TP 14 - Code de Huffman

5 juillet 2023

L'objectif de ce TP est d'implémenter en C l'algorithme de Huffman pour compresser et décompresser du texte. Il se découpe de deux parties : la première concerne les arbres préfixes et la seconde concerne l'algorithme de Huffman lui-même.

1 Arbres préfixes

L'objectif de cette partie est d'implémenter en C des arbres préfixes. Dans le cas présent, on compte se servir de ces arbres préfixes pour implémenter l'algorithme de Huffman, et on souhaite donc que l'arbre contienne les informations suivantes :

- Il s'agit d'un arbre binaire qui décrit le code associé à chaque caractère du texte. Par convention, on choisit que le fils gauche corresponde à l'ajout d'un `'0'` au code et le fils droit un `'1'`.
- Chaque feuille contient donc le caractère encodé par le code lu quand on descend jusqu'à cette feuille depuis la racine.
- Chaque feuille contient aussi un poids qui est le nombre d'occurrences du caractère (ou sa fréquence) dans le texte.
- Chaque nœud interne stocke un poids qui est la somme des poids des feuilles dans ses descendants.

En conséquence, vous vous appuierez sur la structure suivante pour manipuler les arbres préfixes :

```c
struct tree_s
  {
    int weight;
    struct tree_s *left_child;
    struct tree_s *right_child;
    bool is_leaf;
    char leaf_label;
  };
typedef struct tree_s tree_t;
```

Dans cette structure on retrouve tous les éléments attendus pour l'arbre préfixe détaillé plus haut. Le champ `is_leaf` permet de distinguer clairement entre les feuilles et les nœuds internes. Le champ `leaf_label` correspond au caractère présent aux feuilles. Dans un nœud interne, ce caractère peut valoir n'importe quoi et ne doit pas être utilisé.

1. Implémentez une fonction `tree_t* init_tree(char leaf_label, int weight)` qui crée un arbre réduit à une feuille pour un caractère donné en argument. Vous prendrez soin d'initialiser tous les champs, y compris les pointeurs vers les enfants avec une valeur adaptée. Cette fonction permet de créer les arbres associés à chaque caractère comme dans l'algorithme de Huffman.
2. Implémentez une fonction `tree_t* merge_trees(tree_t* left_child, tree_t* right_child)` qui crée un nouvel arbre en fusionnant deux arbres, comme décrit dans l'algorithme de Huffman. Vous prendrez soin de spécifier la valeur adaptée pour chaque des champs de l'arbre créé.
3. Implémentez une fonction d'affichage de ces arbres. Il n'est pas demandé un affichage avec une forme d'arbre, simplement de quoi vérifier ce que contient un arbre. Vous choisirez néanmoins judicieusement la manière dont vous affichez les informations.
4. Implémentez une fonction `free_tree(tree_t* t)` qui libère la mémoire réservée par un arbre. Vous ferez attention à ce que toute la mémoire réservée et uniquement la mémoire réservée soit libérée.
5. Testez les fonctions précédentes. Pour tester la libération correcte vous vous souviendrez des options de compilation adaptées.
6. Implémentez une fonction `int tree_height(tree_t* t)` qui calcule la hauteur d'un arbre.
7. Implémentez une fonction `int tree_size(tree_t* t)` qui calcule le nombre de nœuds dans un arbre.
8. Testez ces fonctions.

2 (Bonus mais à lire) File de priorité

Pour implémenter l'algorithme de Huffman, une file de priorité est nécessaire. Celle-ci va contenir les arbres définis plus haut. Une implémentation des files de priorités est possible à l'aide de tas. C'est ce qui est proposé de faire dans cette section. Cette section est un bonus alors qu'une file de priorité est nécessaire pour la suite. Pour cette raison, une implémentation de file de priorité (mutable) pour les arbres est fournie avec le sujet. Si vous souhaitez réaliser ce bonus, ne lisez pas cette implémentation mais seulement son interface `heap.h`. Si vous ne comptez pas réaliser ce bonus, vous n'avez besoin que de l'interface pour la suite, mais il pourrait être intéressant de savoir comment vous auriez pu réaliser cette implémentation. L'implémentation fournie utilise la fonction `assert` qui est obtenue via `#include <assert.h>`. Cette fonction permet de vérifier une propriété : on lui donne une expression booléenne en argument, si cette expression est vraie, il ne passe rien, sinon le programme est interrompu.

9. (Bonus) Implémentez une file de priorité min contenant des arbres dont le type est décrit dans `tree.h` à l'aide de tas. Cette implémentation devrait fournir les fonctions dont la signature est décrite dans le fichier d'interface `heap.h`. Vous pourrez bien entendu vous appuyer de fonction auxiliaire dont il n'est pas nécessaire qu'elles apparaissant dans le fichier d'interface.

3 Algorithme de Huffman

3.1 Préliminaire

En raison du nombre important de fonctions qui vont interagir ici et de leur groupement par objet, il est particulièrement adapté de séparé son code en plusieurs fichiers. Le premier fichier (ou module) concerne l'implémentation des arbres. Le second sera pour l'implémentation de la file de priorité et le troisième pour l'algorithme de Huffman proprement dit. Vous aurez donc l'organisation suivante :

— L'implémentation des arbres sera réalisée dans `tree.c` avec l'interface associée `tree.h`.

— L'implémentation de file de priorité sur les arbres sera dans `heap.c` avec l'interface `heap.h`.

— L'implémentation de l'algorithme de Huffman sera dans `huffman.c` et `huffman.h`.

Pour l'organisation du code entre `.c` et `.h` vous pourrez revenir aux TP précédents.

Ici on peut remarquer le point suivant : `huffman` a besoin de `tree` et `heap`, `heap` a besoin de `tree`. Lors de l'inclusion des dépendances, `heap` va donc inclure `tree` puis `huffman` va intégrer `tree` et `heap` (et donc `tree` une deuxième fois). Ceci va donc poser des problèmes de double définition pour les éléments de `tree`.

Pour prévenir ces problèmes, vous allez implémenter une *condition de garde* pour l'inclusion de manière à rendre celle-ci *idempotente* c'est-a-dire que son inclusion n'aura lieu que la première fois.

Ceci est réalisé exclusivement dans le fichier d'interface (`.h`) et utilise les directives préprocesseur `#define`, `#ifndef` `#endif`. L'idée est la suivante : lors de la première inclusion d'un fichier interface, le processus de compilation va stocker l'information selon laquelle cette inclusion a été réalisée à l'aide de la directive `#define NOM_DE_VARIABLE`. Avant d'inclure le fichier, il testera l'existence de `NOM_DE_VARIABLE` et pourra donc savoir si l'inclusion a déjà été réalisée via la directive `#ifndef NOM_DE_VARIABLE`. Ceci permet de ne tenir compte de ce qui vient après que dans le cas où l'inclusion n'avait pas été réalisée auparavant. Son effet s'arrête avec la directive `#endif`.

Pour rendre l'inclusion idempotente, il suffit d'encadrer le contenu du fichier d'interface par :

```
#ifndef INTERFACE_H
#define INTERFACE_H
//contenu du .h
#endif
```

`INTERFACE_H` est un nombre de variable qui doit être adapté et unique à chaque fichier d'interface. Une convention est d'utiliser le nom du fichier d'interface en capitales suivi d'un `_H`. Par exemple dans le fichier d'interface des files de priorité on peut voir :

```
#ifndef HEAP_H
#define HEAP_H
//...
#endif
```

10. Organisez le code implémenté jusque-là de la manière indiquée plus haut.
11. (Bonus) Écrivez un fichier `Makefile` pour organiser la compilation séparée de tous ces fichiers.
12. Écrivez un programme séparé ne contenant qu'une fonction `main` et incluant les modules `huffman`, `heap` et `tree` et déplacez les tests des fonctions de ces modules dans la fonction `main`. Les sources (`.c`) des modules `huffman`, `tree` et `heap` ne devront pas contenir de fonction `main`.

3.2 Algorithme de Huffman

13. Implémentez une fonction `int* count_occurences(char* text)` qui renvoie un tableau contenant pour chaque caractère le nombre d'occurrences de ce caractère dans le texte : la case `i` du tableau contiendra la nombre d'occurrences du caractère numéro `i` (dans l'ordre ASCII).
14. Implémentez une fonction `heap_t* build_heap_from_text(char* text)` qui renvoie une file de priorité contenant les arbres feuilles tels que décrite par l'algorithme de Huffman. Dans cette question vous pourrez, temporairement, ne pas vous préoccuper des fuites mémoire.
15. Implémentez une fonction `tree_t* build_code_from_heap(heap_t* heap)` qui calcule l'arbre correspondant au code de Huffman à partir de la file de priorité initialisée à la question précédente. Dans cette question vous pourrez à nouveau vous préoccuper de libérer correctement la mémoire réservée.
16. Implémentez une fonction `char** code_from_tree(tree_t* huffman_tree)` qui renvoie un tableau de chaînes de caractères contenant les codes associés à chaque caractère. La case `i` contiendra une chaîne de caractère égale au code associé au caractère numéro `i` dans l'ordre ASCII. (indication) Vous pourrez pour ceci passer par une fonction récursive qui remplit un tableau à l'aide d'un arbre et d'un code courant contenant le chemin depuis la racine de l'arbre de Huffman à l'arbre courant.
17. Implémentez une fonction `char* encode(char* text, tree_t* huffman_tree)` qui encode/compresse un texte à l'aide de son arbre de Huffman. Vous prendrez soin de dimensionner correctement la chaîne de caractère contenant le résultat.
18. Implémentez une fonction `char* decode(char* compressed_text, tree_t* huffman_tree)` qui décode un texte compressé à l'aide de son arbre de Huffman. (indication) Vous pourrez implémenter pour ceci une fonction récursive qui lit le caractère suivant en descendant dans l'arbre.
19. Testez ces fonctions sur un texte de votre choix et vérifiez que la compression puis la décompression est bien sans pertes.

3.3 Pour aller plus loin

Maintenant qu'on a implémenté le cœur de l'algorithme, il reste une question importante à traiter : pourrait-on vraiment se servir de cet algorithme pour compresser des textes ? Pour répondre à cette question, cette partie propose d'implémenter les étapes suivantes à savoir :

— Compresser un texte directement écrit dans un fichier
— Écrire dans un fichier les informations concernant le code utilisé pour compresser le texte
— Lire ce code dans un fichier
— Décoder directement un fichier
— Comparer la place occupée par les versions compressées et non compressées.

20. Implémentez une fonction `void write_tree(tree_t* t, char* filename)` qui écrit un arbre dans un fichier. (indication) Vous pourrez vous appuyer sur une fonction récursive auxiliaire pour réaliser ceci. Vous prendrez soin que cette fonction inscrive dans le fichier de manière à pouvoir reconstruire d'arbre par la suite. Cette fonction fera partie du module `tree`.
21. Implémentez une fonction `tree_t* read_tree(char* filename)` qui reconstruit un arbre à partir de sa version écrite dans un fichier. Vous pourrez vous servir des propriétés de caractérisation des parcours d'arbres. Cette fonction fera partie du module `tree`.
22. Implémentez une fonction `char* read_file_as_string(char* filename)` qui lit le contenu d'un fichier et renvoie ce contenu sous la forme d'une chaîne de caractère. Vous prendrez soin de bien dimensionner la taille de la chaîne dans laquelle stocker la valeur de retour. Cette fonction fera partie d'un nouveau module `io`.
23. Implémentez une fonction `void write_content_to_file(char* filename, char* content)` qui écrit une chaîne de caractère dans un fichier. Cette fonction fera partie du module `io`.
24. À l'aide des fonctions précédentes, implémentez une fonction `void encode_from_file(char* src_filename, char* dst_filename, char* tree_filename);` qui lit un texte, le compresse et écrit ce texte compressé dans un fichier dont le nom est `dst_filename` et écrit l'arbre du code dans un fichier séparé. On choisira d'écrire les bits du code sous forme de `char` (`'0'` ou `'1'`) que directement sous forme binaire. Cette fonction fera partie du module `huffman`.
25. Faites de même pour une fonction `void decode_from_file(char* src_filename, char* dst_filename, char* tree_filename)` qui lit un texte compressé dans le fichier `src_filename` et écrit une version décompressée de ce texte dans un fichier `dst_filename` en ayant lu l'arbre pour le décoder dans le fichier `tree_filename`.
26. Testez vos fonctions sur un petit texte.
27. Testez vos fonctions sur le texte `long_text.txt` fourni avec le code.
28. Que dire des poids du fichier obtenu après compression par rapport au poids pré-compression ? Est-ce attendu ? Comment pourrait-on estimer le taux de compression réel ? Dans quelle mesure la description de l'arbre contenant le code vient modifier ce taux de compression ?