Projet IN608

Manipulation de Graphes orientés
Cahier des spécifications

AIRIAU Vincent, DAFRANE Salsabil, DE FREITAS Alexandre, GODINEAU Camille,
MALO Amandine, NARDIN Théo, OILLO Sébastien, PEUGNET Guillaume, TEDESCHI Hugo

2019 - 2020

Contents

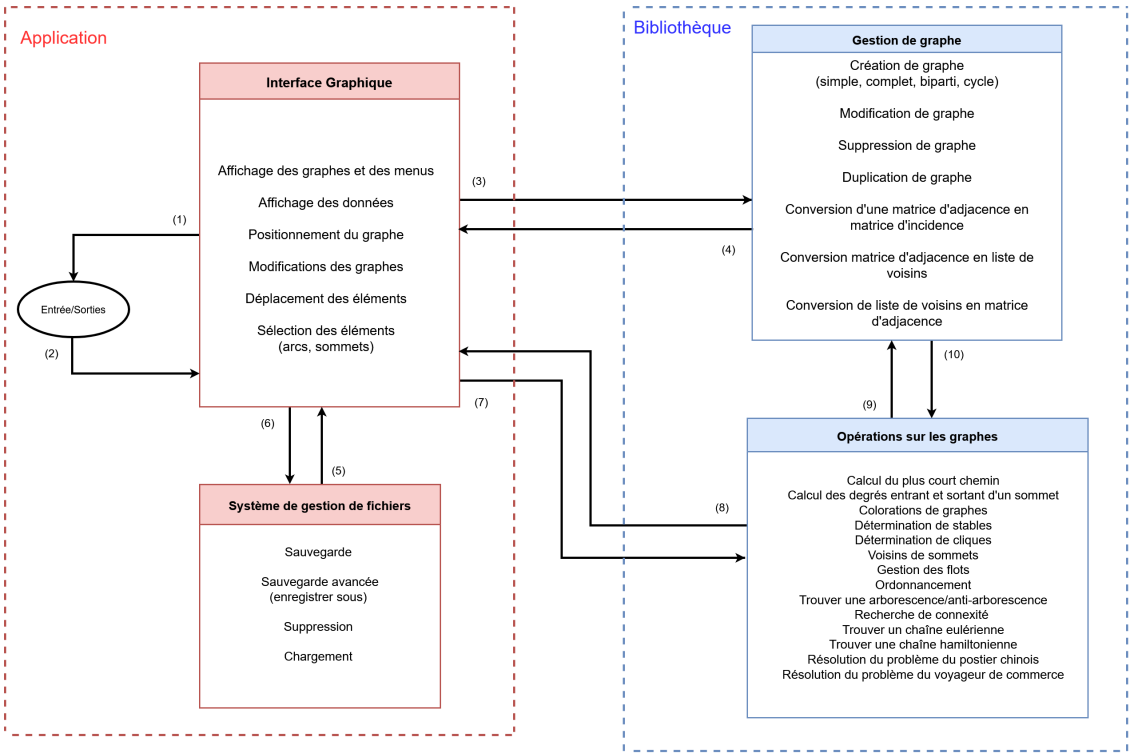
1 Introduction

Ce document est le cahier contenant les outils techniques que nous allons utiliser afin de réaliser notre projet de bibliothèque et d'application permettant la manipulation des Graphes orientés. Nous avons choisi d'aborder ce sujet dans le cadre universitaire car l'aspect pluridisciplinaire du sujet nous a semblé le point le plus pertinent. Ce projet nécessite le développement d'une application dotée d'une interface graphique et d'une bibliothèque.
L'objectif de notre projet est de permettre une utilisation sur deux fronts:
- Une bibliothèque permettant la modélisation et la manipulation de Graphes orientés.
- Une interface graphique permettant d'afficher les graphes et faciliter leur manipulation.
Elle s'adresse d'abord à un utilisateur qui souhaite générer des graphes orientés de son choix ou de façon aléatoire afin d'y appliquer des algorithmes usuellement utilisés en théorie des graphes. Elle permet aussi à un développeur tiers d'utiliser notre bibliothèque pour réaliser son propre projet.
Pour répondre aux problématiques posées par le sujet du projet, nous développerons avec le langage C++. Son choix a été effectué pour plusieurs raisons. Dans un premier temps, nous serons confrontés à la manipulation de nombreux objets, tandis que dans un second temps, de par la nature des algorithmes nécessaires, nous aurons besoin de fonctions procédurales. En conséquence, il est nécessaire d'utiliser un langage multi-paradigmes.
Enfin, notre application fonctionnera avec beaucoup de données, parfois dans des quantités exponentielles. Le langage choisi se devra donc d'être le plus performant possible. De cette manière, le C++ répond à l'ensemble de ces prérequis.
Nous présenterons d'abord les structures de données utilisées au cours de ce projet puis la description des quatre modules internes au projet agrémentés des cas de test requis pour un développement dirigé par les tests.

2 Rappels du cahier des charges

Afin de commencer l'implémentation de notre application, nous avons besoin, dans un premier temps, de détailler les structures, les classes et les fonctions que nous utiliserons. Pour ce faire, nous devons nous référer à notre cahier des charges, plus particulièrement à l'organigramme et à la liste des fonctionnalités.

2.1 Organigramme



1 :	Clics et touches du clavier	6 :	Nom du fichier, message de validation (suppression, sauvegarde) ou contenu du fichier
2 :	Affichage	7 :	Graphe, sommet du graphe, message de demande d'actions
3 :	Sommets sélectionnés, Arcs sélectionnés, graphe, choix des interactions, taille du graphe	8 :	Liste des sommets, sous-graphe, graphe, message de validation de l'opération
4 :	Graphe	9 :	Graphe modifié
5 :	Nom du fichier, graphe	10 :	Graphe

2.2 Fonctionnalités

2.2.1 Gestion de graphes

- **Création de graphe :** Créer un graphe qui peut être aléatoire ou à partir d'une liste de voisins fournie en amont.
- **Modification de graphe :** Permet de modifier un graphe, en rajoutant un sommet par exemple.
- **Duplication de graphe :** Copie un graphe.
- **Suppression de graphe :** Supprimer un graphe précédemment créé.
- **Conversion d'une matrice d'adjacence en liste de voisins :** Permet à partir d'une Matrice d'adjacence de créer une liste de voisins et inversement.
- **Conversion en matrice d'incidence :** Conversion d'une Matrice d'adjacence en Matrice d'incidence : Prend en paramètre une Matrice d'adjacence M1 et permet de la convertir dans la Matrice d'incidence M2 équivalente.

2.2.2 Système de gestion de fichiers

- **Sauvegarde :** Permet de sauvegarder dans un fichier l'ensemble des informations propres au graphe courant, son nom, les positions de ses Sommets, son nombre de Sommets, leur nom, le sens des Arcs, leur destination et leur poids.
- **Sauvegarde avancée (enregistrer sous) :** Permet de sauvegarder le graphe courant sous un nouveau nom et à un autre emplacement.
- **Chargement:** Permet de charger un fichier dans lequel est sauvegardé un graphe et de l'afficher.
- **Suppression :** Permet de supprimer un fichier dont le lieu de stockage est placé en paramètre.

2.2.3 Interface graphique

- **Affichage des graphes et des menus :** Affiche à l'écran les différents menus et le dessin en cours.
- **Affichage des données :** Afficher le nom des sommets et le poids des Arcs.
- **Positionnement du graphe :** Permet dans le cas où le graphe est aléatoire de placer les sommets de façon à ce qu'ils ne se superposent pas, mais aussi, de réorganiser à la demande de l'utilisateur les positions de ses sommets de façon à maximiser la lisibilité du graphe courant.
- **Création et modifications des graphes :** Permet à l'utilisateur de créer un graphe (en ajoutant des sommets ou des Arcs), d'affecter le poids des Arcs ou de nommer les sommets ou de changer le poids des Arcs ou le nom des Sommets.
- **Déplacement des éléments :** Permet à l'utilisateur de déplacer à l'aide de son curseur les points et de changer les destinations et le départ des Arcs.
- **Sélection des éléments (arcs, sommets) :** Permet de sélectionner un certain nombre de sommets et les Arcs qui y sont liés pour pouvoir travailler sur un sous-graphe du Graphe courant.

2.2.4 Opérations sur les graphes

- **Calcul du plus court chemin :** Permet de trouver l'ensemble des plus courts chemins ainsi que le plus court chemin entre un Sommet et tous les Sommets.
- **Calcul des degrés entrant et sortant d'un Sommet** Permet de calculer le degré d'un Sommet.
- **Coloration de graphe, détermination de stables et de cliques :** Permet de trouver une coloration du graphe mis en entrée ainsi que les stables et les cliques.
- **Gestion des flots :** Permet de résoudre le problème de flot maximal sur le graphe placé en entrée.
- **Ordonnancement :** Permet, à partir d'un formulaire rempli par l'utilisateur, de créer un diagramme de PERT, avec le calcul des tâches et du chemin critique.
- **Trouver l'arborescence:** Permet de trouver une arborescence couvrante du Graphe G.
- **Trouver l'anti-arborescence :** Permet de trouver une anti-arborescence couvrante sur le Graphe G.
- **Recherche de connexité :** Permet de vérifier la connexité du graphe grâce à l'arborescence et à l'anti-arborescence.
- **Trouver une chaîne eulérienne :** Permet de trouver la ou les chaînes eulériennes du Graphe placées en entrée si il en existe.
- **Trouver une chaîne hamiltonienne :** Permet de trouver la ou les chaînes hamiltoniennes du Graphe placé en entrée si il en existe.

- **Résoudre le problème du postier chinois :** Permet de résoudre le problème du postier chinois avec une recherche de chaîne eulérienne.
- **Résoudre le problème du voyageur de commerce :** Permet de résoudre le problème du voyageur de commerce sur un Graphe placé en entrée.
- **Voisins d'un sommet :** Permet de trouver les voisins d'un Sommet.

3 Description des structures

3.1 Vecteurs

Cette structure sera utile pour stocker des données sur les Sommets et les Arcs. Cela sera géré par la structure Vector, déjà implémentée dans la bibliothèque de base du C++. De cette manière, toutes les méthodes de gestion de Vector sont déjà developpées.

```
typedef struct Valeur_Vecteur {
    bool type;
    int valeur_entiere;
    float valeur_reel;
} VectVal;
```

Paramètres de la structure:

- *bool type:* Permet de différencier plus efficacement les entiers et les réels dans le vecteur. Cela sera utile pour l'analyse des données stockées dans la liste des valeurs contenues par un Sommet donné.
  type = 0 (entier), type = 1 (réel)
- *int valeur_entiere:* valeur de type entière.
- *int valeur_réel:* valeur de type réelle.

3.2 Ordonnancement

Cette structure est utile lors de l'implémentation de l'algorithme de PERT afin de gérer des tâches.

```
typedef struct ROW {
    int tache;
    string nom_tache;
    int duree;
    vector <int> taches_anterieures;
    vector <int> taches_posterieures;
} pert_row;
```

Paramètres de la structure:

- *int tache* : Identifiant de la tâche afin de la reconnaître plus facilement.
- *string nom_tache* : Étiquette de la tâche.
- *int duree* : Durée de la tâche.
- *vector <int> taches_antérieures* : Vecteur d'entiers qui contient les identifiants de toutes les tâches antérieures.
- *vector <int> taches_postérieures* : Vecteur d'entiers qui contient les identifiants de toutes les tâches postérieures.

4 Description des modules

Lors de la description de chaque module, nous supposons que les méthodes d'accès et d'affectation (get et set) pour chaque attribut d'une classe sont déjà définies pour éviter le superflu de méthodes. Il y aura des données qui circuleront entre les modules:
- Le module opérations sur les Graphes utilisera les données du modules gestion de Graphe et du module interface graphique.
- Le module gestion de Graphes recevra des données du module opération sur les Graphes et du module interface graphique.
- Le module interface graphique recevra des données du module gestion de Graphes, du module opérations sur les Graphes et du module système de gestion de fichiers.
- Le module système de gestion de fichier recevra des données uniquement du module interface Graphique.

4.0.1 Framework de test

Nous avons choisi comme framework de test Catch2, pour les raisons suivantes :

- Très aisé à implémenter et à utiliser.
- Pas de dépendances supplémentaires nécessaires autre que la bibliothèque standard.
- Les cas de test sont écrits comme pour des méthodes et fonctions en C++. De plus, ils sont considérés comme des String, il n'y a donc pas de conflits avec des mots réservés ou encore des noms d'objets ou de variables.
- Possibilité d'exécuter les cas de tests de façon indépendante ou par blocs.
- Mêmes opérateurs de comparaison et macro qu'en C++.

4.1 Gestion de Graphes

Le module de gestion de Graphe est présent dans la bibliothèque, ce module est utile lors de la création des Graphes ainsi que les différentes conversions nécessaires aux autres modules tels que opérations sur les Graphes. Ce module est composé de 4 classes, qui sont les classes nécessaires à la création d'un Graphe orienté (Sommet, Arc, Matrice et Graphe). Il communique aussi avec l'interface graphique pour afficher les Graphes que l'utilisateur crée. Il y aura donc circulation de données telles que les objets suivants: Sommet, Arc, Matrice et Graphe. Il communique aussi avec le module opération sur les graphes en envoyant les données relatives à l'objet Graphe à traiter.

4.1.1 Classe Sommet

```
class Sommet {

    private :
        int x;
        int y;
        int ID;
        string etiquette;
        vector <int> vecArc;
        map <String, VectVal> SCharge_utile;

    public :
        Sommet (int pos_x, int pos_y ,string etiq, int id, VectVal v);
        Sommet (int pos_x, int pos_y ,string etiq, int id);
        Sommet (string etiq, int id);
        Sommet (int id);
        Sommet (&Sommet S);
        ~Sommet ();

        bool operator==(Sommet const& S1, Sommet const& S2);
        bool operator!=(Sommet const& S1, Sommet const& S2);
        Sommet operator=(Sommet &S1, Sommet const& S2);

};
```

Attributs de classe Sommet :

- *int x* : Coordonnées entières pour placer le Sommet sur l'axe des abscisses. Nous avons choisi un entier car le plan est représenté par des pixels.
- *int y* : Coordonnées entières pour placer le Sommet sur l'axe des ordonnées. Nous avons choisi un entier car le plan est représenté par des pixels.
- *int ID* : Identifiant du Sommet. Nous avons choisi un entier car ils sont facilement manipulables et que nous pouvons les utiliser pour représenter les cases d'une Matrice. Les identifiants iront de 0 à V-1 où V est le nombre de sommets du Graphe.
- *string etiquette* : Nom du Sommet utilisé lors de l'affichage pour montrer à l'utilisateur ce que représente un Sommet (nom d'une ville, nom d'une piste de ski, nom d'une gare, etc...). Nous utilisons un String pour nommer le Sommet explicitement, par défaut l'étiquette est égale à l'id.
- *vector <int> vecArc* : Vecteur contenant les Arcs sortants d'un Sommet représentés dans le vecteur par leur id. Nous utilisons un vecteur car le type vecteur est un type dynamique déjà présent en C++.

- *map <string,vectVal> SCharge_utile* : Par définition, map est un conteneur trié associatif. Les clés sont triées entre elles par comparaison grâce à la fonction "compare" inclue dans la bibliotèque standard C++. Elle permet de stocker les valeurs sur lesquelles nous travaillerons plus tard (les dates au plus tôt, au plus tard par exemple) et de pouvoir les retrouver aisément.

Les constructeurs de la classe :

- *Sommet (int pos_x, int pos_y ,string etiq, int id, vector<int> vecArc, map <string, VectVal> v)* : Constructeur qui permet d'instancier un objet Sommet avec deux positions, un nom (etiq), une map embarquant les données utiles pour les algorithmes et un identifiant (id) pour pouvoir l'identifier lors des opérations sur les Graphes, il est aussi initialisé avec un vecteur embarquant les ID des Arcs sortants.
- *Sommet (int pos_x, int pos_y, string etiq, int id)* : Constructeur qui permet de tout initialiser sauf la map embarquant la charge utile.
- *Sommet(String etiq, int id)* : Constructeur d'objets Sommet composés d'une étiquette et d'un identifiant afin de repérer le Sommet plus facilement.
- *Sommet(int id)* : Constructeur d'objets Sommet composés d'un identifiant afin de repérer le Sommet plus facilement. Ce constructeur est le constructeur avec lequel on peut initialiser l'objet avec le plus petit nombre d'arguments. L'étiquette du sommet est initialisée avec l'id.
- *Sommet (&Sommet S)* : Constructeur par copie, il permet de construire un nouveau Sommet à partir d'un Sommet existant. L'objet créé possédera les mêmes attributs, cependant nous pouvons changer certains de ces attributs en fonction de la situation.

Le destructeur de la classe :

- $\sim$ *Sommet ()* : Destructeur de l'objet Sommet, il sera employé pour la gestion de la mémoire et pour l'allocation dynamique des objets Sommets.

Surcharge des opérateurs de la classe:

- *bool operator==(Sommet const& S1, Sommet const& S2)* : Surcharge de l'opérateur d'égalité. Cela permettra de vérifier si deux objets Sommet ont tous les deux leurs attributs de classe deux à deux identiques. Renvoie 1 si tous les attributs deux à deux sont identiques entre les deux Sommets, 0 si il y au moins une différence entre les deux attributs.
- *bool operator !=(Sommet const& S1, Sommet const& S2)* : Surcharge de l'opérateur d'égalité. Cela permettra de vérifier si deux objets Sommets ont tous les deux leurs attributs de classe deux à deux différents. Renvoie le booléen 1 si les deux sommets ont au moins un attribut différent entre eux, 0 sinon.
- *Sommet operator=(Sommet const &S1, Sommet const& S2)* : Surcharge de l'opérateur d'affectation de valeur. Cela permettra d'associer aux attributs du premier Sommet S1, toutes les valeurs des mêmes attributs du second Sommet S2.

Test de la classe Sommet:

- TEST_CASE("Test des constructeurs ", "[Sommet]") : Teste les différentes formes de constructeurs.
- TEST_CASE("Test du destructeur", "[Sommet]") : Le destructeur va être testé avec un sommet construit avec tous les paramètres puis on va vérifier que le Sommet soit correctement détruit.
- TEST_CASE("Test des getters", "[Sommet]") : Les getters vont être testés, en vérifiant les données renvoyées par la méthode avec celles qu'on estime recevoir.
- TEST_CASE("Test des setters", "[Sommet]") : Les setters vont être testés, en modifiant les données avec la méthode et en vérifiant le résultat.
- TEST_CASE("Test de l'operateur =", "[Sommet]") : Teste si le Sommet retourné est bien égal au Sommet passé en paramètres.
- TEST_CASE("Test de l'operateur ==", "[Sommet]") : Teste si les deux Sommets sont égaux c'est-à-dire que les attributs des Sommets sont égaux, la fonction retourne 1 si c'est la cas, 0 sinon.
- TEST_CASE("Test de l'operateur !=", "[Sommet]") : Teste si les deux Sommets sont différents c'est-à-dire que les attributs des Sommets sont différents, la fonction retourne 1 si c'est la cas, 0 sinon.

4.1.2 Classe Arc

```
class Arc {

    private:
        int ID;
        string etiquette;
        int IDdepart;
        int IDarrive;
        map <string, VectVal> ACharge_utile;

    public:
        Arc (string etiq, int id, int Sdepart, int Sarrivee,  map <string, VectVal> Map);
        Arc (string etiq, int id, int Sdepart, int Sarrivee);
        Arc (int id, int Sdepart, int Sarrivee);
        Arc (&Arc a);
        ~Arc();

        bool operator==(Arc const& A1, Arc const& A2);
        bool operator!=(Arc const& A1, Arc const& A2);
        Arc operator=(Arc &A1, Arc const& A2);

};
```

Attributs de classe:

- *int ID* : Identifiant de l'Arc. Nous avons choisi un entier pour les mêmes raisons que les Sommets. Les identifiants iront de 0 à E - 1.
- *string etiquette* : Nom de l'Arc. On utilise la classe String pour avoir un nom d'Arc explicite. Par défaut l'étiquette est égale à l'ID. Cet attribut sera principalement utilisé lors de l'affichage.
- *int IDdepart* : ID du Sommet de départ. On utilise l'ID car nous n'avons pas besoin de l'objet Sommet entier.
- *int IDarrive* : ID du Sommet d'arrivée. On utilise l'ID pour les mêmes raisons que IDdepart.
- *map <string, VectVal > ACharge_utile* : map mettant en relation un string (nom de la charge utile à stocker) et un entier ou un réel (correspondant à la valeur de la charge utile), il permet de stocker toutes les valeurs sur lesquelles on veut pouvoir travailler: poids, flots maximum, etc.

Les constructeurs de la classe :

- *Arc (string etiq, int id, int Sdepart, int Sarrivee, map <string, VectVal> Map)* : construit un Arc avec son nom, numéro, son Sommet de départ et d'arrivée et une map comprenant les informations sur l'Arc (son poids, flot max).
- *Arc (string etiq, int id, int Sdepart, int Sarrivee)* : Constructeur permettant de créer un Arc avec son étiquette (qui est son nom affiché à l'écran pour permettre à l'utilisateur d'avoir une représentation plus compréhensible), son numéro, son Sommet de départ et son Sommet d'arrivée. La différence avec le précédent constructeur est qu'il ne nécessite pas de map pour être créé et va mettre la map par défaut (vide). Cela permet de créer des Arcs sans poids ou flot maximal.
- *Arc (int id, int Sdepart, int Sarrivee)* : Permet de créer un Arc avec seulement son numéro et ses Sommets de départ et d'arrivée tout en ne lui affectant pas de map. L'étiquette de l'arc est initialisée avec l'id.
- *Arc (&Arc a)* : Constructeur de copie. Il construit un Arc en utilisant les mêmes attributs que l'Arc en entrée, cependant nous pouvons changer certains de ces attributs en fonction de la situation.

Le destructeur de la classe :

- $\sim$ *Arc ()* : Destructeur de l'objet Arc, il sera employé pour la gestion de la mémoire et pour l'allocation dynamique des objets Arc.

Surcharge des opérateurs de la classe Arc:

- *bool operator == (Arc const& A1, Arc const& A2)* : Surcharge de l'opérateur d'égalité. Cela permettra de vérifier si deux objets Arc ont tous les deux leurs attributs de classe deux à deux identiques. Renvoie 1 si tous les attributs sont deux à deux identiques entre les deux Arc, 0 si il y au moins une différence entre les deux attributs.
- *bool operator*! *=(Arc const& A1, Arc const& A2)* : Surcharge de l'opérateur d'égalité. Cela permettra de vérifier si deux objets Arc ont tous les deux leurs attributs de classe deux à deux différents. Renvoie le booléen 1 si les deux Arc ont au moins un attribut différent entre eux, 0 sinon.

- *Arc operator=(Arc& A1, Arc const& A2)* : Surcharge de l'opérateur d'affectation. On associera aux attributs de l'Arc A1, les mêmes valeurs contenues aux attributs de l'Arc A2.

Test de la classe Arc :

- TEST_CASE("test des constructeurs", "[arc]") : Teste les différentes formes de constructeurs.
- TEST_CASE("test du destructeur", "[arc]") : Le destructeur va être testé avec un Arc construit avec tous les paramètres puis on va vérifier que l'Arc soit correctement détruit.
- TEST_CASE("test des getters", "[arc]") : Les getters vont être testés, en vérifiant les données renvoyées par la méthode avec celles qu'on estime recevoir.
- TEST_CASE("test des setters", "[arc]") : Les setters vont être testés, en modifiant les données avec la méthode et en vérifiant le résultat.
- TEST_CASE ("Test de l'operateur =", "[arc]") : Teste si l'objet Arc passé en argument est bien égal à l'objet Arc renvoyé.
- TEST_CASE ("Test de l'operateur ==", "[arc]") : Teste si le booléen renvoyé est bien conforme à la comparaison de l'objet Arc en argument.
- TEST_CASE("Test de l'operateur !=", "[arc]") : Teste si les deux Arc sont différents c'est-à-dire qu'au moins un des attributs des Arcs est différent, la fonction retourne 1 si c'est le cas, 0 sinon.

4.1.3 Classe Matrice

```
class Matrice {

    private:
        int taille V;
        int taille E;
        int type;
        vector<vector<int>> tab;

    public :
        Matrice (Graphe G, int type);
        Matrice (int tailleV);
        Matrice (int tailleV, int tailleE, int t)
        Matrice (Matrice &M);
        ~Matrice ();

        Matrice conversion_incidence ();
        Matrice inversion_Matrice ();
        Graphe conversionGraphe();
        int Sommet_non_isole();
        int modifTab(int x, int y, int n);
        void supprLigne(int x);
        void supprCol(int y);

        bool operator==(Matrice const& M1 ,Matrice const& M2);
        bool operator!=(Matrice const&  M1 ,Matrice const& M2);
        Matrice operator=(Matrice &M1, Matrice const& M2 );
};

enum typeM
{
    ADJACENCE = 0,
    INCIDENCE = 1,
    POIDS = 2,
    PARENT = 3,
    QUELCONQUE = 4;
};
```

Attributs de classe:

- *int taille V* : Représente le nombre de Sommets dans la Matrice.
- *int taille E* : Représente le nombre d'Arcs dans une Matrice, il est utilisé dans les matrices d'incidence sinon il est initialisé à NULL.

- *int type* : Les valeurs que peut prendre "type" sont énumérées dans "typeM" ci-dessus. Cet attribut nous permet donc d'identifier les matrices en fonction de leur type pour les sélectionner lors de l'application d'algorithme.
- *vector<vector<int>> tab* : Tableau à deux dimensions représentant la Matrice.

Constructeurs de la classe:

- *Matrice (Graphe G, int type)* : Constructeur d'un objet Matrice décrivant un Graphe G. L'entier "type" permet de savoir quel type de Matrice est souhaité : adjacence, incidence, de poids, parent ou quelconque. Dans le cas où l'on veut construire une Matrice d'adjacence de taille V*V on initialise tab avec des 0 et des 1 dans le cas où les Arcs ne possèdent pas de poids. Dans le cas où les Arcs possèdent des poids alors le tab est initialisé avec le poids des Arcs. Dans le cas d'une Matrice d'incidence, la Matrice sera de taille V*E avec V = le nombre de sommets et E = le nombre d'arcs. Le tab est initialisé à 1 si l'arc est entrant et à -1 si l'arc est sortant.
- *Matrice (int tailleV)* : Constructeur d'un objet Matrice dans le cas où la Matrice ne représente pas un Graphe. Par exemple lors de l'algorithme de Floyd-Warshall nous avons besoin de Matrice de poids.
- *Matrice (int tailleV, int tailleE, int t)* : Permet de générer une Matrice de taille V*E de type t.
- *Matrice (Matrice &M)* : Constructeur de copie de la classe Matrice.

Destructeur de la classe Matrice:

- $\sim$ *Matrice()* : Destructeur d'un objet Matrice, il sera employé pour la gestion de la mémoire et pour l'allocation dynamique des objets Matrice.

Méthodes de la classe:

- *Matrice conversion_incidence ()* : Conversion d'une matrice d'adjacence en matrice d'incidence. Si la matrice ne peut pas être convertie en matrice d'incidence renvoie une matrice de 1 sur 1 avec comme seule valeur "-1".
- *Matrice inversion_Matrice ()* : Inversion d'une Matrice afin de l'utiliser dans les opérations sur les Graphes.
- *Graphe conversionGraphe ()* : Renvoie le graphe correspondant à la matrice. Ne fonctionne qu'en cas de matrice d'adjacence ou de d'incidence. Si la matrice ne peut pas être convertie en matrice d'incidence renvoie une matrice de 1 sur 1 avec comme seule valeur "-1".
- *int Sommet_non_isole ()* : Recherche si un Sommet est isolé dans le Graphe, c'est-à-dire si aucun autre Sommet est relié à celui-ci. Retourne 0 si la Matrice contient un Sommet ou plus sans Arc et 1 si la Matrice ne contient aucun Sommet isolé.
- *int modifTab(int x, int y, int n)* : Permet de modifier la case aux positions tab[x][y] et de lui affecter n. Renvoie -1 en cas d'erreur.
- *void supprLigne(int x)* : permet de supprimer la ligne x de la Matrice, renvoie -1 en cas d'erreur.
- *void supprCol(int y)* : permet de supprimer la colonne y de la Matrice, renvoie -1 en cas d'erreur.

Surcharge des opérateurs de la classe Matrice:

- *bool operator==(Matrice const& M1, Matrice const& M2)* : Surcharge de l'opérateur d'égalité. Cela permettra de vérifier si deux objets Matrice ont tous les deux leurs attributs de classe deux à deux identiques. Renvoie 1 si tous les attributs sont deux à deux identiques entre les deux Matrices, 0 si il y au moins une différence entre les deux attributs.
- *bool operator*! *=(Matrice const& M1, Matrice const& M2)* : Surcharge de l'opérateur d'égalité. Cela permettra de vérifier si deux objets Matrice ont tous les deux leurs attributs de classe deux à deux différents. Renvoie le booléen 1 si les deux Matrices ont au moins un attribut différent entres elles, 0 sinon.
- *Matrice operator=(Matrice & M1, Matrice const& M2)* : Surcharge de l'opérateur d'affectation. Teste si l'objet Matrice passé en argument est bien égal à l'objet Matrice renvoyé.

Test de la classe Matrice :

- TEST_CASE("Test des constructeurs", "[matrice]") : Teste les différentes formes de constructeurs.
- TEST_CASE("Test du destructeur", "[matrice]") : Le destructeur est testé avec une Matrice existante puis on vérifie qu'elle est bien détruite.
- TEST_CASE("Test des getters", "[matrice]") : Les getters sont testés, par vérification des données renvoyées.
- TEST_CASE("Test des setters", "[matrice]") : Les setters sont testés, en modifiant les données avec la méthode par vérification des données renvoyées.
- TEST_CASE("Test de conversion en Matrice d'incidence", "[matrice]") : La conversion de la Matrice d'adjacence en Matrice d'incidence est testée par comparaison des deux Matrices.

- TEST_CASE("Test détection erreur conversion en matrice d'incidence", "[matrice]") : Pour vérifier que la fonction de conversion détecte les cas dans lesquels elle ne peut pas faire de conversion. Pour cela on lui passe une Matrice qui n'est pas adjacente et on vérifie que la Matrice de retour est une matrice d'erreur.
- TEST_CASE("Test d'inversion de la matrice"', "[matrice]") : L'inversion de la Matrice est testée. Une Matrice M et son inverse M_inverse sont créées, puis la méthode d'inversion est appelée sur M. La Matrice obtenue est ensuite comparée avec M_inverse.
- TEST_CASE("Test de la conversion en graphe", "[matrice]") : La conversion d'une Matrice en Graphe est testée. Pour cela une Matrice M et un Graphe G_convert correspondant à cette Matrice sont préalablement créés. La méthode est appelée sur la Matrice M puis le Graphe obtenu est vérifié par comparaison avec le Graphe G_invert.
- TEST_CASE("Test détection erreur conversion en graphe", "[matrice]") : Créer une Matrice n'ayant pas un type autorisant cette transformation. Si la fonction renvoie une Matrice d'erreur alors le test est passé.
- TEST_CASE("Test de sommet isolé d'un graphe", "[matrice]") : On vérifie, si pour un résultat retourné, le bon test est effectué. Pour 1 : Tous les Sommets du Graphe ont, au moins, un Arc. Pour 0 : le Graphe a au moins un Sommet sans Arc.
- TEST_CASE ("Test de l'operateur =", "[matrice]" : Teste si l'objet Matrice passé en argument est bien égal à l'objet Matrice renvoyé.
- TEST_CASE ("Test de l'operateur ==", "[matrice]" : Teste si le booléen renvoyé est bien conforme à la comparaison de l'objet Matrice en argument.
- TEST_CASE("Test de l'operateur !=", "[matrice]") : Teste si les deux Matrices sont différentes c'est-à-dire que les attributs des Matrices sont différents, la fonction retourne 1 si c'est la cas, 0 sinon.

4.1.4 Classe Graphe

```
class Graphe {

    private:
        string etiquette;
        vector <Arc> liste_Arcs;
        vector <Sommet> liste_Sommets;
        string path;

    public:
        Graphe (string nom, vector <Sommet> listeS ,vector <Arc> listeA ,
                string path);
        Graphe (string nom);
        Graphe (Matrice& M);
        Graphe (Graphe& G);
        Graphe (vector<vector<int>> liste_voisin)
        ~Graphe ();

        Matrice conversion_vers_Matrice_adj ();
        Matrice conversion_vers_Matrice_inc ();
        vector<vector<int>> conversion_vers_listeDeVoisins ();
        int ajout_Sommet(int id, int posx, int posy);
        int supprimer_Sommet (int id);
        int ajout_Arc (int id_Sdepart, int id_Sarrive);
        int supprimer_Arc (int id);
        vector<Sommet> getVecteurSommet(vector<int> ID);

        bool operator==(Graphe const& G1 ,Graphe const& G2);
        bool operator!=(Graphe const&  G1 ,Graphe const& G2);
        Graphe operator=(Graphe &G1, Graphe const& G2 );

};
```

Attributs de classe:

- *string etiquette* : String qui contient l'identifiant du Sommet.
- *vector <Arc> liste_Arcs* : Vecteur contenant l'ensemble des Arcs du Graphe.
- *vector <Sommet> liste_Sommets* : Vecteur qui stocke les Sommets contenus dans le Graphe.

- *string path* : Chaîne de caractères stockant le chemin où enregistrer le Graphe.

Les constructeurs de la classe:

- *Graphe(string nom, vector <Sommet> listeS, vector <Arc> listeA, string path)* : Ce constructeur de Graphe prend en argument un string correspondant au nom du Graphe, un vecteur de Sommet nommé listeS qui correspond à la liste des Sommets à stocker dans ce Graphe, un vecteur d'Arc listeA qui stockera les Arcs à associer entre les Sommets de ce Graphe et un string path indiquant où enregistrer le Graphe.
- *Graphe (string nom)* : Constructeur de Graphe prenant seulement en argument son nom sous la forme d'un string. Les vecteurs liste_Arcs et liste_Sommets seront des vecteurs vides et l'attribut path sera à NULL.
- *Graphe (Matrice& M) :* Construit un Graphe à partir d'une Matrice d'adjacence ou d'incidence.
- *Graphe (Graphe&G)* : Constructeur de Graphe à partir d'un Graphe déjà existant. On fait donc une copie des éléments du Graphe rentré en paramètre.
- *Graphe(vector<vector<int>> liste_voisin) :* Construit un graphe à partir d'une liste de voisins

Destructeur de la classe:

- $\sim$ *Graphe()* : Destructeur du Graphe. Il sera utile pour la gestion de la mémoire.

Méthodes de la classe:

- *Matrice conversion_vers_Matrice_adj()* : Cette méthode convertit le Graphe qui a appelé cette méthode en renvoyant une Matrice d'adjacence, à partir de ses listes de voisins et d'Arcs.
- *Matrice conversion_vers_Matrice_inc()* : Cette méthode convertit le Graphe qui a appelé cette méthode en renvoyant une Matrice d'incidence, à partir de ses listes de voisins et d'Arcs.
- *vector<vector<int>> conversion_vers_listeDeVoisin ()* : Cette méthode utilise les listes de Sommets et d'Arcs pour construire une liste de voisins représentée sous forme de vecteur de vecteurs d'entiers qui sera utilisé dans certains algorithmes. Le premier vecteur sera de taille V et contiendra des vecteurs contenants les ID des Sommets voisins.
- *int ajout_Sommet(int id, int posx, int posy)* : Cette méthode prend en argument un entier id pour l'id du Sommet à ajouter au Graphe, un entier posx pour la position en x sous forme d'entier et posy pour la position en y sous forme d'entier de ce même Sommet. Cela permet d'ajouter un Sommet au Graphe. Il y aura donc incrémentation du nombre de Sommets contenu dans le Graphe à int nbSommets et on ajoutera au vector liste_Sommet le nouveau Sommet. Cette méthode va renvoyer 1 si l'ajout de Sommet a été fait avec succès, -1 si il y a eu une erreur.
- *int supprimer_Sommet(int id)* : Cette méthode permet de supprimer un Sommet du Graphe à partir de l'id entré en argument. Cette méthode va renvoyer 1 si la suppression du Sommet a été faite avec succès, -1 si il y a eu une erreur. Il y aura décrémentation du nombre de Sommets ainsi qu'une modification des IDs des sommets suivant celui supprimé.
- *int ajout_Arc(int id_Sdepart, int id_Sarrive)* : Cette méthode permet d'ajouter un Arc au Graphe. On rentre en argument l'id du Sommet de départ et l'id du Sommet d'arrivée. Cette méthode va renvoyer 1 si l'ajout de l'Arc a été effectué avec succès, -1 si il y a eu une erreur.
- *int supprimer_Arc(int id)* : Cette méthode supprime un Arc dont l'id est passé en paramètre. Cette méthode va renvoyer 1 si la suppression de l'Arc a été effectuée avec succès, -1 sinon. Les IDs des Arcs seront modifiés en conséquence (de la même manière que pour supprimer$_S$*ommet*).
- *vector<Sommet> getVecteurSommet(vector<int> ID)* : Cette méthode permet de récupérer un vecteur de sommet correspondant au vecteur d'ID passé en paramètre.

Surcharge des opérateurs de la classe Graphe:

- *bool operator==(Graphe const& G1, Graphe const& G2)* : Surcharge de l'opérateur d'égalité. L'opérateur permettra de vérifier si deux objets Graphes ont tous les deux leurs attributs de classe deux à deux identiques. Renvoie 1 si tous les attributs deux à deux sont identiques entre les deux Graphes, 0 si il y au moins une différence entre les deux attributs.
- *bool operator*! *=(Graphe const& G1, Graphe const& G2)* : Surcharge de l'opérateur d'inégalité. L'opérateur permettra de vérifier si deux objets Graphe ont tous les deux leurs attributs de classe deux à deux différents. Renvoie le booléen 1 si les deux Graphes ont au moins un attribut différent entre eux, 0 sinon.
- *Graphe operator=(Graphe G1, Graphe const& G2)* : Surcharge de l'opérateur d'affectation. On associera aux attributs du Graphe G1, les mêmes valeurs contenues aux attributs du Graphe G2.

Test de la classe Graphe :

- TEST_CASE("Test des constructeurs", "[graphe]") : Teste si l'objet Graphe passé en argument est bien égal à l'objet Graphe renvoyé.

- TEST_CASE("Test du destructeur", "[graphe]") : Teste si le booléen renvoyé est bien conforme à la comparaison de l'objet Graphe en argument et celle de l'objet Graphe this.
- TEST_CASE("Test des getters", "[graphe]") : Les getters sont testés, par vérification des données renvoyées.
- TEST_CASE("Test des setters", "[graphe]") : Les setters sont testés, en modifiant les données avec la méthode par vérification des données renvoyées.
- TEST_CASE("Test des conversions en matrice d'incidence et en Matrice d'adjacence", "[graphe]") : Pour vérifier ces deux méthodes nous allons créer un Graphe G0 et pré-définir les Matrices d'adjacence et d'incidence.
  On va vérifier que chacune de nos deux méthodes renvoient respectivement la Matrice d'adjacence et d'incidence que nous savons juste.
- TEST_CASE("Test d'ajout de sommets"', "[graphe]") : On vérifie que le constructeur de Sommet a bien été appelé en vérifiant que ce Sommet a bien été ajouté à la liste des Sommets du Graphe.
- TEST_CASE("Test suppression de sommet", "[graphe]") : On ajoute un Sommet à un Graphe G, avec un id défini. Puis on utilise la méthode et on vérifie que le Sommet n'est plus dans la liste des Sommets du Graphe et que les IDs des Sommets ont bien été modifiés.
- TEST_CASE("Test d'ajout d'arcs", "[graphe]") : On vérifie que le constructeur d'Arc a bien été appelé en vérifiant que cet Arc a bien été ajouté à la liste des Arcs du Graphe.
- TEST_CASE("Test de suppression d'arcs", "[graphe]") : On supprime un Arc d'un Graphe, on vérifie que l'Arc n'appartient plus au Graphe et que les IDs des Arcs ont bien été modifiés.
- TEST_CASE("Test de la conversion en liste de voisins", "[graphe]") : On teste si la liste de voisins correspond au Graphe.
- TEST_CASE("Test de l'opérateur ==","[graphe]") : Teste si l'objet Graphe passé en paramètre est bien égal à l'objet Graphe renvoyé.
- TEST_CASE("Test de l'opérateur =","[graphe]") : Teste si le booléen renvoyé est bien conforme à la comparaison de l'objet Graphe.
- TEST_CASE("Test de l'opérateur !=", "[graphe]") : Teste si les deux Graphes sont différents c'est-à-dire que les attributs des Graphes sont différents, la fonction retourne 1 si c'est le cas, 0 sinon.
- TEST_CASE("Test de getVecteurSommet ", "[graphe]") : Teste si pour un vecteur d'IDs donnés cela renvoie bien la bonne liste de Sommets.

4.2 Système de gestion des fichiers

Le module système de gestion des fichiers présent dans l'application, permet de gérer les fichiers soit en les sauvegardant, en les chargeant ou encore en les supprimant. L'utilisateur pourra gérer les fichiers de Graphes à travers l'interface graphique. Pour ce module nous avons décidé d'utiliser la bibliothèque RapidJSON. Ce module envoie des données au module interface graphique (Graphe, int, bool), il communique les données relatives au Graphe stocké.
Nous précisons que pour un Graphe, d'après la structure proposée, il y est donc inclus ses Sommets, ses Arc et l'emplacement du fichier ou est stocké le graphe.

La Bibliothèque JSON :

Nous souhaitons avoir un langage de format de données textuelles permettant de rendre lisible les fichiers de sauvegarde par l'utilisateur. Pour permettre à l'utilisateur de lire les fichiers de sauvegarde ou les faire lire par un autre programme nous devons avoir un langage de description suffisamment clair pour être lu par un humain, mais suffisamment utilisé et avec des règles d'écritures fortes pour permettre à l'utilisateur de pouvoir s'en servir avec d'autres programmes.
Le JSON nous est donc apparu comme une évidence, en effet, ce langage est doté d'une syntaxe claire et lisible, il est très utilisé et de nombreuses bibliothèques dans plusieurs langages existent, permettant ainsi à nos fichiers de sauvegarde d'être ré-exploitables. Sa syntaxe suit également des règles suffisamment fortes pour communiquer efficacement et sans ambiguïté.
Comme dit précédemment le JSON bénéficie de très nombreuses bibliothèques et cela dans un large panel de langages différents, le C++ n'échappe pas à la règle. Si on regarde sur le site officiel du JSON on peut voir près de 23 bibliothèques différentes rien que pour le C++. Certaines sont prévues pour des objectifs précis tels que les Arduinos.
Rapidement nous nous sommes intéressés à la bibliothèque RapidJSON car au vu de la taille potentielle des objets que nous voulons stocker, manipuler la vitesse n'est pas un paramètre à prendre à la légère.
Nous l'avons comparé à d'autres bibliothèques et RapidJSON est bien l'une des plus rapides, en effet son efficacité est comparable à celle de la fonction strlen().

Fonctions du système de gestion de fichiers:

```
int sauvegarde(string path = NULL, Graphe G);
Graphe chargement(string path);
Bool verif_file(Document D);
int supression(string path);
```

- *int sauvegarde(string path = NULL, Graphe G)* : Si la valeur de path n'est pas NULL, nous sommes alors dans le cas d'une "sauvegarde sous" ou d'un nouveau document car on spécifie un chemin de sauvegarde.
  La fonction va parser l'objet G et va écrire dans le fichier placé en paramètre ou, le cas échéant, dans le chemin placé en attribut. L'objet sera écrit comme expliqué dans la partie "Exemple d'un fichier JSON" en annexe. Si la sauvegarde se passe sans encombre alors la valeur 0 sera renvoyée sinon -1.

- *Graphe chargement(string path)* : Cette fonction permet de parser le fichier JSON où était sauvegardé un Graphe lors d'une précédente session et de le charger. Elle prend en entrée le chemin path du fichier et donne en sortie le Graphe présent dans le fichier.

- *Bool verif_file(Document D)* : Fonction qui va se charger de vérifier que le document JSON correspond à l'archétype de notre document de sauvegarde. La structure Document est un objet en JSON capable de stocker des informations en JSON. Si l'architecture est la bonne alors la fonction retourne 1, sinon elle retourne 0.

- *int supression(string path)* : Supprime le fichier du chemin path donné en entrée. Cette fonction renvoie un code de réussite ou d'erreur.

Tests du système de gestion de fichier :

- TEST_CASE("Test du nom du fichier", "[fichier]") : Teste pour savoir si le nom du fichier est équivalent à celui du Graphe.

- TEST_CASE("Test du chemin du fichier", "[fichier]") : Teste pour savoir si le path du fichier est équivalent à celui du Graphe.

- TEST_CASE("Test de validité du fichier", "[fichier]") : Permet de vérifier que la fonction valide les documents qui doivent l'être (respectant le document de sauvegarde type décrit en annexe) et invalide ceux ne le respectant pas.

- TEST_CASE("Test de chargement de fichier", "[fichier]") : Vérifie que la fonction charge bien un Graphe présent dans un fichier de test.

- TEST_CASE("Test de suppression du fichier", "[fichier]") : Teste pour savoir si le nom du fichier existe encore après sa suppression.

4.3 Opérations sur les Graphes

Le module d'opération des Graphes, présent dans la bibliothèque, permet à l'utilisateur tiers tout comme à notre application d'utiliser les différents algorithmes sur nos structures de Graphes et de Matrice. Les différentes fonctions utilisent des Matrices, l'objet Graphe,Sommets ou juste des IDs en fonction des besoins de l'algorithme et pour éviter un surplus de données inutiles. Le formalisme choisi du Graphe (Matrice, liste de voisins ou objets Graphe), provient du module "Gestion de Graphe" auquel le module peut renvoyer un Graphe modifié, principalement dans le cas d'utilisation de la bilbiothèque par un développeur tiers. L'exécution des algorithmes est commandée par l'Interface Graphique sur des Graphes qu'elle fournit sous forme d'objets. Le module d'Opération sur les Graphes lui renverra alors une liste de Sommets (sous forme de vecteur), un sous-Graphe (sous forme d'objet), ou un message d'erreur.

Fonctions des opérations sur les Graphes :

- *pair<vector<vector<int>>, vector <int >> calcul_Bellman (Matrice M, Sommet S)* : Calcul du plus court chemin grâce à l'algorithme de Ford-Bellman entre un Sommet S et tous les autres Sommets du Graphe. Il prend en entrée une Matrice d'adjacence. Il renvoie une paire de vecteurs, le premier vecteur représente les chemins pour aller du Sommet S vers les autres sommets du Graphe. Le deuxième vecteur indique la distance entre le Sommet S et les autres Sommets du Graphe.

- *pair<Matrice, Matrice> calcul_Floyd_Warshall(Matrice M)* : Calcul du plus court chemin grâce à l'algorithme de Floyd-Warshall entre chaque Sommet du Graphe. Il prend en entrée la Matrice d'adjacence représentant un Graphe avec les poids des Arcs. En sortie, la fonction retourne une paire de Matrices, la première Matrice est la Matrice de poids minimum pour aller d'un Sommet à un autre et la deuxième Matrice indique le prochain Sommet à atteindre pour effectuer le plus court chemin d'un Sommet.

- *vector <int> liste_floyd(Matrice Parent, int deb, int fin)* : Cette fonction va interpréter la Matrice parent donnée par la fonction de Floyd-Warshall. Elle va retourner une liste d'ID qui correspond au plus court chemin entre un Sommet deb (départ) et un Sommet fin (arrivée).

- *int calcul_degres_entrant(Matrice M, int ID)* : Calcule le degré entrant d'un Sommet d'identifiant égal à ID. Elle prend en entrée une Matrice d'adjacence et un ID qui permet de calculer le degré du bon Sommet. Elle retourne un entier donnant le nombre d'Arcs entrant dans ce Sommet.

- *int calcul_degres_sortant(Sommet S)* : Calcule le degré sortant d'un Sommet S. Elle prend en entrée le Sommet pour lequel on cherche à calculer le degré sortant du Sommet S. Elle retourne un entier donnant le nombre d'Arcs sortant dans ce Sommet.

- *pair<int, int> calcul_degres_entrant_sortant(Matrice M, Sommet S)* : Cette fonction regroupe les deux fonctions précédentes afin de calculer le degré entrant ainsi que le degré sortant du Sommet S passé en paramètre. La fonction retourne donc une paire d'entiers (le degré entrant et le degré sortant).

- *vector<int> coloration_Graphe(Graphe G)* : Coloration de Graphe grâce à l'algorithme DSATUR. La fonction prend en entrée le graphe à colorier et retourne un vecteur, où pour chaque position de ce vecteur, un numéro est attribué pour une coloration de Sommet.

- *int couleur_adjacente (Sommet S)* : Cette fonction est une sous fonction de la fonction coloration_Graphe. Elle calcule les couleurs disponibles pour un Sommet en fonction des Sommets adjacents à ce dernier.

- *vector<vector<int>> stables_Graphe(Matrice M)* : Calcule les stables présents dans un Graphe à partir de la Matrice représentant ce dernier. Cette fonction retourne un vecteur contenant la liste des identifiants des Sommets formant un stable.

- *vector<vector<int>> cliques_Graphe(Matrice M)* : Calcule les cliques présentes dans un Graphe à partir de la Matrice représentant ce dernier. Cette fonction retourne un vecteur contenant la liste des identifiants des Sommets formant une clique.

- *vector<int> voisin_Sommet (Matrice M, int ID)* : Cette fonction calcule les voisins du Sommet d'identifiant ID. Elle prend en entrée la Matrice associée à un Graphe ainsi qu'un identifiant pour reconnaître le Sommet sur lequel on veut faire les calculs. Elle nous renvoie un vecteur rempli des identifiants des Sommets adjacents au Sommet passé en paramètres.

- *int gestion_flots (Graphe G, int ID_source, int ID_puit)* : Cette fonction utilise l'algorithme d'Edmond Karp, elle calcule le flot maximum d'un Graphe à partir d'un Graphe et de deux Sommets, le Sommet de départ (Sommet source) et le Sommet d'arrivée (Sommet puits), et renvoie le flot maximum sous forme d'entier.

- *vector<pert_row> calcul_posteriorite (vector<pert_row>)* : Cette fonction calcule les contraintes de postériorités pour chaque pert_row du vecteur passé en argument. Elle renvoie le même vecteur avec les vecteurs taches_posterieures mis à jours.

- *Graphe pert (vector<pert_row>)* : La fonction crée pour chaque tâche un ou plusieurs Sommets et Arcs modélisant les contraintes d'antériorités, le déroulement de la tâche ainsi que la fin de la tâche. L'ensemble des tâches est donné en argument à travers le vecteur.

- *Graphe arborescence (Graphe G)* : Cette fonction prend en entrée le graphe courant et retourne le graphe correspondant à son arborescence. Si aucune arborescence n'existe alors le graphe renvoyé aura comme nombre de sommets la valeur -1.

- *Graphe anti_arborescence (Graphe G)* : Cette fonction prend en entrée le graphe courant et retourne le graphe correspondant à son anti-arborescence. Si aucune anti-arborescence n'existe alors le graphe renvoyé aura comme nombre de sommets la valeur -1.

- *int connexite (Matrice M)* : Recherche de la connexité d'un Graphe à l'aide de l'arborescence et l'anti-arborescence de l'arbre du Graphe. Elle prend en entrée la Matrice d'adjacence sur laquelle faire la recherche, elle retourne un entier indiquant si le Graphe est connexe ou non.

- *vector<vector<int>> chaine_euleriennne(Matrice M)* : Cette fonction prend en entrée un Graphe et effectue une recherche de chemins eulériens dans le Graphe. Si elle en trouve alors la fonction retourne la liste dans l'ordre des Sommets compris dans cette chaîne eulérienne.

- *vector<vector<int>> chaine_hamiltonienne(Matrice M)* : Cette fonction prend en entrée une matrice d'adjacence et effectue une recherche de chemins hamiltoniens dans le Graphe. Si elle en trouve alors la fonction retourne la liste dans l'ordre des Sommets compris dans cette chaîne hamiltonienne.

- *vector<int> postier_chinois(Matrice M)* : Cette fonction résout le problème du postier chinois en l'appliquant sur la Matrice d'adjacence passée en paramètres. En sortie, la fonction retourne un vecteur contenant les identifiants des Sommets contenus dans le cycle eulérien dans l'ordre du parcours.

- *vector<int> Voyageur_de_commerce(vector<int>, Matrice M)* : La fonction résout le problème du voyageur de commerce, elle prend en argument un vecteur d'ID de Sommets ainsi qu'une Matrice d'adjacence. Les ID des Sommets présents dans le vecteur sont ceux sur lesquels l'algorithme de Little va itérer. La fonction retourne une liste des identifiants des Sommets, correspondant au plus court chemin passant par chacun de ces Sommets.

Tests d'Opération sur les Graphes:

- TEST_CASE("Test de Bellman ", "[graphe]") : Ce test prendra en paramètre un Graphe G dont on connaît déjà le plus court chemin puis on vérifiera que la paire retournée par la fonction correspond bien à ce chemin.

- TEST_CASE("test de Floyd_Warshall", "[graphe]") : De la même façon que sur le test précédent, ici sera déclarée une Matrice décrivant un graphe dont on connaît d'ores et déjà le plus court chemin. Une paire de Matrices décrivant ce résultat sera aussi déclarée permettant de la comparer avec le retour effectif du test.

- TEST_CASE("test des degres", "[graphe]") : Dans ce test les trois fonctions liées au calcul de degrés seront testées en simultané. Pour cela une Matrice d'adjacence décrivant un Graphe et un sommet S dudit Graphe. Le degré du Sommet sera connu à l'avance et on vérifiera que calc_deg_ent et calc_deg_sor renvoient le bon degré chacun. Puis que calc_deg_ent_sor renvoie bien une paire avec ces deux degrés aux bonnes positions. (entrant en position 0 et sortant en position 1).

- TEST_CASE("test coloration", "[graphe]") : Une Matrice décrivant un Graphe G dont la coloration est connue est déclarée. Le vecteur de retour est comparée avec la coloration de la Matrice. Le test est validé s'il s'agit de la même.
- TEST_CASE("test des stables", "[graphe]") : Une Matrice d'adjacence décrivant un Graphe G dont on connaît le nombre de stables est déclarée. Un vecteur de vecteurs d'entiers décrivant le résultat déjà connu sera comparé avec celui renvoyé par la fonction. S'ils sont identiques le test est validé.
- TEST_CASE("test des cliques", "[graphe]") : Une Matrice d'adjacence décrivant un Graphe G dont on connaît le nombre de cliques est déclarée. Un vecteur de vecteurs d'entiers décrivant le résultat déjà connu sera comparé avec celui renvoyé par la fonction. S'ils sont identiques le test est validé.
- TEST_CASE("test des voisins", "[graphe]") : Une Matrice d'adjacence M est déclarée et l'ID d'un de ses Sommets dont on connaît les voisins est choisi. La liste des voisins du Sommet est déclarée sous forme de vecteur d'entiers (liste d'ID). La Matrice M passe dans la fonction avec l'ID du Sommet. Si la liste renvoyée par la fonction correspond à celle déclarée plus haut alors le test est validé.
- TEST_CASE("test de la gestion de flots", "[graphe]") : Un Graphe G, puits et sources sont choisis sur ce dernier de façon à ce que le flot maximum de ce parcours soit connu. Si la fonction renvoie la même valeur que celle prévue plus haut, alors le test est validé.
- TEST_CASE("test de l'ordonnancement", "[graphe]"): On crée un vecteur de pert_row dont on connaît le Graphe d'ordonnancement correspondant. Si l'objet graphe retourné est identique, le test est validé.
- TEST("test de l'arborescence", "[graphe]") : On crée un Graphe dont on connaît l'arborescence, arborescence qu'on va décrire dans un objet Graphe. Si le Graphe retourné par la fonction est identique à celui créé plus haut alors le test est validé.
- TEST_CASE("test de l'anti-arborescence", "[graphe]") : On crée un Graphe dont on connaît l'anti-arborescence, arborescence qu'on va décrire dans un objet Graphe. Si le Graphe retourné par la fonction est identique à celui créé plus haut alors le test est validé.
- TEST_CASE("test detection erreur arborescence", "[graphe]") : On va créer un Graphe G qui ne remplit pas les conditions de "pré-algorithme" puis on va le passer dans la fonction. Si le retour de fonction est le Graphe d'erreur (un Graphe ayant -1 sommet) alors le test validé.
- TEST_CASE("test detection erreur anti-arborescence", "[graphe]"): On va créer un Graphe G qui ne remplit pas les conditions de "pré-algorithme" puis on va le passer dans la fonction. Si le retour de fonction est le Graphe d'erreur (un Graphe ayant -1 sommet) alors le test est validé.
- TEST_CASE("test de la connexite", "[graphe]") : Nous allons créer un Graphe G connexe. Si le retour de la fonction est correct dans chaque cas, nous estimons que le test est validé, de même pour un graphe non connexe.
- TEST_CASE("test des chaines ", "[graphe]"): Test de la fonction "eulerien" et test de la fonction "hamiltonien".

  Nous allons créer un Graphe G dont les chaînes hamiltoniennes et eulériennes sont connues et vérifier les listes renvoyées par les deux fonctions. Si les chaînes renvoyées sont correctes, le test est réussi.
- TEST_CASE("test detection erreur des chaines", "[graphe]"): Nous ferons comme pour le test des chaînes avec un Graphe G' qui n'est ni hamiltonien ni eulérien. Si les listes renvoyées par les fonctions "eulerien" et "hamiltonien" sont vides, alors le test est validé.
- TEST_CASE("test du postier chinois", "[graphe]"): Test de la fonction "postier_chinois". La fonction prend en entrée un Graphe dans lequel, on sait qu'on peut appliquer l'algorithme du postier chinois. On teste, pour un degré total impair, si la création d'un Graphe G' est bien effectuée. On vérifie également, si l'algorithme renvoie bien un vecteur de listes d'ID.
- TEST_CASE("test detection erreur postier chinois","[graphe]") : Test de la fonction "postier_chinois" sur un graphe où il ne peut pas s'appliquer. La fonction prend en entrée un graphe sur lequel elle ne peut s'appliquer. L'algorithme doit renvoyer un vecteur vide.
- TEST_CASE("test du voyageur de commerce", "[graphe]") : On fait un test pour vérifier que la Matrice prise en entrée est complète (C'est-à-dire qu'il y a un arc entre chaque paire de Sommets pris en entrée). On déclare une Matrice décrivant un Graphe sur lequel on connaît le résultat du voyageur de commerce en fonction de certains Sommets. Les ID de ses Sommets seront passés dans un vecteur d'entiers. La liste des Sommets décrivant le plus court chemin sera elle aussi déclarée. Si le résultat de la fonction correspond avec ce dernier alors le test est validé.

4.4 Interface graphique

Ce module a pour rôle l'affichage des données traitées par l'application. Il y aura donc un flux d'informations du module Opération sur les Graphes afin d'afficher le Graphe à l'utilisateur. Il y aura aussi utilisation des informations du module Opération sur les Graphes (pair<vector<vector<int>>,vector<int>>,pair<Matrice,Matrice>,vector<int>,pair<int,int>, Graphe, vector<vector<int>>,vector<pert_row>) pour proposer à l'utilisateur quel algorithme il souhaite appliquer. Pour produire l'interface graphique nous avons décidé de nous tourner vers Qt, véritable référence de l'interface graphique en C++.

Ce choix a été motivé par différents facteurs :

- Une documentation importante et très claire présente sur le site de l'éditeur couvrant l'intégralité de Qt.
- Le fonctionnement de Qt est facilement compréhensible. De nombreux outils existent pour travailler avec Qt tel que Qt Creator.
- Le fait que Qt soit une bibliothèque écrite en C++ permet de préserver les avantages du langage.
- Le fait que Qt soit utilisé dans le monde professionnel sera valorisable pour notre intégration sur le marché du travail.

Enfin, ce module recevra aussi les possibilités de gestion des fichiers relatifs au Graphe traité. Il y aura donc échange de données entre ce module et le module Système de Gestion de Fichiers (Graphe) afin de proposer à l'utilisateur quelles actions il souhaite réaliser (enregistrement, suppression ou modification du fichier). L'interface graphique sera découpée en plusieurs fenêtres. Il y a aussi envoi d'instructions au module de gestion de graphe sous forme d'objets à modifier (Sommet, Arc, Matrice et Graphe). Et enfin des instructions peuvent être envoyées au module opération sur les graphes afin de modifier le Graphe courant.

4.4.1 La fenêtre principale

```
class Ui_MainWindow
{
    public:
        /* Attributs des menus deroulants */

        QMenuBar *menubar;
        QMenu *menuFichier;
        QMenu *menuEdition;
        QMenu *menuAlgorithmes;
        QMenu *menuCalcul_du_plus_court_chemin;
        QMenu *menuCalcul_des_degres_entrant_ou_sortant_dun_sommet;
        QMenu *menuTrouver_arborescence_anti_arborescence;
        QMenu *menuAide;


        // Menu FICHIER //
        QAction *actionNouveau_graphe;
        QAction *actionNouveau_graphe_aleatoire;
        QAction *actionEnregistrer;
        QAction *actionCharger;
        QAction *actionEnrengistrer_sous;

        // Menu EDITION //
        QAction *actionDupliquer_graphe;
        QAction *actionSupprimer_graphe;
        QAction *arrangerSommet;
        QAction *extraireSousGraphe;
        QAction *fermerGraphe;

        // Menu ALGORITHMES //
        QAction *actionFord_Bellman;
        QAction *actionFloyd_Warshall;
        QAction *actionDegr_sortant;
        QAction *actionDegr_entrant;
        QAction *actionDegrs_entrant_et_sortant;
        QAction *actionColoration_de_graphe;
        QAction *actionD_Termination_de_stables;
        QAction *actionD_Termination_de_cliques;
        QAction *actionVoisins_de_sommets;
        QAction *actionGestion_de_flots;
        QAction *actionCreer_un_graphe_dordonnancement;
        QAction *actionArborescence;
        QAction *actionAnti_Arborescence;
        QAction *actionRecherche_de_la_connexite;
        QAction *actionTrouver_chaine_eulerienne;
        QAction *actionTrouver_chaine_hamiltonienne;
        QAction *actionR_Postier_chinois;
        QAction *actionR_Voyageur_de_commerce;

        // Menu AIDE //
        QAction *actionDocumentation;
```

```
        QAction *actionGithub;

        /* Onglet */

        /* Liste des boutons permettant d'ajouter ou de supprimer des elements du dessin */

        QRadioButton *addSommetButton;
        QRadioButton *addArcButton;
        QRadioButton *selectButton;
        QRadioButton *deleteSommetButton;
        QRadioButton *deleteArcButton;

        /* Attributs des onglets */

        QTabWidget *tabWidget;
        QWidget *tab;
        QZoneDeDessin *zoneDessin;
        QTextBrowser *caraSelection;
        QTextBrowser *console;
        QStatusBar *statusbar;

        void setupUi(QMainWindow *MainWindow)
        void retranslateUi(QMainWindow *MainWindow)
};
```

Les menus:

Attributs des menus déroulants:

- *QMenuBar *menubar* : Cet objet englobe l'ensemble des menus déroulants détaillés ci-dessous.
- *QMenu *menuFichier* : Cet objet représente le menu qui concerne les fichiers, il permet de créer un nouveau Graphe, d'ouvrir un nouveau Graphe, d'enregistrer un Graphe et d'enregistrer le Graphe dans un fichier précis.
- *QMenu *menuEdition* : Cet objet représente le menu déroulant permettant l'édition d'un Graphe, tel que la suppression et duplication.
- *QMenu *menuAlgorithmes* : Cet objet représente le menu déroulant proposant tous les algorithmes, que l'application possède, applicables au Graphe.
- *QMenu *menuCalcul_du_plus_court_chemin* : Cet objet représente le double menu déroulant proposant les différents algorithmes de calcul de plus court chemin.
- *QMenu *menuCalcul_des_degres_entrant_ou_sortant_dunsommet* : Cet objet représente le double menu déroulant qui propose de sélectionner le calcul des Sommets entrants ou bien des Sommets sortants.
- *QMenu *menuTrouver_arborescence_anti_arborescence* : Cet objet représente le double menu déroulant permettant de sélectionner la fonction calculant l'arborescence, ou l'anti-arborescence.
- *QMenu *menuAide* : Cet objet représente le menu déroulant d'aide proposant la documentation du projet ainsi que le GitHub.

- *QAction *actionNouveau_graphe* : Ce bouton permet la création d'un nouveau graphe vide et l'ouverture d'un nouvel onglet.
- *QAction *actionNouveau_graphe_aleatoire* : Ce bouton permet la création d'un nouveau Graphe avec un nombre de Sommets aléatoirement définis dans une fenêtre et l'ouverture d'un nouvel onglet.
- *QAction *actionCharger* : Ce bouton permet d'ouvrir un Graphe déjà créé.
- *QAction *actionEnregistrer* : Ce bouton permet de sauvegarder un Graphe.
- *QAction *actionEnrengistrer_sous* : Ce bouton permet d'enregistrer sous un Graphe.
- *QAction *actionDupliquer_graphe* : Ce bouton permet la création d'un Graphe.
- *QAction *actionSupprimer_graphe* : Ce bouton permet la suppression d'un Graphe.
- *QAction *arrangerSommet* : Ce bouton applique Force Atlas 2 sur le Graphe courant.
- *QAction *extraireSousGraphe* : Ce bouton permet d'extraire un sous Graphe du Graphe courant grâce aux Sommets selectionnés.

- *QAction *fermerGraphe* : Ce bouton permet de fermer l'onglet sans sauvegarder.
- *Le menu des algorithmes* : Les objets composant ce menu permettent l'appel d'un algorithme sélectionné.

- *QAction *actionDocumentation* : Ce bouton renvoie vers la documentation du projet.
- *QAction *actionGithub* : Ce bouton renvoie le lien Github du projet.

Les boutons de la fenêtre:

- *QRadioButton *addSommetButton* : Ce bouton permet d'ajouter un Sommet au Graphe en cliquant dessus.
- *QRadioButton *addArcButton* : Ce bouton permet de créer un Arc entre 2 Sommets.
- *QRadioButton *selectButton* : Ce bouton permet de selectioner des Sommet ou des Arcs dans le graphe en cliquant dessus.
- *QRadioButton *deleteSommetButton* : Ce bouton permet de supprimer les Sommets en cliquant sur ces derniers.
- *QRadioButton *deleteArcButton* : Ce bouton permet de supprimer les Arcs sur lesquels on clique.

Les onglets de la fenêtre:

- *QTabWidget *tabWidget* : Ce widget contient tous les onglets.
- *QWidget *tab* : Ce widget permet de manipuler un onglet.
- *QZoneDeDessin *zoneDessin* : Zone de dessin de Graphe sur chaque onglet.
- *QTextBrowser *caraSelection* : Représente les caractéristiques de la sélection.
- *QTextBrowser *console* : Console affichant les résultats des différentes fonctionnalités.
- *QStatusBar *statusbar* : Barre de statut.

Les fonctions de la fenêtre:

- *void setupUi (QMainWindow *MainWindow)* : Cette fonction utilise les setters des attributs de MainWindow pour leur affecter les valeurs par défaut.
- *void retranslateUi(QMainWindow *MainWindow)* : Cette fonction affecte les textes des éléments, la séparation est rendue nécessaire par les fonctionnalités de traduction de Qt.

```
QT_BEGIN_NAMESPACE
namespace Ui {
    class MainWindow;
}
QT_END_NAMESPACE
class MainWindow : public QMainWindow
{
    Q_OBJECT

    private:
        Ui::MainWindow *ui;
        Graphe grapheCourant;
        int dernierBoutonEnclenche;


    public:
        MainWindow(QWidget *parent = nullptr);
        ~MainWindow();

        int printConsole(string nomMethode, string valRetFunc);
        int printCaraSelection();
        int ajouterOnglet(Qstring nomOnglet, Graphe G);
        int supprimerOnglet(Qstring nomOnglet);


    public slots:
        void nv_graphe_vide();
```

```
        void nv_graphe_aleatoire();
        void Enregistrer();
        void Charger();
        void Enregistrer_sous();
        void Dupliquer_graphe();
        void Supprimer_graphe();
        void Ford_Bellman();
        void Floyd_Warshall();
        void Degr_sortant();
        void Degr_entrant();
        void Degrs_entrant_et_sortant();
        void Coloration_de_graphe();
        void Determinaison_de_stables();
        void Determinaison_de_cliques();
        void Voisins_de_sommets();
        void Gestion_de_flots();
        void Creer_un_graphe_dordonnancement();
        void Arborescence();
        void AntiArborescence();
        void Recherche_de_la_connexite();
        void Trouver_chaine_eulerienne();
        void Trouver_chaine_hamiltonienne();
        void Postier_chinois();
        void Voyageur_de_commerce();
        void Documentation();
        void Github();
        void extraireSousGraphe();
        void arrangerSommets();
        void fermer_graphe();
        void DBEselection();
        void DBEaddSommet();
        void DBEaddArc();
        void DBEdeleteSommet();
        void DBEdeleteArc();
};
```

Les attributs de la classe:

- *QT_BEGIN_NAMESPACE*
  namespace Ui {
  class MainWindow;
  }: Permet de renommer la classe MainWindow écrite dans UI_MainWindow en "ui"
- *Ui::MainWindow *ui* : Fenêtre de l'Interface.
- *Graphe grapheCourant* : Ce Graphe permet d'avoir en temps réel un Graphe affiché à l'écran, il change lors du changement d'onglet.
- *int dernierBoutonEnclenche* : Cet entier permet de sauvegarder le dernier bouton enclenché:
  - 0 : aucune valeur de base
  - 1 : selection
  - 2 : ajout de Sommet
  - 3 : ajout d'Arc
  - 4 : suppression de Sommet
  - 5 : suppression d'Arc

Les méthodes de la classe:
Au cours de ces détails de méthodes et de tests, nous parlerons du Graphe étudié à l'instant présent par l'application. Ce Graphe sera appelé sous le nom de "Graphe courant".

- *MainWindow(QWidget *parent = nullptr)* : Constructeur de la classe.
- $\sim$ *MainWindow()* : Destructeur de la classe.

- *int printConsole(string nomMethode, string valRetFunc)* : Cette méthode sera appelée dans une fonction pour afficher le résultat de la fonction dans la console. Elle prend en paramètres le nom de la fonction appelante et le résultat renvoyé par celle-ci qui pourra être casté. Elle retourne 0 en cas de succès et -1 en cas d'erreur.
- *int printCaraSelection()* : Cette méthode est appelée par les fonctions de sélection des sommets et affiche en temps réel dans l'espace d'affichage les caractéristiques de la sélection. Elle retourne 0 en cas de succès, -1 en cas d'erreur.
- *int ajouterOnglet(Qstring nomOnglet, Graphe G)* : Cette méthode permet d'ouvrir un nouvel onglet respectant le placement (zone de dessin, spécification de la selection et console) et lui affecte le nom placé en paramètre. Le choix de mettre un Qstring plutôt qu'un string s'explique par le fait qu le comportement des deux objets est très similaires et l'utilisation d'une Qstring simplifie le nommage de l'onglet dans la fonction (nativement le type du nom d'un onglet est la Qstring). Le Graphe G sera dessiné dans la zone de dessin de l'onglet. L'attribut "graphe_courant" va se voir affecté le Graphe G et l'attribut "DernierBoutonEnclenche" est mis à 0. Cette méthode retourne l'index de l'onglet si cela réussi, -1 dans l'autre cas.
- *int supprimerOnglet(Qstring nomOnglet)* : Cette méthode va rechercher l'onglet portant le nom passé en paramètre puis va le fermer. Elle retourne l'index de l'onglet supprimé si cela se déroule normalement, sinon elle retounera -1.

- *void nv_graphe_vide()* : Cette méthode va ouvrir un nouvel onglet en passant un Graphe vide en paramètres.
- *void nv_graphe_aleatoire()* : Cette méthode va ouvrir en premier lieu une fenêtre de type QInputDialog permettant de récupérer le nombre de Sommets à créer aléatoirement. Ensuite le nombre demandé de Sommets va être créé et un certain nombre d'entre eux seront liés. Un nouvel onglet sera ensuite ouvert avec le Graphe ainsi créé en paramètre. Pour finir, la méthode "force_Atlas2" sera appelée et permettra d'éloigner les Sommets les uns des autres.
- *void Enregistrer()* : Cette méthode va appeler la fonction de sauvegarde sur le chemin présent dans GrapheCourant.path et remplacer le fichier déjà existant s'il existe.
- *void Charger()* : Cette méthode va ouvrir le gestionnaire de fichier du système d'exploitation permettant de choisir quel fichier ouvrir en utilisant la fonction load(). Cette fonction va ouvrir un nouvel onglet pour y afficher le Graphe ouvert.
- *void Enregistrer_sous()* : Cette méthode va permettre de changer le chemin de sauvegarde du Graphe en appelant la fonction éponyme dans gestion de fichier. Cette méthode va ouvrir une fenêtre de type QInputDialog pour récupérer le nom.
- *vid Dupliquer_graphe()* : Cette méthode va créer un nouvel onglet et y mettre une copie du Graphe courant. Pour cela le nouvel onglet aura comme paramètre l'étiquette du Graphe courant et le Graphe courant.
- *void Supprimer_graphe()* : Cette méthode supprime le Graphe courant en appelant le destructeur du Graphe courant. Ferme l'onglet et supprime le fichier trouvable si on suit le chemin graphe.path
- *void Ford_Bellman()* : Cette méthode appelle la fonction "calc_pcc_Bellman" de calcul de plus court chemin par l'algorithme de Ford-Bellman, en lui passant en paramètre une sélection de deux Sommets ainsi que la Matrice d'adjacence du Graphe courant. La méthode "convert_to_matrice_adj" est utilisée. Le résultat est affiché dans la console.
- *void Floyd_Warshall()* : Cette méthode appelle la fonction "calc_pcc_Floyd_Warshall" et va lui passer en paramètre la Matrice d'adjacence décrivant le Graphe courant en utilisant la méthode "convert_to_matrice_adj". Le résultat est affiché dans la console.
- *void Degr_sortant()* : Cette méthode appelle la fonction "calc_deg_sortant" en lui passant en paramètre les/le sommet/s correspondant/s aux ID sélectionné/s dans la "QZoneDeDessin" si la liste est composée de plusieurs ID alors la fonction "calc_deg_sortant" sera appelée successivement sur chacun des Sommets dont l'ID est présent dans le vecteur. Le résultat est affiché dans la console.
- *void Degr_entrant()* : Cette méthode appelle la fonction "calc_deg_entrant" en lui passant en paramètre les/le sommet/s correspondant aux ID sélectionné/s dans la "QZoneDeDessin" si la liste est composée de plusieurs ID alors la fonction "calc_deg_entrant" sera appelée successivement sur chacun des Sommets dont l'ID est présent dans le vecteur. La Matrice d'adjacence décrivant le Graphe courant sera aussi passée en paramètre. Le résultat est affiché dans la console.
- *void Degrs_entrant_et_sortant()* : Cette méthode appelle la fonction "calc_deg_ent_sor" en lui passant successivement les Sommets dont l'ID est présent dans la liste des Sommets selectionnés et le Graphe courant décrit par sa Matrice d'adjacence en paramètre. Affiche le résultat dans la console.
- *void Coloration_de_graphe()* : Cette méthode appelle la fonction "color_graphe" en lui passant en paramètre la Matrice d'adjacence du Graphe courant grâce à la méthode "convert_to_matrice_adj". Le résultat est affiché sur le Graphe courant par coloration des Sommets.
- *void Determinaison_de_stables()* : Cette méthode appelle la fonction de "stables_graphe" et détermine le nombre de stables dans le Graphe courant, puis affiche celui-ci dans la console et les colorie sur le Graphe.
- *void Determinaison_de_cliques()* : Cette méthode appelle la fonction "cliques_graphe" avec en paramètre la Matrice d'adjacence du Graphe courant en utilisant la méthode "convert_to_matrice_adj". Le résultat est affiché dans la console et sur le Graphe courant, par coloration des différentes cliques.

- *void Voisins_de_sommets()* : Cette méthode appelle la fonction "voisin_sommet" avec en paramètre la matrice d'adjacence du Graphe courant par appel de la méthode "convert_to_matrice_adj". Si plusieurs sommets sont sélectionnés la fonction sera exécutée pour chacun d'eux.
- *void Gestion_de_flots()* : Cette méthode appelle la fonction "Edmond_Karp" et lui passe le Graphe courant en paramètre. Les ID des Sommets source et puits seront les deux derniers sommets de "selected_list". Affiche le flot maximum dans la console.
- *void Creer_un_graphe_dordonnancement()* : Cette fonction va ouvrir une seconde fenêtre permettant d'ajouter les tâches. Une fois ceci terminé un nouvel onglet sera ouvert avec comme Graphe courant le Graphe PERT correpondant.
- *void arborescence()* : Cette fonction va ouvrir un nouvel onglet avec l'arborescence du Graphe courant. Pour la générer, la fonction "arborescence" du module opération sur les Graphes sera appelée. Puis un nouvel onglet sera ouvert avec le graphe ainsi généré.
- *void antiArborescence()* : Va ouvrir un nouvel onglet avec l'anti-arborescence du Graphe courant. Pour la générer, la fonction "anti_arborescence" du module opération sur les graphes sera appelée.Puis un nouvel onglet sera ouvert avec le graphe ainsi généré.
- *void recherche_de_la_connexite()* : Cette fonction appelle la fonction "connexite" et lui passe en paramètre la Matrice d'adjacence du Graphe courant grâce à la méthode "convert_to_matrice_adj". Cette fonction vérifie si à partir d'un sommet, il est possible d'atteindre les autres Sommets. Si ce n'est pas le cas pour un ou plusieurs Sommets, il y a donc présence de composante(s) non connexes. Le nombre de composantes connexe est affiché à la console.
- *void trouver_chaine_eulerienne()* : Cette fonction appelle la fonction "eulerien" avec en paramètre la Matrice d'adjacence du Graphe courant en utilisant la méthode "convert_to_matrice_adj". La liste des chemins eulériens du Graphe est affichée dans la console.
- *void trouver_chaine_hamiltonienne()* : Cette fonction appelle la fonction "hamiltonien" en lui passant la Matrice d'adjacence du Graphe courant en paramètre par utilisation de la méthode "convert_to_matrice_adj". La liste des chemins hamiltoniens du Graphe est affichée dans la console.
- *void postier_chinois()* : Cette fonction appelle la fonction "postier_chinois" du module Opération sur les Graphes avec en paramètre la matrice d'adjacence du Graphe courant en utilisant la fonction "convert_to_matrice_adj". Le résultat renvoyé par cette fonction est affiché dans la console.
- *void Voyageur_de_commerce()* : Cette méthode appelle la fonction "Voyageur_commerce" du module d'Opération sur les Graphes. Le vecteur de Sommet passé en paramètre sera déterminé par la sélection de Sommets, la Matrice passée en paramètre est la Matrice d'adjacence du Graphe courant. La liste des Sommets correspondants au plus court chemin passant par chacun de ces Sommets sera affichée dans la console.
- *void Documentation()* : Cette méthode affiche un lien renvoyant sur la documentation complète du projet.
- *void Github()* : Cette méthode affiche un lien renvoyant sur la page Github du projet.
- *void extraireSousGraphe()* : Cette méthode va récupérer le Graphe courant et la liste des Sommets sélectionnés (dans la zone de dessin courante) puis va ouvrir un nouvel onglet avec le Graphe issue de la sélection des Sommets et du Graphe courant.
- *void arrangerSommets()* : Cette méthode va lancer la méthode Force atlas 2 sur la zone dessin pour ré-arranger les Sommets à la demande de l'utilisateur.
- *void fermer_graphe()* : Cette méthode ferme l'onglet courant et détruit le Graphe courant, si un autre onglet existe le Graphe courant devient le Graphe dessiné dans cet onglet.
- *void DBEselection()* : Cette méthode va modifier "dernierBoutonEnclenche" et va le mettre sur 1.
- *void DBEaddSommet()* : Cette méthode va modifier "dernierBoutonEnclenche" et va le mettre sur 2.
- *void DBEaddArc()* : Cette méthode va modifier "dernierBoutonEnclenche" et va le mettre sur 3.
- *void DBEdeleteSommet()* : Cette méthode va modifier "dernierBoutonEnclenche" et va le mettre sur 4.
- *void DBEdeleteArc()* : Cette méthode va modifier "dernierBoutonEnclenche" et va le mettre sur 5.

Tests de la classe Mainwindow:

- void testMainWindow() : Teste si la MainWindow est créée.
- void testDestructeur() : Teste si une MainWindow créée est bien détruite après l'appel du destructeur.
- void testprintConsole() : Compare les QString de la console avec ce que l'on met en argument dans la fonction "printConsole()".

- void testprintCaraSelection() : Test comparant les QString de la zone d'affichage avec les QString attendu dans notre cas de test.
- void testajouterOnglet : Test qui vérifie qu'un Widget tab est bien créé avec le QString que l'on passe en argument et que le Graphe passé en argument est bien dessiné dans la zone de dessin.
- void testsupprimerOnglet() : Test qui créé un nouvel onglet puis le supprime. Vérifie si le code de retour est le bon et si le widget a bien été supprimé.
- void testChangeOnglet() : Va simuler un changement d'onglet et vérifier que l'attribut "graphe_courant" de mainwindow est bien changé par le graphe présent dans l'attribut "graphe_dessine" de la zone de dessin de l'onglet.
- void testnv_graphe_vide() : Teste si la fonction créé un graphe vide dans un nouvel onglet.
- void testnv_graphe_aleatoire() : Teste si la fonction créé un graphe avec le bon nombre de Sommet a été créé dans un nouvel onglet.
- void testenregistrer() : Teste si la fonction enregistrer du module gestion de fichier a bien été appelée.
- void testcharger() : Teste si la fonction charger du module gestion de fichier a bien été appelée.
- void testenregistrer_sous() : Teste si la fonction enregistrer_sous du module gestion de fichier a bien été appelée.
- void testdupliquer_graphe() : Vérifie si le graphe créé dans le nouvel onglet est bien équivalent à celui du premier onglet.
- void testsupprimer_graphe() : Vérifie si le fichier et l'onglet correspondant a un Graphe créé ont bien été supprimés.
- void testford_Bellman() : Test vérifiant si la fonction "calc_pcc_Bellman" du module opérations sur les Graphes a bien été appellé et si les résultats s'affichent dans la console.
- void testfloyd_Warshall() : Test vérifiant si la fonction "calc_pcc_Floyd_Warshall" du module opérations sur les Graphes a bien été appelée et si le résultat s'affiche dans la console.
- void testdegr_sortant() : Test vérifiant si la fonction "calc_deg_sortant" du module opérations sur les Graphes a bien été appelée et si le résultat s'affiche dans la console.
- void testdegr_entrant() :Test vérifiant si la fonction "calc_deg_entrant" du module opérations sur les Graphes a bien été appelée et si le résultat s'affiche dans la console.
- void testdegrs_entrant_et_sortant() : Test vérifiant si la fonction "calc_deg_ent_sor" du module opérations sur les Graphes a bien été appelée et si le résultat s'affiche dans la console.
- void testcoloration_de_graphe() : Test verifiant si la fonction "color_graphe" du module opérations sur les Graphes a bien été appelée et si les bons Sommets sont colorés.
- void testdetermination_de_stables() : Test vérifiant si la fonction "stables_graphe" du module opérations sur les Graphes a bien été appelée. Vérifie aussi si les bons Sommets sont colorés et si les résultats sont affichés dans la console.
- void testdetermination_de_cliques() : Test vérifiant si la fonction "convert_to_matrice_adj" du module opérations sur les Graphes a bien été appelée. Vérifie aussi si les bons Sommets sont coloriés et que les résultats sont affichés dans la console.
- void testvoisins_de_sommets() : Test vérifiant si la fonction "calc_deg_ent_sor" du module opérations sur les Graphes a bien été appelée et si le résultat s'affiche dans la console.
- void testgestion_de_flots() : Test vérifiant si la fonction "edmond_Karp" du module opérations sur les Graphes a bien été appelée et si le résultat s'affiche dans la console.
- void testcreer_un_graphe_d_ordonnancement() : Test qui vérifie si la fenêtre permettant d'ajouter les tâches s'affiche et si la fonction "pert" du module opérations sur les Graphes a été appelée. Vérifie si le bon Graphe a bien été créé dans un nouvel onglet.
- void testarborescence() : Test vérifiant si la fonction "arborescence" du module opération sur les Graphes est appelée et si le Graphe créé s'affiche correctement dans un nouvel onglet.
- void testantiArborescence() : Test vérifiant si la fonction "anti_arborescence" du module opération sur les Graphes est appelée et si le Graphe créé s'affiche correctement dans un nouvel onglet.
- void testrecherche_de_la_connexite() : Test qui vérifie si la fonction "connexite" du module opérations sur les graphes a été appelée et si le résultat est bien affiché dans la console.
- void testtrouver_chaine_eulerienne() : Test qui vérifie si la fonction "eulerien" du module opérations sur les Graphes a été appelée et si le résultat est bien affiché dans la console.
- void testtrouver_chaine_hamiltonienne() :Test qui vérifie si la fonction "hamiltonien" du module opérations sur les Graphes a été appelée et si le résultat est bien affiché dans la console.

- void testpostier_chinois() : Test qui vérifie si la fonction "postier_chinois" du module opérations sur les Graphes a été appelée et si le résultat est bien affiché dans la console.
- void testvoyageur_de_commerce() : Test qui vérifie si la fonction "voyageur_commerce" du module opérations sur les Graphes a été appelée et si le résultat est bien affiché dans la console.
- void testextraireSousGraphe() : Test vérifiant si le bon sous-Graphe est récupéré et affiché dans un nouvel onglet.
- void testarrangerSommets() : Test vérifiant si la méthode "Force_Atlas2" a bien été appelée.
- void testDBEselection() : Teste si la fonction modifie bien la valeur de l'entier dernierBoutonEnclenche a 1.
- void testDBEaddSommet() : Teste si la fonction modifie bien la valeur de l'entier dernierBoutonEnclenche a 2.
- void testDBEaddArc() : Teste si la fonction modifie bien la valeur de l'entier dernierBoutonEnclenche a 3.
- void testDBEdeleteSommet() : Teste si la fonction modifie bien la valeur de l'entier dernierBoutonEnclenche a 4.
- void testDBEdeleteArc() : Teste si la fonction modifie bien la valeur de l'entier dernierBoutonEnclenche a 5.

4.4.2 Zone de dessin :

```
class QZoneDeDessin : public QGraphicView {
    Q_OBJECT

private:
     QGraphicsScene * scene;
     vector<int> selected_list;
     Graphe graphe_dessine;

public :
    explicit QZoneDeDessin(QWidget *parent = 0);
    void force_Atlas2();
    pair<int,int> distance(QSommet a, QSommet b);
    void addSelect_Sommet(int ID);
    void deleteSelect_Sommet(int ID);
    void razSelected_list();
    void afficher_Graphe(Graphe G);
    void afficher_Sommet(int id);
    void afficher_arc(int id);

public slots :
    void mousePressEvent(QMouseEvent * e);
    void dessiner_sommet(int x, int y);
    void dessiner_arc(int xA, int yA, int xB, int yB);

};
```

Les attributs de la classe:

- *QGraphicsScene * scene* : Dans Qt, une QGraphicsScene fournit une surface permettant de gérer un grand nombre d'objet 2D.
- *vector<int> selected_list* : Cet attribut permet de stocker les sommets selectionnés.
- *Graphe graphe_dessine* : Cet attribut stocke le Graphe dessiné actuellement dans sa forme objet et permet égalemment de savoir ce qui est dessiné sur chaque Zone de Dessin.

Les méthodes de la classe:

- *explicit QZoneDeDessin(QWidget *parent = 0)* : Constructeur de la zone de dessin.
- *void force_Atlas2()* : Applique l'algorithme de force appelé par le bouton "arranger_bouton", qui permet de distancer les points sur le Graphe courant.
- *pair<int,int> distance(QSommet a, QSommet b)* : Renvoie la distance en pixel entre 2 sommets sur l'axe des abscisses et des ordonnées. Le premier "int" de la paire est la différence en x et la seconde la différence en y.
- *void addSelect_Sommet(int ID)* : Ajoute l'ID d'un sommet a "selected_list".
- *void deleteSelect_Sommet(int ID)* : Supprime de la "selected_list" l'ID du sommet passé en paramètre.
- *void razSelected_list()* : Remet à 0 la "list_selected".

- *void afficher_Graphe(Graphe G)* : Cette fonction va grâce aux sous fonction "afficher_sommet" et "afficher_arc" générer, sans l'intervention de l'utilisateur, le graphe G sur la QZoneDeDessin.
- *void afficher_Sommet(int id)* : Fonction d'affichage qui va, à partir des informations contenues dans l'objet Sommet, dessiner le sommet dont on lui passe l'ID sur la QZoneDeDessin.
- *void afficher_arc(int id)* : Fonction d'affichage qui va, à partir des informations contenue dans l'objet Arc, dessiner l'arc dont on lui passe l'ID sur la QZoneDeDessin.
- *void mousePressEvent(QMouseEvent * e)* : Surcharge de la fonction s'enclenchant lorsque l'on clique sur la QgraphicView. On récupère les informations du mouseEvent tel que ses positions pour nous permettre de dessiner un objet sur cette position.
  - Si le bouton "addSommetButton" a été cliqué alors dessine un Sommet sur la position du curseur.
  - Si le bouton "addArcButton" a été cliqué alors dessine un Arc entre le précédent Sommet et celui venant d'être cliqué. Si le clic est dans le vide alors la liste des Sommets selectionnés est remise à zéro.
  - Si le bouton "selectButton" est enclenché et que "select" du Sommet cliqué est sur false alors l'ID du Sommet cliqué est ajouté à la liste des ID_selected et ce dernier change de couleur.
  - Si le bouton "selectButton" est enclenché et que "select" du Sommet cliqué est sur true alors l'ID du Sommet cliqué est supprimé de la "selected_list" et ce dernier redevient de sa couleur original.
  - Si le bouton "deleteSommet" est enclenché et que l'utilisateur clic sur un sommet alors ce dernier est supprimé du graphe avec tous ses arcs sortant.
  - Si le bouton "deleteArc" est enclenché et que l'utilisateur clic sur un arc alors ce dernier est supprimé du graphe.

  On met à jour l'affichage permettant d'afficher les objets dessinés et de remettre les Sommets désélectionnés à la bonne couleur ainsi que les Arcs dont le puits ou la source ont changés d'emplacement.
- *void dessiner_sommet(int x, int y)* : Cette méthode permet de dessiner sur la QGraphicView un Sommet en lui précisant les positions x et y. Ces dernières peuvent provenir soit du public slots "mousePressEvent" de ZoneDeDessin, soit de la fonction "afficher_sommet" qui va passer en paramètre les positions d'un point déjà défini par le passé. Cette fonction va aussi ajouter un Sommet à la liste des Sommets du Graphe courant.
- *void dessiner_arc(int xA, int yA, int xB, int yB)* : Cette méthode permet de dessiner sur la QGraphicView un Arc en lui précisant les positions x et y de son Sommet de départ et de son Sommet d'arrivée. Ces dernières peuvent provenir soit du public slots "mousePressEvent" et des positions d'un Sommet sélectionné de ZoneDeDessin, soit de la fonction "afficher_arc" qui va passer en paramètre les positions d'un Arc déjà défini par le passé. Cette fonction va aussi ajouter un Arc à la liste des Arcs du Graphe courant.

Tests de la classe Zone de Dessin :

- void testZoneDeDessin() : Teste si la zone de dessin est créée.
- void testDestructeur () : Teste si une zone de dessin créée est bien détruite après l'appel du destructeur.
- void testForce_Atlas2() : Test de l'algorithme de placement en créant un Graphe dont les Sommets sont surperposés. La fonction est ensuite apliquée et le test réussi, par vérification des coordonnées des Sommets qui devraient être bien positionnés.
- void testDistance() : Teste la distance sur deux Sommets dont nous connaissons déjà celle-ci, et validation du test si cette distance est correcte.
- void testAddSelect_Sommet() : ajout de l'ID connu d'un Sommet à une liste vide et validation du test par vérification de la "selected_liste".
- void testDeleteSelect_Sommet() : Appel de la fonction pour supprimer le sommet sélectionné. Le test sera réalisé avec l'ID d'un Sommet connu, contenu dans la "selected_liste". Validation du test si la "selected_liste" ne comporte plus l'ID de ce Sommet.
- void testRazSelected_list() : Test sur une liste non vide. Validation du test si celle-ci est à 0.
- void testAfficher_Graphe() : Test d'affichage. Ce test est réalisé à l'aide de la méthode itemAt de QGraphicView.
- void testAfficher_Sommet() : Test qui vérifie la position des Sommets d'un Graphe. Ce test est réalisé à l'aide de la méthode itemAt de QGraphicView.
- void testAfficher_arc() : Test qui vérifie la position des Arcs d'un Graphe. Ce test est réalisé à l'aide de la méthode itemAt de QGraphicView.
- void testDessiner_Sommet() : Test qui vérifie que le sommet de test est présent dans la scene.
- void testDessiner_Arc() : Test qui vérifie que l'arc de test est présent dans la scene.

- void mousePressEvent(): Test par simulation d'un clic sur l'objet. Vérification de l'état de l'attribut de mainwindow "dernierBoutonEnclenche". Si on clic sur un Sommet :
  - Pour la valeur 1, on vérifie que le Sommet sélectionné est ajouté à la liste des Sommets sélectionné. S'il avait déjà sélectionné alors il est supprimé de la liste.
  - Pour la valeur 2, on vérifie qu'un Sommet est crée sur les positions du curseur.
  - Pour la valeur 4, on vérifie que le Sommet cliqué est supprimé.

  Si on clic sur un Arc :
  - Pour la valeur 3, on vérifie qu'un Arc est dessiné entre le Sommet cliqué et le dernier Sommet selectionné.
  - Pour la valeur 5, on vérifie que l'Arc cliqué est supprimé.

  Si on clique dans le vide il faut vérifier que la liste est remise à 0

4.4.3 QSommet

```
class QSommet : public QGraphicsItem{

private:
    int id;
    int posx;
    int posy;
    int rayon;
    QColor coul;
    bool select;

public:
    QSommet(Sommet S);
    void mouseDoubleClickEvent(QGraphicsSceneMouseEvent *event);
    QRectF boundingRect();
    void paint(QPainter *painter, const QStyleOptionGraphicsItem *option, QWidget *widget);
    void setSelect(bool B);
    void ~QSommet();

};
```

Les attributs de la classe:
- *int id*: ID du Sommet à partir duquel il est construit.
- *int posx*: Représente la position x du Sommet sur la graphicView.
- *int posy*: Représente la position y du Sommet sur la graphicView.
- *int rayon*: Représente le rayon du Sommet en pixel.
- *QColor coul*: Représente la couleur du Sommet.
- *bool select*: Cet attribut permet de savoir si le Sommet est actuellement sélectionné.

Les méthodes de la classe:
- QSommet(Sommet S) : Constructeur d'un QSommet à partir d'un Sommet.
- void mouseDoubleClickEvent(QGraphicsSceneMouseEvent *event) : Cette méthode ouvre une fenêtre permettant de modifier l'étiquette ou la charge utile d'un Sommet en récupérant la paire renvoyée par la fenêtre de dialogue.
- QRectF boundingRect() : Cette méthode permet de déterminer les limites extérieures de la forme qui permettront d'interagir avec elle.
- void paint(QPainter *painter, const QStyleOptionGraphicsItem *option, QWidget *widget) : Cette méthode permet de dessiner le Sommet sur une ZoneDeDessin de sa couleur "coul" ou de la couleur de sélection. Et avec son étiquette.
- void setSelect(bool B) : Cette méthode permet de modifier l'attribut select du Sommet mais aussi sa couleur (si sélectionnée ou non).
- void ~ QSommet() : Destructeur de l'objet QSommet.

Test de la classe QSommet :
- *void testSommet()*: Teste si un Sommet est créé.

- *void testDestructeur()*: Teste si un Sommet créé est bien détruit après l'appel du destructeur.
- *void testMouseDoubleClickEvent()*: Vérifie que la fenêtre est ouverte.
- *void testBoundingRect()*: Teste si pour un clic simulé, le bon Sommet est sélectionné.
- *void testPaint()*: Vérifier que le Sommet a été dessiné aux positions du curseur.
- *void testSetSelect()*: Va vérifier la modification des attributs "coul" et selected en fonction du booleen passé en paramètre.
- Test des getters.
- Test des setters.

4.4.4 Qarc

```
class Qarc : public QGraphicsItem{

private:
    int id;
    int posxA;
    int posyA;
    int posxB;
    int posyB;

public:
    Qarc (Arc A);
    void mouseDoubleClickEvent(QGraphicsSceneMouseEvent *event);
    QRectF boundingRect();
    void ~QArc();

};
```

Les attributs de la classe:
- *int id* : ID de l'Arc à partir duquel il est construit
- *int posxA* : Représente la position x du Sommet duquel sort l'Arc.
- *int posyA* : Représente la position y du Sommet duquel sort l'Arc.
- *int posxB* : Représente la position x du Sommet dans lequel rentre l'Arc.
- *int posyB* : Représente la position y du Sommet dans lequel rentre l'Arc.

Les méthodes de la classe:
- *Qarc (Arc A)* : Constructeur d'un QArc à partir d'un Arc.
- *void mouseDoubleClickEvent(QGraphicsSceneMouseEvent *event)* : Cette méthode ouvre une fenêtre permettant de modifier l'étiquette ou la charge utile d'un Arc en récupérant la paire renvoyée par la fenêtre de dialogue.
- *QRectF boundingRect()* : Cette méthode permet de déterminer les limites extérieures de la forme qui permettront d'intéragir avec elle.
- *void ∼ QArc()* : Destructeur de l'objet QArc.

Tests de la classe QArc :
- void testArc(): Teste si un Arc est créé.
- void testDestructeur() : Teste si un Arc créé est bien détruit après l'appel du destructeur.
- void testMouseDoubleClickEvent() : Vérifie que la fenêtre est ouverte et que les attributs du Sommet ont bien été modifiés.
- void testBoundingRect() : Teste si pour un clic simulé, le bon Arc est sélectionné.
- void testPaint() : Vérifie que l'Arc a été dessiné entre les deux Sommets sélectionnés.
- Test des getters.
- Test des setters.

4.4.5 ordoCreate

```
class Ui_Dialog {
public:
    QTableWidget *tableWidget;
    QLineEdit *DureelineEdit;
    QLineEdit *TacheAnterieurlineEdit;
    QLineEdit *IDlineEdit;
    QLineEdit *nomDeLaTachelineEdit;
    QPushButton *ajoutTacheButton;
    QPushButton *supprimeTacheButton;
    QDialogButtonBox *buttonBox;
    void setupUi(QDialog *Dialog)
    void retranslateUi(QDialog *Dialog)

};

namespace Ui {
class ordoCreate;
}


class ordoCreate : public QDialog {
Q_OBJECT

public:
    ordoCreate(QWidget *parent = nullptr);
    ~ordoCreate();

private:
    Ui::ordoCreate *ui;
    vector<row_pert> res;

public slots:

    void ajoutTache();
    void supprimerTache();
};
```

Les attributs de la classe:

- *Ui::ordoCreate *ui* : Fenêtre de la boite de dialogue.
- *QTableWidget *tableWidget* : Tableau où sont affichées les informations sur les tâches.
- *QLineEdit *DureelineEdit* : Zone de texte permettant de donner la durée d'une tâche.
- *QLineEdit *TacheAnterieurlineEdit* : Zone de texte permettant de donner les ID des tâches antérieures.
- *QLineEdit *IDlineEdit* : Zone de texte permettant de donner l'ID de la tâche que l'on souhaite ajouter au PERT.
- *QLineEdit *nomDeLaTachelineEdit* : Zone de texte permettant de donner le nom de la tâche que l'on souhaite ajouter au PERT.
- *QPushButton *ajoutTacheButton* : Lorsque l'utilisateur clique sur ce bouton l'ensemble des informations présentes dans les QLineEdit vont rejoindre le tableau mais aussi être converties en pert_row puis envoyées dans la fonction "calc_post" d'Opération sur les Graphes permettant d'afficher en temps réel les tâches postérieures.
- *QPushButton *supprimeTacheButton* : La tâche actuellement sélectionnée par l'utilisateur va être supprimée.
- *QDialogButtonBox *buttonBox* : Ici sont présents deux boutons "OK" et "Annuler". Si "Annuler" est cliqué, la fenêtre se ferme sans rien faire. Si "OK" est cliqué les informations présentes dans le tableaux vont être converties ligne après ligne en pert_row, mises dans un vecteur et passées en paramètre dans la fonction "pert" d'opération sur les Graphes.

Les méthodes de la classe:

- *ordoCreate(QWidget *parent = nullptr) :* Constructeur de la fenêtre permettant l'entrée d'informations servant de base à la création d'un graphe de PERT. Aucun champ n'est pré-rempli.
- *ordoCreate()* : Destructeur de la fenêtre.
- *void setupUi(QDialog *Dialog)* : Utilise les setters des attributs de MainWindow pour leur affecter les valeurs par défaut (déterminer par le biais de Qt Designer). Cette fonction est générée par Qt et ne sera donc pas testée.

- *void retranslateUi(QDialog *Dialog)* : Affecte les textes des éléments, la séparation est rendue nécessaire par les fonctionnalités de traduction de Qt. Cette fonction est générée par Qt et ne sera donc pas testée.
- *void ajoutTache* : Permet d'ajouter les informations présentes dans les QLineEdit dans le tableau.
- *void supprimerTache* : Permet de supprimer les informations sélectionnées du tableau.

Tests de la classe ordoCreate :

- void testOrdoCreate() : Test si le fenêtre est construite.
- void testDestructeur() : Test si l'objet est détruit.
- void testAjoutTache() : Teste si les informations des QlineEdit sont les mêmes que celles enregistrées dans "row_pert" et dans le tableau.
- void testSupprimerTache() : Test si une tâche connue est bien supprimée du tableau des tâches.
- Test des Getters.
- Test des Setters.

4.4.6 modifObjet

```
class Ui_Dialog {
public:

    QLineEdit *lineEditEtiquette;
    QTableWidget *tableWidget;
    QPushButton *supprimeButton;
    QLineEdit *NomlineEdit;
    QLineEdit *ValeurlineEdit;
    QPushButton *addTacheButton;
    QDialogButtonBox *buttonBox;

};

namespace Ui {
class modifObjet;
}

class modifObjet : public QDialog {
Q_OBJECT

public:
    modifObjet(QWidget *parent = nullptr);
    ~modifObjet();

private:
ui::modifObjet *ui;
pair<string, map<string,vectVal>> res;


public slots:
    void supprimeCU();
    void addCU();

};
```

Les attributs de la classe:

- *Ui::ordoCreate *ui* : Fenêtre de la boite de dialogue.
- *QLineEdit *lineEditEtiquette* : Permet de modifier l'étiquette qui s'affiche en placeholder.
- *QTableWidget *tableWidget* : Affiche l'ensemble des charges utiles (ndlr: stockée sous forme de map)
- *QPushButton *supprimeButton* : Permet de supprimer la charge utile sélectionnée par l'utilisateur.
- *QLineEdit *NomlineEdit* : Permet d'entrer le nom d'une future charge utile.
- *QLineEdit *ValeurlineEdit* : Permet d'entrer la valeur d'une nouvelle charge utile.

- *QPushButton *addTacheButton* : Permet d'afficher dans le tableau les valeurs entrées dans les lineEdit.
- *QDialogButtonBox *buttonBox* : Valide ou annule la modification. Si c'est validé, les informations du tableau et de la zone d'édition iront remplacer celle déjà présente dans "res".

Les méthodes de la classe:

- *modifObjet(QWidget *parent = nullptr) :* Constructeur de la fenêtre qui va afficher dans le tableau les informations de l'objet cliqué et son étiquette dans le champ dédié, ces informations sont aussi affectées dans res, res[0] sera égal à l'étiquette et res[1] à la map embarquant la charge utile.
- *modifObjet()* : Destructeur de la fenêtre.
- *void supprimeCU()* : Permet de supprimer la charge utile en surbrillance.
- *void addCU()* : Ajoute une charge utile respectant les informations entrées dans les lineEdit.

Tests de la classe modifObjet :

- void testmodifObjet : Test si l'objet est construit.
- void testDestructeur() : Test si l'objet est détruit.
- *void testSuppripmerCU()*: Vérifier que la ligne du tableau qui est sélectionnée a été correctement supprimée.
- *void testAddCu()*: Vérifie qu'une ligne est correctement ajoutée au tableau avec les informations présentes dans les QLineEdit.

5 Conclusion

Après une réflexion commune sur les différents besoins liés aux fonctionnalités de chaque module nous avons pu définir, à l'aide de l'organigramme, les signatures des classes et des méthodes. L'organigramme nous a aussi permis de structurer les différentes parties des spécifications en fonction des informations qui circulent entre les modules. Cette réflexion fut délicate, mais une fois réalisée celle-ci nous facilitera la tâche pour l'implémentation des différents modules. Pour réaliser ce projet nous aurons besoin de différentes bibliothèques et applications détaillées dans les différents modules tels que RapidJSON dans les gestions de fichiers, Qt pour l'interface graphique, Catch2 et Qtest pour les tests unitaires.
Maintenant que nous avons identifié les besoins et que les signatures des classes et méthodes ont été réalisées, nous allons désormais pouvoir implémenter le projet.

6 Références

1 Bibliothèque de parsing et génération de JSON : RapidJson, http://rapidjson.org/

2 Framework de tests pour les tests unitaires: Catch2, https://github.com/catchorg/Catch2

3 Framework de création d'application: Qt :

- https://doc.qt.io/qt-5/reference-overview.html
- https://stackoverflow.com/questions/7830054/how-to-draw-a-point-on-mouseclick-on-a-qgraphicsscene
- https://www.youtube.com/watch?v=hgDd2QspuDg
- https://doc.qt.io/qt-5/qgraphicsitem.htmlpaint

4 bibliothèque de tests liés à Qt: Qtest :

- https://doc.qt.io/qt-5/qttest-index.html
- https://doc.qt.io/qt-5/qtest-tutorial.html

5 Implémentation de l'algorithme de force: ForceAtlas2 :

- https://github.com/gephi/gephi
- https://medialab.sciencespo.fr/publications/Jacomy_Heymann_Venturini-Force_Atlas2.pdf
- http://www-igm.univ-mlv.fr/ dr/XPOSE2012/visualisation_de_graphes/algorithmes.html

7 Annexe

7.1 Exemple de code JSON :

```
{
    "nom": 0
    "_name", 0
    "list_Arc":
    [
        {
            "id":0,
            "etiquette": "_etiquette",
            "ID_depart": 0,
            "ID_arrivee": 1,
            "Charge_utile": [...]
    },
    {
        "id":1,
        "etiquette": "_etiquette",
        "ID_depart": 1,
        "ID_arrivee": 0,
        "Charge_utile": [...]
    }
],
"list_Sommet":
[
    {
        "posX":0,
        "posY":0,
        "id": 0,
        "etiquette": "_etiquette",
        "liste_ID_Arc":
        [
            0
        ],
        "Charge_utile": [...]
    },
    {
        "posX":0,
        "posY":0,
        "id": 1,
        "etiquette": "_etiquette",
        "liste_ID_Arc":
        [
            1
        ],
        "Charge_utile": [...]
    }
],
"path": "/path/name"
}
```

7.2 Aperçu de l'interface graphique

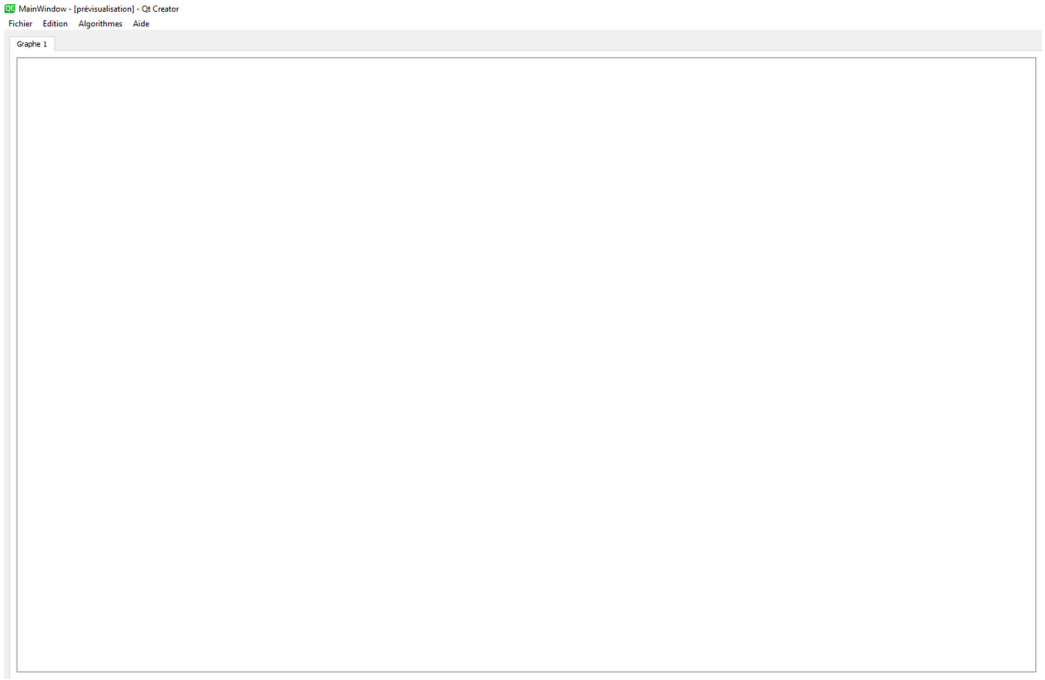


Figure 1: Fenêtre principale

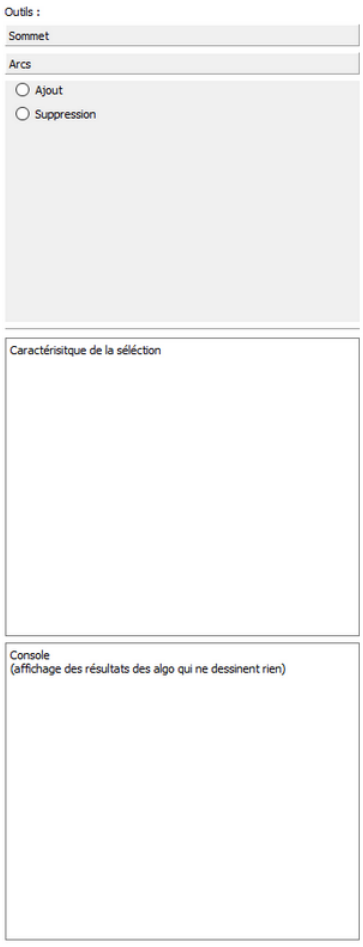


Figure 2: Fenêtre principale (suite)

Figure 3: Fenêtre de modification de sommet

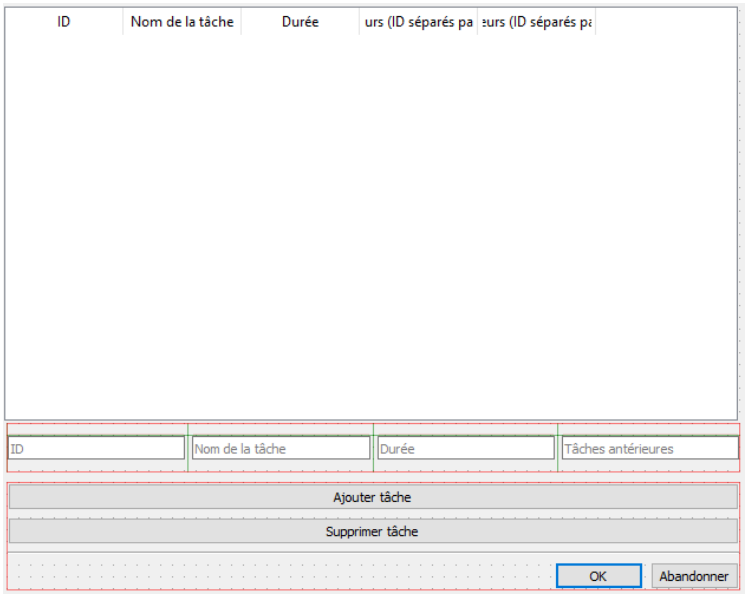


Figure 4: Fenêtre d'ordonnancement)