Projet Partie 2 - Compte rendu

▼ **Binôme :** Samy BOUMALI & Amine ATEK

Dans cette partie 2 du projet, le cryptage du texte en entrée a été implémenté suivant l'algorithme de Huffman et à partir des classes `Sommet` et `ArbreB` codées lors de la première partie, augmentées des classes `Lecteur` et `Cryptage`.

Ce **compte rendu** présente dans un premier temps **le contenu de l'archive,** puis les **commandes d'utilisations et précisions,** et détaille ensuite le **déroulement complet de notre programme** en cette deuxième partie.

Contenu de l'archive

L'archive de notre projet contient :

- Un `Listing commenté` des fichiers du projet sous format PNG (à retrouver directement sur notre documentation Doxygen) ;
- Ce présent `Compte rendu PDF` détaillant l'ensemble du projet ;
- Le mécanisme de construction `Makefile` dont les commandes sont répertoriés après cette introduction ;
- Le fichier `Doxyfile` utile à la génération d'une documentation complète (listing compris) ;
- Le répertoire `app/` contenant les sources et headers de la partie 2 ;
- Le répertoire `qdarkstyle/` permettant la personnalisation de l'affichage graphique.

Commandes et utilisation

1. `make` pour exécuter la partie 2 (sous interface graphique) ;
2. `make listing` pour générer la documentation Doxygen et le listing des fichiers ;
3. `make cli` pour exécuter en mode CLI (affichage sur terminal sans faire intervenir les classes Qt) ;
4. `make debug` pour exécuter en mode debug (Valgrind et mem-check).

Précisions

1. La compilation graphique nécessite le programme `qmake`, outil de compilation pour les projet Qt.
2. La génération de la documentation nécessite le programme `Doxygen`. *Un répertoire `doc/` sera créé avec le listing en `html`. Le `Makefile` tentera par défaut d'ouvrir la documentation avec `firefox` mais cela pourrait ne pas fonctionner si vous n'avez pas le programme. Je vous recommande donc d'ouvrir le fichier `doc/html/files.html` avec votre navigateur habituel pour accéder au listing détaillé et commenté des fichiers.*

Lecture et calcul des occurrences de chaque lettre d'un texte

À partir d'un texte en entrée via un **fichier** ou une **chaîne de caractère,** la classe `Lecteur` se chargera de **calculer les occurrences** de chaque lettre dans un texte et pour un Alphabet donné.

La classe est constitué des **attributs privés** suivants :

- `std::string` Représente le texte en entrée sous forme de chaîne de caractère
- `std::vector<char>` Vector de char contenant l'ensemble des lettres d'un texte
- `std::vector<float>` Vector de flottants contenant les occurrences pour chaque lettre

L'algorithme de lecture stocke l'ensemble des lettres dans le vecteur de char de la classe, prenant en compte tous les caractères ASCII (sauf `espace` et certains signes de ponctuation) et compte dans les mêmes itérations son **nombre d'occurrences** dans le texte.

Nous avons fait le choix d'encoder les caractères accentués (ASCII étendus) bien que ces caractères ne soit pas supportés par l'affichage. Après avoir compté le nombre de caractères distincts, on calcule dans la boucle finale le pourcentage correspondant à chacune de ces occurrences pour les stocker dans le vecteur de flottants.

Construction de l'arbre binaire de Cryptage

Une fois la **map des lettres et occurrences** formée, le `Lecteur` envoie ses attributs à la classe `Cryptage` qui se charge dans un premier temps de **construire l'arbre binaire de Cryptage** en suivant l'algorithme de Huffman.

La fonction `construction_arbre()` représente cet algorithme et créera l'arbre binaire de cryptage à partir de la map des lettres et occurrences.

On créé initialement un arbre (`Sommet`) étiqueté par l'occurrence pour chacune des lettres en entrée, le vecteur d'arbres `arbres_restants` gère le nombre d'arbres restants lors de l'exécution de l'algorithme pour la condition d'arrêt de la boucle (tant qu'il reste plus d'un arbre dans le vecteur).

1. On tri le vecteur de sorte à considérer A1 et A2 les deux arbres portant les étiquettes e1 et e2 les plus faibles ;
2. Création de l'arbre A (Sommet `newSomm`) étiqueté e1 + e2 et attribution de ses fils A1 et A2.

La **manipulation de ces arbres** est évidemment réalisée à l'aide des fonctions et opérateurs définis lors de la partie 1 de ce projet, permettant la copie, l'égalité et la fusion d'arbres.

À la fin de l'exécution de cette fonction, on ajoute l'arbre final à l'attribut privé de la classe `arbre_huffman` et enregistré l'arbre construit pour un texte en entrée.

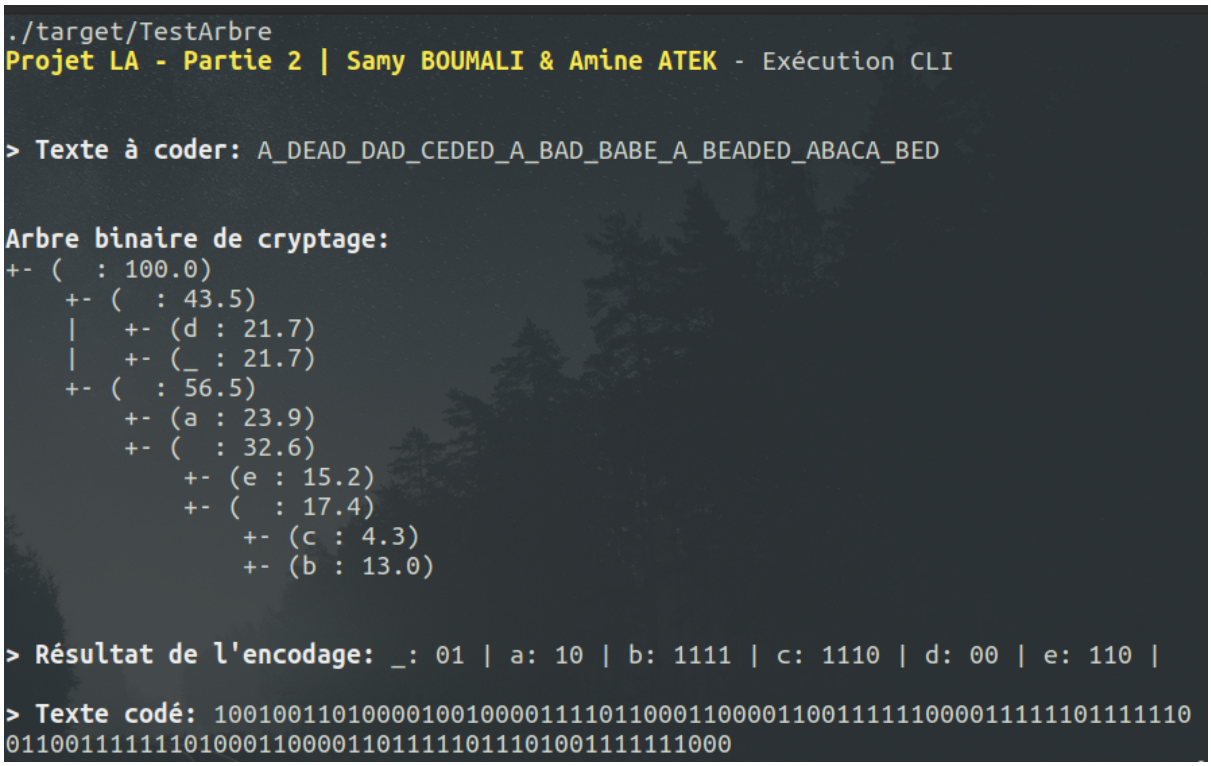
Codification du texte en entrée

La classe `Cryptage` constituant le corps de la Partie 2 du projet, elle contient également la fonction permettant l'encodage du texte en entrée une fois l'arbre construit, toujours dans la continuité de l'exécution de l'algorithme de Huffman.

La fonction `encodage()` stocke le résultat de l'exécution de l'algorithme dans une `std::map<char, std::string>` associant chaque lettre à son encodage.

Pour chaque nœud de l'arbre, la fonction se comporte comme la fonction de `recherche` dans l'arbre binaire, en stockant pendant le parcours le chemin emprunté grâce aux **étiquettes 0 et 1 selon si le parcours s'effectue vers le fils gauche ou droit.** La dernière étiquette ajoutée est retirée si le parcours atteint une feuille de l'arbre et doit revenir en arrière.

Le code associé à une lettre est finalement formé de la chaîne de caractère finale enregistrée dans la `std::map`, et une chaîne de caractère du résultat de l'encodage mis bout à bout est renvoyé à la fin de l'exécution de la fonction.

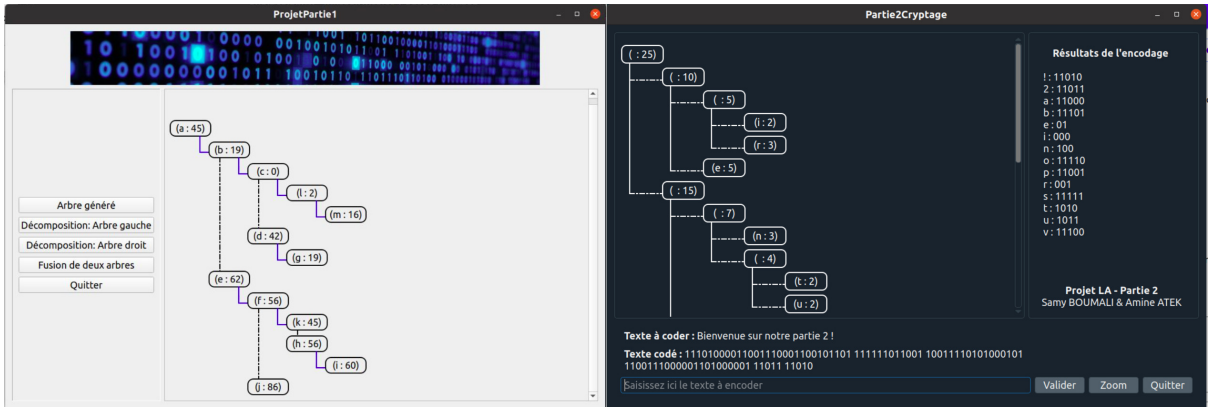


Résultat de l'exécution de notre programme sur le terminal de commandes.

```
Projet LA - Partie 2 | Samy BOUMALI & Amine ATEK - Exécution CLI

> Texte à coder : A_DEAD_DAD_CEDED_A_BAD_BABE_A_BEADED_ABACA_BED

Arbre binaire de cryptage:
+- (  : 100.0)
    +- (  : 43.5)
    |   +- (d : 21.7)
    |   +- (_ : 21.7)
    +- (  : 56.5)
        +- (a : 23.9)
        +- (  : 32.6)
            +- (e : 15.2)
            +- (  : 17.4)
                +- (c : 4.3)
                +- (b : 13.0)

> Résultat de l'encodage: _: 01 | a: 10 | b: 1111 | c: 1110 | d: 00 | e: 110

> Texte codé: 1001001101000010010000111101100011000011001111110000111111011111100
110011111111010001100001101111101110100111111100
```

Enrichissement de l'interface graphique

Nous avons entièrement revu dans cette partie notre **interface graphique,** qui avait non seulement un souci de compatibilité couleur entre les OS (clairs / foncés) et ne permettait pas d'afficher l'arbre binaire de manière compréhensive.



Amélioration de l'interface graphique entre les deux parties du projet. Les résultats sur la droite sont réalisés avec les occurrences sans pourcentages.

L'arbre de Huffman s'affiche toujours **de la gauche vers la droite,** mais les liaisons des nœuds ont été clarifiées de sorte à ce que chaque sommet ait un lien envers son sommet parent.

Notre `GridLayout` principale de la classe `MainWindow.cc` s'organise toujours en une **zone principale de dessin** contenue dans une `ScrollArea`, à laquelle nous avons ajouté un espace pour afficher les **résultats de l'encodage** pour chaque caractère.

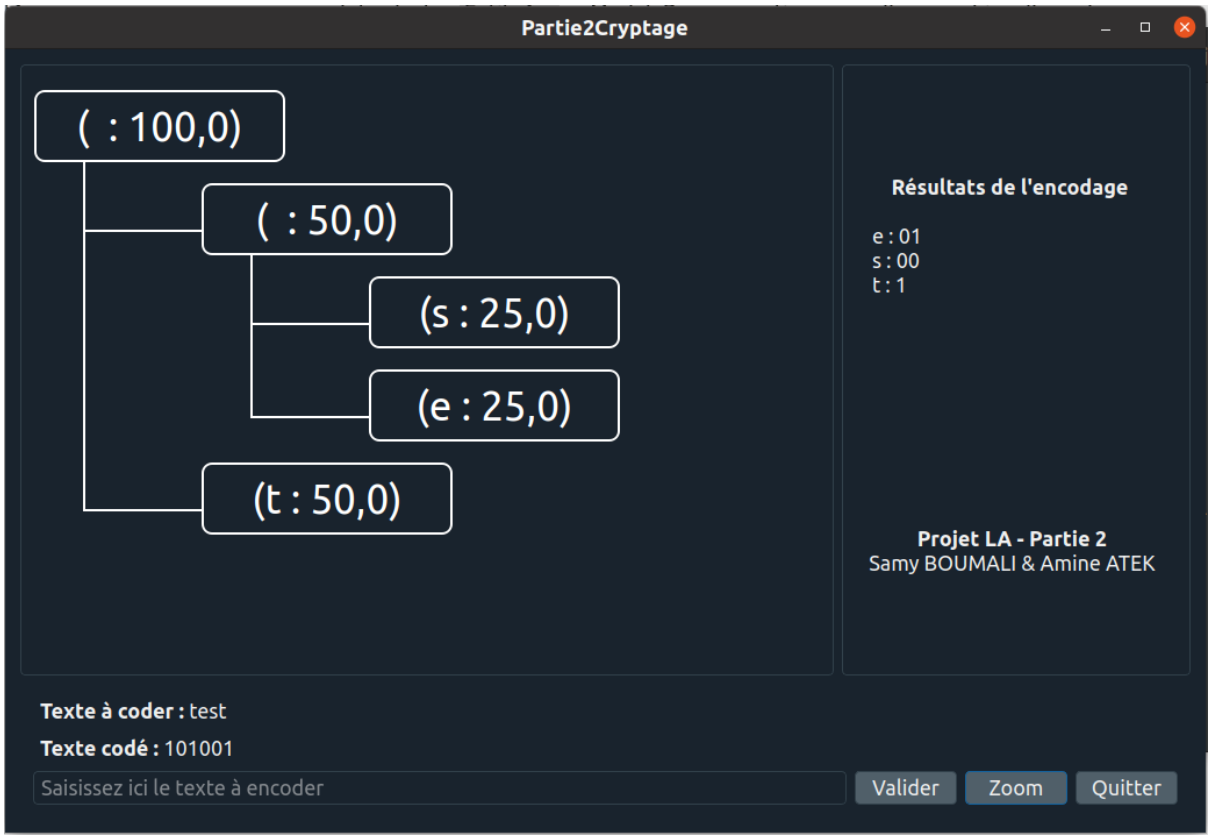
Le menu a été déplacé au bas de l'application, augmenté d'une **barre de saisie** permettant à l'utilisateur d'entrer directement son texte puis de valider à l'aide de la touche `Entrée` ou du bouton `Valider`. Deux `QLabel` assurent l'affichage du **texte en entrée** ainsi que du **résultat encodé** par nos algorithmes implémentés.

Le bouton `Zoom` augmente simplement un multiplicateur d'échelle et re-dessine l'arbre pour une meilleure lisibilité.

Afin de palier aux soucis de compatibilité couleur (thème foncé déclenché par défaut sur certaines configurations, empêchant l'affichage de notre arbre en couleur noire), nous avons décidé d'intégrer un thème personnalisé `QDarkStyle` customisant et modernisant l'affichage des différents éléments graphiques.

La **partie technique** de notre interface réside dans l'utilisation d'une classe `Singleton` (`Context.cc`) assurant la gestion de l'instance unique du projet contenant **l'arbre de Huffman, le texte en entrée et le résultat de l'encodage** sous forme d'une `std::map`. Cette classe est pivot entre les classes définissant nos algorithmes de cryptage et les classes permettant l'affichage graphique de l'application. Les informations sont transmises à l'aide de plusieurs `Getters` appelés via les `Slots` et `Signaux` de la librairie graphique `Qt`.

Aperçu concret de la bonne exécution de l'algorithme



Exemple d'exécution pour tester l'encodage du texte « Test » saisi directement sur l'application.