RapportDaarProjet1

Keita Lamine
Memmi Sacha

October 2020

Part I
Algorithmes

1 Introduction

Dans la suite de ce rapport, nous pourrons discuter de l'implémentation de différents algorithmes visant à effectuer une recherche d'expression régulière au sein d'un texte.

Pour effectuer cette recherche nous avons opté pour un mélange de la première stratégie, visant à représenter les structures reg ex en tant qu'automate, et de la seconde, de l'algorithme de recherche KMP (Knuth-Morris-Pratt).

Nous considererons dans nos recherches des automates déterministes ayant un nombre minimal d'états.

Nous effectuerons par ailleurs, au cours de la partie expérimentale des comparaisons de temps d'exécution entre la commande bash grep et notre recherche, ainsi qu'une comparaison entre notre méthode et les méthodes 1 et 2.

2 Construction de l'automate

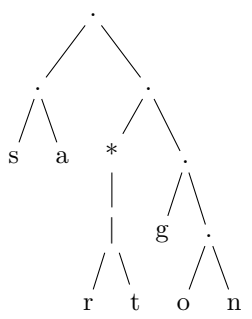
La conversion d'une expression régulière en automate optimisée et déterministe (selon l'algorithme Aho-Ullman [**aho**]) comporte 4 étapes principales.

Dans cette section nous nous intéresserons aux deux premières étapes qui aboutiront à la création d'un automate non déterministe.

2.1 Arbre regex

Une regex peut, récursivement, être défini comme un arbre. Cet arbre sera constitué de trois types de noeuds internes (les symboles "*", "." et "|") et des feuilles représentant des caractères quelconques ou le caractère universel ("." à ne pas confondre avec le noeud "." qui représente la concaténation).

Ainsi l'expression regex $sa(r|t)*gon$ peut être redéfini en l'arbre regex (ou regex tree) suivant:



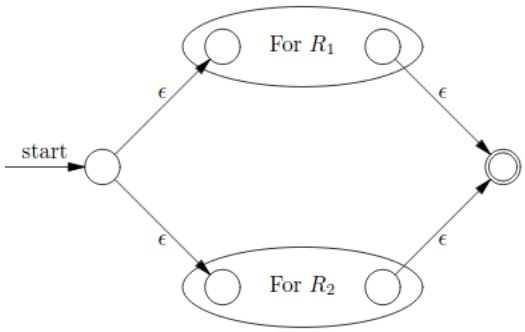
Les structures d'arbres sont des éléments fondamentaux de la programmation car ils ont l'avantage d'être facilement manipulables par des programmes informatiques.

2.2 Automate non déterministe

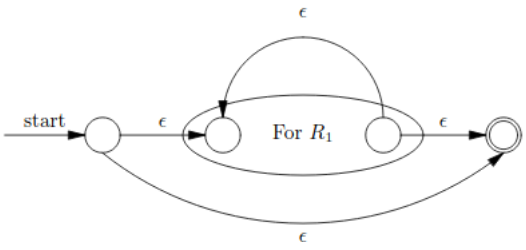
Une fois l'arbre du regex défini, nous pourrons construire un automate non déterministe en parcourant l'arbre précédemment créé.

Comme rappelé plus haut, les feuilles de l'arbre représentent des caractères alors que les noeuds internes représentent des symboles d'opérations (unaires et binaires).

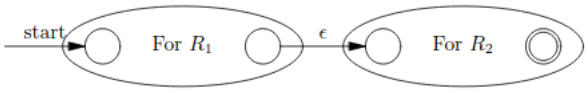
Nous profitons de cette propriété des arbres pour ajouter à l'automate des structures lors du parcours (dans l'ordre préfixe gauche) des éléments de l'arbre RegEx. La traduction de l'arbre RegEx en structure de l'automate se fera des manières suivantes: Le symbole |, qui représente l'union, sera défini par une structure de la forme:



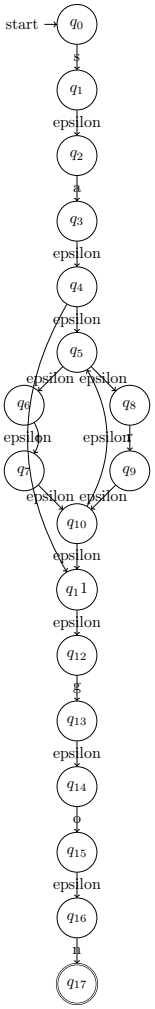
Le symbole *, qui représente la closure, sera défini par une structure de la forme:



Le symbole ., qui représente la concaténation sera défini par une structure de la forme:



Les noeuds start et puits ne seront pas construits lors de l'assemblage des structures.
Ces deux noeuds sont présents dès l'initialisation de la structure. La construction de l'automate de la RegEx $sa(r|t) * gon$ aura pour automate:


start
q_0
s
q_1
epsilon
q_2
a
q_3
epsilon
q_4
epsilon
q_5
epsilon epsilon
q_6
q_8
epsilon
epsilon
q_7
q_9
epsilon epsilon
q_{10}
epsilon
q_{11}
epsilon
q_{12}
s
q_{13}
epsilon
q_{14}
o
q_{15}
epsilon
q_{16}
n
q_{17}

Dans notre automate, si nous avons une chaîne de caractères **sa**, nous nous trouverons dans le noeud non final q3, q4, q5, q6, ou q8.
Or aucun de ces noeuds n'est l'état acceptant q17, cette chaîne de caractères n'est donc pas acceptée par l'automate ($sa \neq (sa(r|t) * gon)$).

3 Déterminisation

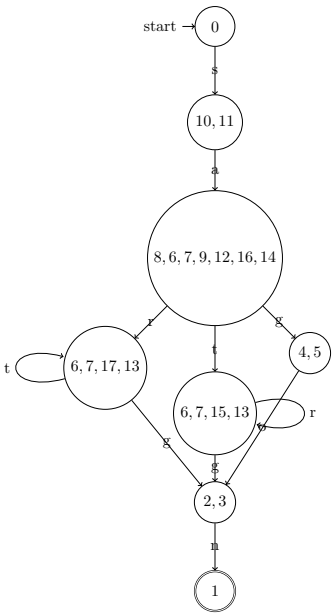
Un automate non déterministe est un automate qui n'a pas nécessairement à chaque étape une sortie constante.
C'est à dire que l'on peut se retrouver dans un noeud et ne pas savoir quelle transition sera utilisée pour passer à un autre état. Les transitions epsilon en sont un bon exemple car elles peuvent ou non être traversées. Le but de cette partie est de construire un automate déterministe à partir d'un automate non déterministe.
Il est important de souligner que les automates déterministes et non déterministes doivent être équivalents (ils doivent accepter et refuser les mêmes entrées).

3.1 Description de l'algorithme

1. Nous partons d'un noeud ainsi que d'un ensemble de départ
2. Nous ajoutons à cet ensemble de départ les noeuds liés au noeud de départ par des liens epsilon
3. Nous continuons d'ajouter tous les noeuds liés par des liens epsilon avec un noeud de l'ensemble de départ
4. Récursivement, nous allons:
    - identifier une transition possible depuis un noeud (ou plusieurs) depuis un ensemble
    - Créer un nouvel ensemble lié à l'ensemble duquel on a tiré le noeud (ou les noeuds) dont on tire la transition par cette même transition
    - Ajouter à l'ensemble nouvellement crée tous les noeuds connectés aux noeuds de l'ensemble par des transition epsilon
    - Continuer à ajouter des noeuds liés par des transitions epsilon tant que possible
    - Si un noeud contenu dans un des ensembles est le noeud final on considère que cette ensemble est dit un ensemble d'acceptation

On considère des ensembles comme un état d'un automate déterministe, les ensembles d'acceptation deviennent des noeuds d'acceptation.

La déterminisation de l'automate précédent donnera:

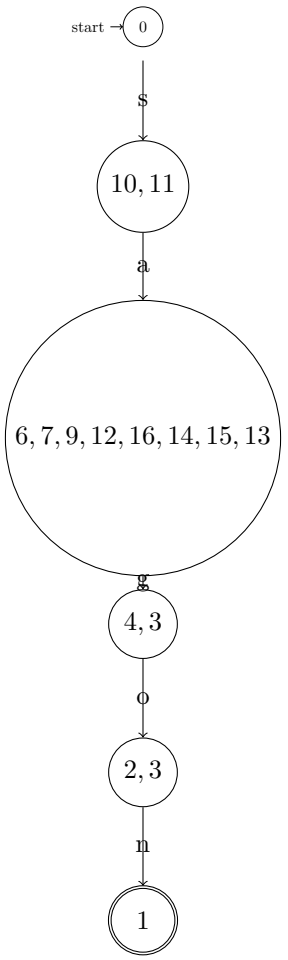


3.2 Optimisation

Les automates obtenus par la méthode des tableaux décrit précédemment ne sont pas nécessairement optimisés. C'est pour cette raison que la dernière étape à savoir l'optimisation nous sera utile. L'algorithme d'optimisation de l'automate peut être défini comme suit:

1. On considère les états de l'automate déterministe deux a deux
2. On détermine si les deux états sont équivalents dans le sens où:
    - Les liens sortants mènent aux mêmes états, en considérant les cas spéciaux tel qu'un même lien sortant pour les deux états mène à l'autre.
    - Éviter tout conflit tel que l'un des deux est un puits et pas l'autre
3. Si deux états sont équivalents on les fusionne en faisant attention à modifier les transitions des autres noeuds.

On obtient ainsi un automate optimisé dont le nombre d'état est minimal. L'optimisation de l'automate précèdent produira la structure suivante:



4 KMP

Dans une comparaison texte, nous faisons des comparaisons inutiles au sein du mot.

Par exemple dans le texte:
Ma mère est morte, ma grand-mère vient de m'en informer, elle est morte après avoir été mordue par notre chien.
Si par exemple nous recherchons le mot "mordue" on aurait dans une recherche naïve à comparer les mots possibles les uns après les autres. Ce qui ferait que l'on compare chaque caractère du texte en moyenne le nombre de caractère que contient le mot que l'on recherche.

Sauf que certaines de ces comparaisons ne sont pas utiles, en effet si on

se l'on compare le mot morte dans ce texte au mot que l'on recherche mordue on remarque que la première comparaison retournant faux serait celle entre t et d. Dans ce cas-là on sait que l'on n'a pas besoin de comparer le mot orte à mordue tout simplement parce qu'aucune lettre ne ses répète dans le mot que l'on recherche. Par conséquent, nous ferons notre prochaine comparaison sur te.

En suivant cette logique on affecte à chaque caractère du mot que l'on recherche une retenue, celle-ci est calculée en déterminant le plus long suffixe se terminant par ce mot (sans l'inclure) égale au plus grand préfixe du mot.

On aura par exemple:

$$\mathbf{F=[m,a,m,a,m,i,a]} \rightarrow \mathbf{C=[-1,0,-1,0,-1,3,0,0]}$$

$$\mathbf{F=[c,h,i,c,h,a]} \rightarrow \mathbf{C=[-1,0,0,-1,0,2,0]}$$

$$\mathbf{F=[S,a,r,g,o,n]} \rightarrow \mathbf{C=[-1,0,0,0,0,0,0]}$$

Pour chaque caractère du mot que l'on recherche, dès que celui-ci ne correspond pas dans le texte, on incrémente la recherche du négatif de la retenue du caractère.

Prenons ici le mot mamamia, pour déterminer la retenue du caractère i nous calculons le plus grand suffixe se terminant par m dans le mot égal au plus grand préfixe du mot, nous nous retrouvons avec mam présente en début et fin de mot (sans y inclure i et à), mam étant de taille 3 la retenue d'i et 3. Nous répétons cette opération pour chaque lettre du mot. Il y a aussi une retenue en fin de mot dans le cas où le mot est bien trouvé dans le texte, dans ce cas là il n'est pas impossible qu'une apparition de ce mot en cache une autre. Par exemple la chaîne de caractères mamama est présente trois fois dans le mot mamamamama.

L'avantage d'une telle technique est de diminuer drastiquement le nombre de comparaisons nécessaires.

La retenue peut être adaptée pour un automate. Il est maintenant nécessaire de préciser que la retenue ne fonctionne correctement qu'avec un facteur, c'est-à-dire un mot simple, pour les regs ex constitués de caractères spéciaux (union, closure etc.) nous devons impérativement adapter notre méthode de calcul de la retenue.

C'est dans notre projet ce que nous avons fait, nous calculons pour

chaque noeud le plus long suffixe fixe de ce noeud égal au plus grand préfixe fixe de l'automate.

Nous appelons préfixe et suffixe fixe un ensemble de noeud qui forme une descendance directe (le premier noeud n'a pour descendant que le suivant dans la chaîne).

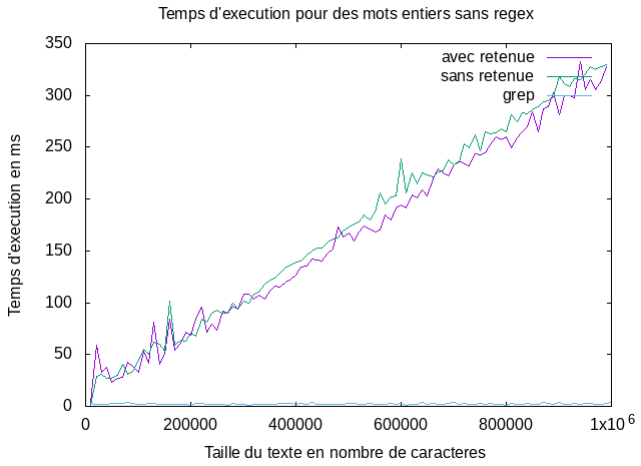
Un problème évident qui survient dans cette technique est l'efficacité de la retenue diminuera avec les unions et encore plus avec les closures. Et c'est en effet ce que nous observons, plus les regex testées contiennent des closures et plus l'efficacité de la retenue diminue.

Part II

expérimentation

1 La recherche d'un mot

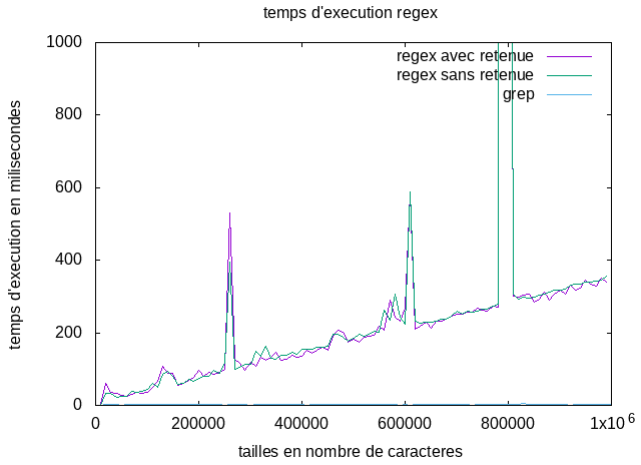
Graphe de comparaison des temps de recherches d'un mot avec et sans utilisation de l'algorithme KMP (construction de l'automate non pris en compte).



Nous pouvons observer que plus le texte dans lequel nous effectuons la recherche est long et plus l'usage de l'algorithme KMP nous fait gagner en temps.

2 La recherche d'une RegEx

Graphe de comparaison des temps de recherche d'un regex avec et sans notre adaptation de l'algorithme KMP (construction de l'automate non pris en compte).



Nous constatons plusieurs choses:

1. D'une part, que la commande grep est beaucoup plus efficace que notre implémentation ce qui n'est pas étonnant.
2. D'autre part, nous constatons que la recherche avec retenue est légèrement plus efficace dès lors que la regex est complexe. La recherche avec retenue s'appuie sur l'algorithme kmp qui est plus efficace que la méthode générale pour trouver une chaîne de caractère dans un texte.

Encore une fois l'usage de notre version de l'algorithme KMP se révèle plus efficace avec des textes longs, mais ce gain en efficacité n'est pas aussi évident qu'avec des mots. Comme nous l'avons déjà précisé il n'est pas facile d'adapter l'usage d'une retenue à une expression régulière.
En effet plus celle-ci contient de caractères spéciaux et moins l'usage de l'algorithme KMP se révèle pertinent.

Part III

Conclusion

Pour conclure, nous pouvons, à la lumière des différents éléments expérimentés durant ce projet, observer que notre méthode de recherche de chaînes de caractères n'est pas particulièrement efficace dans le traitement des expressions régulières.
En effet, plus celles-ci sont grandes et comportent de nombreuses opérations et plus l'usage de l'algorithme KMP se rélève inefficace.
Une solution future serait de mieux optimiser les automates pour l'usage de KMP mais aussi de mieux optimiser notre implémentation de celui-ci.