UFR de mathématique et d'informatique
Université de Strasbourg

Rapport de Projet Informatique Structures de Données et Algorithmiques 2

COMPRESSION DE FICHIER

Équipe

BONBON-PERIAC Adam
DINTE Alexandru

Encadrants

HETROY-WHEELER Franck
MORGENTHALER Sébastien
HAAS Antoine

Introduction:

Ce rapport a pour but de décrire le déroulement de notre projet d'informatique sur l'algorithme LZW qui permet la compression d'un fichier.

Dans un premier temps, l'idée était d'implémenter un dictionnaire pour permettre le bon fonctionnement de l'algorithme. On nous propose, pour cela, trois différentes structures.

Tout d'abord, la première implémentation se fera sous forme d'une liste chaînée. Ensuite, la deuxième implémentation sera un trie, qui permet sa représentation sous forme d'arbre. Enfin, sous forme d'un tableau de hachage.

Après l'implémentation du dictionnaire, il ne reste plus qu'à implémenter l'algorithme de compression et décompression de LZW.

Des ressources, tels que des fonctions pour lire dans un fichier, changer des caractères en chaîne de caractères et la libraire hashmap,h étaient mises à notre disposition

Implémentation de la liste chaînée:

Une liste liée est un moyen de stocker une collection d'éléments. Comme dans un tableau, il peut s'agir de caractères ou d'entiers. Chaque élément d'une liste liée est stocké sous la forme d'un nœud. Une liste chaînée est formée lorsque plusieurs nœuds de ce type sont reliés entre eux pour former une chaîne. Chaque nœud pointe vers le nœud suivant présent dans l'ordre. Le premier nœud est toujours utilisé comme référence pour parcourir la liste, et est appelé tête. Le dernier nœud pointe vers NULL.

Avantages de la liste chaînée :

- Lorsque l'utilisateur a besoin d'une insertion/suppression peu coûteuse, car il peut insérer/supprimer les nœuds à tout moment.
- Lorsque vous êtes imprévisible quant au nombre d'éléments (taille de la liste) car dans une liste chaînée, vous n'avez pas besoin de déclarer la taille.
- Il n'est pas nécessaire de déplacer les éléments lorsque vous insérez/supprimez.

Déclaration d'une liste chaînée :

En langage C, on a implémenté la liste chaînée en utilisant la structure suivante.

```
typedef struct List
{
	int c; // key
	int p; // prefix
	int value; // index
	struct List * next;
} List;
```

La définition ci-dessus est utilisée pour créer chaque nœud de la liste. Les champs c, p, value stockent l'élément et le champ next est un pointeur pour stocker l'adresse du nœud suivant.

Initialisation du dictionnaire:

```
List * initialize_dictionary()
{
	List * dictionary = (List*)malloc(sizeof(List));
	for (int i = 0; i < CHAR_MAX; i++)
	{
		List * node_to_add = (List*)malloc(sizeof(List));
		node_to_add->c = i;
		node_to_add->p = -1;

		add_element(dictionary, node_to_add);
	}
	return dictionary;
}
```

L'algorithme d'initialisation est simple. On crée 256 nœuds, chaque nœud possède une valeur de 0 a CHAR_MAX, ou CHAR_MAX = 256 (extended ASCII table) et on les rajoute dans le dictionnaire. La fonction malloc() est utilisé pour allouer dynamiquement la mémoire en C. La procédure add_element permets d'insérer un élément en queue de la liste. En résumant, la fonction initialize_dictionary crée une liste avec 256 nœuds.

Insertion d'un nœud en queue de la liste:

L'algorithme d'insertion d'un nœud en queue de la liste est trivial. Il suffit de suivre quatre étapes:

1. Allouer de la mémoire pour le nouveau nœud
2. Stocker les données
3. Traverser jusqu'au dernier nœud
4. Changer le suivant du dernier nœud en nœud récemment créé

```
List * add_element(List * dictionary, List * node_to_add)
{
        List * last_node = find_last_node(dictionary);

        if (dictionary != NULL)
                last_node->next = node_to_add;
        else dictionary = node_to_add;

        last_node = node_to_add;
        node_to_add->next = NULL;

        return node_to_add;
}
```

Vu qu'on alloue la mémoire pour le nouveau nœud dans l'initialisation de dictionnaire, il est pas nécessaire de le faire ici. Si le dictionnaire est vide, l'élément a ajouter va être le premier élément de la liste, sinon, on va le rajouter après le dernier nœud.

La procédure find_last_node prend en argument le dictionnaire, cherche le dernier élément et le return. La complexité de la fonction add_element est de O(n), où n est le nombre de nœuds dans la liste. Comme il y a une boucle de la tête à la fin, la fonction effectue un travail de O(n).

Recherche dans la liste:

Pour rechercher un élément dans le dictionnaire , il faut suivre le pseudo-code suivant:

1) Initialiser un pointeur de noeud, element = dictionnaire.
2) Tant qu'on n'est pas a la fin de la liste
   a) element->key est égal à la clé recherchée, return element->value.
   b) element = element->next
3) Retourner -1

En termes de complexité, la recherche prend O(n).

L’implémentation pour la compression et décompression LZW a l’aide d’une liste chaînée était plutôt accessible car il s’agit d’une structure de données qu’on a vu au semestre dernier, et par conséquent, nous savions déjà toutes les implémentations des fonctions, les complexités et comment le code doit être structuré pour ce projet. Même si cette structure des données était la plus facile des trois, ce n’est pas la plus efficace pour la compression/décompression LZW.

Implémentation du trie:

Trie est une structure de données arborescente, utilisée pour la recherche efficace d'une clé dans un grand ensemble de données de chaînes de caractères. Contrairement à un arbre de recherche binaire, où un nœud stocke la clé associée à ce nœud, la position du nœud Trie dans l'arbre définit la clé à laquelle il est associé, et la clé est uniquement associée aux feuilles. Il est également connu sous le nom d'arbre à préfixe car tous les descendants d'un nœud ont un préfixe commun de la chaîne associée à ce nœud, et la racine est associée à la chaîne vide.

Il existe plusieurs façons de représenter un Trie, correspondant à différents compromis entre l'utilisation de la mémoire et la vitesse des opérations. La forme de base est celle d'un ensemble lié de nœuds, où chaque nœud contient un tableau de pointeurs enfants, un pour chaque symbole. Le nœud Trie maintient également un flag qui spécifie s'il correspond à la fin de la clé ou non.

On a choisi d’implémenter le tri en utilisant la structure suivante:

```
struct Trie {
        bool isWord;
        char * value;
        struct Trie * children[CHAR_MAX];
};
```

Elle consiste d’un tableau de pointeurs vers ses fils, de taille 256. Une case i de ce tableau est un pointeur vers son noeud fils si il est possible de continuer un mot (dans notre cas la clé) avec la lettre d’indice i à partir du préfixe encodé par le noeud courant, ou un pointeur NULL sinon. IsWord est l’attribut permettant de savoir si le mot formé en partant de la racine et en allant jusqu’au nœud courant est une clé valide du dictionnaire. Et on a char* value, pour stocker la valeur en hexa.

Implémentation du dictionnaire

- L’initialisation de trie se fait en allouant de la mémoire et en initialisant chaque case de tableau avec NULL.
- L'insertion d'une clé dans Trie est une approche simple. Chaque caractère de la clé d'entrée est inséré comme un nœud Trie individuel. Notez que le tableau children est un tableau de pointeurs (ou de références) vers les nœuds Trie de niveau suivant. Le caractère de la clé agit comme un index dans le tableau children. Si la clé d'entrée est nouvelle ou une extension de la clé existante, nous devons construire les nœuds inexistants de la clé, et marquer la fin du mot pour le dernier nœud. Si la clé d'entrée est un préfixe de la clé existante dans Trie, nous marquons simplement le dernier nœud de la clé comme la fin d'un mot. La longueur de la clé détermine la profondeur de Trie.

- La recherche d'une clé est similaire à l'opération d'insertion, cependant, nous ne comparons que les caractères et nous descendons. La recherche peut se terminer en raison de la fin d'une chaîne de caractères ou de l'absence de clé dans le tri. Dans le premier cas, si le champ isWord du dernier noeud est vrai, alors la clé existe dans le tri. Dans le second cas, la recherche se termine sans examiner tous les caractères de la clé, puisque la clé n'est pas présente dans le tri.

Performances de Trie

La complexité temporelle d'une structure de données Trie pour une opération d'insertion/recherche est juste O(n), où n est la longueur de la clé.

Étant donné que c'était notre première fois sur l'arbre Trie, nous avions trouvé que cette structure des données est la plus difficile des trois. La plupart de temps, nous avons eu beaucoup de mal à comprendre quel était le principe et comment l'implémenter pour notre projet. Les opérations d'insertion et de recherche d'une paire clé/valeur nous ont posé le plus de problème, nous avons d'ailleurs encore des doutes la validité de l'implémentation. Cela dit, c'était une bonne expérience. Nous avons appris une nouvelle structure de données.

Implémentation de la librairie hashmap:

L'objectif a été d'intégrer une structure de donnée existante sous forme de libraire et d'utiliser les fonctions adéquates pour résoudre notre problème.

Dans la librairie, il existait plusieurs fonctions qui utilisait la structure hashmap. Les suivantes, sont celles utilisés dans notre implémentation:

hashmap_create qui initialise une hashmap
hashmap_put qui intègre une clé et sa valeur dans une hashmap
hashmap_get qui permet de récupérer la valeur en fonction d'une clé
hashmap_destroy qui libère l'espace mémoire alloué à la hashmap

Implémentation du dictionnaire:

initialisation du dictionnaire

Pour commencer, on alloue le hashmap dans la mémoire puis on appelle la fonction hashmap_create.
Cette fonction permet de créer un tableau de hachage d'une taille défini, ici 1024.

Ensuite, on ajoute les éléments dans le dictionnaire, nous avons choisi les 256 caractères dans la table ASCII. Pour cela, nous utiliserons la fonction hashmap_put.
La clé se crée à l'aide d'une valeur i qu'on associe à un caractère ASCII. On le transformera plus tard en chaîne de caractère pour l'intégrer dans la fonction. La valeur se calcule avec l'adresse de i et on y ajoute sa valeur. Cela permettra à ce que chaque caractère ont des valeurs différentes.

Enfin, nous retournons le dictionnaire initialisé en tant que tableau de hachage.

```
Algo init_dict:
        allocation hashmap dictionnaire
        hashmap_create(dictionnaire)
        pour tout i qui va de 0 à 255
                j ← i
                chara ← int2char(i)
                string ← char2string(chara)
                valeur ← adresse de i + j
                hashmap_put(dictionnaire, string, taille(string), valeur)
        finpour
        retourne dictionnaire
```

libérer le dictionnaire

Pour libérer le dictionnaire, on appelle la fonction hashmap_destroy pour détruire le dictionnaire et libérer son espace mémoire.

```
Algo free_dict(dictionnaire):
        hashmap_destroy(dictionnaire)
```

ajouter un élément dans un dictionnaire

On appelle la fonction hashmap_put comme lorsque nous avions initialisé le dictionnaire. Pour l'ajouter, nous avons besoin de la clé, la taille de la clé et sa valeur.

```
Algo add_to_dict(dictionnaire, clé, valeur):
        hashmap_put(dictionnaire, clé, taille(clé), valeur)
```

chercher un élément dans un dictionnaire

On utilise la fonction hashmap_get pour chercher un élément dans un dictionnaire. Si, nous avons trouver l'élément, on retourne 0 sinon -1.

```
Algo cherche_element(dictionnaire, clé):
        si hashmap_get(dictionnaire, clé, taille(clé)) != HASHMAP_NULL
                retourne 0
        sinon
                retourne -1
```

La complexité en pire cas est de O(n) qui représente la dernière case du tableau de hachage mais a une complexité moyenne de O(1), ce qui est nettement mieux que les structures précédentes théoriquement.

Compression de fichier

En utilisant, l'algo de LZW, nous pouvons grâce l'implémentation du dictionnaire via la structure hashmap compresser notre fichier.

On récupère les caractères du fichier à l'aide de fgets. Nous avions eu toujours l'habitude avec les autres structures de données, d'utiliser fgetc ou getc car nos caractère étaient de type int. Ici, notre librairie impose que nos clés soient de type char*.
On utilisera alors deux buffer. Un pour stocker le préfixe et un pour stocker le caractère suivant.
On initialise nos valeurs i et j à 256 lorsqu'on devra ajouter un caractère dans le dictionnaire.
Enfin, nous suivons le déroulé de l'algorithme comme avec les deux autres structures de données.
On n'oubliera pas d'incrémenter i et j lorsqu'on on repasse à la boucle pour parcourir les chaînes de caractères

Performances

Cette structure est la structure la plus efficace pour l'implémentation du dictionnaire et de la compression des fichiers.
En effet, en comparant le temps de compression entre chaque structure, la plus rapide est celle du tableau de hachage.

Pourtant, elle n'est pas parfaite. En effet, si nous compressons un petit fichier de 7 octets, alors le fichier compressé aura comme taille plus de 20 octets. Le but de la compression était de réduire la taille d'un fichier mais, ici, la taille est plus grande que le fichier originale.
En revanche, si on compare la taille du fichier lorem-ipsum.txt et ce même fichier compressé, il y a un rapport de plus 95%. Cela paraît trop peu comparé à ce que pourrait donner la compression en format zip ou tar.gz. N'ayant plus eu le temps de faire la décompression, nous n'avons pas pu vérifier

Analyse théorique et pratique

Bien que l'implémentation du hashmap semble plus facile que le trie et la liste, elle a quand même nécessité pas mal de temps de compréhension de la structure. De plus, il fallait adapter notre code par rapport aux fonctions qui nous étaient donnés.
Si par exemple, nous avions décidé d'un certain type à une donnée mais que la structure nous imposait une autre, alors il fallait changer tout notre programme. Contrairement, aux structures précédentes, nous n'étions pas «libres» à faire ce que nous avions en tête ou imaginés, nous avions comme base une structure toute faîte. Il est certes vrai, que cela fait moins de programmes à faire mais plus de compréhensions à acquérir.

Cela nous a permis, toutefois, à apprendre à implémenter une table de hachage car nous en avons jamais eu l'occasion durant les TP où nous avons appris à programmer jusqu'au graphes. C'est un bon commencement, pour comprendre et implémenter la table.

Conclusion

Le projet était très intéressant. Le concept de compresser et décompresser un fichier nous a plu même si au départ, nous le pensions difficilement réalisable. Bien que nous n'ayons pas eu le temps de nous occuper de la décompression et d'optimiser un peu plus notre code pour rendre le résultat vraisemblable, nous avons quand même réussi à avoir la démarche nécessaire pour mettre en œuvre les différentes structures et les implémenter pour la compression en se servant de l'algorithme LZW. De plus, nous avons discuté des différentes performances pratiques et de la facilité d'implémentation de chaque structure.

La première structure, la liste chaînée était la plus simple parmi les trois mais la moins efficace en terme de temps théoriquement. Tandis, que l'implémentation du trie était la plus compliquée mais gagne en gain de temps théorique. Enfin, la hashmap se basait sur la compréhension de la structure hashmap.h. Elle est la structure qui a un temps le plus efficace entre les trois structures.
En ce qui concerne la pratique, en testant les différents temps de compression dans le fichier lorem-ipsum.txt, il advient que c'est le trie qui prend un tout petit plus de temps que la liste chaînée et hashmap qui est le plus rapide.

Annexe: Temps de compression de lorem-ipsum.txt en fonction des structures

	Hashmap.time	Trie.time	Liste.time
12356	31561.000	88660.000	85054.000