Ministère de l'Enseignement Supérieur et de la Recherche Scientifique

Université de Ghardaïa

Faculté des Sciences et de la Technologie

Département des Mathématiques et d'Informatique

Rapport de Mini-Projet

Recherche de Motifs

Réalisé par : AMINI Safa

Niveau : 2e Année Licence Informatique

Groupe : 02

Année Universitaire : 2024–2025

Table des matières

1 Introduction

En S4 de la 2e année de Licence Informatique à l'Université de Ghardaïa, le cours de Théorie des Langages (THL) est enseigné. Ce mini-projet s'inscrit dans ce cadre et porte sur la problématique de la recherche de motifs dans un texte. En informatique, cette notion consiste à identifier toutes les occurrences d'une chaîne de caractères, appelée motif, à l'intérieur d'une chaîne plus longue, appelée texte. La recherche de motifs est une tâche importante, avec des applications dans divers domaines, notamment le traitement de texte, la bioinformatique et la recherche d'information.

Le projet a pour objectif principal d'implémenter et de comparer deux approches pour la recherche de motifs : l'algorithme naïf, construit à partir de la technique d'une fenêtre glissante, et l'algorithme basé sur un Automate à États Finis Déterministe (AFD), qui repose sur une structure de données permettant une recherche optimale. Le rapport explique l'approche utilisée pour la réalisation, présente l'implémentation en langage C, les tests et l'analyse des performances des deux algorithmes implémentés.

Le rapport est organisé comme suit : une introduction aux objectifs, une explication des algorithmes, les détails de l'implémentation, des exemples d'exécution, une analyse des performances, les problèmes rencontrés, une conclusion et une annexe comprenant le code source et quelques exemples de cas de test.

2 Objectifs du projet

Ce projet a pour but de répondre aux exigences suivantes :

- **Implémentation des algorithmes** : Développer deux algorithmes de recherche de motifs (naïf et AFD) en langage C.
- **Modularité du code** : Structurer le code de manière claire en séparant les définitions (fichiers .h) des implémentations (fichiers .c).
- **Exemples d'utilisation** : Fournir des scénarios d'exécution via l'entrée standard (console) et la lecture de fichiers externes.
- **Comparaison des performances** : Évaluer et comparer les performances des deux algorithmes sur différents textes et motifs.
- **Rédaction du rapport** : Produire un rapport détaillé illustrant les étapes du projet et les résultats obtenus.

3 Présentation des algorithmes

3.1 Algorithme naïf

L'algorithme naïf utilise une technique de fenêtre glissante de taille m, où m est la longueur du motif. Pour chaque position i dans le texte (de longueur n), il compare les m caractères du motif avec la sous-chaîne du texte commençant à la position i. Si tous les caractères correspondent, une occurrence est enregistrée.

La complexité temporelle de cet algorithme est $O(n \cdot m)$, car pour chaque position (au plus $n - m + 1$), il effectue jusqu'à m comparaisons. Cet algorithme est simple à implémenter et ne nécessite pas de structures de données complexes, mais il devient inefficace pour des textes longs ou des motifs fréquents, en raison de sa complexité quadratique.

3.2 Recherche par AFD

L'algorithme basé sur un AFD construit un automate déterministe pour reconnaître le motif. L'AFD est défini comme suit :

- **États** : L'ensemble des préfixes du motif, $Q = \text{Pref}(P)$.
- **État initial** : Le préfixe vide, $q_0 = \varepsilon$.
- **État final** : Le motif complet, $F = \{P\}$.
- **Transitions** : Définies par la fonction $\delta(u, a) = h(ua)$, où $h(u)$ est le plus long suffixe de u qui est un préfixe du motif.

La construction de l'AFD nécessite $O(m^2 \cdot |\Sigma|)$ opérations, où $|\Sigma|$ est la taille de l'alphabet, car pour chaque état et chaque caractère, on calcule la fonction h. Une fois construit, l'AFD permet une recherche en temps linéaire $O(n)$, car chaque caractère du texte entraîne une transition en temps constant. Cet algorithme

est plus efficace pour des textes longs, bien que la construction initiale puisse être coûteuse pour des motifs longs ou des alphabets larges.

4 Conception et Implémentation

4.1 Structure générale du code

Le projet est organisé de manière modulaire pour assurer clarté et maintenabilité :

- **Fichiers d'en-tête (.h)** : Contiennent les définitions des structures de données (ex. `AFD`), les constantes (`ALPHABET_SIZE`, `MAX_PATTERN_LENGTH`), et les prototypes des fonctions.
- **Fichiers d'implémentation (.c)** : Contiennent les implémentations des fonctions déclarées dans les fichiers .h.
- **Fichier principal (main.c)** : Gère l'interaction avec l'utilisateur et coordonne les appels aux fonctions des autres modules.

Schéma du projet :

```
projet/
 pattern_matching.h
 naive_search.c
 afd_search.c
 main.c
```

4.2 Détails de l'implémentation en C

- **Algorithme naïf** (`naive_search.c`) :

  La fonction `naive_pattern_matching` parcourt le texte avec une fenêtre de taille m, comparant caractère par caractère. Elle stocke les positions des occurrences dans un tableau `matches`.

- **Algorithme AFD** (`afd_search.c`) : Les fonctions clés sont :
  - `create_afd` : Construit la matrice de transition en calculant $h(ua)$ pour chaque état et caractère.
  - `afd_pattern_matching` : Parcourt le texte, applique les transitions, et enregistre les occurrences lorsque l'état final est atteint.
  - `free_afd` : Libère la mémoire allouée.
- **Utilitaires** (`pattern_matching.h`) :

  La fonction `char_to_index` convertit un caractère en indice pour l'alphabet {a, ..., z, espace}.
- **Lecture des entrées** : Le programme `main.c` lit le motif et le texte soit via la console (scanf), soit à partir de fichiers texte utilisant `fopen` et `fgets`.

4.3 Organisation des fichiers

- `pattern_matching.h` : Définitions des structures (`AFD`), constantes, et prototypes.
- `naive_search.c` : Implémentation de l'algorithme naïf.
- `afd_search.c` : Implémentation de l'algorithme AFD.
- `main.c` : Point d'entrée, gestion des entrées/sorties et coordination.

5 Exemples d'exécution

5.1 Entrée depuis la console

Exemple d'exécution avec saisie manuelle :

```
Entrez le texte : ababacabab
Entrez le motif : abac
Résultat (Naïf) : Occurrence à la position 2
Résultat (AFD) : Occurrence à la position 2
```

5.2 Entrée depuis un fichier externe

Fichiers `motif.txt` (`abac`) et `texte.txt` (`ababacabab`). Résultat :

```
Lecture depuis fichiers...
Résultat (Naïf) : Occurrence à la position 2
Résultat (AFD) : Occurrence à la position 2
```

5.3 Comparaison des résultats

Pour le texte `ababacabab` et le motif `abac`, les deux algorithmes retournent la même position (2). Les durées mesurées (approximatives, via `clock()`) montrent que l'algorithme AFD est plus rapide pour des textes longs, bien que la construction initiale prenne du temps.

6 Analyse des performances

6.1 Complexité théorique

— **Naïf** : $O(n \cdot m)$, dû aux comparaisons pour chaque fenêtre.

— **AFD** : Construction en $O(m^2 \cdot |\Sigma|)$, recherche en $O(n)$.

6.2 Observations expérimentales

Tests effectués :

— **Texte court** ($n = 10$, motif `abac`) : Temps similaire pour les deux algorithmes.

— **Texte moyen** ($n = 1000$) : AFD plus rapide après construction.

— **Texte long** ($n = 10000$) : AFD nettement plus performant, malgré le coût initial.

L'algorithme naïf devient prohibitif pour des textes longs, tandis que l'AFD excelle après la construction.

7 Problèmes rencontrés et solutions apportées

— **Gestion des fichiers** : Problèmes de lecture si les fichiers n'existent pas. Solution : Vérification avec `fopen` et messages d'erreur.

— **Construction de l'AFD** : Calcul de $h(ua)$ initialement incorrect pour certains cas. Solution : Débogage en testant tous les suffixes possibles.

— **Mémoire** : Fuites potentielles dans `create_afd`. Solution : Ajout de `free_afd` pour libérer correctement la mémoire.

8 Conclusion et perspectives

Ce projet a permis de maîtriser la recherche de motifs à travers deux approches complémentaires : l'algorithme naïf, simple mais limité, et l'algorithme AFD, plus complexe mais performant. L'implémentation en C a renforcé les compétences en programmation modulaire et en gestion de la mémoire.

Perspectives d'amélioration :

— Développer une interface graphique pour visualiser les occurrences.

— Prendre en charge plusieurs motifs simultanément (ex. via Aho-Corasick).

— Optimiser la construction de l'AFD pour réduire le coût en $O(m^2 \cdot |\Sigma|)$.

9 Annexes

9.1 Code source

Le code source complet est fourni séparément dans les fichiers `pattern_matching.h`, `naive_search.c`, `afd_search.c`, et `main.c`. Extrait de `naive_search.c` :

```
void naive_pattern_matching(char* text, char* pattern, int*
   matches, int* match_count) {
    int n = strlen(text);
    int m = strlen(pattern);
    *match_count = 0;
    if (m == 0 || n < m) return;
    for (int i = 0; i <= n - m; i++) {
        int j;
        for (j = 0; j < m; j++) {
            if (text[i + j] != pattern[j]) break;
        }
        if (j == m) {
            matches[*match_count] = i;
            (*match_count)++;
        }
    }
}
```

9.2 Cas de test supplémentaires

— Texte : `aabaabaa`, Motif : `aa`, Résultat : Positions 0, 3, 6.

— Texte : `abcd efgh`, Motif : ` `, Résultat : Position 4.