Projet Bzip2

JAVAID Mohammad-Habib L2X

25/05/2022

Contents

1 Présentation du projet **2**

2 Comment utiliser le programme ? **2**
2.1 Compilation 2
2.2 Aide 2
2.3 Compression - Décompression 2

3 Structure du code **2**
3.1 Compression 2
3.1.1 La fonction `write_compressed_file` 3
3.1.2 Transformée de Burrows-Wheeler 6
3.2 Décompression 7

4 Conclusion **7**

1 Présentation du projet

L'objectif du projet que j'ai choisi était de faire un programme de compression qui utilise Burrows-Wheeler et Huffman.

2 Comment utiliser le programme ?

2.1 Compilation

Pour commencer, il faut faire la commande `make`, l'exécutable est le fichier `./compresseur`.

2.2 Aide

Pour avoir toutes les indications, il suffit d'écrire `./compresseur`, `./compresseur -h` ou `./compresseur --help`.

2.3 Compression - Décompression

Pour compresser un programme, l'on fait : `./compresseur -c votre_fichier.txt votre_output`.
Pour décompresser un programme, l'on fait : `./compresseur -d votre_fichier.bz2 votre_output`.
Dans les deux cas, définir un nom d'output particulier est optionnel.

3 Structure du code

Le code est composé d'un ensemble de fonction pour compresser, pour décompresser, et du main où l'on gère la gestion des commandes du terminal. Je ne détaillerais pas le main car c'est assez bien commenté dans le `main.cpp`, je me contenterais d'expliquer les fonctions principales permettant de comprendre mon code.

3.1 Compression

Voici la fonction `compression` qui réalise tout le processus de compression du fichier entré par l'utilisateur :

```
void compression(string to_compress, string output_file) {
  string the_text_file = text_file_to_string(to_compress);
  vector<pair<char, int>> occurences_of_each_char =
      string_to_vector_of_occurrences(the_text_file);

  // On transforme notre vecteur d'occurence en arbre
  Node *my_tree = vector_of_occurences_to_tree(occurences_of_each_char);

  // on transforme notre arbre d'occurence en une liste de codage pour chaque
  // char
 map<deque<bool>, char> my_coding_list;
  deque<bool> my_coding;
  tree_to_dict_of_binary_codage(my_coding_list, my_coding, my_tree);

  // on libère la mémoire de notre arbre, il nous sert plus
  free_tree(my_tree);

  // ecrire le fichier
  write_compressed_file(output_file, the_text_file, my_coding_list);

  cout << "Compression completed!" << endl;
}
```

Il y a 5 grandes étapes pour compresser :

- on récupère le contenu du fichier avec `text_file_to_string()`
- on récupère les occurence de chaque caractère avec `string_to_vector_of_occurrences`
- à partir des vecteur d'occurences, on créer un arbre de Huffman avec `vector_of_occurences_to_tree`
- on transforme l'arbre en dictionnaire de codage, avec l'association de chaque caractère de l'arbre avec son codage, avec `tree_to_dict_of_binary_codage`
- on passe à l'écriture du fichier compressé à partir du dictionnaire, avec `write_compressed_file`

3.1.1 La fonction `write_compressed_file`

J'aimerais m'attarder un peu sur cette fonction car c'est sur celle-ci que je pense avoir passé le plus de temps.

En effet, c'était un véritable défi pour moi d'écrire la compression dans un fichier et ce, pour deux raisons :

- il faut que le texte compressé soi séparer du dictionnaire, et qu'on puisse identifier quelle partie correspond à quoi lors de la décompression
- on ne peut pas écrire bit par bit en C++, on est obligé d'écrire octet par octet au minimum, sauf que nos codages sont parfois inférieur ou supérieur à un octet. De plus, écrire un codage par octet provoquerait un gaspillage de place rendant la compression inutile, voire plus lourde que le fichier original car le plus souvent, les codages écrit sont inférieurs à 1 octet.

1. Régler le problème de l'écriture par octet :

   (a) Le texte à compresser :
   Pour régler ce problème, j'ai pris chaque caractère de mon texte à compresser, j'ai pris leur traduction en codage binaire que j'ai ajouter à une liste d'int bit par bit, tout coller, on a alors notre texte compressé sous forme de bits, plus légère en nombre de bits que le texte original.

   Ensuite, on rajoute des bits si la taille de la liste de bits n'est pas un multiple de 8, car il va ensuite falloir écrire dans le fichier sous forme d'octet de 8 bits.
   Ces bits ajouté doivent être supprimé lors de la décompression, donc on écrit leur nombre dans les bits d'information (voir plus bas). Et c'est lors de l'écriture dans notre tableau array (tableau contenant le contenu du fichier compressé) qu'on converti cette liste de bits tout collé en octet de int8_t.

   On peut voir ça dans la fonction `text_to_list_of_bits`.

   (b) Le dictionnaire :
   En ce qui concerne le dictionnaire, je passe directement par un vector<int8_t>, l'écriture se passe dans la fonction `dict_in_binary_form`.
   Pour chaque codage, j'écris le caractère, puis la taille du codage de ce caractère en bit, et enfin son codage, auquel j'ajoute des bits pour remplir mon octet. La taille du codage est essentielle pour plusieurs raisons :

- dans le cas où un codage est inférieur à 1 octet, on doit pouvoir retirer les octets suplémentaires ajouter pour compléter à 1 octet lors de la décompression, donc connaître la taille permet de savoir combien de bit il faut retirer pour avoir notre codage
- s'il fait plus d'un octet, alors on doit savoir combien d'octet à la suite on va prendre, et également comme précédemment, combien de bits il faut enlever
- cela permet également de savoir où est-ce que commence le prochain codage

Ces deux fonctions `dict_in_binary_form` et `text_to_list_of_bits` sont appelés dans `write_compressed_file`.

2. Les parties du fichier compressé :

   Avant d'être écrit dans le fichier, le contenu du fichier est enregistré dans un tableau array comme dit précédemment. Ce tableau est remplit dans la fonction `write_array`, qui est appelé par notre fonction `write_compressed_file` :

```
void write_array(int8_t *array, int bit_added,
                 vector<int8_t> &my_dict_in_binary,
                 list<int> &text_to_write_in_list_of_bits) {

  // on ajoute les bits d'info dans l'array
  add_info_bits_in_array(array, bit_added, my_dict_in_binary.size());

  // on écrit le dict dans l'array
  write_dict_in_array(my_dict_in_binary, array);

  // on écrit le text dans l'array
  write_list_of_bits_of_text_in_array(text_to_write_in_list_of_bits, array,
                                      index_end_of_dict(array));
}
```

   On voit très clairement que notre fichier compressé est découpé en 3 parties :

- les bits (3 octets) d'informations au début de fichier, permettant de délimiter la partie dictionnaire et la partie texte compressé
  - le premier octet représente le nombre d'octet ajouter dans le texte compressé, à retirer lors de la décompression
  - les deux autres octets permettent de délimiter la taille du dictionnaire, c'est détaillé dans la fonction `add_info_bits_in_array`.
    * la meilleure méthode aurait été d'écrire la taille du dictionnaire sur 16 bits, puis diviser ça en deux octets, mais ça allait apporter de la complexité à mon code donc je suis resté sur une méthode simple et efficace. On peut aller jusqu'à un dictionnaire de plus de 10000 octets, ce qui est largement suffisant.
    * j'aurais également pu créer une taille de dictionnaire dynamique, non limité à 2 octets, à la manière de ce que j'ai fais pour les codages de caractère, mais même soucis : ça aurait apporter de la complexité inutilement, car un dictionnaire est souvent très court
- le dictionnaire, permettant de lire le texte compressé
- le texte compressé

3.1.2 Transformée de Burrows-Wheeler

J'ai écrit des fonctions pour la transformée de Burrows-Wheeler. J'ai eu beaucoup de problème avec cette partie là, les fonctions sont présentent dans le `namespace burrow_wheeler`.

J'ai principalement des problèmes vis à vis de la fonction `recontruct_original_string` qui est censé retrouvé le string original à partir du codage produit par la transformée comme on peut le voir sur la page Wikipédia. Elle fonctionne, mais à cause du grand nombre d'opérations de tri dans l'ordre alphabétique, et du grand nombre de `push_fronti`, ou plutôt insert(0, char) dans les string, alors que les string ne sont pas fait pour ça, c'est extrêmement lent.

J'ai essayer de contourné une partie du problème, en inversant la chose : faire des `push_back`, ce qui améliore la vitesse, mais cela nécessite donc d'inversé aussi le trie, donc trier dans l'ordre alphabétique en partant de la fin des string, et pour ça j'ai dû créer mon propre comparateur pour ma fonction de trie `sorting_in_alphabetical_order_from_end_v2`.
Pourquoi ? Car il n'y a pas de comparateur dans la bibliothèque standard

qui permet de trier dans l'ordre alphabétique à l'envers, c'est-à-dire en partant depuis la fin des string vers le début plutôt que du début vers la fin.
Cela fonctionnait, mais c'était encore plus lent que la méthode originale, à cause du besoin de création de string et d'utilisation de `push_back` dans la fonction comparatrice qu'utilise `sorting_in_alphabetical_order_from_end_v2`, et qui est énormément sollicité.

J'ai donc décidé de retirer la partie Burrows-Wheeler de mon code. De toutes façon, il faut savoir que même si j'aurais réussi à ce niveau, il n'y aurait pas eu d'amélioration de compression car il aurait fallut utiliser un algorithme MTF.

Je l'ai laisser dans le doute mais n'hésitez pas à effacer si cela gêne votre visibilité, c'est inutile dans le code, et je n'ai pas "laver" (en divisant en sous-fonctions) cette partie de code donc c'est assez lourd.

3.2 Décompression

La décompression se passe dans la fonction `decompression`. Néanmoins, c'est une très petite fonction servant "d'interface", car la majorité du code permettant de décompresser est présent dans sa fonction `retrieve_content_from_compressed_file`, `decompression` s'occupe seulement d'écrire notre fichier décompressé.

La fonction `retrieve_content_from_compressed_file` récupère le contenu du fichier compresser et le met dans un `vector<int8_t>`, et à partir de ce buffer, récupère le contenu du dictionnaire avec `retrieve_dict`, puis appelle la fonction `retrieve_text` qui prend la partie texte compressé dans le buffer, et la traduit à l'aide du dictionnaire récupérer. Enfin, elle renvoie un string avec tout le contenu du fichier décompressé à écrire.

4 Conclusion

J'ai beaucoup aimer ce projet même si il a été assez difficile.
J'ai enfin réussi à créer une compression qui fonctionne : le fichier compressé est plus léger que l'original. J'avais déjà essayer le semestre dernier, en groupe, sans succès.
Je suis plutôt content du résultat.