06/10/2024

Master Science et Technologie Logiciel (STL)

Rapport de projet N° 1 : DAAR

EGREP

Binôme :

Ghita Mikou 21423710

Zhengdao Yu 21304260

Enseignant :

Binh-Minh Bui-Xuan

Table des matières

1- Introduction

1-1 But de projet

Le but de ce projet est de proposer un clone de commande ***egrep.***

1-2 Stratégie de recherche

Nous avons implémenté 2 stratégies de recherche dans ce projet:

1. le motif n'est pas réduit à une suite de concaténations:
   (0)Transformer le motif en un arbre de syntaxe
   (1) puis en un automate fini non-déterministe avec e-transitions
   (2) puis en un automate fini déterministe avec la méthode des sous-ensembles
   (3) puis en un automate équivalent avec un nombre minimum d'états
   (4) et enfin l'automate est utilisé pour tester si un suffixe d'une ligne du fichier textuel donné initialement est reconnaissable par cet automate

2. Si le motif est réduit à une suite de concaténations:
   Il s'agit de la recherche d'un facteur dans une chaîne de caractères : l'algorithme Knuth-Morris-Pratt (KMP)

2- Etat d'art

2-1 Expression Régulière (RegEx)

Une expression régulière est une représentation d'un langage algébrique reconnaissable par un automate fini déterministe, dans le cadre de la théorie des langages formels.

Les expressions régulières sont une syntaxe indépendante du contexte, capable de représenter une grande variété d'ensembles de caractères et de séquences de ces ensembles, où ces caractères sont interprétés en fonction du paramètre local. Bien que certaines expressions régulières puissent être interprétées différemment selon les paramètres locaux, de nombreuses fonctionnalités, telles que les classes de caractères, garantissent une invariance contextuelle à travers différents locaux.

Nous nous limitons aux opérateurs suivants concernant les expressions régulières (RegEx) :

- Parenthèses : **()**
- Alternation : **|**
- Concaténation
- Étoile : *****
- Dot : **.**
- Lettres

2-2 Algorithme Knuth-Morris-Pratt (KMP)

L'algorithme de Knuth-Morris-Pratt **(KMP)** est un algorithme de recherche de motifs dans un texte qui améliore l'efficacité en évitant les comparaisons inutiles. Il utilise un tableau auxiliaire appelé **LPS** (Longest Prefix Suffix) pour déterminer les parties du motif qui peuvent être réutilisées lors de la recherche. Ainsi, lorsqu'une correspondance échoue, l'algorithme utilise ce tableau pour sauter certaines comparaisons et continuer efficacement.
Sa complexité est linéaire, **O(n + m)**, où n est la taille du texte et m celle du motif, ce qui en fait une solution bien plus rapide que les approches naïves.

Algorithm 1 KMP Algorithm

```
1:  procedure KMPALGO(text, pattern)
2:      t ← length of text
3:      m ← length of pattern
4:      lps ← array of size m
5:      j ← 0                                          ▷ index for pattern
6:      COMPUTELPS(pattern, m, lps)
7:      i ← 0                                          ▷ index for text
8:      while i < t do
9:          if pattern[j] = text[i] then
10:             j ← j + 1
11:             i ← i + 1
12:         end if
13:         if j = m then
14:             output occurrence at i − j
15:             j ← lps[j − 1]
16:         else if i < t and pattern[j] ≠ text[i] then
17:             if j ≠ 0 then
18:                 j ← lps[j − 1]
19:             else
20:                 i ← i + 1
21:             end if
22:         end if
23:     end while
24: end procedure
```

Algorithm 2 Compute LPS Array

```
1:  procedure COMPUTELPS(pattern, m, lps)
2:      len ← 0
3:      lps[0] ← 0
4:      i ← 1
5:      while i < m do
6:          if pattern[i] = pattern[len] then
7:              len ← len + 1
8:              lps[i] ← len
9:              i ← i + 1
10:         else
11:             if len ≠ 0 then
12:                 len ← lps[len − 1]
13:             else
14:                 lps[i] ← 0
15:                 i ← i + 1
16:             end if
17:         end if
18:     end while
19: end procedure
```

3- Implementation

3-1 Stratégie 1: Automate

- ***Structure de donnée***

```
class DFA_State {
private int id;
Map<Character, DFA_State> transitions;
private Set<State> subStates;  //sous-ensemble
protected boolean isFinal = false; //si c'est une etat final
}

class DFA{
protected DFA_State debut_State;
protected Set<DFA_State> final_States;
private Map<Set<State>,DFA_State> dfa_registre = new HashMap<>();
}
```

- ***Création du automate(NFA) - logique de traitement***

Alternation (|)

- Créer un nouvel état initial
- Ajouter des transitions ε vers les états initiaux des deux sous-automates
- Créer un nouvel état final
- Ajouter des transitions ε des états finaux des sous-automates vers le nouvel état final

Concaténation

- Relier l'état final du premier sous-automate à l'état initial du second par une transition ε
- L'état initial du résultat est celui du premier sous-automate
- L'état final du résultat est celui du second sous-automate

Étoile de Kleene (*)

- Créer un nouvel état initial et un nouvel état final
- Ajouter une transition ε du nouvel état initial vers l'état initial du sous-automate
- Ajouter une transition ε du nouvel état initial vers le nouvel état final
- Ajouter des transitions ε des états finaux du sous-automate vers son état initial et vers le nouvel état final

Caractère simple ou point (.)

- Créer un état initial et un état final
- Ajouter une transition étiquetée par le caractère (ou tout caractère pour le point) entre ces deux états

- ***Construction du DFA (Méthode : Sous-ensemble)***

Création de l'état initial du DFA

- Calculer l'ε-fermeture de l'état initial du NFA
- Créer un nouvel état DFA correspondant à cet ensemble d'états NFA

Traitement itératif des états DFA

- Utiliser une liste de travail pour traiter les nouveaux états DFA
- Pour chaque état DFA : a. Calculer les transitions possibles pour chaque symbole d'entrée b. Pour chaque transition, créer un nouvel état DFA si nécessaire c. Ajouter les nouveaux états à la liste de travail

Calcul de l'ε-fermeture

- Pour chaque ensemble d'états NFA, calculer tous les états accessibles par ε-transitions
- Utiliser une pile pour explorer récursivement les ε-transitions

Gestion des états finaux

- Un état DFA est final si l'un de ses états NFA sous-jacents est final

Optimisation avec un registre

- Utiliser un registre (Map) pour éviter de créer des états DFA en double
- Chaque ensemble unique d'états NFA correspond à un seul état DFA

Construction des transitions du DFA

- Pour chaque symbole d'entrée, regrouper les transitions des états NFA sous-jacents
- Créer une transition DFA vers l'état correspondant à l'ensemble des états d'arrivée

- ***Minimisation du DFA(Méthode naïve du Acho et Ullman)***

Ajout d'un état mort

- Ajouter un état mort au DFA pour gérer toutes les transitions manquantes
- Cet état a des transitions vers lui-même pour tous les symboles d'entrée

Initialisation de la table d'association

- Créer une table pour tous les pairs d'états
- Marquer comme distinguables les pairs où un état est final et l'autre ne l'est pas

Remplissage de la table d'association

- Itérer jusqu'à ce qu'aucun changement ne se produise
- Pour chaque paire non distinguable, vérifier si leurs transitions mènent à des états distinguables
- Si oui, marquer la paire comme distinguable

Création des classes d'équivalence

- Regrouper les états non distinguables en classes d'équivalence

Construction du DFA minimal

- Créer un nouvel état pour chaque classe d'équivalence
- Définir les transitions entre ces nouveaux états en se basant sur les transitions du DFA original
- Déterminer le nouvel état initial et les nouveaux états finaux

Nettoyage

- Supprimer les transitions vers l'état mort pour rendre l'automate plus propre

3-1-1 Test: solution de visualisation: .dot

Un fichier .dot est un fichier texte qui utilise le langage DOT pour décrire des graphes. Il permet de représenter visuellement des structures de données comme des automates ou des organigrammes.

Pour faciliter le test et comprendre le résultat du code,nous avons donc choisi le moyen de print le fichier .dot et générer les graphes. Voici une exemple de .dot dans ce projet:

Example Pattern: a|bc*

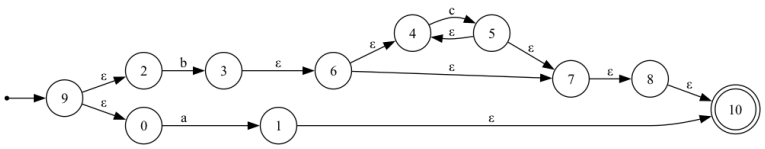
Text dot génère pour NFA:

```
digraph Automate {
  rankdir=LR;
  node [shape=circle];
  start [shape=point];
  start -> 9;
  10 [shape=doublecircle];
  9 -> 2 [label="ε"];
  9 -> 0 [label="ε"];
  2 -> 3 [label="b"];
  0 -> 1 [label="a"];
  3 -> 6 [label="ε"];
  1 -> 10 [label="ε"];
  6 -> 4 [label="ε"];
  6 -> 7 [label="ε"];
  4 -> 5 [label="c"];
  7 -> 8 [label="ε"];
```

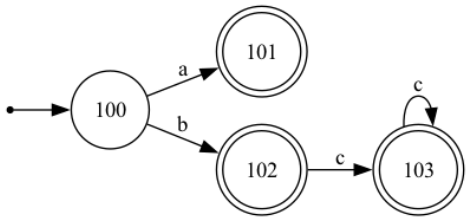
```
  5 -> 4 [label="ε"];
  5 -> 7 [label="ε"];
  8 -> 10 [label="ε"];
}
```

Tree RegEx:
|(a,.(b,*(c))).

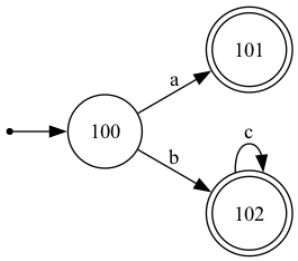
NFA:



DFA:



DFA Minimisation:



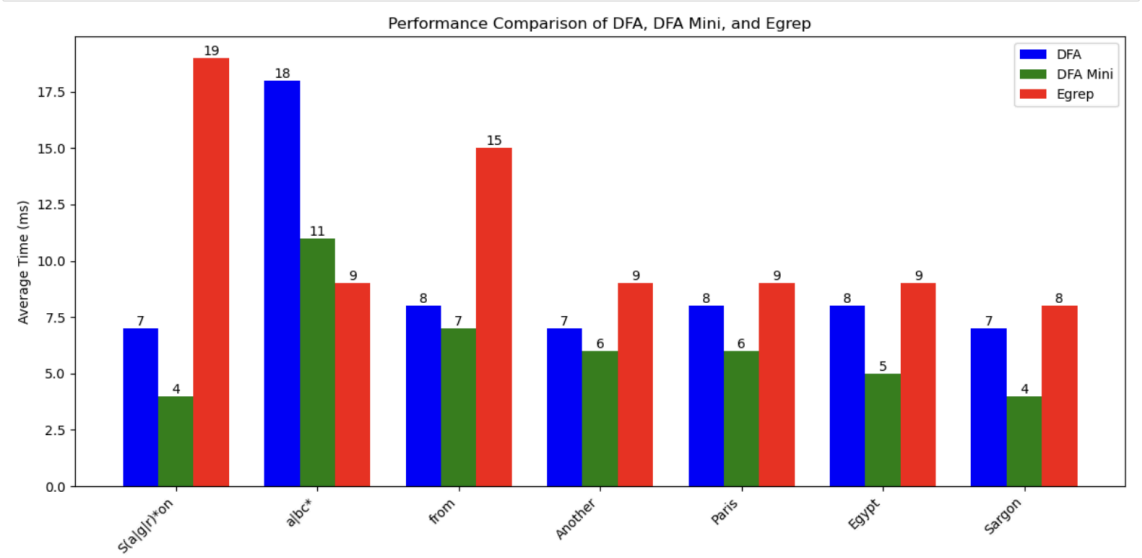
3-1-2 Test: Validation du résultat

À la fin de l'exécution du code, nous utilisons egrep pour vérifier la validité de nos résultats. C'est-à-dire que dans le code, nous comparons le nombre de lignes correspondant au motif trouvées par DFA, DFA mini, et egrep. Une fois qu'ils sont égaux, nous avons confirmé la validité des résultats :

```
if (
matchedLines_DFA_mini == matchedLines_DFA
&& matchedLines_DFA == matchedLines_egrep
)
{
System.out.println(">>>>>>>>>>>>>>>>>>>>Résultat<<<<<<<<<<<<<<<<<<<<");
System.out.println(">>>>>>>>>Bravo! Les résultats sont corrects");
}
```

4- Performance

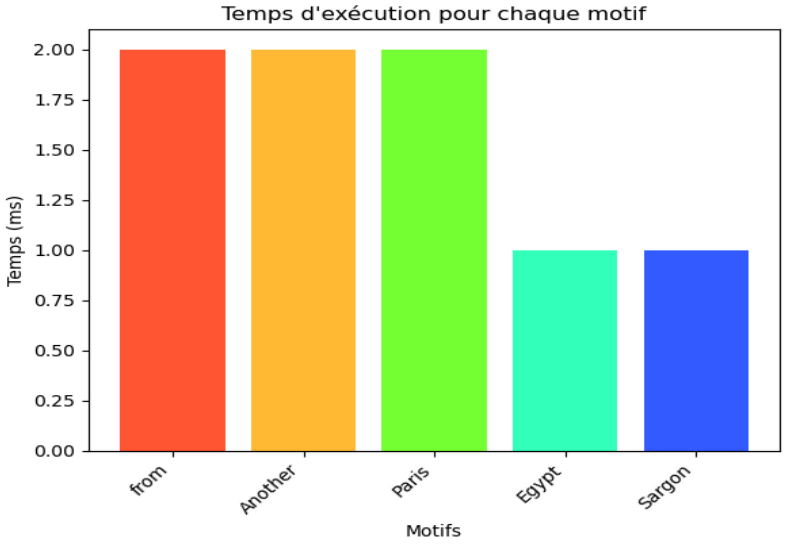
4-1 Performance de Automate



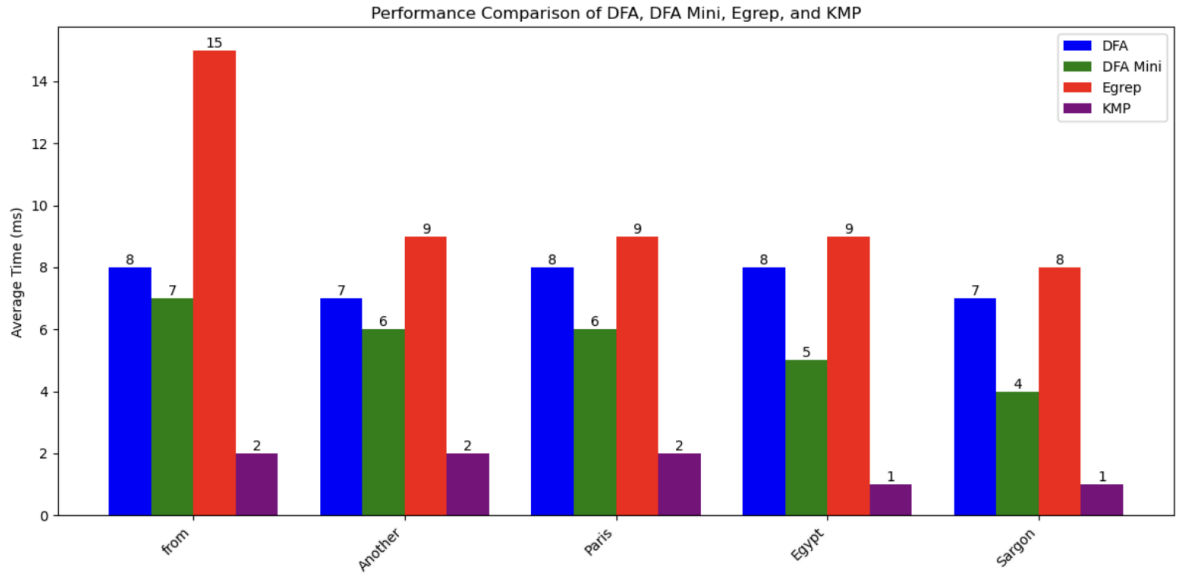
Ce graphique compare les performances (c'est-à-dire le temps de calcul) de trois méthodes différentes : DFA, DFA minimisé et egrep. Nous avons utilisé des motifs tels que "a|bc*", "from" et "another" …etc pour effectuer nos tests.

Nous pouvons clairement voir que le DFA optimisé, c'est-à-dire le DFA minimisé, offre les meilleures performances. En réduisant les états superflus de l'automate, on peut améliorer significativement les performances.

4-2 Performance de KMP



4-3 Comparaison de la performance:



4-4 Conclusion des résultats

À la fin, nous comparons les performances de deux stratégies sur le même motif.

L'écart est significatif : si le motif peut être réduit à une séquence de concaténations, l'algorithme KMP offre les meilleures performances, réduisant considérablement le temps de calcul par rapport à la minimisation DFA, qui présente également un temps de calcul beaucoup plus élevé.

Cependant, le problème avec KMP est également évident : il ne peut être appliqué que lorsque le motif peut être réduit à une séquence de concaténations.

6- Conclusion et Perspectives

Sur la base de ces deux stratégies, et ces résultats que nous avons obtenue,nous concluons que :

- **Performance et applicabilité** : La transformation via automates non déterministes offre une capacité de recherche étendue pour les motifs complexes, au prix d'une charge calculatoire élevée. En revanche, l'algorithme KMP est très efficace pour les motifs simples, mais moins flexible pour les motifs complexes.
- **Axes de recherche** : Les travaux futurs pourraient viser à combiner les avantages des deux stratégies pour créer un nouvel algorithme, efficace et capable de gérer des motifs complexes. Il serait également pertinent d'explorer l'adaptabilité et l'optimisation des algorithmes sur divers types de données textuelles pour améliorer leur utilité pratique.
- **Applications technologiques** : L'amélioration de ces stratégies est cruciale pour le développement des technologies de moteurs de recherche hors ligne, en particulier pour le traitement de grandes quantités de données textuelles et de requêtes complexes, augmentant ainsi l'efficacité de la gestion et de la recherche d'informations.

7- Référence

[AU92] Al Aho et Jeff Ullman. Foundations of Computer Science (FCS). 1992. Chap. Chapter 10 Patterns, Automata, and Regular Expressions, p. 529-590.

[open group] *https://pubs.opengroup.org/onlinepubs/7908799/xbd/re.html*