Rapport de Projet C : Analyseur de Fréquence de Mots

Binôme : CHING Brian & ENG Johnatan

5 janvier 2026

Table des matières

1 Architecture du Projet

Le projet est structuré de manière modulaire pour séparer la logique algorithmique, la gestion de la mémoire et l'interface utilisateur. Voici le contenu de l'archive .zip :

- **Fichiers sources C (.c) et en-têtes (.h) :**
  - `main.c` : Point d'entrée du programme. Gère le menu interactif, la lecture des arguments et l'appel des algorithmes.
  - `algo1.c / algo1.h` : Implémentation de l'Algorithme 1 (Liste chaînée + Tri Fusion).
  - `algo2.c / algo2.h` : Implémentation de l'Algorithme 2 (Tableau dynamique).
  - `algo3.c / algo3.h` : Implémentation de l'Algorithme 3 (Liste chaînée + Insertion Dichotomique).
  - `fonction_allocation.c / .h` : Module de gestion mémoire (`myMalloc`, `myFree`) pour mesurer la consommation.
  - `fonction_commun.h` : Définitions communes (structure `MotNode`).
- **Scripts Python :**
  - `benchmark.py` : Script d'automatisation pour générer des fichiers de test de tailles variées et lancer les algorithmes (import subprocess)
  - `visualisation.py` : Génère les courbes de comparaison des performances (Temps vs Taille, Mémoire vs Taille).
  - `graphique_mots.py` : Génère un histogramme des mots les plus fréquents pour un fichier donné.
- **Autres fichiers :**
  - `makefile` : Automatise la compilation du projet ('make', 'make clean').
  - `performances.csv` : Fichier généré contenant les métriques de performance.
  - `test.md` : Fichier texte exemple pour les tests unitaires.

2 Organisation du Groupe

2.1 Méthodologie de travail

Nous avons adopté une approche itérative. Dans un premier temps, nous avons défini les structures de données communes et le module de gestion mémoire. Ensuite, nous avons travaillé en parallèle sur l'implémentation des algorithmes avant de les intégrer dans le `main.c`.

La phase de test et d'optimisation a été réalisée ensemble, notamment pour la mise en place du benchmark et des visualisations.

2.2 Répartition des tâches

Tâche	Étudiant 1 (%)	Étudiant 2 (%)
Conception globale	50%	50%
Gestion mémoire	80%	20%
Algorithme 1 (Liste)	90%	10%
Algorithme 2 (Tableau)	10%	90%
Algorithme 3 (Dichotomie)	90%	10%
Scripts Python	20%	80%
Rapport	20%	80%

TABLE 1 – Contribution approximative au projet

Chaque membre maîtrise l'ensemble du code, les phases de fusion et de debugging ayant été réalisées ensemble.

3 Algorithmes et Structures de Données

3.1 Algorithme 1 : Liste Chaînée et Tri Fusion

Cet algorithme utilise une liste chaînée simple pour stocker les mots.

- **Structure :** `MotNode` (mot, compteur, suivant).
- **Insertion :** Ajout en tête après vérification de l'existence.
- **Tri :** Tri Fusion appliqué à la liste chaînée à la fin du traitement.
- **Complexité théorique :**
  - Insertion : $O(N \times M)$ où N est le nombre de mots du texte et M le nombre de mots uniques.
  - Tri : $O(M \log M)$.

Listing 1 – Structure de Liste Chaînée

```
typedef struct MotNode {
    char *mot;
    int compteur;
    struct MotNode *suivant;
} MotNode, *Liste;
```

3.2 Algorithme 2 : Tableau Dynamique et QuickSort

Utilise un tableau contigu en mémoire, redimensionné dynamiquement.

- **Structure :** Tableau de structs `MotTab`, avec gestion de `taille` et `capacite`.
- **Insertion :** Recherche linéaire dans le tableau. Si inexistant, ajout à la fin (avec `realloc` si nécessaire).
- **Tri :** Utilisation de `qsort` (QuickSort) de la bibliothèque standard.
- **Complexité théorique :**
  - Insertion : $O(N \times M)$ (recherche linéaire).
  - Tri : $O(M \log M)$.

3.3 Algorithme 3 : Liste Chaînée et Insertion Dichotomique

Amélioration de l'Algorithme 1 en maintenant la liste triée alphabétiquement dès l'insertion.

- **Insertion :** Recherche dichotomique pour trouver la position d'insertion.
- **Tri final :** Tri par fréquence (Tri Fusion) car la liste est triée alphabétiquement, pas par fréquence.
- **Note :** La recherche dichotomique sur une liste chaînée est théoriquement moins efficace que sur un tableau car l'accès au milieu est en $O(M)$, rendant la complexité totale $O(N \times M)$.

4 Étude Comparative des Performances

Les tests ont été réalisés sur des fichiers générés aléatoirement allant de 1 000 à 100 000 mots (avec des mots prédéfinis dans une liste générés par le benchmark).

4.1 Complexité Temporelle

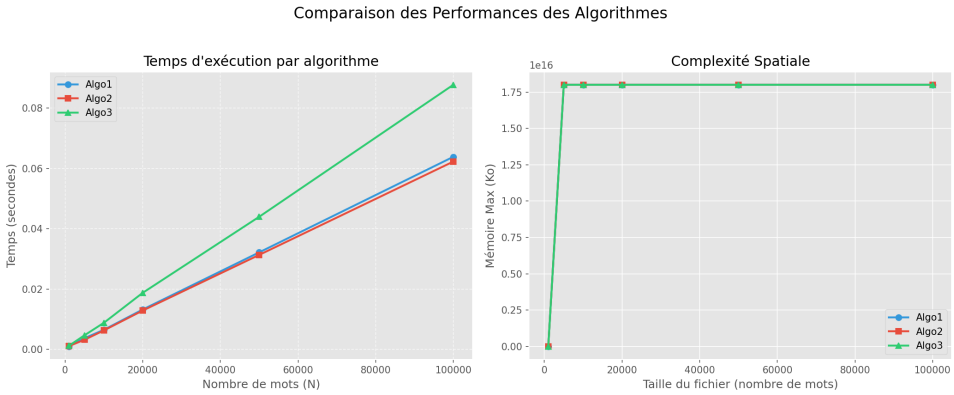

FIGURE 1 – Comparaison des performances (Temps et Mémoire)

Analyse des résultats :

- **Algo 2 (Tableau)** se révèle le plus performant sur les grands volumes de données.
- **Algo 1 (Liste)** est performant pour les petits fichiers. Néanmoins, le coût des allocations dynamiques pour chaque élément de la liste devient trop élevé lorsque la taille des fichiers augmente.
- **Algo 3** est le moins performant. Bien que l'idée de la dichotomie soit bonne, son application sur une liste chaînée est contre-productive.

4.2 Consommation Mémoire

Les trois algorithmes ont une complexité spatiale théorique identique de $O(M)$ où M est le nombre de mots uniques. Cependant, il y a des différences significatives dans la consommation mémoire réelle.

L'Algorithme 2 (Tableau dynamique) est généralement plus économe en mémoire car il n'a pas besoin de stocker un pointeur `suivant` pour chaque mot, contrairement aux listes chaînées (+8 octets par mot sur système de 64 bits). De plus, la mémoire allouée

est côte à côte, ce qui permet de savoir où elle est allouée plus facilement et améliore l'efficacité du cache processeur.

Cependant, le redimensionnement du tableau (parfois doublé quand on agrandit le tableau) peut provoquer des hausses temporaires de consommation mémoire. Les algorithmes basés sur les listes chaînées (Algo 1 et 3) allouent la mémoire de manière plus progressive, mais chaque allocation nécessite un coût supplémentaire pour les données de gestion de la mémoire.

Les mesures montrent que l'Algorithme 2 consomme moins de mémoire maximale sur les grands volumes de données, principalement grâce à la meilleure utilisation de la mémoire et à l'absence de pointeurs supplémentaires.

5 Améliorations Réalisées

En plus des trois algorithmes de base, nous avons implémenté plusieurs améliorations pour enrichir le projet.

5.1 Filtrage par longueur minimale ()

Nous avons ajouté un paramètre optionnel `longueur_min` permettant à l'utilisateur de filtrer les mots selon leur longueur minimale. Cette fonctionnalité permet d'ignorer les mots trop courts (comme les articles, prépositions) qui peuvent polluer l'analyse.

Implémentation : Le paramètre `k` est passé aux fonctions `MotFichier`, `MotFichierTab`, `MotFichier3`. Avant d'ajouter un mot à la structure de données, une vérification `if (len >= longueur_min)` permet de filtrer les mots trop courts.

Utilisation :

Listing 2 – Exemple d'utilisation

```
./programme 1 test.md 10 1 resultat.txt 3
```

Cette commande ne prend en compte que les mots d'au moins 3 lettres.

5.2 Génération de graphique des mots les plus courants ()

Nous avons développé le script `graphique_mots.py` qui génère un histogramme en barres verticales des mots les plus fréquents à partir d'un fichier de résultats.

Utilisation :

Listing 3 – Exemple d'utilisation

```
./programme 1 test.md 10 1 resultat.txt 1
python3 graphique_mots.py resultat.txt
```

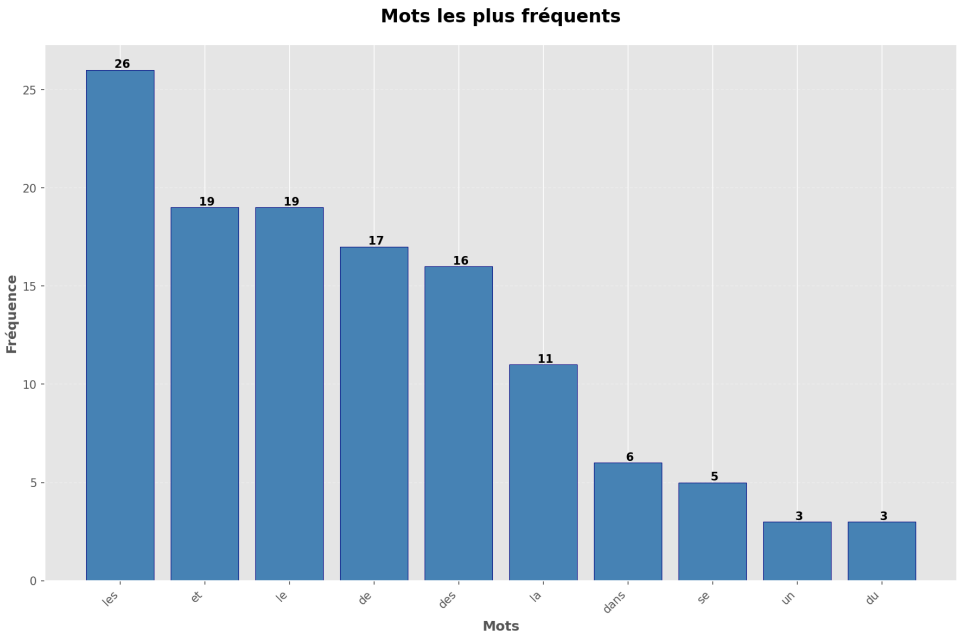

FIGURE 2 – Graphique des mots les plus fréquents généré à partir de `test.md`

5.3 Script de benchmark automatisé

Bien que non mentionné dans l'énoncé, nous avons créé le script `benchmark.py` pour automatiser les tests de performance sur des fichiers de tailles variées.

Fonctionnalités :

— Génération automatique de fichiers de test de différentes tailles (1 000 à 100 000 mots)
— Lancement automatique des trois algorithmes sur chaque fichier
— Collecte et fusion des résultats dans un fichier `performances.csv` unique
— Compilation automatique du programme avant les tests

Ce script facilite grandement la comparaison des performances et permet de générer rapidement les données nécessaires pour les courbes de visualisation. Seul bémol, les mots sont générés aléatoirement à partir des mots insérés dans le code.

6 Conclusion

Ce projet a permis de mettre en évidence l'écart entre la théorie et la pratique. Si la complexité théorique donne une tendance, les détails d'implémentation (cache, allocations mémoire) le sont d'autant.

L'algorithme basé sur les tableaux dynamiques semble être le meilleur compromis performance/mémoire pour compter le nombre de mots dans un texte.