Sorbonne Université

Master 2 Science et Technologie du Logiciel

Rapport du projet1 DAAR : Automate

Clone de egrep avec support partiel des ERE

Auteur : Florian CODEBECQ

Auteur : Nandraina RAZAFINDRAIBE

Responsable : Binh-Minh BUI-XUAN

Année scolaire : 2024/2025

1 Introduction

Dans ce projet, nous avons développé un clone partiel de la commande Linux egrep, capable de rechercher des motifs dans un fichier textuel à l'aide d'expressions régulières étendues (ERE). L'objectif principal est d'explorer plusieurs approches d'implémentation des algorithmes de recherche de motifs, et d'analyser leurs performances.

Pour la réalisation du projet, nous avons décidé d'utiliser Java comme langage de programmation.

2 Problématique et Objectifs

Le problème à résoudre est la recherche de motifs dans des fichiers textuels en utilisant des expressions régulières conformes à la norme ERE. Deux stratégies principales sont considérées :

— Automates : Transformation de l'expression régulière en automate fini non déterministe (AFND) avec ϵ-transitions, puis en automate fini déterministe (AFD) via la méthode des sous-ensembles.

— Recherche linéaire : Utilisation de l'algorithme Knuth-Morris-Pratt (KMP) pour la recherche de motifs simples.

3 Méthodologie

3.1 Méthode d'Aho-Ullman

L'algorithme Aho-Ullman est une méthode efficace pour effectuer une recherche simultanée de plusieurs motifs dans un texte.

Cette approche consiste à construire un automate fini à partir d'une expression régulière. Pour parvenir à notre objectif, on doit passer sur plusieurs étapes :

— Construction d'un arbre syntaxique à partir d'**une expression régulière**.

— Conversion de cet arbre en **un automate fini non déterministe (AFND)** avec ϵ-transition.

— Transformer de l'automate fini non déterministe en **un automate fini déterministe (AFD)**.

— Optimiser l'AFD afin d'obtenir un automate avec un nombre minimum d'états.

— Rechercher les motifs avec l'automate ainsi obtenu.

3.1.1 Une expression régulière vers l'arbre syntaxique

Cette étape réalise une analyse syntaxique du motif entré par l'utilisateur en identifiant les opérations à effectuer sur les caractères qui le composent. Les opérations respectent l'ordre de priorité suivant : les parenthèses "()", les quantificateurs "*", la concaténation ".", et l'alternance "|".

Pour ce projet, un code en Java a été fourni dès le départ, qui prend en entrée une expression régulière et retourne l'arbre syntaxique correspondant. Nous avons ensuite basé la suite du projet sur ce code.

3.1.2 Construire l'automate fini non déterministe

Nous devons définir des règles pour chaque opération dans l'arbre syntaxique afin de les combiner et, en fin de compte, générer l'automate correspondant. Pour clarifier ce processus, prenons l'exemple de l'arbre " .(a,b)".

Ci-dessous, nous montrons l'équivalence entre sa représentation en graphe et son équivalent sous forme de tableau. Lors de la transformation en automate, chaque lettre est représentée par un minimum de 4 états.

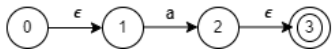

FIGURE 1 – Automate fini non déterministe de "a"
[1]

a	0	1	2	3
0		ϵ		
1			a	
2				ϵ
3				

FIGURE 2 – Représentation sous forme de tableau de "a"
[2]

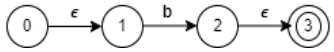

FIGURE 3 – Automate fini non déterministe de "b"
[3]

b	0	1	2	3
0		ϵ		
1			b	
2				ϵ
3				

FIGURE 4 – Représentation sous forme de tableau de "b"
[4]

Cette étape consiste à remonter dans l'arbre tout en combinant les feuilles, en commençant par le niveau le plus profond. Les opérations indiquées sur les nœuds dictent les règles à suivre pour effectuer la combinaison.

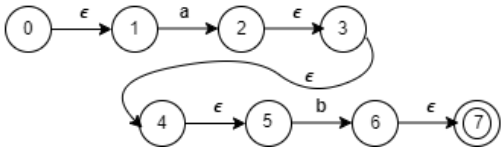

FIGURE 5 – Automate fini non déterministe de l'arbre ".(a,b)"
[5]

Pour la règle de concaténation ".", nous avons ajouté une nouvelle transition ϵ entre l'état 3 (ancien état d'acceptation de l'AFND gauche) et l'état 4 (état de départ de l'AFND droit).

Le tableau suivant correspond à l'AFND du graphe [5] :

Pour les autres opérations, telles que l'étoile "*" et l'alternance "|", l'idée reste similaire, mais les règles diffèrent légèrement. Ces règles sont définies dans des méthodes d'une classe. Des appels de ces méthodes sont ensuite effectués jusqu'à atteindre la racine de l'arbre, afin de générer **un tableau à deux dimensions de caractères** représentant les transitions de l'automate .

3.1.3 Convertir l'automate fini non déterministe en déterministe

À partir de l'AFND, nous cherchons à éliminer toutes les transitions ϵ intermédiaires pour rendre l'automate déterministe en utilisant la méthode des sous-ensembles. Cette méthode commence par définir l'état de départ de l'AFD, qui est lui-même un ensemble d'états de l'AFND. Les états de l'AFD sont stockés dans des structures de type **HashSet** afin de garantir leur **unicité**.

En parcourant les sous-ensembles de tous les états de l'AFD, nous cherchons à déterminer si une transition existe pour chaque caractère d'entrée possible dans l'AFND.

Si c'est le cas, nous continuons le parcours tant que les transitions vers les autres états successeurs sont des transitions ϵ. Ensuite, nous ajoutons les sous-ensembles des états successeurs comme nouveaux états de l'AFD. Le parcours s'arrête lorsqu'il n'y a plus d'états dans l'AFD nécessitant d'être parcourus.

Si l'un des sous-ensembles d'états de l'AFD contient l'état initial de l'AFND, cet état de l'AFD est défini comme état initial. De même, si un sous-ensemble contient l'état d'acceptation de l'AFND, cet état de l'AFD est

a.b	0	1	2	3	4	5	6	7
0		ϵ						
1			a					
2				ϵ				
3					ϵ			
4						ϵ		
5							b	
6								ϵ
7								

FIGURE 6 – Représentation sous forme de tableau de l'arbre ".(a,b)"
[6]

défini comme état d'acceptation.

La structure de données utilisée pour stocker l'AFD est le **HashMap<String, HashMap<String, String>**. Les états de départ et d'acceptation sont stockés avec un **HashMap<String, Boolean>**. Ces choix de structures de données visent à optimiser l'allocation mémoire tout en permettant également une récupération rapide des états et des transitions grâce à son accès en temps constant O(1).

3.1.4 Réduction de l'AFD (Minimisation)

L'optimisation la plus courante pour un AFD est la **minimisation**. Réduire le nombre d'états tout en conservant les mêmes propriétés de reconnaissance du langage. Dans ce projet, on a utilisé un algorithme de partitionnement des états souvent appelé algorithme de Moore.

La première étape consiste à identifier les états atteignables à partir de l'état initial. Cette opération est réalisée en utilisant une exploration en largeur (BFS). Tous les états qui ne peuvent être atteints sont supprimés de l'automate, ce qui simplifie l'AFD en éliminant des états inutiles, rendant ainsi le calcul plus efficace, plus rapide et optimise l'utilisation de la mémoire.

Dans le processus de partitionnement des états équivalents, on commence par diviser les états en deux groupes principaux : les **états finaux** et les **états non-finaux**, qui ne peuvent jamais être équivalents.

Par la suite, chaque groupe est progressivement subdivisé en fonction des transitions. Deux états sont dits équivalents s'ils ont les mêmes transitions vers des groupes identiques d'états. Pour cela, une méthode de signature est utilisée, où chaque état est caractérisé par la "signature" des états vers lesquels il se dirige. Si deux états partagent la même signature, ils sont considérés comme équivalents.

Cette étape se répète jusqu'à ce que les partitions ne puissent plus être subdivisées, assurant ainsi que chaque groupe contient des états totalement équivalents. Une fois les partitions stables identifiées, les états équivalents sont fusionnés. Cela se traduit par la création d'un mapping qui associe chaque ancien état à un état représentant l'ensemble des états équivalents. Cette fusion réduit le nombre total d'états dans l'automate tout en gardant son comportement.

À la fin du processus, un nouvel automate est construit en appliquant le mapping des états équivalents, créant ainsi un **AFD minimisé** qui contient moins d'états mais accepte le même langage que l'automate original.

3.1.5 Recherche de motif

Les étapes de recherche des motifs dans un fichier .txt donnée peuvent être réparties en 3 étapes :

— Une méthode qui prend une ligne de texte à partir d'une position donnée et vérifie si une sous-chaîne est acceptée par l'AFD .Le parcours commence à l'état initial et se déplace selon les transitions de l'automate, vérifiant chaque symbole. Si un état final est atteint, la sous-chaîne est acceptée.

— La méthode qui vérifie dans une ligne parcourt chaque sous-chaîne possible d'une ligne pour trouver des correspondances avec l'automate. Si une sous-chaîne est acceptée, la ligne est validée.

— On parcourt chaque ligne d'un fichier et applique la vérification. Les lignes contenant des sous-chaînes acceptées sont affichées avec leur numéro.

3.2 Algorithme Knuth-Morris-Pratt (KMP)

L'algorithme de Knuth-Morris-Pratt (KMP) est un algorithme de recherche de sous-chaîne dans une chaîne de caractères.

Cet algorithme utilise une technique de prétraitement pour créer un tableau d'indices de décalage (LPS - Longest Prefix Suffix), ce qui permet de sauter des comparaisons inutiles et d'optimiser la recherche. Il est basé sur l'idée de ne pas comparer les caractères qui ont déjà été comparés. Au lieu de cela, il utilise un tableau d'indices de décalage pour sauter les comparaisons inutiles.

Voici la méthodologie du déroulement de l'algorithme KMP :

— Prétraitement du motif : Calculer le tableau LPS du motif. Ce tableau aide à déterminer jusqu'où le motif peut être décalé en cas de désaccord partiel lors de la recherche dans le texte.

— Recherche dans le texte : Parcourir le texte de manière linéaire tout en exploitant le tableau LPS pour éviter de revérifier les caractères qui correspondent déjà.

4 Test et Analyse de performance

5 Analyse critique

La sélection entre l'algorithme KMP et la méthode d'Aho-Ullman dépend des besoins spécifiques de l'application, notamment du nombre de motifs à rechercher et de l'environnement dans lequel l'application sera utilisée.

La méthode Aho-Ullman est rapide dans les contextes où plusieurs motifs doivent être recherchés en une seule passe. Elle est donc très performante dans des scénarios de filtrage, tels que les moteurs de recherche. Cependant, pour atteindre cette performance, l'implémentation de l'algorithme requiert une allocation mémoire conséquente. La construction de l'automate a une complexité algorithmique de $O(m \cdot k)$, où m est la longueur totale des motifs et k est le nombre de motifs. En revanche, la recherche dans le texte est linéaire, avec une complexité de $O(n+z)$, où n est la longueur du texte et z est le nombre de correspondances trouvées. Ainsi, la méthode Aho-Ullman présente une complexité globale de $O(m \cdot k + n + z)$. Pour utiliser cette méthode efficacement, il est nécessaire d'avoir un environnement souple en termes de stockage.

L'algorithme KMP, en revanche, est plus adapté aux applications où le motif recherché est unique. Il est utile, par exemple, pour valider une sous-chaîne dans des données ou analyser des séquences avec des motifs fréquents. Cet algorithme est simple et léger en termes d'espace mémoire. Il présente une complexité linéaire $O(m)$ pour le prétraitement du motif (par rapport à sa longueur m), et une complexité linéaire $O(n)$ pour la recherche dans le texte (par rapport à sa longueur n), pour une complexité totale de $O(n + m)$. KMP est donc idéal dans des contextes où les ressources sont limitées.

Au cours de l'implémentation de ce projet, nous avons constaté que la mise en œuvre de la méthode Aho-Ullman est bien plus complexe que celle de l'algorithme KMP. Dans les contextes où la recherche de motif est effectuée motif par motif, il est préférable d'utiliser KMP.

Pour la méthode Aho-Ullman, nous avons pris en compte uniquement les opérateurs "*", "|", ".", ainsi que les lettres et concaténations. Il serait possible d'étendre le projet en ajoutant d'autres opérateurs d'expressions régulières, tels que "?", "$", "+", etc.

6 Conclusion

Le choix entre KMP et Aho-Ullman dépend d'un compromis entre performance, complexité d'implémentation et contraintes mémoires. Pour des recherches multiples et complexes, Aho-Ullman est un excellent choix. Pour des recherches simples et rapides, KMP est souvent suffisant.

Ce projet nous a permis d'approfondir notre compréhension des algorithmes de recherche de motifs et de mettre en pratique nos connaissances théoriques. La construction de l'automate d'Aho-Ullman nous a familiarisés avec les concepts de théorie des graphes, tandis que l'analyse de la complexité algorithmique de chaque méthode a souligné l'importance du choix des structures de données adaptées.

En somme, ce travail a mis en évidence la richesse et la diversité des approches algorithmiques pour un problème aussi fondamental que la recherche de motifs dans des textes.

Références

[1] Nandraina RAZAFINDRAIBE. Automate fini non déterministe de "a", 2024. Réalisé avec draw.io.

[2] Nandraina RAZAFINDRAIBE. Représentation de l'afnd de "a", 2024. Réalisé avec draw.io.

[3] Nandraina RAZAFINDRAIBE. Automate fini non déterministe de "b", 2024. Réalisé avec draw.io.

[4] Nandraina RAZAFINDRAIBE. Représentation de l'afnd de "b", 2024. Réalisé avec draw.io.

[5] Nandraina RAZAFINDRAIBE. Automate fini non déterministe de "ab", 2024. Réalisé avec draw.io.

[6] Nandraina RAZAFINDRAIBE. Représentation de l'afnd de "ab", 2024. Réalisé avec draw.io.