

Modélisation et Recherche Opérationnelle

Algorithmes sur les graphes :

(Prim, Floyd, Dijkstra et Bellman-Ford)



VOURIOT Thierry
Maîtrise d'Informatique
9 mai 2003
Groupe 1

Sommaire :

Introduction page 3

Structures de données page 4

- Graphe page 4
- Ensemble page 5

Algorithmes page 6

- Prim page 6
- Floyd page 7
- Dijkstra page 8
- Bellman-Ford page 9

Jeu d'essais page 10

Code source page 15

- graph.h page 15
- graph.cpp page 16
- ens.h page 19
- ens.cpp page 20
- main.h page 21
- main.cpp page 22

Introduction

La théorie des graphes est apparue en 1736 grâce à Euler qui démontra qu'il était impossible de traverser chacun des sept ponts, de la ville russe de Königsberg, une fois exactement et de revenir au point de départ. A partir de là, les graphes ont été utilisés dans différents domaines, tel que les mathématiques mais aussi la chimie (modélisation de structures), la biologie (représentation du génome), les sciences sociales (modélisation des relations), en industrie et aussi en informatique.

Cette notion a trouvé sa place dans le besoin de modéliser des liens entre différents objets ou entités. D'une manière générale, un graphe permet de représenter simplement et visuellement des structures complexes avec leurs connexions. C'est un moyen rapide et efficace d'obtenir et de présenter des données de structures complexes.

De ce fait, il est nécessaire d'avoir des méthodes de traitements de ces données afin de les interpréter pour mener une exploration. C'est dans ce contexte que le sujet de projet intervient. En effet, il est parfois nécessaire d'extraire d'un graphe un cheminement entre tous les éléments de ce graphe (c'est la notion d'arborescence recouvrante) ou encore de trouver le plus court chemin d'un élément du graphe à un autre.

Ainsi le projet consiste à implanter les graphes et à programmer des algorithmes sur ces graphes. L'algorithme de Prim permettant de trouver l'arbre couvrant minimal ainsi que les algorithmes de Floyd, Dijkstra et Bellman-Ford permettant de résoudre le problème des plus courts chemins ont été implanté dans le logiciel.

Le langage de programmation choisi est le C++ car son orientation objet permet une meilleure représentation des structures de données. Le logiciel fonctionne en mode console (pas d'interface graphique) et tourne aussi bien sous Windows que sous Linux/Unix.

Structures de données

Graphe :

Spécification algébrique de la structure de données permettant la représentation des graphes :

spec GRAPHE

sorte Graphe

operations

```
# Création d'un graphe vide
GrapheNouv: -> Graphe
# Retourne la taille du graphe (sommets de 0 à taille-1)
GetSize: Graphe -> Int
# Change la taille du graphe (sommets de 0 à taille-1)
SetSize: Graphe Int -> Graphe
# Retourne la valuation d'un arc ou d'une arête
GetArc: Graphe Int Int -> Int
# Change la valuation d'un arc (infini si l'arc n'existe pas)
SetArc: Graphe Int Int Int -> Graphe
# Change la valuation d'une arête (infini si l'arête n'existe pas)
SetArete: Graphe Int Int Int -> Graphe
# Teste si un arc ou une arête existe
IsArc: Graphe Int Int -> Bool
# Teste si un sommet appartient au graphe
IsElem: Graphe Int -> Bool
# Permet à l'utilisateur de rentrer un graphe au clavier
SetGraph: -> Graphe
# Copie un graphe
Copy: Graphe -> Graphe
# ajoute le sommet taille au graphe (augmente la taille de +1)
AddElem: Graphe -> Graphe
# Teste si l'ensemble des sommets est egal à un autre ensemble
ElmsEqualEns: Graphe Ensemble -> Bool
# Retourne l'ensemble des successeurs d'un sommet
Succ: Graphe Int -> Ensemble
```

fspec

Cette spécification permet la représentation des graphes pondérés orientés ou non orientés, par contre l'ensemble des sommets est exactement les nombres allant de 0 à taille-1 (par exemple un graphe de taille 3 aura obligatoirement comme sommet 0 1 et 2).

La définition de la classe CGraph est disponible dans le code source dans le fichier `graph.h` et l'implantation est dans le fichier `graph.cpp`.

Ensemble :

Spécification algébrique de la structure de données permettant la représentation des ensembles :

```
spec ENSEMBLE

    sorte Ensemble

    operations

    # Création d'un ensemble vide
    EnsNouv: -> Ensemble
    # Ajoute un élément à l'ensemble
    AddElem: Ensemble Int -> Ensemble
    # Supprime un élément à l'ensemble
    DelElem: Ensemble Int -> Ensemble
    # Teste si un élément appartient à l'ensemble
    IsElem: Ensemble Int -> Bool
    # Affiche l'ensemble à l'écran
    ShowEns: Ensemble ->
    # Retourne le nombre d'éléments de l'ensemble
    GetSize: Ensemble -> Int

fspec
```

Cette spécification permet la représentation des ensembles d'entiers.

La définition de la classe CEns est disponible dans le code source dans le fichier `ens.h` et l'implantation est dans le fichier `ens.cpp`.

Algorithmes

Algorithmes permettant de trouver l'arbre couvrant minimum.

Le problème abordé ici correspond à la minimisation du coût de construction d'un réseau minimal de communications connaissant le coût de chaque liaison possible entre les noeuds du réseau à construire. Puisqu'on souhaite un réseau minimal, on recherche un arbre couvrant (i.e. le graphe induit connexe qui a le minimum d'arêtes). On doit de plus tenir compte du coût de construction : chaque arête sera affectée d'un poids par une fonction p à valeurs dans \mathbb{R}^+ et le coût total sera égal à la somme des coûts des arêtes de l'arbre. On dit que l'on recherche un arbre couvrant minimal.

Prim :

Soit X l'ensemble des sommets du graphe G . On utilisera un ensemble d'arêtes T qui sera en sortie l'arbre couvrant minimal et un ensemble de sommets S qui contiendra les sommets de T .

L'algorithme de Prim fait pousser un arbre couvrant minimal en ajoutant au sous-arbre T déjà construit une nouvelle branche parmi les arêtes de poids minimal joignant un sommet de T à un sommet n'appartenant pas à ce dernier.

L'algorithme s'arrête lorsque tous les sommets du graphe appartiennent à T .

```
procedure prim(G : Graphe)

    choisir un sommet x de G.
    S ← S ∪ x
    T ← 0

    tantque S ≠ X faire

        trouver une arête {ys} de poids minimal tel que y ∈ X-S et s ∈ S
        T ← T ∪ {ys}

        S ← S ∪ y

    fintantque
finprocedure
```

Algorithmes permettant de résoudre le problème du plus court chemin.

Le problème posé ici est de déterminer dans un graphe valué (orienté ou non orienté) le plus court chemin entre deux sommets. Les arêtes (arcs) du graphe seront donc affectées d'une valeur par une fonction v à valeurs dans \mathbb{R} et on calculera la longueur d'un chemin (chaîne) par la somme des valeurs de ses arêtes (arcs).

Floyd :

On représente un graphe valué orienté à n sommets par une matrice carré (n,n) de nombres : $M[i,j]$ a pour valeur le poids de l'arc $i \rightarrow j$ ($+\infty$ si l'arc n'existe pas).

L'algorithme permet de calculer la matrice D telle que $D[i,j]$ soit le poids minimal d'un chemin de i à j .

Cet algorithme résout le problème des plus courtes distances de tout sommet du graphe à tout autre sommet, mais ne donne pas le chemin correspondant.

```
procedure floyd(G : Graphe)

  D <- M
  pour i de 1 à n faire
    pour j de 1 à n faire
      si D[j,i] < +∞ alors
        pour k de 1 à n faire
          si D[j,k] > D[j,i] + D[i,k] alors
            D[j,k] <- D[j,i] + D[i,k]
          finsi
        finpour
      finsi
    finpour
  finpour
finprocedure
```

Dijkstra :

Un sommet x étant fixé, cet algorithme est un algorithme glouton qui construit progressivement un ensemble de sommets pour lesquels on connaît un plus court chemin depuis x . A chaque étape on choisit un sommet dont la distance à x est minimale parmi ceux qui n'ont pas encore été choisis.

On utilisera un ensemble C de sommets et deux tableaux D et P indexés par les sommets du graphe : D tableau de réels et P tableau de sommets.

n représente le nombre de sommets du graphe.

$v(u,v)$: permet d'obtenir le coût d'un arc du sommet u au sommet v .

Le tableau P contiendra les pères des sommets dans un plus court chemin depuis le sommet x .

```

procedure dijkstra(G : Graphe, x : sommet)

    C <- {x} ; D[x] <- 0

    pour tout sommet s!=x faire
        D[s] <- v(x,s) ou  $+\infty$  si s n'est pas un successeur de x
        P[s] <- x
    finpour

    tantque |C| < n faire
        choisir y tel que D[y] = min{D[z] ; z !∈ C}
        si D[y] =  $+\infty$  alors fin finsi

        C <- C U {y}

        pour chaque sommet z !∈ C faire
            si D[z] > D[y] + v(yz) alors
                D[z] <- D[y] + v(yz)
                P[z] <- y
            finsi
        finpour
    fintantque

finprocedure

```


Bellman-Ford :

Le troisième algorithme qui répond à la question du plus court chemin dans des temps raisonnables est celui de Bellman-Ford. L'apport de cet algorithme à celui de Dijkstra est la possibilité d'avoir des poids négatifs entre les sommets. De plus cet algorithme permet de détecter les circuits absorbants (en renvoyant faux).

V : l'ensemble des sommets du graphe;

E : l'ensemble des arcs (ou arêtes) du graphe;

D : un tableau de coût de longueur |V| (le nombre de sommet du graphe);

P : tableau contenant les pères des sommets dans un plus court chemin depuis le sommet x.

x : l'indice du nœud de départ;

v(u,v) : permet d'obtenir le coût d'un arc du sommet u au sommet v appartenant à E.

```
procedure bellmanford(G : Graphe, x : sommet)

  pour tout sommets i de V faire
    D[i] <- +∞
    P[i] <- x
  finpour

  D[x] <- 0
  P[x] <- x

  pour k de 1 à |V| - 1 faire
    pour tout arcs (u,v) de E faire
      D[v] <- min(D[v], D[u] + v(u,v))
    finpour
  finpour

  pour tout arcs (u,v) de E faire
    si D[v] > D[u] + v(u,v) alors
      retourne faux
    finsi
  finpour

  retourne vrai

finprocedure
```

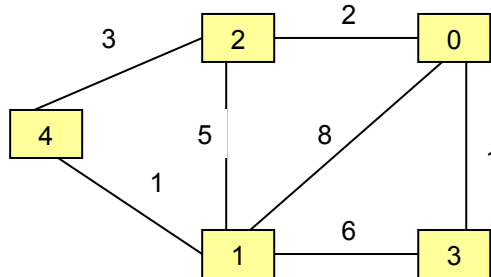
Tous les algorithmes décrits précédemment sont implantés dans le fichier `main.cpp`.

Jeu d'essais

Exemple 1 :

Graphe valué positivement non orienté :

(algorithmes applicables : Prim, Floyd, Dijkstra et Bellman-Ford)



On applique l'algorithme de Floyd :

```

D:\Programmation\C_Cpp\Structures de donnees\GraphAlgos\Release\GraphAlgos.exe
Projet Maitrise Informatique 2003
Auteur: Vouriot Thierry
http://thierry.vouriot.free.fr

Algorithmes s'appliquant sur des graphes simples orientes et values :
Graphe :
[1] Saisir manuellement un graphe
[2] Utiliser graphe d'exemple 1 <Prim, Dijkstra, Bellman-Ford ou Floyd>
[3] Utiliser graphe d'exemple 2 <Floyd, Bellman-Ford>
[4] Utiliser graphe d'exemple 3 <Bellman-Ford>
[5] Utiliser graphe d'exemple 4 <Prim, Dijkstra, Bellman-Ford ou Floyd>
[6] Utiliser graphe d'exemple 5 <Dijkstra, Bellman-Ford ou Floyd>
[7] Quitter le programme
Choix : ? 2

Graphe exemple 1 :
non oriente (valuation positive)
avec circuits pour l'algo de prim, bellman-ford, dijkstra ou floyd
  0  1  2  3  4
0  x  8  2  1  x
1  8  x  5  6  1
2  2  5  x  x  3
3  1  6  x  x  x
4  x  1  3  x  x

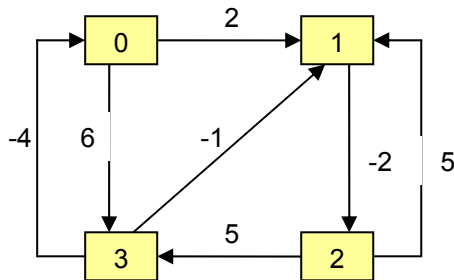
Algorithme a appliquer :
[1] Floyd
[2] Prim
[3] Dijkstra
[4] Bellman-Ford
[5] Revenir au menu principal
Choix : ? 1

Algorithme de Floyd
Matrice montrant les plus courtes distances
de tout sommet du graphe a tout autre sommet.
  0  1  2  3  4
0  x  6  2  1  5
1  6  x  4  6  1
2  2  4  x  3  3
3  1  6  3  x  6
4  5  1  3  6  x

Appuyez sur entree pour continuer ...
  
```

Exemple 2 :

Graphe valué positivement et négativement orienté :
(algorithmes applicables : Floyd et Bellman-Ford)



On applique l'algorithme de Bellman-Ford:

```

C:\Sélectionner D:\Programmation\C_Cpp\Structures de donnees\GraphAlgos\Release\GraphAlgos.exe
Projet Maitrise Informatique 2003
Auteur: Vouriot Thierry
http://thierry.vouriot.free.fr

Algorithmes s'appliquant sur des graphes simples orientes et values :

Graphe :
[1] Saisir manuellement un graphe
[2] Utiliser graphe d'exemple 1 <Prim, Dijkstra, Bellman-Ford ou Floyd>
[3] Utiliser graphe d'exemple 2 <Floyd, Bellman-Ford>
[4] Utiliser graphe d'exemple 3 <Bellman-Ford>
[5] Utiliser graphe d'exemple 4 <Prim, Dijkstra, Bellman-Ford ou Floyd>
[6] Utiliser graphe d'exemple 5 <Dijkstra, Bellman-Ford ou Floyd>
[7] Quitter le programme
Choix : ? 3

Graphe exemple 2 :
orienté (valuation positive et negative)
avec circuits pour l'algo de floyd ou bellman-ford
  0  1  2  3
0  x  2  x  6
1  x  x -2  x
2  x  5  x  5
3 -4 -1  x  x

Algorithme a appliquer :
[1] Floyd
[2] Prim
[3] Dijkstra
[4] Bellman-Ford
[5] Revenir au menu principal
Choix : ? 4

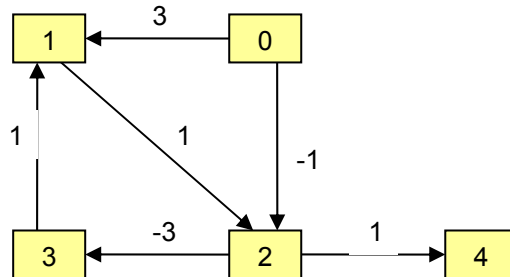
Donner le sommet de depart de l'algo : ?
0

Algorithme de Belman-Ford
Le tableau suivant contient les peres des sommets
dans un plus court chemin depuis le sommet x
Predecesseur [0] = 0
Predecesseur [1] = 0
Predecesseur [2] = 1
Predecesseur [3] = 2

Appuyez sur entree pour continuer ...
  
```

Exemple 3 :

Graphe valué positivement et négativement orienté contenant un circuit absorbant :
(algorithme applicable : Bellman-Ford)



On applique l'algorithme de Bellman-Ford :

```

D:\Programmation\C_Cpp\Structures de donnees\GraphAlgos\Release\GraphAlgos.exe
Projet Maitrise Informatique 2003
Auteur: Vouriot Thierry
http://thierry.vouriot.free.fr

Algorithmes s'appliquant sur des graphes simples orientes et values :

Graphe :
[1] Saisir manuellement un graphe
[2] Utiliser graphe d'exemple 1 <Prim, Dijkstra, Bellman-Ford ou Floyd>
[3] Utiliser graphe d'exemple 2 <Floyd, Bellman-Ford>
[4] Utiliser graphe d'exemple 3 <Bellman-Ford>
[5] Utiliser graphe d'exemple 4 <Prim, Dijkstra, Bellman-Ford ou Floyd>
[6] Utiliser graphe d'exemple 5 <Dijkstra, Bellman-Ford ou Floyd>
[7] Quitter le programme
Choix : ? 4

Graphe exemple 3 :
orienté (valuation positive et negative)
avec circuit absorbant pour l'algo de bellman-ford
  0  1  2  3  4
0 x  3 -2 x x
1 x  x  1 x x
2 x  x  x -3 1
3 x  1  x x x
4 x  x  x x x

Algorithme a appliquer :
[1] Floyd
[2] Prim
[3] Dijkstra
[4] Bellman-Ford
[5] Revenir au menu principal
Choix : ? 4

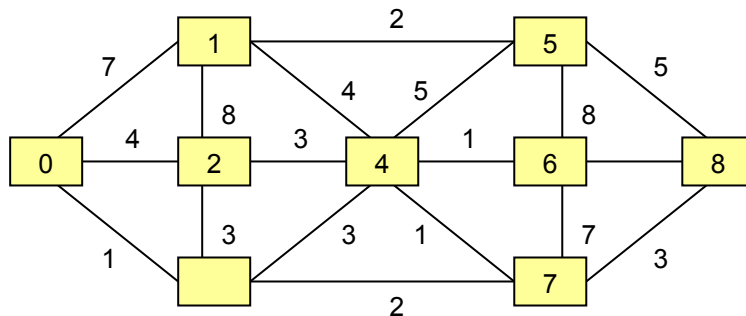
Donner le sommet de depart de l'algo : ?
3

Algorithme de Belman-Ford
Circuit absorbant detecte

Appuyez sur entree pour continuer ...
  
```

Exemple 4 :

Graphe valué positivement non orienté :
(algorithmes applicables : Prim, Floyd, Dijkstra et Bellman-Ford)



On applique l'algorithme de Prim :

```

D:\Programmation\C_Cpp\Structures de donnees\GraphAlgos\Release\GraphAlgos.exe
Projet Maitrise Informatique 2003
Auteur: Uouriot Thierry
http://thierry.vouriot.free.fr

Algorithmes s'appliquant sur des graphes simples orientes et values :

Graphe :
[1] Saisir manuellement un graphe
[2] Utiliser graphe d'exemple 1 (Prim, Dijkstra, Bellman-Ford ou Floyd)
[3] Utiliser graphe d'exemple 2 (Floyd, Bellman-Ford)
[4] Utiliser graphe d'exemple 3 (Bellman-Ford)
[5] Utiliser graphe d'exemple 4 (Prim, Dijkstra, Bellman-Ford ou Floyd)
[6] Utiliser graphe d'exemple 5 (Dijkstra, Bellman-Ford ou Floyd)
[7] Quitter le programme
Choix : ? 5

Graphe exemple 4 (ou en ID):
non oriente (valuation positive)
pour l'algo de prim, bellman-ford, dijkstra ou floyd
  0 1 2 3 4 5 6 7 8
0 x 7 4 1 x x x x x
1 7 x 8 x 4 2 x x x
2 4 8 x 4 3 x x x x
3 1 x 4 x 3 x x 2 x
4 x 4 3 3 x 5 1 1 x
5 x 2 x x 5 x 8 x 5
6 x x x x 1 8 x 7 10
7 x x x 2 1 x 7 x 3
8 x x x x x 5 10 3 x

Algorithme a appliquer :
[1] Floyd
[2] Prim
[3] Dijkstra
[4] Bellman-Ford
[5] Revenir au menu principal
Choix : ? 2

Algorithme de Prim
S = { 0 }
Arete de poids minimum dont un seul sommet appartient a S : {3 0}
S = { 0 3 }
Arete de poids minimum dont un seul sommet appartient a S : {7 3}
S = { 0 3 7 }
Arete de poids minimum dont un seul sommet appartient a S : {4 7}
S = { 0 3 4 7 }
Arete de poids minimum dont un seul sommet appartient a S : {6 4}
S = { 0 3 4 6 7 }
Arete de poids minimum dont un seul sommet appartient a S : {2 4}
S = { 0 2 3 4 6 7 }
Arete de poids minimum dont un seul sommet appartient a S : {8 7}
S = { 0 2 3 4 6 7 8 }
Arete de poids minimum dont un seul sommet appartient a S : {1 4}
S = { 0 1 2 3 4 6 7 8 }
Arete de poids minimum dont un seul sommet appartient a S : {5 1}
S = { 0 1 2 3 4 5 6 7 8 }

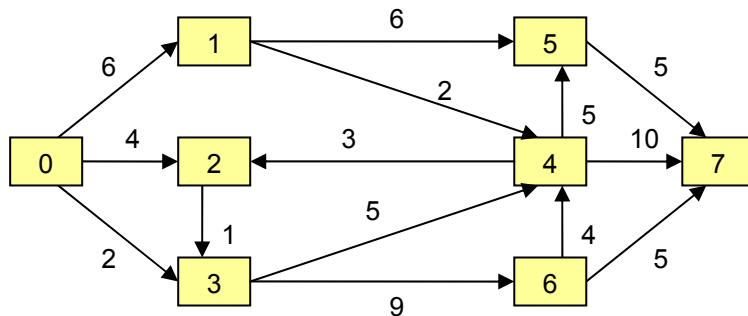
Matrice montrant l'arbre couvrant minimal
  0 1 2 3 4 5 6 7 8
0 x x x x 1 x x x x
1 x x x x x 4 2 x x x
2 x x x x x 3 x x x x
3 1 x x x x x x x 2 x
4 x 4 3 x x x 1 1 x
5 x 2 x x x x x x x
6 x x x x 1 x x x x
7 x x x x 2 1 x x x 3
8 x x x x x x x 3 x

Appuyez sur entree pour continuer ...

```


Exemple 5 :

Graphe valué positivement orienté :
(algorithmes applicables : Floyd, Dijkstra et Bellman-Ford)



On applique l'algorithme de Dijkstra :

```

D:\Programmation\C_Cpp\Structures de donnees\GraphAlgos\Release\GraphAlgos.exe
Projet Maitrise Informatique 2003
Auteur: Vouriot Thierry
http://thierry.vouriot.free.fr

Algorithmes s'appliquant sur des graphes simples orientes et values :
Graphe :
[1] Saisir manuellement un graphe
[2] Utiliser graphe d'exemple 1 <Prim, Dijkstra, Bellman-Ford ou Floyd>
[3] Utiliser graphe d'exemple 2 <Floyd, Bellman-Ford>
[4] Utiliser graphe d'exemple 3 <Bellman-Ford>
[5] Utiliser graphe d'exemple 4 <Prim, Dijkstra, Bellman-Ford ou Floyd>
[6] Utiliser graphe d'exemple 5 <Dijkstra, Bellman-Ford ou Floyd>
[7] Quitter le programme
Choix : ? 6

Graphe exemple 5 <vu en TD>:
orienté <valuation positive>
pour l'algo de bellman-ford, dijkstra ou floyd
  0  1  2  3  4  5  6  7
0 x  6  4  2  x  x  x  x
1 x  x  x  x  2  6  x  x
2 x  x  x  1  x  x  x  x
3 x  x  x  x  5  x  9  x
4 x  x  3  x  x  5  x  10
5 x  x  x  x  x  x  x  5
6 x  x  x  x  4  x  x  5
7 x  x  x  x  x  x  x  x

Algorithme a appliquer :
[1] Floyd
[2] Prim
[3] Dijkstra
[4] Bellman-Ford
[5] Revenir au menu principal
Choix : ? 3

Donner le sommet de depart de l'algo : ?
0

Algorithme de Dijkstra

Le tableau suivant donne les distances depuis le sommet de depart x
au cours des differentes etapes de l'algo
sommet  0  1  2  3  4  5  6  7
init     0  6  4  2  I  I  I  I
        0  6  4  2  7  I  11  I
        0  6  4  2  7  I  11  I
        0  6  4  2  7  12  11  I
        0  6  4  2  7  12  11  17
        0  6  4  2  7  12  11  16
        0  6  4  2  7  12  11  16
        0  6  4  2  7  12  11  16

Le tableau suivant contient les peres des sommets
dans un plus court chemin depuis le sommet x
Predecesseur [0] = 0
Predecesseur [1] = 0
Predecesseur [2] = 0
Predecesseur [3] = 0
Predecesseur [4] = 3
Predecesseur [5] = 1
Predecesseur [6] = 3
Predecesseur [7] = 6

Appuyez sur entree pour continuer ...
  
```

Code source

graph.h

```
// GraphAlgos
// Programme implantant les algos de Floyd, Prim, Dijkstra et Bellman-Ford
// graph.h : classe CGraph definissant les graphes dont les sommets sont entiers
// "attention les sommets vont de 0 à la taille du graphe"
// Date : 08/05/2003
// Auteur: Vouriot Thierry
// http://thierry.vouriot.free.fr
// Date : 08/05/2003

#ifndef _GRAPH_H
#define _GRAPH_H

#include <iostream>
#include <math.h>
#include <malloc.h>
#include "ens.h"

using namespace std;

#define INF 99999

// #####
// Classe CGraph
// #####
class CGraph
{
public:

    // Constructeur
    CGraph();
    // Destructeur
    ~CGraph();
    // Retourne le nombre de sommets du graphe
    int GetSize();
    // Change le nombre de sommets du graphe
    void SetSize(int som);
    // Retourne l'arc m_gr[i][j]
    int GetArc(int i, int j);
    // Change l'arc [i][j] à la valeur val
    void SetArc(int i, int j, int val);
    // Change l'arete (les 2 arcs dans les deux sens)
    void SetArete(int i, int j, int val);
    // Retourne vrai si un arc i-j existe dans le graphe
    bool IsArc(int i, int j);
    // Teste si i est un sommet du graphe
    bool IsElem(int i);
    // Affiche la matrice représentant le graphe
    void ShowGraph();
    // Permet de saisir un graphe au "clavier"
    void SetGraph();
    // Retourne la valuation minimale de tous les arcs
    int GetArcMin();
    // Retourne le nombre d'arcs
    int GetNbArcs();
    // Retourne une copie du graphe
    CGraph * Copy();
    // Ajoute le sommet size au graphe
    void AddElem();
    // Teste si l'ensemble e est egale à l'ensemble des sommets du graphe
    bool ElemsEqualEns(CEns e);
```

```

        // Retourne l'ensemble des successeurs d'un sommet
        CEns Succ(int x);
private:

        // Nombre de sommets (les sommets du graphe vont de 0 à size-1)
        int m_size;
        // Matrice correspondants aux arcs
        int m_gr[1000][1000];

};

#endif

```

graph.cpp

```

// GraphAlgos
// Programme implantant les algos de Floyd, Prim, Dijkstra et Bellman-Ford
// graph.cpp : implantation de la classe CGraph
// Date : 08/05/2003
// Auteur: Vouriot Thierry
// http://thierry.vouriot.free.fr
// Date : 08/05/2003

#include "graph.h"

// #####
// Classe CGraph
// #####

// Constructeur
CGraph::CGraph()
{
    m_size = 0;
}

// Destructeur
CGraph::~CGraph()
{ }

// Retourne le nombre de sommets du graphe
int CGraph::GetSize()
{
    return m_size;
}

// Change le nombre de sommets du graphe
void CGraph::SetSize(int size)
{
    m_size = size;
    for (int i=0; i<this->m_size; i++)
        for (int j=0; j<this->m_size; j++)
        {
            m_gr[i][j] = INF;
        }
}

// Retourne l'arc m_gr[i][j]
int CGraph::GetArc(int i, int j)
{
    return m_gr[i][j];
}

// Change l'arc [i][j] à la valeur val
void CGraph::SetArc(int i, int j, int val)
{
    this->m_gr[i][j] = val;
}

```



```

// Change l'arete (les 2 arcs dans les deux sens)
void CGraph::SetArete(int i, int j, int val)
{
    this->m_gr[i][j] = val;
    this->m_gr[j][i] = val;
}

// Retourne vrai si un arc i-j existe dans le graphe
bool CGraph::IsArc(int i, int j)
{
    return (this->m_gr[i][j] != INF);
}

// Teste si i est un sommet du graphe
bool CGraph::IsElem(int i)
{
    return (i < m_size);
}

// Affiche la matrice représentant le graphe
void CGraph::ShowGraph()
{
    int i, j;
    cout << endl << " ";

    for (i=0; i<m_size; i++) cout << i << " ";
    cout << endl;
    for (i=0; i<m_size; i++)
    {
        cout << i << " ";
        for (j=0; j<m_size; j++)
            if (this->m_gr[i][j] != INF)
                cout << m_gr[i][j] << " ";
            else
                cout << "X ";
        cout << endl;
    }
}

// Permet de saisir un graphe au "clavier"
void CGraph::SetGraph()
{
    int tmp, size;
    tmp=3;
    cout << "Nombre de sommets du graphe : ";
    cin >> size;
    this->SetSize(size);
    for (int i=0; i<this->m_size; i++)
        for (int j=0; j<this->m_size; j++)
        {
            if(i!=j)
            {
                cout << "Arc entre " << i << " et " << j << " (1 pour oui
et 0 pour non) : ";
                cin >> tmp;

                while (tmp < 0 || tmp > 1)
                {
                    cout << "1 pour oui et 0 pour non" << endl;
                    cin >> tmp;
                }
                if (tmp == 1)
                {
                    cout << "Valuation de l'arc entre " << i << " et "
<< j << " :";
                    cin >> m_gr[i][j];
                }
            }
        }
    }
}

```

```

        else
            this->m_gr[i][j] = INF;
    }
    else this->m_gr[i][j] = INF;
}

// Retourne la valuation minimale de tous les arcs
int CGraph::GetArcMin()
{
    int i,j,min=0;
    for (i=0; i<m_size; i++)
        for (j=0; j<m_size; j++)
            if (this->m_gr[i][j] != INF)
            {
                min = this->m_gr[i][j];
                break;
            }

    for (i=0; i<m_size; i++)
        for (j=0; j<m_size; j++)
            if (m_gr[i][j]<=min && i!=j && this->m_gr[i][j] != INF)
                min = m_gr[i][j];

    return min;
}

// Retourne le nombre d'arcs
int CGraph::GetNbArcs()
{
    int x,i,j;
    x=0;
    for (i=0; i<m_size; i++)
        for (j=0; j<m_size; j++)
            if (m_gr[i][j] != INF)
                x++;

    return x;
}

// Retourne une copie du graphe
CGraph * CGraph::Copy()
{
    CGraph * tmp = new CGraph();

    tmp->m_size = this->m_size;

    for (int i=0; i<m_size; i++)
        for (int j=0; j<m_size; j++)
        {
            tmp->m_gr[i][j]=this->m_gr[i][j];
        }

    return tmp;
}

// Ajoute le sommet size au graphe
void CGraph::AddElem()
{
    m_size++;
    for (int i=0; i<m_size; i++)
    {
        this->m_gr[i][m_size-1] = INF;
        this->m_gr[m_size-1][i] = INF;
    }
}

// Teste si l'ensemble e est egale à l'ensemble des sommets du graphe
bool CGraph::ElmsEqualEns(CEns e)

```

```

{
    for (int i=0; i<m_size; i++)
        if (!e.IsElem(i)) return false;
    return true;
}

// Retourne l'ensemble des successeurs d'un sommet
CEns CGraph::Succ(int x)
{
    CEns ens = CEns();
    for(int j=0; j<m_size; j++)
    {
        if (IsArc(x,j))
            ens.AddElem(j);
    }
    return ens;
}

```

ens.h

```

// GraphAlgos
// Programme implantant les algos de Floyd, Prim, Dijkstra et Bellman-Ford
// ens.h : classe CEns definissant les ensembles d'entiers
// Date : 08/05/2003
// Auteur: Vouriot Thierry
// http://thierry.vouriot.free.fr
// Date : 08/05/2003

#ifndef _ENS_H
#define _ENS_H

#include <iostream>
#include <math.h>
#include <malloc.h>

using namespace std;

#define N 256

// #####
// Classe CEns
// #####
class CEns
{
public:
    // Constructeur
    CEns();
    // Destructeur
    ~CEns();
    // Ajoute un élément à l'ensemble
    void AddElem(int i);
    // Supprime un élément de l'ensemble
    void DelElem(int i);
    // Teste si un élément appartient à l'ensemble
    bool IsElem(int);
    // Affiche l'ensemble
    void ShowEns();
    // Retourne le nombre d'éléments
    int GetSize();

private:
    // Ensemble
    bool m_ens[N];
};
#endif

```

ens.cpp

```
// GraphAlgos
// Programme implantant les algos de Floyd, Prim, Dijkstra et Bellman-Ford
// ens.cpp : implantation de la classe CEns
// Date : 08/05/2003
// Auteur: Vouriot Thierry
// http://thierry.vouriot.free.fr
// Date : 08/05/2003

#include "ens.h"

// #####
// Classe CEns
// #####

// Constructeur
CEns::CEns()
{
    for (int j=0; j<N; j++)
        m_ens[j] = false;
}

// Destructeur
CEns::~CEns()
{ }

// Ajoute un élément à l'ensemble
void CEns::AddElem(int i)
{
    m_ens[i] = true;
}

// Supprime un élément de l'ensemble
void CEns::DelElem(int i)
{
    m_ens[i] = false;
}

// Teste si un élément appartient à l'ensemble
bool CEns::IsElem(int i)
{
    return m_ens[i];
}

// Affiche l'ensemble
void CEns::ShowEns()
{
    cout << "{ ";
    for (int i=0; i<N; i++)
        if (m_ens[i]) cout << i << " ";
    cout << "}" << endl;
}

// Retourne le nombre d'éléments
int CEns::GetSize()
{
    int size = 0;
    for (int i=0; i<N; i++)
        if (m_ens[i]) size++;
    return size;
}
```

main.h

```
// GraphAlgos
// Programme implantant les algos de Floyd, Prim, Dijkstra et Bellman-Ford
// main.h
// Date : 08/05/2003
// Auteur: Vouriot Thierry
// http://thierry.vouriot.free.fr
// Date : 08/05/2003

#ifndef _MAIN_H
#define _MAIN_H

// Inclusions standards
#include <stdio>
#include <stdlib>
#include <string>
#include <math.h>
#include <iostream>

using namespace std;

// Windows inclusions
#ifdef _WIN32
#include <windows.h>
#define WIN32_LEAN_AND_MEAN
#endif

// Inclusions persos
#include "graph.h"
#include "ens.h"

// Prototypes

// Algorithme de Floyd
// Cet algorithme résout le problème des plus courtes distances
// de tout sommet du graphe à tout autre sommet, mais ne donne pas
// le chemin correspondant
void floyd(CGraph *graph);

// Algorithme de Prim
// Permet de trouver l'arbre couvrant minimal d'un graphe non orienté
void prim(CGraph *graph);

// Algorithme de Dijkstra
// Permet de trouver les plus courts chemins partant du sommet x à tous
// les autres sommets du graphe (pour un graphe avec valuation positive)
void dijkstra(CGraph *graph, int x);

// Algorithme de Bellman-Ford
// Permet de trouver les plus courts chemins partant du sommet x à tous
// les autres sommets du graphe et permet de détecter des circuits absorbants
void bellman_ford(CGraph *graph, int x);

// Menus
int menu();
int menu2();

// Fonctions creant des graphes utilisés comme exemples
// graphe non orienté (valuation positive) avec circuits pour l'algo de prim,
// dijkstra et floyd
void example1(CGraph *graph);
// graphe orienté (valuation positive et négative) avec circuits pour l'algo de
// floyd
void example2(CGraph *graph);
// graphe orienté (valuation positive et négative) avec un circuit absorbant pour
// l'algo de bellman-ford
```

```

void example3(CGraph *graph);
// graphe non oriente (valuation positive) pour l'algo de prim, dijkstra, bellman-
ford et floyd
void example4(CGraph *graph);
// graphe oriente (valuation positive) pour l'algo de dijkstra, bellman-ford et
floyd
void example5(CGraph *graph);

#endif

```

main.cpp

```

// GraphAlgos
// Programme implantant les algos de Floyd, Prim, Dijkstra et Bellman-Ford
// main.cpp : programme principal contenant les algos
// Date : 08/05/2003
// Auteur: Vouriot Thierry
// http://thierry.vouriot.free.fr
// Date : 08/05/2003

#include "main.h"

// Main
int main()
{
    CGraph *graph = new CGraph();
    int choix, choix2, x;

    cout << "\nProjet Maitrise Informatique 2003\nAuteur: Vouriot
Thierry\nhttp://thierry.vouriot.free.fr\n\nAlgorithmes s'appliquant sur des graphes
simples orientes et values :\n";
    choix = menu();

    while (choix != 7)
    {
        switch (choix)
        {
            case 1:
                cout << "\n";
                graph->SetGraph();
                break;
            case 2:
                cout << "\nGraphe example 1 :\n";
                cout << "non oriente (valuation positive) \navec circuits pour
l'algo de prim, bellman-ford, dijkstra ou floyd";
                example1(graph);
                break;
            case 3:
                cout << "\nGraphe example 2 :\n";
                cout << "oriente (valuation positive et negative) \navec
circuits pour l'algo de floyd ou bellman-ford";
                example2(graph);
                break;
            case 4:
                cout << "\nGraphe example 3 :\n";
                cout << "oriente (valuation positive et negative) \navec circuit
absorbant pour l'algo de bellman-ford";
                example3(graph);
                break;
            case 5:
                cout << "\nGraphe example 4 (vu en TD):\n";
                cout << "non oriente (valuation positive) \npour l'algo de prim,
bellman-ford, dijkstra ou floyd";
                example4(graph);
                break;
            case 6:

```

```

        cout << "\nGraphe example 5 (vu en TD):\n";
        cout << "orienté (valuation positive) \npour l'algo de bellman-
ford, dijkstra ou floyd";
        example5(graph);
        break;
    }

    graph->ShowGraph();

    choix2 = menu2();
    switch (choix2)
    {
    case 1:
        floyd(graph);
        break;
    case 2:
        prim(graph);
        break;
    case 3:
        cout << "\nDonner le sommet de depart de l'algo : ? \n"; cin >> x;
        dijkstra(graph,x);
        break;
    case 4:
        cout << "\nDonner le sommet de depart de l'algo : ? \n"; cin >> x;
        bellman_ford(graph,x);
        break;
    }

    cout << "\nAppuyez sur entree pour continuer ...\n";
    getchar();getchar();
    choix = menu();
}

return 0;
}

// Algorithme de Floyd
// Cet algorithme résout le problème des plus courtes distances
// de tout sommet du graphe à tout autre sommet, mais ne donne pas
// le chemin correspondant
void floyd(CGraph *graph)
{
    CGraph *res = new CGraph();
    res = graph->Copy();

    for (int i=0; i<res->GetSize(); i++)
        for (int j=0; j<res->GetSize(); j++)
            if (res->IsArc(j, i))
                for (int k=0; k<res->GetSize(); k++)
                    if (res->GetArc(j,k) > (res->GetArc(j,i) + res->GetArc(i,k)))
                        if (j!=k) res->SetArc(j,k, (res->GetArc(j,i) + res->GetArc(i,k)));

    cout << "\nAlgorithme de Floyd\nMatrice montrant les plus courtes
distances\nde tout sommet du graphe a tout autre sommet.";
    res->ShowGraph();

    delete res;
}

// Algorithme de Prim
// Permet de trouver l'arbre couvrant minimal d'un graphe non orienté
// La matrice d'adjacences doit donc être symétrique
void prim(CGraph *graph)
{
    int i,j;
    CGraph *res = new CGraph(); // arbre couvrant minimum
    CEns S = CEns();
    res->SetSize(graph->GetSize());

```

```

S.AddElem(0);
cout << endl << "Algorithme de Prim\n" << "S = "; S.ShowEns();

while (!graph->ElemsEqualEns(S))
{
    // Cherche l'arete de poids mini partant de res
    int m=-1,n=-1;
    for (i=0; i<graph->GetSize(); i++)
        for (j=0; j<graph->GetSize(); j++)
        {
            if (!S.IsElem(i) && S.IsElem(j))
            {
                if (m==-1) m = i;
                if (n==-1) n = j;
                if (graph->GetArc(i,j) < graph->GetArc(m,n))
                {
                    m = i; n = j;
                }
            }
        }

    cout << "Arete de poids minimum dont un seul sommet appartient a S :
(" << m << " " << n << ") " << endl;
    S.AddElem(m);
    cout << "S = "; S.ShowEns();
    res->SetArete(m,n,graph->GetArc(m,n));
}

cout << "\nMatrice montrant l'arbre couvrant minimal";
res->ShowGraph();
}

// Algorithme de Dijkstra
// Permet de trouver les plus courts chemins partant du sommet x à tous
// les autres sommets du graphe (pour un graphe avec valuation positive)
void dijkstra(CGrahp *graph, int x)
{
    int y,z,i;
    CEns tmp = CEns();
    CEns C = CEns();
    int *D = new int[graph->GetSize()];
    int *P = new int[graph->GetSize()];

    C.AddElem(x);
    D[x] = 0;
    P[x] = x;

    tmp = graph->Succ(x);
    for (int s=0; s<graph->GetSize(); s++)
        if (s != x)
        {
            if (tmp.IsElem(s)) D[s] = graph->GetArc(x,s);
            else D[s] = INF;
            P[s] = x;
        }

    // -----Affichage-----
    cout << "\nAlgorithme de Dijkstra\n";
    cout << "\nLe tableau suivant donne les distances depuis le sommet de depart
x\nau cours des differentes etapes de l'algo";
    cout << "\nsommets\t";
    for (i=0; i<graph->GetSize(); i++)
        cout << i << " ";
    cout << "\ninit\t";
    for (i=0; i<graph->GetSize(); i++)
        if (D[i] != INF) cout << D[i] << " ";

```



```

        else cout << "I ";
// -----

while (C.GetSize() < graph->GetSize())
{
    y = -1;
    for (i=0; i<graph->GetSize(); i++)
        if (!C.IsElem(i))
        {
            if (y == -1) y = i;
            if (D[i] < D[y]) y = i;
        }
    if (D[y] == INF) break;

    C.AddElem(y);

    for (z=0; z<graph->GetSize(); z++)
        if (!C.IsElem(z))
        {
            if (D[z] > D[y] + graph->GetArc(y,z))
            {
                D[z] = D[y] + graph->GetArc(y,z);
                P[z] = y;
            }
        }
// -----Affichage-----
    cout << "\n\t";
    for (i=0; i<graph->GetSize(); i++)
        if (D[i] != INF) cout << D[i] << " ";
        else cout << "I ";
// -----
}
// -----Affichage-----
cout << endl << endl << "Le tableau suivant contient les peres des
sommets\ndans un plus court chemin depuis le sommet x\n";
for (i=0; i<graph->GetSize(); i++)
    cout << "Predecesseur [" << i <<"] = " << P[i] << "\n";
// -----

delete [] D;
delete [] P;
}

#define MIN(a, b) ((a) < (b)) ? (a) : (b))

// Algorithme de Bellmann-Ford
// Permet de trouver les plus courts chemins partant du sommet x à tous
// les autres sommets du graphe et permet de détecter des circuits absorbants
void bellman_ford(CGraph *graph, int x)
{
    int u,v,k,i;
    int *D = new int[graph->GetSize()];
    int *P = new int[graph->GetSize()];

    bool circuitabsorbant = false;

    for (int i=0; i<graph->GetSize(); i++)
    {
        D[i] = INF;
        P[i] = x;
    }

    D[x] = 0;
    P[x] = x;

    for (k=1; k<graph->GetSize(); k++)
    {
        for (u=0; u<graph->GetSize(); u++)

```

```

        for (v=0; v<graph->GetSize(); v++)
            if (graph->IsArc(u,v))
            {
                if (D[v] > D[u] + graph->GetArc(u,v))
                {
                    D[v] = D[u] + graph->GetArc(u,v);
                    P[v] = u;
                }
            }
    }

    for (u=0; u<graph->GetSize(); u++)
        for (v=0; v<graph->GetSize(); v++)
            if (graph->IsArc(u,v))
            {
                if (D[v] > D[u] + graph->GetArc(u,v))
                {
                    circuitabsorbant = true;
                    break;
                }
            }

    // -----Affichage-----
    cout << "\nAlgorithme de Belman-Ford\n";
    if (!circuitabsorbant) {
        cout << "Le tableau suivant contient les peres des sommets\ndans un
plus court chemin depuis le sommet x\n";
        for (i=0; i<graph->GetSize(); i++)
            cout << "Predecesseur [" << i << "] = " << P[i] << "\n";
    }
    else cout << "Circuit absorbant detecte\n";
    // -----

    delete [] D;
    delete [] P;
}

// Menus
int menu()
{
    int choix;

    cout << "\nGraphe :";
    cout << "\n  [1] Saisir manuellement un graphe";
    cout << "\n  [2] Utiliser graphe d'exemple 1 (Prim, Dijkstra, Bellman-Ford ou
Floyd)";
    cout << "\n  [3] Utiliser graphe d'exemple 2 (Floyd, Bellman-Ford)";
    cout << "\n  [4] Utiliser graphe d'exemple 3 (Bellman-Ford)";
    cout << "\n  [5] Utiliser graphe d'exemple 4 (Prim, Dijkstra, Bellman-Ford ou
Floyd)";
    cout << "\n  [6] Utiliser graphe d'exemple 5 (Dijkstra, Bellman-Ford ou
Floyd)";
    cout << "\n  [7] Quitter le programme";
    cout << "\nChoix : ? ";

    cin >> choix;

    while (choix<=0 || choix>7)
    {
        cout << "\nErreur de saisie, entrez votre choix : ? ";
        cin >> choix;
    }

    return choix;
}

int menu2()
{

```

```

    int choix;

    cout << "\nAlgorithme a appliquer :";
    cout << "\n  [1] Floyd";
    cout << "\n  [2] Prim";
    cout << "\n  [3] Dijkstra";
    cout << "\n  [4] Bellman-Ford";
    cout << "\n  [5] Revenir au menu principal";
    cout << "\nChoix : ? ";

    cin >> choix;

    while (choix<=0 || choix>5)
    {
        cout << "\nErreur de saisie, entrez votre choix : ? ";
        cin >> choix;
    }

    return choix;
}

// Fonctions creant des graphes utilisés comme exemples

// graphe non orienté (valuation positive) avec circuits pour l'algo de prim,
// dijkstra, bellman-ford et floyd
void example1(CGraph *graph)
{
    graph->SetSize(5);

    graph->SetArete(0,1,8);
    graph->SetArete(0,2,2);
    graph->SetArete(0,3,1);

    graph->SetArete(1,3,6);
    graph->SetArete(1,2,5);
    graph->SetArete(1,4,1);

    graph->SetArete(2,4,3);
}

// graphe orienté (valuation positive et négative) avec circuits pour l'algo de
// floyd, bellman-ford
void example2(CGraph *graph)
{
    graph->SetSize(4);

    graph->SetArc(0,1,2);
    graph->SetArc(1,2,-2);
    graph->SetArc(2,3,5);
    graph->SetArc(0,3,6);
    graph->SetArc(3,0,-4);
    graph->SetArc(3,1,-1);
    graph->SetArc(2,1,5);
}

// graphe orienté (valuation positive et négative) avec un circuit absorbant pour
// l'algo de bellman-ford
void example3(CGraph *graph)
{
    graph->SetSize(5);

    graph->SetArc(0,1,3);
    graph->SetArc(0,2,-2);
    graph->SetArc(1,2,1);
    graph->SetArc(2,3,-3);
    graph->SetArc(3,1,1);
    graph->SetArc(2,4,1);
}

// graphe non orienté (valuation positive) pour l'algo de prim, dijkstra, bellman-
// ford et floyd

```

```
void example4(CGraph *graph)
{
    graph->SetSize(9);

    graph->SetArete(0,1,7);
    graph->SetArete(0,2,4);
    graph->SetArete(0,3,1);

    graph->SetArete(1,2,8);
    graph->SetArete(1,4,4);
    graph->SetArete(1,5,2);

    graph->SetArete(2,3,4);
    graph->SetArete(2,4,3);

    graph->SetArete(3,4,3);
    graph->SetArete(3,7,2);

    graph->SetArete(4,5,5);
    graph->SetArete(4,6,1);
    graph->SetArete(4,7,1);

    graph->SetArete(5,8,5);
    graph->SetArete(5,6,8);

    graph->SetArete(6,8,10);
    graph->SetArete(6,7,7);

    graph->SetArete(7,8,3);
}
// graphe oriente (valuation positive) pour l'algo de dijkstra, bellman-ford et
floyd
void example5(CGraph *graph)
{
    graph->SetSize(8);

    graph->SetArc(0,1,6);
    graph->SetArc(0,2,4);
    graph->SetArc(0,3,2);

    graph->SetArc(1,4,2);
    graph->SetArc(1,5,6);

    graph->SetArc(2,3,1);

    graph->SetArc(3,4,5);
    graph->SetArc(3,6,9);

    graph->SetArc(4,5,5);
    graph->SetArc(4,7,10);
    graph->SetArc(4,2,3);

    graph->SetArc(5,7,5);

    graph->SetArc(6,4,4);
    graph->SetArc(6,7,5);
}
```