Projet d'implémentation d'un logiciel de compression

Contacts
paul.viallard@univ-st-etienne.fr emilie.morvant@univ-st-etienne.fr
Salon discord "Projet" de l'UE Programmation Impérative 2

1 - Avant-Propos

Le projet doit être réalisé en **binôme**, les binômes doivent nous êtes communiqués sur discord dans le salon "projet" de l'UE programmation impérative 2 le **vendredi 22 avril** au plus tard.

Vous devez implémenter dans le langage C l'application décrite ci-dessous, puis le déposer sur claroline **avant le lundi 16 mai 2022 à 8h00** dans le rendu dédié (les soutenances auront lieu **mercredi 18 mai**). Le dépôt doit prendre la forme d'une archive `tar.gz` et doit UNIQUEMENT contenir :

- l'ensemble du code source (fichiers sources, Makefile)
- une notice README expliquant comment créer l'exécutable, principal, l'exécutable de la génération de fichiers des tests et comment générer les fichiers de tests.

Remarques importantes

- N'oubliez pas de commenter votre code et de faire un rendu "propre" !!
- De nombreuses ressources existent sur le codage de Huffman, à vous de faire votre propre recherche si vous êtes bloqués.
- **Le code que vous allez rendre doit être votre propre code, nous rappelons que le plagiat/la copie de code d'une source extérieure impliquera la saisie de la section disciplinaire, nous ne ferons preuve d'aucune intransigeance.**
- Votre code source doit compiler silencieusement sur les machines de l'université (mira2).

2 - Organisation du projet

2.1 - Objectif du projet

L'objectif du projet est d'implémenter un petit logiciel (utilisable uniquement en ligne de commande) de compression/décompression de fichiers textes avec l'algorithme du codage de Huffman (`https://en.wikipedia.org/wiki/Huffman_coding`) sans perte d'information.

2.2 - Évaluation

La notation sera faite au terme d'une présentation de votre travail (avec ou sans diapo, un vidéo-projecteur sera à votre disposition si besoin). Le temps de la présentation est de 7 minutes (démos comprises). Vous devez donc être concis et préparer vos démos au préalable afin de limiter le temps de manipulation.

2.3 - Organisation du projet

L'ensemble du projet devra **obligatoirement** comporter :

- Toutes les versions de votre logiciel en commençant par `huff_v0.c`
- Un fichier `versions.txt` dans lequel vous écrirez la description de chacune des versions de votre logiciel que vous avez écrit

- Un fichier `Makefile` qui permet de compiler votre version finale via la commande `make`. Le Makefile doit également permet de compiler toutes les versions de votre logiciel.

Nous vous conseillons de respecter les propositions de prototypages des fonctions faites dans l'énoncé afin que le débugage soit facilité.

3 - Principe du codage de Huffman

Une idée apparue très tôt en informatique pour compresser les données a été exploitée indépendamment dans les années 1950 par Shannon et Fano. Elle se base sur la remarque suivante : les caractères d'un fichier sont codés en octet, donc tous sur le même nombre de bits. Il serait plus économique en terme d'espace disque, pour un fichier donné, de coder ses caractères sur un nombre variable de bits, en utilisant moins de bits pour les caractères fréquents et plus de bits pour les caractères rares. Le codage choisi dépend donc du fichier à compresser. Les propriétés d'un tel codage sont :

1. les caractères sont codés sur un nombre différent de bits,
2. les codes des caractères fréquents sont courts, ceux des caractères rares sont longs,
3. bien que les codes soient de longueur variable, on peut décoder le fichier compressé de façon unique,
4. le point 3 est assuré si la propriété suivante est vérifiée : si `c1` et `c2` codent deux caractères différents, `c1` ne commence pas par `c2` et `c2` ne commence pas par `c1`. En effet si le point 4 est assuré, lorsqu'on décode le fichier compressé en le lisant linéairement, dès que l'on reconnaît le code d'un caractère, on sait que l'on ne pourra pas le complété par un autre code.

L'algorithme de Huffman garantit les 4 points ci-dessus, il fonctionne comme décrit dans la suite.

3.1 - Calcul des occurrences des caractères

Prenons l'exemple d'un fichier texte nommé `exemple.txt` et dont le contenu est :

```
Une banane
```

On calcule le nombre d'occurrence de chaque caractère dans le fichier à compresser. Dans `exemple.txt`, on calcule donc le nombre d'occurrence des caractères : `'U'`, `'n'`, `'e'`, `'␣'`, `'b'`, et `'a'` .

`'␣'`	`'U'`	`'b'`	`'n'`	`'a'`	`'e'`
(1)	**(1)**	**(1)**	**(3)**	**(2)**	**(2)**

Figure 1: Fréquence des caractères pour `exemple.txt`.

3.2 - Création de l'arbre de Huffman

On prend ensuite les caractères qui possèdent la plus faible occurrence et on ajoute leur nombre d'occurrence pour réaliser un nouveau nœud.

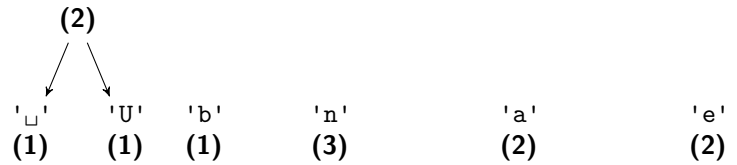

Figure 2: Création d'un premier nœud pour `exemple.txt`

La procédure est re-itérée en considérant le nouveau nœud (non plus les caractères `␣` et `U`. Dans `exemple.txt` à partir de la situation de la Figure 2, l'occurrence la plus faible est celle de `b` et celle du nœud créé (remarque : on aurait pu prendre `a` ou `e` puisque leur valeur est 2 également).

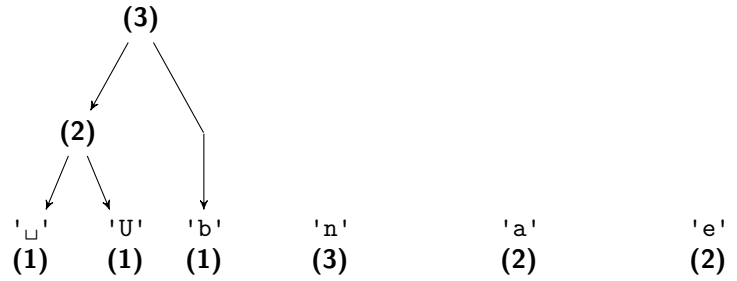

Figure 3: Création d'un deuxième nœud pour `exemple.txt`.

A la fin de la procédure, l'arbre de Huffman créé pour `exemple.txt` est le suivant:

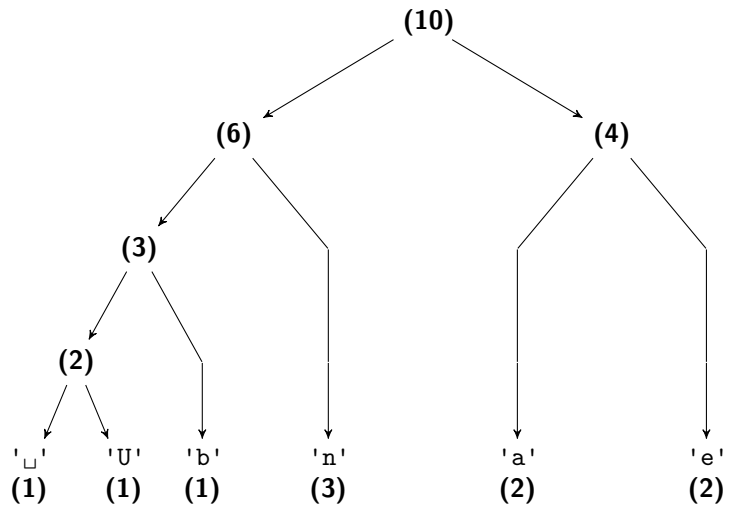

Figure 4: Création complète de l'arbre de Huffman pour `exemple.txt`.

Cependant, avec d'autres choix de nœuds (lors des égalités d'occurrence), il est possible d'obtenir des arbres différents. Cependant l'algorithme fonctionne avec tout choix respectant la construction ci-dessus.

Ensuite, le codage est construite à partir de l'arbre en lisant le chemin qui va de la racine au caractère. Un pas à gauche est lu comme la valeur `0` et un pas à droite comme la valeur `1`.

Le code binaire de chaque caractère à partir de l'arbre de la Figure 5 est le suivant :

Caractère	U	n	e	␣	b	a
Fréquence	1	3	2	1	1	2
Code	0001	01	11	0000	001	10

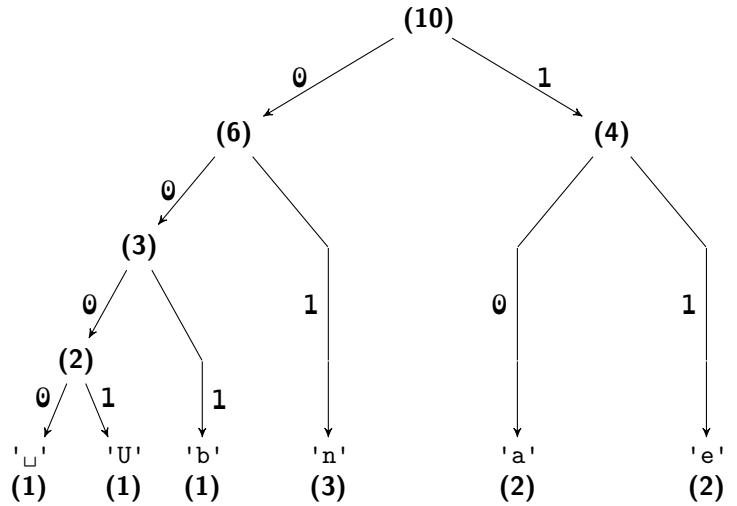

Figure 5: L'arbre de Huffman pour `exemple.txt` avec le codage des caractères.

On obtient donc la suite de bits `000101110000001100110011 1` pour coder le contenu de `exemple.txt`.

Ce codage vérifie les 4 propriétés mentionnées plus tôt, ce qui permet la compression et la décompression. Avec cet algorithme, le décompresseur doit connaître les codes construits pendant la compression. Lors de la compression, le compresseur doit écrire les codes en début de fichier compressé sous un format à définir, connu du compresseur et du décompresseur. Le fichier compressé aura donc deux parties disjointes :

- La première partie doit permettre au décompresseur de retrouver le code de chaque caractère.
- La seconde partie contient la suite des codes des caractères du fichier à compresser.

Attention, le décompresseur doit toujours pouvoir trouver la séparation entre ces deux parties. De plus, vous devez toujours vérifier que le code généré pour coder un caractère ne correspond pas au début du code d'un autre caractère. A noter également que le format doit être bien défini pour ne pas avoir un fichier compressé plus lourd que le fichier d'origine...

4 - Le travail à réaliser

4.1 - Première partie : Occurrences des caractères

L'objectif est de travailler avec des fichiers (qui par la suite seront les fichiers à compresser).

QUESTION 1 : Écrire un programme qui ouvre un fichier texte et qui affiche une partie du contenu à l'écran. Vous devez utiliser la fonction `fread()`.

QUESTION 2 : Écrire un programme qui ouvre un fichier texte et qui affiche la totalité du contenu à l'écran. Vous ferez une lecture grâce à la fonction `fgetc()` jusqu'à ce que vous rencontriez le caractère de fin de fichier.

Nous nous intéressons au comptage des occurrences des caractères présents dans le fichier texte. Pour cela, vous utiliserez un tableau de caractères. Ce tableau comporte 256 éléments. Chaque case de ce tableau doit contenir le nombre d'occurrence du caractère dont le code ASCII est donné par l'index du tableau.

QUESTION 3 : Définir une fonction `void occurence(FILE *fic, int tab[256])` qui comptera les occurrences des caractères du fichier `fic` et stockera les occurrences dans le tableau `tab` passé en paramètre. Pensez à faire des affichages pour vérifier le bon fonctionnement de votre fonction.

4.2 - Deuxième partie : Création de l'arbre de Huffman

Afin de manipuler les caractères et les nœuds de l'arbre, vous devez définir une structure de données. Cette structure permettra de créer l'arbre et de créer le code. Vous appellerez cette structure `noeud` (elle doit également être utilisée pour les feuilles de l'arbre).

La structure `noeud` doit contenir 6 champs : le caractère initial, l'occurrence dans le fichier, le codage dans l'arbre, le nombre de bits sur lesquels est codé le caractère, les pointeurs vers les nœuds suivants.

QUESTION 4 : Écrire une boucle dans votre programme pour créer dynamiquement une structure `noeud` pour chaque caractère contenu dans le fichier. Vous testerez votre programme en affichant le contenu des champs contenant le caractère initial et son occurrence. De plus, afin de conserver l'ensemble des structures créés, vous sauvegarderez les pointeurs vers ces structures `noeud` dans le tableau `noeud * arbre_huffman[256]`

QUESTION 5 : Réalisez exactement la même chose mais cette fois-ci en faisant appel à une fonction dont le prototype est `noeud * creer_feuille(int *tab, int index)`.

QUESTION 6 : En dehors du contexte de l'algorithme de compression, définir une fonction pour cherche dans un tableau les deux entiers les plus petits.

Une fois votre fonction validée, utilisez cette fonction pour parcourir votre tableau de Huffman `noeud arbre_huffman[256]` et trouver les deux nœuds associés aux plus petites valeurs d'occurrence. À partir de ces deux nœuds, vous devez créer un nouveau nœud dont la valeur d'occurrence vaut la somme des deux nœuds d'origine.

QUESTION 7 : Définir la fonction `void creer_noeud(nœud *tab[], int taille)` qui doit rechercher les plus petites occurrences, créer le nouveau nœud et le sauvegarder dans le même tableau de Huffman.

À la fin de la création de l'arbre, vous devez obtenir un seul pointeur qui doit correspondre à la racine de l'arbre binaire et qui pointe vers les deux nœuds suivants (qui eux même possèdent deux références vers les nœuds suivants, etc).

4.3 - Troisième partie : Création du code

En parcourant l'arbre binaire depuis la racine de l'arbre, nous allons créer le codage des caractères.

Le parcours de l'arbre de la racine à une feuille doit se faire de manière récursive. L'algorithme est le suivant :

```
Si est_feuille(noeud_courant)
    alors
        stockage du code et de sa taille
Sinon
    appel récursif à droite en injectant un 0 dans le codage
    appel récursif à droite en injectant un 1 dans le codage
```

QUESTION 8 : Définir une fonction récursive `void creer_code(noeud * element, int code, int profondeur)` qui réalise le parcours de l'arbre en calculant le codage. Dans votre fonction, lorsque vous êtes dans une feuille, faites appel à une fonction `void affichage_code(int nbr_bits, int codage)` pour afficher le code du caractères sous forme binaire.

Tout est prêt pour effectuer la compression. Pour cela vous devez modifier la fonction `void creer_code()` pour sauvegarder les pointeurs vers les structures `nœud` des caractères existants dans le fichier. Ceci sera très similaire au tableau `tab_caracteres[256]`. Vous devez conserver le pointeur sur la structure de chacun des caractères du fichier dans un tableau : `noeud * alphabet[256]`.

4.4 - Quatrième partie : Compression

Le fichier compressé doit posséder 2 parties :

- Une partie ENTÊTE contenant le nombre de feuilles (de caractères) différentes de votre fichier et la structure "alphabet" correspondant à chacun de ces caractères *(dans l'entête vous pouvez également spécifier le nom d'origine du fichier)*.
- Une partie CONTENU contenant le code compressé des caractères du fichier.

Remarque importante : Vous allez devoir remplir votre fichier bit par bit (ou paquet de bits par paquet de bits, par forcément multiple de 8). INDICATION : vous pouvez passer par les opérateurs sur les bits. Lors de votre présentation, vous devrez OBLIGATOIREMENT nous expliquer la solution mise en œuvre pour remplir le fichier.

QUESTION 9 : Définir un format d'entête et définir la fonction permettant d'écrire cette entête dans le fichier compressé. Grâce à cette entête, la décompression pourra prendre connaissance des caractères présents dans le fichier et pour chacun d'entre eux connaître son code.

NB : lors de vos tests, pensez à supprimer la structure `alphabet` avant la lecture de l'entête (puisque la lecture de l'entête doit permettre de recréer ce tableau sans en avoir la connaissance au préalable.

QUESTION 10 : Définir la fonction qui écrit l'ensemble des codes correspondant à chaque caractère du fichier d'origine les uns à la suite des autres.

4.5 - Cinquième partie : Décompression

L'objectif maintenant est de réaliser l'opération inverse. À partir d'un fichier compressé avec votre programme, vous devez reconstruire le fichier d'origine. Pour cela, votre programme doit, d'une part, recréer le tableau `noeud * alphabet[256]` et, d'autre part, recréer le tableau de Huffman pour décoder les bits du fichier compressé.

QUESTION 11 : Vous devez implémentez la création *inverse* de l'arbre de Huffman depuis le tableau `alphabet`.

QUESTION 12 : Implémentez la reconstruction du fichier d'origine

4.6 - Sixième partie : Étude du taux de compression

L'objectif de cette partie est d'étudier le comportement de la compression avec le codage de Huffman et, en particulier, le taux de compression. Il est conseillé d'avoir la première version du logiciel, comme décrit dans la section suivante, avant de vous lancer dans cette étude.

QUESTION 13 : Étudiez le taux de compression sur différents fichiers textes de tests. Par exemple des fichiers quelconque de différentes tailles, des fichiers créés aléatoirement, des fichiers n'utilisant qu'un sous ensemble de caractères (pensez aux cas extrêmes comme les cas où les fichiers ne sont composés que de 2 caractères différents), etc. Quelles sont vos conclusions ? Vous devrez les présenter le jour de la soutenance.

Pour cette partie, vous pouvez vous aidez de différentes ressources sur internet afin d'identifier les cas particuliers.

5 - Finalisation du logiciel

À partir du travail effectué ci-dessus sur l'algorithme de Huffman, vous devez créer votre logiciel.

Votre logiciel doit se lancer en ligne de commande. Par défaut,

```
>huffman -c archive_finale dossiers_ou_fichiers_a_compresser

>huffman -d archive_a_decompresser

>huffman -d archive_a_decompresser dossier_cible
```

Vous devez également implémenter les options suivantes :

- `-c` : compression
- `-d` : décompression (si un chemin `dossier_cible` est spécifié alors la décompression doit se faire dans `dossier_cible`, sinon elle doit s'effectuer dans le dossier courant)
- `-h` : doit afficher une page d'aide à l'utilisation de votre programme (comme une page du `man`)

Pensez à traiter le cas de noms de fichiers/dossiers déjà existants.

Procédez étape par étape, de manière incrémentale (n'oubliez pas que votre rendu doit contenir toutes les versions de votre logiciel).

1. La première version de votre logiciel ne doit traiter que la compression d'un seul fichier texte, c'est-à-dire que `dossiers_ou_fichiers_a_compresser` doit être un fichier texte.
2. la deuxième version de votre logiciel doit traiter également le cas où vous souhaitez compresser plusieurs fichiers (en une seule archive), c'est-à-dire que `dossiers_ou_fichiers_a_compresser` peut être une liste de fichiers texte. Lors de la décompression, vous devez reconstruire tous les fichiers texte d'origine dans le répertoire courant.
3. la troisième version de votre logiciel doit traiter le cas où les fichiers textes à compresser sont situé dans un dossier (aucun sous-dossier à l'intérieur), c'est-à-dire que `dossiers_ou_fichiers_a_compresser` peut être un dossier qui doit uniquement contenir des fichiers texte. Lors de la décompression, le dossier doit être créé et les fichiers doivent être décompressés dans ce dossier.
4. la quatrième version de votre logiciel doit traiter le cas où le dossier passé en argument peut être une arborescence (uniquement des répertoires et des fichiers textes), c'est-à-dire que `dossiers_ou_fichiers_a_compresser` peut être un dossier qui doit uniquement contenir des fichiers texte et des dossiers contenant eux-même des fichiers texte. Lors de la décompression vous devez re-créer l'arborescence.
5. la cinquième version de votre logiciel doit pouvoir traiter le cas où `dossiers_ou_fichiers_a_compresser` peut être une liste de dossiers et/ou fichiers (plusieurs arborescences et fichiers texte). Lors de la décompression, vous devez re-créer les fichiers et les arborescences.

NB : Une fois toutes les versions réalisées, vous pouvez, en guise d'extension, ajouter une version graphique de votre logiciel.