Rapport du projet C++ : Partie 1

Gabriel Dos Santos, Raphael Marouani

15 décembre 2020

Table des matières

1 Introduction

Le but principal de ce projet est de développer un système de codage et de décodage basé sur le code de Huffmann. Le codage d'Huffmann a pour but de coder un texte en binaire préfixé qui consiste à coder chaque lettre par un mot sur 0, 1 (toujours le même pour une lettre). Étant donné le pourcentage d'occurrence de chaque lettre dans le texte à coder, l'algorithme de codage des lettres est le suivant :

— Initialement, chaque lettre est un arbre binaire ramené à un sommet étiqueté par la proportion d'occurrences de cette lettre dans le texte. Tant qu'il y a plus d'un arbre, réaliser les opérations suivantes :

1. Considérer `A1` et `A2` les deux arbres dont les racines portent les plus petites étiquettes `e1` et `e2`.
2. Construire un nouvel arbre `A` dont la racine `r` a pour fils les racines de `A1` et `A2`.
3. La racine `r` est étiquetée par `e1 + e2`.

— Pour chaque noeud de l'arbre final, l'arête vers son fils gauche est étiquetée 0 et celle vers son fils droit, 1.

— Le code associé à une lettre est le mot binaire composé des étiquettes sur les arêtes entre la racine de l'arbre final et la feuille étiquetée avec cette lettre.

L'algorithme de Huffman étant avant tout un algorithme de compression de données, les mots "cryptage" et "compression", ainsi que tous leurs dérivés grammaticaux, pourront être utilisés de façon interchangeable dans la suite de ce rapport.

2 Utilisation de l'application

2.1 Contenu de l'archive du projet

L'archive que vous avez reçu contient les fichiers suivants :

- Le rapport du projet `compte_rendu.pdf`, que vous êtes en train de lire.
- Le fichier `README.md`, qui résume le projet et explique rapidement comment utiliser l'application.
- Le fichier `listing.pdf`, qui est la documentation complète du projet (partie 1 et 2).
- Les fichiers exécutables `build.sh` et `.check_dependencies.sh` pour vérifier la présence des dépendances nécessaires au projet et simplifier le processus de compilation avec CMake.
- Le fichier `CMakeLists.txt`, qui contient la configuration de l'outil CMake pour créer le Makefile qui compilera l'application.
- Le dossier `src`, qui contient les fichiers sources du projet. Celui-ci est divise en trois sous-dossiers :
  - `bin`, qui contient les fichiers `main1.cpp` et `main2.cpp`, les programmes de test.
  - `lib`, qui contient les fichiers (`*.cpp`) des classes `Sommet`, `ArbreB`, `Part1` et `AppWindow` ainsi que le fichier des fonctions nécessaires a la partie 2, `Part2`.
  - `headers`, qui contient les fichiers headers (`*.hpp`) pour chacun des fichiers du dossiers `lib`.

2.2 Configuration requise

Afin de pouvoir utiliser cette application, vous aurez besoin des programmes suivants :

- Le compilateur `g++` (version 7.5.0 ou supérieure). Alternativement, vous pouvez utiliser le compilateur `clang++` (testé avec la version 9.0.0 uniquement).
- Le standard C++ 17 (le minimum requis est le standard C++ 11 mais il est préférable de compiler en C++ 17).
- L'outil de compilation `cmake` (version 3.1 ou supérieure).
- L'outil pour la réalisation d'interface graphique Qt (version 5).

2.3 Compilation et exécution avec l'interface en ligne de commande

Ce projet utilisant l'outil CMake, le script `build.sh` vous est fourni avec l'archive du projet afin de simplifier les étapes de compilation. Ce script vous permettra également de vérifier si vous avez les paquets nécessaires et vous proposera d'installer les éventuelles dépendances manquantes.
Les commandes qui suivent sont à taper à la racine du projet.

Pour compiler le projet, il vous suffit de taper la commande suivante :

```
./build.sh
```

Il vous suffit de suivre les instructions affichées dans le terminal afin de compiler le projet.
Dans le cas ou le script `build.sh` n'aurait pas les droits pour être exécutable, utilisez la commande suivante avant de relancer le script :

```
chmod +x build.sh .check_dependencies.sh
```

Une fois le projet compilé, il vous suffit de l'exécuter en tapant :

```
./partie_2
```

3 Partie 1

3.1 Répartition des tâches

Nous avions initialement prévu de travailler de manière équitable sur l'ensemble de la partie 1 du projet. Le but était que chacun implémente une partie des classes et des fonctionnalités demandées dans l'énoncé, afin que chaque membre du binôme comprenne les choix d'implémentation et la façon dont est écrit le programme.

Devant le manque de réponses et de retours de Raphael (ceux-ci étant dûs à des problèmes familiaux), Gabriel a finalement réalisé l'intégralité de l'implémentation des classes `Sommet`, `ArbreB`, `PartOneTests` ainsi que le programme de test `main1.cpp`. La rédaction du Makefile, de la documentation, des commentaires dans le code source, du `README` et d'une partie de ce rapport ont également été réalisées par Gabriel. Raphael s'est chargé de l'interface graphique et a rédigé les parties qui y correspondent dans ce compte-rendu.

3.2 Description des classes

3.2.1 Classe Sommet

La classe `Sommet` représente un noeud dans un arbre binaire. Par choix d'implémentation et pour faciliter les parties suivantes du projet, la classe `Sommet` contient déjà des attributs pour stocker un caractère et sa fréquence. Chaque instance de cette classe contient également deux pointeurs, qui permettent une implémentation similaire à une liste chaînée. Les attributs de la classe sont les suivants :

- `char m_Data` : le caractère affecté au Sommet.
- `double m_Freq` : la fréquence du caractère affecté au Sommet. L'énoncé demandant que la fréquence soit représentée par un pourcentage, la fréquence est représentée par un flottant à double précision.
- `Sommet* m_Left` : le fils gauche affecté au Sommet. C'est un pointeur vers un autre Sommet.
- `Sommet* m_Right` : le fils droit affecté au Sommet. C'est également un pointeur vers un autre Sommet.

3.2.2 Classe ArbreB

La classe `ArbreB` représente un arbre binaire. Pour uniformiser les fonctions d'insertions, de recherches et de suppressions, on suppose que ces opérations sont faites sur un arbre binaire de recherche (étant donné qu'il nous est impossible d'implémenter certaines de ces opérations sur un arbre binaire de Huffman, et sans commencer la partie 2 du projet). Elle ne possède qu'un seul attribut, un pointeur sur la racine de cet arbre :

— `Sommet* m_Root` : la racine de l'arbre, depuis laquelle on peut accéder à tous les sommets qui le composent, grâce à l'implémentation en liste chaînée de la classe `Sommet`.

3.2.3 Classe PartOneTests

La classe `PartOneTests` ne sert qu'à implémenter des tests qui valident ou non les fonctionnalités des classes `Sommet` et `ArbreB`. Les méthodes qui y sont implémentées sont utilisées dans le programme de test afin de tester l'application de manière précise. Elle possède des attributs qui ne servent qu'à suivre l'état des tests :

— `static unsigned int total_tests` : le nombre total de tests qui ont été écris, ce qui permet de déterminer si des tests n'ont pas été exécutés (*skipped*).

— `unsigned int tests_run` : le nombre de tests qui ont été exécutés.

— `unsigned int tests_failed` : le nombre de tests qui ont échoué.

3.3 Interface en ligne de commande

Pour l'interface en ligne de commande, Gabriel a écrit un programme de test (`src/bin/main1.cpp`) qui utilise les méthodes de la classe de test `PartOneTests`. Le but était de décomposer chaque fonctionnalité des classes de façon à pouvoir les tester le plus précisément possible. Chaque test unitaire est appelé dans la fonction (`main`), puis évalué par une boucle (`if`). Le résultat du test est alors affiché dans le terminal. L'ordre dans lequel les tests sont exécutés est également logique, les tests plus avancés s'appuient sur des fonctionnalités précédemment validées. L'affichage dans le terminal est inspiré de bibliothèques de tests unitaires telle que JUnit pour le Java.

```
[root@gnix project]# make run
mkdir -p target/debug/
mkdir -p target/release/
g++ src/bin/main1.cpp src/lib/Sommet.cpp src/lib/ArbreB.cpp src/test/PartOneTests
.cpp -std=c++17 -Wall -Wextra -g -o target/debug/main1
./target/debug/main1

        Running tests for class Sommet...
Test for default constructor:           PASSED!
Test for parameterized constructor:     PASSED!
Test for copy constructor:              PASSED!
Test for setting values:                PASSED!
Test for overload of '=' operator:      PASSED!
Test that copies are not linked:        PASSED!

        Running tests for class ArbreB...
Test for default constructor:           PASSED!
Test for parameterized constructor:     PASSED!
Test for constructor from Sommet:       PASSED!
Test for copy constructor:              PASSED!
Test for overload of '=' operator:      PASSED!
Test that copies are not linked:        PASSED!
Test for inserting Sommet:              PASSED!
Test for inserting existing Sommet:     PASSED!
Test for finding a character:           PASSED!
Test for not finding a character:       PASSED!
Test for removing Sommet (leaf):        PASSED!
Test for removing Sommet (1 child):     PASSED!
Test for removing Sommet (2 children):  PASSED!
Test for fusing two ArbreBs:            PASSED!
Test for decomposing one ArbreB:        PASSED!
Test for finding a character (BFS):     PASSED!
Test for not finding a character (BFS): PASSED!

Tests run: 23,  Tests failed: 0,  Tests skipped: 0
Status: BUILD SUCCESS
[root@gnix project]# scrot -q 100 -u
```

FIGURE 1 – Voici ce que vous devriez obtenir dans le terminal en tapant la commande 'make run' à la racine du projet.

3.4 Interface graphique

Recherche de documentation sur Qt, vidéo Youtube, Openclassroom. J'ai débuté la création de l'interface graphique sur Geany en incluant les bibliothèques Qt.

La classe `MainWindow` représente la fenêtre principale qui va accueillir : -le Menu ; -Les layouts ; -Zone de scroll ; -Zone saisie ; -Message BOX ; -Affichage de L'Arbre ; La classe `MainMenu` est le menu afficher dans MainWindow contenant : -Boutton Arbre "servant à afficher l'arbre Binaire" -Boutton Postfixe "servant à afficher le chemin postfixe" en cours -Boutton Préfixe ; "servant à afficher le chemin préfixe" en cours -Boutton Infixe ; "servant à afficher le chemin infixe" en cours -Boutton Quitter ; -Affichage de l'Arbre ; en cours

Tentative de création d'un singleton, class `Context` malheureusement une erreur c'est affiché juste avant de vous rendre ce devoir que je n'ai su résoudre : `surchargement de la recette de la cible «main.o»", "ancienne recette ignorée pour la cible «main.o»"`

4 Partie 2

4.1 Ajouts et modifications sur la partie 1

Pour la réalisation de la partie 2, nous avons dû modifier certains aspects des classes écrites dans la partie 1 :

- La classe `PartOneTests` a été renommée en `Part1` pour harmoniser les noms de fichiers entre la partie 1 et la partie 2.
- Les attributs `m_Left` et `m_Right` de la classe `Sommet` ont été rendus privés, de même pour l'attribut `m_Root` de la classe `ArbreB`. Les accesseurs pour ces attributs ont également été implémentés.
- Les noms de méthodes des accesseurs et des mutateurs ont été modifiés pour être plus lisible dans le code.
- Les constructeurs par recopie des classes `Sommet` et `ArbreB` initialisent désormais leurs attributs qui sont des pointeurs a `nullptr` grâce à la liste d'initialisation. Ceci permet d'éviter des fuites de mémoire qui apparaissaient dans certains cas biens particuliers. Afin de prendre en compte ce changement, la surcharge de l'opérateur `"+"` et les destructeurs de chacune des classes ont également été modifiés.
- La méthode `print()` qui affiche l'arbre binaire dans le terminal a été complétée par Raphael (l'affichage final a été légèrement modifiée par Gabriel pour la partie 2).

4.2 Répartition du travail

Pour la partie 2 du projet, Raphael a corrigé son travail sur l'affichage qu'il avait commencé pour la partie en ligne de commande. Il a aussi essayé de poursuivre la réalisation de l'arbre dans l'interface graphique avec Qt. Gabriel a quant à lui écrit l'intégralité des fonctions et méthodes des fichiers `Part2.hpp`, `Part2.cpp` et la classe `AppWindow` (qui gère l'interface graphique avec Qt). Il a également écrit les scripts `build.sh` et `.check_dependencies.sh`, le fichier `CMakeLists.txt` ainsi que la documentation et le compte-rendu relatif à la partie 2 du projet.

4.3 Travail réalisé

4.3.1 Fonctions implémentées dans le fichier `Part2.cpp`

Afin de répondre à l'énoncé de la partie 2, Gabriel a implémenté les différentes fonctions qui permettent de récupérer un texte, calculer les occurrences de chaque caractères qui y sont présents et en construire l'arbre binaire qui y est associé en suivant l'algorithme de Huffman.

Dans le cas où l'utilisateur souhaiterait récupérer le texte à crypter depuis un fichier, il peut se servir de la fonction `parse_file_to_string()` qui prend en argument le nom d'un fichier et renvoie une chaîne de caractères `std::string` avec le contenu du fichier.

Afin de créer l'arbre de Huffman pour le texte à encrypter, nous avons choisi de d'abord construire un tableau dynamique `std::vector` d'ArbreB. Pour cela, on utilise la fonction `build_btree~vector()` qui prend en argument une chaîne de caractères `std::string` (à noter que ceci permet à l'utilisateur de donner directement une chaîne plutôt qu'un fichier texte). Chaque caractère présent dans la chaîne, ainsi que son occurrence, est représenté par un ArbreB dans le tableau. Cette fonction est relativement simple : on parcourt la chaîne et pour chaque caractère lu, on regarde dans le tableau s'il existe déjà un ArbreB pour celui-ci (a l'aide de la fonction `find()`). Si c'est le cas, on augmente l'occurrence de 1. Si ce n'est pas le cas, on construit un nouvel ArbreB avec le caractère lu et une occurrence de 1, puis on l'ajoute au tableau. Après avoir lu l'intégralité de la chaîne, on transforme simplement l'occurrence de chaque ArbreB du tableau en pourcentage afin de répondre a l'énoncé (cette étape n'étant pas nécessaire pour la suite, elle peut être omise pour des raisons de performance). Procéder de cette façon nous permet de simplifier l'étape suivante, qui est la construction de l'arbre de Huffman.

Nous construisons ensuite l'arbre de Huffman en suivant l'algorithme du même nom grâce à la fonction `build_huffman_tree()`. Tant que la taille du tableau construit à l'étape précédente est supérieure à 1 :

— On récupère dans le tableau les deux ArbreB ayant le plus petit pourcentage d'occurrence grâce a la fonction `find_lowest()`.

— On fusionne ces deux ArbreB ensemble grâce a la surcharge de l'opérateur "+" implémentée dans la partie 1.

— On ajoute au tableau le résultat de cette fusion.

La fonction `find_lowest()` prend en argument une référence du tableau et le parcourt. Elle garde en mémoire l'indice de l'ArbreB avec la plus petite occurrence, en crée une copie qui sera ensuite renvoyée à `build_huffman_tree()` et efface l'original du tableau. Ici, il est important de noter que c'est une référence du tableau qui doit être passée à `find_lowest()`. Sans cela, on supprimerait l'ArbreB sur une copie du tableau et on aurait ainsi une boucle infinie dans `build_huffman_tree()` qui causerait un "stack overflow". Pour des raisons d'optimisation des performances et afin d'éviter des copies inutiles, on fusionne directement le résultat des appels à `find_lowest()`. De même, on utilise la méthode `emplace_back()` plutôt que `push_back()` sur le `std::vector` afin de construire l'ArbreB résultant de la fusion directement dans le tableau plutôt que de devoir l'y copier.

La dernière étape est d'encrypter en binaire la chaîne de caractères de départ. Pour cela, on se sert de la fonction `compress_to_bin()` qui prend en argument la `std::string` de départ ainsi qu'une `std::map` (initialisée grâce une méthode de la classe ArbreB décrite dans la section suivante), et renvoie une `std::string` contenant la version cryptée de la chaîne en entrée.

Les fonctions `print_input()`, `print_output()` et `print_map()` sont de simples fonctions d'affichage dans le terminal que nous jugeons inutiles de détailler ici.

4.3.2 Méthodes implémentées dans la classe ArbreB

Pour faciliter le cryptage de la chaîne de caractères, Gabriel a implémenté la méthode publique `build_huffman_map()` qui renvoie une `std::map` contenant des caractères en clés en des chaînes en valeurs. Cette méthode initialise la map ainsi qu'une `std::string` vide, puis appelle récursivement la méthode privée `map_to_char_code()` qui effectue un parcours infixe de l'arbre de Huffman.

On ajoute un "0" à la chaîne de caractères à chaque fois que l'on prend une branche gauche dans l'arbre, et un "1" à chaque fois que l'on prend une branche droite. Lorsque l'on arrive sur une feuille de l'arbre, on ajoute a la map le caractère contenu dans la feuille et la chaîne courante qui nous donne l'encodage pour ce caractère. Enfin, à chaque fois que l'on remonte dans les appels récursifs, on effectue un `pop_back()` sur la `std::string` afin d'actualiser celle-ci selon notre position dans l'arbre.

4.3.3 Implémentation de la classe `AppWindow` pour l'interface graphique avec Qt

Étant donné que nous n'avions pas d'interface graphique fonctionnelle pour la partie 1, nous avons dû en réaliser une pour la partie 2. Gabriel a donc implémenté celle-ci avec Qt et la classe `AppWindow`.

Cette dernière hérite de la classe `QWidget`. Les différents éléments qui la compose sont :

- Un `QGridLayout` global qui regroupe les éléments suivants :
  - Un `QGridLayout` qui regroupe les éléments relatifs au menu. Ces derniers sont :
    - Un `QPushButton` pour compresser un texte.
    - Un `QPushButton` pour effacer les champs.
    - Un `QPushButton` pour quitter l'application.
  - Un `QGridLayout` qui regroupe les éléments relatifs au texte. Ces derniers sont :
    - Une `QTextEdit` dans laquelle l'utilisateur pourra taper le texte qu'il souhaite encrypter. Vous pouvez également directement y coller du texte plutôt que de le taper.
    - Une `QTextEdit` dans laquelle le texte crypte en binaire apparaîtra. L'utilisateur ne peut pas interagir avec le texte qui s'affichera dans cette "text box".

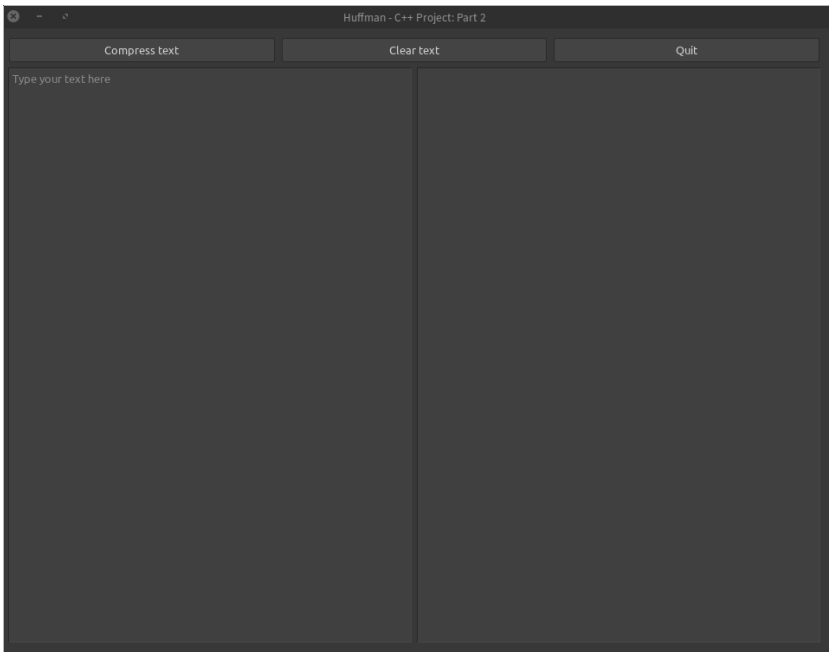

FIGURE 2 – Voici la fenêtre Qt qui vous sera affichée à l'exécution du programme.

La classe `AppWindow` implémente également deux méthodes qui sont appelées lorsque l'utilisateur clique sur un bouton :

- `run_compression()` lorsque l'utilisateur clique sur "Compress text". Cette fonction récupère le texte à compresser, crée le tableau d'ArbreB associé, construit l'arbre de Huffman pour le texte en entrée, remplie la map pour obtenir le code de chaque caractère, initialise la chaîne contenant la version cryptée du texte et enfin l'affiche a l'écran. L'arbre de Huffman ainsi que l'encodage de chaque caractère s'affiche également dans le terminal.
- `clear_text()` lorsque l'utilisateur clique sur "Clear text". Cette fonction efface le texte précédemment entré par l'utilisateur, ainsi que sa version binaire si l'utilisateur l'a déjà compresser. A noter que cette fonction n'est absolument pas nécessaire au bon fonctionnement de l'application et que l'utilisateur peut effacer manuellement le texte qu'il a écrit.

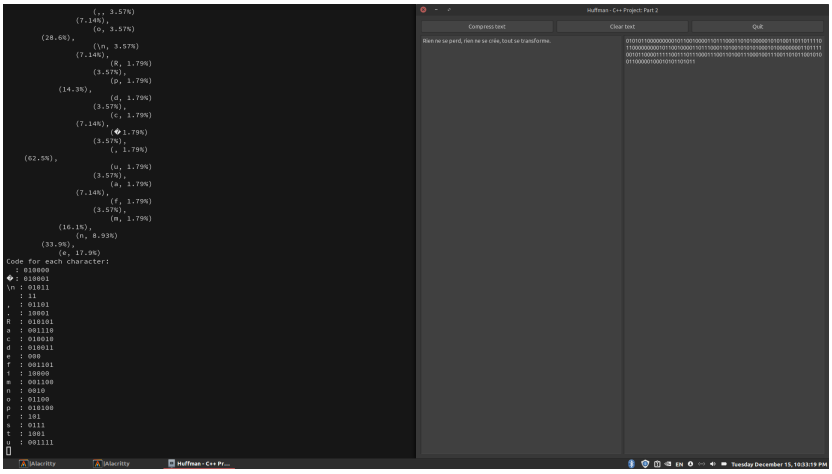

FIGURE 3 – Un exemple de compression d'une chaîne de caractères en binaire avec Qt et l'affichage de l'arbre et de l'encodage de chaque caractère dans le terminal.