Rapport Daar : Copie de la commande egrep

Zakaria DJEBBES

8 Octobre 2022

Table des matières

Table des figures

1 Définition du problème

Nous voulons réaliser une copie simplifiée de la commande `egrep` disponible sur Linux. Cette commande prend en entrée un fichier et une expression régulière et rend en sortie toutes les occurrences de l'expression régulière dans ledit fichier.

1.1 Expressions Régulières

Une expression régulière est une suite de caractères qu'on appelle plus simplement « motif » ou « pattern » décrivant un ensemble de chaînes de caractères.

Il est possible de faire plusieurs manipulations sur ces patterns à l'aide de caractères spéciaux :

— ***** Répétition d'un caractère zéro ou une fois.

— . N'importe quel caractère une fois.

— | Le motif à droite ou le motif à gauche.

— + Répétition d'un caractère au moins une fois.

— ? Le caractère zéro ou une fois.

Ces expressions régulières sont ensuite utilisées pour chercher les occurrences des chaînes de caractères dans un texte fourni en entrée.

1.2 Automates

Il est donc de transformer une expression régulière en un arbre syntaxique, suivi d'un automate fini non déterministe et enfin en un automate déterministe algorithmiquement, puis d'utiliser cet automate pour trouver toutes les occurrences de cette expression régulière dans un texte. [AU92]

Pour cela, il suffira de tester chaque caractère du texte et voir s'il est franchissable sur notre automate, si oui et le nouvel état est final alors ceci est une occurrence. Sinon, si ce caractère n'est pas franchissable, alors, il n'y a pas d'occurrences du motif.

Il est également possible de minimiser l'automate déterministe [AU92], mais vu que nous travaillons sur des expressions régulières de petite à moyenne taille, il n'est pas très utile de le faire.

1.2.1 Complexité et Critiques

La construction de l'automate en elle-même ne devrait pas prendre énormément de temps, vu que nous travaillons sur des expressions régulières de petite taille. Cela dit, elle est quand même de complexité $O(n^2)$ [AU92] dans le pire des cas, par exemple une expression de ce genre `(0+1)*1(0+1)...(0+1)`

La recherche de motifs sur ces automates est d'une complexité $O(n^2)$ telle que n est le nombre de caractères du texte en entrée. Si le texte est découpé en un nombre m de lignes, alors la complexité devient simplement la somme des complexités par lignes, soit $O(n^2 * m)$.

Même s'il existe des moyens de minimiser les automates en sortie, plus le pattern est grand et complexe, plus temps d'exécution sera long. Surtout dans les cas des patterns se finissant avec un `.*`. Voir un `.*` simplement, dans ce pour chaque caractère, on passera sur tous les autres caractères [Poi].

1.2.2 Alternatives

Pour des expressions régulières complexes incluant des manipulations telles que des + ou des *, il n'est pas possible de trouver une autre façon, plus rapide, de trouver les occurrences d'un motif dans un texte.

Cela dit, il y a une bonne nouvelle, si l'on cherche un motif formé uniquement de concaténations, ce qui est le cas de la majorité des recherches faites sur des moteurs de recherches, car très ressemblante au langage humain (Personne n'utilise d'expression régulière dans la vraie vie !), alors, il est possible de trouver des algorithmes de pattern matching plus intéressants tels que KMP.

1.3 Algorithme de Knuth-Morris-Pratt

L'algorithme de Knuth-Morris-Pratt (ou KMP) comme les automates permet de trouver les occurrences d'un motif simple formé uniquement de concaténation de caractères dans un texte. [Tho01]

Sa particularité réside en un pré-traitement de la chaîne, qui fournit une information suffisante pour déterminer où continuer la recherche en cas de non-correspondance. Ainsi, l'algorithme ne réexamine pas les caractères qui ont été vus précédemment, et donc limite un nombre conséquent de comparaisons nécessaires.

Cet algorithme se déroule en deux phases [Tho01] :

1. Construire le tableau des « décalages » indiquant pour chaque position un point de retour, c'est-à-dire la prochaine position où peut se trouver une occurrence potentielle de la chaîne.
2. La deuxième phase effectue la recherche à proprement parler, en comparant les caractères de la chaîne et ceux du texte. En cas de différence, il utilise le tableau pour connaître le décalage à prendre en compte pour continuer la recherche sans faire des tests inutiles, car assurer d'échouer.

1.3.1 Complexité et Critiques

Malgré une complexité linéaire de $O(n)$, cet algorithme n'est pas idéal car... bla bla.

Il est intéressant de noter que cet algorithme, bien que plus rapide que les automates, n'est pas le plus approprié pour rechercher des motifs dans des langages complexes (comme des mots en français ou en anglais !) mais plus intéressant pour des grammaires avec un nombre de caractères réduit.

1.3.2 Alternatives

Il existe d'autres alternatives plus rapides pour des grammaires plus compliquées et surtout qui ressemblent aux langages humains tels que l'algorithme de Boyer–Moore.

2 Structures de données

J'ai choisi d'implémenter cette commande en Csharp mais toutes les structures sont valables pour d'autres langages !

2.1 Structure de données représentant l'automate

Listing 1 – Structure d'automate

```
public class Automata
{
    private List<int> initialStates;
    private List<int> finalStates;
    private List<List<int>> transitions;
}
```

1. **initialStates** : représente la liste des états initiaux de l'automate.
2. **finalStates** : représente la liste des états finaux de l'automate.
3. **transitions** : liste des transitions de l'automate, tel que chaque élément est lui-même une liste d'entiers, $transitions[0]$ est l'état de départ, $transitions[1]$ est l'état de fin et $transitions[2]$ est le caractère (en ASCII) avec lequel la transition est possible.

Cette structure modélise aussi bien un automate déterministe qu'un automate non déterministe.

2.2 Structure de données représentant le KMP

Listing 2 – Structure KMP

```
public class KnuthMorrisPratt
{
    private int[] carryOver;
    private string pattern;
}
```

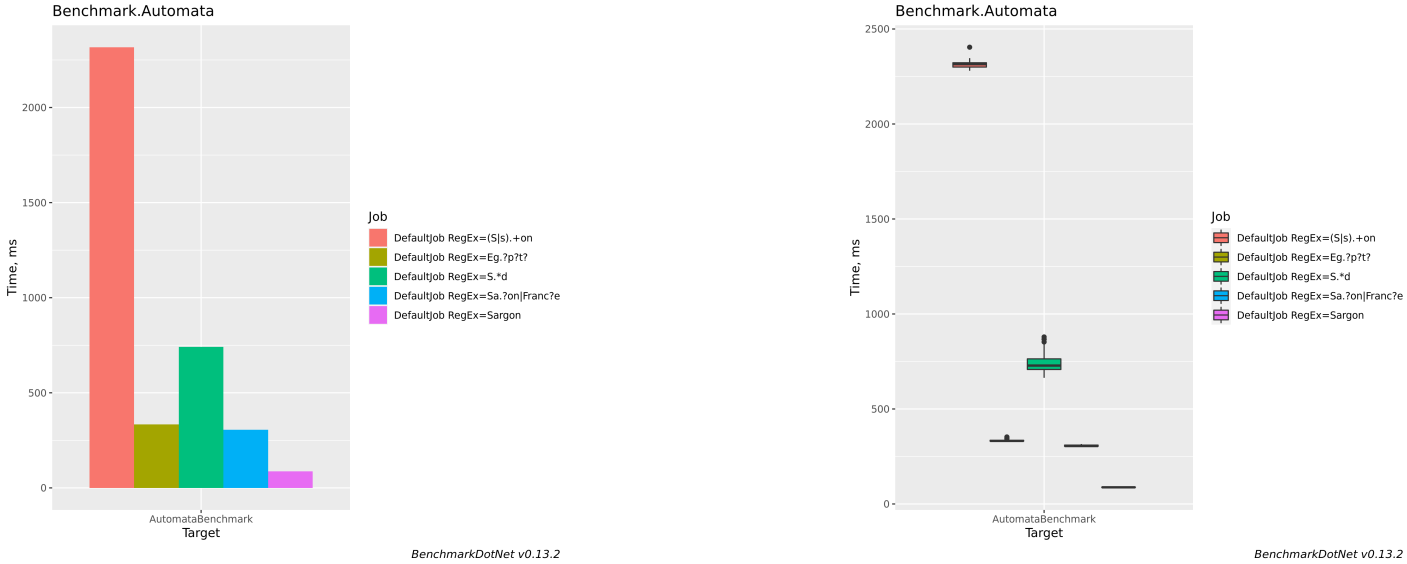
1. **carryOver** : un simple tableau d'entiers qui représente la table des carryovers.
2. **pattern** : le motif permettant de calculer la table *carryOver*

3 Tests et performances

3.1 Performances des Automates

Je prends le texte babylone sur le site de gutenberg project.

Je prends plusieurs expressions régulières de plus en plus compliquées pour lancer mes benchmarks sur le texte 100 fois et je prends la moyenne des temps d'exécution.



(a) Bar Plot

(b) Box Plot.

FIGURE 1 – Performences de l'automates sur différentes expressions.

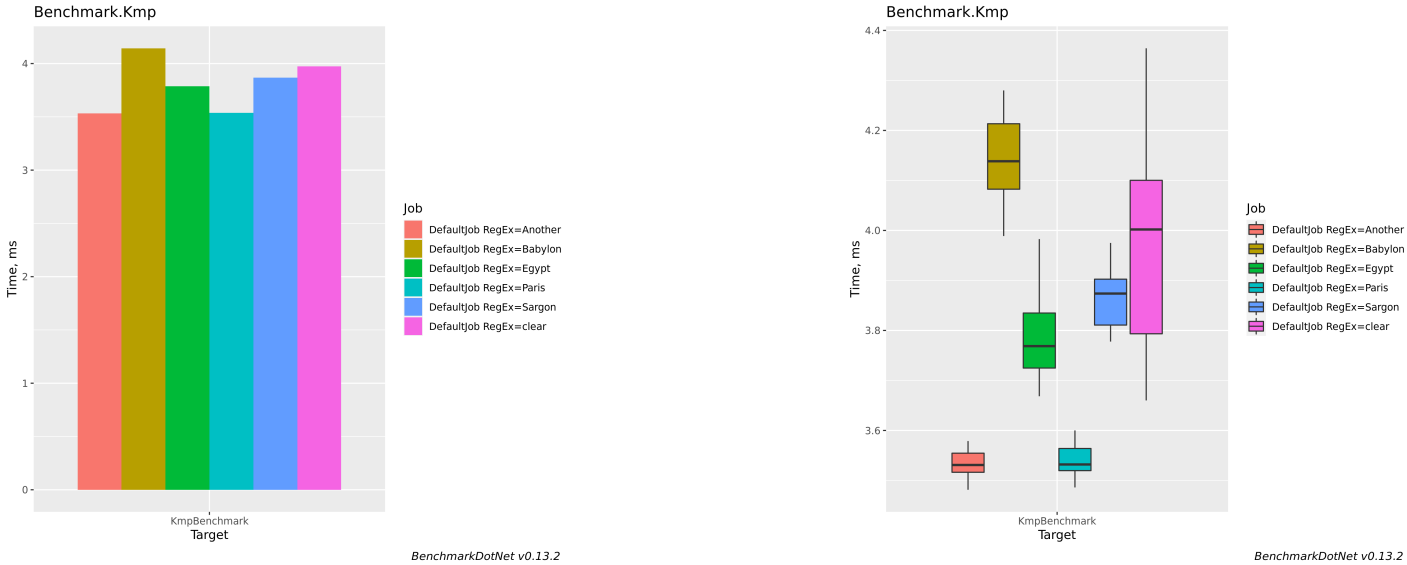
Il est évident que plus l'expression est compliquée, plus le temps moyen d'exécution est élevée. Et la disparité entre les résultats est claire, 35 ms environ pour des simples concaténations, mais des temps beaucoup plus élevés pour les autres cas.

Vu que le nombre d'états de l'automate augmente en fonction de sa taille et surtout dans les cas de l'utilisation des .+ et .* ou chaque caractère devra être testé une multitude de fois.

3.2 Performances du KMP

De la même manière, je prends le texte babylone sur le site de gutenberg project.

J'obtiens ainsi les résultats suivants pour le KMP :



(a) Bar Plot

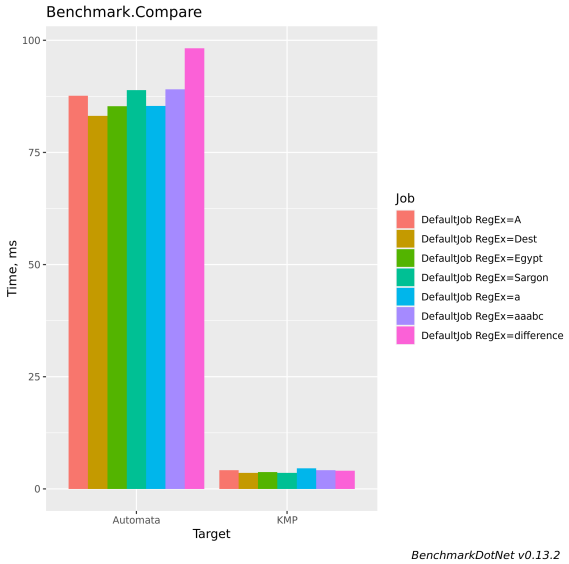
(b) Box Plot.

FIGURE 2 – Performences du KMP sur différentes expressions.

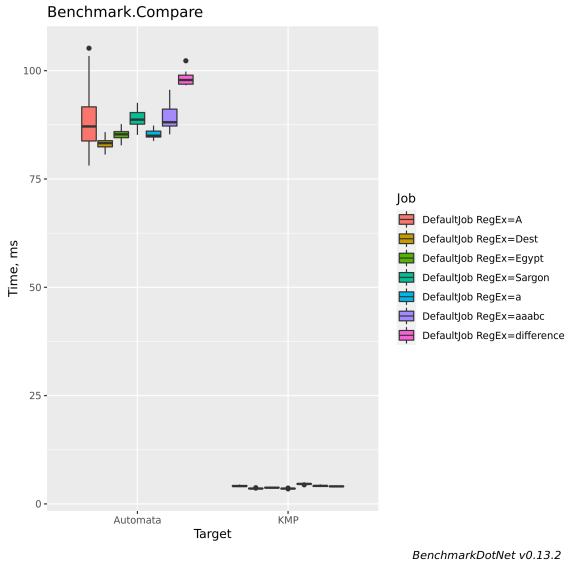
Le temps moyen semble stable, voir linéaire. Les benchmarks confirment que l'algorithme de KMP, comme prévu, est beaucoup plus intéressant en termes de temps d'exécution sur les simples concaténations !

3.3 Comparaison des performances

Toujours sur le même texte, je prends plusieurs expressions régulières et j'applique mes benchmarks en utilisant les deux méthodes simultanément.



(a) Bar Plot



(b) Box Plot.

FIGURE 3 – Comparaison des performences.

On remarque clairement, selon les benchmarks, que le temps d'exécution du KMP est beaucoup plus intéressant (4 à 5 ms) que les automates.

Le box plot montre la disparité de façon bien plus claire.

Les résultats sont (selon moi) totalement cohérents, le KMP avance plus vite, car il évite algorithmiquement des tests inutiles, tant dit que quel que soit le caractère, l'automate est obligé de faire au moins un test.

4 Conclusion

La recherche de motifs dans un texte peut se faire de plusieurs façons différentes, Automates, KMP ou autres algorithmes.

Les automates, bien que n'ayant pas une complexité intéressante, permettent de traiter des cas complexes du pattern matching, des répétitions...

La solution algorithmique est bien plus rapide, bien plus propre et bien plus facile à implémenter, mais ne permet de traiter que des concaténations simples. Cela dit, dans la plupart des cas, les utilisateurs de moteur de recherche utilisent des concaténations simples et non pas d'expression régulière.

Il convient donc de penser à cela lors de la création d'un moteur de recherche, et de choisir convenablement la façon la plus rapide de trouver les matchs dans un texte en fonction du motif demander.

Références

[AU92] Al Aho et Jeff Ullman. *Foundations of Computer Science (FCS)*. 1992. Chap. Chapter 10 Patterns, Automata, and Regular Expressions, p. 529-590.

[Tho01] et Clifford Stein Thomas H. Cormen Charles E. Leiserson Ronald L. Rivest. *Introduction à l'Algorithmique (IA)*. 2001, p. 923-931.

[Poi] Tutorials Point. "Learn Automata Theory (LAT)". In : (). url : `https://www.tutorialspoint.com/automata_theory/index.htm`.