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

KFC Crazy Thursday V Me 50€ !

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 CarryOver[j] et d'ajouter i à un.
  La raison pour laquelle la valeur de j est toujours fixée à CarryOver[j] est que le texte peut survivre à des correspondances répétées. En supposant que le P que nous devons faire correspondre est "mama" et que le texte est "mamama", l'avantage d'utiliser CarryOver[j] au lieu de 0 est que nous ne manquerons pas le deuxième "mama".

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

Algorithm 1: 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
5      i ← 0
6      while i < N do
7          if j = -1 || Text[i]=P[j] then
8              i++
9              j++
10         end
11         else
12             j ← CarryOver[j]
13         end
14         if j = M then
15             Print(LineNumber et Text a ce ligne)
16             j ← CarryOver[j]
17         end
18     end
19 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

7 Conclusion