Projet 1 DAAR 2025

Breton Noé

n°21516014

Boudrouss Réda

n°28712638

Sorbonne Université
France

Table des matières

1. Introduction

La recherche de motifs dans des fichiers textuels est une opération fondamentale en informatique, utilisée quotidiennement par des millions de développeurs et administrateurs systèmes. Les expressions régulières (regex), formalisées par Stephen Cole Kleene en 1956 dans le cadre de la théorie des langages formels, constituent l'outil privilégié pour décrire et rechercher des motifs complexes dans des chaînes de caractères. L'utilitaire Unix `egrep` (Extended Global Regular Expression Print), qui supporte la norme ERE (Extended Regular Expressions), est l'une des implémentations les plus utilisées de cette technologie.

1.2. Contexte

Ce projet vise à développer un clone fonctionnel de `egrep` supportant un sous-ensemble de la norme ERE POSIX. Les opérateurs implémentés sont :

— Les parenthèses pour le groupement
— L'alternation (`|`) pour le choix entre motifs
— La concaténation de motifs
— L'étoile de Kleene (`*`) pour la répétition
— Le point (`.`) comme caractère universel
— Les caractères ASCII littéraux

L'approche classique décrite par Aho et Ullman dans *Foundations of Computer Science* consiste à :

1. Parser l'expression régulière en un arbre syntaxique
2. Construire un automate fini non-déterministe (NFA) avec ϵ-transitions
3. Convertir le NFA en automate fini déterministe (DFA) par la méthode des sous-ensembles
4. Minimiser le DFA pour réduire le nombre d'états
5. Utiliser l'automate pour matcher les lignes du fichier

1.3. Démarche

L'exécution d'un automate peut s'avérer parfois couteuse en temps (NFA) ou en mémoire (DFA). Nous avons donc aussi implémenté des algorithmes de recherche de motifs littéraux pour améliorer les performances dans certains cas. Notamment :

— Knuth-Morris-Pratt (KMP) : recherche linéaire garantie O(n+m) pour les motifs courts
— Boyer-Moore : recherche optimisée pour les motifs longs avec heuristique du mauvais caractère
— Aho-Corasick : recherche multi-motifs pour les alternations de littéraux

D'autres améliorations ont été apportées pour gérer efficacement les fichiers volumineux :

— Analyse du pattern pour identifier les segments fixes
— Utilisation de Boyer-Moore ou Aho-Corasick pour éliminer rapidement les lignes non-candidates avant le matching regex complet
— Choix du meilleur algorithme en fonction de la complexité du pattern
— Traitement efficace de fichiers volumineux avec gestion mémoire optimisée

Nous nous sommes énormement inspirés de l'implémentation de GNU grep pour ce projet. Notamment, l'architecture par chunks et le préfiltrage sont directement inspirés de cette implémentation.

1.4. Architecture Technique

L'implémentation est réalisée en TypeScript dans une architecture monorepo comprenant :

- `lib` : bibliothèque core contenant tous les algorithmes (NFA, DFA, KMP, Boyer-Moore, Aho-Corasick, préfiltrage)
- `cli` : interface en ligne de commande compatible avec `egrep`

Le projet inclut une suite de tests exhaustive (Vitest) et des benchmarks de performance sur des corpus du projet Gutenberg.

Le choix de TypeScript a été fait pour son coté fonctionnel et son typage statique, mais aussi pour sa rapidité d'exécution après transpilation en JavaScript.

1.5. Organisation du Rapport

Ce rapport présente d'abord les fondements théoriques des algorithmes implémentés (Section 2), puis détaille les stratégies d'optimisation développées (Section 3). La méthodologie de test est exposée en Section 4, suivie d'une analyse de performance comparative (Section 5). Nous concluons par une discussion des résultats et des perspectives d'amélioration (Section 6).

2. Fondements Théoriques

2.1.3 Arbre Syntaxique Abstrait (AST)

Pour faciliter la manipulation et la transformation des expressions régulières, nous les représentons sous forme d'arbre syntaxique abstrait (Abstract Syntax Tree, AST).

Nous définissons un type `SyntaxTree` en TypeScript :

```
export type SyntaxTree =
  | { type: "char"; value: string }      // Caractère littéral
  | { type: "dot" }                       // Caractère universel '.'
  | { type: "concat"; left: SyntaxTree; right: SyntaxTree }  // Concaténation
  | { type: "alt"; left: SyntaxTree; right: SyntaxTree }     // Alternation
  | { type: "star"; child: SyntaxTree };                     // Étoile de Kleene
```

Exemple : L'expression régulière `(a|b)*c` est représentée par l'arbre :

```
        concat
       /      \
     star      char('c')
      |
     alt
    /   \
char('a') char('b')
```

2.1.4 Algorithme de Parsing

Le parsing de l'expression régulière en arbre syntaxique est réalisé par un parseur récursif descendant. Cette technique consiste à définir une fonction de parsing pour chaque niveau de la grammaire, en respectant la hiérarchie de précédence.

Structure du parseur :

1. `parseAlternation()` : Point d'entrée, gère l'opérateur `|` (priorité minimale)
   — Parse le membre gauche avec `parseConcatenation()`
   — Tant qu'on rencontre `|`, parse le membre droit et construit un nœud `alt`
2. `parseConcatenation()` : Gère la concaténation implicite
   — Accumule une séquence de facteurs avec `parseFactor()`
   — Réduit la séquence en un arbre binaire gauche avec des nœuds `concat`
3. `parseFactor()` : Gère l'opérateur `*`
   — Parse la base avec `parseBase()`
   — Tant qu'on rencontre `*`, enveloppe dans un nœud `star`
4. `parseBase()` : Gère les éléments atomiques
   — Parenthèses : appel récursif à `parseAlternation()`
   — Point : retourne un nœud `dot`
   — Échappement : consomme le backslash et retourne le caractère suivant comme littéral
   — Caractère simple : retourne un nœud `char`

Complexité : Le parsing s'effectue en $O(n)$ où n est la longueur de l'expression régulière, avec un seul passage sur l'entrée.