Marsso Denn
Mougamadoubougary Mohamed

Rapport Projet 1 DAAR

1 - Définition du problème et des structure de données utilisées

Le but de ce projet est d'implémenter un moteur de recherche de motifs textuels basé sur les expressions régulières conformes à la norme ERE (Extended Regular Expressions), telle que définie par le standard POSIX. Il faut ensuite le tester (tout d'abord de manière fonctionnelle puis au niveau des performances).
L'objectif est de construire un programme équivalent à la commande egrep, capable de lire un fichier texte et d'afficher les lignes qui correspondent à un motif donné.

Le programme doit :

-Lire un motif RegEx saisi en argument.

- Lire un fichier texte ligne par ligne.

-Vérifier pour chaque ligne si une partie du mot correspond au motif.

-Afficher toutes les lignes correspondantes comme egrep.

Deux grandes stratégies de recherche sont proposées :

-Approche générale (Aho-Ullman) :
Transformation du motif en un arbre syntaxique en suivant la méthode du livre Aho-Ullman, puis en un automate non déterministe avec epsilon transition (NFA), ensuite en automate fini déterministe (DFA) et enfin en automate minimal (DFA) utilisé pour la recherche.

-Approche optimisée (KMP) :

Si le motif est réduit à une suite de concaténations, il s'agit de la recherche d'un facteur dans une chaîne de caractères : nous pouvons alors utiliser l'algorithme Knuth-Morris-Pratt (KMP) à la place de la machine de guerre décrite ci-dessus

Nous avons choisi de suivre la première stratégie :

RegEx → Arbre syntaxique → NFA → DFA → DFA minimal → Recherche dans le texte

Nous avons effectué tous nos tests sur le texte "babylone.txt", contenant plusieurs milliers de lignes.

Afin de suivre cette méthode, nous avons dû créer plusieurs structures de données :

RegExTree, NFA, DFA, RegEx :

-RegExTree correspond à l'arbre syntaxique.
Il contient :

root : un identifiant correspondant à un opérateur (CONCAT, ALTERN, ETOILE, DOT, PLUS) ou à un caractère ASCII.

sub : un tuple contenant les sous-arbres associés.

-NFA correspond à l'automate non déterministe avec transition epsilon.
Il contient:

start : l'état initial.

accept : l'état d'acceptation.

transitions : un dictionnaire Dict[int, List[Tuple[Optional[str], int]]] représentant les transitions étiquetées par un caractère ou par epsilon (ε).

Et des méthodes pour :

Créer de nouveaux états (newState()), ajouter des transitions (addEdge()), convertir le NFA en DFA (toDfa()).

Les transitions epsilon permettent de relier plusieurs sous-automates sans consommer de caractère.

-DFA correspond à l'automate fini déterministe.
Il contient:

start : état de départ.

accepts : ensemble d'états acceptants.

transitions : un dictionnaire Dict[int, Dict[str, int]] des transitions déterministes.

alphabet : ensemble des symboles utilisés.

Et des méthodes pour :
Identifier les états atteignables depuis l'état initial (reachable()), compléter l'automate en ajoutant un état puit pour les transitions manquantes (makeTotal()), réduire le nombre d'états par fusion des états équivalents (minimize()) et appliquer l'automate minimal sur une ligne de texte(search(text)).

-RegEx correspond à la classe qui encapsule l'ensemble du processus :
Elle contient:

CONCAT, ETOILE, ALTERN, DOT, PLUS : constantes représentant les différents opérateurs d'expression régulière.

_PREC : dictionnaire définissant la priorité des opérateurs.

pattern : motif saisi par l'utilisateur sous forme de chaîne de caractères (regex).

tree : arbre syntaxique (RegExTree).

Et des méthodes pour :

Découper le motif en une liste de symboles et d'opérateurs (parse()),
insérer explicitement les opérateurs de concaténation manquants (insertConcats()), convertir la liste de tokens en notation postfixée (toPostfix()), construire l'arbre syntaxique à partir de la forme postfixée (postfixToTree()), transformer l'arbre obtenu en un automate fini non déterministe (toNfa()).

2 - Analyse et présentation théorique des algorithmes connus dans la littérature

Nous allons ici expliquer les différents algorithmes connus que nous avons utilisés dans ce projet.

-NFA par construction de Thompson / Aho–Ullman

On transforme le RegEx en automate non déterministe (NFA) en assemblant des gadgets pour chaque opérateur (xy, *, |, ., +) et en reliant les sous-automates avec des transitions ε. La construction est linéaire et dépend de la taille du motif (O(m)) et sert de base à la construction de l'automate déterministe.

- Déterminisation, méthode des sous-ensembles

Le NFA est converti en DFA par la construction par puissances : chaque état du DFA représente un ensemble d'états du NFA.

- Minimisation du DFA

Une fois le NFA déterminisé en DFA, on minimise l'automate par raffinement itératif des classes d'états :

On part d'une partition initiale {acceptants} / {non-acceptants} et à chaque itération, on regroupe les états de chaque bloc par signature. Si un bloc contient plusieurs signatures différentes, on le scinde en autant de sous-blocs. On répète jusqu'à ne plus avoir aucune scission Donc dans le pire cas on est en O(n^2).
On a donc à la fin un automate avec un état par bloc (DFA minimal).

- Moteur à base de DFA

Une fois le DFA (minimal) construit, la recherche est quasi-linéaire. On parcourt chaque ligne et on applique le DFA sur cette ligne, indépendamment de la structure interne du motif (O(n log n)).

3 - Argumentation concise appuyant toute appréciation, amélioration, ou critique à propos de ces algorithmes existants dans la littérature.

L'efficacité de ces algorithmes est que la création de l'automate minimale à un coût très faible, et que le résultat est sûr. Cependant, l'étape de recherche (application du DFA) peut être extrêmement coûteuse et peut avoir un impact énorme sur le temps d'exécution de l'algorithme en fonction du nombre de lignes (Ce que nous pourrons observer sur les tests).

4 - Partie test fonctionnel

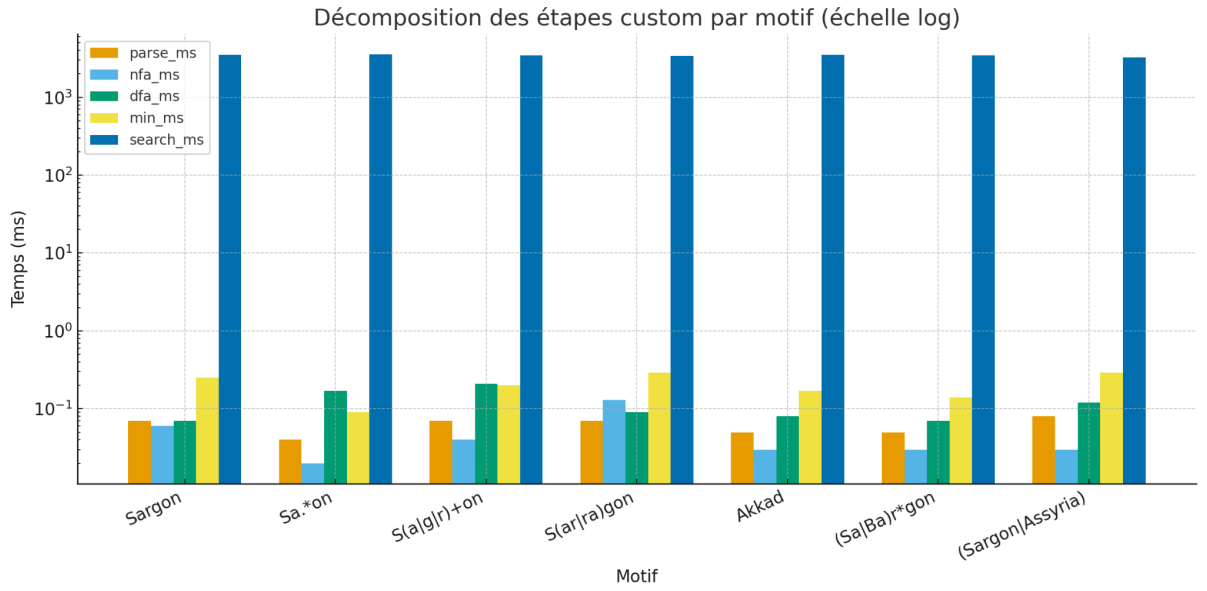
Pour les tests, nous n'avons pas récupéré de "testbeds" et n'avons pas utilisé la totalité de la base de donnée de gutenberg. Nos tests sont focalisés sur le texte de babylone. Ce texte contient beaucoup de ligne (assez pour pouvoir analyser les performances), nous avons ensuite testé différents motifs RegEx pour s'assurer que tous les opérateurs

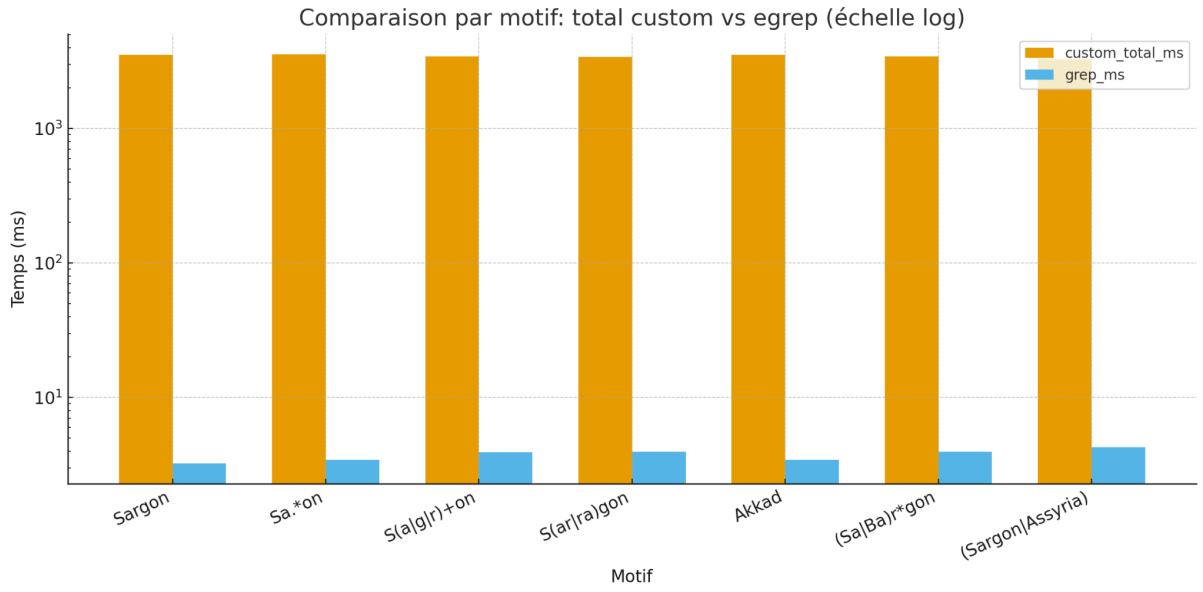
fonctionnent et effectuer une comparaison du résultat de notre programme avec la commande egrep (car notre programme est fait pour envoyer la réponse sous le même format que egrep).

5 - Partie test de performance

Pour le test de performance nous avons repris exactement les mêmes conditions que le test fonctionnel, mais nous avons séparé les différentes étapes de notre egrep customisé afin de pouvoir calculer le temps de chaque étape. Nous avons fini en comparant le temps total mis par notre programme pour effectuer les différentes recherches par rapport au temps total mis par egrep.

```
regex parse_ms nfa_ms dfa_ms min_ms search_ms custom_total_ms grep_ms
Sargon 0.07 0.06 0.07 0.25 3529.10 3529.54 3.25
Sa.*on 0.04 0.02 0.17 0.09 3564.78 3565.09 3.45
S(a|g|r)+on 0.07 0.04 0.21 0.20 3442.01 3442.52 3.92
S(ar|ra)gon 0.07 0.13 0.09 0.29 3408.51 3409.10 3.95
Akkad 0.05 0.03 0.08 0.17 3522.53 3522.87 3.43
(Sa|Ba)r*gon 0.05 0.03 0.07 0.14 3441.07 3441.36 3.94
(Sargon|Assyria) 0.08 0.03 0.12 0.29 3278.45 3278.97 4.26
TOTAL_CUSTOM_MS 24189.45
TOTAL_GREP_MS   26.20
```





6 - Analyse des résultats du test de performance et conclusion

On peut tout d'abord remarquer comme dit plus tôt lors de la description des algorithmes que ce qui prend beaucoup de temps. C'est l'application du DFA. Cela s'explique car pour se faire, nous effectuons une double boucle sur chaque ligne (le texte contenant beaucoup de lignes) cela atteint très rapidement un grand coût. On peut aussi remarquer que le egrep est beaucoup beaucoup plus efficace que notre algorithme. Cela s'explique sûrement car nous utilisons l'algorithme brute (pas l'optimisé) et que egrep utilise des algorithmes certainement plus poussés que ce que nous pouvons implémenter.
Pour conclure, on voit bien que le problème de recherche de motif est bien plus compliqué qu'il n'en a l'air, les coûts des algorithmes de recherches peuvent atteindre rapidement des proportions énormes si les bonnes optimisations ne sont pas utilisées en fonction de la recherche voulue, cependant avec les méthodes actuelles déjà existantes (comme egrep) on peut atteindre des performances excellentes.