Rapport Projet offline

Ivan Delgado
Elisabeth Abbas Zadeh

14 octobre 2019

Première partie

Introduction

Ce document est le Rapport du projet *offline* dans le cadre de l'UE DARR.

Le but de ce projet est de réaliser la partie *offline* d'un moteur de recherche ; à savoir la recherche d'un motif sous le format RegEx dans un texte issu du Projet Gutemberg.

Pour cela nous allons faire une copie de la commande *egrep* en nous restreignant pour les éléments de RegEx aux parenthèses, à l'alternative, à la concaténation, à l'opération étoile, au point et aux lettres ASCII.

Deuxième partie

Mode d'emploi

Le code du projet a été réalisé en Java, il est fourni avec un fichier *build.xml* pour *apache ant* qui permet les commandes suivantes :

- *ant compile*
  pour compiler le projet
- *ant run -Darg0="<motif a rechercher>" -Darg1="<fichier texte>*
  pour lancer la recherche du motif sur un texte
- *ant clean*
  pour nettoyer le répertoire du projet

Troisième partie

Stratégies implémentées

Dans un premier temps nous considérons que le texte est une matrice, c'est un tableau de lignes qui contient chacune un tableau de caractères. Pour dé-

terminer la position d'un match, il suffit donc d'un numéro de ligne et d'une position dans la ligne, cette information est dans la classe *TextPosition.java*.

1 Recherche par Automates

Le traitement d'un regex passe par différentes opérations :

1. Dans un premier temps, nous allons parser le regex pour en construire un arbre de syntaxe. Cette opération est en $O(n)$, où n est le nombre de caractères du regex.
2. A partir de l'arbre, construire un automate fini non-déterministe avec *epsilon*-transitions selon la méthode Aho-Ullman [1].
3. Déterminiser l'automate en enlevant les *epsilon* transitions avec la méthode des sous-ensembles. La complexité est en $O(n^2)$, avec n le nombre d'états de l'automate.
4. Minimiser l'automate en construisant un automate équivalent avec un nombre minimum d' états avec l'algorithme de Hopcroft. La cmplexité est en $O(|A|.n \log n)$, avec $|A|$ la taille de l'alphabet et n le nombre d'états.
5. Enfin l'automate est utilisé pour tester si une sous-chaîne d'une ligne du fichier textuel donné initialement est reconnaissable par cet automate. Des parallelStream sont utilisés sont chaque ligne du fichier dans un souci d'optimisation, ainsi la complexité est inférieure à $O(|L|.|C|.n)$, où $|L|$ est le nombre de lignes, $|C|$ le nombre de caractères par ligne et n le nombre d'états de l'automate.

La structure de données représentant un automate est semblable à celle de la liste d'adjacence d'un graphe, il y a une ligne par état, chacune contenant un tableau de 256 entiers, un par caractère ASCII, la valeur de la case est l'état résultant de la transition par ce caractère, la valeur est *null* s'il n'y a pas de transition.

On ajoute à cela trois tableaux, un pour les états de départ, un pour les états finaux et un pour les *epsilon* transitions.

Le code de la structure de données, ainsi que les algorithmes de détermination et de minimisations se trouvent dans le fichier Automate.java.

Implémentation

public static ArrayList<TextPosition> getOccurencesOnText(ArrayList<String> text, String regEx)

Cette fonction permet d'obtenir une liste de couple de positions (ligne, colonne). En entrée, elle prend une liste de lignes du texte et une regex.

L'automate déterministe minimal associé à la regex est calculé.

Ensuite, on fait un stream pour appliquer à chaque ligne getOccurencesOnLine pour récupérer la liste des colonnes où se trouve le mot.

1. http ://infolab.stanford.edu/~ullman/focs/ch10.pdf

public static boolean isWord(Automate a, String substring)

A partir de l'état initial, pour chaque lettre de la sous-chaîne, on cherche à trouver un état final dans l'automate. Dans ce cas on renvoie vrai. S'il n'y pas de chemin possible dans l'automate ou si on a atteint la fin de la sous-chaine, le mot n'a pas été trouvé. Elle est utilisée dans getOccurencesOnLine.

public Automate minimize(String regEx)

Il s'agit de l'algorihtme de Hopcroft.

La partition est composée de deux ensembles au départ : les états terminaux et les autres. W est un ensemble qui contient les couples des lettres et du plus petits des 2 sous-ensembles.

Tant que W n'est pas vide, on prend un couple dans W, on teste si son ensemble coupe un ensemble de la partition. Si oui, on met à jour la partition en séparant cet ensemble en deux et on met à jour W.

On recrée enfin un automate en créant un état par classe d'équivalence.

public Automate determinize()

L'état de départ est formé des états de départ et des états pour lesquels il existe un chemin depuis un état de départ composé uniquement d'epsilon-transitions. Ensuite pour chacun de ses états, on crée un ensemble pour chaque transition (qui est l'ensemble de l'état résultant de cette transition et des états qui possèdent un chemin composé d'epsilon-transition depuis cet état-là). Si l'ensemble de ces états n'existe pas déjà, on en fait un nouvel état.

2 Algorithme de Knuth-Morris-Prat

Si le RegEx recherché n'est composé que de concaténations, il est alors superflu d'en construire un automate, une représentation en chaîne de caractères suffit.

Nous pouvons alors utiliser l'Algorithme de Knuth-Morris-Pratt (KMP) qui est un algorithme de recherche de sous-chaînes, permettant de trouver les occurrences d'une chaîne P dans un texte S avec une complexité linéaire $O(|P|+|S|)$. KMP se base sur un prétraitement de P qui donne un tableau d'indices appelé retenue qui donne la position de la chaîne où reprendre les comparaisons en cas de non-concordance, permettant de ne pas avoir à ré-examiner les répétitions de caractères et de préfixes et limitant ainsi les comparaison nécessaires.

KMP est un algorithme simple ne nécessitant pas de structure de données particulière, son code se trouve dans le fichier *KMP.java*.

Implémentation

public static ArrayList<TextPosition> kmp (ArrayList<String> text, char[] factor)

On souhaite avoir toutes les positions dans le texte des début du mot à chercher. Pour cela, on parcourt le texte.

Si une lettre correspond à la lettre courante du mot, on vérifie si le mot n'est pas terminé. Si c'est le cas, on ajoute à l'ensemble des solutions la position de départ du mot dans le texte, puis on récupère la retenue à cet indice et on met à jour l'indice.

Si la lettre ne correspond pas à la lettre courante du mot, on récupère la retenue et on met à jour l'indice.

On renvoie l'ensemble des positions trouvées.

public static int[] retenue(char[] factor)

Cette fonction crée un tableau de retenues pour un facteur.

Pour chaque lettre du facteur, si la lettre correspond à la première lettre du facteur, on met à jour le retenue et on comparera par la suite la deuxième lettre des facteurs.

Sinon, si on ne comparait pas la première lettre du facteur, on reprend à la retenue, alors que si c'était la première lettre, on met à jour la retenue courante.

3 Index et Radix Tree

Un texte peut être décomposé en une liste de mots (une suite de caractères uniquement alphabétiques), avec *awk* nous pouvons facilement créer une liste des mots uniques d'un texte dans un fichier *.txt* (situé dans le répertoire *index).* Puis, en utilisant l'algorithme KMP sur chacun de ces mots, nous pouvons trouver la position de l'ensemble des occurrences, qui, triées par ordre décroissant d'occurrences, seront mises dans un fichier *.index; cette suite d'opérations est coûteuse mais elle n'est à faire qu'une fois par fichiers.*

Ensuite avec cet index, nous pouvons en construire un *radix tree,* ce qui permet de dire rapidement si le mot recherché est dans le texte.

L'arbre contient, pour chaque nœud, une liste de positions lorsque le nœud correspond à un mot de l'index et une liste des fils pour la lettre suivante correspondante. S'il n'y a plus de lettre, les positions retournées seront celle du noeud et de ses fils récursivement.

Implémentation

public static String createIndex(String path, ArrayList<String> text)

Cette méthode fait appel au *script.bash*, composé de awk qui va trouver les mots uniques, enlever les mots de 2 caractères ou moins et les trier par nombre d'occurrences croissant et les écrire dans le fichier *.txt*.

Puis va lancer KMP sur chacun de ces mots afin de trouver les positions de ses occurrences, va les mettre dans un fichier *.index* et retourne le path de ce fichier.

public ArrayList<TextPosition> getPositions(String valeur)

Parcours le *radix tree* lettre par lettre et a la dernière lettre renvoie la valeur dans le noeud courrant si il y en a, et celle contenues dans ses fils.

L'indexation se fait dans *Indexing.java* et le *radix tree* est défini dans *IndexTree.java*.

La stratégie que nous allons appliquer est la suivante :
Si le motif est un mot, nous allons le chercher avec la méthode de l'index, s'il n'y a pas de résultat ou si le motif n'a que des concaténations, on utilise KMP.
Si c'est un regex, nous utilisons la méthode de l'automate.

Quatrième partie
Test et Analyse

A History of Babylon (124 304 mots) :

- Recherche de « Sargon » :
  temps par egrep 15 ms
  temps par index :
  - construction de l'index : 37 s
  - chargement de l'arbre 300 ms
  - recherche dans l'arbre : <1µs
- Recherche de « S(a|g|r)*on » :
  temps par egrep : 18 ms
  temps par l'automate : 161 ms
- Recherche de « argon »
  temps egrep : 23 ms
  temps KMP : 40 ms

Dracula (164 424 mots) :

- Recherche de « Dracula » :
  temps par egrep 21 ms
  temps par index :
  - construction de l'index : 44 s
  - chargement de l'arbre 500 ms

    — recherche dans l'arbre : <1µs
— Recherche de « D(r|a|c)*ula » :
  temps par egrep : 10 ms
  temps par l'automate : 125 ms
— Recherche de « racula »
  temps egrep : 25 ms
  temps KMP : 50 ms

Don Quixote (430 267 mots) :

— Recherche de « Quixote » :
  temps par egrep 10 ms
  temps par index :
  — construction de l'index : 223 s
  — chargement de l'arbre 650 ms
  — recherche dans l'arbre : <1µs
— Recherche de « Q(u|i|x)*ote » :
  temps par egrep : 10 ms
  temps par l'automate : 600 ms
— Recherche de « uixote »
  temps egrep : 10 ms
  temps KMP : 150 ms

Dans un premier temps, on peut constater la constance et l'optimisation de egrep, fruit d'années de maturation.
On observe que KMP est bien plus rapide que l'automate, tient mieux le passage a l'échelle mais est beaucoup moins versatile.
L'index, quant à lui, est extrêmement coûteux dans le calcul de l'index et dans la construction en mémoire du *radix tree*, mais une fois en place, les recherches dans l'arbre sont proches de l'instantanée et de bien loin plus rapides que les autres méthodes.

Cinquième partie
Conclusion

Dans cet exercice d'implentation d'un clone de la commande egrep, qui est rarement utilisée plusieurs fois sur le même fichier, l'automate et KMP suffisent. L'automate permet de prendre en charge l'ensemble des motifs possibles et KMP vient accélérer la recherche sur les mots naturels, qui sont de loin les plus fréquemment recherchés.

Néanmoins, si on regarde la situation d'un point de vue plus global, celui d'un moteur de recherche sur le projet Guntemberg, on sera plus enclin à lancer plusieurs fois des recherches dans un même document et l'investissement payé à construire et à charger des index pour ces documents se rentabilise rapidement.

Plus les recherches sur un même document sont fréquentes, plus la méthode de l'index est payante.