NGUY Thomas **2A SLE**

HACKETT Damien **Phelma/Ensimag**

TP APMA :

Compression de fichiers textes par codage Huffman

Introduction :

L'objectif de ce TP est de réaliser un programme de compression/décompression de fichiers textes en se basant sur l'algorithme du codage de Huffman.

Le principe est de remplacer le codage ASCII des caractères dans un fichier texte, occupant toujours une place mémoire constante (1 octet), par un codage de longueur variable, dépendant de la fréquence d'apparition des caractères. Les caractères apparaissant le plus souvent se verront attribué un codage court tandis que les caractères qui apparaissent le moins souvent un codage long. Il est ainsi possible de réduire l'espace mémoire occupé par le fichier texte de cette façon.

Le programme de compression/décompression repose sur plusieurs packages.

Package « File à Priorité »

Une file à priorité sera utilisée afin de construire l'arbre de Huffman, la priorité étant ici la fréquence d'apparition du caractère.

Nous avons choisis d'implémenter la file à priorité dans un tableau. Cette structure nous permet de réaliser toute les opérations en O(1). De plus étant donné que le nombre de caractères représentable par le codage ASCII est fini (256), la file de taille « finie » n' a pas une grosse empreinte mémoire et est donc très avantageuse.

Chaque élément de la file est un « Nœud » qui contient *une donnée* (un arbre) et *une priorité* (fréquence d'occurrence). Nous ferons particulièrement attention à l'ordre de la priorité, plus la valeur celle-ci sera faible et moins l'élément sera prioritaire dans la file.

Package « Arbre Huffman »

Nous utiliserons un arbre d'Huffman afin de générer le codage des caractères du fichier texte.

Un nœud de l'arbre contient une valeur (un caractère), un fils gauche (pointeur sur arbre) et un fils droit (pointeur sur arbre).
Un nœud interne est caractérisé par l'existence un fils gauche ou fils droit non nuls (leur valeur n'a pas d'importance) tandis qu'une feuille possède un des caractères que l'on veut coder et ne dispose d'aucun fils.

Package « Code»

Une classe Code sera utilisée afin de représenter le code binaire des caractères. Un code binaire est une suite de 0 et de 1, on peut donc l'assimiler à une liste d'éléments disposant chacun d'une valeur (Bit) et d'un pointeur vers l'élément suivant.

La lecture du code se fera à l'aide d'un itérateur afin de s'assurer de ne pas « altérer » le code lors d'une lecture.
L'itérateur est implémenté via un troisième pointeur sur le bit courant de la liste Code. On vérifie qu'il existe un élément suivant grâce à la fonction has_next et on récupère sa valeur le cas échéant avec next.

Classe « Dictionnaire»

Cette classe permet de générer le dictionnaire qui réalise la correspondance entre un caractère et son code.

La structure employée pour implémentation du dictionnaire est un tableau de 256 éléments. Les informations d'un caractère de code ASCII « i » sont stockées dans l'élément du tableau d'indice « i ». Cette structure simple et efficace nous permet d'accéder rapidement au code d'un caractère lu (O(1)) qui est l'opération la plus utilisée dans le dictionnaire. De plus la contrainte d'une taille fixe nous importe peu étant donné que la table ASCII des caractères est finie et de faible taille.

Chaque élément du tableau est une structure (Caracteres_Info) contenant le codage du caractère (Code) et la fréquence d'occurrence du caractère (Integer). De plus, nous stockerons dans le dictionnaire, en plus du tableau, l'arbre d'Huffman du fichier texte généré afin d'éviter de devoir le reconstruire à lorsque l'on en aura besoin.

Package « TP_Huffman »

C'est le package contenant le programme principal et contient donc aussi les procédures Compresse et Decompresse permettant de compresser ou décompresser un fichier.

Compression

Pour compresser un fichier, après l'avoir parcouru pour remplir le dictionnaire (cout en O(n)), on génère l'arbre correspondant ainsi que les codes de chaque caractères.

Puis, on reparcourt le texte original, caractère par caractère. Pour chaque caractère trouvé, on parcoure son code , bit par bit.

Comme on ne peut manipuler au minimum que des octets (Character), on doit concaténer les bits dans un char pour pouvoir les écrire. On ajoute la valeur du bit (0 ou 1) à un entier « valeurAecrire » puis on multiplie cet entier par deux.

Cela équivaut donc à écrire notre bit sur une « bande » (notre valeurAecrire) et a décaler (x2) cette bande vers la gauche afin de pouvoir écrire le caractère suivant et ainsi de suite. Lorsque notre bande contient 8 bits et a donc la taille d'un octet, on écrit le caractère correspondant dans le fichier cible.

On aura ainsi plusieurs caractères stockés dans un seul octet, ce qui peut faire gagner une place mémoire très importante.

Pour la décompression :

Pour pouvoir décrypter le texte compressé et ainsi le décompresser, il faut connaître la correspondance Code <=> Caractère. On aura donc besoin :

- de la table des correspondances : elle donne le code et le nombre d'occurrences d'un caractère. Elle permet de regénérer facilement l'arbre de décryptage. Elle sera placée au début du fichier, juste avant le code « compressé » du fichier source.

- du nombre de clés pressentes dans le fichier : cela nous permet de connaître la taille du tableau et ainsi trouvé où commence réellement le code dans le fichier compressé. On placera cette information dans le premier entier du fichier cible.

- du nombre de caractères du texte original: ce qui permet de savoir où s'arrête le code pertinent d'un fichier. Il sera calculé grâce à la fonction Nb_Total_Caractères de la classe Dico, lorsque l'on aura besoin de l'information.

  N.B : Une optimisation pourrait être de stocker ce nombre directement dans le fichier afin d'éviter son calcul lors de la décompression.

Décompression

Lors de la décompression, on récupère donc la table de correspondance et on génère l'arbre de Hoffman. Ensuite, on calcule le nombre de caractères présents dans le fichier original et on commence à lire le code crypté bit par bit.

Pour cela, on pratique la méthode inverse de celle utilisée lors de la compression.

Le bit lu est celui en bout de bande, on applique donc une division entière par 2^(longueur de la bande-1). Ainsi, on commence bien à lire le bit de poids fort (#7) du Character. Ensuite, on « coupe » la partie déjà lue en divisant effectivement la valeur par 2. On réutilise ainsi cette valeur jusqu'à épuisement de la « bande ». On créera alors une autre bande avec la valeur du prochain caractère du fichier compressé.

Les bits récoltés nous permettent de nous déplacer dans l'arbre de Huffman de manière analogue à l'étape de compression. Si l'on reçoit un 1, on se déplace vers le fils droit du nœud actuel, vers la gauche sinon. Lorsque l'on tombe sur une feuille, on écrit le caractère qui y est stocké dans le fichier « décompressé ».Et on repart à la racine de l'arbre afin de décoder un nouveau caractère.

On parcourt et décode ainsi tout le fichier compressé jusqu'à posséder autant de caractères dans le fichier « décompressé » que l'on en avait dans l'original. On retrouve alors un fichier identique à notre fichier original.

Conclusion :

Ce TP nous a permis de conjugué la pratique du langage de haut niveau ADA, à la conception d' algorithme de compression/décompression et à l'utilisation optimale des structures de données (file de priorités, listes de bits...). Cela nous donne donc un bon aperçu de ce que l'on demande à un bon programmeur : une bonne réflexion sur les algorithmes associée à une implémentation efficace de celui avec l'aide du langage choisi.

Nous vous laissons maintenant le soin d'examiner/exécuter notre programme.

Nous vous conseillons d'utiliser le Makefile et de lire le README associé pour lancer la compilation et les tests automatiques.