Projet 1 DAAR 2025

Breton Noé

n°21516014

Boudrouss Réda

n°28712638

Sorbonne Université
France

Table des matières

1. Introduction

1.1. 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

1.2. Démarche

En étudiant l'implémentation de grep, nous avons remarqué que GNU grep utilise une approche similaire mais avec plusieurs optimisations supplémentaires. Notamment :

- Un préfiltrage des lignes candidates avant le matching regex complet (quand c'est pertinent)
- Une lecture par chunks pour gérer efficacement les fichiers volumineux
- Une sélection automatique des algorithmes les plus adaptés en fonction de la complexité du pattern et de la taille du texte

Nous avons donc décidé d'implémenter des optimisations supplémentaires dans notre projet. Voici dans un premier temps les algorithmes de recherche de motifs littéraux implémentés :

- 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

Puis nous avons implémenté les automates finis :

- NFA : automate fini non-déterministe avec ϵ-transitions
  - Construit à partir de l'arbre syntaxique par la méthode de Thompson
- DFA : automate fini déterministe obtenu par la méthode des sous-ensembles
  - Construit à partir du NFA par la méthode des sous-ensembles
- Min-DFA : automate fini déterministe minimisé pour réduire la mémoire
  - Construit à partir du DFA par l'algorithme de partitionnement
- NFA+DFA-cache : simulation de l'NFA avec construction de DFA à la volée

Enfin, nous avons implémenté un préfiltrage des lignes candidates avant le matching regex complet, en utilisant KMP, Boyer-Moore ou Aho-Corasick en extractant les littéraux du pattern.

Dans ce rapport, nous discuterons des algorithmes implémentés, de leurs performances respectives, et des situations dans lesquelles ils sont les plus pertinents dans le cadre d'un outil de recherche de motifs tel que `egrep`.

1.3. 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.

Pour savoir comment lancer le projet, voir le fichier `README.md` à la racine du projet.

1.4. 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). L'analyse de performance est exposé dans la section 4, puis nous concluons par une discussion des résultats et des perspectives d'amélioration (Section 5).

2. Fondements Théoriques

2.1 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 :

```ts
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')
```

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. Chaque fonction lit l'entrée jusqu'à rencontrer un opérateur de niveau inférieur, puis appelle la fonction de parsing correspondant au niveau inférieur.

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

2.2 Construction du NFA (Automate Fini Non-Déterministe)

2.2.1 Définition

Nous représentons un NFA par un objet TypeScript contenant :

```ts
type NFA = {
  states: state_ID[]; // ensemble fini d'états
  // Dictionnaire des transitions, la clé est l'état source, la valeur est un dictionnaire avec la clé
  // le symbole de transition et la valeur l'ensemble des états cibles
  transitions: { [state: state_ID]: { [symbol: string]: state_ID[] } };
  start: state_ID; // état initial
  accepts: state_ID[]; // ensemble d'états acceptants
};
```

Nous utilisons deux symboles spéciaux, `EPSILON` et `DOT`, pour représenter les ϵ-transitions et le caractère universel, respectivement.

Le caractère non-déterministe signifie que depuis un état donné, plusieurs transitions peuvent être possibles pour un même symbole, et que des ϵ-transitions (transitions sans consommer de caractère) sont autorisées.

2.2.2 Construction de Thompson (Algorithme d'Aho-Ullman)

Pour transformer un arbre syntaxique en un NFA, nous utilisons l'algorithme de construction de Thompson. La construction de Thompson est un algorithme récursif qui construit un NFA à partir d'un arbre syntaxique en appliquant des règles de composition pour chaque opérateur. Chaque sous-expression est transformée en un fragment de NFA avec un état initial et un état final unique.

La complexité est $O(n)$ où n est la taille de l'arbre syntaxique. Chaque nœud est visité exactement une fois, et chaque opération (création d'états, ajout de transitions) est en temps constant.

Son implémentation est dans le fichier `lib/src/NFA.ts` dans la fonction `nfaFromSyntaxTree`.

2.2.3 Fermeture ϵ (Epsilon Closure)

La fermeture ϵ d'un ensemble d'états S est l'ensemble de tous les états accessibles depuis S en suivant uniquement des ϵ-transitions. Cette opération est fondamentale pour le matching avec NFA et la conversion NFA->DFA.

La complexité est $O(|Q| + |\delta|)$ où Q est l'ensemble des états et δ l'ensemble des transitions. Dans le pire cas, on visite tous les états et toutes les ϵ-transitions.

2.2.4 Matching avec NFA

Pour vérifier si une chaîne w est acceptée par un NFA, on simule l'exécution de l'automate en maintenant l'ensemble des états actifs à chaque étape.

Initiallement les états actifs sont la fermeture ϵ de l'état initial. Ensuite, pour chaque caractère de l'entrée, on calcule l'ensemble des états atteignables en suivant les transitions marquées par ce caractère, puis on applique la fermeture ϵ. Si à la fin de l'entrée, l'ensemble des états actifs contient un état acceptant, alors la chaîne est acceptée.

Une implémentation est dans le fichier `lib/src/NFA.ts` dans la fonction `matchNfa`. Mais une implémentation plus complexe répondant à nos contraintes (notamment celle de trouver toutes les correspondances, et leur positionnement dans la chaîne) est dans le fichier `lib/src/Matcher.ts` dans la fonction `findAllMatchesNfa` et `findLongestMatchNfa`.

La complexité est $O(|w| \times |Q|^2)$ où w est la longueur de l'entrée et Q l'ensemble des états. Pour chaque caractère, on peut avoir jusqu'à $|Q|$ états actifs, et la fermeture ϵ peut visiter tous les états.

2.3 Conversion NFA vers DFA

2.3.1 Définition du DFA

Nous représentons un DFA par un objet TypeScript contenant :

```
type DFA = {
  states: state_ID[];
  // On remarquera que contrairement au NFA, on a une seule cible par état et symbole
  transitions: { [state: state_ID]: { [symbol: string]: state_ID } };
  start: state_ID;
  accepts: state_ID[];
};
```

2.3.2 Principe de la Méthode des Sous-Ensembles

La construction par sous-ensembles (subset construction) transforme un NFA en un DFA équivalent. Le principe est que chaque état du DFA représente un ensemble d'états du NFA.

Pour chaque ensemble d'états NFA, on calcule les transitions pour chaque symbole de l'alphabet. Si un nouvel ensemble d'états est découvert, on crée un nouvel état DFA.

On notera que pour chaque NFA, il est toujours possible de construire un DFA équivalent, mais il peut y avoir une explosion combinatoire du nombre d'états (jusqu'à 2^n).

La compléxité temporelle et spatiale est au pire des cas $O(2^n)$ avec n le nombre d'états de l'NFA. En pratique, le nombre d'états générés est souvent beaucoup plus faible (souvent $O(n)$ ou $O(n^2)$).

On notera que la méthodes des sous-ensemble ressemble en partie à la simulation de l'NFA, mais au lieu de maintenir un ensemble d'états actifs, on explore de manière systématique tous les ensembles possibles. Il est donc peut-être

possible de fusionner les deux algorithmes pour obtenir un algorithme plus efficace ? Nous en parlerons dans la prochaine section.

l'implémentation est dans le fichier `lib/src/DFA.ts` dans la fonction `dfaFromNfa`.

2.3.3 Gestion du Caractère Universel (DOT)

Notre implémentation traite spécialement le symbole `DOT` (caractère universel `.`), afin de garantir qu'il correspond à n'importe quel caractère, y compris le caractère spécifique recherché.

Lors du calcul des transitions, si on cherche un symbole `s`, on considère aussi les transitions `DOT` du NFA car `DOT` peut matcher n'importe quel caractère, y compris le symbole spécifique.

2.3.4 Matching avec DFA

Le matching avec un DFA est beaucoup plus simple et rapide qu'avec un NFA, car il n'y a pas de non-déterminisme. Pour chaque caractère de l'entrée, on suit la transition correspondante. Si à la fin de l'entrée, on est dans un état acceptant, alors la chaîne est acceptée.

La complexité est $O(|w|)$ où w est la longueur de l'entrée. Chaque caractère nécessite une transition, et la recherche de la transition est en temps constant.

2.4 Minimisation du DFA

2.4.1 Motivation

Le DFA obtenu par la méthode des sous-ensembles peut contenir des états équivalents c'est à dire des états qui ont le même comportement pour toutes les entrées possibles. La minimisation consiste à fusionner ces états pour obtenir un DFA avec le nombre minimal d'états.

Pour notre implémentation, nous avons choisi de l'algorithme de partitionnement car il est simple à implémenter et suffisamment efficace pour nos besoins.

2.4.2 Principe de l'Algorithme de Partitionnement

L'algorithme de minimisation repose sur le raffinement itératif de partitions, en effet à chaque itération, on subdivise les partitions en sous-partitions si nécessaire. L'algorithme s'arrête lorsque toutes les partitions sont stables, c'est à dire que toutes les états de la partition ont la même signature (comportement identique pour toutes les entrées).

La complexité temporelle est en $O(n^2 \times |\Sigma|)$ où n est le nombre d'états et Σ la taille de l'alphabet. En pratique, la complexité est souvent plus faible car l'algorithme s'arrête généralement avant d'avoir parcouru toutes les itérations.

Notez qu'il existe un algorithme plus efficace, celui de Hopcroft, mais nous n'avons pas eu le temps de l'implémenter.

2.4 Simulation du NFA avec Construction de DFA à la Volée (NFA+DFA-cache)

2.4.1 Principe

Comme dit précédement, il y a beaucoup de similarité entre la construction de sous-ensembles et la simulation d'un NFA. En effet, à chaque étape de la simulation, on calcule l'ensemble des états atteignables à partir de l'ensemble courant par une transition marquée par le symbole courant. Cela ressemble beaucoup à la construction d'un état DFA à partir d'un ensemble d'états NFA.

L'idée de l'approche NFA+DFA-cache est de mémoriser les ensembles d'états NFA visités et leurs transitions pour éviter de recalculer les fermetures epsilon à chaque fois. Ainsi, à chaque étape de la simulation, on regarde si l'ensemble d'états courant a déjà été visité. Si c'est le cas, on réutilise l'état DFA correspondant. Sinon, on crée un nouvel état DFA, on calcule ses transitions (en utilisant la fermeture epsilon), et on les mémorise pour les futures étapes.

2.4.2 Implémentation

L'implémentation se trouve dans le fichier `lib/src/NFAWithDFACache.ts`. La classe `LazyDFACache` gère le cache et la création des états DFA. Les fonctions `matchNfaWithDfaCache` et `findAllMatchesNfaWithDfaCache` utilisent ce cache pour simuler l'NFA et trouver les correspondances.

2.3 Algorithmes de Recherche Littérale

Lorsque le pattern regex est réduit à une simple chaîne de caractères (sans opérateurs `*`, `|`, `.`), il est inefficace de construire un automate complet. Nous utilisons alors des algorithmes de recherche de sous-chaîne optimisés.

2.3.1 Knuth-Morris-Pratt (KMP)

L'algorithme KMP permet de rechercher un motif dans un texte en temps linéaire garanti $O(n + m)$ où n est la longueur du texte et m la longueur du motif.

KMP évite de revenir en arrière dans le texte en utilisant une table de préfixes (LPS - Longest Prefix Suffix) qui indique, pour chaque position du motif, la longueur du plus long préfixe qui est aussi un suffixe.

La complexité temporelle se divise en 2 composantes, une pour la phase de prétraitement du motif en $O(m)$ et une pour la phase de recherche dans le texte en $O(n)$ ce qui nous donne une complexité globale de $O(n + m)$.

2.3.2 Boyer-Moore

L'algorithme de Boyer-Moore est souvent plus rapide que KMP en pratique, notamment pour les motifs longs, car il peut sauter plusieurs caractères à la fois. L'algorithme parcourt le texte de gauche à droite mais compare le motif de droite à gauche.

L'algorithme utilise deux tables précalculées :

- Une table des mauvais caractères qui indique, pour chaque caractère du motif, la dernière position à laquelle il apparaît. Cela permet de décaler le motif de manière à aligner le mauvais caractère avec sa dernière occurrence dans le motif.
- Une table des bons suffixes qui indique, pour chaque suffixe du motif, le décalage à effectuer lorsque ce suffixe correspond. Cela permet de décaler le motif de manière à aligner le suffixe correspondant avec son occurrence la plus à droite dans le motif.

L'implémentation se trouve dans le fichier `lib/src/BoyerMoore.ts`.

La complexité spatiale est en $O(m)$ pour les deux tables précalculées avec m la longueur du motif.

La complexité temporelle est plus difficile à estimer car elle dépend de la distribution des caractères dans le texte. En moyenne, on obtient $O(n/m)$ mais dans le pire cas, on peut avoir $O(n \times m)$ avec n le nombre de caractères dans le texte et m la longueur du motif.

2.4 Aho-Corasick : Recherche Multi-Motifs

L'algorithme d'Aho-Corasick permet de rechercher plusieurs motifs simultanément en un seul passage sur le texte. Il est particulièrement utile pour les patterns regex de type alternation de littéraux (ex : `from|what|who`).

2.4.1 Principe

Aho-Corasick combine deux structures de données :

1. Un trie : arbre préfixe contenant tous les motifs à rechercher
2. Des liens de failure : permettent de passer efficacement d'un motif à un autre lors d'un mismatch

Au lieu de recommencer la recherche depuis le début après un mismatch, les liens de failure permettent de sauter vers le plus long suffixe du chemin courant qui est aussi un préfixe d'un motif.

2.4.2 Structure de Données

Chaque nœud du trie contient :

```
interface TrieNode {
  children: Map<char, TrieNode>;  // Transitions vers les enfants
  failure: TrieNode | null;        // Lien de failure
  output: number[];                // Indices des motifs qui se terminent ici
}
```

L'implémentation se trouve dans le fichier `lib/src/AhoCorasick.ts`.

2.4.3 Construction des Liens de Failure

La construction des liens de failure se fait en BFS (parcours en largeur) après avoir construit le trie.

L'algorithme initialise les enfants directs de la racine avec un lien de failure vers la racine. Ensuite, pour chaque nœud en BFS, on cherche le lien de failure de ses enfants en remontant les liens de failure du nœud courant jusqu'à trouver un ancêtre qui a une transition par le même caractère.

Pour les motifs `["he", "she"]`, le nœud correspondant à "she" aura un lien de failure vers le nœud "he", car "he" est le plus long suffixe de "she" qui est aussi un préfixe d'un motif. Cela permet de détecter "he" même si on était en train de chercher "she".

Un aspect important est l'héritage des outputs : lorsqu'un nœud a un lien de failure vers un nœud final, il hérite de ses outputs. Cela permet de détecter tous les motifs qui se terminent à une position donnée, y compris ceux qui sont des suffixes du motif principal.

2.4.4 Matching avec Aho-Corasick

Lors de la recherche dans le texte, les liens de failure permettent de ne jamais revenir en arrière dans le texte. Pour chaque caractère, si aucune transition n'est possible depuis le nœud courant, on suit les liens de failure jusqu'à trouver un nœud qui a une transition pour ce caractère, ou jusqu'à revenir à la racine.

À chaque position, on vérifie si le nœud courant contient des outputs, ce qui indique qu'un ou plusieurs motifs se terminent à cette position.

2.4.5 Complexité

La complexité de la construction du trie est $O(\sum_{i=1}^{k} |p_i|)$ où k est le nombre de motifs et $|p_i|$ la longueur du motif i.

La complexité de la construction des liens de failure est aussi $O(\sum_{i=1}^{k} |p_i|)$. Chaque nœud est visité une fois en BFS, et pour chaque nœud, on remonte au plus $|p_i|$ liens de failure.

La complexité de la recherche est $O(n + z)$ où n est la longueur du texte et z le nombre total de matches trouvés. Le parcours du texte est linéaire, et les liens de failure peuvent être suivis plusieurs fois, mais le nombre total de suivis est borné par n (analyse amortie).

3. Stratégies d'Optimisation

Au-delà de l'implémentation classique des automates, nous avons développé plusieurs optimisations inspirées des moteurs de recherche modernes comme GNU `grep`. Ces optimisations permettent d'améliorer significativement les performances pour de nombreux cas d'usage.

3.1 Lecture par Chunks

Pour les fichiers volumineux (plusieurs GB), charger tout le fichier en mémoire est inefficace. Nous utilisons une lecture par chunks (blocs de 64 KB par défaut).

3.2 Extraction de Littéraux et Préfiltrage

L'idée centrale du préfiltrage est d'extraire les segments littéraux (chaînes fixes) d'un pattern regex et de les utiliser (avec les algorithmes de recherche de sous-chaîne) pour éliminer rapidement les lignes qui ne peuvent pas matcher, avant d'appliquer le matching regex complet.

Exemple : Pour le pattern `.*hello.*world.*`, on extrait les littéraux `["hello", "world"]`. Une ligne ne peut matcher que si elle contient à la fois "hello" et "world". On peut donc utiliser Boyer-Moore ou Aho-Corasick pour filtrer rapidement les lignes candidates.

Le préfiltrage est d'abords appliqué sur les chunks de texte, avant de découper les lignes. Cela évite de découper inutilement les lignes qui ne contiennent pas le motif recherché.

3.3 Sélection Automatique d'Algorithme

Nous analysons automatiquement le pattern et la taille du texte pour choisir l'algorithme de matching et de préfiltrage optimal. Nous prennons en compte plusieurs métriques notamment les types de sous-expressions, la longueur des littéraux, la complexité globale du pattern, et la taille du texte.

Pour l'algorithme de préfiltrage : - Si le pattern ne contient pas de littéraux, on désactive le préfiltrage. - Si le texte à analyser est petit (< 10KB), on désactive le préfiltrage, car l'overhead que cela inclue n'est pas amorti (notamment la construction d'Aho-Corasick ou le fait d'analyser une ligne dans un premier temps avec le préfiltrage puis par la suite avec le matcher regex). - Si le pattern ne contient pas de sous-expressions (`|`, `*`, `.`), on désactive le préfiltrage. (Car l'algorithme de matching sera probablement ceux utilisés pour les patterns littéraux, qui sont déjà très rapides). - Si le pattern contient un seul littéral, on utilise Boyer-Moore. - Si le pattern contient plusieurs littéraux, on utilise Aho-Corasick.

Pour l'algorithme de matching : - Si le pattern est une alternation pure de littéraux (ex : "from|what|who"), on utilise Aho-Corasick. - Si le pattern est un simple littéral (ex : "hello"), on utilise KMP si le littéral est court (moins de 10 caractères) et Boyer-Moore sinon. (Boyer Moore est plus rapide en pratique pour les motifs longs, mais KMP offre une garantie de complexité linéaire) - Si le texte à analyser est très petit (< 500 bytes), on utilise NFA. (Car le coût de construction du DFA n'est pas amorti) - Si le texte à analyser est petit (500 bytes - 10KB), on utilise un Nfa avec cache DFA (on-the-fly). Cela permet d'éviter la construction coûteuse du DFA tout en bénéficiant de la rapidité de l'exécution DFA. - Si le pattern n'est pas trop complexe, on utilise un DFA minimisé.

4. Résultats et Analyse de Performance

4.1. Présentation des Résultats

Les résultats des tests sont issus de la commande `./egrep --test-all --test-folder data --csv report/projet1/perfor`
Le fichier `data` contient 9 fichiers de test, le même livre du projet Gutenberg, mais de taille croissante (de 1KB à 2MB). Nous testons les différents algorithmes sur plusieurs scénarios que vous pouvez retrouver dans le fichier `cli/src/test-all.ts`.

4.2. Boyer-Moore vs KMP

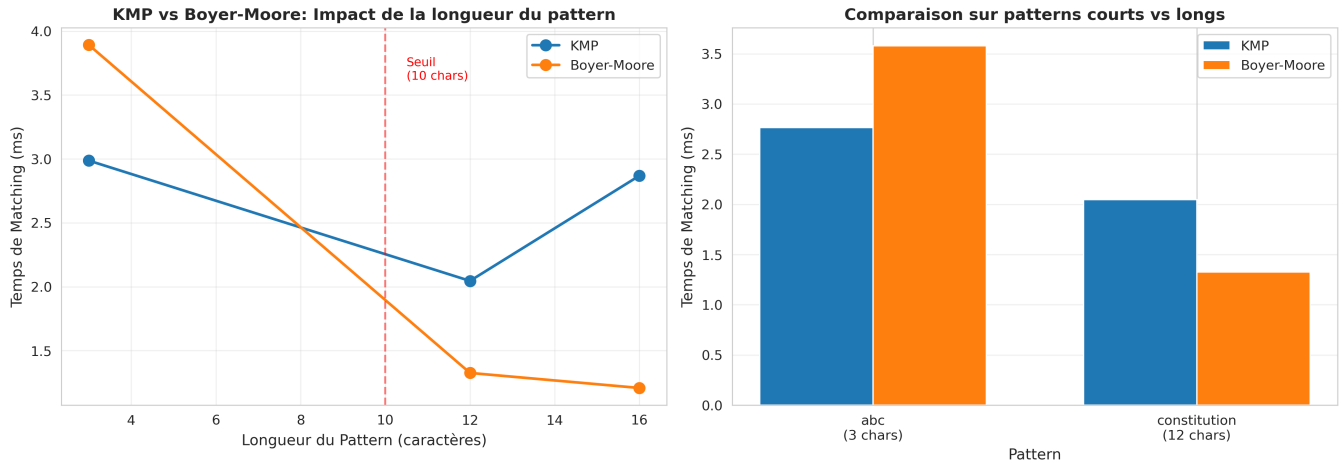

FIGURE 1 – KMP vs Boyer-Moore

Les graphiques de la figure 1 compare le temps de matching de KMP et Boyer-Moore pour des patterns littéraux de longueur variable. On observe bien que KMP est plus rapide que Boyer-Moore pour des patterns courts, mais que la situation inverse se produit pour des patterns longs. Cela justifie notre choix de KMP pour les patterns courts et Boyer-Moore pour les patterns longs.

4.3. Structure de l'automate : intérêt de la minimisation

Les graphiques de la figure 2 compare la taille de la structure des automates NFA, DFA et min-DFA en moyenne sur les différents scénarios. On observe que le NFA est toujours plus grand que le DFA, qui lui est lui-même plus grand que le min-DFA. Cela confirme l'intérêt de la minimisation pour réduire la taille de la structure.

Cependant, la taille du DFA est souvent très proche de la taille du DFA minimisé, au vu de la complexité de l'algorithme de minimisation, il est possible de discuter de son interêt. Dans notre choix automatique, nous avons décidé de toujours opté pour un min-DFA car nous utilisons une structure DFA que si le text est assez grand pour ammortir le coût de construction.

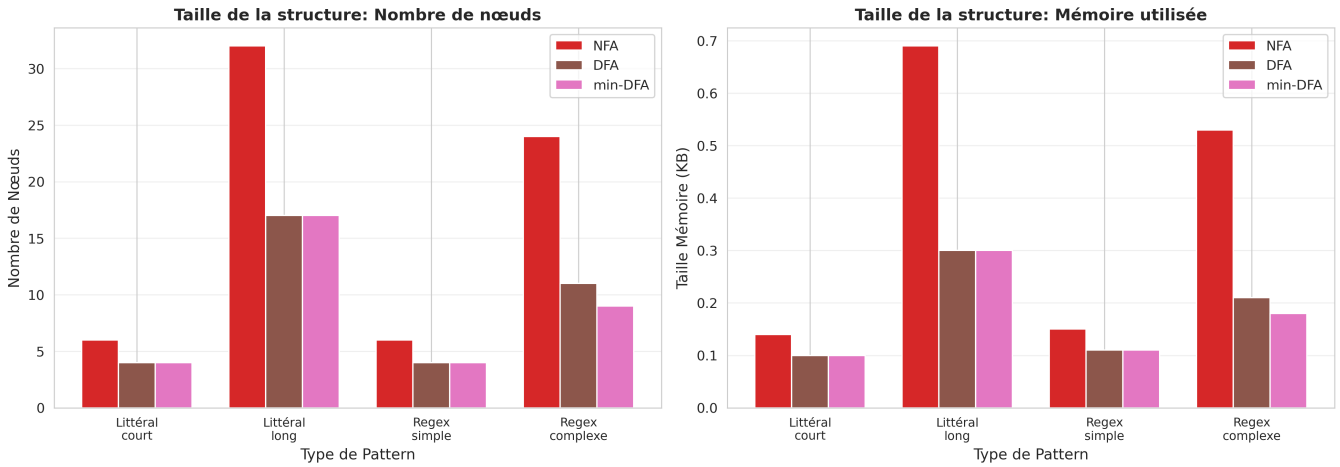

FIGURE 2 – Taille de la structure

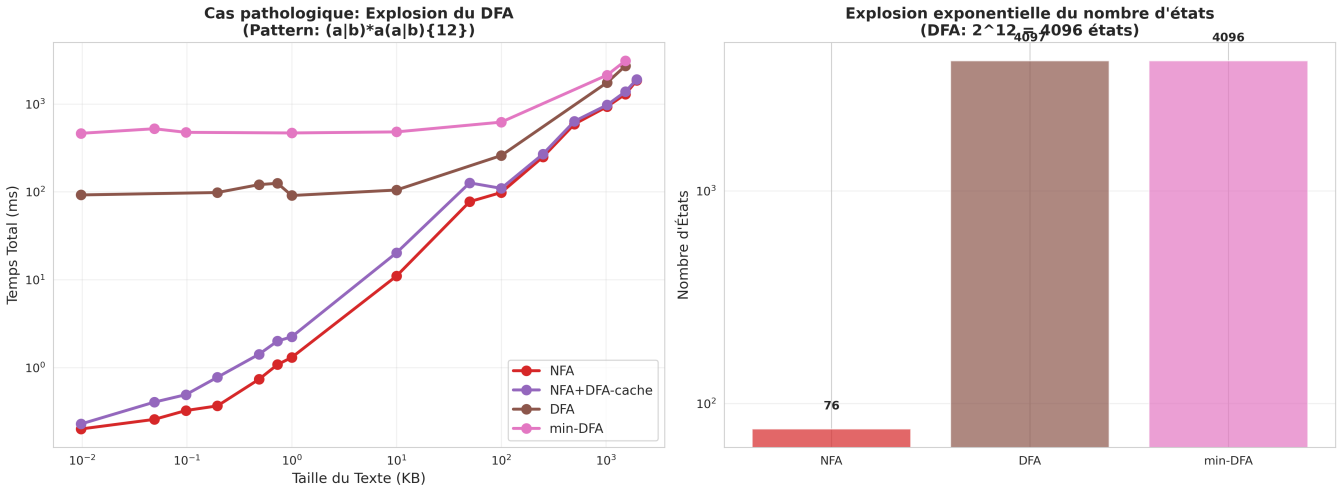

FIGURE 3 – Cas pathologique du DFA

Il est aussi nécessaire de mentionner que pour certain patternes, il peut arriver que le DFA soit une très mauvaise solution. Par exemple, pour des patterns contenant beaucoup d'alternations après une étoile, le nombre d'états du DFA peut exploser exponentiellement. Cela est le cas pour le scénario "Edge Case - DFA Exponential Blowup" affiché sur la figure 3 qui contient 12 groupes alternatifs, nous nous retrouvons avec un DFA de 4097 états contre 76 pour le NFA.

4.4. Impact du préfiltrage

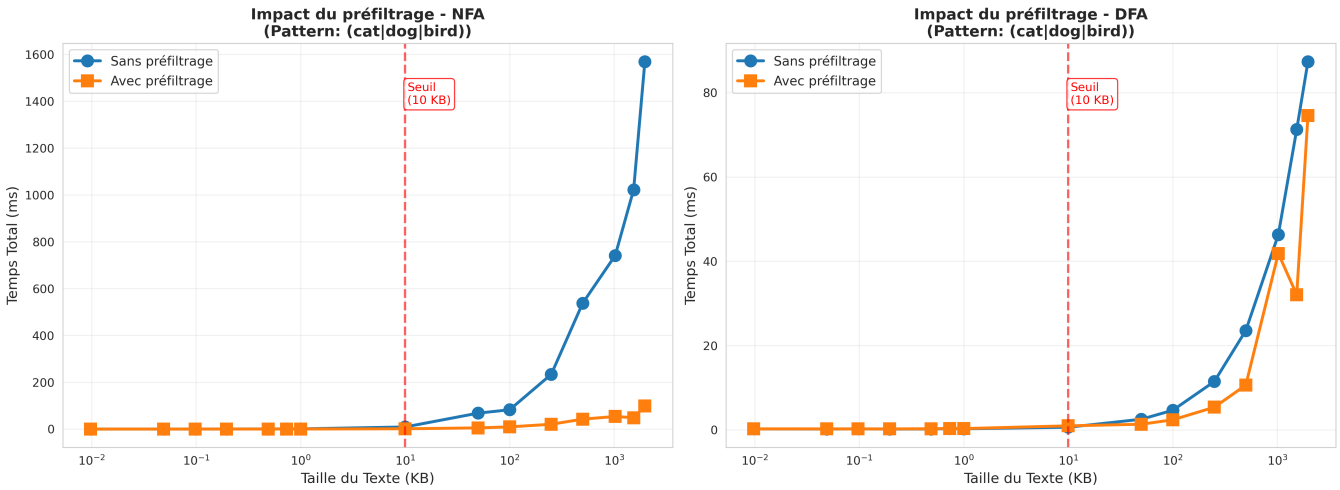

FIGURE 4 – Impact du préfiltrage

La figure 4 compare le temps de matching avec et sans préfiltrage pour des textes de taille variable pour le NFA et le DFA. Pour les tailles de textes inférieurs à 10kb, nous observons que le préfiltrage est inutile voir même contre-productif. Car l'overhead que cela cause (construction d'Aho-Corasick quand il y a plusieurs littéraux, ou le fait d'analyser une ligne dans un premier temps avec le préfiltrage puis par la suite avec le matcher regex) n'est pas amorti par le gain de temps lors du matching.

On remarquera aussi que le préfiltrage est moins efficace pour le DFA que pour le NFA. Cela est dû au fait que le DFA est déjà un algorithme très rapide.

4.5. Choix d'algorithme selon la taille du texte

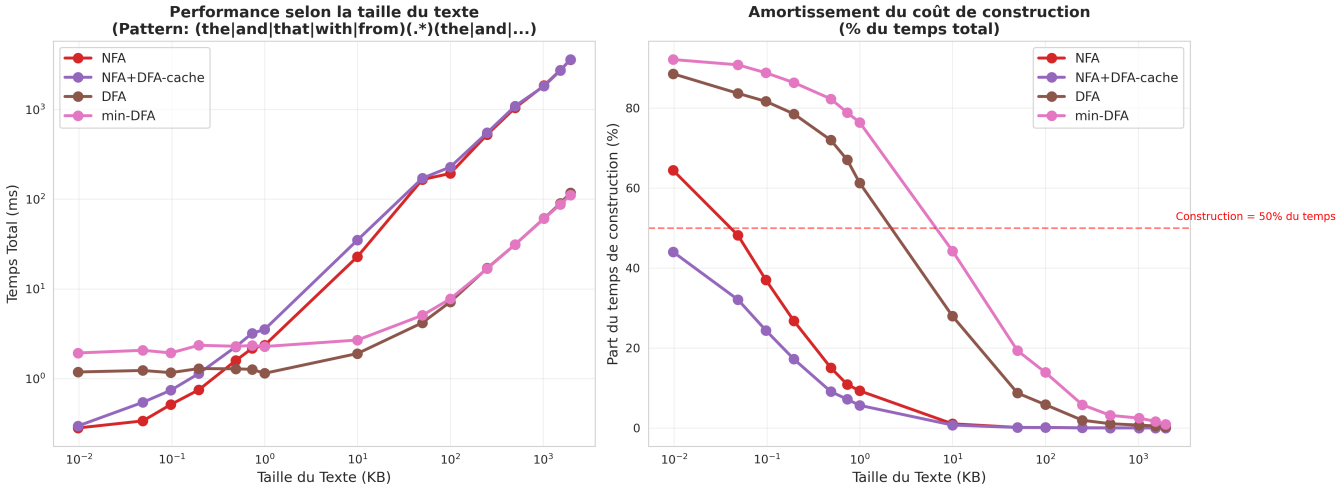

FIGURE 5 – Choix d'algorithme selon la taille du texte

La figure 5 compare le temps de matching de différents algorithmes selon la taille du texte.

Nous remarquons que pour des textes de taille inférieure à 10kb, le NFA est le meilleur algorithme. Cela est cohérent avec notre analyse car le coût de construction du DFA n'est pas amorti pour de petits textes.

La droite du NFA est droite sur un plan log-log, la pente étant normal (proche de 1) alors nous pouvons dire que notre implémentation est linéaire comme prévu par la théorie.

Pour la droite du DFA, au delà de 10kb, nous observons que le temps de matching est linéaire comme prévu par la théorie. Cependant, pour des textes de taille inférieure à 10kb, le temps d'exécution semble constant. Celà est du au fait que le temps de construction du DFA n'est pas amorti pour de petits textes.

5. Conclusion

5.1. Résultats Obtenus

L'implémentation d'un clone fonctionnel de `egrep` nous a permis d'explorer différentes approches algorithmiques pour la recherche de motifs. Au-delà de la construction classique d'automates finis (NFA avec ϵ-transitions par construction de Thompson, DFA par méthode des sous-ensembles, minimisation par partitionnement, NFA avec cache DFA à la volée), nous avons intégré des algorithmes de recherche littérale (KMP pour les motifs courts, Boyer-Moore pour les motifs longs, Aho-Corasick pour les alternations de littéraux) et développé des stratégies d'optimisation inspirées de GNU grep, notamment le préfiltrage par extraction de littéraux et la sélection automatique d'algorithme.

Les résultats expérimentaux confirment les prédictions théoriques tout en révélant des comportements intéressants. Le NFA présente une complexité linéaire en pratique pour des textes de taille modérée (< 10KB), tandis que le DFA devient plus efficace pour des textes volumineux une fois son coût de construction amorti. La minimisation du DFA réduit effectivement la taille de la structure, bien que le gain soit souvent marginal par rapport au DFA non minimisé. Le préfiltrage améliore significativement les performances pour les textes volumineux (> 10KB), mais introduit un overhead non négligeable pour les petits textes. Ces observations justifient notre approche de sélection automatique d'algorithme basée sur l'analyse du pattern et de la taille du texte.

5.2. Limitations et Points de Discussion

Plusieurs limitations doivent être soulignées. Pour certains patterns pathologiques contenant de nombreuses alternations après une étoile (ex : `(a|b|c|...)*`), le nombre d'états du DFA peut croître exponentiellement (2^n dans le pire cas). Nos tests ont montré un cas avec 4097 états DFA contre 76 états NFA. Dans ces situations, la construction récursive du DFA peut dépasser la limite de la pile d'appels. Une refonte en version itérative serait nécessaire pour garantir la robustesse de l'implémentation.

Par ailleurs, bien que nous ayons mesuré la taille des structures (nombre d'états et de transitions), nous n'avons pas pu mesurer précisément l'empreinte mémoire réelle en production. JavaScript/TypeScript ne fournit pas d'API fiable pour mesurer la consommation mémoire d'objets spécifiques, et les outils de profiling disponibles donnent des résultats approximatifs influencés par le garbage collector et les optimisations du moteur V8. Cette limitation rend difficile l'évaluation précise du coût mémoire des différentes approches, notamment pour comparer le NFA avec cache DFA au DFA complet. Une implémentation en langage bas niveau (C, Rust) permettrait une analyse plus rigoureuse de ces aspects.

D'autres points méritent discussion. Le seuil de 10KB pour activer le préfiltrage a été déterminé empiriquement sur notre corpus de test et pourrait varier selon les caractéristiques du texte et du pattern. L'algorithme de partitionnement utilisé pour la minimisation a une complexité $O(n^2 \times |\Sigma|)$, alors que l'algorithme de Hopcroft, plus efficace en $O(n \log n)$, pourrait améliorer les performances sur de grands automates.

5.3. Perspectives

L'extension du support à d'autres opérateurs ERE constitue une évolution naturelle du projet. Les classes de caractères `[a-z]`, les quantificateurs `+` et `?`, ainsi que les ancres `^` et `$` permettraient de couvrir davantage de cas d'usage réels. Cependant, l'ajout de certains opérateurs comme les backreferences `\1` nécessiterait une approche différente. Les backreferences introduisent des dépendances contextuelles qui ne peuvent pas être exprimées par des automates finis réguliers, et requièrent généralement un algorithme de backtracking. Cette approche, bien que plus expressive, présente des risques de complexité exponentielle dans le pire cas, ce qui explique pourquoi de nombreux moteurs modernes limitent leur usage ou les traitent séparément.

Une piste intéressante serait l'exploration des automates finis déterministes tagués (TDFA - Tagged DFA). Cette approche, utilisée notamment par RE2 de Google, permet de capturer les groupes de capture tout en conservant les garanties de complexité linéaire des DFA. Les TDFA associent des tags aux transitions pour marquer les positions de début et de fin des groupes, évitant ainsi le backtracking tout en supportant une partie des fonctionnalités avancées des regex modernes. Cette technique pourrait être particulièrement pertinente pour notre implémentation, car elle s'intègre naturellement dans notre architecture basée sur les automates.

Du côté des optimisations bas niveau, l'utilisation d'instructions SIMD pour la recherche de littéraux pourrait accélérer significativement le préfiltrage, notamment pour les patterns contenant plusieurs littéraux courts. La détection automatique de préfixes fixes dans les patterns permettrait également d'optimiser davantage les cas simples. Pour les fichiers volumineux, la parallélisation du traitement de chunks constitue une piste prometteuse, bien que la gestion des correspondances à cheval sur les frontières de chunks nécessite une attention particulière.

Ce projet a permis de constater que l'efficacité d'un moteur de recherche de motifs repose autant sur le choix algorithmique que sur les optimisations pratiques. La théorie des automates fournit un cadre solide, mais les performances réelles dépendent fortement de l'adaptation des algorithmes aux caractéristiques des données traitées. La sélection automatique d'algorithme et le préfiltrage se révèlent essentiels pour obtenir des performances compétitives sur des cas d'usage variés.