Rapport de Projet: Automate - Clone de egrep

Auteur: Mickael VILAYVANH et Ryad AITTOUARES

Dans le cadre du projet de DAAR, nous devions implanter la construction de l'automate à partir d'une expression régulière fournie aussi appelé **motif** afin de palier au problème de reconnaissance de motif dans un texte. Dans ce rendu, vous trouverez en Section 1 une définition courte du problème, l'implantation en automate comme décrite dans le chapitre 10 de l'ouvrage **Fondations of computer science** de A.Aho et J.Ullman en Section 2 ainsi que quelques tests de performance temporels en Section 3 entre la version **egrep native de unix écrite en C optimisée**, la version de **egrep implantée par nos soins** ainsi qu'un egrep utilisant le **pattern matcher de la classe Java Pattern** sur différents textes pour un même pattern.

1. Introduction au problème

Pour un texte donné, il serait pratique de pouvoir en un coup d'oeil surligner en couleur tous les mots que l'on souhaite reconnaître. Humainement, il est plutôt fastidieux de parcourir un texte de milliers de pages mais facile mais extrêmement long de reconnaître tous **les mots stricts** comme **"baguette"** ou bien **"bois"** mais pour des reconnaissances plus flexibles comme par exemple un mot commençant par **"sa"** pouvant être suivi de soit **"u"** ou **"l"** et terminant par **"t"**, la tâche devient d'autant plus compliquée car il y a plusieurs mots composables qui sont **"sat"**, **"saut"**, **"salt"**. Naivement, ces deux tâches sont très longues et même en délégant le parcours à une machine plutôt qu'un humain, le texte sera parcourrut plus rapidement mais le problème de reconnaissance demeure le même. Pour palier à cette question de reconnaissance de motif dans un texte, plusieurs stratégies ont été présentés pour ces deux types de motifs spécifiés précedemment: ceux étant des motifs stricts donc **uniquement des concaténations de lettres** et ceux **utilisant les opérateurs des expressions régulières** pour composer un motif. On distinguera les deux types de motif en notant **motif strict** et motif flexible ou bien **regex**. Par ailleurs, nous nous restreignons aux opérateurs étoile (*), alternance (|), concaténation, plus (+) et point (.) et au caractère ASCII pour le regex.
Plusieurs algorithmes comme l'algorithme de KMP ou bien celui de Boyer-Moore permettent de résoudre le problème pour les motifs stricts en complexité linéraire mais ceci n'est pas l'objet principal de ce rendu donc nous n'irons pas plus en détail à ce propos (cf .cours). Concernant les motifs regex, nous proposons dans ce rendu l'implantation de la méthode de construction d'automate mentionnés dans l'ouvrage cité précédemment qui se fait en plusieurs étapes: construction d'un arbre syntaxique du motif, transformation en automate non déterministe à epsilon-transition et enfin la déterminisation de cet automate. La minimisation ne sera pas faite dans ce rendu.

2. Implantation et structure de donnée

2.1 Construction de l'arbre syntaxique

La construction de l'arbre syntaxique est faite à partir de la classe Java fournie dans le sujet mais revenons brièvement dessus pour expliquer comment est construit l'arbre. On

considère le regex suivant abc*. Pour construire son arbre associé, on veut qu'il soit formatté pour suivre la priorité des opérateurs décrites dans les spécifications unix à savoir le groupement "(expr)" est prioritaire, puis l'opérateur "*", ensuite la concaténation et enfin l'alternance "|". Les lettres seront toujours des feuilles et les opérateurs seront toujours des noeuds internes de l'arbre. L'opération la plus prioritaire sera toujours la plus éloignée de la racine de l'arbre et la raison est simple, lors du parcours pour la construction de l'automate, on lit en remontant à partir du noeud le plus en bas de l'arbre et on remonte. Ainsi le dernier noeud exécuté pour la création de l'automate sera la racine de l'arbre.

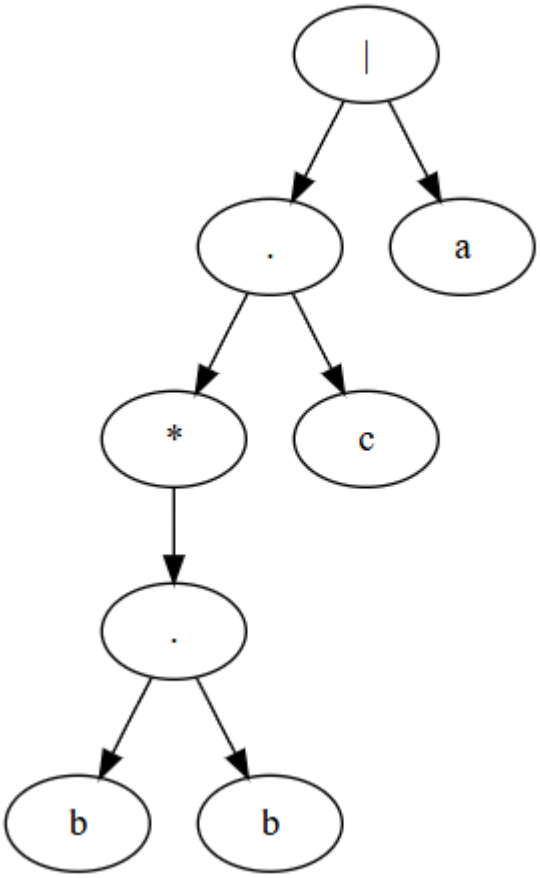

Figure 1. Arbre pour a|(bb)*c

Ici, on lit l'expression régulière: soit on a "a" ou bien un bloc répétant le motif "bb" suivi d'un "c". La façon de le réprésenter sous la format UNIX est importante pour la construction de l'automate mais elle diffère de la façon dont on le lit naturellement.
Le fichier java ainsi fourni permet de **parser une chaine de caractère représentant une expression régulière et de le formater au format UNIX** à la différence que dans la construction, le fils droit sera l'opérateur, à l'inverse de l'arbre en Figure 1. mais cela revient quasiment au même pour le parcours.

2.2. Parsing de l'arbre vers l'automate

Pour représenter un automate, on définit une classe Automaton A (T, Te, Ed, Ef, S) qui possède un ensemble T qui définit les transitions labellé par une lettre, un ensemble Te pour les transitions epsilon, un ensemble Ed des états de départ, un ensemble Ef des états finaux et un ensemble S de lettre de l'alphabet de l'automate. Les ensembles T et Te sont des

maps, Ed et Ef sont des sets particuliers et S est un set. Ed et Ef agissent comme une pile contenant des éléments uniques. Cette façon de les représenter est essentielle pour la construction de l'automate car cela nous permettra de dépiler et empiler les états lors de la fusion de deux automates.
La première étape consiste à transformer l'arbre syntaxique en un automate non déterministe à epsilon transition. Enfin, en partant de cet automate non déterministe, il faudra le déterminiser en regroupant les fermetures epsilons dans des états qui n'ont que des transitions étiquettées non epsilon et toute différentes. En pratique, il faudrait le minimiser ensuite pour retirer les états redondants mais cela ne sera pas fait pour ce rendu. Une fois l'automate déterministe et fini, il est utilisable pour reconnaître efficacement un motif dans un texte.
Revenons sur la structure de donnée. Nous avons dit que Ed et Ef sont des set en lecture LIFO et S est un set. T est une map dont les clés sont les états de départ et les valeurs sont une autre map dont la clé est la lettre et la valeur un set d'état d'arrivée. Cela permet de représenter les transitions sans faire intervenir un nombre inconsidérable de transitions vides comme l'aurait pu faire apparaître l'implantation en tableau. Te est aussi une map dont la clé est l'état de départ et la valeur est un set d'état d'arrivée. Les états sont des entiers et les lettres sont aussi des entiers représentant le code ascii associé.

```
private final Map<Integer, Map<Integer, Integer>> states = new HashMap<>();
private final Map<Integer, List<Integer>> epsilons = new HashMap<>();
private final Stack<Integer> acceptings = new Stack<>();
private final Stack<Integer> starts = new Stack<>();
private final Set<Integer> alphabet = new HashSet<>();
```

Figure 2. Champs privés de la classe Automaton

Pour démarrer la construction de l'automate à epsilon transition, on suit le fonctionnement décrit dans l'ouvrage fourni. Il faut partir des automates les plus basiques pour ensuite les fusionner ou bien les transformer pour en former un nouveau et ainsi de suite. On initialise une variable nstate à 0 qui permet d'étiquetter les différents états de l'automate. On démarre la lecture de l'arbre en lisant en postfixe, ie les sous-arbres d'abord puis le noeud parent ensuite. En particulier, cela signifie qu'on forme d'abord les automates basiques en allant au feuille, puis on cascade les transformations lors de la remontée vers les noeuds internes. Cette variable sera incrémentée à chaque création d'un nouvel état. Les automates basiques sont ceux qui vont de l'état e1 vers l'état e2 par une transition étiquetté x. Ainsi, le cas de base nécessite de créer deux états avec une transition de e1 vers e2 avec e1 l'état de départ et e2 l'état final acceptant. Cela est la même chose pour le cas des transitions epsilon à la différence que l'étiquetage n'est pas le même. Maintenant que nous savons construire l'automate le plus basique, nous pouvons passer à la transformation.
Plusieurs transformations sont possibles: la fusion si l'opérateur est la concaténation ou bien l'alternance et une transformation si l'opérateur est l'étoile. La concaténation est la plus simple à faire, soit ed1 (resp. ed2) l'état de départ de l'automate R1 (resp. R2) et ea1 (resp. ea2) l'état d'arrivée de R1 (resp. R2). Pour fusionner R1 et R2 et avoir un automate représentant leur concaténation, il suffit de placer une transition epsilon entre ea1 et ed2. Dans la structure de donnée, cela représente le fait de dépiler les deux derniers états de départ de Ef et de même pour les états d'arrivée Ef et de suivre cette procédure. Une fois la fusion terminée, il faut empiler l'état de départ issu de la fusion, ie. ed1 et empiler l'état d'arrivée issu de la fusion, ie. ea2.

```
private static void transformConcat(Automaton current) {
     int r1end, r2start, r1start;
     r1start = current.starts.pop();
     r1end = current.acceptings.pop();
     r2start = current.starts.pop();
     current.epsilons.get(r1end).add(r2start);
     current.starts.push(r1start);
 }
```

Figure 3. Fonction pour construire la concaténation

On reproduit les différentes manières d'assemblage décrites pour les 2 autres opérations de la similairement mais adapter à la transformation respective et cela nous permet d'effectuer la transformation vers l'automate à epsilon transitions en remontant l'arbre syntaxique et en cascadant les transformations. Par ailleurs, par construction d'un regex, il est impossible de faire la concaténation ou l'alternance s'il n'y a pas au moins 2 automates intermédiaires A1 et A2 pouvant former la transformation A étant donné que la traduction en arbre syntaxique d'une concaténation implique qu'il y ait 2 sous arbres représentant aussi un automate. Lorsqu'on revient à la racine, les ensembles T, Te, Ed, Ef et S seront remplis et décriveront l'automate Ae fini non-déterministe à epsilon transition.

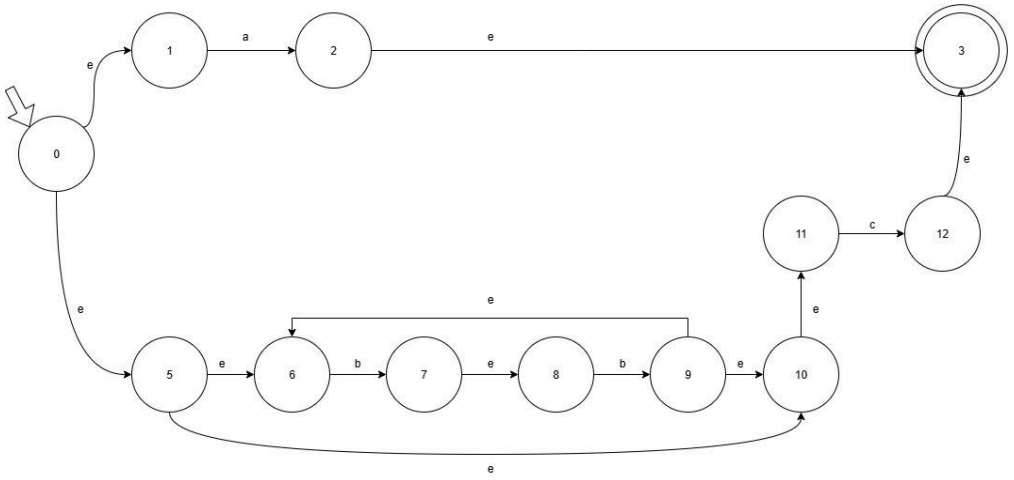

Figure 2. Automate à epsilon transition non-déterministe fini

Pour déterminiser Ae, on effectue **le parcours en largeur** et il faut **réunir les états superflues** qui ont des transitions epsilons sortantes. A la fin de la transformation en automate à epsilon transition, celui-ci aura **obligatoirement un seul état de départ et un seul état d'arrivée**. Le but sera de faire cascader les états qui sont sur le chemin entre deux états e1 et e2 et de transition epsilon. Par exemple, si on considère e1 vers e2 labellé par epsilon et e2 vers e3 étiquetté t, on peut réduire en un état nv1 qui fusionne e1 et e2 et nv1 transite vers e3 par t.
Un autre cas possible est d'avoir un état e1 qui va vers e2 qui est final par une transition epsilon. Dans ce cas, e1 et e2 fusionne en devenant nv1 et nv1 est final.
En effectuant ceci lors du parcours en largeur, on peut créer un deuxième automate qui est déterministe et fini à partir de celui non-déterministe à epsilon transition. C'est l'implantation qui est proposé dans ce rendu, on crée un ensemble de noeud visité et un ensemble de

noeud à traiter. On commence par l'unique point d'entrée e0 de l'automate et on ajoute dans les états à traiter les états dont e0 possèdent des transitions sortantes. On effectue ensuite les procédures que l'on a défini selon les cas et on continue jusqu'à ce que tous les noeuds ont été parcourru.

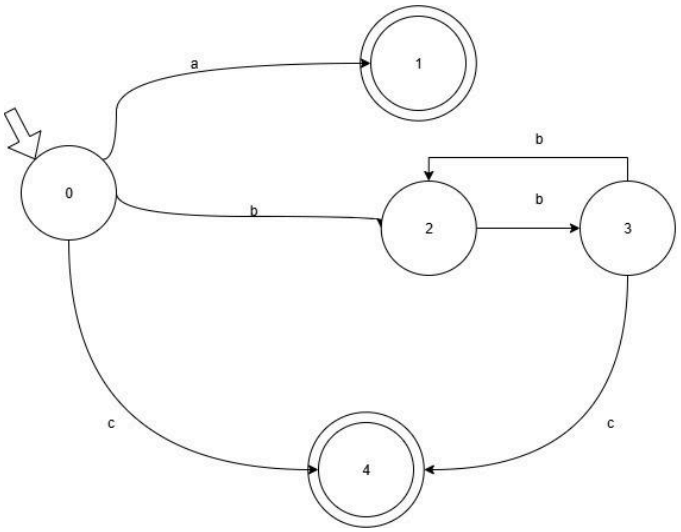

Figure 3. Automate après la transformation en DFA

Voici ce que renvoient notre egrep et le egrep unix sur le texte fourni pour le regex S(a|r|g)*on

```
432:    state--Sargon and Merodach-baladan--Sennacherib's attempt
436:    under the Sargonids--The policies of encouragement and
949:that empire's expansion, and the vacillating policy of the Sargonids
1016:to Sargon of Akkad; but that marked the extreme limit of Babylonian
1019:Arabian coast. The fact that two thousand years later Sargon of
1788:A: Sargon's quay-wall. B: Older moat-wall. C: Later moat-wall of
1820:It is the work of Sargon of Assyria,[44] who states the object of
1827:upon it."[45] The two walls of Sargon, which he here definitely names
1832:the quay of Sargon,[46] which run from the old bank of the Euphrates
1833:to the Ishtar Gate, precisely the two points mentioned in Sargon's
1844:A: Sargon's quay-wall. B: Older moat-wall. O: Later moat-wall of
1853:quay-walls, which succeeded that of Sargon. The three narrow walls
1868:Sargon's earlier structure. That the less important Nimitti-Bêl is not
1870:in view of Sargon's earlier reference.
1914:excavations. The discovery of Sargon's inscriptions proved that in
1919:precisely the same way as Sargon refers to the Euphrates. The simplest
3548:[Footnote 44: It was built by Sargon within the last five years of
5555:Sargon of Akkad had already marched in their raid to the Mediterranean
6065:Babylonian tradition as the most notable achievement of Sargon's reign;
8957:for Sargon's invasion of Syria. In the late omen-literature, too, the
10920:Sargon's army had secured the capture of Samaria, he was obliged to
10930:Sargon and the Assyrian army before its walls. Merodach-baladan was
10937:After the defeat of Shabaka and the Egyptians at Raphia, Sargon was
10941:their appearance from the north and east. In fact, Sargon's conquest of
10946:Sargon was able to turn his attention once more to Babylon, from
10954:On Sargon's death in 705 B.C. the subject provinces of the empire
11001:party, whose support his grandfather, Sargon, had secured.[43] In 668
11244:Sargon's death formed a period of interregnum, though the Kings' List
12453:fifteen hundred years before the birth of Sargon I., who is supposed
```

```
432:    state--Sargon and Merodach-baladan--Sennacherib's attempt
436:    under the Sargonids--The policies of encouragement and
949:that empire's expansion, and the vacillating policy of the Sargonids
1016:to Sargon of Akkad; but that marked the extreme limit of Babylonian
1019:Arabian coast. The fact that two thousand years later Sargon of
1788:A: Sargon's quay-wall. B: Older moat-wall. C: Later moat-wall of
1820:It is the work of Sargon of Assyria,[44] who states the object of
1827:upon it."[45] The two walls of Sargon, which he here definitely names
1832:the quay of Sargon,[46] which run from the old bank of the Euphrates
1833:to the Ishtar Gate, precisely the two points mentioned in Sargon's
1844:A: Sargon's quay-wall. B: Older moat-wall. O: Later moat-wall of
1853:quay-walls, which succeeded that of Sargon. The three narrow walls
1868:Sargon's earlier structure. That the less important Nimitti-Bêl is not
1870:in view of Sargon's earlier reference.
1914:excavations. The discovery of Sargon's inscriptions proved that in
1919:precisely the same way as Sargon refers to the Euphrates. The simplest
3548:[Footnote 44: It was built by Sargon within the last five years of
5555:Sargon of Akkad had already marched in their raid to the Mediterranean
6065:Babylonian tradition as the most notable achievement of Sargon's reign;
8957:for Sargon's invasion of Syria. In the late omen-literature, too, the
10920:Sargon's army had secured the capture of Samaria, he was obliged to
10930:Sargon and the Assyrian army before its walls. Merodach-baladan was
10937:After the defeat of Shabaka and the Egyptians at Raphia, Sargon was
10941:their appearance from the north and east. In fact, Sargon's conquest of
10946:Sargon was able to turn his attention once more to Babylon, from
10954:On Sargon's death in 705 B.C. the subject provinces of the empire
11001:party, whose support his grandfather, Sargon, had secured.[43] In 668
11244:Sargon's death formed a period of interregnum, though the Kings' List
12453:fifteen hundred years before the birth of Sargon I., who is supposed
12635:and 2 (Sonderabdruck, 16 pp.); see further, pp. 304, 308.]
```

Figure 4. egrep unix à gauche et notre egrep à droite

Les deux fournissent le même résultat ce qui est déjà bon signe ! Il faut maintenant effectué des tests de performance et des tests sur d'autres regex (tester d'autres motifs n'est pas intéressants donc nous ne le feront pas car c'est ininteressant)

3. Test de performance

Pour tester la performance de notre egrep, on a utilisé ChatGPT pour générer un script bash que l'on a réajusté qui va exécuter 20 fois chacun des programme et faire la moyenne pour chacun. Ensuite, on a générer un deuxième script python pour dresser un diagramme en barre du temps d'exécution en ms.

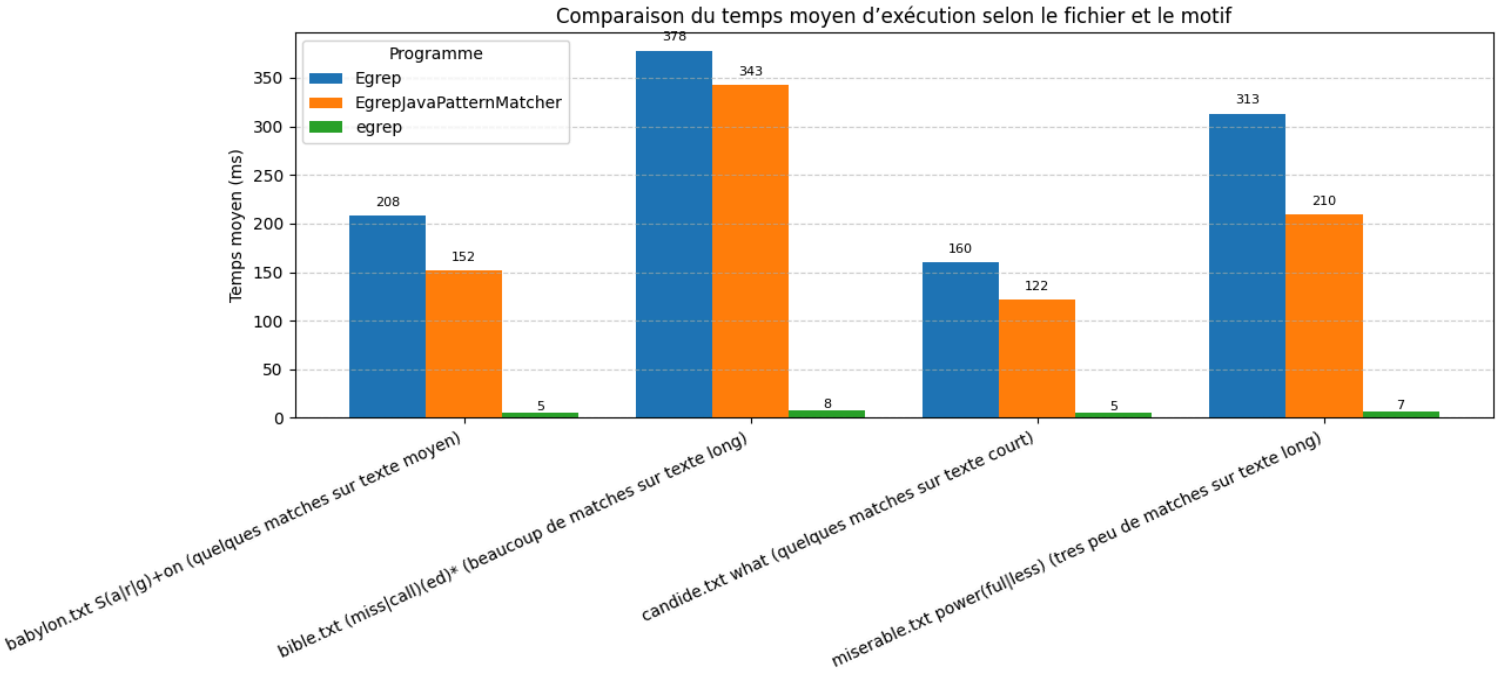

Figure 5. Diagramme sur le texte babylon fourni dans le sujet avec le motif S(a|r|g)*on

On s'aperçoit qu'en globalité, notre egrep est plus lent que celui natif à unix pour les deux implantation. Il y a une raison à celà, déjà les tests ont été effectué grossièrement, c'est à dire en prenant en compte le déploiement de la jvm, du garbage collector etc… Ausi, on a inclut dans le calcul du temps d'exécution l'ouverture du fichier ce qui est extrêmement coûteux mais comme nous ne savons pas si le egrep unix l'effectue aussi, nous l'avons inclut dans les tests. Pour des temps plus fins, il aurait fallu encadrer la fonction/méthode qui effectue l'analyse mais pour rester fidèle au fait qu'on ne sait pas comment est écrit egrep unix, on a préféré inclure tous les temps depuis le début de l'exécution. Mais celà décrit la lenteur de java. Pour donner une idée, les temps moyen mesuré lorsqu'on encadrait la méthode qui effectue la même tâche que egrep, on était à l'ordre de 70ms pour notre egrep en moyenne et 50ms en moyenne pour le egrep avec la classe de pattern matching sur le texte de Babylon. On constate notamment un facteur 30/40 de lenteur pour l'implantation en Java.

Comparons maintenant les résultats entre notre EGrep (E1) avec l'algorithme décrit dans l'ouvrage et celui venant de la classe Pattern en Java (E2). On s'appercoit que E2 est légèrement plus rapide que E1. D'après quelques recherches effectuées pour savoir ce qui est fait derrière le pattern matcher java, il s'avère qu'il s'appuie sur l'algorithme de Aho Corasick qui est légèrement plus efficace et aussi que l'automate n'est pas DFA et reste

NFA, donc il y a potentiellement moins de temps attribué à repasser l'automate pour le transformer en DFA qui entre en compte. On aurait certainement été plus rapide en implémentant KMP pour le motif "what" sur le texte de Candide.
Hormis les différences en temps d'exécution, on se rend compte que ce temps pour chacun est proportionnel à la taille du fichier, ce qui était attendu.

4. Conclusion

Notre Egrep est fonctionnel et reconnaît les motifs dans un texte donné. On se rend compte que le problème de reconnaissance de motif dans un texte nécessite de comprendre comment les automates fonctionnent mais aussi les autres algorithmes qui existent dans la littérature et que nous n'avons pas encore implanté comme KMP et Boyer-Moore. Cependant, nous savons par le cours que KMP sera la stratégie à faire pour les motifs contenant uniquement des concaténations. Cela sera obligatoire pour la suite car ce egrep sera utilisé pour le projet 3 de DAAR dans la tâche réalisé par le moteur de recherche et il faut impérativement qu'il soit efficace. De plus, humainement, nous recherchons tous des mots avec des lettres concaténées. Seuls les programmeurs fous recherchent en regex.