Project CPA
Minimum enclosing circle problems

Submitted by

Haotian Xue
Hejun Cao
M2 Science et Technologie du Logiciel 2024-2025
Sorbonne Universite (SU UPMC)

Under the guidance of

Binh-Minh Bui-Xuan
Laboratoire d'Informatique de Paris 6 (LIP6)
Centre National de la Recherche Scientifique (CNRS)
Sorbonne Universite (SU UPMC)

05/10/2024

Contents

1 Introduction

Le projet que nous entreprenons vise à créer un clone du célèbre outil egrep utilisé dans les systèmes Unix pour la recherche de motifs dans des fichiers texte. L'objectif principal de ce projet est de concevoir une version offline et desktop de ce moteur de recherche, en se limitant à certaines conventions de la norme ERE (Expressions Régulières Étendues). Les éléments de base que nous allons prendre en compte sont : les parenthèses (), l'alternative —, la concaténation (implicite), l'opération étoile *, le point (caractère universel .), ainsi que les lettres ASCII.

L'enjeu de ce projet est double : non seulement il permet de se familiariser avec des concepts algorithmiques fondamentaux liés à la recherche de motifs, mais il met également en pratique des structures de données efficaces comme les automates finis déterministes (DFA) et non déterministes (NFA), ainsi que l'algorithme de recherche de motifs de Knuth-Morris-Pratt (KMP). Chacune de ces approches présente des avantages et des inconvénients spécifiques selon les contextes d'utilisation, que ce projet permet d'explorer.

L'un des aspects les plus complexes de ce projet réside dans la conversion des expressions régulières en automates finis, notamment en NFA, puis en DFA, permettant ainsi un traitement plus rapide des recherches. De plus, nous implémentons une recherche par KMP pour les cas où les motifs sont des chaînes simples. L'objectif final est de fournir un moteur de recherche performant et fonctionnel capable de traiter efficacement de grands volumes de texte, tout en respectant les contraintes de performance et de simplicité imposées par une application desktop.

La phase de test et d'évaluation des performances est cruciale dans ce projet. En effet, nous utiliserons la base de données Gutenberg, qui propose de nombreux textes disponibles en libre accès, pour tester nos algorithmes sur des corpus variés et de grande taille. Ces tests nous permettront d'analyser les performances de nos implémentations et d'identifier les points d'amélioration possibles.

2 Problème Défini

Le projet consiste à concevoir un moteur de recherche offline capable de reproduire le comportement de la commande egrep. Plus précisément, le moteur doit permettre la recherche de motifs complexes au sein de fichiers texte en utilisant des expressions régulières. Cependant, contrairement à un support complet des expressions régulières, notre implémentation se limite à un sous-ensemble des fonctionnalités de la norme ERE (Expressions Régulières Étendues).

L'objectif du projet est donc de proposer une solution qui convertit les expressions régulières décrites ci-dessus en une structure efficace, capable de trouver rapidement les occurrences de motifs dans de grands fichiers texte. L'accent est mis sur la performance du moteur de recherche, ainsi que sur la capacité à traiter efficacement les expressions régulières grâce à des techniques avancées comme les automates finis et l'algorithme KMP.

Cependant, en raison de la nature même de l'algorithme KMP, nous ne pouvons rechercher que le texte donné au lieu de faire correspondre une expression régulière complète.

3 Structure de Données

3.1 NFA et NFAState

Classe NFA contient les états de départ et de fin de l'automate.

Pour classe NFAState, chaque état d'un NFA est représenté par un objet qui stocke un ensemble de transitions vers d'autres états en fonction de caractères ou d'ε-transitions.

3.2 DFA et DFAState

Classe DFA est responsable de la conversion du NFA en DFA, ainsi que de la recherche dans le texte.

Pour classe DFAState, chaque état du DFA possède des transitions déterministes basées sur un seul caractère et indique si l'état est un état acceptant (fin de motif).

4 Aho-Ullman

L'algorithme de conversion NFA en DFA, tel que décrit par Aho et Ullman dans leur ouvrage, constitue l'un des piliers fondamentaux de notre projet. Cet algorithme est crucial pour optimiser la recherche de motifs complexes dans les fichiers texte en transformant des expressions régulières en automates. Nous détaillerons ici les principales étapes de l'algorithme et analyserons la complexité temporelle de chaque étape, ainsi que son comportement dans les cas optimaux et pessimistes.

4.1 Conversion d'une expression régulière en NFA

L'algorithme commence par la conversion d'une expression régulière en un automate fini non déterministe (NFA). Cette étape est relativement simple et suit des règles déterministes :

- Chaque caractère ou élément de l'expression régulière (comme ‘a‘, ‘b‘, etc.) est représenté par un état du NFA. - Les opérateurs réguliers comme l'alternance (‘—‘), l'étoile (‘*‘), et la concaténation sont gérés en ajoutant de nouveaux états et transitions au NFA. Par exemple : - L'opérateur ‘—‘ crée deux chemins parallèles à partir d'un même état initial. - L'opérateur ‘*‘ permet de retourner à un état précédent ou d'avancer sans lire de caractère. - La concaténation relie deux états de manière séquentielle.

La complexité de cette phase est proportionnelle à la taille de l'expression régulière, soit $\mathbf{O(k)}$, où k est la longueur de l'expression régulière. À ce stade, le nombre d'états est contrôlé et reste relativement petit, car les NFA gèrent plusieurs transitions depuis un même état.

4.2 Conversion du NFA en DFA

La partie la plus coûteuse de l'algorithme est la conversion du NFA en un DFA à l'aide de l'algorithme de construction de sous-ensembles. Chaque état du DFA représente un ensemble d'états du NFA, et la transition d'un état à un autre dans le DFA correspond à des transitions multiples dans le NFA.

Cette phase suit les étapes suivantes : - **Fermeture epsilon** : Pour chaque état du NFA, on calcule la fermeture epsilon (‘-closure‘), c'est-à-dire l'ensemble des états accessibles sans lire de caractère (via des transitions epsilon). Cela permet de définir des états déterministes dans le DFA. - **Détermination des transitions** : Pour chaque symbole de l'alphabet, on calcule les transitions possibles à partir des états actuels du NFA et on les regroupe pour former un état unique dans le DFA. - **Ajout des états et transitions au DFA** : Si un nouvel ensemble d'états est trouvé (un ensemble non encore exploré), il est ajouté au DFA avec les transitions correspondantes.

La complexité de cette conversion dépend du nombre d'états du NFA. Dans le pire des cas, un NFA avec n états peut générer un DFA avec jusqu'à 2^n états, ce qui correspond à une complexité exponentielle $\mathbf{O(2^n)}$. Cependant, dans de nombreux cas pratiques, la taille réelle du DFA est bien inférieure à cette limite théorique, car beaucoup d'états potentiels ne sont jamais générés ou explorés.

Algorithm 1: ConvertNFAtoDFA

inputs : NFA, *StartState*
outputs: DFA correspondant au NFA

```
1  begin
2      DFAState ← EpsilonClosure(StartState)
3      Init DFA avec DFAState comme état de départ
4      while DFAState non exploré do
5          foreach symbole du NFA do
6              NextStates ← Move(DFAState, symbole)
7              DFAState' ← EpsilonClosure(NextStates)
8              if DFAState' n'existe pas then
9                  Ajouter DFAState' au DFA
10             end
11             Ajouter transition de DFAState vers DFAState' sur symbole
12         end
13     end
14 end
```

4.3 Recherche dans le texte avec le DFA

Une fois le DFA construit, la recherche dans le texte devient très efficace. Le DFA permet de traiter chaque caractère du texte en temps constant, car chaque état du DFA a une transition unique pour chaque symbole de l'alphabet.

Ainsi, la recherche de motifs dans un texte de longueur m prend un temps linéaire, soit **O(m)**. Comme le DFA ne nécessite pas de revenir en arrière ou de considérer plusieurs chemins possibles, chaque caractère est traité une seule fois. Cela fait de l'approche DFA une méthode extrêmement rapide pour la recherche une fois que l'automate est construit.

4.4 Analyse des Cas Optimaux et Pessimistes

Cas optimal :

Dans le cas optimal, l'expression régulière est relativement simple, sans utilisation excessive d'opérateurs comme '—' ou '*'. Par exemple, une expression régulière comme 'abc' ou même 'a*b' génère un NFA avec peu d'états, et la conversion en DFA est rapide. Dans ce scénario, la complexité temporelle est linéaire par rapport à la taille de l'expression régulière, soit **O(k)** pour la construction du NFA et **O(m)** pour la recherche dans le texte. La taille du DFA reste petite, et la recherche est réalisée en temps linéaire.

Le cas optimal se produit donc lorsque l'expression régulière ne génère pas un grand nombre d'états dans le NFA, et lorsque les transitions entre les états sont simples.

Cas pessimiste :

Dans le pire des cas, l'expression régulière contient de nombreux opérateurs '—', '*', et des parenthèses imbriquées, ce qui entraîne une explosion du nombre d'états dans le NFA. Par exemple, une expression comme '(a—b—c)*' génère un nombre important de combinaisons d'états dans le NFA, et la conversion en DFA peut entraîner une explosion exponentielle du nombre d'états.

Dans ce cas, la complexité de la conversion NFA en DFA devient $\mathbf{O(2^n)}$, où n est le nombre d'états du NFA. Cela peut entraîner des temps de calcul très longs et une utilisation excessive

de la mémoire. Cependant, une fois le DFA construit, la recherche dans le texte reste rapide, avec une complexité linéaire **O(m)**.

Le cas pessimiste survient lorsque l'expression régulière est très complexe, impliquant de nombreuses alternatives et répétitions, ce qui fait croître de manière exponentielle la taille du DFA.

4.5 Conclusion pour Aho-Ullman

L'algorithme de conversion NFA en DFA, bien que potentiellement coûteux en temps et en espace dans le pire des cas, reste une méthode extrêmement efficace pour la recherche de motifs complexes une fois le DFA construit. Le DFA permet une recherche en temps constant pour chaque caractère du texte, rendant cette approche particulièrement adaptée aux recherches répétées dans de grands fichiers texte. Cependant, dans le cas où l'expression régulière est extrêmement complexe, il est important de prendre en compte les limites de cet algorithme en termes de mémoire et de temps de conversion.

5 KMP

L'algorithme Knuth-Morris-Pratt (KMP) est un algorithme classique pour la recherche de motifs dans une chaîne de caractères. Il résout efficacement le problème de recherche en évitant les comparaisons redondantes, en utilisant une pré-analyse du motif pour construire un tableau de préfixes, appelé LTS ("longest proper prefix which is also a suffix").

L'algorithme KMP fonctionne en deux phases :

- Pré-traitement
  On calcule un tableau LTS qui, pour chaque position dans le motif, indique le plus long préfixe du motif qui est aussi un suffixe. Cela permet de déterminer où reprendre la recherche lorsqu'une comparaison échoue.

- Recherche
  Pendant la recherche dans le texte, lorsque le motif ne correspond pas au texte à une certaine position, le tableau LTS est utilisé pour savoir combien de caractères on peut sauter dans le texte sans re-vérifier des comparaisons inutiles.

5.1 LTS et CarryOver

La recherche KMP que nous avons apprise en classe est en fait une version modifiée de l'approche de M. Binh-Minh Bui-Xuan. Nous calculons d'abord le tableau LTS traditionnel, puis le tableau CarryOver basé sur le tableau LTS, ce qui réduit le nombre de comparaisons sous les bandes frontières ainsi que les calculs répétés et les retours en arrière inutiles, en particulier lorsqu'il s'agit de faire correspondre un grand nombre de textes dupliqués.

Prenons l'exemple de **P = mami**.

Nous commençons par construire un tableau LTS, dont la première valeur est fixée à -1 et la seconde à 0, comme nous l'a enseigné notre professeur. Ensuite, la troisième valeur est 0, car a et m ne sont pas identiques. Ensuite, la quatrième valeur est 1, car "mam" constitue un préfixe et un sufixe, et le nombre de lettres identiques est 1. La dernière valeur est 0, car "mami" ne constitue pas un préfixe et un suffixe.

Le tableau **LTS** est donc **[-1, 0, 0, 1, 0]**

La complexité temporelle de cet partie est **O(M)** et M est la longueur des caractères à faire correspondre. En effet, nous n'effectuons qu'une seule opération en boucle sur la chaîne de caractères.

Nous construisons ensuite le tableau CarryOver.

En classe, le professeur avait donné la formule suivante :

- $\forall i \in [0,n-1]$, si P[i] = P[LTS[i]] et LTS[LTS[i]] = -1, alors LTS[i] = -1
- $\forall i \in [0,n-1]$, si P[i] = P[LTS[i]] et LTS[LTS[i]] $\neq$ -1, alors LTS[i] = LTS[LTS[i]]

Sur la base de ces deux formules, nous pouvons déterminer que le tableau **CarryOver** est **[-1, 0, -1, 0, 1]**.

La complexité temporelle de cette partie est également **O(M)**, puisque nous n'effectuons toujours qu'une seule boucle.

Par conséquent, la complexité temporelle totale de cette partie du calcul du tableau LTS et du tableau CarryOver est **O(M)**.

5.2 Recherche

Nous commençons par initialiser deux indices i et j, i étant utilisé pour marquer la position du texte tandis que j est utilisé pour marquer la position de P . Nous parcourons ensuite l'ensemble du texte et les quatre correspondances suivantes se produisent :

- P[j] = Texte[i]
  Cela signifie que la lettre à la position actuelle du texte est la même que la lettre à la position correspondante de P. Nous devons ajouter 1 à chacun des indices i et j.
- j = -1
  Les valeurs négatives ne peuvent provenir que du tableau CarryOver. La présence de -1 signifie que la correspondance a échoué, auquel cas nous avançons i plus un jusqu'à la lettre suivante du texte et remettons j à 0.
- j $\neq$ -1 et P[j] $\neq$ Texte[i]
  Ce cas signifie que le texte ne correspond pas et que nous devons effectuer un saut de correspondance correspondant sur la base du tableau CarryOver au lieu de simplement reléguer j à 0. C'est ce qui rend l'algorithme KMP si efficace. Je mets ici j à CarryOver[j].
- j = M
  Cela signifie que nous avons terminé un match. Il suffit de sortir le contenu de la ligne où se trouve le texte, puis de mettre j à 0 et break.

L'algorithme de cette partie est mis en œuvre comme suit :

Algorithm 2: KMPSearch

inputs : $Text$, P, $LineNum$
outputs: Le texte Res contenant P et le nombre Num de lignes qu'il contient

```
1  begin
2      computeLTS
3      computeCarryOver
4      j ← 0; i ← 0
5      while i < N do
6          if j = -1 || Text[i]=P[j] then
7              i++; j++
8          end
9          else
10             j ← CarryOver[j]
11         end
12         if j = M then
13             Print(LineNumber et Text a ce ligne)
14             j ← 0
15             break;
16         end
17     end
18 end
```

La complexité temporelle de cette partie est O(N), où N est le nombre de caractères contenus dans le texte. Parce que nous n'avons parcouru le texte qu'une seule fois.

5.3 Complexité temporelle

La complexité temporelle de l'ensemble de l'algorithme KMP est O(M+N). J'ai déjà analysé la complexité temporelle de chaque partie.

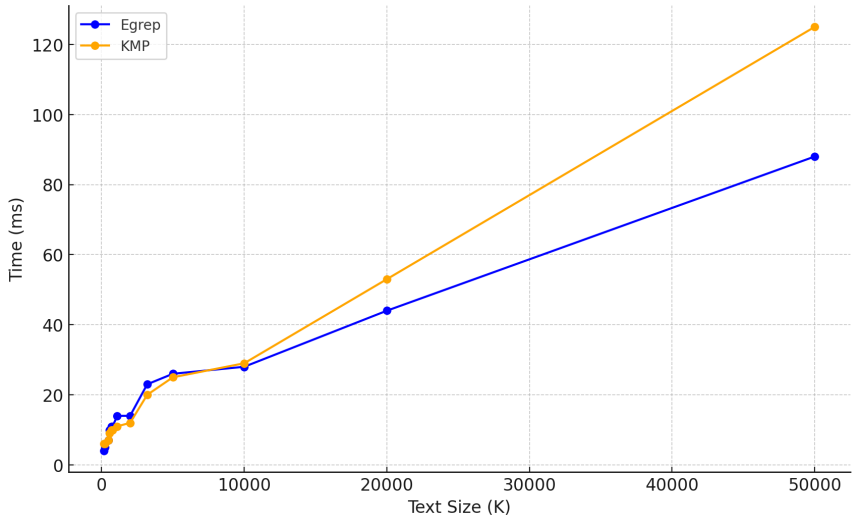
6 Comparaison des performances

Les conditions de l'équipement utilisé pour les tests étaient les suivantes :

- cpu : Intel i7 13650H
- Ubuntu : 24.04
- RAM : 16G 4800MHz
- Java : Java 17

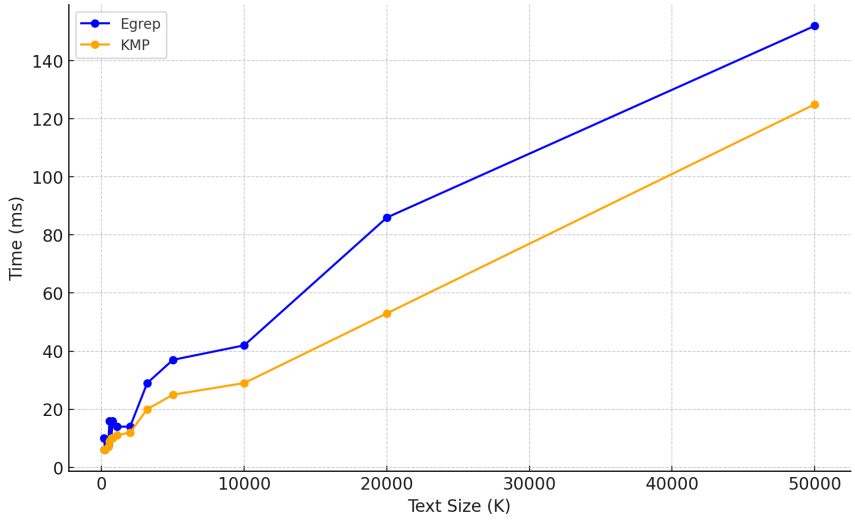
6.1 Comparaison avec Mot

Nous avons comparé les deux scénarios. Dans le premier cas, nous avons cherché des mots sans possibilité de bifurcation, comme "Sargon".



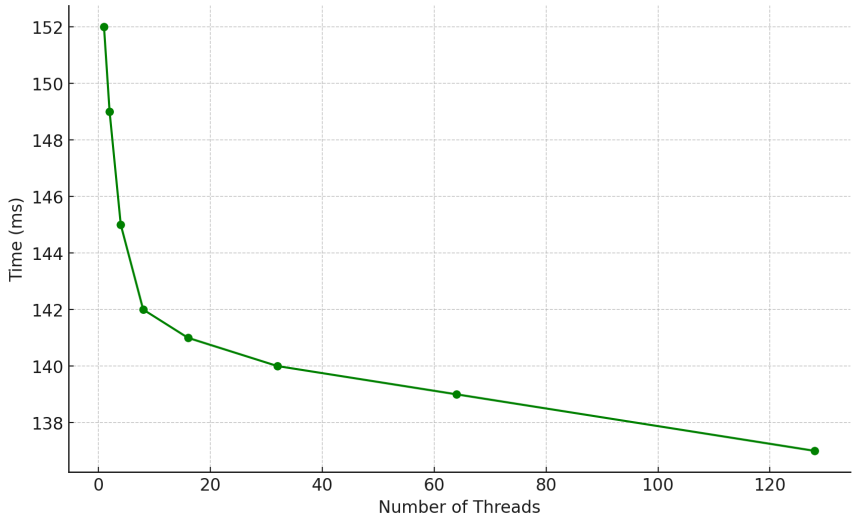
6.2 Comparaison Regex et Mot

Par ailleurs, Egrep effectue une recherche à l'aide de l'expression régulière "S(a|r|g)+on", alors que KMP, en raison de limitations algorithmiques, recherche toujours "Sargon".



7 Optimisation

Nous avons procédé à quelques optimisations de l'algorithme KMP. Normalement, l'algorithme KMP doit lire le texte ligne par ligne avant d'effectuer la recherche. Après l'optimisation, nous utiliserons plusieurs threads pour effectuer la recherche. Voici une comparaison du nombre de threads et du temps passé, pour un texte de 50M.



On peut constater que le multithreading a un certain degré d'effet d'optimisation sur l'utilisation de l'algorithme KMP, mais il n'est pas évident, car le multithreading ne modifie pas la complexité temporelle de l'algorithme, mais accélère seulement la traversée du texte dans une certaine mesure.

8 Conclusion

D'après le tableau comparatif, nous pouvons constater que lors de l'utilisation de mots fixes tels que « Sargon », l'algorithme Egrep rencontrera son cas optimal, dans lequel les algorithmes Egrep et KMP ont la même complexité temporelle O(M+N). Dans ce cas, Egrep et KMP ont la même complexité en temps de O(M+N). Une fois qu'Egrep correspond à des expressions régulières relativement complexes, sa complexité en temps dans le pire des cas sera de $O(2^n)$. En raison de la limitation de la taille du texte, le graphique linéaire ne peut pas bien montrer la courbe de changement.

En ce qui concerne l'optimisation, nous n'avons pas trouvé de bonne solution d'optimisation algorithmique, nous avons donc choisi d'utiliser le multithreading, ce qui était aussi un bon moyen de réviser ce que nous avons appris l'année dernière.

En résumé, le projet a mis en œuvre deux algorithmes classiques de recherche de motifs tout en explorant leurs forces et leurs limites dans différentes situations. Nous avons atteint l'objectif initial du projet en proposant un clone performant d'egrep, tout en explorant la solution algorithmique KMP et son optimisation.