

Projet DAAR: Clone egrep

TOUNSI Nour, NGUYEN TUONG Léo

8 Octobre 2023

Contents

1	Introduction	3
2	Algorithme de Aho-Ullman	3
2.1	Définition du problème et structure de données utilisée	3
2.2	Analyse des algorithmes connus dans la littérature	3
2.3	Implantation	6
2.4	Résultats et performance	6
3	Algorithme de Knuth-Morris-Pratt	8
3.1	Définition du problème et structure de données utilisée	8
3.2	Analyse des algorithmes connus dans la littérature :	8
3.3	Partie Test :	9
3.4	Rendu :	12
4	Conclusion et perspectives	12

1 Introduction

Ce rapport a pour objectif de décrire le travail réalisé par notre équipe dans le cadre du cours DAAR du Master STL de l'Université Sorbonne Sciences, visant à créer un clone de la commande UNIX `egrep`. Nous discuterons donc des différents algorithmes (Aho-Ullman, Knuth-Morris-Pratt) et structures de données utilisés, ainsi que des méthodes et résultats de tests, de la performance atteinte et enfin des différentes perspectives offertes par ce premier projet.

2 Algorithme de Aho-Ullman

2.1 Définition du problème et structure de données utilisée

Le problème de la recherche de motifs par regex dans un texte consiste à trouver toutes les occurrences d'un motif particulier, spécifié sous forme d'une expression régulière (regex), au sein d'un texte donné. Une expression régulière est un ensemble de règles syntaxiques qui permettent de décrire des modèles de chaînes de caractères complexes. L'objectif principal est de localiser toutes les sous-chaînes du texte qui correspondent au modèle spécifié par l'expression régulière, généralement en identifiant leurs positions de début et de fin.

Ce problème de recherche de motifs par regex est couramment rencontré dans le traitement de texte, l'analyse de données, la recherche d'informations, la validation de données, la recherche de correspondances de motifs dans des fichiers log, etc. Il peut être utilisé pour des tâches telles que l'extraction de données structurées à partir de textes non structurés, la validation de formats de courriels, la recherche de mots-clés dans des documents, et bien d'autres applications.

Afin de répondre à ce problème, nous avons utilisé une variation de la méthode vue en cours, et les structures de données suivantes: des listes ainsi que des dictionnaires ou tables de hachage.

2.2 Analyse des algorithmes connus dans la littérature

L'algorithme vu en cours peut être divisé en 5 étapes: création de l'arbre associé à la regex, transformation de cet arbre en automate non-déterministe, détermination de cet automate et minimisation de cet automate. La partie du code responsable de la création d'arbres à partir des regex étant fournie, nous avons dû réaliser la création d'automates non-déterministes à partir de ces arbres, ainsi que leur détermination. Pour ce faire, nous avons utilisé les méthodes de création d'automates décrites dans le livre *Foundations of Computer Science*, par Al Aho et Jeff Ullman. Ces méthodes sont représentées en figure 1, tirée du livre susmentionné¹. On peut remarquer sur cette figure que le sous-automate nécessaire à toutes les opérations d'expressions régulières

¹*Foundations of Computer Science*, chapter 10, p. 575

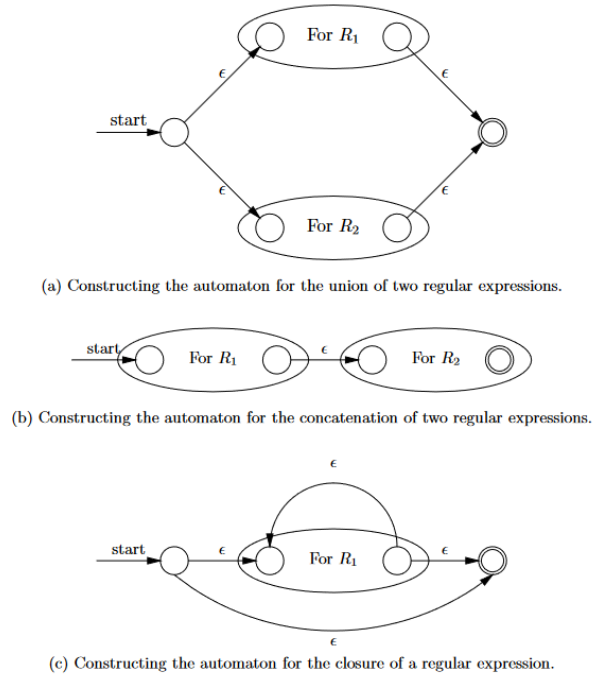


Figure 1: Méthodes de création d'automates non-déterministes

est représenté sous forme abstraite, seuls son état initial et son état final sont représentés. Nous avons donc décidé de suivre la même approche dans notre projet, ainsi la classe Java NDFA ne possède comme attributs que deux états, Initial et Final. Lors de l'ajout d'une opération parmi l'union, la concaténation ou l'étoile, il nous suffit donc de créer les nouveaux états initial et final, de récupérer les états initial et final du sous-automate, puis de créer les transitions nécessaires afin d'obtenir un résultat similaire à la figure 1. Ainsi, lors du parcours de l'arbre obtenu en première étape, lorsqu'on rencontre une lettre, on crée l'automate trivial associé à cette lettre. Puis, on utilise cet automate lorsqu'on atteint l'opérateur associé à cette ou ces lettres, afin d'obtenir le NDFA associé. On continue ainsi jusqu'à atteindre la racine de l'arbre, ce qui construit par induction l'automate désiré.

Détermination du NDFA

L'algorithme vu en cours utilise la méthode dite des sous-ensembles. Cette méthode consiste à créer les états de l'automate déterministe (DFA) désiré à partir d'ensembles d'états de l'automate non-déterministe (NDFA). A chaque transition franchie dans le NDFA, on sauvegarde l'ensemble d'états atteints à cette étape, puis l'on réitère l'opération sur tous les états accessibles depuis cet

ensemble d'états. Lors de notre projet, nous avons opté pour une variation récursive de cette approche, qui est la suivante :

```

Data: Automate non-déterministe ndfa
Result: Automate déterministe associé
first ← ndfa.initial;
etatsCombines ← ajouterEpsilonReachable(first);
// suppression des epsilon-transitions
dfaInitial ← newState;
dfa ← newDFA;
dfa.etats[0] ← dfaInitial;
if etatsVisites.contient(etatsCombines) then
    | dfa.etats[0] ← etatsVisites[etatsCombines];
    | return dfa;
end
keys ← {} // Créer une table de hachage pour stocker les
            transitions sortantes
foreach état s dans etatsCombines do
    // Suppression des transitions doublons
    foreach clé key dans les clés de s.successeurs do
        if keys contient key then
            | nouveauPredecesseur ← keys[key];
            | nouveauPredecesseur.epsilonReachable.ajouter(s.successeurs[key]);

            | s.successeurs.supprimer[key];
            | // on supprime la transition doublon, on ajoute une
            |   eps-transition depuis le successeur déjà pointé
            |   par la clé vers le nouvel état
        end
        else
            | keys.ajouter(key, s.successeurs[key]);
        end
    end
end

```

```

foreach état s dans etatsCombines do
    if s.estInitial then
        | dfaInitial.estInitial  $\leftarrow$  true;
    end
    if s.estFinal then
        | dfaInitial.estFinal  $\leftarrow$  true;
    end
    foreach clé key dans s.successeurs.clés do
        if etatsCombines contient s.successeurs[key] then
            | dfaInitial.successeurs.ajouter(key, dfaInitial);
            continuer;
        end
        ajouter s à etatsVisites;
        sousAutomate  $\leftarrow$  créerDFA(créerNdfa(s.successeurs[clé])) ;
        // Appel récursif de cet algorithme pour obtenir le
        sous-automate ayant le successeur pointé par la clé
        comme état initial
        Ajouter tous les états de sousAutomate aux états de dfa;
    end
end
return dfa;

```

Algorithm 1: Détermination par récursion du NDFA

Ce pseudocode décrit l'algorithme de création du DFA, en éliminant tout d'abord les transitions epsilon et doublons, puis en faisant un appel récursif sur chaque état accessible depuis l'état initial. Cet appel récursif assure que toutes les sous-parties de l'automate retourné sont déterministes, et donc que le résultat global est lui aussi déterministe.

2.3 Implantation

Pour l'implantation de notre algorithme, nous avons utilisé le langage Java, avec les structures de données `ArrayList` et `HashMap`. L'utilisation des `HashMap` permet d'obtenir un temps constant pour la recherche et l'insertion, et nous a servi à représenter les transitions non epsilon, avec le caractère comme clé et l'état successeur comme valeur. Nous avons aussi défini une `ArrayList` `epsilonReachable` dans la classe `State` afin de représenter les états accessibles via epsilon-transition. Enfin, nous avons défini dans la classe `DFA` une `ArrayList` statique `visitedStates` permettant de stocker les états déjà visités lors de la récursion afin d'empêcher un bouclage récursif infini, plus couramment appelé `Stack Overflow`.

2.4 Résultats et performance

Voici en figure 2 le résultat de la commande `time java -jar egrep.jar "S(a||r||g)+on" sargon.txt`. Le programme affiche un résultat similaire à la commande `egrep`,

```

431:11      state--Sargon and Merodach-baladan--Sennacherib's attempt
435:14      under the Sargonids--The policies of encouragement and
948:59 that empire's expansion, and the vacillating policy of the Sargonids
1015:3 to Sargon of Akkad; but that marked the extreme limit of Babylonian
1018:54 Arabian coast. The fact that two thousand years later Sargon of
1787:3 A: Sargon's quay-wall. B: Older moat-wall. C: Later moat-wall of
1819:18 It is the work of Sargon of Assyria,[44] who states the object of
1826:31 upon it."[45] The two walls of Sargon, which he here definitely names
1831:12 the quay of Sargon,[46] which run from the old bank of the Euphrates
1832:58 to the Ishtar Gate, precisely the two points mentioned in Sargon's
1843:3 A: Sargon's quay-wall. B: Older moat-wall. O: Later moat-wall of
1852:36 quay-walls, which succeeded that of Sargon. The three narrow walls
1867:0 Sargon's earlier structure. That the less important Nimitti-Bêl is not
1869:11 in view of Sargon's earlier reference.
1913:30 excavations. The discovery of Sargon's inscriptions proved that in
1918:26 precisely the same way as Sargon refers to the Euphrates. The simplest
3547:30 [Footnote 44: It was built by Sargon within the last five years of
5554:0 Sargon of Akkad had already marched in their raid to the Mediterranean
6064:56 Babylonian tradition as the most notable achievement of Sargon's reign;
8956:4 for Sargon's invasion of Syria. In the late omen-literature, too, the
10919:0 Sargon's army had secured the capture of Samaria, he was obliged to
10929:0 Sargon and the Assyrian army before its walls. Merodach-baladan was
10936:57 After the defeat of Shabaka and the Egyptians at Raphia, Sargon was
10940:51 their appearance from the north and east. In fact, Sargon's conquest of
10945:0 Sargon was able to turn his attention once more to Babylon, from
10953:3 On Sargon's death in 705 B.C. the subject provinces of the empire
11000:38 party, whose support his grandfather, Sargon, had secured.[43] In 668
11243:0 Sargon's death formed a period of interregnum, though the Kings' List
12452:42 fifteen hundred years before the birth of Sargon I., who is supposed

real      0m0,387s
user      0m0,893s
sys       0m0,069s

```

Figure 2: Résultat du programme avec la regex $S(a||r||g) + on$

avec en supplément les numéros de lignes et de colonnes correspondant au motif retourné par l'automate.

3 Algorithme de Knuth-Morris-Pratt

3.1 Définition du problème et structure de données utilisée

L'algorithme naïf de recherche de motif dans une chaîne de texte consiste à examiner toutes les positions possibles du texte pour vérifier si le motif s'y trouve. Bien que cet algorithme soit simple à comprendre et à implémenter, il peut devenir inefficace pour de longues chaînes de texte ou de grands motifs. Avec cet algorithme nous sommes contraint à une complexité temporelle en $O((n-m+1)*m)$ avec n la longueur du texte et m la longueur du motif. Une performance médiocre et un manque d'efficacité pour la recherche de motifs multiples. L'objectif de l'algorithme KMP est de localiser toutes les occurrences d'un motif spécifié dans une chaîne de texte. En d'autres termes, il s'agit de repérer les emplacements précis, ou indices, au sein de la chaîne de texte où le motif apparaît. Pour résoudre ce problème nous avons utilisé l'algorithme de Knuth-Morris-Pratt (KMP) vu en cours. L'algorithme KMP utilise une structure de données appelée "table des décalages". Dans notre code nous avons utilisé un tableau de la même taille que le pattern.

Nous avons également utilisé des indices d'occurrences, Une fois que nous avons localisé des occurrences du motif dans le texte, nous procédons à la mise à jour de notre tableau de préfixes les plus longs qui sont également des suffixes (LPS). Ensuite, nous utilisons ce LPS mis à jour pour récupérer les indices dans le texte où le LPS correspond à la longueur totale de notre motif. Cette approche nous permet d'efficacement repérer toutes les positions du texte où le motif se trouve, tout en optimisant la recherche grâce à notre tableau de préfixes mis à jour.

3.2 Analyse des algorithmes connus dans la littérature :

L'algorithme vu en cours commence par précalculer la fonction p , qui à une position j dans le texte nouvellement lu, associe la longueur du bord maximal du texte jusqu'à cette position. Ensuite, il examine chaque lettre du texte, de la première à la dernière, et regarde la valeur de la fonction p à chaque étape. Si la fonction p atteint la longueur du motif, alors une occurrence du motif est trouvée dans le texte, et la recherche réussit.

Bien que cela puisse sembler similaire à une recherche naïve, l'algorithme de Morris-Pratt est plus efficace car il utilise les informations sur les bords maximaux précédemment calculés pour déterminer rapidement la longueur du bord à la position actuelle. Cela évite de répéter des comparaisons inutiles avec le motif, ce qui améliore considérablement les performances de la recherche.

L'algorithme nécessite également le calcul préliminaire d'un automate spécifique pour aider à la recherche. Cette technique permet de rendre la recherche encore plus efficace en évitant des comparaisons redondantes.

En résumé, l'algorithme de Morris-Pratt, avec l'amélioration de Knuth, est une méthode efficace pour rechercher un motif dans un texte en utilisant la notion de bord maximal et la fonction p pour déterminer rapidement si le motif apparaît dans le texte.

Parmi les différents algorithmes de recherche de motifs dans une chaîne de texte décrits dans la littérature, l’Algorithme de Boyer-Moore se distingue par son approche consistant à examiner les caractères du motif en commençant par la fin, ce qui réduit les comparaisons inutiles. Il repose sur l’utilisation de deux tables clés : la table des décalages de caractères et la table des décalages de préfixe. Ces tables permettent de sauter certains caractères en cas de non-correspondance, ce qui améliore considérablement l’efficacité de l’algorithme. Cette efficacité est particulièrement remarquable lorsque le motif à rechercher est de grande longueur.

L’Algorithme de Boyer-Moore offre généralement des performances supérieures, avec une complexité moyenne en $O(n/m)$ dans le pire des cas, par rapport à une complexité de $O(n+m)$ pour l’algorithme de Knuth-Morris-Pratt (KMP).

3.3 Partie Test :

Dans le cadre de ce projet, nous avons utilisé JUnit, un framework dédié aux tests unitaires pour le langage de programmation Java. Notre démarche a consisté à obtenir un jeu de données adapté pour évaluer notre algorithme dans différents contextes.

En tant que plateformes de test, nous avons exploité la base de données Gutenberg en utilisant deux méthodes distinctes. La première approche consistait à rechercher le fichier et le motif localement. La deuxième possibilité consistait à extraire les données en utilisant des requêtes HTTP, en utilisant à la fois la fonction Curl et l’algorithme de Knuth-Morris-Pratt (KMP).

Nous avons effectué plusieurs types de tests, notamment :

Test Unitaires : Ces tests évaluent individuellement deux unités de code.

Tests de performance : Nous avons mesuré la rapidité du test à traiter la demande de l’utilisateur.

Tests des cas limites : Nous avons testé des motifs très courts (dans notre test, nous avons choisi la lettre "a") ainsi que des motifs très longs. Pour ce dernier cas, nous avons dû nous limiter à 65535 octets. En Java, le pool constant est une table de structures utilisée pour stocker des valeurs constantes telles que des chaînes, des nombres et des références de classes, et il existe une limite à la taille d’une seule entrée du pool constant, qui est de 65535 octets. Par conséquent, nous nous sommes limités à ce nombre d’octets pour effectuer notre test.

Tests de correspondance multiple : Nous avons testé l’algorithme sur des fichiers de test contenant plusieurs occurrences du motif recherché et vérifié que toutes les occurrences étaient correctement identifiées.

Test de correspondance null : Nous avons choisi un motif qui ne correspondait pas au texte pour tester cette condition spécifique.

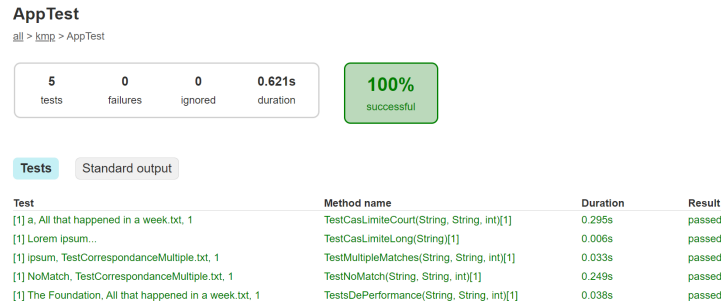


Figure 3: Résultat des tests

En observant les données du tableau précédemment affiché, il est possible de conclure que les tests avec des cas limites courts sont ceux qui présentent la durée d'exécution la plus élevée en raison de l'overhead initial, de moins d'opportunités d'optimisation, des caractéristiques des données d'entrée et de la constante associée à la complexité algorithmique. Le test "no match" est souvent long aussi en termes de durée d'exécution, car il nécessite que l'algorithme parcoure tout le texte sans trouver de motif correspondant, ce qui implique des comparaisons avec chaque caractère du texte. Le Test le plus rapide est le test des cas limites longs, en raison de la capacité de l'algorithme à éviter efficacement les comparaisons inutiles grâce à la table de préfixe, notamment lorsque le motif est long et que le texte ne contient pas de répétitions fréquentes du motif. Et pour finir nous avons lancé l'algorithme KMP plusieurs fois avec des données simples pour tester la variabilité et la stabilité de notre programme et les résultats varient entre 0.029 et 0.066 secondes.

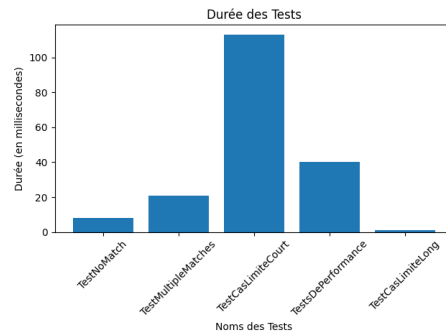
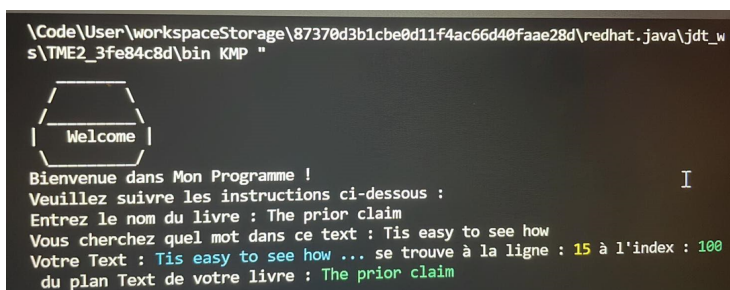


Figure 4: Durée des différents tests

Nous avons développé un programme en Python spécialement conçu pour créer des graphiques à barres afin de faciliter l'analyse des performances de nos tests. Lors de l'exécution des tests, nous recueillons des données sur les temps d'exécution, que nous transformons ensuite en format YAML. Par la suite, notre programme Python lit ces données pour générer des graphiques, offrant ainsi une visualisation claire des performances de nos tests. Sur la figure 4 on voit bien la variation des temps d'exécution selon le type de test exécuté. Lorsque nous organisons nos tests en fonction de leurs temps d'exécution, nous observons une séquence distincte. En dernier lieu, nous trouvons le test des cas limites courts, suivi du test de performance avec une variation de seulement 0.03 secondes entre tous les tests. Ensuite, nous rencontrons le test à multiples correspondances, suivi du test "no match". Finalement, le test des cas limites longs s'avère être le plus rapide de tous. Cette organisation des tests selon leurs temps d'exécution offre une perspective claire de la performance relative de chacun d'entre eux.

3.4 Rendu :



```
\Code\User\workspaceStorage\87370d3b1cbe0d11f4ac66d40faae28d\redhat.java\jdt_ws\TME2_3fe84c8d\bin KMP "
Welcome
Bienvenue dans Mon Programme !
Veuillez suivre les instructions ci-dessous :
Entrez le nom du livre : The prior claim
Vous cherchez quel mot dans ce text : Tis easy to see how
Votre Text : Tis easy to see how ... se trouve à la ligne : 15 à l'index : 100
du plan Text de votre livre : The prior claim
```

Figure 5: Résultat du programme

Ici, nous présentons les résultats de la recherche du texte 'Tis easy to see how' dans le livre 'The prior claim'. Comme mentionné précédemment, nous avons également développé une fonction nommée 'HTTPRequest' qui permet de récupérer des données depuis le site web fourni dans ce projet en utilisant une combinaison des fonctions CURL et l'algorithme KMP. Cependant, nous n'avons pas inclus son appel dans la fonction principale (main) parce que ceci n'était pas explicitement demandé dans le rendu. Vous pouvez consulter cette fonction dans le code source fourni.

4 Conclusion et perspectives

En conclusion, ce projet nous aura permis de mieux comprendre les différents algorithmes servant à répondre au cas du moteur de recherche *offline*. Nous avons pu acquérir des connaissances dans ce domaine qui pourront s'avérer utiles dans la conception et la réalisation de nombreux logiciels, notamment le projet final de l'UE DAAR. Une perspective intéressante est la portabilité de notre projet liée au langage Java, ce qui permettrait de réutiliser notre code dans un grand nombre d'environnements différents sans recourir à la conteneurisation. De même, les algorithmes Aho-Ullman et Knuth-Morris-Pratt sont des bases utiles et constituent une entrée dans le domaine des moteurs de recherche que nous avons trouvé pertinente et intéressante.