HE-ARC INGÉNIERIE - 2282.1 PROJET P2

PROJET SEMESTRIEL P2 : GRAPH ++

MANUEL DÉVELOPPEUR

Flückiger Jonas, Plumey Simon, Tschan Damien
26/05/2023

TABLE DES MATIÈRES

Tous les termes utilisés dans ce document sont à comprendre dans leur sens épicène.

INTRODUCTION

Ce document a pour but d'expliquer, dans les grandes lignes, la structure de notre projet, les technologies utilisées et les choix que nous avons faits. Ce document est principalement destiné aux développeurs qui reprendraient le projet dans le futur. Nous considérons que le développeur en question possède les bases nécessaires en C++ et notamment avec le Framework Qt.

CONTEXTE

Graph++ est un projet réalisé durant le cours "2282.1 Projet P2" du programme de bachelor en informatique et systèmes de communication.

"Graph++" comporte une application desktop qui offre aux utilisateurs une plateforme flexible pour créer, manipuler et analyser des graphes. Le projet lui-même est séparé entre librairie analytique et présentation graphique, afin de permettre la réutilisation du code dans d'autres domaines. Signifiant que Graph++ n'est pas seulement une application graphique, mais aussi une librairie C++ standard réutilisable. Cet aspect est l'un des atouts principaux que l'équipe de développement voulait mettre en avant et a été un facteur important dans de nombreuses décisions du projet.

Pour plus d'informations et compléments, nous vous invitons à consulter le rapport complet du projet et la documentation Doxygen.

SPÉCIFICATIONS TECHNIQUES

	Outil	**Version(s) utilisée(s)**
Framework	Qt	6.4.2, 6.5.0
IDE	Qt Creator	10.0.0, 10.0.1
Langage	C++	17
OS	Windows 10/11	Education 64 Bits
Gestion de version	Git	2.38.1.windows.1

STRUCTURE DE PROJET

APPLICATION

Voici la structure globale de la partie application :

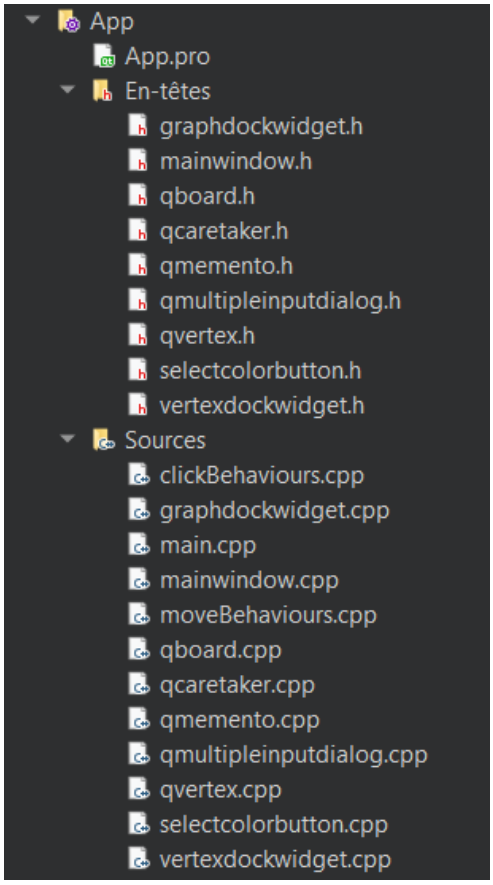


Figure 1 Structure de l'application GUI

GRAPHDOCKWIDGET.H – GRAPHDOCKWIDGET.CPP

La classe GraphDockWidget est une classe Qt héritant de la classe QWidget. C'est cette classe qui permet d'afficher les résultats des analyses du graphe sélectionné. Elle est utilisée au sein de la classe MainWindow qui l'utilise comme *Dock* faisant partie de l'overlay.

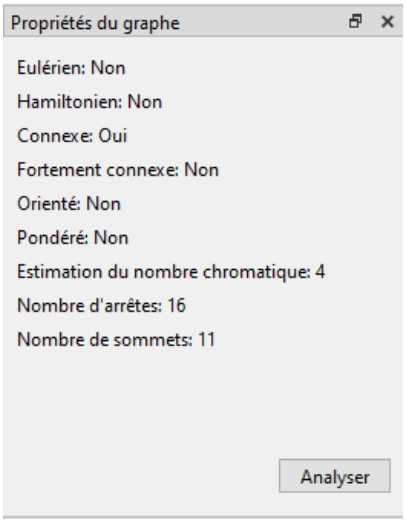


Figure 2 – Exemple de la classe GraphDockWidget

MAINWINDOW.H – MAINWINDOW.CPP

La classe MainWindow est une classe Qt héritant de la classe QMainWindow. C'est cette classe qui possède tous les éléments de l'application telle que les QMenu, les QActions, la QToolBar, l'overlay et une zone QMdiArea (Multiple Document Interface), qui permet la gestion de plusieurs onglets dans l'application.

QBOARD.H – QBOARD.CPP

La classe QBoard est une classe Qt de type QWidget. Elle contient l'environnement de dessin qui permettra à l'utilisateur de dessiner son graphe. C'est également cette classe qui sera responsable d'effectuer toutes les actions en fonction de l'outil sélectionné.

Afin de ne pas surcharger le fichier QBoard.cpp, l'implémentation des fonctions propre à un click se trouve dans le fichier *clickBehaviours.cpp* et l'implémentation des fonctions propres à un mouvement se trouve dans le fichier *moveBehaviours.cpp*.

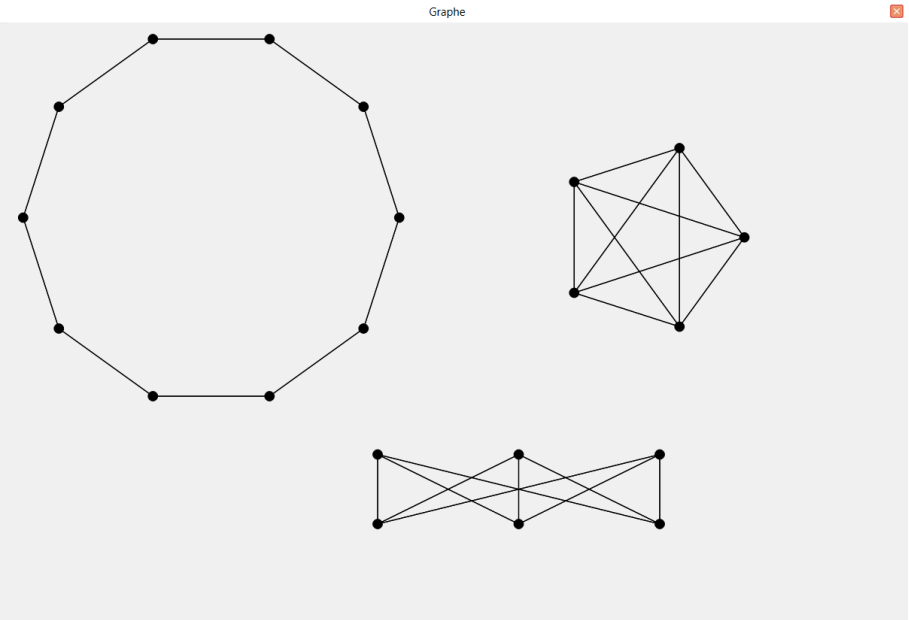


Figure 3 Exemple de la classe QBoard

QCARETAKER.H – QCARETAKER.CPP

La classe QCaretaker est une classe Qt héritant de la classe QObject. C'est une des deux classes nécessaires à l'implémentation du design pattern *Memento*. Celci s'occupe de garder en mémoire la pile des états de l'application. C'est elle qui possède les actions « d'undo » et « redo ».

QMEMENTO.H – QMEMENTO.CPP

La classe QMemento est une classe faisant partie d'une des deux classes nécessaires à l'implémentation du design pattern *Memento*. Cette classe est un simple *container* possède la liste d'adjacence du graphe à un moment t.

QMULTIPLEINPUTDIALOG.H - QMULTIPLEINPUTDIALOG.CPP

La classe QMultipleInputDialog est une classe Qt héritant de QDialog. C'est une classe utilitaire qui permet de demander à l'utilisateur de rentrer plusieurs champs de type QSpinBox. Cette classe a été implémentée pour les outils de création de graphes remarquables. En effet, l'utilisateur doit entrer un certain nombre de paramètres variable selon le graphe à créer. Il nous fallait donc une classe générique permettant de demander à l'utilisateur un nombre variable de paramètres.

Cette classe a été inspirée d'un exemple trouvé sur un poste internet[1] et a été adaptée à nos besoins.

[1] https://stackoverflow.com/a/53332748

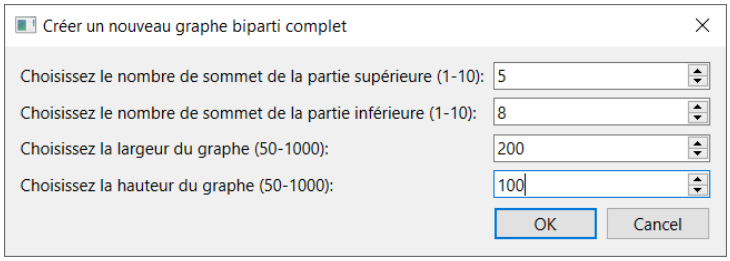


Figure 4 Exemple de la classe QMultipleInputDialog

QVERTEX.H - QVERTEX.CPP

La classe QVertex est une classe symbolisant un sommet du graphe. Cette classe est utilisée pour templatée la classe *Graph* de la librairie graph++. QVertex est une classe *container* qui possède tous les attributs nécessaires à l'identification d'un sommet, en l'occurrence, un nom, une position, des couleurs …

SELECTCOLORBUTTON.H - SELECTCOLORBUTTON.CPP

La classe SelectColorButton est une classe Qt héritant de la classe QPushButton. Cette classe permet de créer un bouton qui, lorsqu'on clique dessus, nous permette de sélectionner une couleur. Elle est notamment utilisée pour sélectionner les couleurs d'un sommet sélectionné.

Cette classe est entièrement reprise d'un poste internet[2].

VERTEXDOCKWIDGET.H - VERTEXDOCKWIDGET.CPP

Au même titre que la classe GraphDockWidget, la classe VertexDockWidget est une classe Qt héritant de la classe QWidget. C'est cette classe qui permet d'afficher les propriétés du sommet sélectionné et de les modifier en appuyant sur le bouton à cet effet. Cette classe est utilisée au sein de la classe MainWindow qui l'utilise comme *Dock* faisant partie de l'overlay.

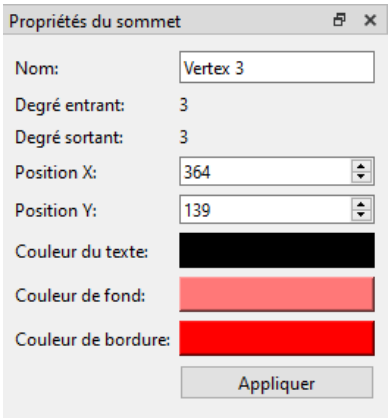


Figure 5 Exemple de la classe VertexDockWidget

[2] https://stackoverflow.com/a/43871405

MAIN.CPP

Le fichier main.cpp est le point d'entrée du programme. C'est lui qui va s'occuper d'instancier une nouvelle MainWindow.

LIBRAIRIE

STRUCTURE

La structure de fichiers de la librairie est très simple :

- **graph.h** : Fichier principal contenant la description de la classe *Graph<T>*
- **edge.h** : Fichier contenant la description de la classe *Edge<T>*
- **queue_element.h** : Fichier contenant la description de la structure *queue_element.*

Comme vous pouvez le voir, la librairie ne contient pas de fichier .cpp. Cette particularité est dû au fait que les classes *template* ne sont pas décrites de la même manière qu'une classe normale et la compilation ne fonctionnait pas s'il l'on séparait la déclaration de la classe de son implémentation. Une autre raison, valable pour la structure *queue_element* est que la description est très courte et ne nécessite pas de fichier d'implémentation.

La structure du graphe est stockée en mémoire avec une liste d'adjacence. Cette liste d'adjacence est implémentée avec une map C++ standard non triée, ou la clé de la map est une référence à un sommet et la valeur de la map est une liste de références à des arcs.

```
/// @brief Represents a graph as an adjacency list
std::unordered_map<T *, std::list<Edge<T> *>> adjacencyList;
```

Cette map peut être parcourue très facilement avec une boucle *foreach* C++, puis la liste elle-même peut être parcourue, elle aussi très simplement avec une boucle *foreach* C++. Voici un exemple de parcours de la liste d'adjacence :

```
for(auto pair : this->adjacencyList){
    // pair is a std::pair struct
    // with a pointer to a T instance as its first element
    // and a list of pointer to Edge<T> instances as its second member

    for(auto edge : this->adjacencyList[pair.first]){
        // edge is a pointer to an Edge<T> instance
    }
}
```

UTILISATION

Voici un simple exemple d'utilisation de la libraire :

```
#include <graph.h>

///.......
// Create some vertices
int vertices[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};

// Create a graph for those vertices
Graph<int>* graph = new Graph<int>();

// Add the vertices to the graph
for(int i : vertices){
    graph->addVertex(&i);
}

// Add some edges to the graph (Circular graph)
for(int i : vertices){
    int nextIndex = i + 1;
    if(nextIndex >= nbVertices){
        nextIndex = 0;
    }
    graph->addDoubleEdge(&vertices[i], &vertices[nextIndex]);
}
```

Nous faisons ici un graphe composé d'entiers, mais n'importe quelle classe peut être utilisée. Une fois le graphe créé, vous pouvez accéder aux différentes méthodes :

```
// Basic properties
std::cout << graph->getNbVertices() << std::endl;
std::cout << graph->getNbEdges() << std::endl;
std::cout << graph->isOriented() << std::endl;
std::cout << graph->isWeighted() << std::endl;

// Algorithms
Graph<int>* hamiltionianPath = graph->getHamiltonianPath();
Graph<int>* shortestPathGraph = graph->getMinimumDistanceGraph(&vertices[3]);
Graph<int>* minimumSpanningTree = graph->getMinimumSpanningTree();
```

Toutes les méthodes et tous les algorithmes sont documentés. Référez-vous simplement à la documentation pour obtenir des informations supplémentaires sur certains algorithmes de la librairie.

TESTS

Le dossier « Tests » est un projet Qt de type « subdirectory ». Ce projet contient les batteries de tests de la librairie et nécessite que la librairie soit compilée au préalable. Chaque batterie de tests est un sous-projet de type « autotest » et peut être compilée/exécuté individuellement.

Les batteries de tests unitaires de bases (c.-à-d. « BasicGraphTest » et « ComplexGraphTest ») testent que les fonctionnalités les plus simples de la librairie fonctionnent. Ils construisent des graphes cycliques et complets, s'assurent que les nombres de sommets et d'arcs correspondent aux valeurs attendues et que la détermination des propriétés les plus simples est correcte. Ensuite, des tests unitaires sont dédiés aux algorithmes plus complexes. Ces tests construisent d'abord un graphe de base, puis construisent le sous-graphe exact de ce graphe correspondant à l'algorithme testé. L'algorithme est ensuite déroulé et le résultat est comparé au graphe attendu sommet par sommet, arc par arc. Si les nombres d'arcs et de sommets sont les mêmes, que tous les arcs et tous les sommets sont les mêmes et que les poids totaux des deux graphes sont égaux, le résultat est considéré comme correct.

Pour ajouter une nouvelle batterie de tests dans Qt Creator : **clic droit sur le dossier « Tests » > New subproject > Other project > Auto Test Project.**

Cela va créer un nouveau sous-projet pour votre batterie de tests, contenant un simple fichier C++ avec la description d'une classe de test automatiquement générée. Avant de commencer à développer, ajoutez à votre nouveau projet une dépendance à la librairie en modifiant le fichier .pro :

```
QT += testlib
QT -= gui

CONFIG += qt console warn_on depend_includepath testcase
CONFIG -= app_bundle

TEMPLATE = app

# Ligne à ajouter ---------------------------------
INCLUDEPATH += $$PWD/../../Lib/
DEPENDPATH += $$PWD/../../Lib/
LIBS += -L$$OUT_PWD/../../Lib/debug -lLib
# -------------------------------------------------


SOURCES +=  tst_basicgraphtest.cpp
```

Votre test est maintenant prêt à être utilisé. Rédigez les méthodes de tests comme signaux de la classe générée par Qt et lancez les tests à travers l'interface de Qt Creator dans le menu **Tools > Tests > Run all tests**.

Notez que chaque batterie de tests ne doit contenir qu'une seule classe. Si vous désirez séparer votre code, créez une nouvelle batterie de tests dans un projet dédié. Si vous rencontrez des problèmes, n'hésitez pas à vous référer aux tests déjà rédigés pour comprendre le fonctionnement du framework Qt tests. Vous pouvez notamment ignorer les erreurs concernant les 2 dernières lignes du fichier C++ :

```
QTEST_APPLESS_MAIN(MinimumDistanceGraphTest)

#include "tst_minimumdistancegraphtest.moc"
```

Ces lignes sont propres au framework Qt tests et sont parfois mal interprétée par Qt Creator. Redémarrer Qt Creator ou reconstruire le projet permet de résoudre le problème la majorité du temps. N'hésitez pas non plus à consulter la documentation[3] officielle du framework de tests Qt pour en apprendre plus.

INSTALLATION À PARTIR DE GIT

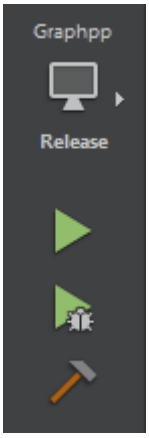
Voici une petite procédure d'installation permettant d'initialiser un projet Qt dans QtCreator à partir du repository git suivant :

https://gitlab-etu.ing.he-arc.ch/isc/2022-23/niveau-2/2282.1-projet-p2-sp-il/graphpp

1. Cloner le projet à partir du repository git en ligne
2. Ouvrir QtCreator. Fichier > Ouvrir un fichier ou un projet
3. Sélectionner le fichier graphpp\sources\Graphpp\Graphpp.pro
4. Et ensuite, cliquer sur le bouton « Configure Project »

CRÉER UN EXÉCUTABLE

La création d'un exécutable pour Windows 10 64 bits est simple, mais comprend quelques détails importants à savoir.



Pour commencer, veuillez compiler les sources du projet. Pour cela, nous allons d'abord modifier la configuration Qt pour que QMake *build* les sources en mode *release.* Pour ce faire vous pouvez modifier la configuration du projet avec le bouton de configuration en bas à gauche de votre écran sur Qt Creator. Sélectionnez l'option *release* plutôt que *debug.* Une fois cette modification effectuée, modifiez les fichiers .pro dans lesquels vous définissez une dépendance sur la librairie. En effet, le chemin de la librairie est spécifié avec un dossier « debug » qui doit être modifié en « release ». Une fois ces modifications effectuées, vous pouvez compiler les sources en mode *release*.

[3] Disponible sur https://doc.qt.io/qt-6/qtest-overview.html

Une fois les sources compilées, vous devriez trouver un dossier généré par le compilateur. Par défaut ce dossier est frère du dossier source « Graphpp » et se nomme « build-Graphpp-Desktop_Qt_6_5_0_MinGW_64_bit-Release », dans ce dossier, récupérez l'exécutable « App.exe » et placez-le dans le dossier où vous désirez déployer l'application. Nous imaginons dans notre cas déployer l'application dans un dossier « bin », à la racine du *repository*. Copiez l'exécutable dans ce dossier et ouvrez un terminal de ligne de commande Qt.

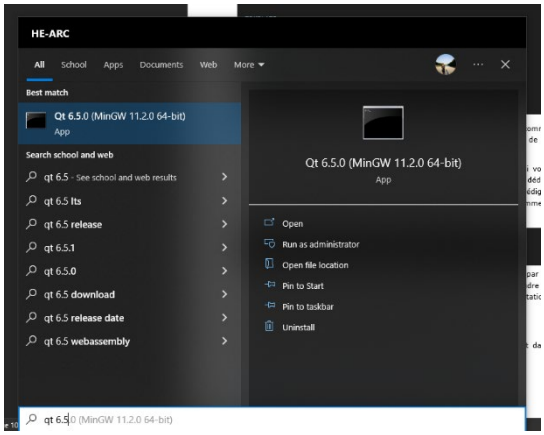


Figure 6 - Terminal Qt 6.5.0

Attention : Veuillez bien ouvrir une ligne de commande Qt et non pas un terminal standard, dans quel cas la procédure ne fonctionnera pas.

Ensuite, rendez-vous dans le dossier « bin » et exécutez la commande `windeployqt App.exe`. Si la commande n'est pas dans vos variables d'environnement, vous pouvez la trouver dans le dossier d'installation de Qt sous « Qt\6.5.0\mingw_64\bin ». Votre application est maintenant déployée.