Rapport de projet

par Niekita Joseph et Stephany Santos Ferreira de Sousa

Le manuel utilisateur se trouve dans l'archive `.zip` du projet, rédigé dans le `ReadMe.txt`.

Sommaire

Architecture des fichiers

`/src` - fichier source

`/structs` - fichiers de structures

`/textes` - fichiers textes d'exemples utilisés

`plot_perf.py` - script python générant des courbes des performances des algorithmes.

`README.txt` - fichier read me contenant le manuel utilisateur

`main.c` - fichier principal à compiler

`perf.txt` - fichier texte utilisable afin d'enregistrer les performances du programme

`resultats.txt` - fichier txt utilisable afin d'enregistrer les résultats

Algorithmes et Structures

Listes

Le premier algorithme utilise des listes simples comme structure. Après avoir parcouru le texte et récupéré chaque caractère, l'algorithme va séparer les mots et créer des objets de type Mot dans lequel on pourra stocker la chaîne de caractère ainsi que son nombre d'occurrences.

La structure non triée lors de l'analyse du fichier permet à l'algorithme de compléter sa tâche rapidement ; le tri se fait à la fin, grâce à un tri par sélection.

Pendant le développement de l'algorithme 1, lors de la compilation, un message d'erreur s'affichait :

```
In file included from structs/algo1.h:4,
                 from main.c:7:
structs/alloc.h:9:3: error: conflicting types for
'InfoMem'; have 'struct <anonymous>'
    9 | } InfoMem;
etc…
```

Ce problème apparaît lorsqu'on utilise plusieurs fois un même fichier est inclu dans d'autres, par exemple *alloc.h,* De ce fait, on a rajouté par la suite #ifndef, #define et #endif dans tous les fichiers .h pour plus de sécurité et éviter ce genre de problème par la suite.

Listes chaînées

Le deuxième algorithme utilise la structure des listes chaînées, chaque mot du texte est représenté par une cellule contenant trois informations : la chaîne de caractère (mot), le nombre d'occurrences (nb_occ), ainsi qu'un pointeur vers la cellule suivante de la liste (suivant).

Comme l’algorithme précédent, on utilise la recherche linéaire. Quand un nouveau mot, s’il est déjà présent, on augmente simplement le compteur, sinon, on crée une nouvelle cellule pour ce mot et on l’ajoute dans la liste.

Piles triées

Ce troisième algorithme dépend principalement de structures de piles - le premier élément ajouté sera le dernier sorti. Cependant, l’implémentation avec les listes chaînées permet aussi une lecture ordonnée du début jusqu’à la fin. Ainsi, chaque mot, encore inséré dans une structure Mot, lors d’une première apparition dans le texte, est ajouté à la fin de la pile; les mots avec le plus d'occurrences, finissent au fond de la pile grâce un tri par insertion sur liste chaînées au fur et à mesure du programme.

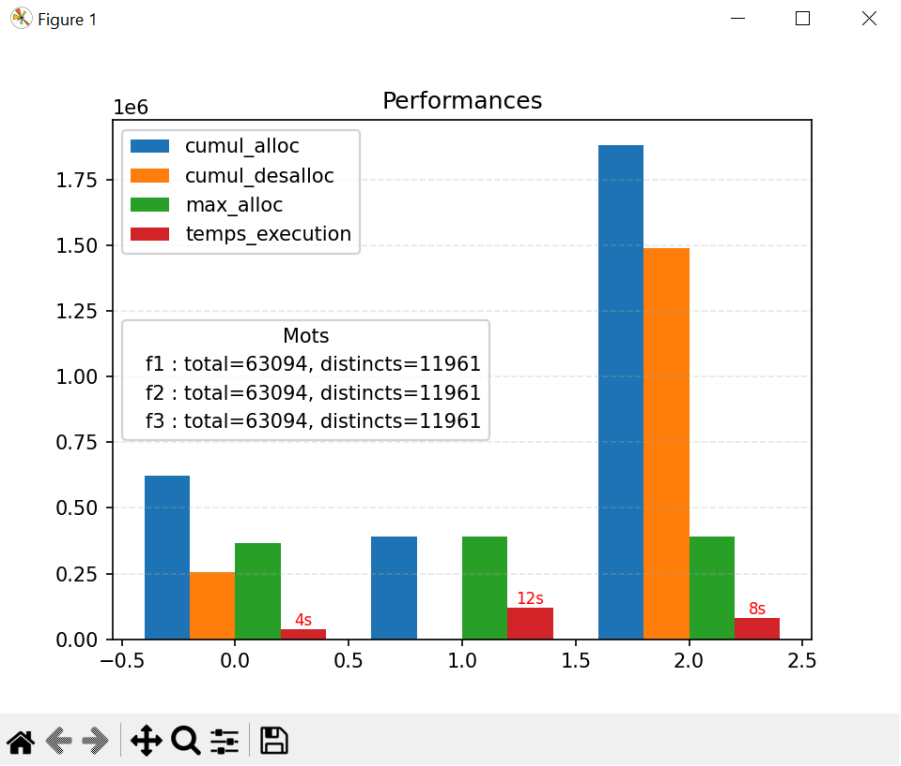
Dans les premiers essais de développement de cet algorithme, il était prévu de créer une pile non triée, ce qui a remis en question l’intérêt de construire un algorithme structuré de piles. L’idée de trier les mots selon leur occurrence a permis d’ainsi récupérer plus aisément les mots les plus répétés, se retrouvant ainsi au début de la pile. Par la suite, un tri bulle a été testé, ce qui a rendu l’algorithme bien trop long à s’exécuter.

Performances

Algorithme	Liste	Liste chaînée	Piles
Allocation	609 Ko	383 Ko	1766 Ko
Désallocation	251 Ko	0	1383 Ko
Allocation maximale	358 Ko	383 Ko	383 Ko
Temps d'exécution	4 secondes	12 secondes	8 secondes

Note : Ces données peuvent varier selon les exécutions, et sont toutes basées sur le même fichier que le graphique.

Ci-dessous se trouve le graphique comparant les performances des trois algorithmes, les données de mémoire étant en octets.



Les tests ont été faits sur le même fichier texte suffisamment massif pour pouvoir comparer les données. Les statistiques de ce fichier sont affichées ; chaque algorithme parvient à trouver le même résultat. Lequel des trois est le plus efficace ?

Le premier algorithme, utilisant les listes, est le plus rapide des trois, mais le second algorithme se démarque par son utilisation minime de l'espace mémoire. En effet, l'algorithme se reposant sur des listes chaînées alloue l'espace minimal requis à son fonctionnement; en comparant le cumul d'allocation et l'allocation totale utilisée, on remarque qu'il s'agit de la même quantité d'octets. Enfin, le troisième algorithme est le plus consommateur des trois en termes de mémoire, mais se place deuxième en termes de rapidité.

Organisation

Il a tout d'abord été choisi de traiter des algorithmes dont les structures seraient la plus grosse différence. Ainsi, nous nous sommes mises d'accord pour que Stephany s'occupe des listes triées et listes chaînées, et que Niekita fasse les listes non triées et les piles.

A la fin d'une première partie de développement, il était évident que les algorithmes des listes non triées et triés étaient similaires ; nous avons donc gardé l'algorithme et poursuivi le développement des listes chaînées et piles.

Au final, s'il nous est demandé d'attribuer un pourcentage de participation, 65% du travail reviendrait à Stephany pour son travail sur deux des algorithmes implémentés ainsi que sur le fichier d'exécution principal, que Niekita aura complété de son troisième algorithme par la suite.