Projet Partie 3 - Compte rendu

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

Dans cette partie 3 du projet, le décryptage d'un texte en entrée a été implémenté suivant deux algorithmes dans la classe `Décryptage`. L'un prenant un texte codé et une map des encodages en entrée, l'autre prenant un texte codé et un arbre binaire de `Cryptage` construit en Partie 2.

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 troisième 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 *gif* `Apercu.gif` affichant un exemple d'exécution graphique complète détaillant les possibilités de résultats du décryptage ;
- 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 le fichier `CMakeLists`, les sources et headers de la partie 3 ;
- Le répertoire `qdarkstyle/` permettant la personnalisation de l'affichage graphique.

Commandes et utilisation

1. `make` pour exécuter la partie 3 sous interface graphique avec l'outil de compilation CMake ;
2. `make gui` pour exécuter la partie 3 sous interface graphique avec QMake ;
3. `make listing` pour générer la documentation Doxygen et le listing des fichiers ;
4. `make cli` pour exécuter en affichage sur terminal sans faire intervenir les classes Qt ;
5. `make debug` pour exécuter en mode debug (Valgrind et mem-check).

Précisions

1. La compilation graphique nécessite le programme `cmake`, outil de compilation disponible dans le VM UVSQ ; ou `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

Traité dans le compte rendu de la partie 2.

Construction de l'arbre binaire de Cryptage

Traité dans le compte rendu de la partie 2.

Codification du texte en entrée

Traité dans le compte rendu de la partie 2.

Décryptage d'un texte codé à l'aide d'une map ou d'un arbre

La classe `Décryptage` constituant le corps de la Partie 3 du projet définit nos algorithmes de décryptage d'un texte codé en entrée à l'aide d'une map des encodages (Caractère / Symbole binaire) ou d'un arbre binaire de cryptage.

Décryptage à partir d'une map

Ce premier algorithme (inutilisé dans le projet) consiste en le décryptage d'un **texte codé accompagné de sa table des encodages.**

Le **dépassement de taille d'encodage maximal de la Map** permet de vérifier si le texte entré a bien **été codé à partir de la partie 2 ou non.**

Pour chaque caractère du texte crypté, on l'ajoute à une chaîne courante et on regarde dans la map si un symbole peut être formé à partir de cette chaîne.

Si la taille de la chaîne courante dépasse la taille maximale, on conclut que les caractères cryptés n'existent pas dans cette map, **impliquant que ce texte n'a pas été codé avec la map donné**. A l'inverse, si le texte est correctement retranscrit, on conclut que **le texte donné pourrait avoir été codé avec cette map** car ses encodages sont inclus dans ceux de la map.

Décryptage à partir d'un arbre

Ce deuxième algorithme consiste en le décryptage d'un **texte codé accompagné d'un arbre binaire de cryptage (Huffman).**

On utilise un pointeur sur un sommet afin de se déplacer dans ce dernier selon le caractère lu (0 ou 1) et, dès qu'on arrive sur une feuille (**sommet avec une lettre du texte clair**), on l'ajoute à la chaîne de décryptage. **On replace ensuite le pointeur sur sommet à la racine de l'arbre pour réitérer l'opération.**

Rappel sur notre choix d'encodage : tous les caractères ASCII étendus sont encodés à l'exception de l'espace ainsi que certains signes de ponctuation que nous avons volontairement voulu laisser en clair lors du cryptage.

Si le caractère lu correspond à un espace ou un signe de ponctuation non encodé, c'est à dire qu'il doit être retranscrit directement en clair :

1. Le visiteur est à la racine : pas d'incohérence, nous n'étions pas en train de lire un symbole encodé donc **on ajoute donc le caractère à la chaîne décryptée.**
2. Le visiteur se situe sur un sommet de l'arbre : cela signifie que l'algorithme attendait une suite aux symboles encodés, ceci nous confirme que **l'arbre défini ne correspond pas au texte codé entré, une erreur est renvoyée.**

Une fois l'ensemble des caractères du texte crypté lus, des vérifications selon le positionnement du visiteur d'arbre sont réalisées afin de savoir **si le texte fourni correspond bien à l'arbre binaire de cryptage.**

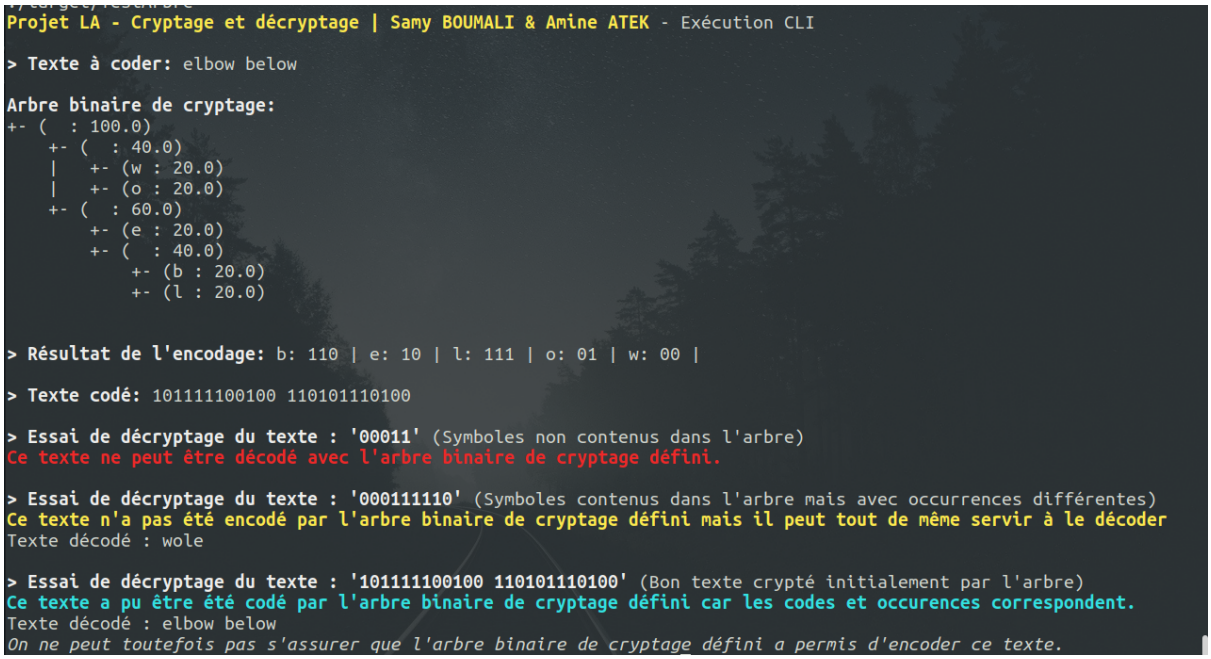
Mais il y a d'autres étapes de vérifications :
On appelle un objet de de type Lecteur afin d'obtenir les occurrences du texte décrypté, et on compare les occurrences obtenues à celles qu'on a dans l'arbre :
- Si ce sont les mêmes on dit que le texte de base a peut-être été encrypté par la partie 2 du projet, mais que cela ne peut pas être sûr à 100% (on peut avoir deux textes qui ont les mêmes fréquences mais pas les mêmes lettres, exemple "abcd" et "efgh")
- Sinon on dit que le texte a pu être décrypté avec l'arbre fourni, **mais il ne correspond définitivement pas à l'arbre qui a codé le texte à l'origine**.

Interprétation des résultats

Les résultats de ces algorithmes se divisent en trois cas :

1. **Le texte ne peut être décodé avec l'arbre défini :** survient lorsqu'après la lecture de tous les caractères du texte codé, le visiteur d'arbre n'a pas atteint une feuille (c'est à dire qu'il y a des **symboles non convertis en une lettre**). Ceci implique que **le texte n'a pas été codé via la partie 2.**
2. **Le texte n'a pas été encodé avec l'arbre défini mais peut toutefois être décodé avec :** lorsque le visiteur d'arbre se trouve bien sur la racine, cela signifie que **la chaîne codée saisie comprend uniquement des symboles présents dans l'arbre** binaire de cryptage. Une vérification est ensuite sur les occurrences de ces symboles *(01, 10, etc.)* entre l'arbre et la chaîne entrée et cette erreur est levée lorsqu'une lettre est manquante ou qu'une autre est présente plus de fois que prévu par l'arbre.
3. **Le texte pourrait avoir été encodé avec l'arbre défini :** lorsque le visiteur d'arbre se trouve sur la racine, et que **les occurrences de chaque**

symbole dans la chaîne entrée correspondent à ceux présents dans l'arbre. Ceci ne peut cependant pas nous prouver réellement que le texte saisi provient de l'arbre défini mais simplement qu'il comprend les mêmes symboles encodés avec les mêmes exactes fréquences d'apparition. *Par exemple, les mots* `abcd` *et* `efgh` *auraient le même encodage et le même arbre binaire de cryptage.*



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

```
Projet LA - Cryptage et décryptage | Samy BOUMALI & Amine ATEK - Exécution CLI

> Texte à coder: elbow below

Arbre binaire de cryptage:
+- (  : 100.0)
    +- (  : 40.0)
    |   +- (w : 20.0)
    |   +- (o : 20.0)
    +- (  : 60.0)
        +- (e : 20.0)
        +- (  : 40.0)
            +- (b : 20.0)
            +- (l : 20.0)

> Résultat de l encodage: b: 110 | e: 10 | l: 111 | o: 01 | w: 00

> Texte codé: 101111100100 110101110100

> Essai de décryptage du texte : '00011' (Symboles non contenus dans l arbre)
Ce texte ne peut être décodé avec l arbre binaire de cryptage défini.
```

```
> Essai de décryptage du texte : '000111110' (Symboles contenus dans l arbre mais
avec occurrences différentes)
Ce texte n a pas été encodé par l arbre binaire de cryptage défini mais il peut t
out de même servir à le décoder
Texte décodé : 'wole'

> Essai de décryptage du texte : '101111100100 110101110100' (Bon texte crypté in
itialement par l arbre)
Ce texte a pu être été codé par l arbre binaire de cryptage défini car les codes
 et occurences correspondent.
Texte décodé : 'elbow below'
On ne peut toutefois pas s assurer que l arbre binaire de cryptage défini a permi
s d'encoder ce texte.
```

Enrichissement de l'interface graphique

Notre `GridLayout` principale de la classe `MainWindow.cc` s'organise 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 - *Partie 2*.

L'interaction utilisateur est au bas de l'application, comprenant une **barre de saisie pour le texte clair à coder,** et une seconde **pour le texte à décoder.** Les deux barres de saisies sont **liées en temps réel au programme,** simplifiant ainsi la nécessité de cliquer sur un bouton pour `valider` le texte.

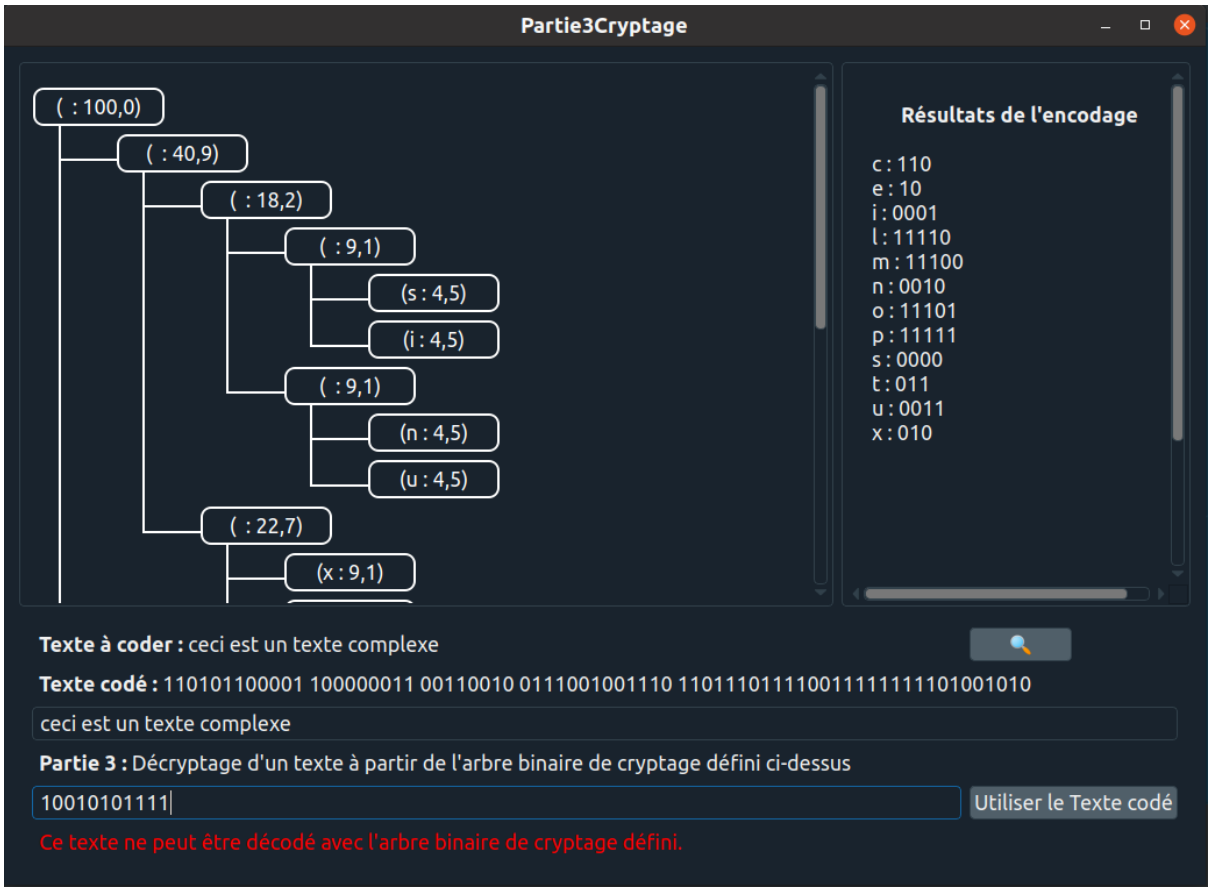
Deux premiers `QLabel` assurent l'affichage du **texte en entrée** ainsi que du **résultat encodé** par nos algorithmes implémentés, le dernier assurera quant à lui le **résultat du décryptage :** le statut donné par **l'algorithme de décryptage** ainsi que le **texte décodé** s'il existe.

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

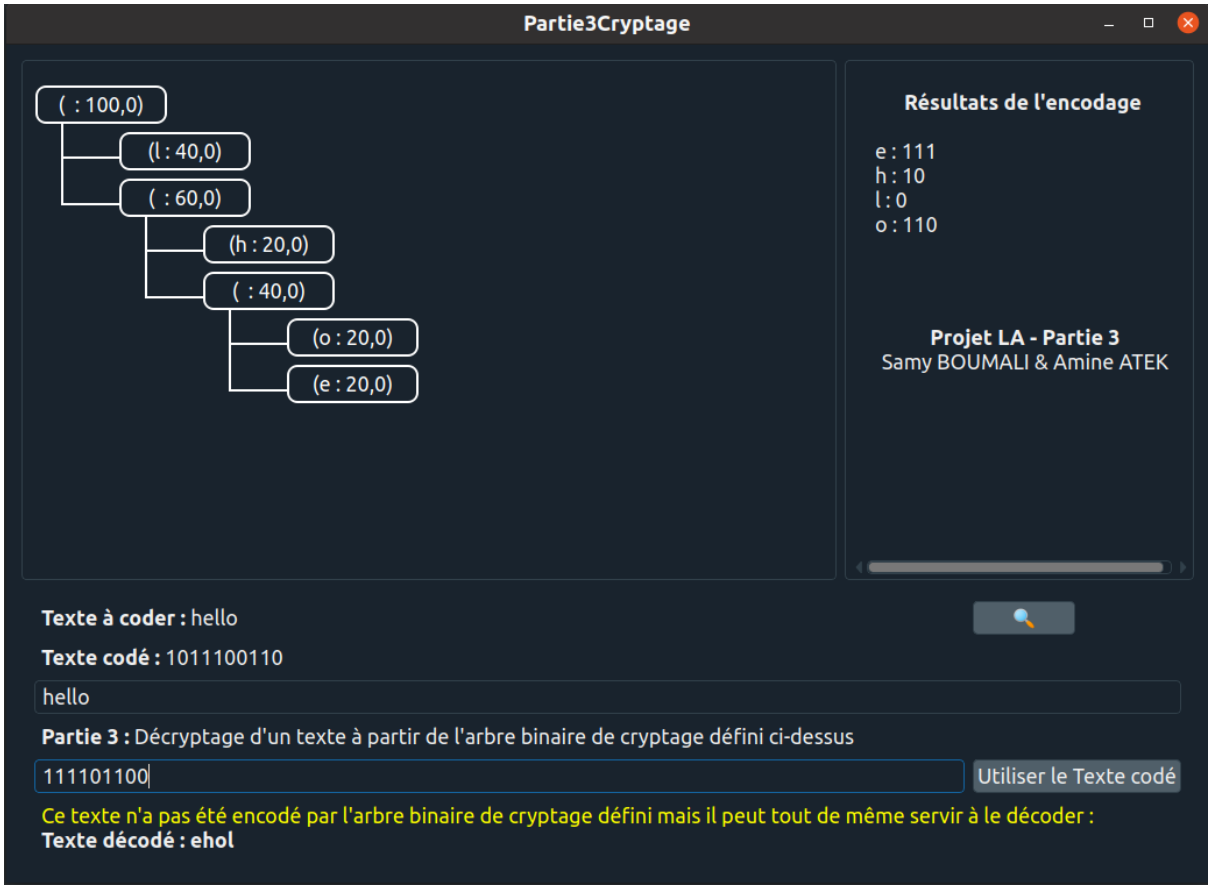
Les `Slots` et `Signaux` échangés par les classes de la GUI et le `Context` ont été détaillés dans le compte rendu de la *partie 2*. La classe `GetUserText`, responsable de l'interaction utilisateur, se charge de **transmettre le texte saisi** au `Context` qui le cryptera / décryptera et le renverra via signal pour l'affichage du résultat.

L'événement suite à la pression sur la touche `ESC` est surchargé de sorte à quitter l'application lorsqu'il est enregistré, ceci expliquant notre choix de retirer le bouton `Quitter`.

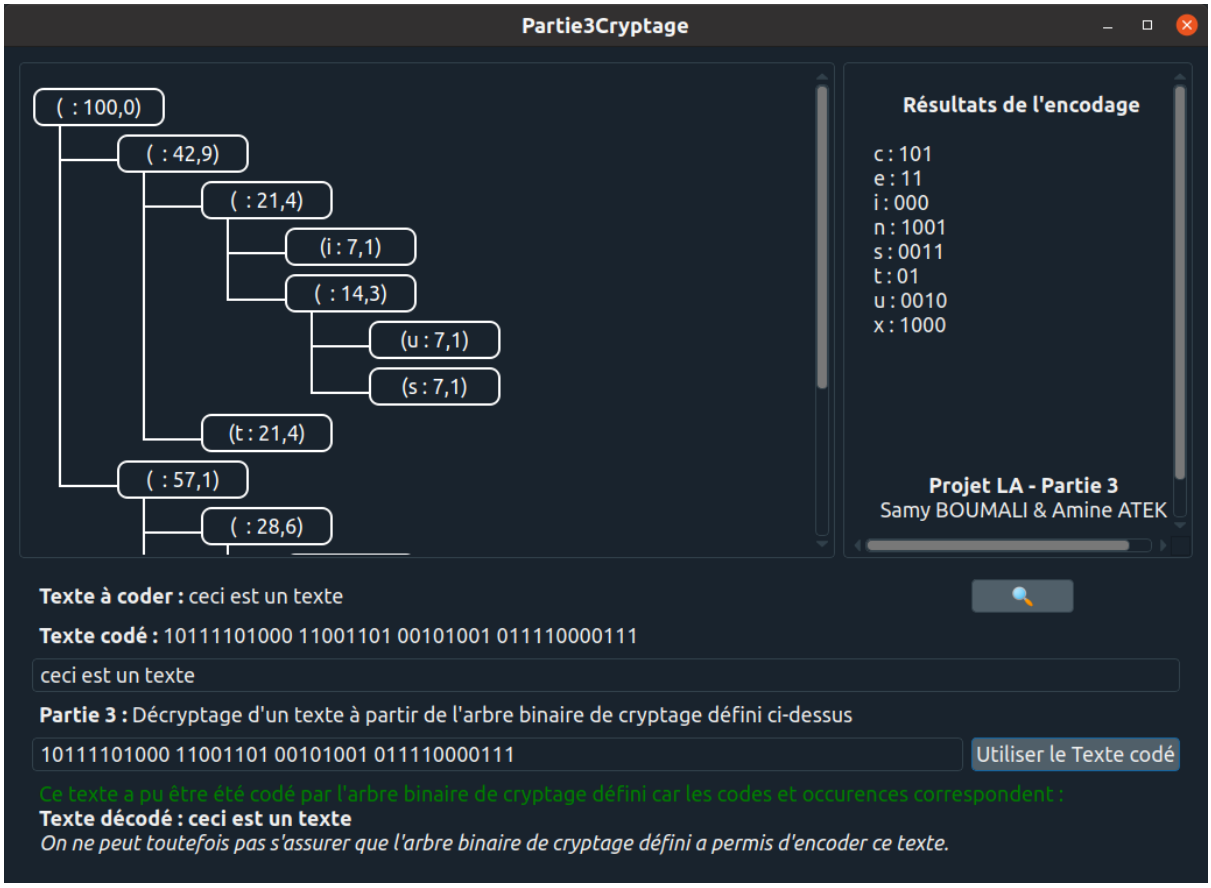
Aperçus concrets de l'exécution du programme



Exemple d'exécution pour tester le décryptage d'un mauvais texte (utilisant des symboles non contenus dans l'arbre).



Exemple d'exécution pour tester le décryptage d'un texte dont les symboles sont contenus dans l'arbre mais avec occurrences différentes. (Ici : un L est manquant)



Exemple d'exécution avec un texte dont les symboles sont contenus dans l'arbre avec les bonnes occurrences. Ce texte a été codé avec l'arbre binaire défini en partie 2.