HE-ARC INGÉNIERIE - 2282.1 PROJET P2

PROJET SEMESTRIEL P2 : GRAPH ++

RAPPORT DE PROJET

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.

RÉSUMÉ

Nous avions pour but de créer une application de bureau simple et intuitive, avec un ensemble de fonctionnalités intéressantes portant sur le domaine de la théorie des graphes. L'application permettrait de créer et générer des graphes différents, puis de les analyser pour déterminer des propriétés intéressantes. Le code faisant fonctionner l'application devait être réutilisable et découplé de l'interface graphique. Après une phase d'analyse où nous avons rédigé notre cahier des charges et planifié le projet avec un diagramme de Gantt, nous avons entamé une phase de conception où nous avons décrit en détail le fonctionnement et la structure de l'application. De cette phase de conception sont ressortis un diagramme de cas d'utilisation, un diagramme de classe, la répartition des tâches ainsi qu'une méthodologie à suivre pour la gestion du code et des tâches du projet. Une fois les spécifications écrites, nous avons commencé la phase de développement durant laquelle l'interface graphique, la création et gestion de graphes ainsi que l'analyse complète des graphes ont été implémentées. Il a fallu prioriser certaines fonctionnalités pour garantir que celles qui seraient implémentées soient réellement utiles et nécessaires. Le développement se déroula comme prévu à quelques détails près. Finalement, la majorité des fonctionnalités prévues a été implémentée. L'application finale permet de générer des graphes standards et d'en modifier la structure. Il est possible de modifier chaque sommet pour lui donner une identité claire, avec des couleurs et des noms. Les graphes de l'application peuvent être enregistrés sous forme de fichiers puis réouverts dans l'application et peuvent aussi être exportés comme images. Les outils d'analyse permettent de déterminer des propriétés connues des graphes avec des algorithmes connus comme l'algorithme de Prim ou de Dijkstra, et de les réutiliser plus tard à travers la librairie de l'application

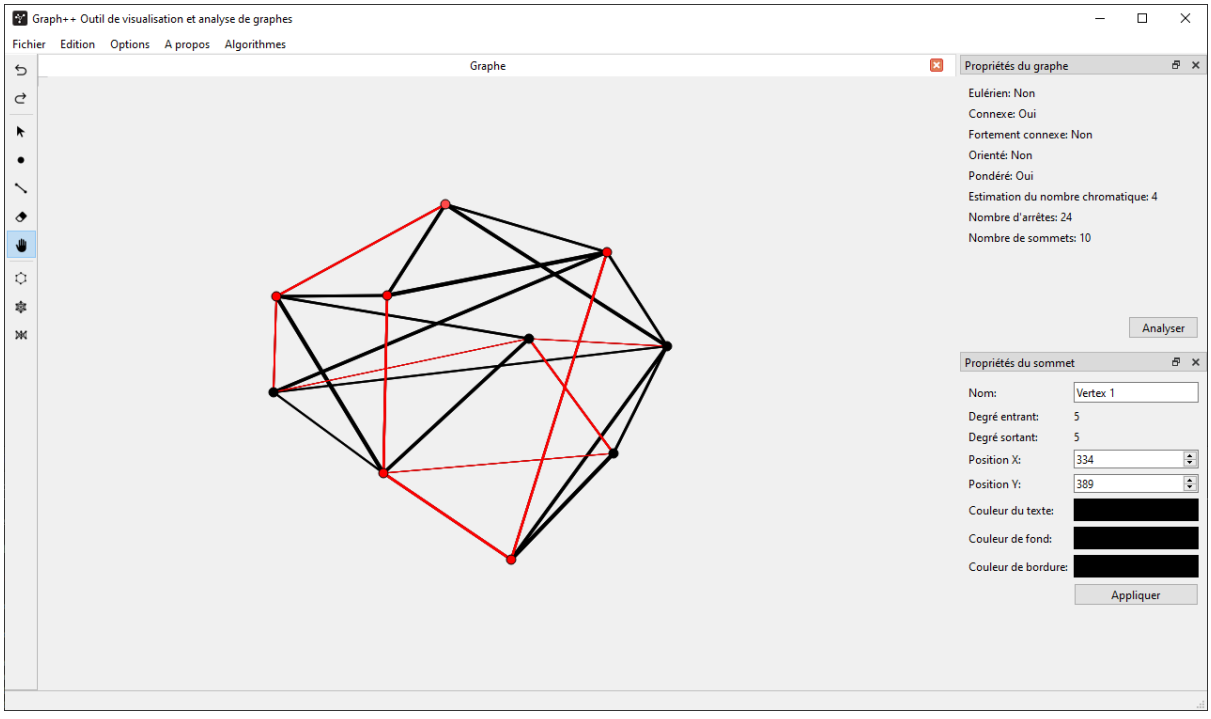


Figure 1 - Capture d'écran de l'application finale.

INTRODUCTION

Ce rapport documente le développement du projet "Graph++", réalisé dans le cadre du module "2282.1 Projet P2" du programme de bachelor en informatique et systèmes de communication. L'objectif de ce projet était de concevoir et développer une application desktop en C++ avec une interface graphique, si possible développée avec le *Framework* graphique Qt, dans des petits groupes de 3 élèves. Le choix du sujet est resté libre. Le projet a été réalisé conjointement avec le module « 2245.2 Génie logiciel II ». Pour ce cours, il était demandé d'appliquer les principes des méthodes agiles et du génie logiciel tout au long du projet.

Notre groupe, composé de M. Flückiger, M. Plumey et M. Tschan, désirait développer une application portant sur le domaine des mathématiques. Pour nous inspirer et trouver des cas d'utilisation réels, nous avons parcouru nos cours de mathématiques et avons choisi de réaliser une application portant sur la théorie des graphes.

"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.

L'application graphique serait intuitive et facile à utiliser, offrant un ensemble varié de fonctionnalités, ainsi que des éléments de bases auxquels les utilisateurs s'attendent lorsqu'ils utilisent une application de bureau, comme la sauvegarde de fichiers. Le tout avec une interface s'inspirant de logiciels de la suite Adobe comme Photoshop ou Illustrator.

La librairie se devait d'être flexible. Elle devrait supporter la majorité des types de graphes. La réutilisabilité du code devait être suffisante pour que la librairie puisse être réemployée dans des projets de domaines différents.

Pour que le projet se déroule sans problèmes et que la collaboration soit la plus efficace possible, les principes du génie logiciel seraient utilisés au sein de l'équipe. Une documentation extensive devait être rédigée, afin de permettre à une autre équipe de facilement pouvoir reprendre le projet et continuer les travaux. Tous ces aspects sont décrits plus amplement dans le reste du document.

Ce rapport présente en détail les différentes phases du projet, notamment l'analyse des besoins, la conception de l'application, les choix technologiques, les fonctionnalités implémentées, ainsi que les tests et les résultats obtenus. Il met également en évidence les défis rencontrés pendant le développement et les solutions mises en œuvre pour les surmonter.

ÉTAT DE L'ART

La théorie des graphes est une branche des mathématiques qui étudie les structures de réseau composées de sommets et d'arêtes (ou arcs). Les graphes sont utilisés pour modéliser des relations entre des entités, que ce soit dans le domaine des réseaux sociaux, de la logistique, de la biologie ou d'autres domaines. La théorie des graphes propose un ensemble de concepts et d'algorithmes pour l'analyse et la manipulation de ces structures.

Théorisée en 1736 par Léonard Euler, l'analyse de graphe est née d'un besoin direct d'un nouvel outil d'analyse. Elle a permis au mathématicien de résoudre le « *Problème des sept ponts de Königsberg* » en modélisant les ponts et les terres de la ville sous la forme d'un graphe[1].

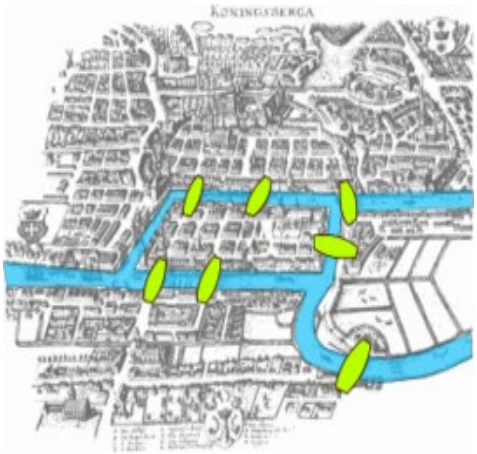


Figure 2 - Représentation visuelle du problème, Bogdan Giuşcă https://upload.wikimedia.org/wikipedia/commons/5/5d/Konigsberg_bridges.png

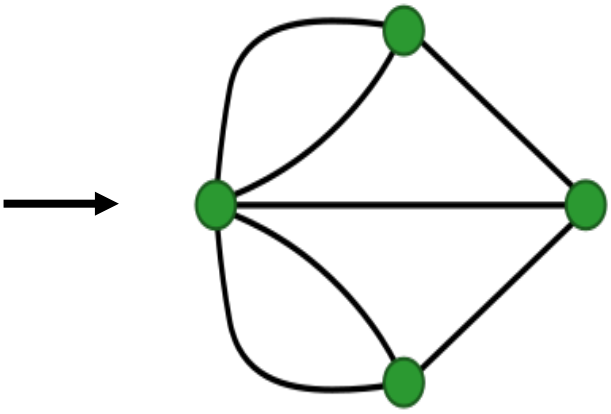


Figure 3 - Graphe correspondant au problème, Mark Foskey, https://upload.wikimedia.org/wikipedia/commons/9/96/K%C3%B6nigsberg_graph.svg

Le problème est de trouver un itinéraire pour le cortège du roi qui passerait par tous les ponts de la ville une seule fois et retournerait à son point de départ. En modélisant le problème comme un graphe, la solution revient à trouver un cycle (c.-à-d. un chemin qui termine sur le même sommet où il a commencé) passant par toutes les arêtes une seule fois. Aujourd'hui, un tel chemin est appelé un chemin eulérien, en honneur au mathématicien qui détermina ses propriétés pour résoudre ce problème. Euler prouva qu'un graphe possède un chemin eulérien si et seulement si chaque sommet possède un degré pair. Le problème n'avait donc pas de solution possible, puisque tous les sommets possèdent un degré impair.

[1] (Contributors to Wikimedia projects, 2003)

Ce n'est cependant que des dizaines d'années plus tard que la théorie des graphes fût plus amplement étudiée et standardisée, notamment en 1969 lorsque Frank Harary publie son ouvrage sur la théorie des graphes et permet aux mathématiciens, chimistes et ingénieurs de discuter le sujet avec les mêmes nomenclatures et standards[2].

Dans le contexte de la manipulation de graphes informatiques, il existe différentes structures de données permettant de modéliser un graphe.

- **Matrice de contiguïté** : La matrice de contiguïté est une représentation tabulaire d'un graphe, où les lignes et les colonnes représentent les sommets et les cellules de la matrice indiquent les arêtes entre les sommets. Si une cellule possède une certaine valeur (ex. la valeur 1), alors les sommets de sa ligne et de sa colonne possèdent un arc les reliant. La valeur de la cellule peut aussi indiquer la pondération de l'arête.
- **Liste d'adjacence** : La liste d'adjacence est une autre représentation courante d'un graphe, où chaque sommet est associé à une liste des sommets voisins auxquels il est directement connecté. Cette représentation permet de facilement accéder aux sommets voisins d'un sommet particulier, mais ne permet pas de pondérer les arêtes dans sa version la plus simple.

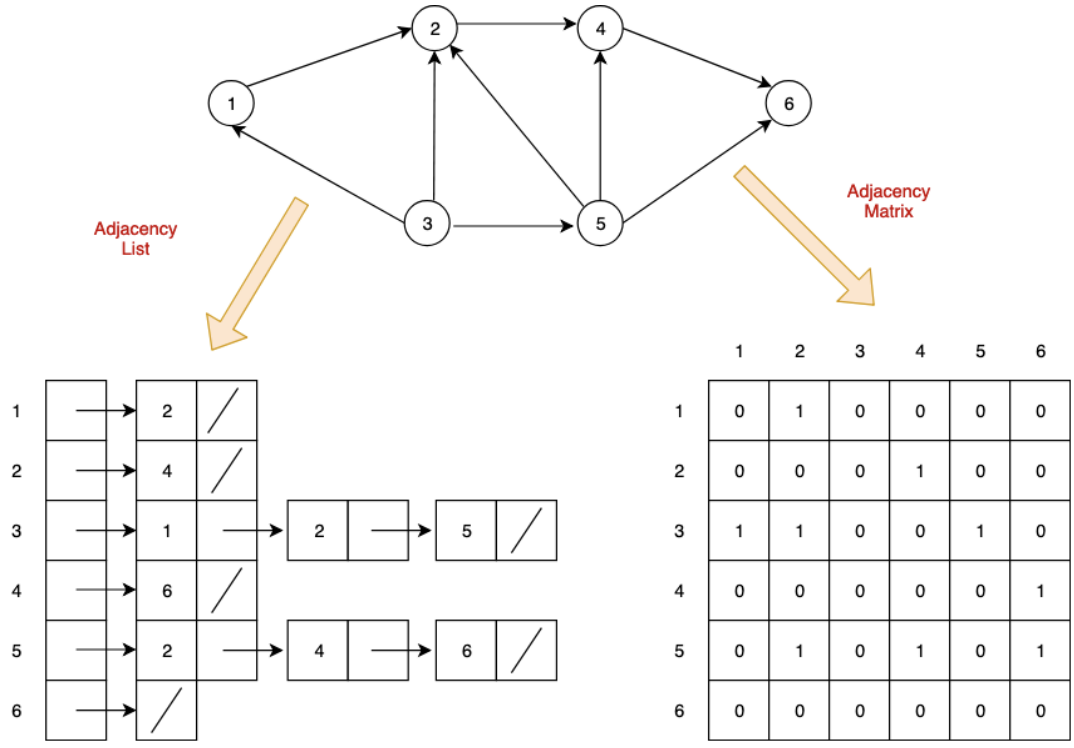


Figure 4 - Représentation d'un graphe sous ses 2 formes les plus communes, la liste d'adjacence et la matrice de contiguïté,

Algorithm Tutor https://algorithmtutor.com/Data-Structures/Graph/Graph-Representation-Adjacency-List-and-Matrix/

[2] (Contributors to Wikimedia projects, 2001)

Ces deux modèles possèdent leurs avantages et leurs inconvénients. Le choix de structures de données pour notre projet est plus amplement décrit dans la section « Structure de données »[3].

Il existe plusieurs outils informatiques permettant de travailler sur des graphes, Graphviz étant largement le plus connu et le plus utilisé. Graphviz est un ensemble d'outils logiciels permettant la description et l'agencement de graphe. Il possède son propre langage standard nommé « DOT » et permet de décrire la structure entre des nœuds, mais aussi d'associer à ces nœuds un nom, des couleurs, des images et d'autres propriétés visuelles. Son moteur de rendu graphique est souvent utilisé dans d'autres applications qui permettent de générer des schémas comme Doxygen, un outil de documentation informatique ou PlantUML, un outil de modélisation de schémas UML.

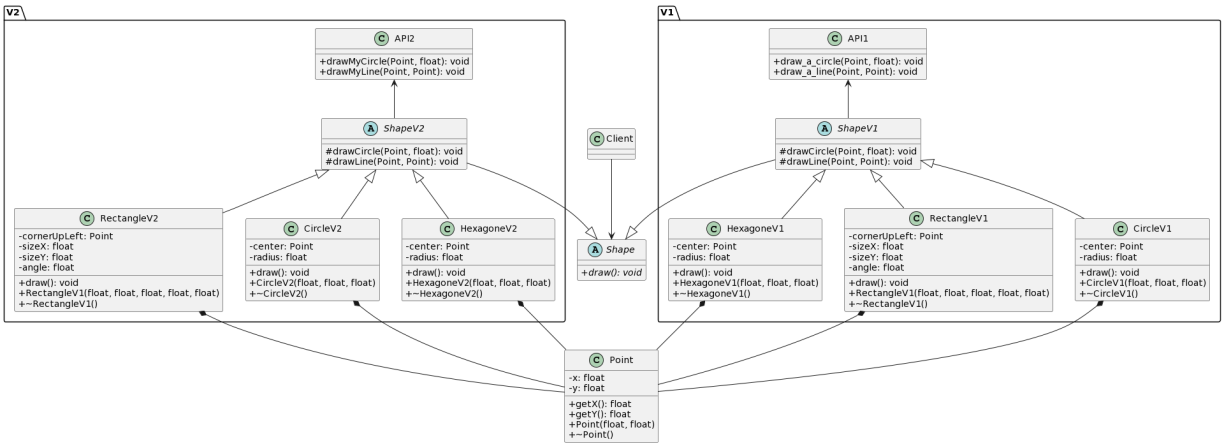


Figure 5 - Exemple de schéma de classes généré avec PlantUML et Graphviz

[3] Voir page 21

ANALYSE

OBJECTIFS

Les objectifs du projet sont les suivants :

PRIMAIRES

- Créer, modifier et visualiser un graphe.
- Enregistrer et charger les données d'un graphe.
- Analyse simple du graphe.

SECONDAIRES

- Analyse avancée du graphe.
- Navigation dans la zone de dessin.
- Gérer les graphes orientés/pondérés.
- Exporter le graphe en fichier image.

Vous pouvez trouver une description plus détaillée des objectifs dans la section « Spécifications détaillées »[4].

PÉRIMÈTRE DU PROJET

Graph++ est une application desktop, imaginée et créée pour Windows. Le framework choisi est Qt. Par conséquent, il théoriquement possible de créer une version pour Linux ou MacOS, mais celles-ci ne seront ni réalisées, ni testées dans ce projet.

DESCRIPTION FONCTIONNELLE DES BESOINS

Cette application a pour but de créer et d'analyser un graphe. Cela pourrait s'avérer très utile et le gain de temps serait conséquent, si ces opérations devaient s'effectuer à la main. Un autre but du projet est de réutiliser la partie algorithmique d'analyse du graphe pour d'autres projets. Cela pourra être réalisable en séparant correctement la partie analyse de la partie affichage.

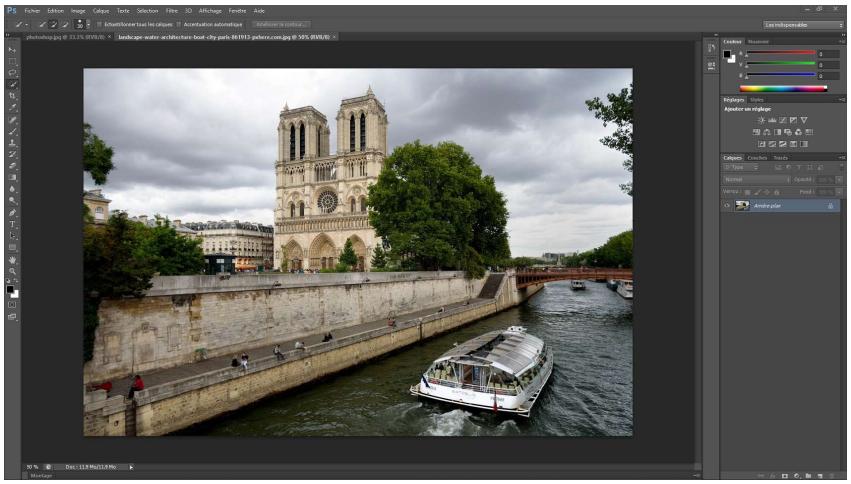


Figure 6 - Adobe Photoshop, inspiration d'interface/disposition des outils/UX

[4] Voir page 10

SPÉCIFICATIONS DÉTAILLÉES

OBJECTIFS PRIMAIRES

Nouveau graphe, ouverture d'un graphe existant, sauvegarde d'un graphe

Un nouveau graphe (non orienté) peut être créé grâce au menu correspondant.

Un graphe existant (créé avec Graph++) peut être ouvert depuis ce même menu.

Lors de la sauvegarde du graphe, le fichier créé contient les caractéristiques globales du graphe ainsi que la liste des sommets, leurs coordonnées et leurs propriétés, la liste des arêtes et leurs propriétés.

Un graphe doit être enregistré avec toutes ses caractéristiques, de manière que lors de l'ouverture, l'utilisateur puisse retrouver visuellement le graphe précédemment enregistré.

Le fichier dans lequel le graphe est enregistré sera si possible compatible avec au moins un autre standard.

Il sera exportable dans d'autres formats comme le langage DOT utilisé dans Graphviz

Les graphes peuvent aussi être exportés en fichier PNG.

Analyse

Les degrés des sommets sont calculés immédiatement lorsqu'une arête ou un sommet est ajouté ou supprimé.

Lorsque l'analyse du graphe est lancée, l'application calcule et affiche dans la section analyse de l'interface :

- Si le graphe est connexe, eulérien, hamiltonien
- Le nombre chromatique du graphe
- Le nombre total de sommets/arêtes

Interface

En haut de la page, l'utilisateur peut accéder à des menus qui lui permettent de créer, d'ouvrir ou de sauvegarder un graphe et d'accéder aux paramètres de l'application. Un menu d'édition permet d'annuler les dernières actions effectuées.

À gauche du graphe ouvert, l'utilisateur a à sa disposition des outils d'édition de graphes tels que :

- **Sommet** : placer un nouveau sommet
- **Arête** : placer une nouvelle arête entre deux sommets existants
- **Propriétés/Édition** : déplacer un sommet, éditer un sommet/une arête, lui donner un nom, voir ses propriétés telles que le degré du sommet (entrant et sortant si graphe orienté) ou la pondération d'une arête.

- **Déplacement** : se déplacer dans le plan, déplacement aussi possible avec Espace maintenu
- **Gomme** : supprimer l'objet (sommet et arêtes correspondantes, arête), un glissement de la souris supprime tous les objets touchés.
- **Annuler** : aussi accessible avec Ctrl + Z
- **Rétablir** : aussi accessible avec Ctrl + Y

À droite du graphe se trouvent les propriétés de l'objet en cours d'édition et l'analyse du graphe dans deux sections séparées. Les sections sont présentées sous la forme d'un volet séparé ou de deux volets (à définir lors de la mise en place selon la praticité)

Tests unitaires

Seules les fonctionnalités mathématiques liées aux graphes (p.ex. calcul du nombre chromatique) seront testées. Il n'est pas jugé utile de tester le bon fonctionnement de l'interface graphique.

OBJECTIFS SECONDAIRES

Types de graphes

Lors de la création d'un graphe, l'utilisateur choisit le type de graphe. Dans le cadre de ce projet, il ne peut que choisir si le graphe est orienté ou non. L'utilisateur peut en plus pondérer les arêtes du graphe.

Analyse

D'autres calculs et mesures sont ajoutés à l'analyse du graphe :

- Si le graphe est planaire
- Le nombre total de faces (si le graphe est planaire)
- S'il contient un sous-graphe (p. ex. $K_{3,3}$ ou K_5)

L'utilisateur peut calculer le chemin le plus court entre deux sommets (avec ou sans pondération des arêtes) grâce à un outil spécialisé.

Interface

L'interface graphique se présente sous la forme d'un MDI (Multiple Document Interface) dans laquelle chaque onglet est un graphe ouvert.

Une action du menu permet de fermer tous les onglets (graphes) ouverts.

Trois outils de génération de graphes prédéfinis sont ajoutés : ils permettent d'ajouter un graphe cyclique, complet ou biparti en choisissant le nombre de sommets.

L'utilisateur peut agrandir/rétrécir (zoom/dézoom) le document ouvert et se déplacer dans celui-ci.

Paramètres

Les paramètres permettent d'activer/de désactiver les fonctionnalités suivantes :

- Coloration du graphe selon le nombre chromatique (exemple de coloration possible)
- Analyse automatique du graphe après chaque modification (non recommandée pour les graphes imposants)

RÉUTILISATION DE GRAPH++

Le but du projet est d'en faire une sorte de librairie réutilisable pour d'autres projets utilisant des graphes. Par exemple, un système de gestion de tâches avec des dépendances temporelles entre les tâches, qui utiliserait les graphes pour symboliser les dépendances.

Le projet est séparé en deux : la partie algorithmique (sous forme de librairie) doit donc rester en C++ pur et ne doit pas inclure de librairies supplémentaires ou de liaisons avec Qt.

REPRIORISATION

Les éléments décrits ci-dessus ont été décidés au début du projet. Après quelques semaines de développement, il a été décidé de prioriser les objectifs. En effet, en en apprenant plus sur les algorithmes de calcul qu'impliquaient certaines fonctionnalités, il a été décidé de prioriser les objectifs pour éviter d'implémenter des algorithmes *brute-force* complexes pour des fonctionnalités qui n'apportaient que peu de valeur. Par exemple, déterminer si un graphe est planaire ou non est un problème complexe. Pour trouver la solution, il faut réussir à déterminer si le graphe étudié contient l'un des deux graphes standards $K_{3,3}$ ou K_5, ce qui n'est pas possible directement avec un algorithme. Il faudrait décomposer le graphe étudié en tous les sous-graphes possibles, et vérifier si chacun de ces graphes est similaire à $K_{3,3}$ ou K_5. Cela représente une grande quantité de code, pour un algorithme peu performant et dont l'utilité pour l'utilisateur n'est pas remarquable. Les fonctionnalités avec un meilleur rapport faisabilité/valeur ajoutée ont donc été priorisées au détriment de ces algorithmes.

PLANIFICATION

Le projet a été planifié en utilisant un diagramme de Gantt. Le projet a été découpé en 6 grandes étapes ou jalons. Toutes les spécifications détaillées ont ensuite été exprimées sous forme de tâche (avec des tâches prérequises si nécessaire) et ont été associées à un jalon. Ces tâches ont ensuite été réparties sur une disposition chronologique du semestre pour estimer la durée du projet et les différentes dates clés.

JALONS

ID	Nom	Description	Date
01	Organisation	Organisation du projet et des tâches	27.02.2023
02	Conception	Conception du logiciel	
03	V0.1	Ajout des fonctionnalités primaires	
04	V0.2	Ajout des fonctionnalités avancées	
05	V1.0	Finition du projet	
06	Rendu	Rédaction de la documentation et rendu	

TÂCHES

ID	Nom	Durée [jour]	Prérequis
AA	Rédaction du Gantt	7	
AB	Rédaction du cahier des charges	7	
AC	Rédaction des spécifications détaillées	7	
01	**JALON 01 : ORGANISATION**		
AD	Réalisation du diagramme de classe	7	
AE	Réalisation d'une maquette	7	
AF	Mise en place d'issues, de jalons, etc. sur GitLab	7	
AG	Mise en place d'une pipeline CI sur GitLab	7	
AH	Réalisation d'un diagramme de séquence	7	
02	**JALON 02 : CONCEPTION**		
AI	Mise en place des modèles de données de base	7	
AJ	Mise en place de l'interface de base	14	
AK	Placements des sommets/arêtes	7	AI, AJ
AL	Export/import des graphes	7	AK
AM	Édition des sommets/arêtes	7	AK
AN	Analyse de base du graphe	21	AK
03	**JALON 03 : V0.1**		
AO	Navigation/Zoom du graphe	7	AK
AP	Nommage d'un sommet	3	AK
AQ	Ajout des graphes orientés	14	AK
AR	Export en png/jpg/etc.	14	AK

AS	Analyse avancée du graphe	21	AK
04	**JALON 04 : v0.2**		
AT	Debug	14	04
AU	Tests	7	AT
AV	Déploiement	7	AU
AW	Documentation du code	7	
05	**JALON 05 : v1.0**		
AX	Rédaction du manuel utilisateur	7	05
AY	Rédaction du manuel développeur	7	05
AZ	Rédaction du rapport	7	05
BA	Réalisation de la vidéo de démonstration	7	05
BB	Création de la présentation	7	05
06	**JALON 06 : CONCEPTION**		

Vous pouvez trouver une représentation complète de la timeline Gantt dans les annexes.

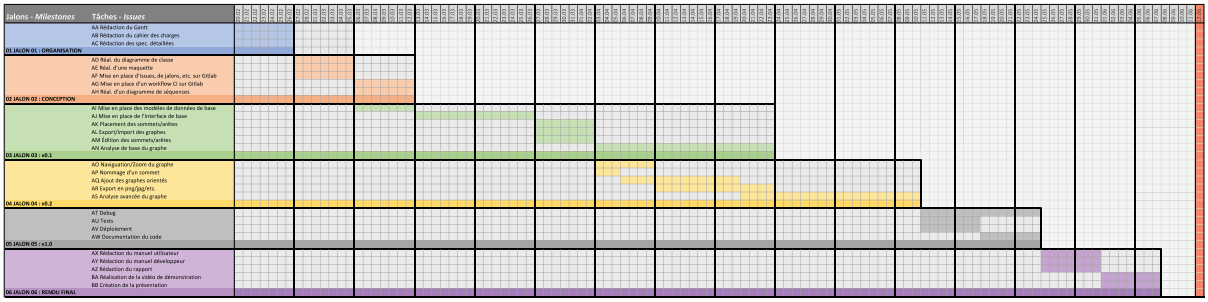


Figure 7 - Miniature de la timeline Gantt

CONCEPTION

MAQUETTE

Nous avons commencé pour réaliser une maquette visuelle de l'application. Cette maquette nous permettrait de facilement discuter sur les différentes fonctionnalités de l'application et d'avoir une vision commune du résultat à atteindre. Cette maquette a été réalisée sur Figma et est inspirée de différentes interfaces des logiciels de la suite Adobe comme Photoshop ou Illustrator.

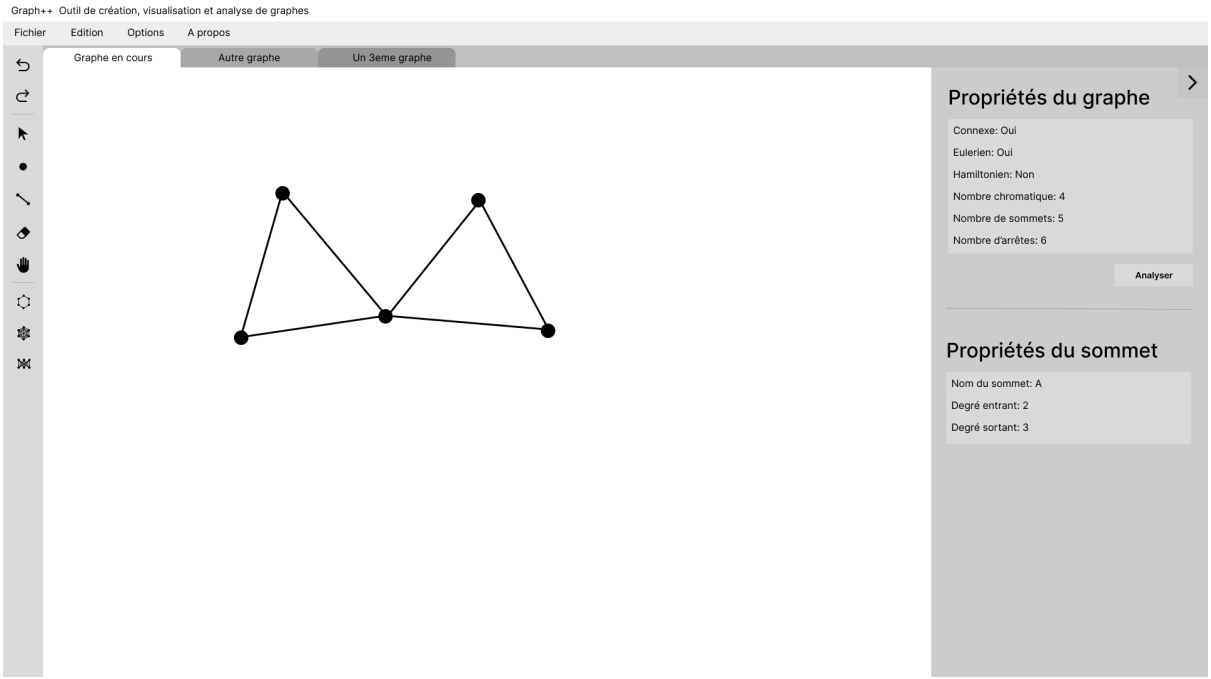


Figure 8 - Maquette de l'application

SCHÉMA DE CLASSES

Le schéma de classe a été un long sujet de débat au début du projet. Trouver une structure flexible et à la fois robuste permettant une réutilisation facile de la librairie a été un défi. Nous pensant cependant avoir surmonté ce défi avec cette structure qui offre de nombreuses possibilités. Nous n'avons décrit ici que les classes les plus importantes de l'interface et de la librairie. Ce diagramme ne correspond pas exactement au résultat final de l'implémentation, mais est présenté comme il a été imaginé au début du développement. Pour des informations à jour sur la structure du code, consultez la documentation Doxygen.

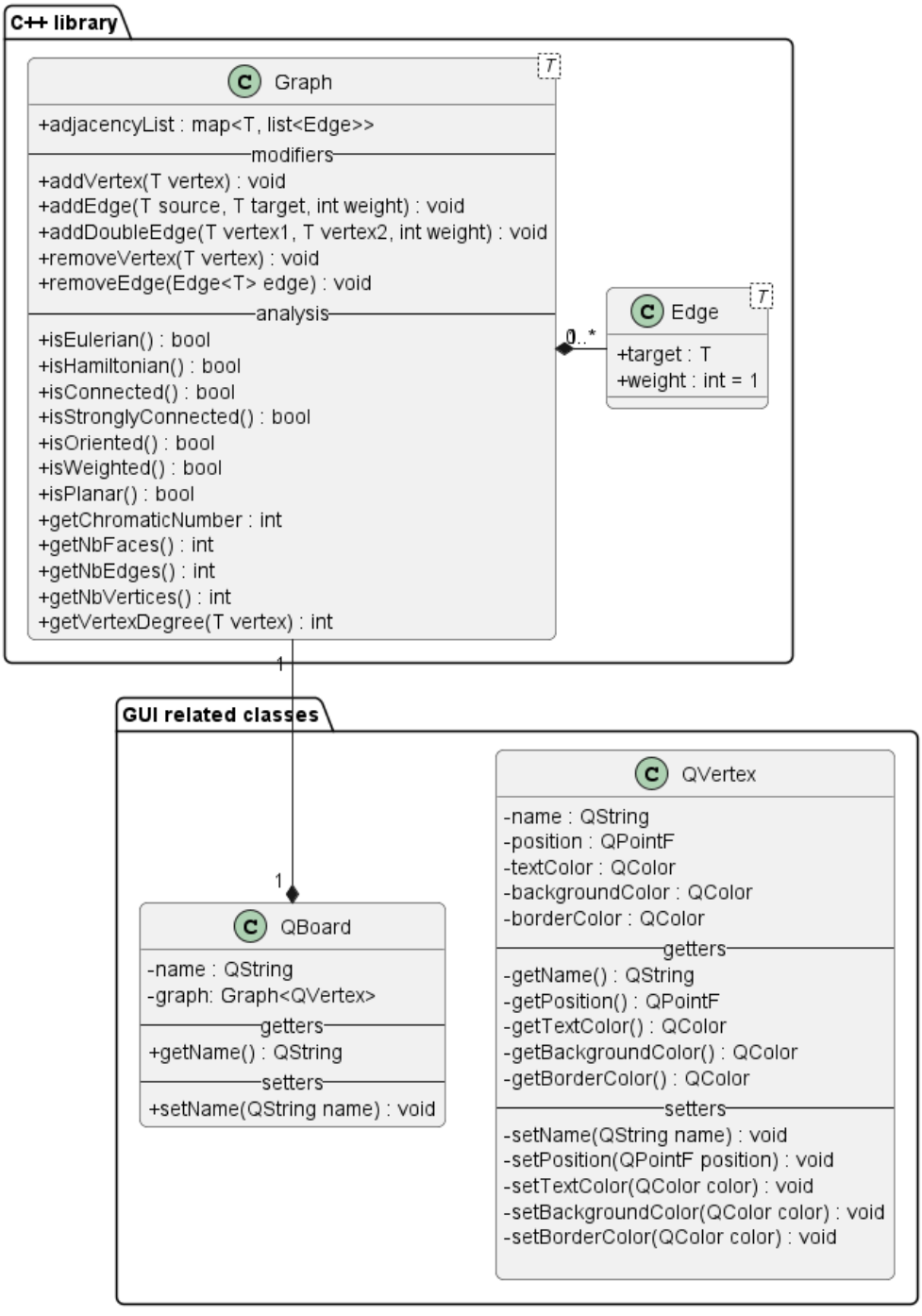


Figure 9 - Schéma de classe de l'application, comme prévu dans la phase de conception

DIAGRAMME DE CAS D'UTILISATION

Le diagramme de cas d'utilisation permet de définir les différentes possibilités offertes par le système et de définir quelle partie du projet en a la responsabilité. On peut voir que nous avions initialement pensé pouvoir faire la totalité de la sérialisation des données dans la librairie, qui n'a pas été finalement le cas.

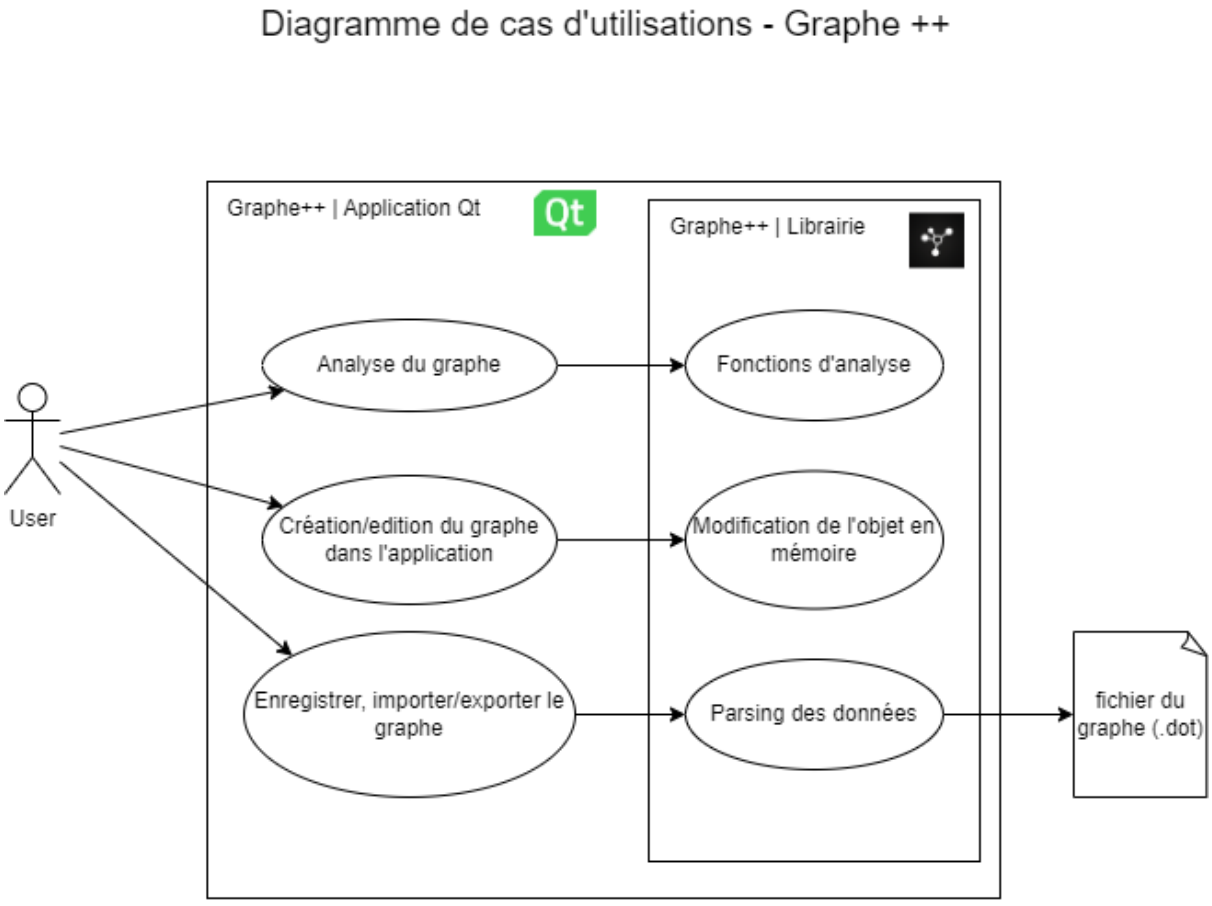


Figure 10 - Diagramme de cas d'utilisation du projet

IMPLÉMENTATION

Dans ce chapitre, nous allons expliquer les différents choix que nous avons réalisés et les choses particulières à noter concernant le développement de l'application. Cependant, nous n'allons pas rentrer dans le détail de chaque classe et/ou de chaque fonctionnalité. Pour cela, nous vous invitons à consulter le « Manuel Développeur » pour avoir plus d'informations techniques, ou encore à consulter la documentation Doxygen du code. Vous y trouverez les informations spécifiques à chaque classe et chaque méthode à travers l'application graphique, la librairie et aussi les batteries de tests unitaires.

STRUCTURE DU PROJET

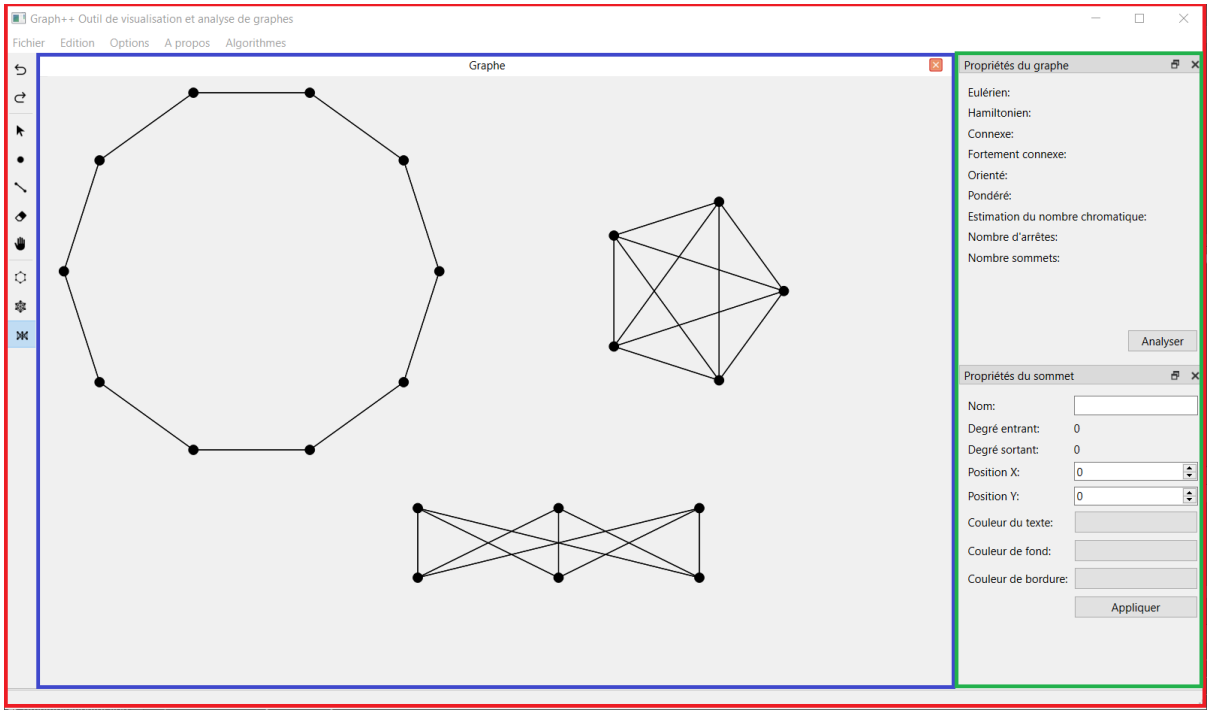
Initialement, tous les aspects du projet étaient regroupés dans un seul projet Qt. L'application graphique, la librairie et les tests. Ce système était le plus simple à compiler et évitait de devoir gérer des dépendances. Cependant après quelques réflexions, nous avons décidé de changer la structure du projet Qt. Le problème avec la structure initiale est qu'il sera plus difficile de garantir l'indépendance de la librairie. Si tous les fichiers se trouvent dans un même projet Qt, il est possible qu'un développeur utilise par accident des librairies du framework Qt dans la librairie, ce qui empêcherait de la réutiliser dans un autre projet non Qt, à moins d'importer toutes les librairies Qt. Nous avons donc décidé de tirer avantage du système des « subdirectory projects » de Qt pour créer une nouvelle structure pour le projet, qui permettrait de garder la librairie séparée de l'application et des tests unitaires. Cette structure nous permettrait aussi de compiler séparément chaque projet et de déployer uniquement les parties nécessaires. La structure est la suivante :

- **Graph++** : Un projet Qt de type « subdirectory » contenant les sous-projets distincts. En lançant la compilation dans ce dossier, tous les sous-projets sont compilés.
  - **Lib** : Un projet Qt de type « C++ Library ». C'est ici que la librairie est décrite. Les 2 autres sous-projets dépendent de cette librairie, signifiant qu'elle doit être toujours compilée en premier.
  - **App :** Un projet Qt de type « Qt Widgets application ». Ce projet contient l'application graphique desktop et dépend de la librairie. Si la compilation est lancée dans ce projet, seule l'application graphique est compilée (il faut avoir compilé la librairie au préalable).
  - **Tests :** Un projet Qt de type « subdirectory ». Ce projet contient les batteries de tests de la librairie et nécessite que la librairie soit compilée. Chaque batterie de tests est un sous-projet de type « autotest ».

APPLICATION GUI

STRUCTURE

L'application GUI se divise en trois parties principales :



1. La « MainWindow » (rouge) est la partie principale qui englobe toute l'application. C'est cet élément qui permet de créer une fenêtre avec des menus, outils, actions, toolTip, … C'est également cet élément qui gère la partie centrale, responsable de la création de plusieurs onglets (MDI, Multiple Document Interface)
2. Le « QBoard » (bleu) est le widget qui s'occupe de la zone de dessin. C'est ce widget qui sera dupliqué lorsque l'utilisateur créera plusieurs graphes dans plusieurs onglets différents. Cet élément possède tout ce qui est nécessaire à la création, la modification ou à la suppression d'un ou plusieurs éléments du graph.
3. « L'overlay » (vert) est la partie qui contient deux widgets, celui qui affiche l'analyse du graph et celui qui affiche les informations du sommet connecté. Cette partie de l'application appartient, tout comme le QBoard, à la MainWindow. Ce qui a pour conséquence qu'il faut que ces docks doivent avoir accès au graphe actuellement sélectionné pour venir récupérer les informations nécessaires. Pour donner cet accès, chaque dock possède un pointeur vers le graphe sélectionné. Lorsque l'utilisateur change d'onglet (donc de graphe), nous mettons à jour ce pointeur.

SUPPRESSION DES ÉLÉMENTS EN MÉMOIRE

Pour commencer, voici peu de contexte comment la fonctionnalité d'annulation et rétablissement (undo/redo) fonctionne. Premièrement il y a plusieurs moyens

d'implémenter cette fonctionnalité selon nos besoins. Nous avons fait le choix d'implémenter un memento qui garde en mémoire la liste d'adjacence de tous les sommets et arêtes du graphe. À noter que c'est uniquement la liste qui est copiée et non les objets « vertex » et « edge » en mémoire. Cela a pour avantage que l'on manipule uniquement des pointeurs.

Nous avons choisi cette solution, car elle était premièrement plus facile et rapide à implémenter (dû au fait que nous l'avions déjà expérimentée pour un autre projet), mais surtout, car le modèle Command n'était pas adapté à nos actions.

Le mémento fonctionne de la manière suivante. Juste avant que l'utilisateur créer un élément du graph, nous venons sauvegarder l'état du graphe actuel dans une pile (undoStack). Lorsque l'utilisateur souhaite annuler sa dernière modification, l'état du graphe est rétabli avec le dernier élément de la pile. Mais avant de supprimer le dernier élément, nous venons le sauvegarder dans la pile de rétablissement (redoStack). Grâce à cela, l'utilisateur peut en tout temps rétablir la modification qu'il vient d'annuler. La redoStack sera vidée lorsque l'utilisateur effectuera une nouvelle modification (dans ce cas-là, il n'y a plus rien à rétablir).

Et c'est à ce moment-là que le problème intervient. Car c'est lors du vidage de cette pile que les objets en mémoire doivent être supprimés. Mais le memento contient la liste d'adjacence de tous les éléments du graphe. Donc nous avons dû comparer les deux listes d'adjacence et supprimer uniquement les éléments présents dans l'une, mais pas dans l'autre.

Même si cette solution est relativement simple, elle nécessite tout de même de parcourir deux listes d'adjacence à la recherche des différences.

PERFORMANCES ET RÉACTIVITÉ

Nous avons effectué quelques tests de performance afin de voir la réactivité de notre application. Dans notre cas d'implémentation de graphes, il faut savoir que l'opération la plus gourmande en ressource est la suppression d'éléments et notamment d'arête. L'outil gomme fonctionne de la manière suivante :

- À chaque mouvement de souris, l'application va récupérer les coordonnées de la souris et va détecter tous les éléments dans un rayon défini autour de la position.
- Puis, la liste d'adjacence va être parcouru et comparer si la position sommets correspondant à la zone d'effacement. Si c'est le cas, le sommet sera enlevé du graphe.
- De même pour les arêtes, on va vérifier si les arêtes passent à travers cette zone d'effacement et si c'est le cas, l'arête également évincée.

Le problème est que cette procédure sera suivie à chaque déplacement de la gomme, autrement dit, plusieurs fois par secondes. Donc lorsque le graphe est conséquent et

que la liste d'adjacence sera parcourue et testée plusieurs fois par seconde, cela peut prendre du temps.

De plus, avec les fonctionnalités d'annulation et de rétablissement, les sommets et arêtes ne sont pas supprimés en mémoire dès leur disparition du graphe. Il faut garder une trace de ces derniers dans le cas où l'utilisateur voudrait revenir en arrière.

A cause ce nombre important de calculs, nous avons dû trouver des solutions pour améliorer les performances de notre application. Notamment le « box bounding » qui est une technique limitant les calculs sur toute la liste. Voici comment il fonctionne :

- Lors du parcours de chaque arête de la liste d'adjacence, l'application va déjà détecter si les coordonnées de la souris se situent dans le rectangle formé avec les deux sommets de l'arête.
- Si c'est le cas, la position exacte de l'arête sera comparée plus précisément pour savoir s'il faut ou non la supprimer.
- Si ce n'est pas le cas, on peut directement passer au test suivant.

Mine de rien, cette méthode limite grandement les calculs effectués et augmente donc les performances.

Malgré cette amélioration majeure, nous avons tout de même dû limiter la taille des graphes complets à 30 sommets maximum. Premièrement, car au-delà de 30 sommets, le graphe n'est plus tellement pratique à utiliser, mais surtout, car ce type de graphe pose de gros problèmes de performances s'il possède trop de sommets.

LIBRAIRIE

STRUCTURE DE DONNÉES

La structure de base de la librairie est composée de deux classes. Une première classe *Graph* pouvant contenir un graphe et la classe *Edge* représentant un arc entre deux sommets. Ces deux classes permettent de représenter la structure d'un graphe complet. Leur particularité principale de ces classes est qu'elles sont en pratique des *templates C++*[5]. Ce qui signifie qu'à l'aide de la librairie, il est possible d'agencer n'importe quelle classe sous la forme d'un graphe. Par exemple, si votre application permet de gérer des données sur des villes et villages avec une classe *City* et que vous désirez calculer un graphe des chemins les plus courts entre vos villes, vous pouvez simplement insérer des instances de la classe *City* dans un *Graph<City>* et utiliser les méthodes de la librairie. En arrière-plan, la classe *Graph* utilise une map pour stocker la structure du graphe sous forme d'une liste d'adjacence. Les clés de la map sont des pointeurs sur une instance de T, et la valeur de la map est une liste d'*Edge<T>*. Quant à elle, la classe *Edge<T>* ne contient qu'une référence vers l'instance T cible ainsi que le poids de l'arête. Cette structure a été créée avec en tête le besoin de fortement découpler la librairie de l'application graphique. En effet, les

[5] Voir https://www.geeksforgeeks.org/templates-cpp/

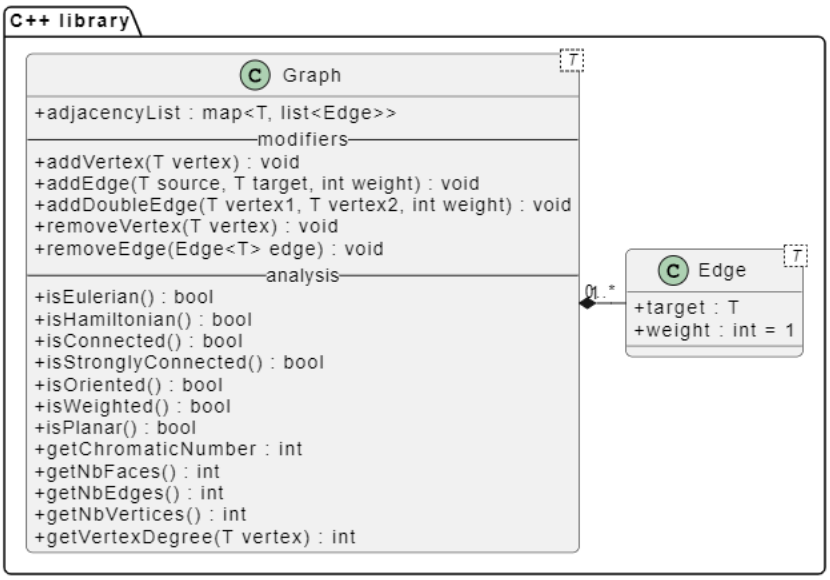


Figure 11 - Schéma de classe de la librairie, comme prévu dans la phase de conception

sommets de l'application graphique possèdent certaines propriétés comme une couleur ou un nom, mais nous ne voulions pas imposer une structure de données possédant des attributs et des méthodes qui ne serait pas utile dans un autre projet. D'ailleurs, la structure de la librairie ne fait aucune distinction entre un graphe orienté et non orienté. En pratique, dans le code, un graphe non orienté et simplement un graphe orienté dont toutes les arêtes ont une arête allant dans le sens inverse. Cette structure est très flexible, mais implique quelques complications. Premièrement, il n'est pas possible de savoir quel type de classe sera utilisé dans le graphe, il est donc impossible d'utiliser des méthodes propres à un sommet. Par exemple, la librairie n'est pas capable de sérialiser un graphe avec le contenu de chaque sommet, car elle ne sait pas si les sommets sont sérialisables. Il en revient donc à la librairie graphique qui elle, connaît le contenu d'un sommet, de sérialiser le graphe. Un autre problème est avec la classe *Edge<T>.* Comme indiqué sur le schéma, celle-ci ne contient que des informations sur la cible de l'arc et le poids de l'arc, mais aucune information sur la source de l'arc. C'est le cas, car théoriquement, cette information est donnée par la clé de l'entrée dans la liste d'adjacence. Pratiquement, cela pose un problème lorsque l'on veut, pour un arc donné, retrouver la source de cette arc. Il faut parcourir la liste et trouver la source correspondante. Dans plusieurs algorithmes il est nécessaire de retrouver la source d'un arc à chaque itération de l'algorithme, ce qui demande un grand nombre d'instructions supplémentaires. Un simple ajout d'un attribut à la classe *Edge<T>* aurait pu régler ce problème, mais les algorithmes utilisant déjà la méthode de recherche, il aurait fallu réadapter plusieurs algorithmes. Un autre problème est l'itération sur les sommets du graphe. Par exemple, l'application graphique a souvent besoin de pouvoir itérer sur les sommets pour les dessiner à l'écran ou les modifier. Cependant, nous n'avons pas eu le temps d'implémenter des itérateurs dans le graphe qui permettrait de parcourir la liste d'adjacence, ce qui nous a contraints à laisser la liste d'adjacence comme attribut

public. Cette décision a été malheureusement faite par manque de temps et de meilleures solutions. Deux solutions possibles seraient de rendre la liste privée et d'obliger les développeurs à maintenir une autre structure de données comme un tableau ou une liste pour contenir leurs sommets et leurs arcs. Le graphe ne servirait alors qu'à contenir les liens entre les données, mais pas les données elles-mêmes. Cela aurait demandé d'adapter le fonctionnement de l'application graphique et n'était donc pas envisageable. Une autre solution serait d'implémenter des itérateurs C++ qui permettent de parcourir le graphe de différente manière, en utilisant par exemple différente méthode de parcours (*Depth-first, Breadth-first, Prim, Dijkstra*) et de donner accès aux sommets et arcs de cette manière. Cette solution serait assez robuste et rajouterait une bonne valeur à la librairie. Cependant nous n'avions pas prévu d'implémenter de telles fonctionnalités au début du projet et cela aurait retardé le développement de nombreuses autres fonctionnalités. Nous avons donc décidé de laisser la structure telle quelle, en sachant que cela représentait une grande faiblesse qu'il faudrait impérativement aborder si l'on voulait mettre la librairie à disposition pour d'autres développeurs.

MÉTHODES

La classe *Graph<T>* possède d'abord un certain nombre de méthodes simples. Ces méthodes sont les outils disponibles pour créer un graphe et en déterminer la structure de base. On retrouve des méthodes pour ajouter des sommets, ajouter des arcs, ajouter des arcs bidirectionnels, supprimer des arcs et des sommets ou encore déterminer des propriétés basiques comme la taille du graphe, s'il est vide, s'il est orienté ou encore s'il est connexe. Ces méthodes peuvent être appelées par l'utilisateur et sont surtout utiles pour déterminer la marche à suivre concernant un graphe. Certains algorithmes fonctionnent différemment ou peuvent être simplifiés s'ils savent qu'un graphe est connexe ou orienté. Toutes ces méthodes sont fonctionnelles, mais pourraient être améliorées avec une meilleure sécurité à l'utilisation et de légères optimisations. Par exemple, il serait utile de développer un système de cache où les propriétés de du graphe ne serait pas recalculée si celui-ci n'a pas été modifié depuis le dernier calcul. De cette manière, quand un algorithme utilise à répétition une propriété comme l'orientation du graphe, il ne serait pas nécessaire de vérifier la totalité des arcs.

ALGORITHMES

La librairie propose 3 algorithmes principaux :

- **getMinimumSpanningTree** : détermine l'arbre recouvrant de poids minimal du graphe.
- **getMinimumDistanceGraph** : détermine les chemins les plus courts pour accéder à tous les sommets à partir d'un sommet de départ.
- **getHamiltonianPath** : détermine un cycle hamiltonien du graphe.

Ces trois méthodes ont de nombreuses applications dans de nombreux domaines et ajoutent une bonne valeur à la librairie. Décrivons leur implémentation.

Arbre recouvrant de poids minimal

L'arbre recouvrant de poids minimal est un sous-graphe pouvant être représenté comme un arbre et dont la somme des arêtes est la plus faible possible. En d'autres termes, l'arbre recouvrant de poids minimal indique les liaisons minimales pour connecter tous les sommets ensemble. L'algorithme utilisé pour déterminer ce graphe est l'algorithme de Prim.

L'algorithme est un algorithme du parcours du graphe, où, à chaque sommet, le sommet ayant la connexion la moins lourde est visité. L'algorithme ne va donc visiter un sommet que si l'arête qui l'atteint possède le coût le plus faible de ceux déjà rencontrés. L'algorithme va donc entretenir une file de priorité de chaque sommet rencontré et où la priorité correspond au poids de l'arête permettant d'atteindre ce sommet. L'algorithme va ensuite visiter le sommet le plus prioritaire, mettre à jour sa file de priorité avec les nouvelles informations obtenues en visitant le sommet et répéter cette procédure jusqu'à avoir visité tous les sommets. L'algorithme va donc se déplacer dans le graphe et prendre la meilleure option au moment donné, ce qui le qualifie d'algorithme *greedy.*

Graphe des chemins les plus courts

Le graphe des chemins les plus court est un sous-graphe qui indique, selon un sommet donné, les chemins les plus courts pour atteindre chaque sommet, c'est-à-dire un algorithme de *path-finding*. Il existe plusieurs algorithmes de ce type très connus et celui implémenté dans notre cas est l'algorithme de Dijkstra.

L'algorithme de Dijkstra est un algorithme du parcours du graphe, où, à chaque sommet, le sommet ayant le chemin d'accès le moins lourd est visité. L'algorithme va donc visiter les sommets où la somme des arêtes pour y accéder est la plus faible. L'algorithme va donc entretenir une file de priorité de chaque sommet rencontré et où la priorité correspond à la somme du poids des arêtes parcourues pour atteindre ce sommet depuis le sommet de départ. L'algorithme va ensuite visiter le sommet le plus prioritaire, mettre à jour sa file de priorité avec les nouvelles informations obtenues en visitant le sommet et répéter cette procédure jusqu'à avoir visité tous les sommets.

Cycle hamiltonien

Un cycle hamiltonien est un cycle, c'est-à-dire un chemin dont le sommet de départ est le même que le sommet d'arrivée, passant par tous les sommets d'un graphe.

Il n'existe pas d'algorithme défini pour déterminer un cycle hamiltonien dans un graphe. La seule possibilité pour cette fonctionnalité est d'implémenter un *brute-force*. Évidemment, l'algorithme brute-force est extrêmement couteux, car il va tenter de composer tous les chemins possibles à l'intérieur d'un graphe et va devoir vérifier si ce chemin est un cycle et s'il est hamiltonien. Cependant, il est possible de prendre

quelques « raccourcis » et d'éviter des tester des possibilités qui seraient absurdes et d'ainsi gagner en efficacité. L'algorithme reste cependant très peu optimal, sachant que la totalité des simplifications possibles n'a pas été mise en place, car elles auraient demandé une grande quantité de code supplémentaire, que le temps commençait à manquer et que la méthode était déjà fonctionnelle dans l'état actuel.

L'algorithme brute-force amélioré utilisé en partie dans le projet est défini dans un document publié par IBM en 1974[6]. Il définit plusieurs règles qui classent les arêtes selon leur état nécessaire pour que le chemin soit acceptable. À chaque étape d'ajout d'un sommet à la fin du chemin, chaque arête est classée comme *Indéfinie* (U), *Nécessaire* (R) ou *Supprimée* (D). Ce classement est propre à l'état dans lequel l'algorithme se trouve : lorsqu'un sommet est ajouté au chemin partiel, toutes les arêtes doivent être reclassées, mais tous les états précédents sont conservés, car ils doivent pouvoir être restaurés si le chemin produit ne mène nulle part. D'autres règles de l'algorithme permettent d'invalider immédiatement le chemin trouvé ou de définir le sens dans lequel les arêtes sont parcourues.

À chaque état, si le chemin est valide jusque-là, un sommet y est ajouté. S'il n'est pas valide, un sommet est retiré et l'algorithme essaie avec un autre sommet. Si aucun sommet ajouté ne pouvait mener à un chemin valide, le chemin actuel est déclaré invalide. Si tous les chemins sont invalides, alors aucun chemin hamiltonien ne peut être trouvé dans le graphe. À l'inverse lorsque l'algorithme trouve un chemin qui passe par tous les sommets et possède comme point d'arrivée le point de départ, un chemin a été trouvé.

Il est important de noter que la complexité de l'algorithme est O(n!), ce qui l'une des pires si ce n'est la pire complexité obtenable. Pour un graphe de 10 sommets, les temps de calcul restent acceptables pour un utilisateur patient, mais augmentent donc rapidement avec le nombre de sommets. Plus le graphe est faiblement connexe, plus l'algorithme prend de temps, ce qui s'explique par le fait qu'il doit essayer beaucoup plus de possibilités pour trouver un chemin valide.

[6] (A Search Procedure for Hamilton Paths and Circuits, 1974)

GÉNIE LOGICIEL

QMAKE

Le projet est configuré avec QMake, l'outil de compilation du framework Qt. Bien que la chaîne de compilation soit fonctionnelle, nous ne referions pas ce choix. QMake est un système propre à Qt et ne possède pas énormément d'utilisateurs, ce qui rend les problèmes de *build* et de dépendances difficiles à résoudre. De plus, QMake n'est plus recommandé et maintenu par les développeurs. Sachant que le projet est développé en C++, il serait plus simple d'utiliser un outil similaire et reconnu comme CMake. CMake est devenu au fil des années un standard du développement C++ et permet énormément de possibilités en termes de compilation et de gestion du code. Malheureusement, le projet était trop avancé pour pouvoir effectuer la migration sans perdre un temps considérable sur la période d'implémentation. Il a donc été décidé de rester sur QMake. La structure du projet, comme décrite dans la section « Structure du projet »[7], est la suivante :

- **Graph++** : Un projet Qt de type « subdirectory » contenant les sous-projets distincts. En lançant la compilation dans ce dossier, tous les sous-projets sous compilés.
  - **Lib** : Un projet Qt de type « C++ Library ». C'est ici que la librairie est décrite. Les 2 autres sous-projets dépendent de cette librairie, signifiant qu'elle doit être toujours compilée en premier.
  - **App :** Un projet Qt de type « Qt Widgets application ». Ce projet contient l'application graphique desktop et dépend de la librairie. Si la compilation est lancée dans ce projet, seule l'application graphique est compilée (il faut avoir compilé la librairie au préalable).
  - **Tests :** Un projet Qt de type « subdirectory ». Ce projet contient les batteries de tests de la librairie et nécessite que la librairie soit compilée. Chaque batterie de tests est un sous-projet de type « autotest ».

QTESTS

Les tests unitaires sont rédigés avec le framework Qt. Le framework possède sa propre librairie de tests unitaires nommée « Qt Tests ». Tout comme pour QMake, nous ne recommanderions pas cette librairie. Puisqu'elle est propre à Qt, très peu d'exemples sont disponibles en dehors de la documentation officielle. Les analyseurs statiques détectent aussi des erreurs à cause de la syntaxe spécifique de ces tests (fichiers moc introuvables notamment). Une meilleure option aurait été d'utiliser les Google tests, qui sont bien plus répandus et bien intégrés à l'IDE de Qt. C'est même l'option par défaut lorsque l'on crée un test unitaire à travers celui-ci. Les tests unitaires n'ont été

[7] Voir page 18

rédigés que pour la librairie et se concentrent sur la justesse des différents algorithmes.

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 sommets sont les mêmes et que les poids totaux des deux graphes sont égaux, le résultat est considéré comme correct.

Dans tous les tests unitaires, les graphes sont construits « manuellement », c'est-à-dire en ajoutant chaque sommet et chaque arc avec une instruction, soit séquentiellement soit à l'intérieur d'une boucle. Cette approche est limitée, car elle ne permet pas de générer de grands graphes avec des structures quelconques. Soit l'on décrit la structure du graphe arc par arc et le code devient rapidement illisible, soit l'on utilise une boucle et la structure du graphe est normalisée (graphe cyclique, complet, etc.). Une amélioration serait d'utiliser les fonctionnalités de sérialisation des graphes pour enregistrer dans des fichiers les graphes de bases et les graphes attendus pour les tests unitaires. De cette manière, les tests dont le seul but est de tester un algorithme pourraient simplement charger les graphes depuis des fichiers, dérouler les algorithmes et comparer les résultats, sans devoir créer chaque graphe « manuellement ». Les graphes testés pourraient être de taille et de complexité variantes, testant différentes orientations et pondérations, ce qui permettrait de s'assurer que les algorithmes fonctionnent dans tous les cas et pas seulement dans le cas d'un graphe cyclique ou complet. Cette solution n'a pas été mise en place, car les méthodes de sérialisation ont été mises en place vers la fin du projet et la nécessité de tests unitaires était présente dès le début du projet. Les tests ont donc été développés avec les méthodes de base de la librairie.

DOXYGEN

Doxygen est l'outil standard de facto pour générer de la documentation à partir de sources C++ annotées, mais il prend également en charge d'autres langages de programmation populaires tels que C, Objective-C, C#, PHP, Java, Python et d'autres[8].

Pour pouvoir générer une documentation Doxygen, nous avons commenté la majorité de notre code, avec un style similaire à la Javadoc[9]. Ces commentaires donnent des

[8] (Heesch, 2023)
[9] Voir https://www.oracle.com/ch-fr/technical-resources/articles/java/javadoc-tool.html

descriptions des classes, de leurs méthodes et de leurs attributs. À partir du code, Doxygen est capable de générer une documentation écrite et une documentation web, mettant en avant toutes les informations trouvées dans le code. Un fichier de configuration se trouve à la racine de notre projet et permet une personnalisation poussée du processus de documentation. La version actuelle de notre documentation est simple et utilise la configuration par défaut à quelques détails près.

Grâce à ces commentaires, les développeurs utilisant des IDE modernes pourront voir les définitions des classes et méthodes avec un simple survol de la souris et si une information n'est pas suffisante, ils peuvent se référer à la documentation complète Doxygen. Pour garder une régularité entre le code et la documentation, les commentaires ont été rédigés en anglais. De cette manière, les noms de méthodes et de classes correspondent aux termes utilisés dans la documentation et celle-ci est utilisable par des développeurs non francophones.

Vous pourrez d'ailleurs trouver la documentation écrite du code dans les annexes de ce document.

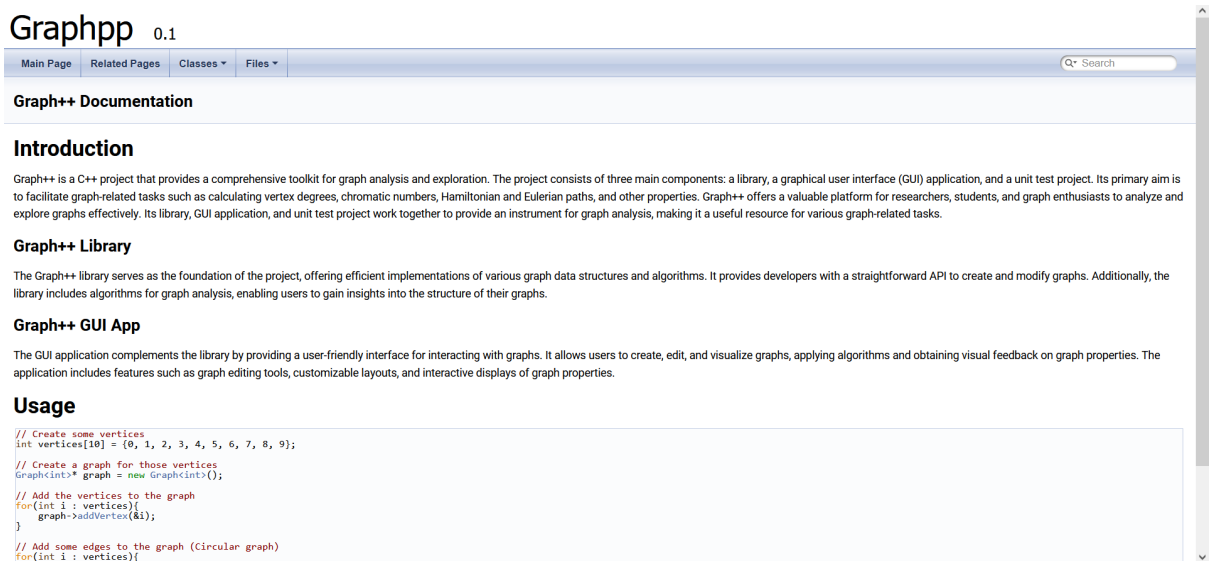


Figure 12 - Page d'accueil de la documentation web

GITLAB

GitLab Inc. est une entreprise open-core qui exploite GitLab, un package logiciel DevOps qui permet de développer, sécuriser et exploiter des logiciels. Le projet de logiciel open source a été créé par le développeur ukrainien Dmytro Zaporozhets et le développeur néerlandais Sytse Sijbrandij. En 2018, GitLab Inc. a été considérée comme la première "licorne" en partie ukrainienne[10].

Le projet a été développé en utilisant GitLab comme plateforme de DevOps. Le but étant de pouvoir collaborer sur une base de code commune, sans conflits et avec un

[10] (Contributors to Wikimedia projects, 2014)

système de tâches et d'objectifs à remplir pour toute l'équipe. Pour ce faire, GitLab propose différents outils de gestion de projet et de code. Voici les différents éléments mis en place :

- **Milestones** : Les milestones (ou jalons) sont des étapes décisives dans le déroulement du projet. Plusieurs jalons ont été créé pour représenter les phases d'analyse, de conception, d'implémentation, mais aussi pour les différentes versions de l'application. GitLab permet ensuite de réutiliser ces jalons dans ses autres systèmes.
- **Issues** : Une issue est une tâche à remplir. Une analyse à effectuer, une fonctionnalité à ajouter ou un bug à corriger. Toutes les tâches à effectuer pour le projet ont été documentées et ajoutées dans la liste d'issues du projet GitLab. Il est possible d'associer à ces issues des jalons pour les lier à des étapes du projet. Il est aussi possible d'assigner des développeurs aux issues, d'indiquer le poids de l'issue ou de quel type il s'agit (bug, fonctionnalité, documentation, etc.).

GIT WORKFLOW

Pour éviter les conflits et pouvoir collaborer efficacement, une méthodologie Git à été mise en place pour les développeurs de l'équipe. Le but de cette méthodologie est de pouvoir permettre le travail en parallèle, d'éviter les conflits et de pouvoir travailler de manière autonome, sans avoir à consulter régulièrement les membres de l'équipe. Cette méthodologie est reprise du cours 2245.2 Génie logiciel II enseigné par M. Le Callennec. Voici comment la marche à suivre se présente :

1. CHOISIR UNE ISSUE SUR GITLAB

Premièrement, sélectionner une issue dans la liste présente en ligne sur GitLab. L'issue doit être libre et ne doit pas avoir d'issues associées à des jalons plus pressants. De cette manière, le travail le plus urgent est effectué en premier et le travail n'est pas fait à double. Une fois l'issue sélectionnée, s'assigner à l'issue pour que les prochains développeurs puissent voir que vous travaillez déjà sur cette tâche.

2. CRÉER LA BRANCHE DE TRAVAIL

Dans son environnement de travail, sélectionner la branche principale et la mettre à jour. Ensuite, créer la branche avec l'acronyme du développeur, le numéro et le nom de l'issue choisie.

```
git checkout main
git pull --rebase
git checkout -b <Dev>_<Id>_<Summary>
```

3. TRAVAILLER SUR LA BRANCHE

Développer les fonctionnalités décrites dans l'issue, commenter et tester le code, puis appliquer ces modifications sur sa branche. Si un bug est rencontré, mais n'est pas lié

à l'issue en cours, tout de suite créer une issue sur GitLab décrivant le bug et comment le reproduire, puis continuer le travail propre à la tâche.

4. FUSIONNER LES BRANCHES

Une fois le travail terminé, ajouter le travail de la branche de développement à la branche principale.

```
git fetch
git checkout main
git pull
git checkout <Dev>_<Id>_<Summary>
git rebase main
rem Conflits ? Résoudre, puis "git rebase --continue", ou "git rebase --abort" pour annuler
rem Pour éviter les conflits, il faut rebase souvent (dès que quelque chose fonctionne / toutes les heures)
git checkout main
git merge --ff-only <Dev>_<Id>_<Summary>
```

CI/CD

L'approche CI/CD permet d'augmenter la fréquence de distribution des applications grâce à l'introduction de l'automatisation au niveau des étapes de développement des applications. Les principaux concepts liés à l'approche CI/CD sont l'intégration continue, la distribution continue et le déploiement continu. L'approche CI/CD représente une solution aux problèmes posés par l'intégration de nouveaux segments de code pour les équipes de développement et d'exploitation (ce qu'on appelle en anglais « integration hell », ou l'enfer de l'intégration).

Plus précisément, l'approche CI/CD garantit une automatisation et une surveillance continues tout au long du cycle de vie des applications, des phases d'intégration et de test jusqu'à la distribution et au déploiement. Ensemble, ces pratiques sont souvent désignées par l'expression « pipeline CI/CD » et elles reposent sur une collaboration agile entre les équipes de développement et d'exploitation, que ce soit dans le cadre d'une approche DevOps ou d'ingénierie de la fiabilité des sites (SRE)[11].

Dans le cadre du projet Graph++, l'approche CI/CD a été appliquée pour permettre un contrôle continu de l'état de l'application et une génération automatique de la documentation. Pour ce faire, plusieurs tâches sont effectuées automatiquement à chaque modification du *repository* en ligne :

- **Build** : Compile de l'application et permet de détecter les erreurs les plus basiques dans le code et dans les fichiers de configuration du projet.

[11] (Red Hat, Inc., 2022)

- **Test** : Exécute tous les tests unitaires et exportent les résultats au format JUnit pour la visualisation dans GitLab. Chaque test unitaire est lancé, chronométré et enregistré pour permettre de visualiser le taux de réussite des tests directement dans l'interface web de GitLab.
- **Documentation** : Génère la documentation Doxygen et l'enregistre sur les serveurs GitLab. De cette manière, il est possible pour un développeur d'aller télécharger la documentation correspondant à chaque *push* effectué sur les différentes branches GitLab.

Plusieurs problèmes ont été rencontrés lors de la mise en place de cette infrastructure. Premièrement, il a fallu créer une image *Docker* personnalisée contenant les différents outils de compilations puisque Qt ne propose aucune image prête à l'utilisation. QMake, l'outil de compilation utilisé, n'étant plus supporté, il a été difficile de trouver des procédures d'installations pour Linux. Ce n'est qu'après plusieurs semaines de tests et une mise à jour du serveur Docker de l'école que l'image a pu être utilisée sans problèmes. À nouveau, l'utilisation de CMake aurait simplifié la tâche. Ensuite, il a été remarqué que les différents chemins de fichiers utilisés dans la configuration QMake (fichiers .pro) pour indiquer les dépendances aux librairies n'étaient pas les mêmes lors de la compilation sur Linux. Il a donc fallu modifier la pipeline pour qu'elle corrige les fichiers avant de lancer la compilation. Finalement, il a été impossible de compiler la documentation LaTeX directement dans la pipeline. En effet, la commande permettant d'installer l'outil de compilation LaTeX contient plusieurs prompts interactifs impossibles à désactiver, ce qui l'empêche d'être installé automatiquement lors de la construction de l'image Docker. Bien que la majorité d'entre eux aient été corrigés, ces problèmes ont malheureusement fait que la pipeline CI/CD n'a été opérationnelle que plusieurs semaines après le début du projet.

CONCLUSION

RÉSULTATS

Au terme de 6 mois de développement, l'application Graph++ contient une interface graphique développée en C++ à l'aide du *Framework Qt* utilisant une librairie d'analyse de graphe développée par nos soins en C++ standard.

L'application permet d'ajouter, modifier, supprimer des sommets. Elle permet de relier deux sommets pour créer une arête entres les deux.

Il est possible de supprimer un sommet ou une arête à l'aide de l'outil gomme.

L'application possède trois outils de création de graphes remarquables tels que le graphe cyclique, le graphe complet et le graphe biparti complet.

Le graphe réalisé peut être enregistré en JSON, au format DOT (Graphviz) ou exporter en image PNG.

Un graphe enregistré au format JSON peut être à nouveau importé dans l'application pour continuer son travail.

L'application permet d'analyser le graphe et d'indiquer les propriétés de ce dernier.

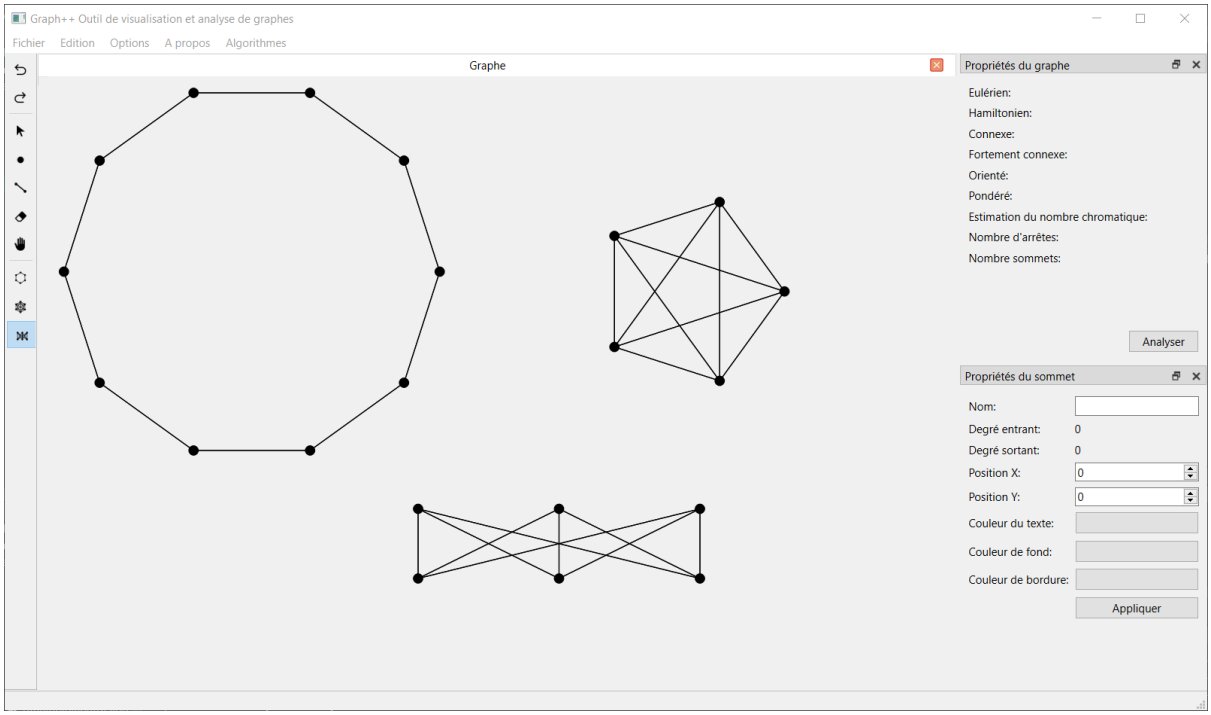


Figure 13 - L'application avec quelques graphes remarquables dessinés

LIMITES

Durant le déroulement du projet, nous avons décidé de prioriser quelques objectifs pour différentes raisons :

- La possibilité de choisir de créer des graphes orientés représentait beaucoup de travail et trop de changements par rapport au temps restant. La librairie implémente cette fonctionnalité, mais pas l'application Qt.
- Le déplacement dans l'espace de dessin et le zoom ont représenté une contrainte technique et nous avons sous-estimé le temps pour faire fonctionner ces fonctionnalités ensemble. Pour des raisons de simplicité, nous avons gardé la fonctionnalité de déplacement jugée plus utile, et désactivé le zoom. Ce déplacement créer cependant quelques tremblements.
- Certaines fonctions d'analyse telles que la détection d'un graphe planaire ou l'analyse de sous-graphe posaient de vrais problèmes techniques, car les algorithmes utilisés en mathématique n'étaient pas adaptés à une implémentation code. Nous avons décidé d'abandonner ces fonctionnalités.

Si nous avons laissé de côté ces objectifs-là, c'était pour avant tout présenter un projet abouti avec certes moins de fonctionnalités initialement prévues, mais avec des fonctionnalités abouties et utiles.

PERSPECTIVES

Concernant les perspectives du projet, nous pouvons tout d'abord citer les éléments du chapitre « Limites » qui pourront être implémentés/corrigés.

En parallèle, voici quelques autres idées qui permettraient d'améliorer l'interface graphe++, mais également la librairie c++.

- Actuellement, il est possible de voir les chemins les plus courts à partir d'un sommet sélectionné vers tous les autres sommets de sa composante connexe. Une voie d'amélioration serait de montrer le chemin le plus court entres deux sommets sélectionnés.
- Améliorer l'algorithme d'estimation du nombre chromatique qui n'est pas fiable à 100%.
- Colorier le graphe selon les couleurs utilisées pour l'estimation du nombre chromatique
- Possibilité de sélectionner un arc du graphe pour en changer ses propriétés (couleur, épaisseur, pondération …)
- Sélectionner plusieurs sommets et arêtes à l'aide d'une zone de sélection
- Ajouter une fonctionnalité de couper/copier/coller
- Déplacer les éléments sélectionnés, les agrandir ou rétrécir
- Redessiner un graphe planaire de manière qu'il soit visiblement planaire (modifier les tracés des arêtes pour mettre en évidence cette caractéristique).
- Ajouter des thèmes clairs/sombres
- Améliorer l'expérience utilisateur. Par exemple enregistrer périodiquement le travail de l'utilisateur ou simplement ouvrir l'application avec les derniers graphes utilisés.

BIBLIOGRAPHIE

A Search Procedure for Hamilton Paths and Circuits. **Rubin, Frank. 1974.** 4, New York, NY, USA : Association for Computing Machinery, 1974, Vol. 21. 0004-5411.

Contributors to Wikimedia projects. 2014. GitLab. *Wikipedia, the free encyclopedia.* [Online] Wikimedia Foundation, Inc., Octobre 25, 2014. [Cited: Juin 09, 2023.] https://en.wikipedia.org/wiki/GitLab.

—. 2001. Graph Theory. *Wikipedia, the free encyclopedia.* [Online] Wikimedia Foundation, Inc., Septembre 30, 2001. [Cited: Juin 05, 2023.] https://en.wikipedia.org/wiki/Graph_theory.

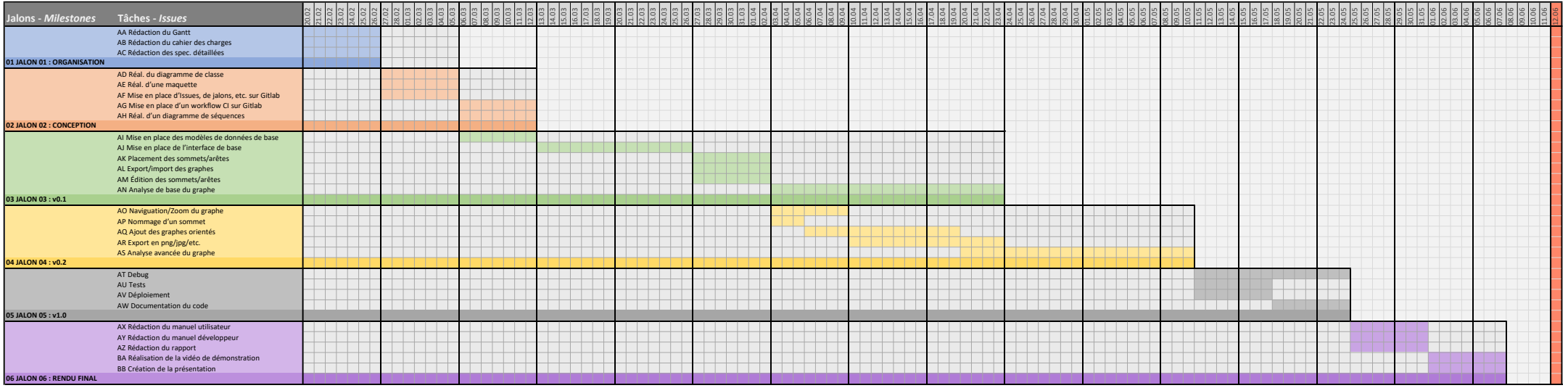
—. 2003. Seven Bridges of Königsberg. *Wikipedia, the free encyclopedia.* [Online] Wikimedia Foundation, Inc., Mars 07, 2003. [Cited: Juin 05, 2023.] https://en.wikipedia.org/wiki/Seven_Bridges_of_K%C3%B6nigsberg.

Heesch, Dimitri van. 2023. Doxygen. *Doxygen.* [Online] Mai 18, 2023. [Cited: Juin 09, 2023.] https://www.doxygen.nl/index.html.

Red Hat, Inc. 2022. What is CI/CD ? *RedHat.* [Online] Red Hat, Inc., Mai 11, 2022. [Cited: Juin 09, 2023.] https://www.redhat.com/en/topics/devops/what-is-ci-cd?cicd=32h281b.

TABLES DES ILLUSTRATIONS

ANNEXES


Jalons - Milestones
Tâches - Issues
AA Rédaction du Gantt
AB Rédaction du cahier des charges
AC Rédaction des spec. détaillées
01 JALON 01 : ORGANISATION
AD Réal. du diagramme de classe
AE Réal. d'une maquette
AF Mise en place d'Issues, de jalons, etc. sur Gitlab
AG Mise en place d'un workflow CI sur Gitlab
AH Réal. d'un diagramme de séquences
02 JALON 02 : CONCEPTION
AI Mise en place des modèles de données de base
AJ Mise en place de l'interface de base
AK Placement des sommets/arêtes
AL Export/import des graphes
AM Édition des sommets/arêtes
AN Analyse de base du graphe
03 JALON 03 : v0.1
AO Naviguation/Zoom du graphe
AP Nommage d'un sommet
AQ Ajout des graphes orientés
AR Export en png/jpg/etc.
AS Analyse avancée du graphe
04 JALON 04 : v0.2
AT Debug
AU Tests
AV Déploiement
AW Documentation du code
05 JALON 05 : v1.0
AX Rédaction du manuel utilisateur
AY Rédaction du manuel développeur
AZ Rédaction du rapport
BA Réalisation de la vidéo de démonstration
BB Création de la présentation
06 JALON 06 : RENDU FINAL