**Rapport du projet Huffman**

Concevoir un algorithme de compression

Table des matières

[Présentation 3](#_Toc58045037)

[Conception 4](#_Toc58045038)

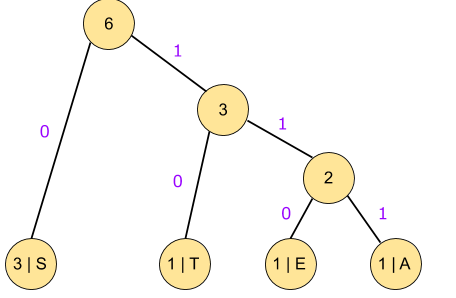
[Conclusion 11](#_Toc58045039)

[Quelques algorithmes 12](#_Toc58045040)

# Présentation

L’objectif de ce projet est de réaliser un algorithme de compression, c’est-à-dire que l’on cherche à réduire la place que prend une information sans perte de données. Par conséquent, nous avons réalisé ce projet sur des fichiers textes en utilisant le codage de Huffman, reposant sur la traduction en un code court d’un caractère en fonction de sa fréquence d’apparition. Plus concrètement parlant, pour un texte de 100 caractères prenant 800 bits de mémoire, nous allons donc chercher à réduire la mémoire qu’il prend grâce au code de Huffman.

Ce projet va nous apprendre à manipuler la table ASCII des caractères et de nous inciter à appliquer ce que nous avons appris lors de nos TD.

Le but du code de Huffman est de pouvoir traduire un code court d’un caractère en fonction de sa fréquence d’apparition. Pour cela nous devons d’abord calculer l’occurrence d’un caractère et de les classer de façon décroissante (de celui qui apparait le plus à celui qui apparaît le moins). Avec cette donnée, il faudra créer l’arbre d’Huffman, son but va être de remplir un arbre en fonction de son occurrence, par exemple, si la lettre ‘’n’’ apparaît le plus de fois, il se mettra à gauche de l’arbre et prendra pour valeur 0, si la lettre ‘’a’’ est en 2e dans le nombre d’occurrence, il se déplacera à droite et prendra la valeur, grâce à cela, on pourra encoder les lettres de façon optimisée.  
Exemple :  
  
Après l’encodage, il faudra décoder l’arbre à l’aide de la table ASCII de chacun des caractères.   
Le but de ce projet est de pouvoir compresser un fichier texte en prenant le moins de place possible.

# Conception

Pour une meilleure compréhension et développement du code, on a choisi de décomposer ce dernier en sous fonctionnalités.

I) Le traitement des caractères

Le principal objectif pour compresser un fichier est de connaitre l’occurrence de chaque lettre afin de leur attribuer le meilleur code possible. En effet, une lettre qui apparait souvent va être codé sur le moins de bits possible et inversement pour une lettre qui apparait beaucoup moins souvent.

A. Partie naïve

On récupère les caractères du fichier à compresser et on les ajoute à une liste chainée qui va stocker tous les caractères ainsi que leur occurrence. Si le caractère n’existe pas dans la liste, on l’ajoute, sinon on incrémente juste son occurrence. La liste n’est pas triée, l’ordre n’est pas important pour l’instant, ainsi lors de l’ajout des caractères, on ajoute le maillon à la fin de la liste.

B. Partie optimisée

Cette fois, l’ordre des caractères stocké est important. En effet, on perd du temp à chercher si un caractère existe dans une liste non-triée. Dans le cas d’une liste trié, la recherche est plus simple car on sait ou on doit aller. La méthode utilisée est la recherche dichotomique : on prend le milieu de la liste et on compare le caractère au caractère que l’on veut insérer, s’il est avant lui dans l’ordre alphabétique (c’est à dire qu’il a une code ascii moins élevé) on va directement dans la partie gauche de la liste en créant un nouvel intervalle beaucoup plus petit (la moitié). On fait le contraire dans le cas où le caractère est après le caractère au milieu de la liste dans l’ordre alphabétique. Si l’on trouve le caractère dans la liste on incrémente juste ses occurrences, sinon on ajoute le caractère à la position spécifique où la recherche s’est arrêtée (c’est à dire l’endroit où l’intervalle est devenue nul).

On trie ensuite la liste en fonction des occurrences grâce à un algorithme de tri à bulle. En effet, trier la liste en fonction des caractères permettait juste de simplifier la détection d’un caractère déjà existant à la liste. Or ce qui nous intéresse réellement est son nombre de répétitions dans le fichier texte.

C. Conversion du fichier texte en fichier binaire

Bien sûr, pour pouvoir compresser un fichier il faut manipuler leur bit et non pas les caractères aux mêmes. C’est pour cela qu’on a créé une fonction qui convertie directement un fichier texte en fichier composé de bit correspondant à chaque caractère.

Pour cela, on extrait chaque caractère du fichier texte, on prend sa valeur entière ASCII en décimal qu’on converti après en binaire dans une liste. On écrit cette dernière dans le fichier qui va contenir la traduction de chaque caractère en binaire. On aura au final la traduction du fichier texte en binaire.

On a aussi ajouté une fonction qui compte le nombre de caractères dans un fichier pour permettre de vérifier si la compression a marché (ex: il faut qu’il y ait exactement le même nombre de caractères dans le fichier d’entré binaire que le fichier décompressé (en binaire aussi).

II) L’arbre d’Huffman

C’est la partie au cœur de la compression, c’est ce qui va permettre d’associer chaque caractère à un code optimale (en fonction de ses occurrences)

A. Partie naïve

L’arbre d’Huffman est composé de nœuds, ainsi nous devons manipuler des nœuds et non plus des maillons de liste chainée. C’est pour cela qu’on a décidé de convertir la liste d’occurrence précédente en liste de nœuds.

L’arbre d’Huffman est créé de la façon suivante : on prend les deux minimums de la liste (minimum associé au nombre d’occurrence), on les supprime de la liste et on les relie à un nœud qu’on créé préalablement avec comme occurrence l’addition de l’occurrence des deux feuilles (les deux minimums trouvés précédemment). On ajoute ensuite ce nœud à notre liste de nœud qui sera utile dans la création de l’arbre par la suite. On fait cela jusqu’à ce que notre liste de nœud soit nulle, c’est-à-dire qu’on a plus de caractère à ajouter à l’arbre. On constate donc que les premiers caractères qu’on a ajouté à l’arbre étant les premiers minimums de la liste de nœuds vont être les caractères qui apparaissent le moins dans le texte, leur profondeur sera plus élevée et on aura donc un code plus grand. En revanche, les derniers caractères qu’on va ajouter à notre arbre (les caractères qui apparaissent le plus dans le texte) seront plus proche de la racine, par conséquent on aura un code moins long.

En effet, on code chaque caractère (étant positionnés à chaque feuille de l’arbre) de la façon suivante : en recherchant le caractère dans l’arbre, lorsqu’on va à gauche on associe la valeur binaire 1, lorsqu’on va à droite on associe la valeur binaire 0. La longueur du code va donc être égale à la profondeur de la feuille associée au caractère que l’on veut coder.

Dans la partie naïve la liste d’occurrence n’est pas triée ainsi on perd du temps à rechercher le minimum de la liste de nœuds. La partie optimisée va arranger ce problème.

B. Partie optimisée

La création de l’arbre ne diffère pas grandement de la partie naïve à la seule exception que la liste d’occurrence est déjà triée en fonction des occurrences. On va donc utiliser une file puisque le premier élément de la liste (étant le minimum) va être le premier élément qui va sortir de la liste. Néanmoins les nœuds qu’on va créer et qui vont être ajoutés à la liste de nœuds peuvent altérer l’ordre des maillons. C’est pour cela que l’on va utiliser deux files, une pour stocker les feuilles de l’arbre et donc les caractères que l’on souhaite coder, et une autre qui stocke les nœuds qui vont relier ces feuilles. En effet, les premiers nœuds qu’on va créer seront forcément les nœuds avec les valeurs d’occurrence les plus faibles vu qu’on ajoute à l’arbre en premier les caractères qui ont les occurrences les plus faibles. Les premiers éléments qu’on va ajouter à notre file de nœuds seront forcément les premiers éléments à sortir de la file dans la recherche de minimum, d’où l’utilisation d’une file.

Ainsi, la recherche du minimum va être simplifiée puisqu’on va juste comparer les premiers éléments de chaque file. Le code sera adapté au cas où une des deux files est nulle.

Le dernier nœud qui sera créé sera le nœud avec l’occurrence la plus élevée, il sera donc forcément la référence de l’arbre. Pour récupérer l’arbre d’Huffman on retourne donc le dernier nœud créé.

III) Le dictionnaire

L’arbre d’Huffman sert de support à la compression, il faut maintenant déterminer à partir de ce dernier le code de chaque lettre. On parcourt donc l’arbre comme dit précédemment dans la partie de la création de l’arbre d’Huffman et on déduit donc le code de chaque caractère.

Pour parcourir chaque lettre, il faut parcourir l’arbre de façon intelligente. C’est-à-dire qu’il ne faut pas qu’on aille sur une feuille qui a déjà été visitée, en effet, lorsqu’on vient de coder une lettre il ne faut plus repasser par celle-ci. Ainsi, on a adopté la stratégie suivante : lorsqu’on est sur une feuille (donc un caractère à coder), après avoir enregistré son code dans le dictionnaire, on la supprime de l’arbre et on vérifie si le nœud précédent est maintenant considéré comme une feuille (pointe à droite et à gauche vers NULL) car ce dernier n’est pas une lettre à coder. Ainsi, si c’est le cas, on doit le supprimer également car il est associé à aucune autre feuille et donc aucune autre lettre à coder. On revérifie par la suite également après suppression du nœud si les nœuds précédents seront aussi considérés comme des feuilles, si c’est le cas on doit également les supprimer. Pour cela, on utilise une pile qui va stocker uniquement les nœuds et non pas les feuilles, lorsqu’on détecte qu’un nœud devient une feuille on dépile la pile puisque le dernier nœud qu’on va ajouter à notre pile sera forcément le premier nœud qu’on a souhaité supprimer de notre arbre.

Par cette méthode nous parcourons l’arbre intelligemment de telle sorte à ce qu’on code les bons caractères, ceux qui sont présents dans le texte. En effet, la structure utilisée pour créer l’arbre aura deux informations : le caractère et son occurrence, le caractère qu’on va associer aux nœuds qui ne seront pas des feuilles sera aléatoire et ne devra pas être pris en compte dans l’encodage car ils ne font pas partie du texte.

A. Partie naïve

Le fonctionnement étant presque similaire à la partie optimisée, nous avons préféré faire une explication globale du code précédemment et se focaliser par la suite sur les petites parties du code qui vont être modifiées. Dans cette partie le code de chaque lettre va être stocké dans un fichier dictionnaire. Lors de la décompression on devra donc chercher chaque caractère dans ce fichier dictionnaire, on constate donc que la recherche n’est pas très optimisée et pourra prendre du temps pour un texte qui est très long.

B. Partie optimisée

Pour pallier ce problème de recherche, nous ajoutons le caractère à coder ainsi que son code dans un AVL qui va stocker les nœuds en fonction des caractères. Nous constatons que cette méthode ressemble à la recherche dichotomique utilisée pour obtenir notre liste d’occurrence, on va pouvoir lors de la décompression parcourir l’arbre intelligemment et donc trouver le caractère beaucoup plus facilement que si on devait parcourir un fichier dictionnaire intégralement sans savoir où on va.

IV) Encodage

Cette partie, plus importante, permet de convertir le fichier binaire traduisant le texte que l’on souhaite encoder en un autre fichier binaire qui va contenir le code de chaque caractère du fichier texte que l’on a préalablement déterminé dans l’élaboration du dictionnaire. Les caractères ne seront plus codés sous 8 bits mais sur un nombre de bit qui va différer en fonction des occurrences de chaque caractère du fichier texte. En encodant le fichier texte a partir du dictionnaire qu’on a créé, on va obtenir un fichier binaire qui, comme dans la partie 2 (conversion), va contenir une suite binaire dont la traduction ne dépend plus de la table ASCII mais du dictionnaire.

1. Partie naïve

Dans cette partie, l’encodage va prendre un certain temps. En effet, pour chaque caractère du fichier texte on va rechercher son code dans le dictionnaire, or la recherche n’est pas optimisée car on ne sait pas où se trouve exactement le caractère recherché : on peut très bien devoir parcourir tout le dictionnaire comme parcourir 2 lignes. On constate que pour un long texte, l’encodage peut prendre un certain temps (sur une bonne machine un texte de 2000 lignes prend 25s à être encoder.

1. Partie optimisée

Pour résoudre ce problème d’optimisation de recherche, au lieu de stocker le code de chacun des caractères dans un dictionnaire, on va stocker le caractère et son code associé dans un AVL qui sera trié en fonction du code ASCII de chaque caractère (ordre alphabétique). En effet, la recherche d’un caractère sera plus simple vu que l’on sait où l’on doit chercher, contrairement à un dictionnaire où nous ne savons pas sur quelle ligne se trouve le caractère recherché.  
Ici on passe d’un encodage qui prend 25s pour un texte de 2000 lignes à un encodage qui ne prend que 1ms. On a alors une optimisation de l’ordre de 2500%. Ce qui n’est pas négligeable.

V) Compression

Cette fonction ne diffère pas de la partie naïve et de la partie optimisée puisqu’on regroupe seulement les fonctions que l’on a créé précédemment, seul le contenu des sous-fonctions de celle-ci va différer. Mais globalement, on ne fait qu’appeler le même type de sous-fonction (la liste d’occurrence, l’arbre d’Huffman, le dictionnaire et l’encodage). Ayant déjà décrit précédemment la description de ces sous-fonctions, il n’est pas nécessaire d’en expliquer plus sur cette partie.

VI) Décompression

Cette partie était en option pour la partie naïve mais nous avons quand même décidé de la faire pour pouvoir vérifier si la compression marche. Pour pouvoir décompresser le fichier, nous avons besoin de l’arbre d’Huffman. En effet, pour chaque caractère du fichier, nous parcourons l’arbre d’une manière précise (pour un 0 nous allons à gauche et pour un 1 nous allons à droite), on s’arrête lorsque nous tombons sur une feuille, c’est-à-dire le caractère correspondant au chemin suivi par l’arbre : son code déduit précédemment dans la partie dictionnaire. Cette fois-ci nous ne créons pas l’arbre d’Huffman mais nous le parcourons à partir de la même logique ce qui nous permet de savoir précisément dans le fichier binaire où s’arrêter pour chaque caractère puisque ces derniers ne sont plus codés sur un nombre de bits constant (8 bits pour un fichier binaire qui traduit un caractère à partir de la table ASCII : cf. partie Conversion). Lorsque l’on arrive sur le caractère correspondant, on écrit dans un autre fichier qui sera le fichier décompressé.

1. Partie naïve

Par soucis de simplification nous devions décompresser le fichier à l’intérieur du fichier de compression puisqu’on devait avoir l’arbre d’Huffman en mémoire. Par la suite nous décompressons le fichier comme la méthode décrite ci-dessus.

1. Partie optimisée

Pour éviter comme dans la partie précédente de devoir décompresser le fichier à partir de la fonction compression (on est censé pouvoir décompresser le fichier en dehors de la fonction décompression, comme lorsqu’on reçoit un fichier compressé (.zip ou .rar par exemple)), nous devons alors recréer l’arbre Huffman à partir du fichier dictionnaire. En effet, même si le dictionnaire de cette partie est un AVL, nous avons quand même créé un fichier dictionnaire pour permettre cette recréation de l’arbre d’Huffman.

Pour ce faire nous prenons chaque caractère ainsi que son code, si ce dernier contient un 0, nous créons une branche qui pointe vers la gauche et inversement pour un 1. Si on est sur un ‘\0’ (stocké sur une chaîne de caractères) cela veut dire qu’on est à la fin du code et qu’on est ainsi être censé sur une feuille de l’arbre d’Huffman et donc sur une lettre à décompresser. Ainsi dans ce cas précis on crée un nœud contenant le caractère que l’on a extrait précédemment du dictionnaire. Ses occurrences importent peu puisqu’on souhaite uniquement décompresser le fichier.

Après avoir recréé entièrement l’arbre d’Huffman, nous décompressons le fichier de compression comme la méthode exprimée dans l’introduction de cette partie.

Difficultés rencontrées

Fuites de mémoire :

Les fuites de mémoire, problème récurrent dans les programmes et souvent oubliées, peuvent créer des difficultés d’exécution et ainsi ralentir le programme, voire le faire planter.   
Nous avons utilisé Visual Studio pour analyser la gestion de la mémoire au cours du programme. Pour cela, nous avons positionné deux breakpoints dans le main, un au début et un à la fin, et nous avons comparé la différence de mémoire allouée entre ces deux étapes. Après l’ajout de fonctions de libération et quelques optimisations, nous passons de 1097 éléments alloués en fin de programme à 73 après désallocation de mémoire. Le temps indiqué à gauche n’est pas représentatif car des changements ont été effectués entre-temps, augmentant le temps d’exécution.







Etat de la mémoire à l’origine, puis après une première puis une deuxième modification.

Warnings :

Toujours grâce à Visual Studio, nous avons encore plus optimisé le programme en résolvant le maximum de warnings possibles.

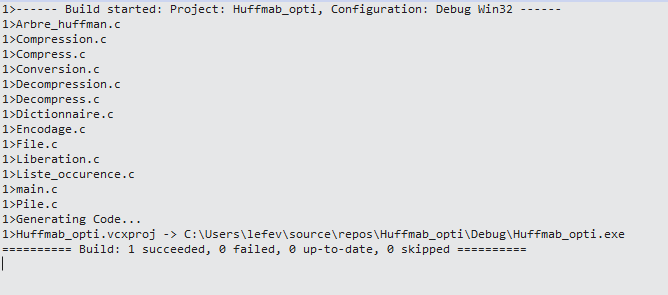
Pour ce faire, nous avons utilisé l’outil Build Solution qui nous a permis d’identifier un certain nombre de warnings, comme représentés ci-dessous :







Après résolution de ces divers warnings, nous arrivons finalement à cette solution :



Autres :

Le principal problème rencontré pour la compression a été le parcours de l’arbre d’Huffman pour pouvoir encoder chaque lettre. En effet, nous devions le parcourir intelligemment sans repasser par des feuilles déjà parcourues. Après avoir mûrement réfléchi à ce problème, nous avons décidé qu’à chaque parcours, nous allions supprimer la feuille qui vient d’être encodée pour éviter de la reparcourir. La logique de parcours a été explicitée dans la partie dictionnaire du rapport. De plus, pour la suppression des nœuds et non pas des feuilles, l’utilisation d’une pile a été décidée tardivement : nous utilisions les listes chaînées auparavant. Nous avons ainsi constaté que cette dernière pouvait remplir les conditions d’une pile.

Nous avions aussi rencontré des problèmes concernant la fonction de tri par dichotomie ; en effet nous avions du mal à comprendre la logique, surtout avec des listes chaînées (l’accès à un maillon avec un indice n’est pas aussi simple qu’avec un tableau classique). La fonction d’origine faisait plus de 100 lignes, nous l’avons raccourcie à à peine 50 lignes.

Enfin, pour la partie dictionnaire dans la partie optimisée, nous devions créer le dictionnaire sous forme de fichier et d’arbre. Nous avions créé ainsi deux fonctions différentes, une qui crée l’arbre et une qui crée le fichier et nous avons constaté que ces deux fonctions se ressemblaient fortement : pour factoriser le code et le rendre plus performant, nous avons décidé de rassembles ces deux fonctions en une seule.

# Conclusion

Ce projet nous a permis d’appliquer tous les enseignements de TD d’Algorithmique et de programmation en C et de comprendre le fonctionnement de tout cela dans un cas concret (ici la création programme permettant la compression d’un fichier). Le niveau de programmation de certaines personnes a grandement évolué grâce à ce projet et de nombreuses compétences acquises aujourd’hui seront très utiles pour le monde professionnel.

Aussi, ce projet a beaucoup de potentiel et pourrait être distribué via une plateforme de partage comme GitHub, il peut aussi être modifié pour permettre la compression d’autres fichiers comme le ferait un WinRAR que nous connaissons tous. Le projet a aussi été fait avec des connaissances de L2 et à l’avenir, il pourrait être peaufiné mais pourrait aussi être programmé sous un autre langage.

En conclusion, ce projet a apporté beaucoup de choses au groupe, il a aussi amélioré le niveau de compréhension des listes chainées, des piles, des files et des arbres binaires qui sont des choses essentielles en C. Cette fusion avec l’Introduction au Génie Logiciel a aussi été bénéfique pour nous, grâce à cette matière, nous sommes capables de programmer de façon propre tout en respectant le lexique du C. Cela nous sera d’autant plus utile pour notre future vie professionnelle. Ainsi, avec ce nouveau bagage acquis lors de ce projet, nous sommes parés pour le prochain projet de programmation.

# Quelques algorithmes

