opération
11:        parse()          ▷ Construit l'arbre syntaxique complet à partir de l'expression régulière
```

2.1.4 Complexité

Complexité temporelle :

— Prétraitement : $O(2^m)$ construction de l'AST, transformation en NFA puis en DFA

— Recherche : $O(n)$ dans le meilleur des cas (avec DFA), mais peut atteindre $O(2^m \cdot n)$ dans le pire cas (avec backtracking)

— **Total :** $O(2^m + n)$ **meilleur cas,** $O(2^m \cdot n)$ **pire cas**

Paramètres : n = longueur du texte, m = longueur du motif

2.2 Knuth-Morris-Pratt Algorithm

L'algorithme Knuth-Morris-Pratt (KMP) permet de rechercher efficacement un motif (pattern) dans un texte. Contrairement aux expressions régulières, il ne prend pas une expression complexe comme pattern, mais un mot simple, et identifie toutes ses occurrences dans le texte.

2.2.1 Exemple

`Text` : 'ABCABDABCABD' `Pattern` : 'ABCABD'
Calcul du vecteur LPS du pattern :

$$\text{Pattern} = ABCABD$$

$$\text{LPS} = [0, 0, 0, 1, 2, 0]$$

Le pattern apparaît dans le texte **2 fois**, aux indices 0 et 5.

2.2.2 Fonctionnement

L'initialisation se fait avec le pattern et le vecteur LPS (Longest Prefix Suffix), où chaque case contient la longueur du plus long préfixe qui est également un suffixe jusqu'à cet indice. La boucle parcourt le pattern à partir du deuxième caractère, en augmentant cette longueur si les caractères correspondent, ou en revenant au préfixe précédent déjà calculé dans le LPS en cas de mismatch. Le tableau LPS permet à l'algorithme KMP d'éviter les comparaisons inutiles lors de la recherche du pattern dans le texte.

La fonction de recherche parcourt le texte caractère par caractère en comparant avec le pattern et utilise le LPS pour ne pas recommencer depuis zéro à chaque mismatch. Le résultat permet de retrouver à la fois **les indices de début de chaque occurrence** et le **nombre total d'occurrences** du pattern dans le texte.

2.2.3 Structure de données

Algorithm 3 Classe KMP

1: **class** KMP :
2: **attributs :** pattern, lps
3: **méthodes :**
4: compute_lps(pattern) ▷ Retourne la liste LPS du pattern fourni.
5: match_kmp(text, max_matches) ▷ Cherche le pattern dans le texte, retourne les indices de début et le nombre total d'occurrences.

2.2.4 Complexité

Complexité temporelle :
— Prétraitement : $O(m)$ construction du tableau LPS (Longest Prefix Suffix)
— Recherche : $O(n)$ - chaque caractère du texte est lu au plus une fois grâce au tableau LPS
— **Total :** $O(n + m)$

2.3 Boyer-Moore Algorithm

L'algorithme de Boyer-Moore est un algorithme heuristique permettant de rechercher efficacement un motif (pattern) dans un texte. Comme KMP, il prend un mot simple comme pattern et identifie toutes ses occurrences, mais il utilise une approche différente basée sur la comparaison du pattern de droite à gauche et des sauts optimisés.

2.3.1 Fonctionnement

L'initialisation se fait avec le pattern et un dictionnaire de sauts appelé *hoops*, qui indique pour chaque caractère combien de positions avancer en cas de mismatch. Pour chaque caractère du pattern, on calcule :

$$\text{hoops}[\textit{pattern}[i]] = \max(1, \text{len}(\text{pattern}) - i - 1)$$

et pour tous les autres caractères qui ne sont pas dans le pattern, le saut par défaut est la longueur du pattern.

La recherche parcourt le texte caractère par caractère, en comparant le pattern de droite à gauche. En cas de mismatch, le dictionnaire *hoops* permet de sauter directement le nombre de positions approprié, au lieu d'avancer d'un seul caractère. Lorsqu'un match est trouvé, on peut également sauter selon le caractère suivant ou utiliser le saut par défaut pour continuer la recherche.

2.3.2 Structure de données

Algorithm 4 Classe Boyer-Moore

1: **class** Boyer :
2: **attributs :** pattern, hoops, m ▷ m est la taille du pattern
3: **méthodes :**
4: compute_hoops(pattern) ▷ Retourne le dictionnaire des sauts pour chaque caractère (heuristique du mauvais caractère).
5: match_boyer(text, max_matches) ▷ Cherche le pattern dans le texte et retourne les indices de début ainsi que le nombre total d'occurrences.

Complexité temporelle :
— Prétraitement : $O(m)$ — construction des tables "bad character" et "good suffix"
— Recherche : $O(n/m)$ meilleur cas, $O(nm)$ pire cas, $O(n)$ cas moyen
— **Total :** $O(n)$ **en moyenne pour du texte aléatoire**
Paramètres : n = longueur du texte, m = longueur du motif

3 Tests des performances

Dans cette section, nous comparons les performances de nos algorithmes afin d'évaluer leur comportement face à un grand volume de données.

3.1 Environnement de test

Pour nos tests, nous avons utilisé des livres en anglais comme source de texte. Chaque test consiste en **100 000 appels de la fonction de recherche**, avec une mesure du temps d'exécution pour chaque itération. À chaque appel, la fonction lit une ligne de texte (les lignes étant traitées une par une) et cherche le mot passé en paramètre comme motif (*pattern*).

Nous avons choisi les mots suivants pour nos tests :

```
ListDesMots = [the, and, man, love, world, responsibility, extraordinary]
```

Les cinq premiers mots ont été sélectionnés car ils sont très fréquents dans les ouvrages en anglais, tandis que les deux derniers ont été choisis pour leur longueur plus importante, afin d'observer si celle-ci influence les performances. Aucune difficulté particulière n'a été constatée, même pour les mots les plus longs.

Pour obtenir un fichier de test plus complet, nous avons combiné les résultats des sept mots, ce qui représente environ **700 000 lignes de données**. Afin de rendre les courbes plus lisibles et d'éviter une trop grande volatilité, nous avons regroupé les données par échantillons de 30 000 lignes et appliqué un léger lissage aux graphes.

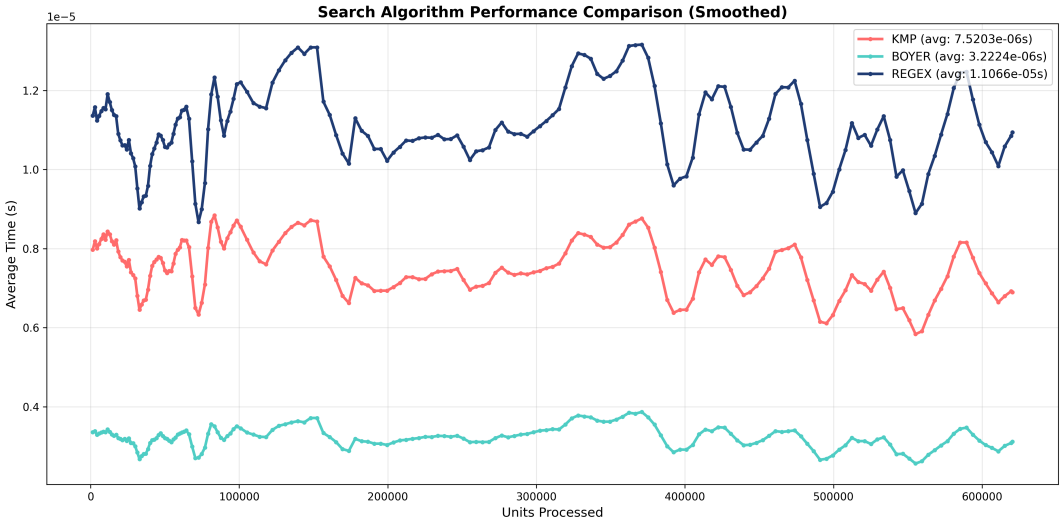

FIGURE 1 – Comparaison des temps d'exécution des algorithmes.

D'après le graphe, on peut voir que les trois algorithmes exécutent leurs recherches très rapidement. Leurs temps d'exécution restent relativement volatils, ils montent et descendent par moments, mais sans jamais exploser ni atteindre des valeurs anormales. On remarque également une différence nette de performance entre eux : **Boyer-Moore** est le plus rapide, ce qui confirme sa complexité théorique, suivi par **KMP**, puis **Regex**, qui reste le plus lent avec un temps d'exécution moyen d'environ 1.1×10^{-5} secondes par ligne.

4 Conclusion

Dans ce projet, nous avons implémenté, testé et comparé trois algorithmes de recherche de motifs : KMP, Boyer-Moore et RegEx.

Les résultats expérimentaux confirment les prévisions théoriques : l'algorithme de Boyer-Moore est le plus performant grâce à sa stratégie de sauts optimisés, suivi de KMP. Les expressions régulières, bien qu'elles soient puissantes et flexibles, présentent un temps d'exécution plus élevé en raison de leur complexité interne.

Ce travail nous a permis de mieux comprendre les différences entre les approches déterministes et heuristiques pour la recherche textuelle, ainsi que l'importance de sélectionner l'algorithme le mieux adapté au contexte d'utilisation.

Dans la suite, il serait intéressant d'étendre le projet en prenant en charge des fonctionnalités plus avancées et en explorant des techniques d'optimisation supplémentaires. Ce projet a ainsi constitué une excellente mise en pratique des notions théoriques et une expérience enrichissante sur le plan algorithmique.

Références

[1] Project Gutenberg. Project Gutenberg. Digital library of public domain books. Date de consultation : 2025-10-12. URL : https://www.gutenberg.org/.

[2] Alfred V. Aho and Jeffrey D. Ullman. Foundations of Computer Science, 1992. Date de consultation : 2025-10-12. URL : http://infolab.stanford.edu/~ullman/focs.html.

[3] The Open Group. Regular Expressions, 1997. POSIX.2 Standard. URL : https://pubs.opengroup.org/onlinepubs/7908799/xbd/re.html.