Rapport du projet 1 de DAAR

Clone of the egrep UNIX command

Rédacteurs :
Qiwei XIAN
Mehdi-Nassim KHODJA

Professeur :
Prof.Bui-Xuan

13 novembre 2020

Table des matières

1 Préface **2**
- 1.1 Objectif . . . 2
- 1.2 Définition . . . 2
  - 1.2.1 Définition des termes utilisés . . . 2
  - 1.2.2 Définition des structure de données . . . 2

2 Algorithme **2**
- 2.1 Transformation de RET à NFA . . . 2
- 2.2 Transformation de NFA à DFA . . . 3
  - 2.2.1 Création des noeuds nouvelle pour DFA . . . 3
  - 2.2.2 Addition des liens dans DFA . . . 3
- 2.3 Minimization de DFA : Hopcroft . . . 3
  - 2.3.1 Classification des noeuds . . . 3
  - 2.3.2 Construction DFA minimal . . . 4
- 2.4 Réalisation de commande egrep . . . 4
- 2.5 Figures . . . 4

3 Analyse de complexité **5**
- 3.1 Cas général . . . 5
- 3.2 Cas particulière . . . 5

4 Partie de test **5**

5 Test de performance **5**

6 conclusion **6**

1 Préface

1.1 Objectif

L'objectif du projet est de chercher tous les lignes qui contiennent les mots correspondant à l'expression régulière saisie. Nous devons comprendre comment transformer l'expression régulière à l'automate fini déterministe minimale correspondante.

1.2 Définition

1.2.1 Définition des termes utilisés

1. RET (Regular Expression Tree) : l'arbre de l'expression régulière.
2. DFA (Deterministic finite automaton) : l'automate fini déterministe
3. NFA (Nondeterministic finite automaton) : l'automate fini non déterministic

1.2.2 Définition des structure de données

1. RegExTree : Il représent chaque noeud de l'arbre de l'expression régulière, l'attribut root est l'opérator ou une lettre, subTrees est une liste des noeuds fils.
2. Node : Cette structure de données représente chaque noeud de l'automate, il contient un hashmap qui permet de stocker les étiquettes des lien ainsi que les voisins.
3. Automaton : Cette structure possède le noeud initial et une liste des noeuds.

2 Algorithme

Pour construire un DFA minimale par RET, il y a 3 étapes : La transformation de RET à NFA, celle de NFA à DFA, la minimization de DFA.

2.1 Transformation de RET à NFA

On d'abord observe la strucutre de RET, on peut savoir que le noeud représente l'opérator si et seulement si c'est un noeud interne, sinon il représente une lettre.
Donc on parcourt RET en suffix, on arrive d'abord les noeuds feuilles et construit l'automate de base, comme l'image1. Ensuite on arrive les noeuds internes et on connecte l'automate de fils gauche et de fils droite par le lien epsilon en fonction de l'opérator actuel. Il y a 2 formes différente de connection, la concaténation et l'alternance, l'image2. En plus une connection spéciale pour l'opérateur étoile. Le noeud interne dont root est l'étoile ne possède qu'un noeud fils dans ce cas, comme l'image4.
Une fois qu'on fait cette étape, on peut obtenir un NFA avec des lien epsilon. Pour les codes de source, la réalisation de cette étape est la fonction $parse(RegExTreeregTree)$.

2.2 Transformation de NFA à DFA

Dans cette étape, on élimine les liens epsilon et mélanger les noeuds qui sont dans le même groupe. On note que l'ensemble de noeuds $\mathcal{P}$.

2.2.1 Création des noeuds nouvelle pour DFA

L'idée est simple, on cherche d'abord les noeuds qui ne reçoivent pas le lien d'epsilon dans $\mathcal{P}$ et on note ces noeuds S. Par exemple, pour NFA5, les noeuds 0,2,5 et 8 qui ne reçoivent pas de lien epsilon.
Pour chaque noeud de S, on crée un nouveau noeud. On va obtenir une nouvelle ensemble S', on construit un hashmap qui permet de trouver noeud de S' par celui de S à la fois, on note $hashmap[s] = s', s \in S, s' \in S'$. DFA sera composé par les noeuds de S'.

2.2.2 Addition des liens dans DFA

Ensuite, pour chaque noeud $s, s \in S$, on fait un DFS, lors qu'on arrive un autre noeud $s', s' \in S$ via le lien l, on ajoute le lien l entre le noeud $hashmap[s]$ et $hashmap[s']$. Si on arrive une noeud final par un lien epsilon, on marque que $hashmap[s]$ est aussi final.

Après cette étape, tous les liens non epsilon sont ajoutés dans DFA. Pour nos codes de source, la réalisation de cet algorithme est la fonction $transformNFAToDFA(AutomatonNFA)$.

2.3 Minimization de DFA : Hopcroft

On a utilisé l'algorithme de Hopcroft pour minimiser DFA actuel. L'idée de minimization est classifier les noeuds en m groupes ($m < n, n == |\mathcal{S}|$), jusqu'à on ne peut plus continuer de séparer. Ensuite on connecte ces m groupes, alors DFA est minimizé.

2.3.1 Classification des noeuds

La principe de classification est mettre les noeuds équivalents dans le même groupe. Si les successeurs de deux noeuds sont dans le même groupe, alors on dit que ces deux noeuds sont équivalent.
On sépare initialement $\mathcal{S}$ en deux groupes, A pour les noeuds acceptables, B pour ceux non acceptables. Pour chaque noeud de groupe, on vérifie si ses successeurs sont dans le groupe où il se trouve, sinon on met ce noeud dans un nouveau groupe. Après plusieur tours de séparation, on ne peut plus continuer à séparer, la classification est terminée.
La réalisation de cette étape est la fonction $classifyNode$.

2.3.2 Construction DFA minimal

Une fois qu'on fait la classification, il y a m groupes. On crée un nouveau noeud pour chaque groupe, et note eux M. DFA minimal se compose des noeuds de M. Le hashmap h est aussi construit, on déclare $h[s] = m, m \in M, s \in S$.
On parcourt DFA précédent par BFS, pour le noeud actuel *cur*, si le successeur de *cur* (on note *suc*) appartient à un autre groupe, alors on connecte $h[cur], h[suc]$ par le lien entre *cur* et *suc*. Sinon on continue le processus de BFS. Après BFS, DFA minimal est bien complété.
La réalisation de cette étape est la fonction $minimize(AutomatonDFA)$.

2.4 Réalisation de commande egrep

Pour cette étape, on applique simplement le générateur de DFA minimal dans la copie de egrep. Pour chaque ligne de l'article. On appelle une fois $DFA.search(ligne)$. Si il peut arriver le noeud acceptable de DFA, alors on rend cette ligne.

2.5 Figures

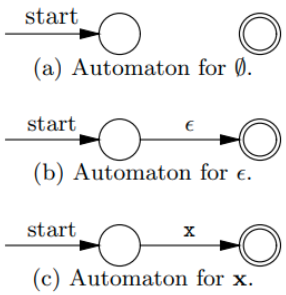

FIGURE 1 – Automate de base

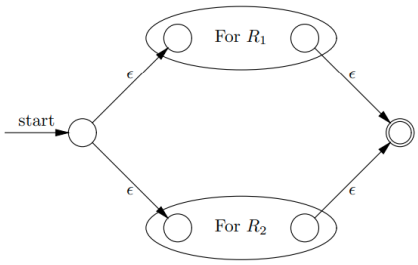

FIGURE 2 – Connection d'alternance

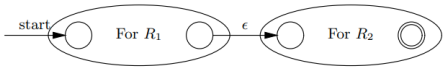

FIGURE 3 – Connection de concatenation

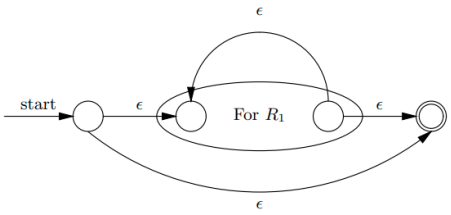

FIGURE 4 – Connection de closure

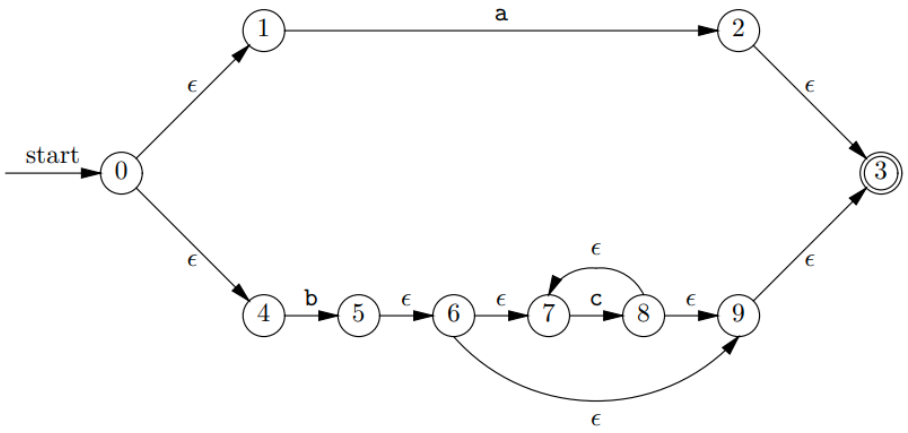

FIGURE 5 – Une exemple de NFA

3 Analyse de complexité

3.1 Cas général

Dans le cas général, DFA permet de trouver tous les lignes qui contient le mot satisfaisant à l'expression régulière saisie. Supposons que la longueur de chaine de caractères est n, pour vérifier si elle contient le mot suffisant, on a besoin d'appeler n fois $DFA.search(str.substr(i, n))$, i est l'indice du commencement, $i \in [0, n]$, dans le pire cas. Donc la complexité est $\frac{(1+n)*n}{2} = \mathcal{O}(n^2)$.

3.2 Cas particulière

Le cas particulier est qui l'expression régulière saisie est seulement une chaine de caractère. Dans ce cas-là, on n'a plus besoin de construire DFA, la solution optimale est l'algorithme KMP. Cela permet de résoudre le problème en $\mathcal{O}(n)$.

4 Partie de test

Rédacteur : mehdi. l'explication de résultat de l'exécution, tu peux mettre une capture d'écran dans la section Annexe, et utiliser ref pour redirect à la position des images, comme ce que j'ai fait au dessus.

5 Test de performance

Rédacteur : mehdi. tu compares le temps d'exécution des cas différents.

6 conclusion

L'utilisation de DFA peut garantir une complexité de base $O(n^2)$ pour vérifier si une chaine de caractère qui contient le mot satisfaisant à l'expression régulière saisie. L'algorithme KMP peut optimiser certain cas particulier.