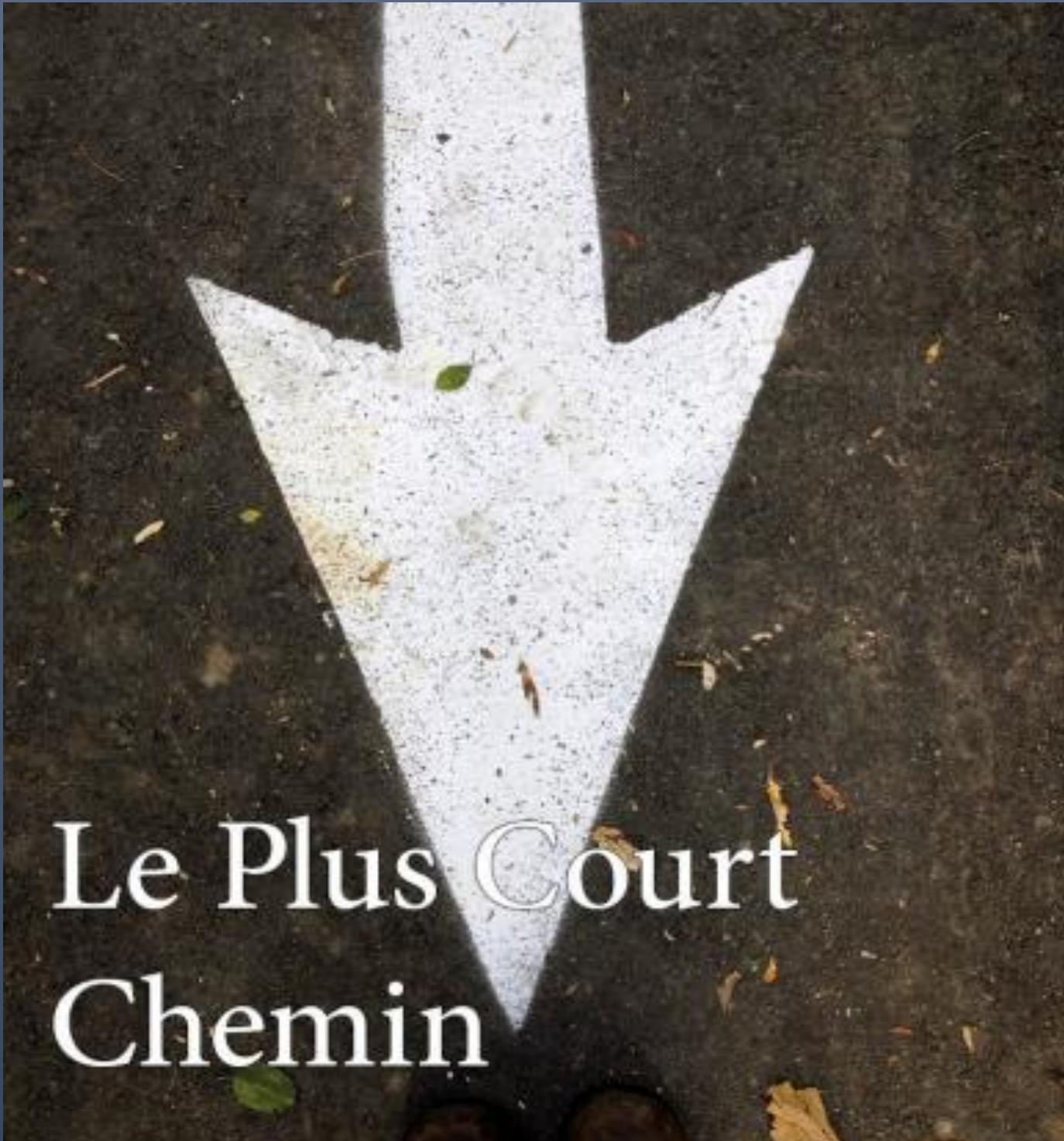


2019-
2020

RAPPORT DU PROJET :



Réalisé par : GROUPE 6
ELAICH Houssam
ELAICH Housni
ELAMRANI ABOUELASSAD Dauha
GHARAFI Hamza
ESSERHIR Mohamed
CHAIB Moncif

Encadre par :
MME ZRIKEM Maria
MR LAHMAIM Miloud

SOMMAIRE

Remerciement	3
Résumé	4
Introduction	5
Chapitre I : Définition du problème	6
Définition du problème	7
Exemples d'application de plus court chemin	7-8
Les algorithmes pour résoudre le PPCC	8-10
Chapitre II : Choix technique	10
Démarche d'apprentissage	11
La notion QT	11
Les classes de Qt	12-13
Chapitre III : Réalisation	14
Les classes utilisés.....	15
Les composantes de l'interface graphiques	16-18
Les méthodes de la classe Mainwindow.....	19
Générer un graphe aléatoirement.....	20
Ajouter un nœud	20
Ajouter une arête	20
Supprimer un nœud	21
Supprimer une arête	21
Les classes et les méthodes utilisés.....	22
Listes Adjacents	23
Classe listenoeuds	23
Classe Adjacents	24
Classe Graphe	24
Fonctions nécessaires pour la réalisation du graphe	25
Les algorithmes utilisés	26
Algorithme de Bellman-Ford	27
Algorithme de Floyd-Warshall	28
Algorithme de Dijkstra avec Tas	29-31
Algorithme de Johnson	32
Résultats et Perspectives	34-37
Réalisation du GPS	38
Choix techniques	41
Save	42
Compétences développées	46
Conclusion Générale.....	47

REMERCIEMENT :

Nous tenons à remercier dans un premier temps, tous les intervenant professionnels responsables de la formation génie informatique.

Avant d'entamer ce rapport, nous profitons de l'occasion pour remercier tout d'abord notre professeur Mme Maria Zrikem qui n'a ménagé aucun effort pour nous aider et nous orienter le long de notre projet, ainsi pour sa générosité en matière de formation et d'encadrement.

Nous remercions également notre professeur Mr Miloud Lahmam pour son encadrement précieux et pour le soutien qu'il nous a donné, et la confiance que Mme Zrikem et Mr Lahmam nous ont témoignée.

Que le corps professoral et administratif de l'ENSA trouve ici nos vifs remerciements.

Nous remercions enfin toute personne qui a contribué de près ou de loin à l'élaboration de ce rapport.

Résumé :

Cet article propose un tour d'horizon sur une interface graphique pour interpréter le problème de plus court chemin, et une application de GPS, créée par des étudiants de la 3eme année génie informatique a l'ENSA de Marrakech.

Dans cet article vous trouverez les méthodes approchées et exactes, de leur performance et de leur complexité théorique, pour différentes versions du problème de plus court chemin. L'étude proposée est faite dans l'optique d'améliorer la résolution d'un problème plus général de couverture dans le cadre d'un schéma de génération de colonnes, dont le plus court chemin apparaît comme le sous-problème.

Introduction

Dans le cadre de notre première année d'étude informatique en cycle ingénieur à l'école nationale des sciences appliquées, nous avons effectué un projet de cent heures réparties sur une période de quatre mois entre février 2020 et juin 2020 encadré par Mme Zrikem Maria et Mr Lahmam Miloud sur la recherche du plus court chemin dans un graphe. La théorie des graphes est une science qui a débuté avec les travaux du célèbre mathématicien Euler quand il a voulu résoudre des problèmes liés à la vie quotidienne comme celui des ponts de Königsberg (les habitants de Königsberg se demandaient s'il était possible, en partant d'un quartier quelconque de la ville, de traverser tous les ponts sans passer deux fois par le même et de revenir à leur point de départ) ou celui du plus court chemin entre deux points.

En général, un graphe permet de représenter la structure, les connexions d'un ensemble complexe en exprimant les relations entre ses éléments. Les graphes constituent donc une méthode de pensée qui permet de modéliser une grande variété de problèmes en se ramenant à l'étude de sommets et d'arcs.

Un graphe G est un couple $G(X, E)$ constitué d'un ensemble X non vide et fini (dont les éléments sont appelés sommets), et d'un ensemble E (dont les éléments sont appelés arêtes) de paires d'éléments de X . Il peut être orienté ou non, selon que l'on munit ou non les arêtes d'un sens de parcours, pondéré ou non, selon que l'on affecte à chaque arête une « valeur » ou pas.

Nous présenterons dans une première partie l'objet de notre étude. Puis, dans une deuxième partie, nous expliquerons notre analyse et les algorithmes que nous avons retenus. Nous terminerons avec une campagne de tests et nous discuterons de la mise en œuvre du projet.

CHAPITRE I:

Définition du
problème.

Le problème de plus court chemin :

Il est courant de chercher le **plus court chemin** en anglais "shortest path", c'est-à-dire celui dont la distance est la plus petite. Si le nombre de trajets possibles entre le point de départ et le point d'arrivée est faible, il suffira de calculer les longueurs de chacun des trajets en additionnant la longueur des liens qui le composent et de comparer directement les longueurs obtenues. Mais une telle solution exhaustive devient rapidement impraticable si le nombre de trajets possibles est grand. Heureusement, il existe des algorithmes qui évitent d'avoir à calculer tous les trajets possibles. Pour cela, ils mettent en œuvre diverses stratégies.

Un problème des plus courants en optimisation combinatoire est celui de la recherche de plus courts chemins dans un graphe. Ce problème se présente comme suit : étant donné un graphe et une fonction cout sur les arcs, le problème consiste à trouver le chemin le moins couteux d'un sommet choisi à un autre. Il se résout aisément grâce à de nombreux algorithmes. Ce rapport de projet présente donc une interface graphique qui comporte différents algorithmes exacts pour résoudre le problème du Plus Court Chemin.

En général, les problèmes de cheminement dans les graphes sont parmi les plus anciens de la théorie des graphes. Ce type de problèmes se rencontre soit directement, soit comme sous problème dans de nombreuses applications.

Quelques exemples d'applications de plus court chemin :

- Trouver le moyen le plus économique pour se déplacer entre les villes
- Les problèmes d'optimisation de réseaux (réseaux routiers ou réseaux de télécommunications).
- Les problèmes d'ordonnancement.
- Les problèmes d'intelligence artificielle tels que la circulation dans un labyrinthe.
- Les problèmes de tournée, de gestion de stock, d'investissement.
- GPS.

Exemples d'utilisation de plus court chemin :

- ❖ Considérons une carte routière et cherchons la route la plus courte pour se rendre d'une localité à une autre. Le graphe est obtenu en prenant comme sommets les localités et en remplaçant chaque route entre deux localités par 2 arcs d'orientations opposées, et ayant pour longueur la longueur de la route.
- ❖ Considérons le projet de construction d'une autoroute entre les villes A et K. Sur le graphe les arcs représentent les différents tronçons possibles de l'autoroute ; chaque arc est valué par le coût total de réalisation du tronçon correspondant.

Les algorithmes pour résoudre le problème de plus court chemin :

Pour résoudre le problème de plus court chemin on devait interpréter des algorithmes qui résolvent ce problème et tenant compte de la moindre complexité et une optimisation maximale.

Il existe deux types d'algorithmes pour résoudre le problème de plus court chemin :

1- Recherche pour tous les couples de sommets.

2- Recherche à partir d'un sommet donné.

Les algorithmes de recherche pour tous les couples de sommets :

Algorithme de Floyd- Warshall :

Il permet de trouver le plus court chemin entre toute paire de sommets.

Les arcs du graphe peuvent avoir des poids négatifs, mais le graphe ne doit pas posséder des circuits absorbants

Robert W. Floyd (né le 8 juin 1936 et mort le 25 septembre 2001 à Stanford (Californie) est un théoricien des graphes et chercheur en informatique américain.



Algorithme de Johnson :

Il résout le problème en absence de circuits négatifs, et peut être plus rapide que Floyd-Warshall dans des graphes creux.

L'algorithme opère en utilisant d'abord l'algorithme de Bellman-Ford pour supprimer les poids négatifs, puis emploi dans un deuxième temps l'algorithme de Dijkstra sur le nouveau graphe.

Les algorithmes de recherche à partir d'un sommet donne :

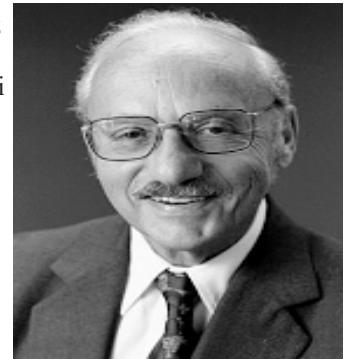
Algorithme de Bellman-Ford :

Cet algorithme calcule les plus courts chemins depuis un sommet source donné dans un graphe orienté pondéré.

Il autorise la présence des arcs de poids négatifs.

Il permet de détecter la présence d'un circuit absorbant

George Bernard Dantzig (8 novembre 1914 à Portland - Oregon - 13 mai 2005 à Palo Alto, en Californie) est un mathématicien américain, notamment inventeur de l'algorithme du simplexe en optimisation linéaire.



Algorithme de Dijkstra : Complexité :

Cet algorithme est utilisable que pour les graphes à poids positifs. Il calcule des plus courts chemins à partir d'une source vers les autres sommets dans un graphe orienté pondéré par des réels positifs

Richard Ernest Bellman, né le 29 août 1920 à Brooklyn et mort le 19 mars 1984 à Los Angeles, est un mathématicien américain. Célèbre pour diverses contributions dans plusieurs domaines des mathématiques, il est surtout l'inventeur de la programmation dynamique.



Edsger Wybe Dijkstra, né à Rotterdam le 11 mai 1930 et mort à Nuenen le 6 août 2002, est un mathématicien et informaticien néerlandais



CHAPITRE II :

**CHOIX
TECHNIQUE**

Démarche d'apprentissage

Le but visionné est la résolution d'une interface graphique qui a pour but résoudre le problème du plus court chemin dans un graphe, par l'application des notions de C++ et la POO bien entendu.

Pour réaliser ce but on a commencé tout d'abord par apprendre les bases de QT. Cela était grâce à plusieurs tutoriels et à l'aide des différents sites, sans oublier notre prérequis des outils de programmation et de design. Après un bon temps on a pu cibler le fonctionnement de notre problématique et nous avons pu acquérir tous les éléments dont nous avions besoin pour avancer dans notre projet.

Le terme Qt :

Qt est une **bibliothèque** multiplateforme pour créer des GUI (programme utilisant des fenêtres). C'est une API orientée objet et développée en C++, et elle est, à la base, conçue pour être utilisée en C++. Toutefois, il est aujourd'hui possible de l'utiliser avec d'autres langages comme Java, Python, Ruby, Visual Basic, etc.

Il intègre directement dans l'interface un débogueur, un outil de création d'interfaces graphiques, des outils pour la publication de code sur Git et Mercurial ainsi que la documentation Qt.

Le tout est tellement énorme qu'on parle d'ailleurs plutôt de Framework : cela signifie qu'il y aura à notre disposition un ensemble d'outils pour développer nos programmes plus efficacement.

Avantages de Qt :

Premièrement le Framework Qt facilite l'internationalisation des applications développées grâce à l'outil Qt Linguist.

Ce Framework Qt permet de développer des applications multi-plates-formes en C++ comme il existe des contraintes permettant de développer dans des langues tels que Java (QtJambi), Python (PyQt) ou encore en Perl (PerlQt). Les applications développées dans ces langages pourront être exécutées sous Windows, Linux et Mac OS.

Il offre la possibilité de développer des jeux grâce au support de l'OpenGL. Le Framework Qt possède un ensemble de bibliothèques permettant de développer des applications qui requièrent le support XML, réseau, manipulation de bases de données.

La Classe QPushButton :

C'est l'une des classes les plus connues que QT fournit .C'est nécessaires pour la création de notre interface graphique .

Le widget QPushButton fournit un bouton de commande. Ce bouton de commande est probablement le widget le plus communément utilisé dans toute interface graphique pour réaliser des actions de la part de l'utilisateur.

Cet élément permet d'aicher un texte ou une image, ainsi qu'optionnellement une petite icône. Le texte peut aicher la touche accélératrice grâce à l'utilisation du caractère "&" devant la lettre symbolisant la touche. Toutes ces propriétés peuvent soit être mises en place grâce au constructeur de la classe, soit par l'intermédiaire d'appel à des méthodes sur l'objet.

La classe QGraphicsView :

La classe QGraphicsView fournit un widget pour l'affichage du contenu d'une QGraphicsScene. QGraphicsView visualise le contenu d'une QGraphicsScene dans une zone d'affichage, avec gestion du défilement.

QGraphicsView fait partie du Framework Graphics View.

La classe QGraphicsScene :

La classe QGraphicsScene fournit une surface pour gérer un grand nombre d'éléments graphiques 2d.

Elle est utilisée en combinaison avec QGraphicsView pour la visualisation graphique d'éléments tels que des lignes, des rectangles, du texte ou même des éléments personnalisés sur une surface 2d. QGraphicsScene fait partie du framework de la vue graphique.

De plus, QGraphicsScene fournit des fonctionnalités qui vous permettent de déterminer efficacement la position des éléments et leur visibilité à l'intérieur d'une zone arbitraire de la scène. Avec le widget QGraphicsView, vous pouvez visualiser la scène entière ou même zoomer à l'intérieur et voir une seule partie de la scène.

La classe QMessageBox :

La classe QMessageBox fournit une boîte de dialogue modale pour informer l'utilisateur ou lui poser une question et recevoir une réponse.

Une QMessageBox (boîte de message) affiche un texte principal pour alerter l'utilisateur d'une situation, un texte informatif pour expliquer d'avantage l'alerte ou pour poser une question à l'utilisateur et un texte détaillé optionnel pour fournir encore plus d'informations si l'utilisateur le demande. Une boîte de message peut également afficher une icône et des boutons standard pour accepter la réponse de l'utilisateur.

Deux API pour utiliser QMessageBox sont fournies, l'API basée sur les propriétés et les fonctions statiques. L'appel d'une des fonctions statiques est l'approche la plus simple, mais elle est moins flexible que d'utiliser l'API basée sur les propriétés et le résultat contient moins d'informations. Il est recommandé d'utiliser l'API basée sur les propriétés.

Les algorithmes utilisés :

Les algorithmes utilisés dans notre projet sont ceux déjà évoqués dans le premier chapitre :

L'algorithme de DIJKSTRA.

L'algorithme de Floyd-Warshall.

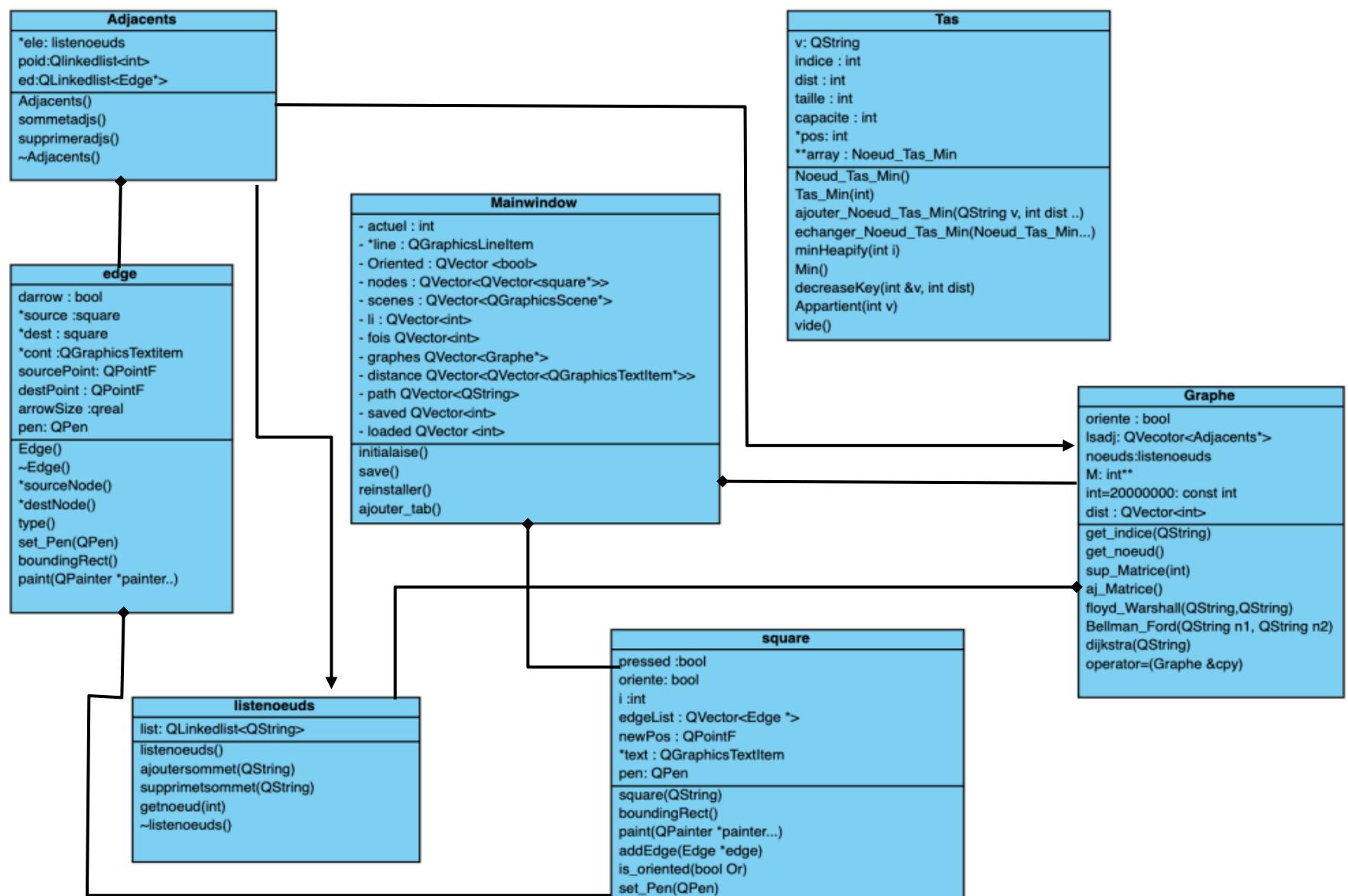
L'algorithme de Bellman-Ford.

L'algorithme de Johsnon.

CHAPITRE III :

REALISATION:

LES CLASSES :



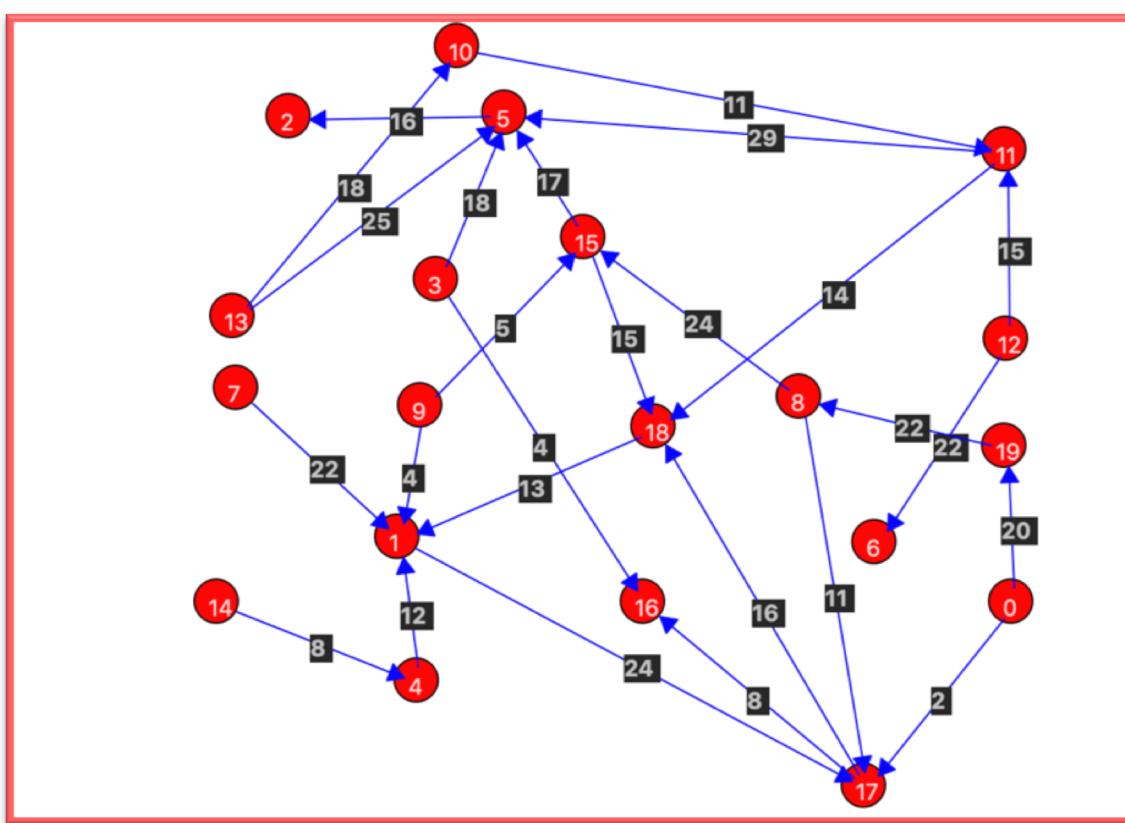
→ Héritage

→ Agrégation



LES COMPOSANTES DE L'INTERFACE GRAPHIQUE

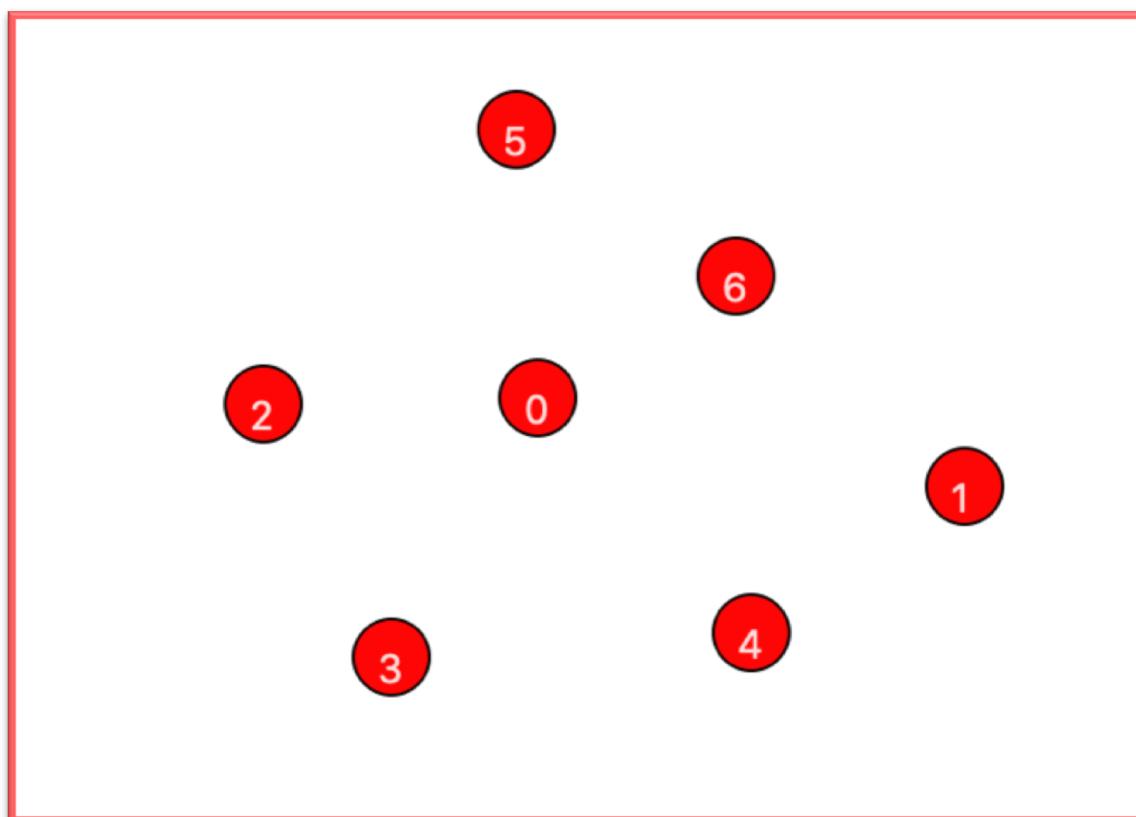
La scène :



Une des composantes de notre interface graphique est la scène, c'est le lieu où s'affiche le graphe qu'on génère et où s'applique les algorithmes pour trouver le plus court chemin dans le graphe.

Dans la scène s'affiche :

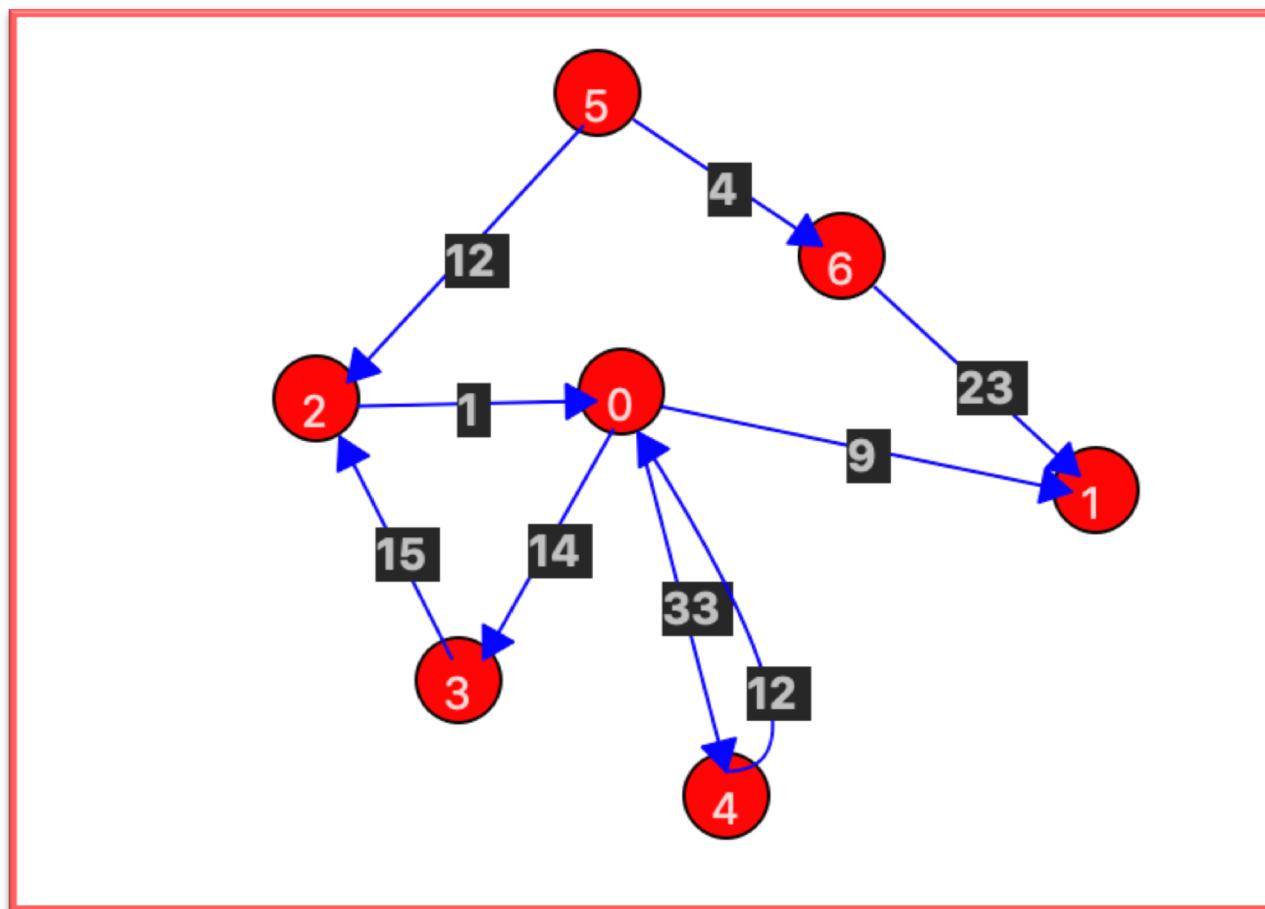
Les nœuds :



Les nœuds sont de forme elliptique, ils sont créés à l'aide de la fonction paint que QT fournit et la classe QPainter.

Ils obtiennent aussi des propriétés comme la possibilité de se déplacer et prennent un texte ou un nombre comme ticket pour les différencier des autres nœuds.

Les arêtes (Arcs) :



Arêtes ou Arcs relient les nœuds entre eux, cela dépend du choix de l'utilisateur.

Les arêtes (arcs) sont créées avec la fonction drawLine ou drawPath que Qt fournit avec la classe QPainter.

Les arêtes (arcs) ont des propriétés comme l'élasticité pour pouvoir rester collés aux noeuds au cas où les nœuds sont déplacés par l'utilisateur.

Les arêtes changent de forme quand ils existent deux arêtes de sens opposé entre deux nœuds.

Les
méthodes de la
classe
MainWindow
(nécessaires à
réaliser notre
programme)

L'interface graphique a besoin des fonctions pour les différentes tâches à faire.

La méthode pour générer un graphe aléatoirement :

Cette méthode permet de générer un graphe aléatoirement, elle prend comme arguments le nombre de sommets qu'on souhaite générer, le nombre d'arêtes, le poids maximal pour les arêtes et une booléenne=true si l'utilisateur souhaite générer un graphe orienté, et false sinon.

La méthode ajouter un nœud dans la scène :

La méthode **ajouter un nœud** sert à ajouter un nœud en se basant sur l'identifiant proposé par l'utilisateur, mais avant de l'ajouter, le programme vérifie d'abord s'il y a déjà un nœud avec le même identifiant dans la liste des nœuds déjà créé, puis il fait la création graphique du nœud et il l'ajoute à la liste des nœuds existants et puis l'affiche dans la scène.

La méthode ajouter une arête :

La méthode permet d'**ajouter une arête** entre deux sommets, le sommet source et le sommet destination que l'utilisateur propose.

L'ajout d'arête maintient le même état après la génération aléatoire du graphe, si l'utilisateur avait choisi que le graphe soit orienté, l'ajout d'une arête vas donc être orienté, (le même cas quand l'utilisateur choisi un graphe sans orientation).

Au cas où une arête est déjà existante entre deux sommets, une alerte s'affiche indiquant l'existence de cette arête.

Au cas où il existe un chemin entre A et B, et l'utilisateur veut ajouter une arête entre B et A, l'arête prend une autre forme pour que l'utilisateur peut distinguer entre les deux arêtes.

La méthode supprimer un nœud :

La méthode sert à **supprimer un nœud** en se basant sur l'identifiant proposé par l'utilisateur, mais avant de le supprimer, le programme vérifie d'abord si ce nœud existe dans la liste des nœuds si oui, il supprime le nœud et les arêtes liés à ce nœud et de la liste d'adjacence, puis il les supprime aussi graphiquement. Au cas où le nœud n'existe pas, il affiche une alerte.

La méthode supprimer une arête :

La méthode sert à **supprimer une arête** en se basant sur les identifiants (source, destination) entre lesquelles se trouve l'arête que l'utilisateur souhaite supprimer, mais avant de le supprimer le programme vérifie d'abord si cette arête existe dans la liste des arêtes, si oui il supprime l'arête de la liste et de la scène.

**POUR SAVOIR PLUS SUR LE FONCTIONNEMENT DES OUTILS
DE L'INTERFACE GRAPHIQUE, Veuillez CONSULTER UNE
DOCUMENTATION MENEE AVEC CE RAPPORT.**

Les classes et les méthodes.

Listes Adjacents :

Notre Graphe est basé sur le principe des listes Adjacents ; listes simplement chaînées, chacune représente chaque sommet de notre Graphe et ses adjacents.

Pour faciliter la construction de notre Graphe, on a travaillé avec "QLinkedList".

Les classes utilisées sont les suivantes :

a/ Classe listenoeuds :

```
#include <QLinkedList>
class listenoeuds
{
public:
    listenoeuds();
    void ajoutersommet(QString);
    void supprimetsommet(QString);
    QString getnoeud(int);
    ~listenoeuds(){};

public :
    QLinkedList<QString> list;
};


```

Un objet de cette classe nous permet de construire notre liste des sommets. Le contenu de ces sommets peut être n'importe quel caractère (chiffres , Alphabet ou les deux).

Les fonctions membres nécessaires pour la réalisation de cette liste :

+Constructeur : Initialisation de notre liste.

+ajoutersommet : Ajoute au liste la valeur entrée dans les paramètres. À condition que la valeur n'existe pas dans la liste.

+supprimetsommet: Supprime de la liste la valeur entrée dans les paramètres. À condition qu'il existe déjà.

NB : On va voir ensuite l'utilité de la fonction "getnoeud " .

b/ Classe Adjacents :

```
#include "listenoeuds.h"
#include "edge.h"
#include <QLinkedList>

class Adjacents :public listenoeuds
{
public:
    Adjacents();
    void sommetadj(QString,QString,int);
    void supprimeradj(QString);
    void changeradj(QString,int);
    ~Adjacents(){};

public:
    listenoeuds *ele;
    QLinkedList<int> poid;
    QLinkedList<Edge*> ed;
};


```

Héritage de la classe "listenoeuds" . Un objet de cette classe nous permet de construire en général la liste des adjacents d'un sommet. Plus précisément trois données membres. Premièrement, liste des sommets adjacents d'un sommet. Deuxièmement, une liste des poids des arêtes(arcs) qui relient ces sommets. Dernièrement, une liste des éléments qui pointent vers des objets d'une classe nommée "Edge".Cette dernière nous aide à afficher les arêtes(arcs) dans la scène de notre MainWindow .

Les fonctions membres nécessaire pour la réalisation de cette liste :

+Constructeur.

+ Sommetadj.

+ supprimeadjs.

+Constructeur : Initialisation des trois listes.

+sommetadj : D'après l'objet de type "listenoeuds" saisi dans les paramètres de cette fonction ,on relie (crée l'arête(arcs)) la source (sommet saisi premièrement) et la destination (sommet saisi deuxièmement), à condition que ces dernières déjà existent dans l'objet et le sommet destination n'existent pas dans la liste adjacent de la source , avec un poids donné .

+ supprimeadjs : Supprime de ces deux premières listes :

1) Le sommet, de la valeur entrée dans les paramètres, de la liste des sommets adjacents.

2) Le poids d'arête (arc) qui a existé entre la source et ce sommet.

À condition que ce sommet existe déjà.

NB : la manipulation de la liste des Edges (Ajouter,Supprimer) on va la voir ultérieurement dans la Classe MainWindow .

c/ Classe Graphe :

```
#ifndef GRAPHE_H
#define GRAPHE_H
#include <adjacents.h>
#include <QVector>
#include <QMMessageBox>

class Graphe:public Adjacents
{
public:

    Graphe(bool);
    void ajouter_noeud(QString);
    void ajouter_arete(QString,QString,int);
    void supprimer_noeud(QString);
    void supprimer_arete(QString,QString);
    void changer_arete(QString,QString,int);
    int Get_Poid(int , int);
    listenoeuds get_noeud(){
        return noeuds;
    }
    int get_indice(QString);
    void sup_Matrice(int);
    void aj_Matrice();

    QLinkedList<QString> floyd_Warshall(QString,QString);
    QLinkedList<QString>* Bellman_Ford(QString);
    QLinkedList<QString> dijkstra(QString);
    QLinkedList<QString>** Johnson();

    ~Graphe(){};

public:
    bool oriente;
    QVector<Adjacents*> lsadj;
    listenoeuds noeuds;
    int** M;
    const int inf=20000000;
    QVector<int> dist; //Dijkstra
    int *Distance; //Bellman
    int **Distance_Johnson;
};

#endif // GRAPHE_H
```

Héritée de la classe "Adjacents" .Un objet de cette classe nous permet de créer notre Graphe.

Cette classe contient comme données membres :

+orienté : Booléen précis si notre Graphe sera orienté ou non.

+lsadj : À l'aide de la classe "QVector", on a pu manipuler facilement un vecteur de la même taille que notre liste des nœuds , ces éléments sont des objets de type "Adjacents". Donc "lsadj" présentent les listes d'adjacences de chaque sommet de notre Graphe.

+noeuds : Objet de la classe "listenoeuds" ; ce sont les sommets de notre Graphe.

+M : Matrice d'adjacence.

+dist : Vecteur, on va voir son utilité ensuite.

NB:

La création de notre Graphe est basée juste sur les listes d'adjacences non pas la matrice d'adjacence. Suivant le principe général de l'algorithme de Floyd-Warshall, on était obligés de travailler avec la matrice d'adjacence. Ainsi, pour la manipuler on a défini les fonctions suivantes :

->sup_Matrice : Prend en paramètre l'indice du nœud qu'on veut supprimer, et il supprime de la matrice d'adjacence la ligne et la colonne correspondant à cet indice .

->aj_Matrice : Ajoute une ligne et une colonne à la matrice d'adjacence.

Les fonctions membres nécessaire pour la réalisation de ce Graphe :

+ajouter_noeud() :

```
void Graphe::ajouter_noeud(QString n)
{
    //Ajouter "n" premièrement à notre liste des noeuds
    noeuds.ajoutersommet(n);
    int i=noeuds.list.size();
    //redimensionner notre vecteur des listes des adjacents
    lsadj.resize(i);
    //Initialiser le nouvel élément de notre vecteur et ajouter "n" à notre vecteur
    lsadj[i-1]=new Adjacents();
    lsadj[i-1]->ele->list.append(n);
    //Ajouter une ligne et une colonne à notre Matrice d'adjacence
    aj_Matrice();
}
```

+ajouter_arete() :

```
void Graphe::ajouter_arete(QString n1, QString n2,int pd)
{
    //les variables "src" et "dst" sont respectivement les indices des noeuds "n1" et "n2"
    int i=0,src,dst;
    //it est l'itérateur de la liste des noeuds
    QLinkedList<QString>::iterator it;
    it=noeuds.list.begin();
    while(it!=noeuds.list.end()){
        if(*(lsadj[i]->ele->list.begin())==n1){
            src=i;
            //Ajouter "n2" à la liste d'adjacence de "n1"
            lsadj[i]->sommetsadj(noeuds,n1,n2,pd);
        }
        if(*(lsadj[i]->ele->list.begin())==n2){
            dst=i;
            //Si notre Graphe est non orientée alors la relation se passe dans les deux sens
            // d'où l'ajoute de "n1" aussi à la liste d'adjacence de "n2"
            if(!orientee) lsadj[i]->sommetsadj(noeuds,n2,n1,pd);
        }
        i++;
        it++;
    }
    //On suite le même principe de la matrice d'adjacence
    M[src][dst]=pd;
    if(!orientee) M[dst][src]=pd;
}
```

+supprimer_noeud() :

```
void Graphe::supprimer_noeud(QString n)
{
    int i=noeuds.list.size();
    int cpy=0,indice=0;
    //La fonction "get_indice" retourne l'indice du noeud "n" dans notre liste des noeuds.
    indice=get_indice(n);
    //Créer temporairement un vecteur pour pouvoir copier notre vecteur "lsadj" sans l'existence de "n".
    QVector<Adjacents*> tmp;
    tmp=new QVector<Adjacents*>(i-1);
    for(int d=0,j=0;d

```

+supprimer_arete():

```
void Graphe::supprimer_arete(QString n1, QString n2)
{
    //les variables "src" et "dst" sont respectivement les indices des noeuds "n1" et "n2".
    int src,dst;
    for(int i=0;i<noeuds.list.size();i++){
        //Si le tête de l'élément de notre vecteur "lsadj" égal "n1" .
        if(*(lsadj[i]->ele->list.begin())==n1){
            src=i;
            //Si l'élément contient "n2",on supprime "n2" de la liste d'adjacence de "n1".
            lsadj[i]->supprimeradjs(n2);
        }
        //Si le tête de l'élément de notre vecteur "lsadj" égal "n2" .
        if(*(lsadj[i]->ele->list.begin())==n2){
            dst=i;
            //Si notre Graphe est non orientée ,on supprime aussi "n1" de la liste d'adjacence de "n2".
            if(!orientee) lsadj[i]->supprimeradjs(n1);
        }
    }
    //Suivant le même principe de la matrice d'adjacence .
    M[src][dst]=inf;
    if(!orientee) M[dst][src]=inf;
}
```

LES ALGORITHMES UTILISES :

-Tous les algorithmes ont été définie pour plus de fluidité, comme fonctions membre de la classe "graphe". En plus, l'application de ces algorithmes se fait sous la forme des boutons (QPushButtons) .

-Le plus court chemin entre deux sommets quelconque apparaître sur la scène sous la forme d'une coloration des arêtes(arcs) et des sommets qui construisent ce chemin.

Algorithme de Bellman-Ford :

La fonction nommée par « Bellman-Ford » retourne un tableau de listes (Type : QLinkedList<QString>) contenant tous les plus courts chemins entre un sommet source et tous les autres sommets du graphe s'ils existent.

-La définition de notre fonction est la suivante :

```

QLinkedList<QString>* Graphe::Bellman_Ford(QString n1){
    QLinkedList<QString> *chemin;
    QString *Previous;
    chemin = new QLinkedList<QString> [noeuds.list.size() + 1];
    //Si le sommet de l'indice n1 n'a aucun voisin:
    if(lsadj[get_indice(n1)]->ele->list.empty()){
        chemin[0].prepend(n1);
        return chemin;
    }
    Distance = new int [noeuds.list.size()];
    Previous = new QString [noeuds.list.size()];
    //initialiser les distances par l'infini et les prédecesseurs par "NULL":
    for(int i = 0; i < noeuds.list.size(); i++){
        Distance[i] = inf;
        Previous[i] = "NULL";
    }
    //le sommet source d'indice n1 doit être initialisé par 0 :
    Distance[get_indice(n1)] = 0;
    int tempDistance;
    for(int k = 0; k < noeuds.list.size() - 1; k++){
        //Pour chaque arc (j,i) comparer entre la distance[i] et la somme de distance[j] et le poids de (j,i):
        for (int i = 0; i < noeuds.list.size(); i++){
            for(int j = 0; j < noeuds.list.size(); j++){
                if(lsadj[j]->ele->list.contains(noeuds.getnoeud(i))){
                    tempDistance = Distance[j] + Get_Poid(j,i);
                    if(tempDistance < Distance[i]){
                        Distance[i] = tempDistance;
                        //le sommet d'indice j est le prédecesseur de i
                        Previous[i] = noeuds.getnoeud(j);
                    }
                }
            }
        }
    }

    // vérifier s'il y a un cycle de poids négatif , si oui retourner la liste vide:
    for (int i = 0; i < noeuds.list.size(); i++){
        for(int j = 0; j < noeuds.list.size(); j++){
            if(lsadj[j]->ele->list.contains(noeuds.getnoeud(i))){
                tempDistance = Distance[j] + Get_Poid(j,i);
                if(tempDistance < Distance[i]){
                    QMessageBox msgBox;
                    msgBox.setIcon(QMessageBox::Warning);
                    msgBox.setText("ATTENTION !!!Vous avez un cycle négatif dans ce graph.");
                    msgBox.exec();
                    return chemin;
                }
            }
        }
    }
    //Remplissage de la liste :
    QString Pre;
    for (int i = 0; i < noeuds.list.size(); i++){
        Pre = noeuds.getnoeud(i);
        chemin[i].prepend(Pre);
        if(i == get_indice(n1)){
            }else{
                do{
                    if(Distance[get_indice(Pre)] < inf){
                        Pre = Previous[get_indice(Pre)];
                        chemin[i].prepend(Pre);
                    }else Pre = n1;
                }while(Pre != n1 );
            }
        }
    return chemin;
}

```

La propriété la plus novatrice de cet algorithme est l'utilisation de la programmation dynamique :

Comme d'autres problèmes de programmation dynamique, l'algorithme calcule les plus courts chemins de manière ascendante. Il calcule d'abord les distances les plus courtes pour les plus courts chemins qui ont au plus une arête dans le chemin. Ensuite, il calcule les chemins les plus courts avec au moins 2 arêtes, et ainsi de suite. Après la ième itération de la boucle externe, les plus courts chemins avec au plus i arêtes sont calculés. Il peut y avoir un maximum (nombre de sommets - 1) arêtes dans n'importe quel chemin simple, c'est pourquoi la boucle externe s'exécute (nombre de sommets - 1) fois. L'idée est, en supposant qu'il n'y a pas de cycle de poids négatif (pour cela on a utilisé un tableau local "Previous" où on a stocker le chemin de chaque sommet selon Bellman-Ford avant de les insérer à la liste de retour.), si nous avons calculé les chemins les plus courts avec au plus i arêtes, alors une itération sur tous les arêtes garantit de donner le chemin le plus court avec au plus (i + 1) arêtes.

Algorithme de Floyd-Warshall :

La fonction nommée par « `Floyd_Warshall` » retourne une liste contenant tous les sommets qui construisent le plus court chemin entre deux sommets source et destination donnés.

La définition de notre fonction est la suivante :

```

QLinkedList<QString> Graphe::floyd_Warshall(QString n1, QString n2){
//Déclaration d'une Matrice temporaire
//Matrice qui contient dans chaque cellule ij, le plus court chemin entre les sommets i et j (partant de i).
    int** tmp;
//Déclaration de la Matrice π
//Matrice qui contient dans chaque cellule ij, le prédecesseur de j dans le plus court chemin entre i et j (partant de i).
    QString** pi;
    bool circuit_absorbant=false;
//initialisation de tmp.
    tmp=new int * [noeuds.list.size()];
    for(int k=0;k<noeuds.list.size();k++){
        tmp[k]=new int[noeuds.list.size()];
    }
    for(int i=0;i<noeuds.list.size();i++){
        for(int j=0;j<noeuds.list.size();j++){
            tmp[i][j]=M[i][j];
        }
    }
//initialisation de π.
    pi=new QString * [noeuds.list.size()];
    for(int k=0;k<noeuds.list.size();k++){
        pi[k]=new QString[noeuds.list.size()];
    }
    for(int i=0;i<noeuds.list.size();i++){
        for(int j=0;j<noeuds.list.size();j++){
            if(M[i][j]==inf || i==j) pi[i][j]="null";
            else pi[i][j]=lsadj[i]->ele->list.front();
        }
    }
//Une fois ces matrices initialisées, on fera les calculs suivants ,avec
// une répétition "k" égale la dimension de notre liste des noeuds .
    for (int k=0;k<noeuds.list.size();k++){
        for (int i=0;i<noeuds.list.size();i++){
            for (int j=0;j<noeuds.list.size();j++){
                //Suivant le principe d'algorithme du floyd-Warshall
                if ((tmp[i][k] + tmp[k][j])< tmp[i][j]){
                    tmp[i][j] = tmp[i][k] + tmp[k][j];
                    pi[i][j]=pi[k][j];
                }
            }
        }
    }
}

//détection d'un circuit absorbant.
for (int i=0;i<noeuds.list.size();i++){
    if(tmp[i][i]<0) circuit_absorbant=true;
}

//Création de la liste des sommets qui construit le plus court chemin
// entre la source " n1 " et la destination " n2 ".

QLinkedList<QString> chemin;
//S'il n'y a aucun circuit absorbant
if(!circuit_absorbant){
//S'il existe un prédecesseur de "n2" dans le plus court chemin entre "n1" et "n2" .
    if(pi[get_indice(n1)][get_indice(n2)]!="null"){
        int l=0;
        QString k;
        for(int i=0;i<noeuds.list.size();i++){
            if(i==get_indice(n1)){
                l=get_indice(n2);
                k=pi[get_indice(n1)][l];
                for(int j=0;j<noeuds.list.size();j++){
                    if(j==l && k!=n1){
                        //Ajouter au début de liste.
                        chemin.prepend(k);
                        l=get_indice(k);
                        k=pi[get_indice(n1)][l];
                        while(j>0) j--;
                    }
                }
            }
        }
        chemin.prepend(n1);
        //Ajouter à la fin de la liste.
        chemin.append(n2);
    }
    else chemin.append("-1");
}

return chemin;
}

```

Algorithme de Dijkstra avec Tas :

Classe Tas:

```

class Tas_Min{
public:
    class Noeud_Tas_Min{
        public :
            QString v;
            int indice;
            int dist;
            Noeud_Tas_Min(){}
        };
    // 
    int taille;
    int capacite;
    int *pos;
    Noeud_Tas_Min **array;
    //
    Tas_Min(int);
    //
    Noeud_Tas_Min* ajouter_Noeud_Tas_Min(QString v,int dist,int indice);
    //
    void echanger_Noeud_Tas_Min(Noeud_Tas_Min** a,Noeud_Tas_Min** b);
    //
    void minHeapify(int i);
    //
    Noeud_Tas_Min* Min();
    //
    void decreaseKey(int &v, int dist);
    //
    bool Appartient(int v);
    //
    int vide();
};

};

```

D'après cette classe on va créer un tas-min. Pour les données membres, elle contient :

- **Classe "Noeud_Tas_Min"** : Qui représente un nœud dans le tas, ce nœud est caractérisé par une valeur (v) ; cette valeur est la même valeur du sommet dans le graphe, son indice dans le tas et une valeur de distance (dist) par laquelle on va tasser ce tas.

- **taille** : Nombre du nœud couramment présenté dans le tas.
- **capacite** : Capacité du tas.
- **pos** : Position des nœuds (Noeud_Tas_Min) dans le tas.
- **array** : Tableau des nœuds (Noeud_Tas_Min).

Pour les fonctions membres, on a :

- **Le constructeur** : Initialise le tas par donner sa capacité, ici on donne au "pos" et "array" une taille égale à la capacité et "taille" égale à 0.

- **ajouter_Noeud_Tas_Min** : Par des valeurs données, elle crée et retourne un nouveau "Noeud_Tas_Min", pour qu'on puisse l'insérer dans "array" après.

- **echanger_Noeud_Tas_Min** : échange deux nœuds du tas, on a besoin de cette fonction pour le tri .

- **minHeapify** : Selon un indice donné, cette fonction compare la valeur "dist" du nœud, où son indice égale à l'indice donné, par ses fils s'ils existent, après si la distance du nœud est supérieure à celle des fils , elle échange la position du nœud avec le fils du plus petit distance.

- **Min** : Extraire du tas le nœud avec la plus petite distance.

- **decreaseKey** : Elle change la distance "dist" d'un nœud donné v. Après, dans une boucle, elle compare la distance de ce nœud avec la distance de son parent et si la distance du nœud est inférieure à celle du parent, elle échange sa position avec le parent, et ainsi de suite.

- **Appartient** : pour tester si un sommet d'indice 'v' appartient au tas-min ou non.

- **vide** : pour tester si le tas est vide ou non.

Voici les étapes que nous avons suivis :

-----NB :

La fonction Dijkstra retourne, pour une source donnée, une liste des prédecesseurs pour les sommets, qu'on va travailler avec après dans le [MainWindow](#) pour l'affichage.

1) Créer et initialiser le tas-min avec le sommet source comme racine (la valeur de distance affectée au sommet source est 0) . La valeur de distance attribuée à tous les autres sommets est INF (infinie). Le tableau "dist" indique les valeurs de distance de chaque sommet par rapport à la source (ce tableau on va l'utiliser après dans le [MainWindow](#) pour qu'on puisse afficher les valeurs au-dessus de chaque sommet).

-> La définition de notre fonction est la suivante :

```
QLinkedList<QString> Graphe::dijkstra(QString src)
{
    int V = this->noeuds.list.size(); // Le nombre des sommets dans le graphe.
    dist.clear();
    dist.resize(V);

    QString predecesseur[V]; //table des prédecesseurs.
    QLinkedList<QString> list_nouv; //liste des prédecesseurs.

    //Création du tas.
    Tas_Min tas_min(V);

    QLinkedList<QString>::iterator it=this->noeuds.list.begin();
    int v=0;
    // Insérer au tas min les sommets avec des distances égale à l'infinie
    // et positions identique à l'ordre d'insertion de chaque sommets.
    // Initialiser la valeur de prédecesseurs par -1.
    it=this->noeuds.list.begin();
    while(it!=this->noeuds.list.end()){
        predecesseur[v]="-1";
        this->dist[v]= INT_MAX;
        tas_min.array[v] = tas_min.ajouter_Noeud_Tas_Min(*it,dist[v],v);
        tas_min.pos[v] = v;
        v++;
        it++;
    }

    // Changer la valeur de distance du source à 0 et mettre à jour le tas.
    it=this->noeuds.list.begin();
    v = 0;
    while(it!=this->noeuds.list.end()){
        if((*it)==src){
            tas_min.array[v] = tas_min.ajouter_Noeud_Tas_Min(src, dist[v],v);
            tas_min.pos[v] = v;
            this->dist[v] = 0;
            tas_min.decreaseKey(v, this->dist[v]);
        }
        v++;
        it++;
    }

    // Initialiser la taille du tas-min à V
    tas_min.taille= V;
}
```

2/ Bien que le tas-min ne soit pas vide, on suit les procédures suivantes :

a) Extraire le sommet avec le nœud de valeur de distance minimale de tas-min, soit le sommet extrait **u** .

b) Pour chaque sommet adjacent **v** de **u**, vérifiez si **v** est dans le tas-min . Si **v** est dans le tas-min et que la valeur de distance est supérieure au poids de **u-v** plus la valeur de distance de **u**, donc mettre à jour la valeur de distance de **v**.

```
// Dans la boucle suivante, le tas-min contient tous les noeuds
// dont la distance la plus courte n'est pas encore finalisée.
int i = 0;
while (!tas_min.vide()){
    // Extraire le sommet avec la valeur de distance minimale.
    Tas_Min::Noeud_Tas_Min* noeud_Tas_Min = tas_min.Min();
    // Stocker l'indice du sommet extrait.
    int u = noeud_Tas_Min->indice;
    // Traversez tous les sommets adjacents de u (le sommet extrait) avec leurs poids
    // et mettez à jour leurs valeurs de distance.
    QLinkedList<QString>::iterator it_1=this->lsadj[u]->ele->list.begin();
    QLinkedList<int>::iterator it_poid=this->lsadj[u]->poid.begin();
    it_1++;
    while (it_poid!= this->lsadj[u]->poid.end()){
        for(int y=0;y<V;y++){
            if(tas_min.array[y]>v==*it_1){
                i = tas_min.array[y]->indice;
                // Si la distance la plus courte jusqu'à *it_1 n'est pas encore finalisée et si la distance jusqu'à *it_1
                // à travers u est inférieure à sa distance calculée précédemment.
                if (tas_min.Appartient(i) && this->dist[u] != INT_MAX && ((*it_poid) + this->dist[u]) < this->dist[i]){
                    // Mettre à jour la valeur du prédecesseur du u.
                    predecesseur[i]=noeud_Tas_Min->v;
                    this->dist[i] = (this->dist[u] + (*it_poid));
                    // Mettre à jour la valeur de distance en tas min.
                    tas_min.decreaseKey(i,this->dist[i]);
                }
            }
            break;
        }
        it_1++;
        it_poid++;
    }
}
```

->Finalement on aura une liste des prédecesseurs :

```
// Remplire notre liste par les prédecesseurs de chaque noeud.
for(int y=0;y<V;y++) {
    list_nouv.push_back(predecesseur[y]);
}
return list_nouv;
```

Algorithme de Johnson :

La fonction nommée par «**Johnson**» retourne une matrice de listes (Type : `QLinkedList<QString>**`) contenant les plus courts chemins entre chaque paire du graphe, et aussi remplit la matrice des distances entre chaque paire de sommets dans ce graphe, et colore le plus court chemin entre deux sommets choisis par l'utilisateur (Les pires peuvent être choisis plusieurs fois).

La définition de notre fonction est la suivante :

```

QLinkedList<QString> **Graphe::Johnson()
{
    if(orienté){
        QString text = "Johnson";
        QLinkedList<QString>::iterator it;
        QLinkedList<QString>::iterator it1;
        QLinkedList<Edge*>::iterator it2;
        QLinkedList<QString>** chemin;
        QLinkedList<QString> * chemin_Bellman;
        chemin = new QLinkedList<QString> *[noeuds.list.size()];
        it = noeuds.list.begin();
        while(it!=noeuds.list.end()){
            int i = get_indice(*it);
            chemin[i] = new QLinkedList<QString> [noeuds.list.size()];
            it++;
        }
        int w;
        Distance_Johnson = new int*[noeuds.list.size()];
        it = noeuds.list.begin();
        while(it!=noeuds.list.end()){
            int i = get_indice(*it);
            Distance_Johnson[i] = new int [noeuds.list.size()];
            it++;
        }
        // Création du nouveau sommet nécessaire à l'application de l'algorithme de Johnson:
        ajouter_noeud(text);
        // Liée ce nouveau sommet à tous les autres sommets du graphe par des arêtes de poids 0;
        it = noeuds.list.begin();
        while(it!=noeuds.list.end()){
            if (*it != text){
                ajouter_arete(text , *it , 0);
            }
            it++;
        }
        // Appliquer l'algorithme de Bellman_Ford au nouveau sommet :
        chemin_Bellman = Bellman_Ford(text);
        // Repondération des arêtes du graphe :
        it = noeuds.list.begin();
        while(it!=noeuds.list.end()+1){
            int i = get_indice(*it);
            it1 = lsadj[i]->ele->list.begin();
            if (*it != text){
                while(it1 != lsadj[i]->ele->list.end()){
                    int j = get_indice(*it1);
                    w = Get_Poid(i,j);
                    changer_arete(*it , *it1 , w + Distance[i] - Distance[j]);
                    it1++;
                }
                it++;
            }
        }
        //Suppression du nouveau sommet et les arêtes liées à ce sommet :
        it = noeuds.list.begin();
        while(it!= noeuds.list.end()){
            if(*it != text){
                supprimer_arete(text , *it );
            }
            it++;
        }

        supprimer_noeud(text);

        //Vérifier si le chemin de Bellman est vide c-à-d s'il y a un cycle de poids négative :
        if(!chemin_Bellman->empty()){
            //Appliquer l'algorithme de Dijkstra à chaque sommets du graphe en utilisant les nouveaux poids:
            // + Remplissage de la matrice de distance entre chaque paire de sommets dans ce graphe :
            it = noeuds.list.begin();
            while(it!=noeuds.list.end()){
                if(*it != text ){
                    int i = get_indice(*it);
                    *chemin[i] = dijkstra(*it);
                    it1 = noeuds.list.begin();
                    while(it1 != noeuds.list.end()){
                        int j = get_indice(*it1);
                        Distance_Johnson[i][j] = dist[j] + Distance[j] - Distance[i];
                        qInfo() << i << "-" << j << " : " << Distance_Johnson[i][j];
                        it1++;
                    }
                }
                it++;
            }
        }
    }
}

```

```

// Repondération des arcs du graphe au poids initial :
    it = noeuds.list.begin();
    while(it!=noeuds.list.end()+1){
        int i = get_indice(*it);
        it1 = lsadj[i]->ele->list.begin();
        if (*it != text){
            while(it1 != lsadj[i]->ele->list.end()){
                int j = get_indice(*it1);
                w = Get_Poid(i , j);
                changer_arete(*it , *it1 , w + Distance[j] - Distance[i]);
                it1++;
            }
        }
        it++;
    }
//Retourner le chemin est la matrice des distance est une variable globale qu'o peut appeler dans "MainWindow" :
    return chemin;
    //Retourner NULL si le gaphe contient un cycle de poids négatif :
}else return NULL;
}

```

La repondération est peut-être la partie la plus **novatrice** de cet algorithme. Le concept de base est le suivant : si toutes les arêtes d'un graphe sont non négatives, l'exécution de l'algorithme de Dijkstra sur chaque sommet est le moyen le plus rapide de résoudre le problème du plus court chemin toutes paires. Si, cependant, certaines arêtes ont des poids négatifs, nous pouvons les repondérer afin que la définition suivante soit vérifiée.

- **La repondération est un processus par lequel le poids du bord est modifié pour satisfaire deux propriétés.**

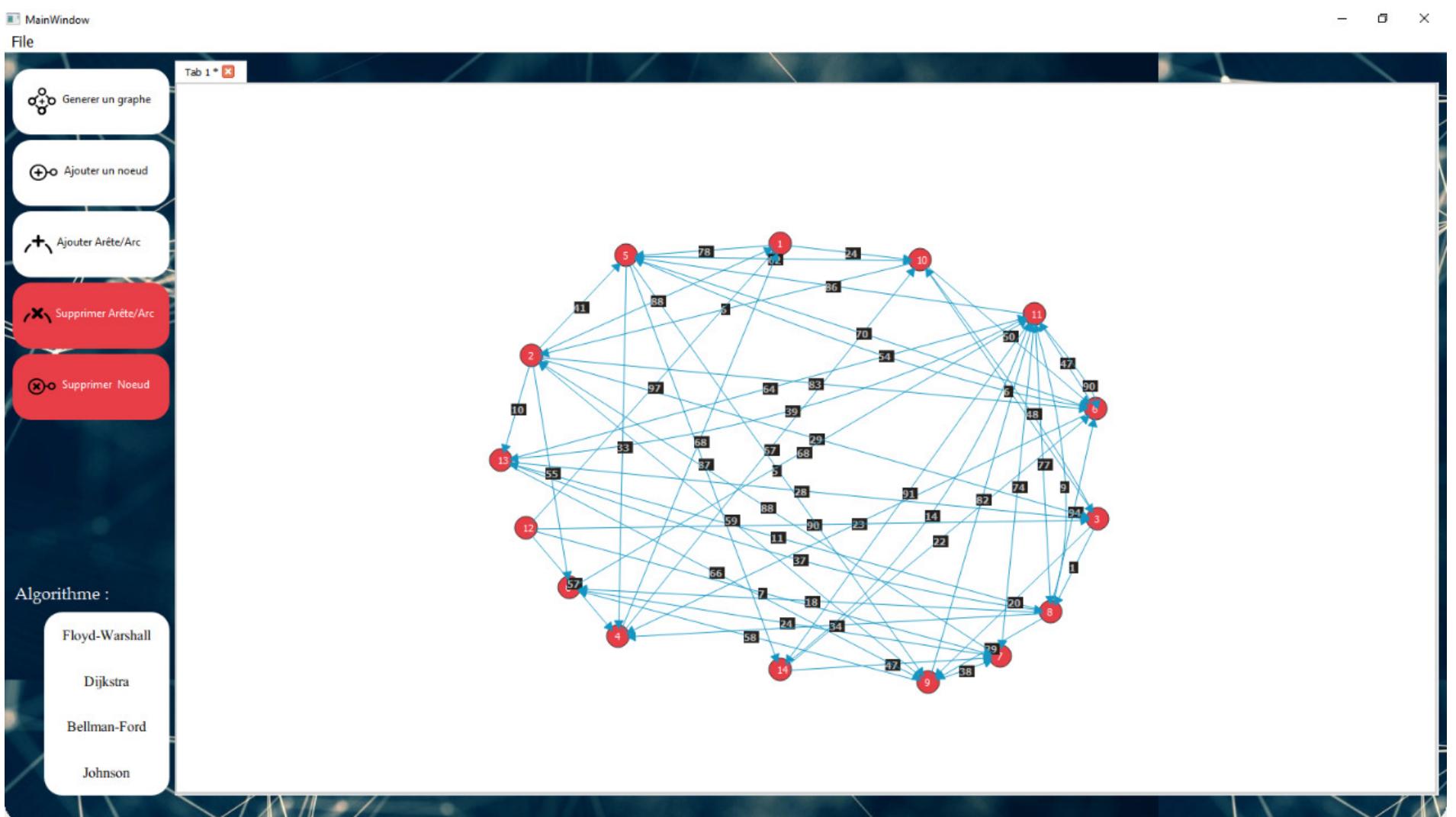
1 - Pour toutes les paires de sommets u, v dans le graphe, si un certain chemin est le chemin le plus court entre ces sommets avant la repondération, il doit également être le chemin le plus court entre ces sommets après la repondération.

2 - Pour toutes les arêtes, (u, v) , dans le graphe, le poids (u, v) doit être non négatif.

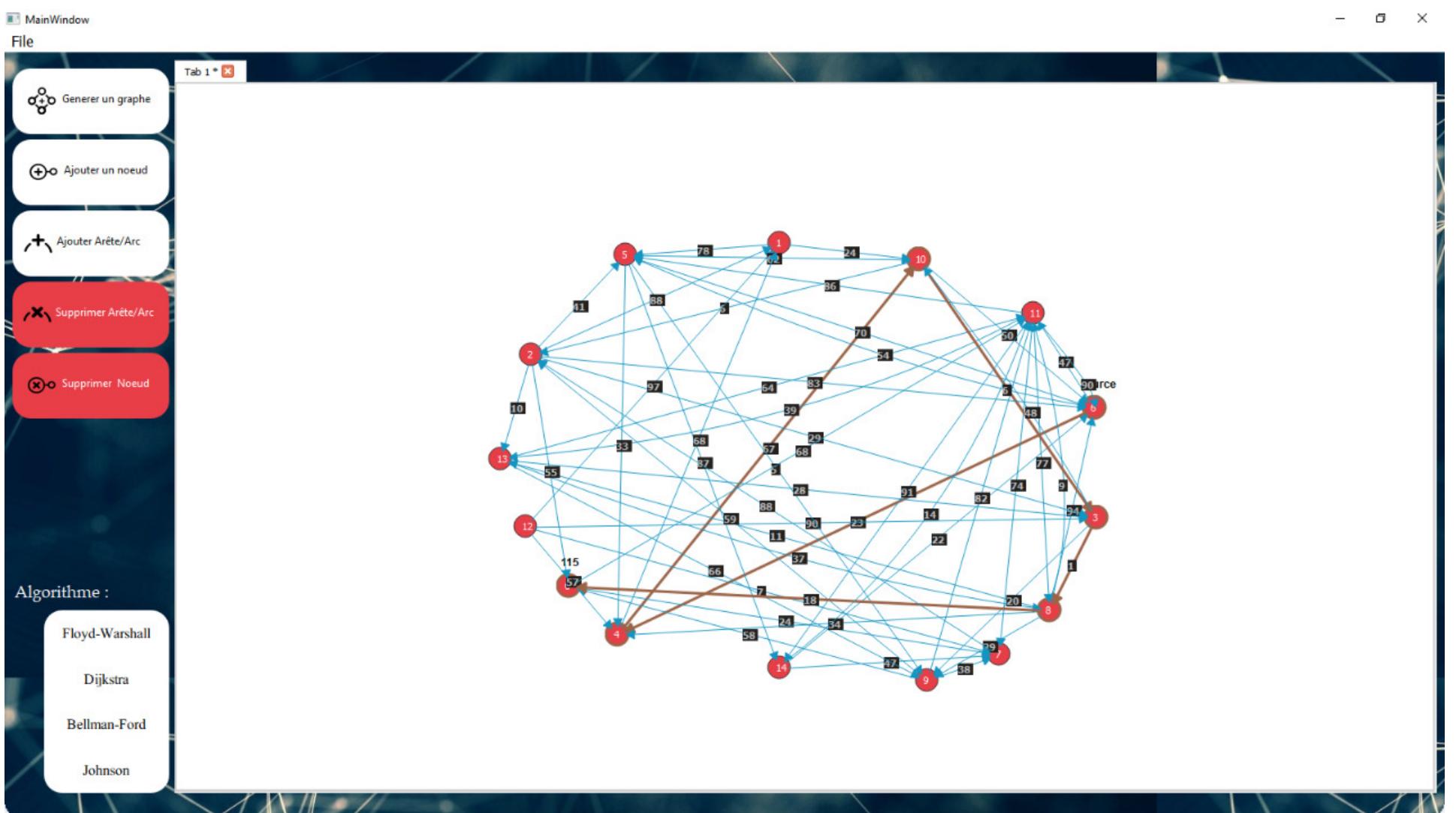
Enfin, l'algorithme de Dijkstra est exécuté sur tous les sommets pour trouver le chemin le plus court. Cela est possible car les poids ont été transformés en poids non négatifs. Il est cependant important de retransformer ces poids de chemin en poids de chemin d'origine afin qu'un poids de chemin précis soit renvoyé à la fin de l'algorithme. Cela se fait à la fin de l'algorithme en inversant simplement le processus de repondération. En règle générale, une structure de données comme une matrice est renvoyée. À chaque cellule i, j de cette matrice est le chemin le plus court du sommet i au sommet j .

Résultats et perspectives

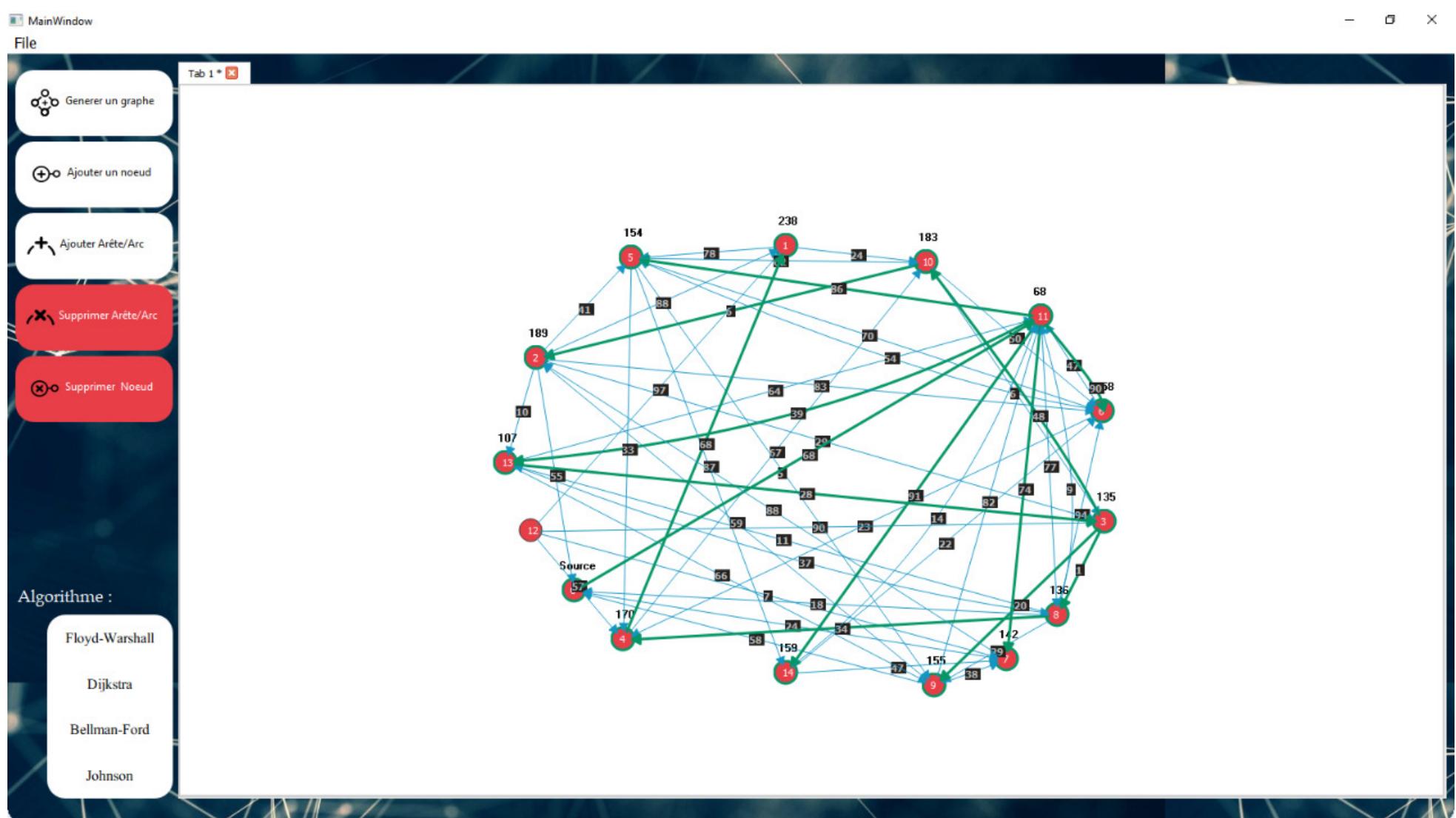
Générer un graphe :



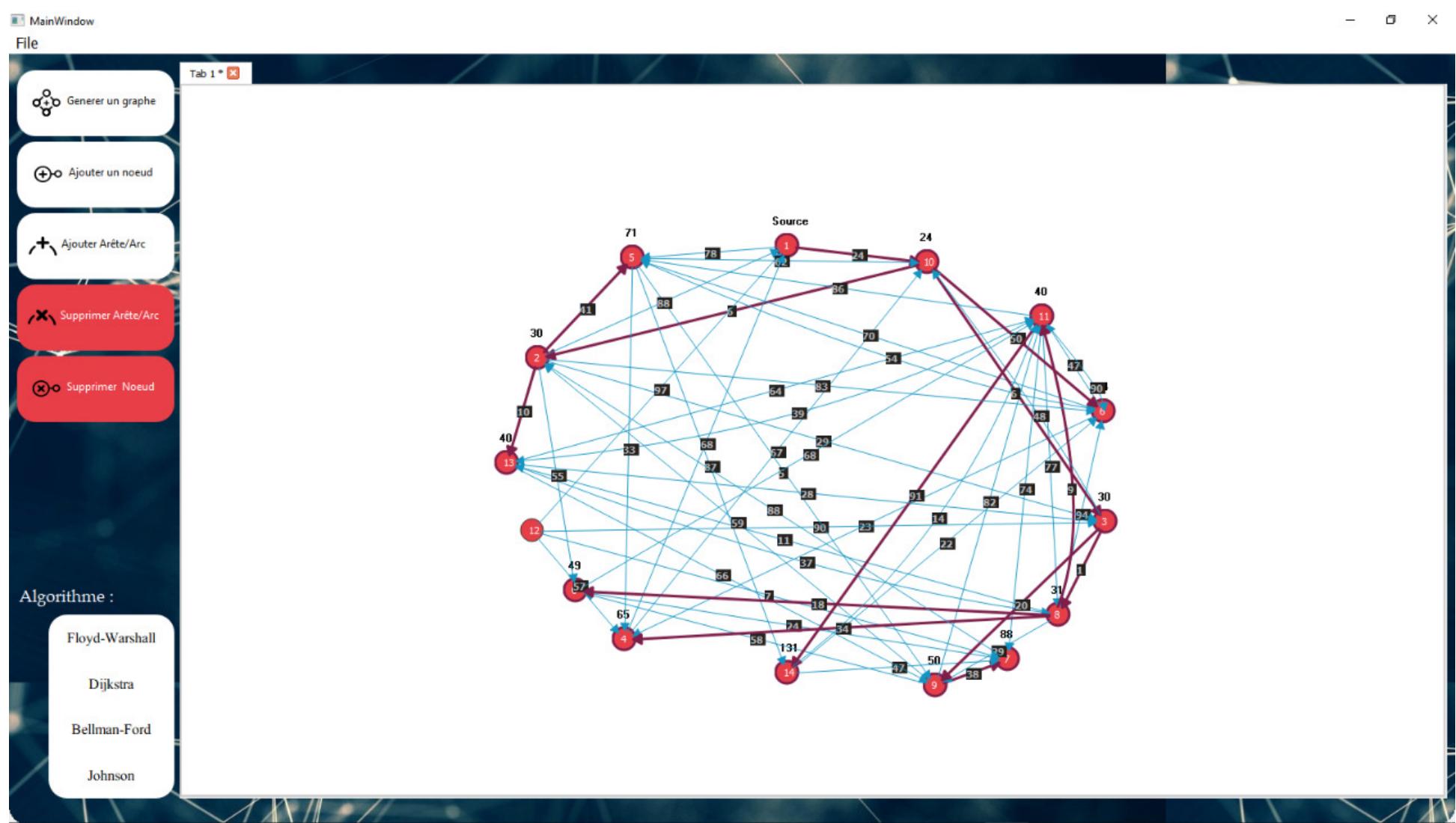
Appliquer l'algorithme de Floyd-Warshall



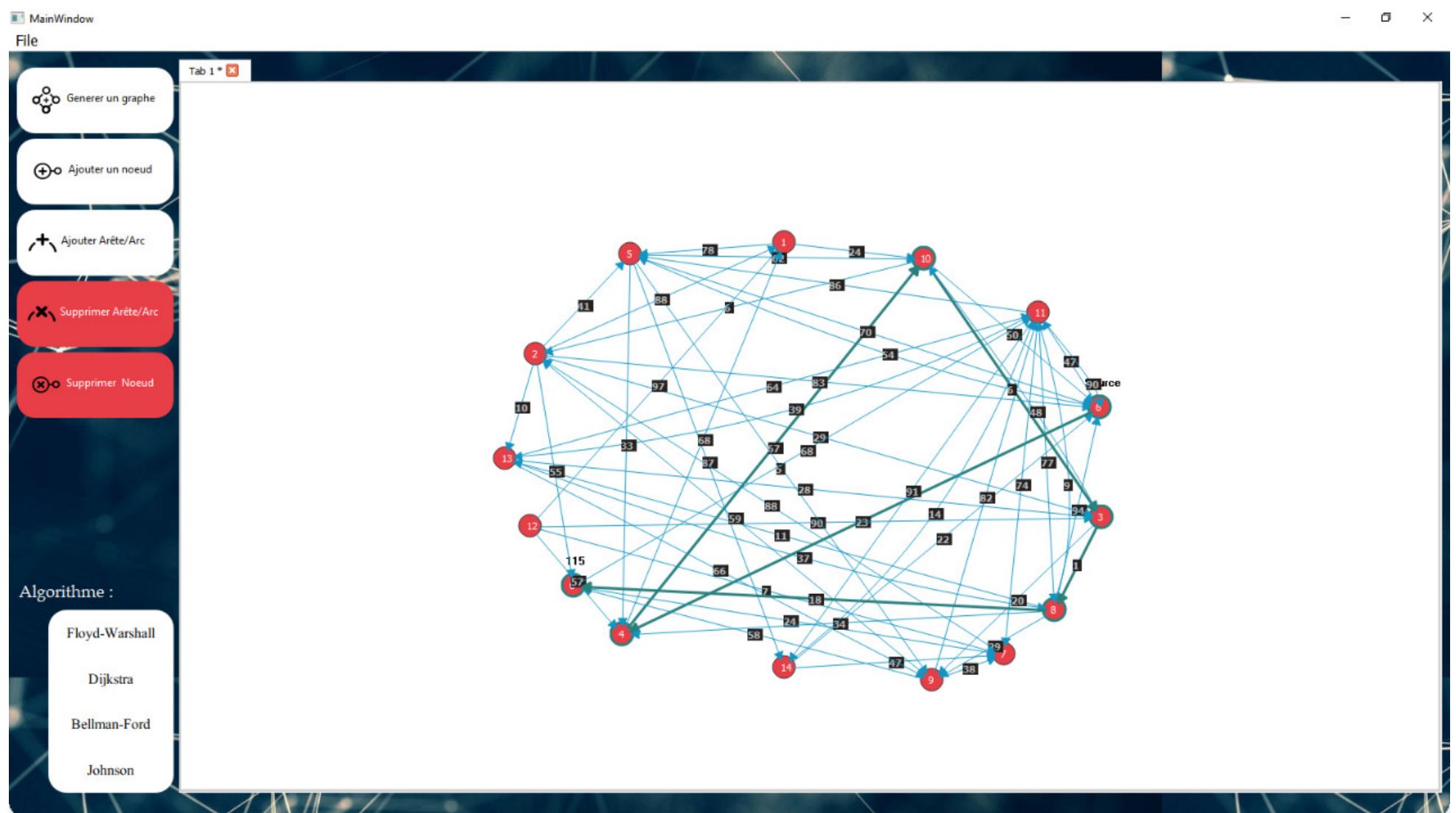
Appliquer l'algorithme de Dijkstra :



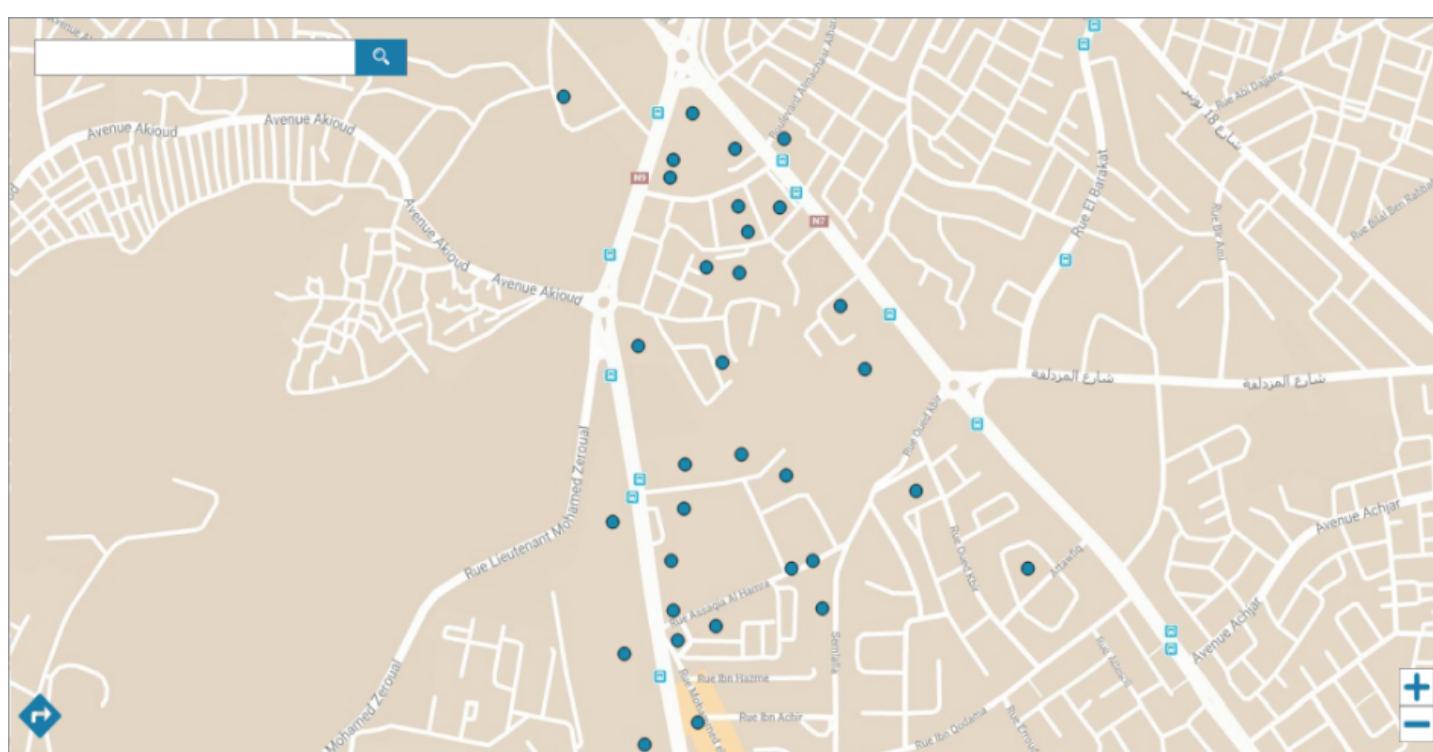
Appliquer l'algorithme de Bellman-Ford :



Appliquer l'algorithme de Johnson :



Réalisation du GPS

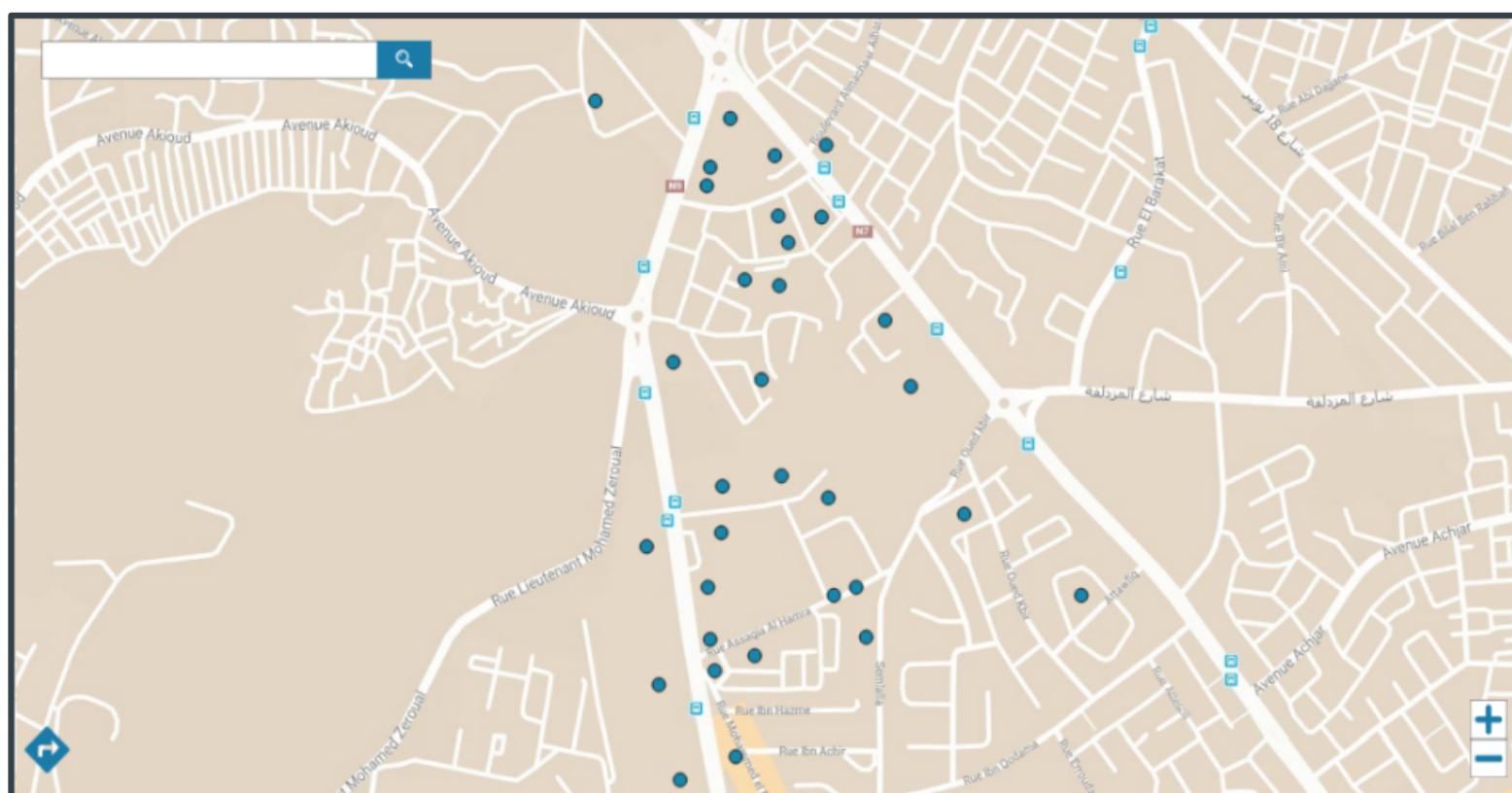


Puisque notre projet est l'interprétation des algorithmes de plus court chemin, le GPS est une des applications qui fonctionnent avec ce concept.

On a donc eu l'idée d'interpréter ce problème en créant un GPS.

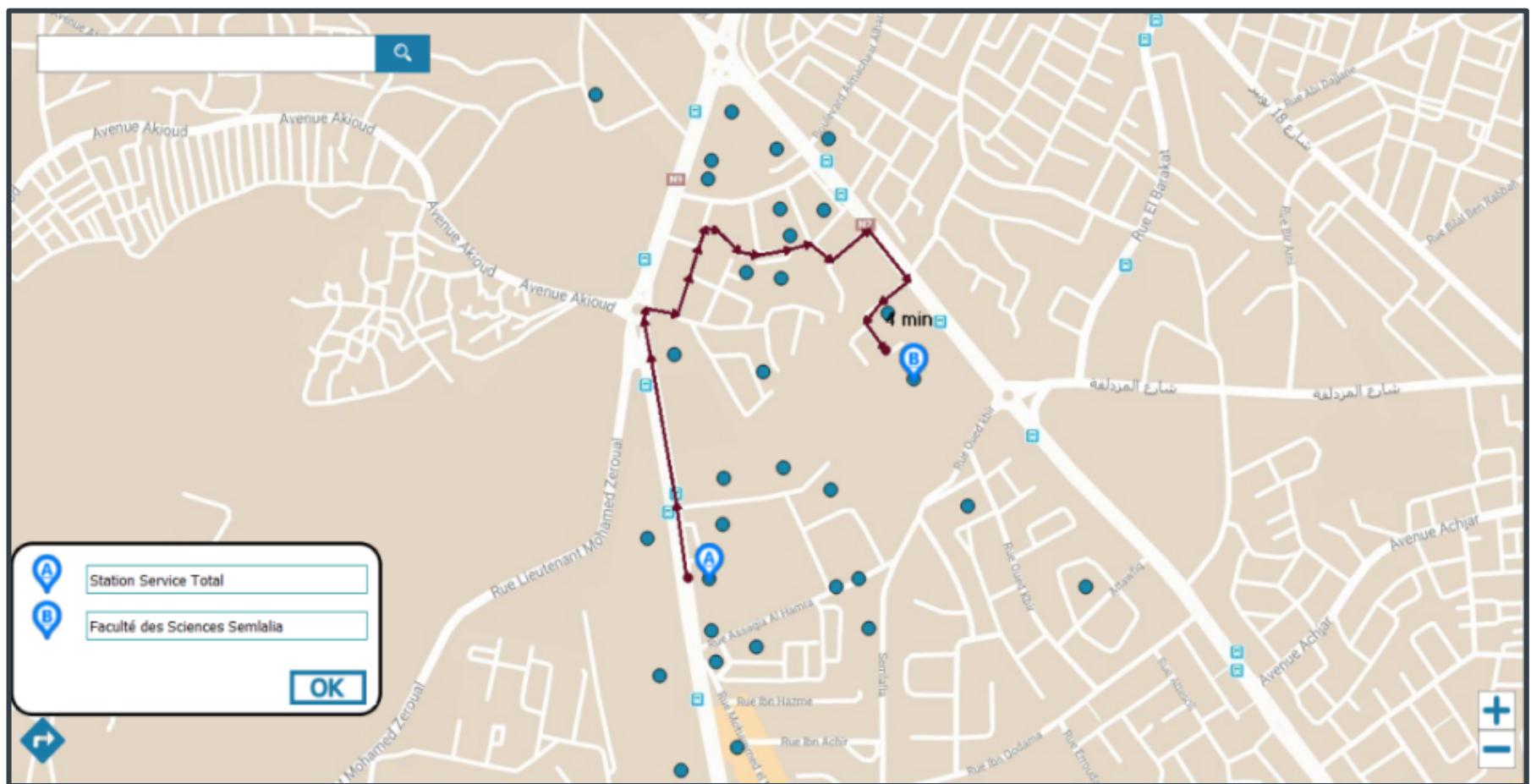
Le **Global Positioning System (GPS)** (en français : « Système mondial de positionnement » [littéralement] ou « Géo-positionnement par satellite »), originellement connu sous le nom de **Navstar GPS**, est un système de positionnement par satellites appartenant au gouvernement des États-Unis. Mis en place par le département de la Défense des États-Unis à des fins militaires à partir de 1973, le système avec vingt-quatre satellites est totalement opérationnel en 1995 et s'ouvre au civil en 2000.

Réalisation du GPS :

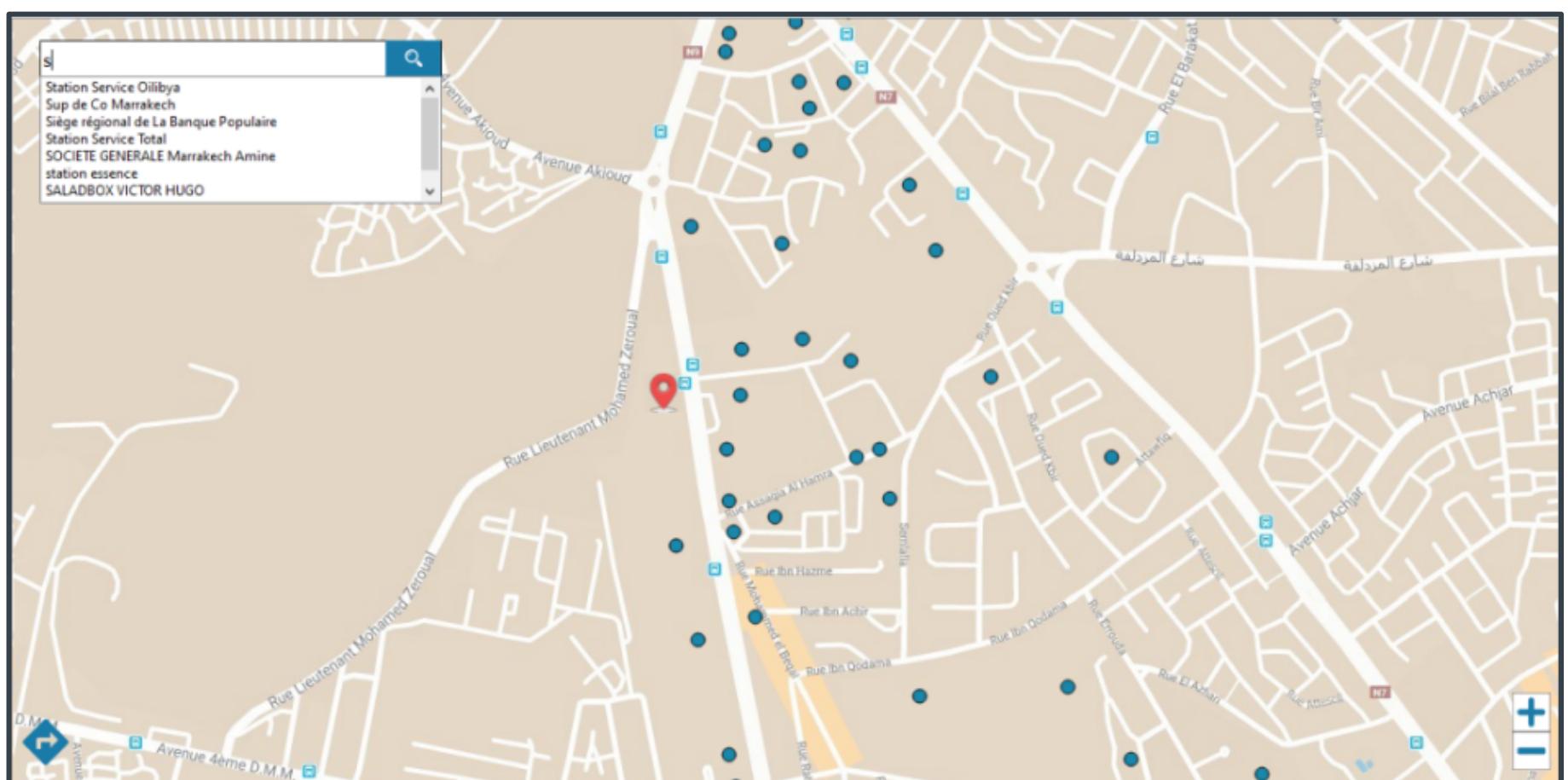


À l'aide du programme principale et quelque modification, nous avons pu créer notre GPS, tel que :

- Nous avons utilisé un fichier texte pour sauvegarder les données du GPS.
- Pour les routes, nous avons utilisé des sommets différents (chiffrés) aux celles utilisés pour les lieux, et entre chaque deux sommets nous avons compté sur des mesures réelles (en mètre).
- Ce qui concerne le plus court chemin entre deux lieux, on a appliqué **l'algorithme Floyd-Warshall**, tel que le résultat s'affichera sous forme d'une coloration du chemin et au-dessus de destination la durée nécessaire pour arriver à cette destination (Pour la calculer , nous avons pris en considération les feux rouges et que nous utilisons une voiture au vitesse de 30km/h).



- Pour la barre de recherche, vous pouvez avoir des suggestions lors du saisi et le résultat de votre recherche s'affiche sous forme d'une icône dans la carte.



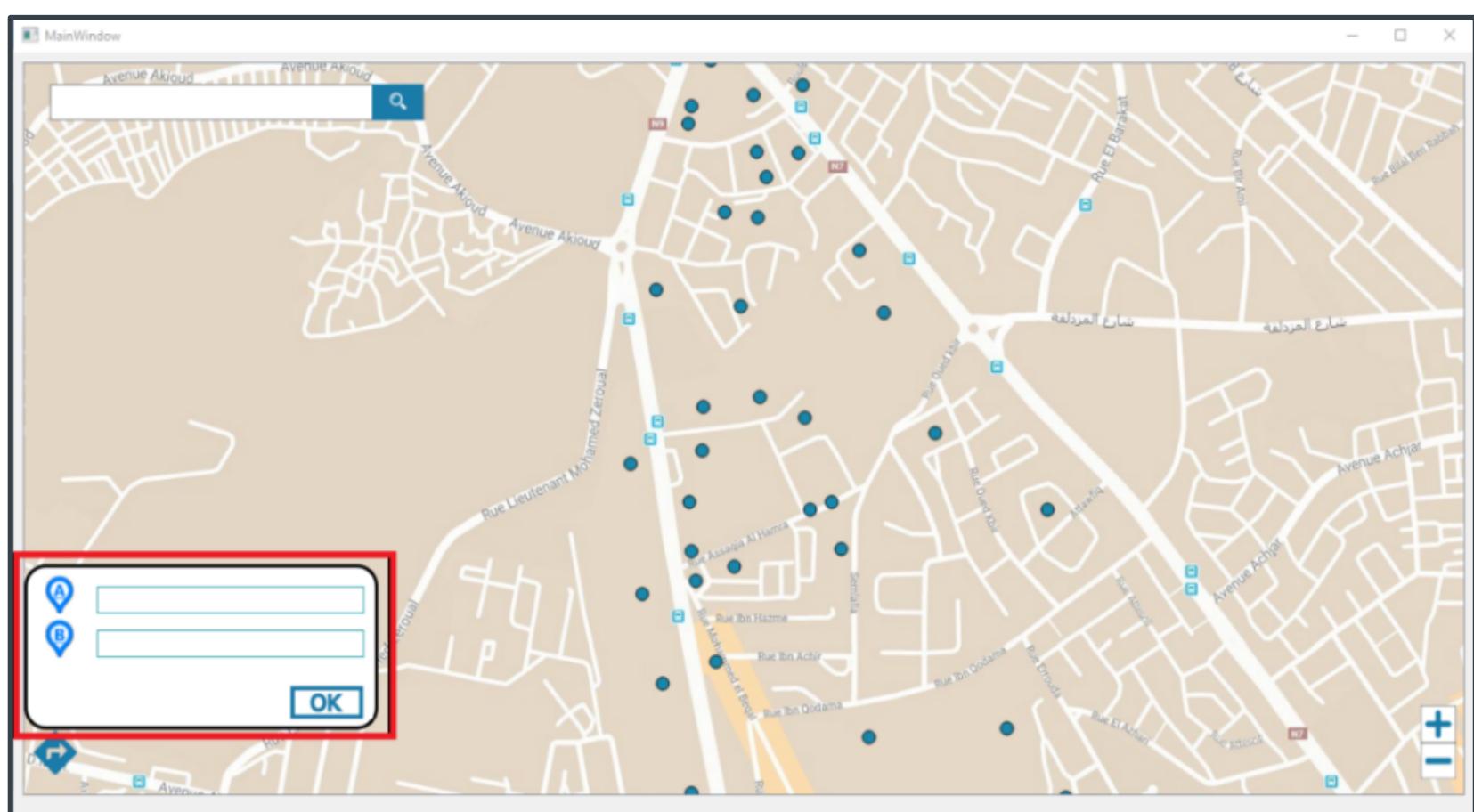
Choix technique :

Pour les classes de QT utilisées, nous avons ajouté :

-**Classe QToolButton** : Cette classe est un peu similaire à la classe **QPushButton**, les deux héritent de la classe **QAbstractButton** et la plupart de leurs fonctions sont les mêmes , mais dans ce cas c'est mieux de travailler avec **QToolButton** puisqu'il est facile de manipuler un objet de cette classe et on peut l'intégrer dans **QGraphicsView** et **QLineEdit** . On l'avait utilisé pour créer le bouton de recherche, les deux boutons de zoom et le bouton de direction.

-**Classe QLineEdit** : Cette classe hérite de la classe **QWidget**. C'est une zone de texte d'une seule ligne, elle est utilisée pour la barre de recherche et la saisie de la source et la destination.

-**Classe QGraphicsPixmapItem** : Cette classe est utilisée dans notre GPS pour afficher les icônes dans la carte.



On le fait apparaître et disparaître par cliquer sur le bouton :

Le principe de
notre
sauvegarde

Principe utilisé dans notre sauvegarde :

La sauvegarde d'un graphe se fait sous forme d'un fichier texte contient comme exemple :

```
Orienter = 1
0 247 24
1*
7..
1 274 332
2*
6..
3*
12..
A*
8..
2 114 23
3*
16..
3 -79 245
2*
18..
A -72 126
```

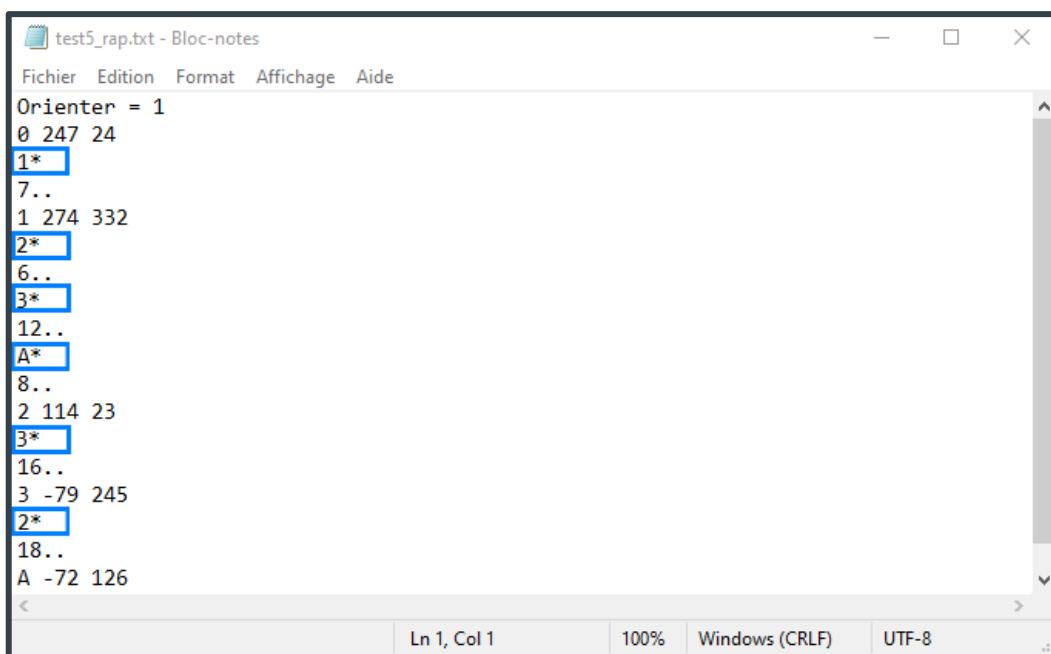
1) Comme vous pouvez voir au-dessus, le fichier commence par un indicateur de type du graphe enregistrer (1 = orienter / 0 = non orienter).

2) l'enregistrement des sommets se fait sous la forme suivante :

```
Orienter = 1
0 247 24
1*
7..
1 274 332
2*
6..
3*
12..
A*
8..
2 114 23
3*
16..
3 -79 245
2*
18..
A -72 126
```

Telle que le première chiffre indique la valeur du sommet et les deux autres chiffres indiquent la position du sommet dans la scène (x,y).

3) En ce qui concerne les adjacents de chaque sommet on les ont identifié par une étoile (*), tel que pour un sommet v , chaque chiffres définis par une étoile entre la ligne où v est définie et la ligne où le prochain sommet est défini, ce sont les adjacents du sommet v . Par exemple pour le sommet 1, il a trois adjacents se sont 2 , 3 et A :



```
test5_rap.txt - Bloc-notes
Fichier Edition Format Affichage Aide
Orienter = 1
0 247 24
1*
7..
1 274 332
2*
6..
3*
12..
A*
8..
2 114 23
3*
16..
3 -79 245
2*
18..
A -72 126
Ln 1, Col 1 100% Windows (CRLF) UTF-8
```

4/Finalement pour les poids nous avons utilisé deux points (..) pour les identifier et chaque adjacent est suivi par son poids.



```
test5_rap.txt - Bloc-notes
Fichier Edition Format Affichage Aide
Orienter = 1
0 247 24
1*
7..
1 274 332
2*
6..
3*
12..
A*
8..
2 114 23
3*
16..
3 -79 245
2*
18..
A -72 126
Ln 1, Col 1 100% Windows (CRLF) UTF-8
```

Compétences développées

Compétences développées

Ce projet nous a permis d'acquérir et de développer plusieurs compétences que ce soit sur le plan organisationnel ou bien sur le plan technique.

De point de vue gestion de projet, nous avons appris comment organiser et planifier un projet en groupe. Nous avons également appris comment gérer un projet en parallèle des cours à l'université et des activités de chacun des membres du groupe.

Ce projet nous a aussi aidé à enrichir nos méthodes de travail en équipe à travers l'organisation des réunions, la définition des objectifs, la correction et l'adaptation du planning en fonction de l'avancement et l'évolution du projet.

De point de vue purement technique, nous avons appris comment résoudre un problème de cheminement, appliquer les algorithmes convenables et avoir la chance de mettre en œuvre un GPS. Nous avons également découvert le FRAMEWORK QT et nous avons mis en pratique nos connaissances en c++.

Difficultés rencontrées et solutions apportées :

La première difficulté que nous avons rencontrée lors de ce projet est bien évidemment le changement des circonstances (les réunions que par Zoom, difficultés en changement des idées). Cette exigence nous a permis d'apprendre à gérer des contraintes liées au temps, en mettant en place une organisation et une planification de l'ensemble de notre travail. N'oublions pas l'apprivoisement et l'apprentissage des langages de programmation, et l'utilisation de Qt. Nous nous sommes auto formée à l'aide d'ouvrages spécialisés, ce qui nous a permis de répondre aux objectifs que l'on s'est fixés.

Conclusion Générale :

Au terme de ce rapport, nous pouvons conclure que la mise en œuvre de ce projet technique était une expérience très enrichissante pour nous.

En effet, le traitement efficace du problème du plus court chemin ouvre une nouvelle voie pour résoudre des problèmes actuels et introduit aussi de nouveaux énoncés.

Les objectifs fixés au départ du projet pour arriver à ce résultat ont pu être atteints.

Par ailleurs, notre solution est encore améliorable. D'une part l'affichage graphique des solutions et d'autre part la sélection des options.

En dehors du cadre de notre projet, notre programme pourrait être encore amélioré en ajoutant de nouvelles fonctionnalités de gestion du graphe. D'autre part, il serait intéressant de pouvoir faire la manipulation autrement selon le choix d'un critère.

Nous espérons que ce projet offrira une meilleure visibilité de notre formation de première année Génie informatique au sein de l'ENSA Marrakech.

اللهم ولا تمكن المرض والبلاء منا ومن أهلكنا، اللهم أن هذا المرض جند من جنودك وأنت وحدك القادر
على رفعه عنا برحمتك وعفوك يا عفو يا كريم