RAPPORT DE PROJET

Réalisé par : DIOUBATE SEKOU et SANY ABDOUGUAFAROU, 1ère année LSI

7 juin 2012

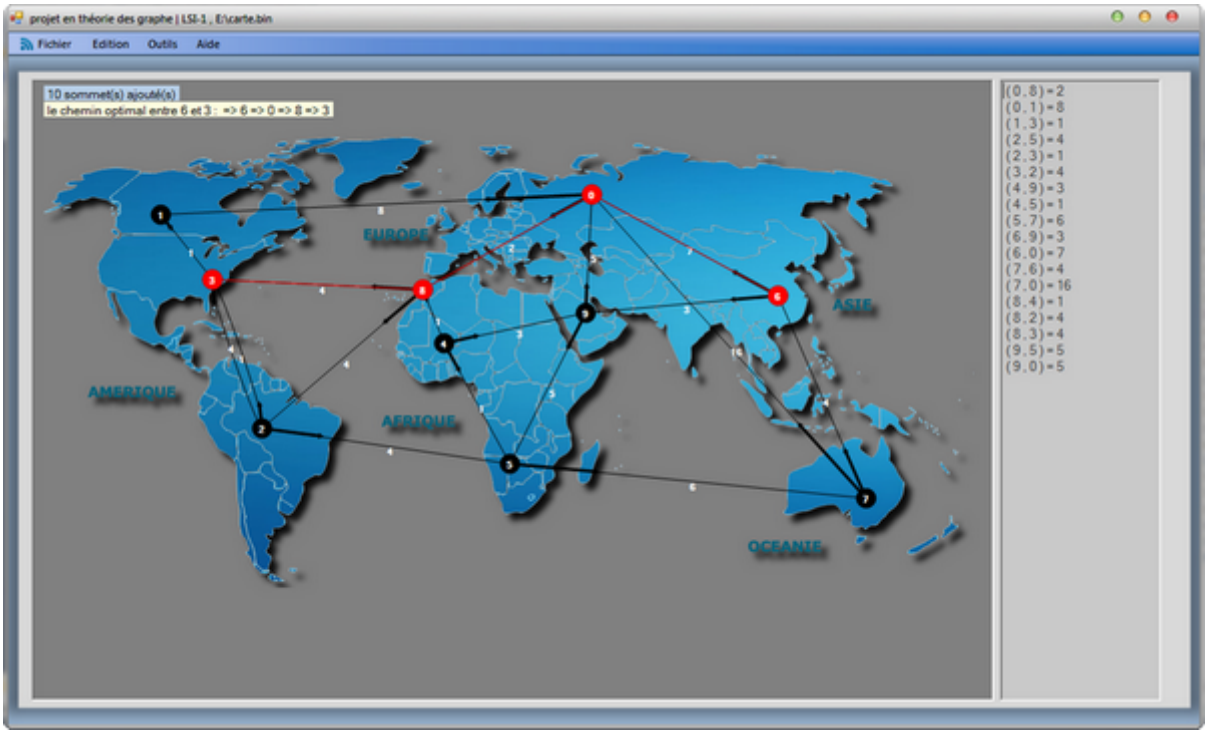


FIGURE 1 – La fenêtre principale avec une carte itinéraire

Théorie des graphes

Table des matières

1 Introduction

Dans ce rapport de projet de théorie des graphes, nous présentons une application desktop que nous avons développée à bases des notions appris en cours.

Pour concrétiser notre projet, nous avons décider de mettre en place un système de d'itinéraire à travers une carte afin de permettre les touristes, voyageurs désirant parcourir plusieurs pays, de mieux optimiser leur chemin. Cette application pourra servir dans plusieurs autres domaine que celui ci.

Cette application permet de faire la représentation d'un graphe orienté et non orienté, ainsi que se compose des fonctionnalités et algorithmes suivantes :

- Ajouter des Nœuds
- Supprimer des Nœuds
- Lier les Nœuds
- Générer la Matrice adjacente
- Donner Liste adjacent
- Modifier le graphe
- Enregistrer le graphe
- Ouvrir un graphe existant
- Faire le parcours en profondeur (algorithme Depth First Search)
- Faire le parcours en largeur (algorithme Breadth First Search)
- Faire une fermeture transitive (algorithme de Warshall)
- Trouver le chemin le plus cours entre deux Nœud (algorithme Dijkstra)
- Générer en détail un rapport général sur le graphe.

Dans ce rapport, nous parlerons d'abord de la théorie des graphes en premier lieu , où nous rappellerons l'historique et ce que c'est qu'un graphe, ainsi que quelques définitions de base de cette matière ;

Ensuite, on présentera les fonctionnalités de notre application ainsi qu'une explication détaillée sur l'implémentation du programme. Puis une conclure.

2 La théorie des graphes

2.1 Historique de la théorie des graphes

La théorie des graphes est très probablement née en 1735 lorsque Leonhard Euler (1707 - 1783) résout le problème des sept ponts de Königsberg (De nos jours Kaliningrad en Russie). Ce problème est très simple sur le principe mais un peu plus compliqué à démontrer, en voici l'énoncé : La vile de Königsberg est une ville autour d'un fleuve, elle compte quatre berges et sept ponts les reliant. Le but du jeu est de savoir s'il existe un chemin permettant d'emprunter tous les ponts une fois et une seule et revenir au point de départ. Le problème s'appelle, de façon plus formelle, la recherche d'un cycle eulérien dans un graphe. Euler a démontré que ce problème n'avait pas de solution.

2.2 Définition du graphe

En informatique, un graphe c'est une structure de donnée tout comme les tableaux, composée de nœuds interconnecter entre eux par des liens. Chaque nœud contient une ou plusieurs données de n'importe quel type. Un graphe est composé de sommets et d'arcs pour les graphes non orientés ou d'arêtes pour les graphes orientés. Un graphe G est défini de manière formelle par un couple (S, A) où :

- **S** est un ensemble fini d'éléments. Chacun de ces éléments est appelé sommet du graphe.
- **A** est un sous ensemble (éventuellement nul) de SxS. Chacun de ces éléments de A est appelé arc ou arête. L'ensemble A est donc composé de paires (x , y) où x et y étant appelés les extrémités de l'arc du graphe non orienté et appelés respectivement le départ et l'arrivé de l'arête pour les graphes orienté.

3 Présentation de application

FIGURE 2 – La fenêtre principale : carte itinéraire

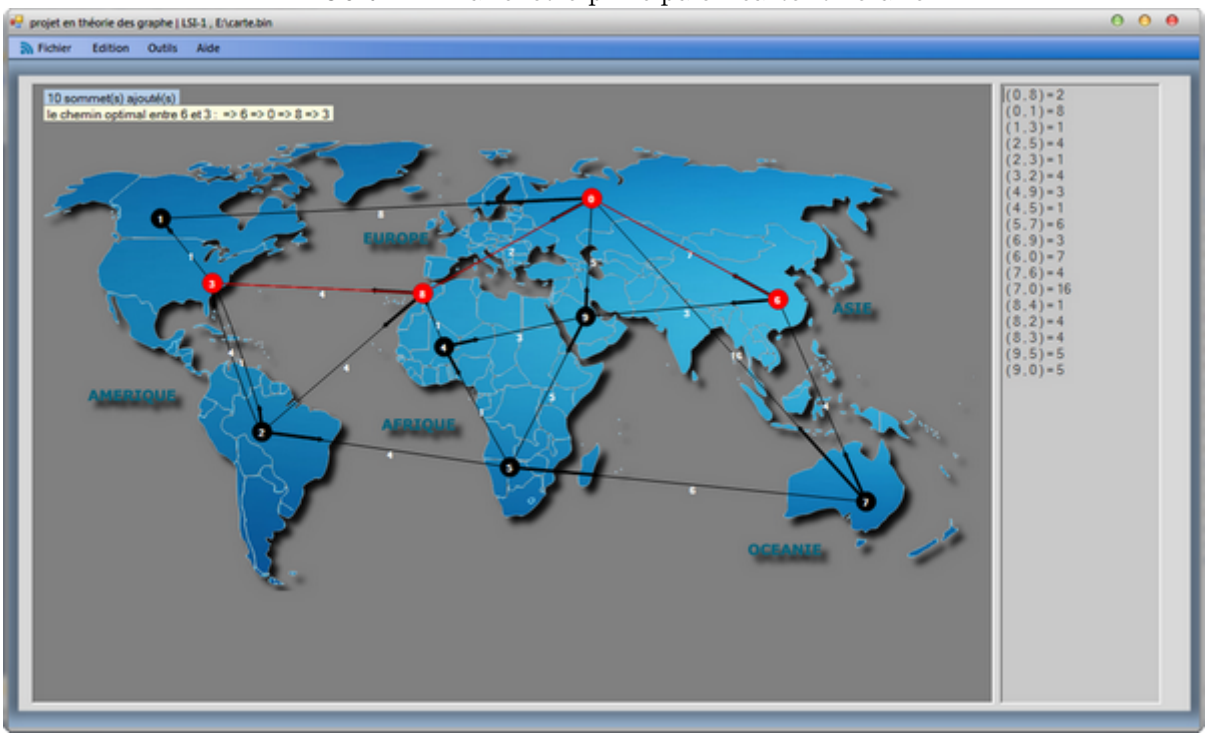
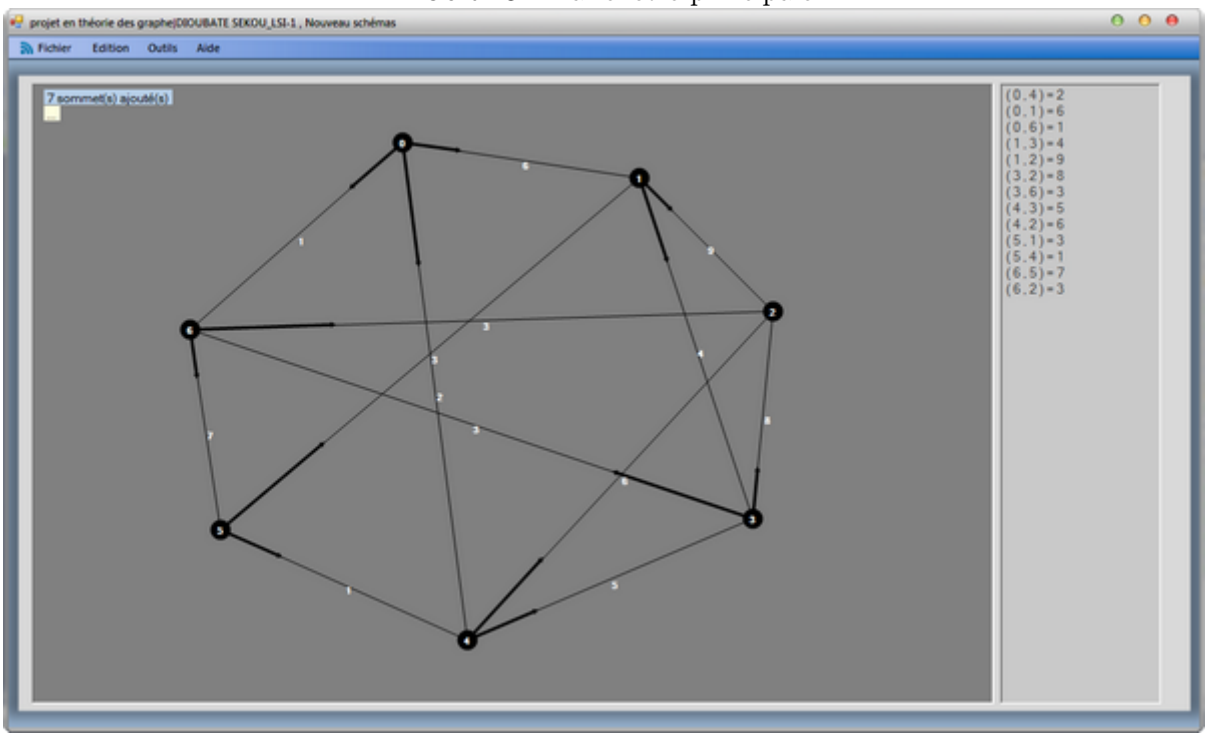


FIGURE 3 – La fenêtre principale



4 Fonctionnalités d'application

4.1 Représentation du graphe

Comme nous l'avons introduit plus haut, cette application permet de faire la représentation d'un graphe orienté et non orienté à travers les fonctions d'ajout et suppression des Nœuds ainsi que la fonction de liaison pour faire, suivant l'orientabilité du graphe, un arc ou une arête entre les Nœuds. **Pour faire un nouveau graphe, on procède comme suit :**

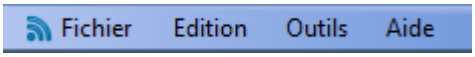


FIGURE 4 – Menu principal

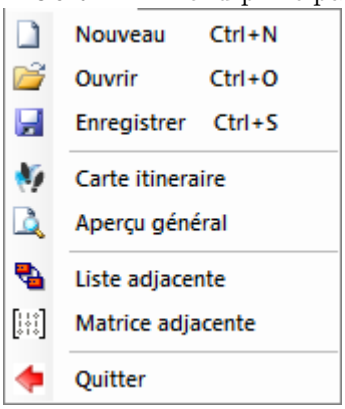


FIGURE 5 – Menu Fichier

- **Fichier => Nouveau** *(raccourci clavier : control + N)* ;
- **Selectionner** le type de graphe(graphe orienté ou graphe non orienté) dans la boite de dialogue apparaissant ;
- **Cocher** si avec ou sans carte dans la même boite de dialogue apparaissant ;

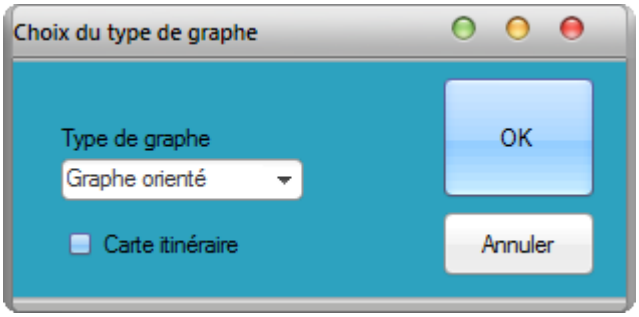


FIGURE 6 – Choix du type de graphe

- puis **OK** pour commencer ;

Le curseurs vous donne la main pour placer les Noeuds :

- **Click gauche** dans l'espace de traçasse pour placer des nœuds (autant de nœuds souhaités) ;
- **Click droite** pour finir, et le curseur redevient à son état initial, une flèche.

4.2 Modification

Une fois le graphe élaboré et tracé, on pourra le modifier par les fonctions d'ajouter, de suppression et de liaison, mais aussi déplacer ces nœuds et les ajuster.

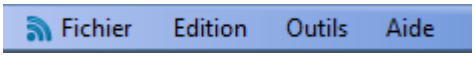


FIGURE 7 – Menu principal

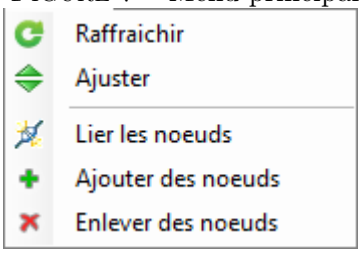


FIGURE 8 – Menu edition

Pour ajouter des nœuds :
- **Edition => Ajouter des nœuds** ;
- **Click gauche** dans l'espace de tracasse pour placer des nœuds (autant de nœuds souhaités) ;
- **Click droite** pour finir.

Pour supprimer des nœuds :
- **Edition => Enlever les Nœuds** ;
- **Cocher** le numéro des nœuds à supprimer, puis OK.

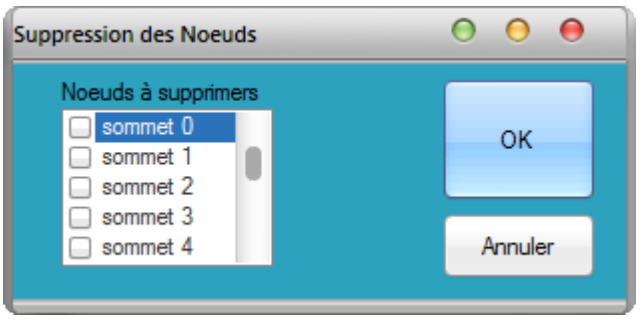


FIGURE 9 – Supression des Noeuds

Pour faire des liaisons :
- **Edition => lier les Nœuds** ;
- **Cliquer** dans un nœud de départ puis dans un nœud d'arrivé.
- **Taper** ensuite le cout de la liaison dans la boite de dialogue apparaissant. Avec la possibilité de faire autant de liaisons souhaitées ;
- **Cliquer** sur le bouton terminer pour finir.

Pour déplacer un nœud :
- **Cliquer** sur le nœud à déplacer puis sur le nouvel emplacement souhaiter dans l'espace de traçasse.

4.3 Modification avec la matrice d'adjacence

On a également la possibilité d'utiliser **la matrice d'adjacence** pour faire des modifications dans la structure du graphe. Pour se faire, il suffit de changer les valeurs et les champs des couts de la liaison dans la matrice.

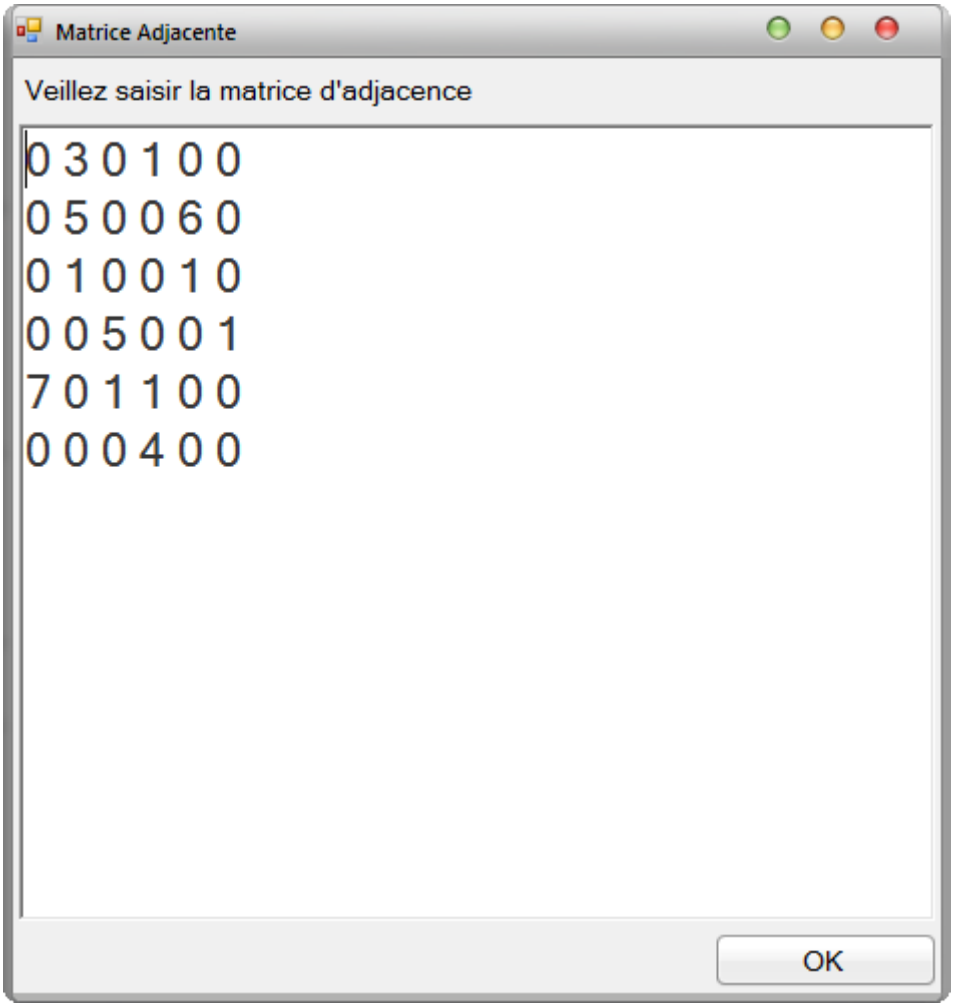


FIGURE 10 – La matrice adjacence

4.4 Sauvegarde et ouverture d'un graphe existant

Par un sérialisation de type binaire, on pourra enregistrer le graphe ainsi qu'avec tous ses variables, y compris sa matrice d'adjacence, dans un fichier sous format *.bin*. Ce travail pour être ouvert lors d'une exécution ultérieure par une procédure inverse qui est la desérialisation.

On procède ainsi :
- **Fichier => Ouvrir** *(raccourci clavier : control + O)*
- **Fichier => Enregistrer** *(raccourci clavier : control + S)*

4.5 Les fonctions d'outils :

Elles constituent l'exécution des différents algorithme du cours.

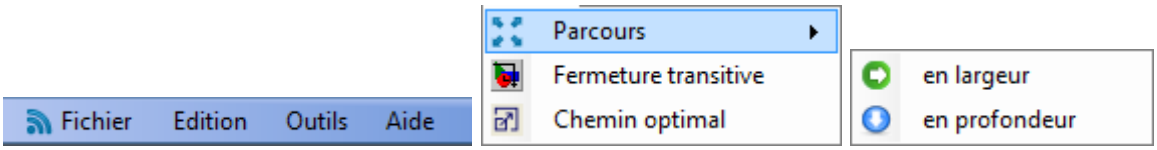


FIGURE 11 – Menu Outil

Pour en exécuter une :

- **Outil => parcours**, puis choisir le Nœud de départ.
- **Outil => fermeture transitive**
- **Outil => chemin optimal**, choisir le nœud de départ et d'arrivé.

4.6 Autres fonctionnalités

Avant de boucler cette partie, nous tenons à signaler d'autres fonctions , à savoir :

- Le menu contextuel : En plus des raccourcis clavier, l'application possède un menu contextuel qui permet d'accéder plus facilement aux différents fonctionnalités du programme.

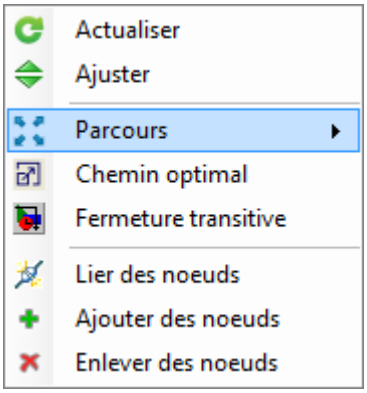


FIGURE 12 – Le menu contextuel

- La barre latéral des arrête : Elle s'interprète ainsi *(noeud A , noeud B) = cout de A à B*
- La liste d'adjacence : Elle permet de visionner pour un noeud donné, toutes les liaisons entrantes et sortant de ce noeud.
- Rapport : Elle donne tous les renseignement sur le graphe, à savoir : la taille du graphe, la matrice, la liste et les résultats des différents algorithmes appliqués au graphe.

5 L'implémentation du programme

Pour implémenté de cette application on a eu recours à l'utilisation du langage de programmation C-Sharp. Ce choix se rapport non seulement par sa simplicité et sa puissance mais surtout par ce que nous la maitrisons mieux que les autres langages de programmation orienté objet (POO).

Le programme est décomposé en deux parties. La première appelée « **forms des graphes** » est , comme son nom l'indique, constituée de *WindowsForm*. Et la seconde appelée « **traitement des graphes** », est une *bibliothèque de classes*. Commençons par la première qui est la plus importante.

5.1 Forms des graphes

C'est l'interface de l'application. Elle contient toutes les fonctions nécessaires au traçage et mise en forme de la représentation du graphe. On y retrouve entre autre :

-les fonctions d'ajout de Nœuds et de liaisons ;

-les fonctions de rafraichissement et d'ajustement de la représentation du schéma ;

-une classe nommée « classeDuDessin » qui implémente les fonctions qui permettent de dessiner des sommets et des liaisons sur l'interface Windows form ;

5.2 Traitement des graphes

Cette bibliothèque est le cœur du projet. Elle regroupe en son sein les classes de structure pour le graphe, telles que :*la classe sommet, la classe liaison, la classe graphe et la classe grapheBO*. Ainsi que les classes de l'implémentation des algorithmes de cours qui seront appliqués au graphe, à savoir :*la classe parcours, la classe Warsall et la classe Djikstra*.

5.2.1 La classe « sommet »

Elle contient les informations sur le Nœud et principalement : son numéro, ses coordonnées x et y dans le plans, un booléen pour le marquer dans les itérations et deux listes de liaisons entrant et sortant.

```
public class sommet
        {
            private int _numero;
            private int _x, _y;

            private bool _marquer = false;

            private List<liaison> _Liaisons_sortants;

            private List<liaison> _Liaisons_entrants;

            private int _cout;
                            // ne pas confondre avec le cout de la liaison
                     //ce cout ne sera utilisé que dans l algo de Djikstra,

            public sommet precedant;
                       //ce variable aussi est  pour  l algo de Djikstra
        }
```

5.2.2 La classe « liaison »

Elle contient deux variables : le cout de la liaison et le sommet lié.

```
public class liaison
   {
       public int cout; // le veritable cout de la liaison
       private sommet _sommet_lier;
       public liaison()
       {
           _sommet_lier = null; cout = -1;
       }
       public liaison(ref sommet _sommet_suivant1, int _cout)
       {
           _sommet_lier = _sommet_suivant1; cout = _cout;
       }
   }
```

5.2.3 La classe « graphe »

Elle est principalement la structure du graphe. Elle comporte non seulement la taille du graphe ou le nombre de Nœuds du graphe, mais aussi un tableau de sommet unidimensionnel qui contient l'ensemble des Nœuds. En plus, un tableau d'entiers double dimension qui constitue la matrice d'adjacence des couts. Elle comporte également un booléen indiquant si le graphe est orienté ou non.

```
public class Graphe
   {
       private bool _graphe_orienter;

       private int _Nombre_de_noeud;

       private sommet[] _table_de_sommets
       public  sommet[] Table_de_sommets
       {
           get { return _table_de_sommets; }
           set { _table_de_sommets = value; }
       }

       int[][] _matrice;//matrice d'adjacence
       public int[][] Matrice
       {
           get { return _matrice; }
           set { _matrice = value; }
       }

       public Graphe()
       {

       }
   }
```

5.2.4 La classe « grapheBO »

Qui pourrait se traduire *« graphe Business Objet »*, est la classe dans laquelle on crée un objet de type *graphe* et qui contient aussi quelques fonctions appliquées au graphe, telle que : la suppression des nœuds, la sérialisation et desérialisation.

```
public class GrapheBO
  {
      public static Graphe graphe = new Graphe(); //creation de l objet graphe

      public static void supprimer(int numero)
      {
          int indice_A_supprimer = -1;

          for (int i = 0; i < graphe.Nombre_de_noeud; i++)
          {
              foreach (liaison p in graphe.Table_de_sommets[i].Liaisons_sortants)
                  if (p.Sommet_lier.Numero == numero)
                  {
                      graphe.Table_de_sommets[i].Liaisons_entrants.Remove(p); break;
                  }

              foreach (liaison q in graphe.Table_de_sommets[i].Liaisons_sortants)
                  if (q.Sommet_lier.Numero == numero)
                  {
                      graphe.Table_de_sommets[i].Liaisons_sortants.Remove(q); break;
                  }

              if (graphe.Table_de_sommets[i].Numero == numero)
              {
                  indice_A_supprimer = i;
                  graphe.Table_de_sommets[i].Liaisons_sortants.Clear();
                  graphe.Table_de_sommets[i].Liaisons_entrants.Clear();
              }
          }
          graphe.Nombre_de_noeud --;
          for (int i = indice_A_supprimer; i < graphe.Nombre_de_noeud; i++)
          {
              graphe.Table_de_sommets[i] = graphe.Table_de_sommets[i + 1];
              graphe.Matrice[i] = graphe.Matrice[i + 1];
          }

          for (int i = 0; i < graphe.Nombre_de_noeud; i++)
          {
              for (int j = indice_A_supprimer; j < graphe.Nombre_de_noeud; j++)
              {
                  graphe.Matrice[i][j] = graphe.Matrice[i][j + 1];
              }
          }
      }
```

```
        public static void serialiser(string filename)
        {
            Stream stream = File.Open(filename, FileMode.Create);
            BinaryFormatter bFormatter = new BinaryFormatter();
            bFormatter.Serialize(stream, graphe);
            stream.Close();
        }

        public static void deserialiser(string filename)
        {
            Graphe mygraf;
            Stream stream = File.Open(filename, FileMode.Open);
            BinaryFormatter bFormatter = new BinaryFormatter();
            mygraf = (Graphe)bFormatter.Deserialize(stream);
            stream.Close();
            graphe.Graphe_orienter = mygraf.Graphe_orienter;
            graphe.Nombre_de_noeud = mygraf.Nombre_de_noeud;
            graphe.Table_de_sommets = mygraf.Table_de_sommets;
            graphe.Matrice = mygraf.Matrice;
        }
    }
```

5.2.5 La classe « Warsall »

Qui, comme son nom l'indique, contient l'implémentation de l'algorithme de WARHALL pour la fermeture transitive.

```
 public class Warshall
    {
       public static bool[][] fermeture_transitive()
       {
           int k, i, j, n=GrapheBO.graphe.Nombre_de_noeud;
           bool[][] matrice_temporaire = new bool[n][];

           for (i = 0; i < n; i++)
           {
               matrice_temporaire[i] = new bool[n];
               for (j = 0; j < n; j++)
                 matrice_temporaire[i][j] = (GrapheBO.graphe.Matrice[i][j] > 0)
                                                                  ? true : false;
           }
           for (k = 0; k < n; k++)
               for (i = 0; i < n; i++)
                   for (j = 0; j < n; j++)
                       matrice_temporaire[i][j] = matrice_temporaire[i][j]
                       || (matrice_temporaire[i][k] && matrice_temporaire[k][j]);
           return matrice_temporaire;
       }
}
```

5.2.6 La classe « Djikstra »

Comme son nom l'indique également, implémente le fameux algorithme de DJIKSTRA pour touver le chemin le plus court entre deux noeuds.

```
public class Dijkstra
    {
        static List<sommet> les_stations=new List<sommet>();

        //On déclare d abord une fonction d ouverture du Sommet de cout minimum
         // parmis les Sommets non visites

        public static sommet ouvre_min(List<sommet> liste)
        {
            sommet res = new sommet();
            res.Cout=int.MaxValue;
            foreach (sommet s in liste)
                {
                    if (s.Cout <= res.Cout)
                        res = s;
                }
            liste.Remove(res);
            return res;
        }

        //l'algo de Dijkstra proprement dit

        public static List<sommet> plus_court_chemin(sommet depart, sommet destination)
        {

           for (int i = 0; i < GrapheBO.graphe.Nombre_de_noeud; i++)
            {
                les_stations.Add( GrapheBO.graphe.Table_de_sommets[i]);
            }

            List<sommet> chemin = new List<sommet>();
            List<sommet> Sommets_a_ouvrir = new List<sommet>();
            sommet Sommet_ouvert = new sommet();

            //on initialise les couts à l'infini pour tous les Sommets
            foreach (sommet sta in les_stations)
                {
                    sta.Cout=int.MaxValue;
                    sta.precedant=null;
                    Sommets_a_ouvrir.Add(sta);
                }

            depart.Cout=0;
            Sommet_ouvert = ouvre_min(Sommets_a_ouvrir);
```

```
        //on verifie qu'il existe au moins un arc liant le Sommet ouvert
       // aux Sommets restants et que l'on n'est pas à l'arrivee

        while (Sommet_ouvert != destination)
        {
            foreach (sommet s in Sommets_a_ouvrir)
                {
                    int val=int.MaxValue;

                    foreach (liaison l1 in Sommet_ouvert.Liaisons_sortants)
                            if (l1.Sommet_lier.Numero == s.Numero)
                               val = Sommet_ouvert.Cout +l1.cout;

                    if (s.Cout > val)
                        {
                             s.Cout=val;
                             s.precedant=Sommet_ouvert;
                    }
                }
                Sommet_ouvert = ouvre_min(Sommets_a_ouvrir);

        }

        Sommets_a_ouvrir.Clear();

        if (destination.Cout == int.MaxValue)
        {
            //le graphe n'est pas connexe, il n y a pas de chemin
             return null;
        }
        else
        {
           //construction du chemin
            while (Sommet_ouvert != depart)
            {
                chemin.Add(Sommet_ouvert);
                Sommet_ouvert = Sommet_ouvert.precedant;
            }
        }

        chemin.Add(depart);

        chemin.Reverse();

        return chemin;
    }
}
```

6 Conclusion

Ce projet a été pour nous une grande opportunité d'approfondir nos connaissance dans cette matière qui est la théorie des graphes. En programmant ces algorithmes, nous les avons parfaitement assimiler si l'on peut dise ainsi. Également, ce projet nous a permit d'apprendre plus sur ce langage de programmation C sharp.