UFR de mathématique et d'informatique
Université de Strasbourg

Licence 2 Informatique
Semestre 4 - Automne 2021

PROJET DE STRUCTURE DE DONNÉES
ET ALGORITHMIQUE

Compression de fichiers avec l'algorithme LZW

SADAY Arhun, ALCINDOR Jodel

1. Introduction

Dans ce projet, nous implémentons et étudions à travers trois structures de données différentes, des algorithmes de compression et de décompression de fichiers sans pertes de données dite LZW du nom de ses auteurs Lempel-Ziv-Welch.
L'objectif principal de cette étude est d'analyser et de comparer les performances de l'algorithme en fonction de la structure de données choisie.

Dans notre analyse, pour chaque structure de données nous prendrons en compte la complexité en temps, en espace mais également la facilité avec laquelle nous avons implémenté chacune d'entre elles.

Dans un premier temps, nous rappellerons brièvement les principes de l'algorithme de compression et de décompression LZW.

Dans un second temps, nous présenterons les fonctionnements des algorithmes écrits pour chaque structure de données.

Ensuite, nous étudierons la performance de chaque structure de données pour la compression et la décompression des fichiers.

Pour finir, nous conclurons en expliquant l'importance des choix des structures de données dans les applications.

2. Algorithme de compression et de décompression LZW

2.1 Principe: Algorithme de compression

L'algorithme de compression LZW commence par initialiser un dictionnaire de 256 caractères correspondants à l'encodage ASCII étendu sur 8 bits. Le dictionnaire peut prendre plusieurs formes mais dans l'idée, c'est une collection d'objets sous la forme clé/valeur. On commence par lire dans le fichier texte, 1 octet à la fois. Si on croise une nouvelle sous-chaîne, on l'ajoute au dictionnaire. Si on croise une sous-chaîne qui est déjà dans le dictionnaire, on lit un nouveau caractère et on ajoute la concaténation de ces deux sous chaînes au dictionnaire. Ainsi, la prochaine fois qu'on croise cette sous-chaîne, on pourra l'encoder avec une seule valeur hexadécimale. Puisque les codes hexadécimaux prennent moins de place que les sous-chaînes qu'ils remplacent, on obtient une compression.

2.2 Pseudocode

```
1 Initialiser la table avec des chaînes de caractères uniques
2 P = premier caractère lu
3 TANT QUE il y a des caractères à lire dans le fichier
4    C = prochain caractère lu
5    SI P + C est dans le dictionnaire
6       P = P + C
7    SINON
8       écrire le code pour P
9       ajouter P + C au dictionnaire
10      P = C
11 TERMINER PENDANT
12 écrire le code pour P
```

2.4 Principe : Algorithme de décompression

Pour l'algorithme de décompression, nous nous servons du fichier compressé pour restituer le fichier original. Ainsi, suivant le même principe de l'algorithme de compression, nous initialisons un dictionnaire avec les 256 premiers caractères ASCII. Ensuite, tant qu'il reste des caractères à lire, nous nous servons du tableau de codes pour décoder les valeurs.

2.5 Pseudocode

```
1 Initialiser la table avec des chaînes de caractères uniques
2 OLD = premier code d'entrée
3 traduction de sortie de OLD
4 TANT QUE pas la fin du flux d'entrée
5     NOUVEAU = prochain code d'entrée
6     SI NEW n'est pas dans la table de chaînes
7           S = traduction de OLD
8           S = S + C
9     AUTRE
10          S = traduction de NOUVEAU
11          sorties S
12          C = premier caractère de S
13          OLD + C à la table des chaînes
14 ANCIEN = NOUVEAU
15 FIN PENDANT
```

3. Structures de données utilisées

a. Liste chaînée

La première structure de données utilisée pour représenter le dictionnaire est une liste chaînée. En effet, nous avons créé une structure `listeNoeud` qui représente chaque élément du dictionnaire et qui contient trois champs. Chaque nœud de la liste est en effet décomposée en un champ clé, un champ valeur suivie d'un pointeur vers le prochain nœud. Pour la compression, la clé correspondra à la chaîne de caractère encodée, la valeur représente l'encodage de cette chaîne sous forme hexadécimale. Afin de simplifier la rédaction des algorithmes, nous avons utilisé les fonctions suivantes:

- initialiser liste (ListeNoeud liste) : pour initialiser le dictionnaire avec les 256 premiers caractères ASCII.
- liberer_liste (ListeNoeud liste): pour libérer les zones de mémoires utilisées par la liste.
- inserer_liste (ListeNoeud liste, char* clé, char* valeur) : renvoyant un pointeur sur le dictionnaire auquel on ajoute un couple clé/valeur.
- recuperer_liste (ListeNoeud liste, char* clé) : renvoyant la valeur associée à la clé.

Quoique l'implémentation du dictionnaire avec cette structure reste assez simple, nous étudierons dans la quatrième partie du rapport, la complexité en temps et en en espace mémoire d'une telle implémentation.

b. Trie

Afin de représenter le dictionnaire avec la structure trie, nous avons implémenté un structure de trie où chaque noeud contient un tableau de 256 pointeurs vers ses noeuds fils (qui correspondent au 256 possibles caractères dans l'encodage ASCII étendu), un booléen permettant de confirmer éventuellement la validité de toute clé formant un mot partant de la racine jusqu'au nœud courant, suivi d'une chaîne de caractères correspondant à la valeur à stocker dans le tableau si la clé courante est valide.
Tout comme pour les listes, nous avons implémenté des fonctions basiques similaires à celles utilisées pour les listes.

c. Hashmap

L'implémentation de la structure hashmap nous avait été fournie. Les fonctions qui nous ont été utiles sont les fonctions pour la création d'une hashmap, ajouter et récupérer un élément. Nous avons également implémenté une fonction liberer_hashmap pour libérer la mémoire associée au dictionnaire puisque la fonction hashmap_destroy n'est pas suffisant.

4. Stratégie d'implémentation

Pour la stratégie de l'implémentation à utiliser, nous avons utilisé le type "unsigned char" pour stocker les codes dans les fichiers .lzw et nous avons opté pour une gestion dynamique du dictionnaire. En effet, lorsque nous remplissons entièrement le dictionnaire, nous vidons le dictionnaire en le réinitialisant pour pouvoir continuer à y ajouter de nouveaux codes.

5. Tests de performances

Afin de bien visualiser et analyser les performances de l'algorithme de compression et de décompression en fonction de chaque structure de données, nous avons utilisé des fichiers de taille différente, dont un vide de taille 0, un autre de 10 octets, puis un autre de 100, suivi de

deux autres fichiers de taille moyenne de 1000 et 10000 octets. Enfin, nous avons utilisé trois autres longs fichiers de taille respective 100000, 250000 et 500000 octets.
Chacun de ces fichiers sont compressés puis décompressés en fonction des trois structures de données et comparés ensuite aux fichiers originaux.

Dans un premier temps, nous utilisons un premier script shell afin de nous assurer de la fonctionnalité du programme. Nous posons des restrictions de test sur les arguments à utiliser sur le programme. En effet, le script shell *test_fonctionnement.sh* utilisé dans le projet se charge dans un premier temps de vérifier la validité du nombre d'arguments passés en paramètre, du nom du fichier à compresser ainsi que son extension, et pour finir l'option à utiliser.
Ensuite, à l'aide de l'utilitaire Linux cmp, après avoir lancé des tests de compression et de décompression sur des fichiers de taille différentes, nous comparons les fichiers décompressés à chaque fois avec les originaux.

Par la suite, afin de tester la performance du programme, nous lançons un script shell, *test_performance.sh* qui s'assure de lancer l'algorithme LZW pour compresser puis décompresser chaque fichier de test 5 fois et ce pour chaque structure de données. La moyenne des temps de calcul de chaque fichier est ensuite sauvegardée respectivement dans trois fichiers de temps, *liste.time*, *trie.time* et *hashmap.time* que nous utilisons ensuite pour tracer les graphes de performance. Celles-ci nous permettent de voir clairement le coût en temps de l'algorithme en fonction de la structure de données utilisée et en en fonction de la taille du fichier passé en paramètre.

Graphique et données

	liste-chainee.time	trie.time	hashmap.time
0	1.083	1.149	1.043
10	1.468	1.816	13.347
100	1.751	2.168	13.176
1000	8.251	4.716	14.814
10000	275.807	25.531	18.930
100000	12429.818	170.630	79.497
250000	26119.940	215.421	115.983
500000	56691.648	424.167	209.604

Figure 1. Tableau de temps pris par chaque structure de données en fonction de la taille du fichier

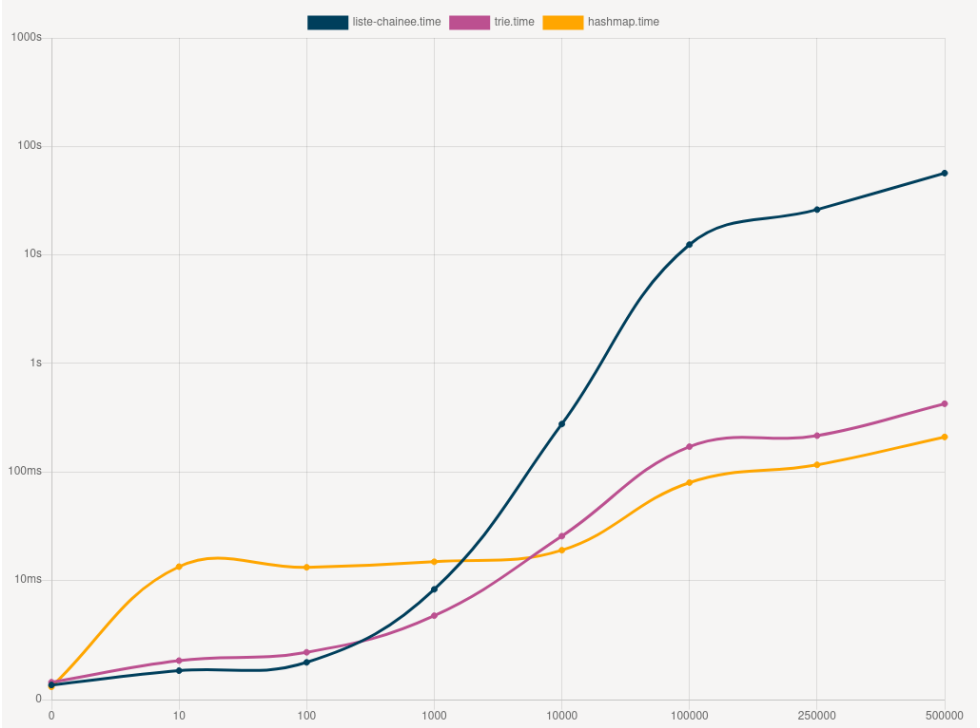

Figure 2. Graphe illustrant la performance de l'algorithme LZW en fonction de la structure de données utilisé

Analyse

Pour la complexité en temps, il faut avoir en tête que dans le cas de l'algorithme LZW, les opérations principales qui nous intéressent sont l'insertion et la recherche. Nous ne faisons pas de suppression, par exemple.

Comme on peut l'observer, la complexité en temps dépend directement de la taille du fichier, plus le fichier est grand plus le temps de compression et de décompression est important. On voit que pour des fichiers de taille relativement petite, le hashmap est plus lent que le trie et la liste chaînée. Cela peut s'expliquer par le fait que le hashmap est une structure de données plus complexe que les deux autres, qui nécessite donc plus d'opérations et d'espace mémoire pour être créé et initialisé. Malgré cela, on sait que la recherche et l'insertion dans une hashmap se fait en temps constant grâce à la fonction de hachage donc ça devrait être très optimal pour notre algorithme. On constate que c'est le cas, pour des fichiers de grande taille c'est clairement la structure de données la plus efficace en temps. Comparé au hashmap, l'algorithme est deux fois plus lent avec la structure de données Trie. En effet le Trie a pris en moyenne 424 ms compresser et ensuite décompresser un fichier de 500 milles octets tandis que le hashmap seulement 209 ms. La recherche et l'insertion dans une Trie dépendent de la longueur de la clé que nous recherchons et donc cela se fait en temps O(longueur_clé) dans le pire cas, mais cela peut se faire plus rapidement si la clé n'existe pas.
Pour ce qui est de la liste chaînée, c'est clairement le pire des 3. Même si pour des petits fichiers, c'est plus rapide, pour un fichier de 500 milles octets, par exemple c'est 270 fois plus lent que le hashmap. C'est totalement logique, puisque pour insérer et rechercher un élément

il faut parcourir tous les éléments de la liste (potentiellement 65536 dans notre cas). C'est possible de diminuer le temps d'insertion en O(1) en faisant une liste doublement chaînée et en insérant à la fin mais le temps de recherche reste néanmoins très important. Dans l'ensemble avoir une taille de dictionnaire assez petite est donc primordial pour la liste chaînée et le Trie puisque la complexité en temps en dépend grandement.

Toutefois, si l'implémentation avec le hashmap possède des avantages, elle comporte également des inconvénients. En effet, nous pouvons remarquer que des trois structures de données, elle est la plus coûteuse en espace mémoire. Suite au débogage de l'algorithme avec l'outil valgrind, nous avons pu constater que l'espace mémoire allouée et désallouée en utilisant la liste chainée tournait autour de 275 000 octets, suivie de l'algorithme du trie qui en utilise 20 fois plus. L'implémentation avec le hashmap quant à elle, dont l'espace mémoire utilisée tourne souvent autour d'une puissance de 2, en utilise 40 fois plus.
Si l'implémentation avec la liste demeure la plus lente, elle reste tout de même la moins coûteuse en espace mémoire.

Pour ce qui est de l'implémentation, la liste chaînée est une des structures de données les plus basiques donc il n'y a aucune difficulté pour l'implémenter. Le Trie est un peu moins évident mais reste tout de même assez facile à implémenter du fait de sa ressemblance avec les arbres. Nous n'avons pas eu à implémenter le hashmap mais c'est clairement le plus difficile des 3 structures notamment à cause de la fonction de hachage dont le choix et la conception est primordiale pour avoir de bonnes performances et éviter les collisions entre les éléments.

5. Conclusion

En réalisant ce projet, nous avons pu nous rendre compte de la complexité et des difficultés qui peuvent être celles d'un développeur lorsqu'il s'agit de mettre en place une application rapide et efficace. En effet, nous nous sommes rendus compte de l'importance de réfléchir de manière assidue au choix de la structure de données à utiliser.

Nous avons également pu constater que l'efficacité de l'algorithme LZW à compresser des fichiers dépend de plusieurs facteurs. Tout d'abord, la taille du fichier est évidemment primordiale. Pour des fichiers très petits, la compression est inexistante voir négative, c'est à dire qu'on peut avoir un fichier avec le format ".lwz" plus lourd que son équivalent en ".txt". Pour des fichiers de grande taille, la compression LZW est très intéressante. Par exemple pour le fichier "the-adventures-of-sherlock-holmes.txt", la version compressé est à peu près 55% moins lourd. Un autre critère qui peut impacter le taux de compression peut être le contenu du fichier. Si les données sont aléatoires et sans aucune logique, la compression ne fonctionnera pas. De même, si le contenu d'un fichier change radicalement, par exemple la moitié du texte est en français et l'autre moitié en russe, la compression ne marchera plus du tout. Une bonne implémentation pourrait donc surveiller le taux de compression et vider le dictionnaire lorsqu'elle passe sous un certain seuil.