DAAR Projet 1 : Clone de *egrep* en Java

Emilie Siau 3700323
Hugo Guerrier 3970839

7 Octobre 2021

Abstract

Ce document résume les méthodes d'implémentation et détaille les algorithmes utilisés durant la réalisation d'un clone partiel de *egrep* en Java. Nous y détaillons le travail qui a été réalisé, les tests unitaires fournis avec le programme ainsi que les performances des différentes solutions imaginées.

1 Définition du problème

Il nous semble aujourd'hui normal d'utiliser des moteurs de recherche avancés et efficaces tels que *Google* ou *Bing* mais derrière ces derniers se cachent des technologies complexes et efficaces permettant de trier, d'indexer et de rechercher dans les volumes de données disponibles sur l'internet mondial. Parmi ces algorithmes, celui de l'expression régulière a une place toute particulière, et c'est à lui que nous allons nous intéresser dans ce document.

Nous définissons comme notre but dans ce projet : Dans un document texte "pur", organisé en lignes, avec potentiellement des dizaines de millier de lignes, de trouver toutes les lignes correspondantes à une expression régulière donnée par l'utilisateur, dans le temps le plus court possible. Nous nous sommes donc basé sur les étapes données dans le livre d'Aho-Ullman (plus précisément au chapitre 10) qui sont les suivante :

1. Création d'un arbre de syntaxe pour l'expression régulière (étape du parsing)
2. Création d'un automate fini non déterministe (NDFA) à partir de l'arbre de syntaxe
3. Détermination de l'automate à l'aide de la méthode des sous-ensemble pour obtenir un automate déterministe fini (DFA)
4. Simplification de l'automate
5. Recherche dans un texte donné

Etant donné que nous avions déjà le code permettant d'effectuer l'étape 1 et que l'étape 4 est optionnelle, nous nous sommes chargé uniquement des étapes 2, 3 et 5. Nous allons donc voir dans la prochaine partie quelle a été notre approche dans la réalisation de toutes ces étapes.

2 Implémentation

Dans cette partie nous détaillons tous les choix que nous avons fait dans l'élaboration de notre solution, que ce soit au niveau des algorithmes ou de la représentation des données. Il est bon de rappeler que tout sera écrit en pseudo code, mais que le projet qui accompagne ce rapport, lui, est rédigé en Java.

2.1 La représentation des données

2.1.1 L'arbre de syntaxe de l'expression régulière

La représentation de l'arbre de syntaxe est somme toute assez classique, à savoir une définition récursive effectuée comme suit.

```
type Tree {
    char root
    sub Tree[]
}
```

2.1.2 L'automate

La représentation de l'automate que nous avons choisie se rapproche de celle proposé par *Aho-Ullman* dans leur ouvrage, à savoir un tableau de tableau de transition. Cependant nous avons défini le type *NodeId* représentant l'identifiant d'un état (ou noeud), permettant de construire une *map* avec en clé un *NodeId* et en valeur un *NodeId[][]*.

```
type NodeId {
    id = int[]
}

automaton = Map<NodeId, NodeId[][]>
```

2.2 La création du NDFA

Le but de cette étape est de transformer un arbre de syntaxe d'expression régulière en automate fini non-déterministe, comme dans l'exemple qui suit issu du livre d'Aho-Ullman :

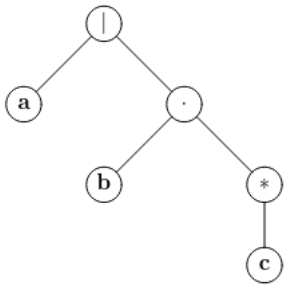

Figure 1: Arbre de l'expression régulière "a—bc*"

A partir de cet arbre de syntaxe d'expression régulière, on doit obtenir l'automate suivant :

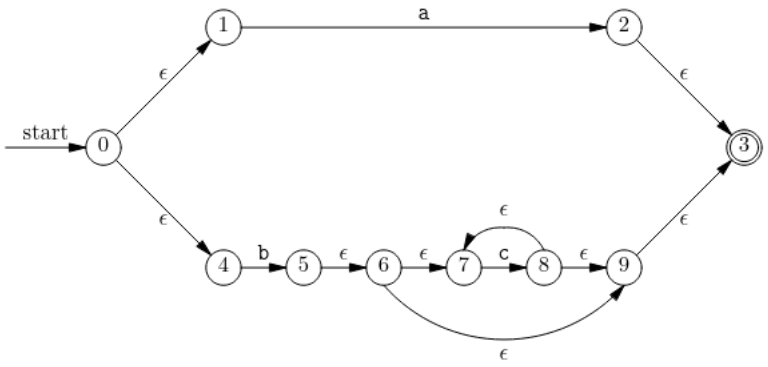

Figure 2: NDFA de l'expression régulière "a—bc*"

Pour réaliser ceci, nous avons utiliser une fonction récursive parcourant l'arbre de syntaxe de la racine jusqu'aux feuilles et construisant l'automate en le parcourant. Voici donc la fonction en question donnée ici en pseudo-code.

```
function get_ndfa(tree, final_node) : NodeId {
    init_node = new NodeId()

    switch(tree.root) {
        case ALTERN:
            left_node = get_ndfa(tree.sub[0], final_node)
            right_node = get_ndfa(tree.sub[1], final_node)
            transition = NodeId[][]
            transition[EPSILON_POS] = [left_node, right_node]
            automaton[init_node] = transition

        case CONCAT:
            right_node = get_ndfa(tree.sub[1], final_node)
            init_node = get_ndfa(tree.sub[0], right_node)

        case STAR:
            loop_exit = new NodeId()
            exit = new NodeId()

            loop_init = get_ndfa(tree.sub[0], loop_exit)
            transition = NodeId[][]
            transition[EPSILON_POS] = [loop_init, exit]
            automaton[init_node] = transition
            automaton[loop_exit] = transition

            transition = NodeId[][]
            transition[ESPILON_POS] = [final_node]
            automaton[exit] = transition

        case DOT:
            next = new NodeId()
            transition = NodeId[][]
            transition[EPSILON_POS] = [final_node]
            automaton[next] = transition
```

```
            transition = NodeId[][]
            transition[DOT_POS] = [next_node]
            automaton[init_node] = transition

        default:
            next = new NodeId()
            transition = NodeId[][]
            transition[EPSILON_POS] = [final_node]
            automaton[next] = transition

            transition = NodeId[][]
            transition[tree.root] = [next_node]
            automaton[init_node] = transition
    }

    return init_node
}
```

Comme vous pouvez le constater, cette fonction ne prend pas en charge tous les opérateurs définis dans la norme *Exetended RegEx*, en effet, dans le cadre de ce projet, nous avons uniquement implémenté certains opérateurs en laissant la possibilité d'enrichir cette liste dans le futur.

2.3 La création du DFA

Une fois que nous avons créer le NDFA, pour évaluer une entrée, il faut avoir un automate déterministe, c'est pourquoi on doit effectuer un traitement pour déterminer celui qu'on obtient après l'étape vue précédemment. Voici l'exemple de l'automate déterministe correspondant à celui de la Figure 2.

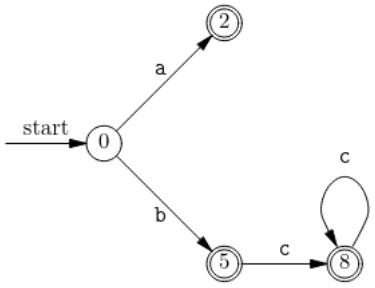

Figure 3: DFA de l'expression régulière "a—bc*"

Pour obtenir cette détermination, nous avons donc utilisé la méthodes des sous-ensembles, consistant à considérer tous les états reliés par une transition ε comme un seul et même état. Voici le pseudo code de cet algorithme.

```
function epsilon_closure(node_list) : NodeId[] {
    res = []
    for(node : node_list) {
        if(!(node in res)) {
            node_list.add_all(automaton[node][EPSILON_POS])
```

```
        }
    }
    return res
}

function get_dfa() : void {
    dfa = Map<NodeId, NodeId[][]>
    working_list = [epsilon_closure([ndfa_init_state])]

    while(!working_list.empty()) {
        current_list = working_list.remove(0)
        current_node = new NodeId(current_list)

        if(dfa[current_node] == null) {
            new_transition = NodeId[][]

            for(node : current_list) {
                for(i = 0 ; i < CHAR_NUMBER ; i++) {
                    t = automaton[node][i]
                    new_transition[i].add_all(t)
                }
            }

            for(node_list : new_transition) {
                if(!node_list.empty()){
                    working_list.add(node_list)
                    epsilon = epsilon_closure(node_list)
                    node_list.clear()
                    node_list.add(new NodeId(epsilon))
                }
            }

            if(current_list contains ndfa_init_state) {
                new_transition[INIT_POS] = new NodeId[]
            }

            if(current_list contains ndfa_final_state) {
                new_transition[ACCEPT_POS] = new NodeId[]
            }

            dfa[current_node, new_transition]
        }
    }

    automaton = dfa
}
```

Avec cet algorithme, l'automate est maintenant déterministe et fini, ce qui nous permet de passer à la dernière étape, la recherche dans un texte.

2.4 La recherche

Il existe plusieurs méthodes de recherche dans un texte à l'aide d'un automate, nous allons détaillé ici deux d'entre-elles, la première, dite naïve et la seconde, suivant l'algorithme de Knuth-Morris-Pratt.

2.4.1 Méthode Naïve

Cette méthode consiste en un parcours naïf de la chaîne de caractère en entrée, essayant pour chaque caractère de vérifier l'automate de l'expression régulière. Voici une implémentation de cette méthode en pseudo-code :

```
function naive_search(input) : boolean {
    for(cursor = 0 ; cursor < input.size() ; cursor++) {
        current_node = dfa_init_node
        for(next_in = cursor ; next_in < input.size() ; cursor++) {
            next_char = input[next_in]
            if(automaton[current_node][next_char] == null) {
                break;
            } else {
                current_node = automaton[current_node][next_char][0]
                if(automaton[current_node][ACCEPT_POS] != null) {
                    return true
                }
            }
        }
    }
    return false
}
```

On constate sans soucis que cet algorithme s'exécute en $O(n^2)$ (les deux *for* imbriqués), ce qui est relativement médiocre au niveau des performances. Il serait judicieux de rechercher et d'implémenter une méthode plus efficace utilisant notre automate.

2.4.2 Algorithme de Knuth-Morris-Pratt

Une autre manière d'opérer la recherche d'une chaîne de caractères C dans un texte T est l'utilisation de l'algorithme de Knuth-Morris-Pratt (*KMP*), publié en 1977 par Knuth, Morris et Pratt, dont l'avantage est d'avoir une complexité linéaire $O(|C| + |T|)$ dans le pire cas. Celui-ci présente cependant une contrainte supplémentaire : la RegEx donnée doit être composée de concaténations simples, c'est à dire sans les opérateurs '*', ' | ' et '+'. Ainsi, pour la recherche de RegEx plus complexes, il doit être complété d'un autre algorithme de recherche capable de les traiter.

L'algorithme KMP fonctionne de la manière suivante : il n'utilise pas de DFA mais il crée un tableau d'entiers *Carry Over* (CO) de la taille de la RegEx (R) en amont de la recherche, qui permettra d'avancer dans le texte T plus efficacement. La CO contient des valeurs qui permettent de revenir en arrière ou de sauter plus loin dans T en fonction de la structure de C lorsque un caractère n'est pas matché avec la RegEx.

Le pseudo-code de l'algorithme se trouve ci dessous :

```
function kmp_searching(text) : boolean {
```

```
    textPos = 0
    regexPos = 0

    while(textPos < T.size()) {
        if (R[regexPos]== '.' || T[textPos] == R[regexPos]) {
            textPos++
            regexPos++
        } else {
            textPos = textPos - CO[regexPos]
            regexPos = 0
        }

        if (regexPos == R.length()) return true
    }
    return false
}
```

CO[i] = taille du plus grand suffixe de F[1:i[qui est également un préfixe de F

En effet, par exemple, lorsque l'on cherche la RegEx "mami", on peut remarquer que si l'on essaie de la faire correspondre avec le texte "maman papa", on avance naïvement dans le texte ainsi : $R[0] = T[0], R[1] = T[1], R[2] = T[2] mais R[3] \neq T[3]$.

Et alors, plutôt que de réessayer la correspondance avec T[1:], on observe : CO[3] = 1. Donc nous allons reculer dans T d'1 caractère seulement, plutôt que de deux. La CO nous permet donc d'éviter des tests inutiles en reculant le moins possible ou en avançant dans T.

Il faut donc construire la CO correspondant à la RegEx donnée. Nous avons implémenté sa construction ainsi qu'une optimisation de celle-ci, dont le pseudo-code se trouve ci dessous :

```
function build_carry_over(R) : void {
    CO = new int[R.length()]
    CO[0] = -1

    for (int i = 1; i < R.length() ; i++) {
        // CO[i] = size of the greatest suffix of R[1:i[,
        // that is also a prefix of R

        suffixes = {}
        for (int j = 1 ; j < i ; j++) {
            suffixes.add(R[j:i[)
        }

        int maxSize = 0;
        foreach (String suffix : suffixes) {
            if (R.startsWith(suffix)) {
                if (suffix.length() > maxSize)
                        maxSize = suffix.length()
            }
        }
        CO[i] = maxSize;
```

```
        }

        // Optimizing
        for (int i = 0; i < R.length() ; i++) {
            if (CO[i] >= 0 && R[i] == R[CO[i]])) {
                CO[i] = CO[CO[i]]
            }
        }
}
```

2.4.3 Méthode Native de Java

Nous avons également implémenté une méthode de recherche faisant appel à l'API native de Java *Pattern* dans un but de comparaison avec les implémentations que nous avons réalisées. En effet, comparer nos algorithmes avec l'implémentation egrep présente dans Linux n'a pas beaucoup de sens, le langage ayant un impact très fort dans les performances des programmes, il est bien plus pertinent de faire cette comparaison au sein de Java. Nous allons donc voir maintenant les tests de performances réalisés sur nos algorithmes.

3 Tests et performances

Etant donnée la taille du projet et sa complexité algorithmique, il était important pour nous de nous assurer du bon fonctionnement de nos implémentations. Il est également crucial, dans un but expérimental, de pouvoir mesurer et comparer l'efficacité des différentes solutions que nous avons proposées, entre elles, mais aussi avec les outils "officiels" (comme la commande *egrep* ou la classe *Pattern* de Java).

Nous avons obtenu nos testbeds sur la base de Gutenberg disponible à cette adresse : `http://www.gutenberg.org/`, et nous avons utilisé des fichiers de tailles différentes pour plus de variété.

3.1 Les tests unitaires

Nombre de tests unitaires ont été réalisés pour assurer le bon fonctionnement du projet. Nous avons pour cela utilisé le framework *JUnit 5*, permettant d'écrire et de lancer des tests unitaires de façon simple et efficace. Il sont tous présent dans le package *egrep.test*.

Nous avons donc prévu des tests pour :

- Le parser
- La création du NDFA
- La création du DFA
- La recherche par méthode naïve
- La recherche à l'aide de l'algorithme KMP

Pour chacun de ces tests, nous avons pris soin de couvrir toutes les fonctionnalités de notre application et de tester les plus d'aspects possible. Malgré tout cela, il reste impossible avec des tests unitaires de prouver une absence de bug, pour cela il faudrait utiliser des outils de preuve plus formels (Coq par exemple).

3.2 Les performances de nos solutions

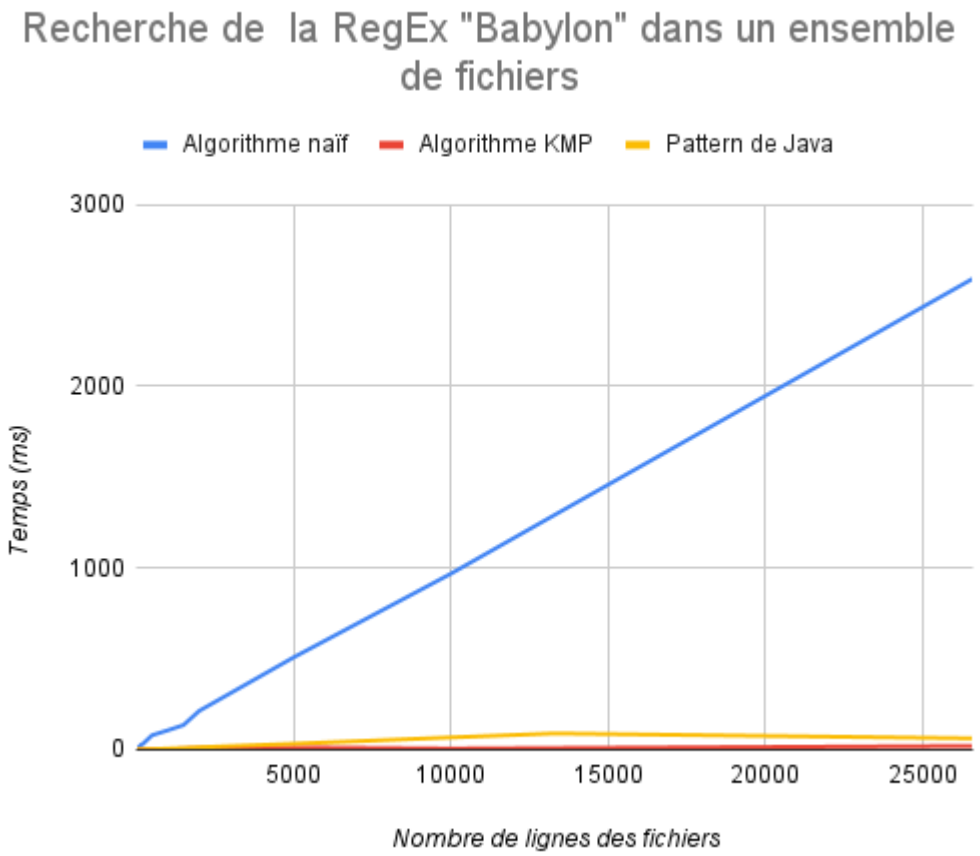

Figure 4: Test de performance des méthodes de recherche

La figure 4 nous permet d'observer la linéarité de l'algorithme naïf mais également sa grande infériorité aux deux autres algorithmes utilisés sur une RegEx simple.

Afin de voir plus en détail les différences entre l'algorithme KMP et le Pattern de Java, la figure 5 permet un zoom sur ces deux méthodes et l'on peut constater la grande efficacité de l'algorithme KMP. Seulement, il est important de se rappeler de la restriction importante imposée par KMP : on ne peut que chercher des RegEx simples. Ainsi, l'implémentation de Pattern de Java est très efficace, car elle permet également une recherche sur des RegEx complexes, en étant largement plus efficace que la méthode naïve.

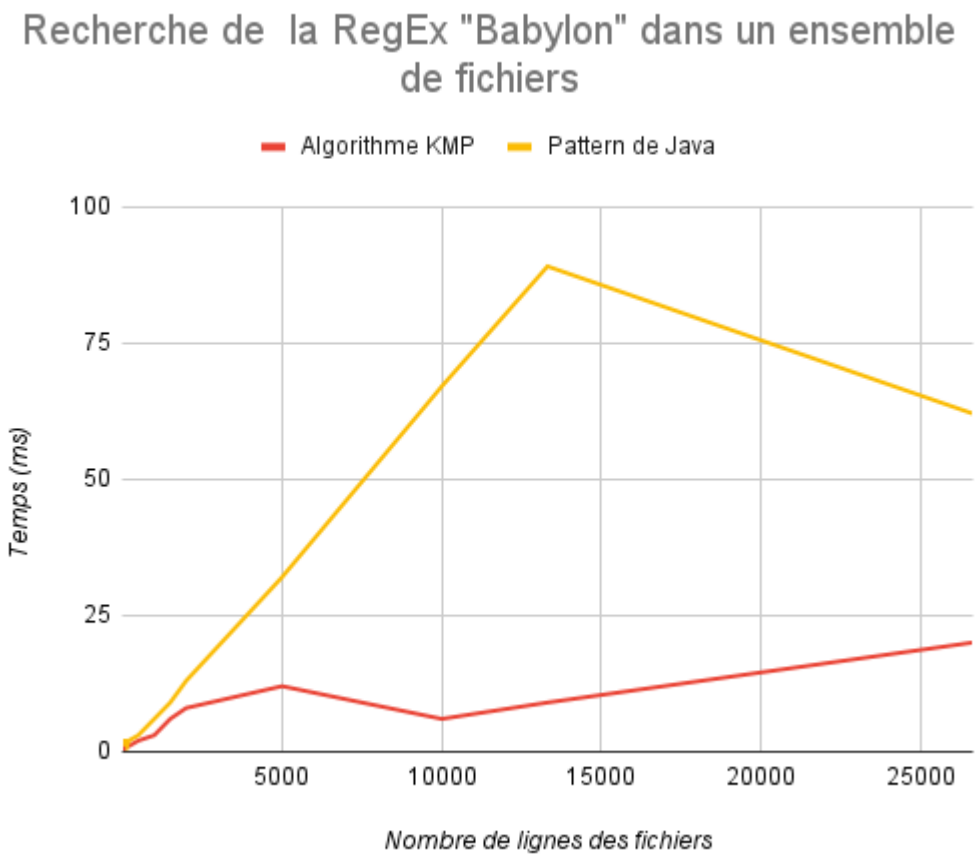

Figure 5: Test de performance des méthodes de recherche (sans la méthode naïve)

4 Conclusion

4.1 Conclusion sur le travail

Ce projet a été pour nous une réussite, nous avons réussi à implémenter tout ce que nous voulions et ce de manière efficace et propre. L'apprentissage sur les automates de recherche est un grand plus pour nous et nous servira sûrement dans la suite de nos travaux, ce concept étant lié à beaucoup d'autres domaines comme le parsing de langage.

Nous aurions aimé pousser plus loin la recherche et améliorer l'efficacité de notre implémentation avec d'autres algorithmes ou d'autre structures de données ainsi que des tests plus poussés sur de nombreuses RegEx différentes, mais le temps nous est limité.

4.2 Idées d'améliorations et ouverture

Voici quelques idées d'améliorations qu'il serait possible d'effectuer :

- Faire hériter l'automate de la classe *Map* pour éviter les indirections
- Utiliser un langage plus performant pour l'implémentation
- Ajouter toutes les opérations définies dans le standard *Extended Regular Expression*, notamment l'échappement avec le *backslash*
- Utiliser une plage de caractères plus importante (utf-8)

- Augmenter la quantité de tests pour une meilleure appréciation de l'efficacité des algorithmes sur des RegEx variées

Bien entendu il faudrait beaucoup plus de temps et de travail pour pouvoir mettre en place ces améliorations, mais cela pourrait-être très instructif et même ouvrir sur d'autres domaines.