Implantations et Manipulation de graphes

Projet *LO42*, Printemps 2002

Mélanie Bats - Thomas Petazzoni
GI01

Université de Technologie de Belfort Montbéliard

Responsable de l'UV : C. Renaud

Table des matières

Introduction

L'objectif du projet de *LO42* est de programmer un logiciel permettant de résoudre différents problèmes classiques sur les graphes valués.

Les trois représentations les plus courantes d'un graphe seront implémentées :
- représentation matricielle
- représentation FS et APS
- représentation structures chaînées

Chaque implémentation permettra de modifier des attributs des éléments du graphe, en implantant toutes les méthodes du Type de Données Abstrait *Graphe* défini plus loin.

Une fois ces implantations réalisées, des algorithmes traitant les problèmes courants sur les graphes seront implantés.

Le logiciel permettra de faire une descente en profondeur, en largeur du graphe à partir d'un sommet, d'inverser le graphe, de rechercher le plus court chemin entre deux sommets, de rechercher le chemin critique entre deux points et enfin de rechercher les composantes fortement connexes d'un graphe.

Ce logiciel sera programmé en Java.

1 Le type de données abstrait (TDA)

1.1 Description fonctionnelle

Avant de définir le type de données abstrait *Graphe*, nous définissons les types de données abstraits *Arc* et *Sommet* utilisés par *Graphe*.

1.1.1 Le type Sommet

Sorte Sommet utilise Entier.

`valeur` :	Sommet	→	Entier
`modifier_identifiant` :	Entier x Sommet	→	Sommet
`identifiant` :	Sommet	→	Entier

1.1.2 Le type Arc

Sorte Arc utilise Entier.

`valuation` :	Arc	→	Entier
`modifier_valuation` :	Entier x Arc	→	Arc
`sommet_destinataire` :	Arc	→	Entier
`sommet_source` :	Arc	→	Entier

1.1.3 Le type Graphe

Le TDA Graphe utilise le type Sommet et Arc.
Sorte Graphe utilise Sommet, Arc, Entier, Booléen.

`est_vide` :	Graphe	→	Booléen
`ajouter_sommet` :	Sommet x Graphe	→	Graphe
`supprimer_sommet` :	Sommet x Graphe	→	Graphe
`est_sommet` :	Sommet x Graphe	→	Booléen
`ajouter_arc` :	Sommet x Sommet x Entier x Graphe	→	Graphe
`supprimer_arc` :	Sommet x Sommet x Graphe	→	Graphe
`est_arc` :	Sommet x Sommet x Graphe	→	Booléen
`degre_plus` :	Sommet x Graphe	→	Entier
`degre_moins` :	Sommet x Graphe	→	Entier
`ieme_successeur` :	Sommet x Entier x Graphe	→	Sommet
`ieme_predecesseur` :	Sommet x Entier x Graphe	→	Sommet
`ieme_arc` :	Entier x Graphe	→	Arc

Compléments pour la manipulation :

`premier_successeur` :	Sommet x Graphe	→	Sommet
`successeur_suivant` :	Sommet x Sommet x Graphe	→	Sommet
`cout` :	Sommet x Sommet x Graphe	→	Entier
`trouver_num_sommet` :	Sommet x Graphe	→	Entier
`trouver_sommet` :	Entier x Graphe	→	Sommet
`trouver_sommet_par_ident` :	Entier x Graphe	→	Sommet
`trouver_arc` :	Sommet x Sommet x Graphe	→	Arc
`nombre_de_sommets` :	Graphe	→	Entier

2 Les algorithmes

2.1 Parcours d'un graphe

2.1.1 Parcours en profondeur

Cet algorithme permet d'effectuer le parcours en profondeur d'un graphe G à partir d'un sommet s. Le principe de cet algorithme est de visiter s, puis de réaliser un appel récursif pour chaque voisin de s non encore visité. Un appel pour un sommet x est terminal si tous les voisins de x sont déjà visités. La procédure PROFONDEUR-RECURSIF ci- dessous implémente cet algorithme.

```
procédure PROFONDEUR-RECURSIF(Graphe g, Sommet s)
var : x : Sommet

Début
  VISITER(s)
  pour tout voisin x de s faire
    si x est non visité alors
      VISITER(x)
      PROFONDEUR-RECURSIF(x)
    finsi
  finpour
Fin
```

La procédure VISITER(x) réalise l'opération que l'on souhaite éxécuter pour chaque sommet visité.

2.1.2 Parcours en largeur

Un parcours du graphe G à partir de s est dit en largeur si, à chaque étape, l'arête de liaison x, y choisie est telle que le sommet x soit le premier sommet visité ouvert.

Une implémentation possible de cet algorithme consiste à utiliser une file de sommets comme structure de stockage des sommets. On obtient ainsi l'algorithme qui suit.

```
procédure LARGEUR(Graphe g, Sommet s)
var : file : file
      N : Entier correspond au nombre de sommets
      marque [1..N] tableau de Booléen
```

```
Début
  Initialisation de marque [1..n] = faux
  Enfiler (s) dans file
  marque[s] <- vrai
  tant que (non file_vide)
    s <- Défiler() de file
    pour tout successeurs k de s dans G faire
      si (marque[k] = faux) alors
        marque[k] <- vrai
        Enfiler (k) dans file
      finsi
    finpour
  fintantque
Fin
```

2.2 Inversion de graphe

L'inversion de graphe est une manipulation qui n'a de sens que si le graphe est un graphe orienté. Cette manipulation consiste à inverser tous les sens des arcs.

Pour ce faire, on parcourt la liste des arcs du graphe, et on ajoute dans une liste chaînée temporaire tous les arcs qui ne sont pas des boucles (la destination et la source de l'arc sont identiques).

Ensuite, pour chaque arc de la liste, si il existe un arc inverse (de source et de destination inversées), alors il suffit d'échanger les valuations, sinon on ajoute l'arc inverse, et on supprime l'arc.

```
Procédure INVERSION(Graphe g)
var :
   a1, a2 : Arc
   i,tmp : Entier
   liste : liste chaînée

Début
  Pour tous les arcs a1 du graphe faire
    Si le sommet source de a1 = le sommet destination de a1 alors
      ajouter_en_tete(liste, a1)
    Finsi
  Finpour
```

```
  Tant que non vide(liste) faire
    a1 <- retirer_premier(liste)
    a2 <- trouver l'arc inverse de a1 si il existe

    Si a2 existe alors
      retirer(liste, a2)
      tmp <- valuation de a1
      valuation de a1 <- valuation de a2
      valuation de a2 <- tmp
    Sinon
      ajouter l'arc inverse de a1
      supprimer l'arc a1
    Finsi
  Fintantque
Fin
```

2.3 Recherche de chemin

2.3.1 Recherche du chemin le plus court

Algorithme de Dijkstra
L'algorithme ne calcule pas le plus court chemin pour aller d'un sommet A à un sommet B. Il calcule tous les plus courts chemins partant d'un sommet origine A et permettant d'aller à n'importe quel autre sommet du graphe.

Le résultat de cet algorithme est un ensemble de deux tableaux : *distances* et *parcours*. Chaque *distances[i]* est la longueur du plus court chemin permettant d'aller du sommet origine au sommet i. Le tableau *parcours* permet de trouver le plus court chemin de A à B en donnant pour chaque sommet du chemin son précédent sommet dans le plus court chemin.

Par exemple si *parcours* est égal à { 0, 0, 3, 1 } et qu'on cherche le plus court chemin pour aller du sommet 0 au sommet 2. On cherche parcours[2], qui nous donne 3. On cherche ensuite parcours[3] qui nous donne 1. On cherche parcours[1] qui nous donne 0. Le chemin le plus court pour aller de 0 à 2 est donc 0, 1, 3, 2.

L'algorithme de calcul des tableaux *distances* et *parcours* est le suivant :

```
Procédure DIJKSTRA(Graphe g, Sommet s)
var :
  marque [1..N] tableau de Booléens
  distances [1..N] tableau d'Entiers
  parcours [1..N] tableau de Sommets
```

```
  nb_marque_ok : Entier
  sommetmin : Sommet
  valmin : Entier

Début
  Pour i variant de 0 à nombre de sommets de g faire
    marque[i] <- faux
    distances[i] <- +OO
    parcours[i] <- nul
  Finpour

  marque[s] <- vrai
  parcours[s] <- s
  nb_marque_ok <- 1

  Pour tous les successeurs i de s dans g faire
    distances[i] <- cout(s, i)
    parcours[i]  <- s
  Finpour

  Répéter
    valmin <- +OO - 1
    Pour i variant de 0 à nombre de sommets de g faire
      Si marque[i] = faux alors
        Si distances[i] < valmin alors
          valmin <- distances[i]
          sommetmin <- i
        Finsi
      Finsi
    Finpour

    marque[sommetmin] <- vrai
    nb_marque_ok <- nb_marque_ok + 1

    Pour i variant de 0 à nombre de sommets de g faire
      Si (marque[i] = faux) ET (est_arc(sommetmin, i)) alors
        Si distances[i] > distances[sommetmin] + cout(sommetmin, i)
        alors
          distances[i] <- distances[sommetmin] + cout(sommetmin, i)
          parcours[i]  <- sommetmin
        Finsi
```

```
      Finsi
    Finpour
  Tant que (nb_marque_ok < nombre de sommets de g)
Fin
```

2.3.2 Recherche du chemin critique

Cet algorithme est dérivé de l'algorithme de Dijkstra. Il calcule aussi deux tableaux : *distances* et *parcours* qui ont les mêmes propriétés que pour l'algorithme de recherche du plus court chemin.

Étant donné l'aspect récursif de cet algorithme, il ne fonctionnera pas avec des graphes comportant des cycles ou des circuits.

L'algorithme de recherche du chemin le plus long est le suivant :

```
procédure CHEMIN_CRITIQUE(Graphe g, Sommet z)
var :
  parcours [1..N] tableau de sommets
  distances [1..N] tableau d'entiers
  liste : liste chaînée
  i : Sommet

Début
  pour tous les sommets i de g faire
    parcours[i]  <- nul
    distances[i] <- +OO
  fpour

  distances[s] <- 0
  parcours[s] <- s
  _CHEMIN_CRITIQUE(parcours, distances, g, s, liste)
Fin

procédure _CHEMIN_CRITIQUE(Entier parcours[1..N], Entier
distances[1..N], Graphe g, Sommet s, Liste liste)
var :
  u : Sommet

Début
  ajouter_en_tete(liste, s)
  pour tous les successeurs u de s dans g faire
    si contient(liste, u) = faux alors
```

```
      si (distances[u] = +OO)
         OU (distances[u] < distances[s] + cout(s,u)) alors
        distances[u] = distances[s] + cout (s,u)
        parcours[u] = s
      finsi
      _CHEMIN_CRITIQUE(parcours, distances, g, u, liste)
  finpour
  retirer(liste, s)
Fin
```

2.4 Recherche des composantes fortement connexes

Cet algorithme recherche les composantes connexes dans un graphe non orienté et les composantes fortements connexes dans un graphe orienté.

L'objectif est de déterminer tous les ensembles de sommets (sous graphes) pour lesquels pour toute paire de sommets distincts A et B il existe un chemin pour aller de A vers B.

Notre algorithme est un parcours en profondeur amélioré dans le sens où la recherche ne s'arrete pas a la composante fortement connexe mais s'applique à tout le graphe. Cet algorithme calcule un tableau d'entiers *numero_cfc* qui donne pour chaque sommet le numero de la composante fortement connexe auquel il appartient.

```
Procédure CFC (Graphe g)
var :
  nb_cfc, i, rang, n : Entiers
  numero_cfc [1..] tableau d'Entiers
  rangs [1..] tableau d'Entiers
  theta [1..] tableau d'Entiers
  liste : liste chaînée
  visite [1..] tableau de Booléens
  empile [1..] tableau de Booléens

Début
  Pour i variant de 0 à n faire
    rangs[i] <- -1
    theta[i] <- -1
    numero_cfc[i] <- -1
    visite[i] <- faux
    empile[i] <- faux
  Finpour
```

```
  Pour i variant de 0 à n faire
    Si non visite[i] alors
      nb_cfc <-  cfcDesc(g,i,nb_cfc,rangs,theta,rang,liste,n,visite,empile)
    Finsi
  Finpour
Fin

Fonction CFCDESC(Graphe g, Sommet s, Entier nb_cfc, numero_cfc[]
Entier, theta[] Entier, Entier rang, Liste liste, n Entier, visite[]
Booléen, empile[] Booléen) : Entier
var :
  top : Sommet
Début
  ajouter_en_tete(liste, s)
  empile[s] <- vrai
  visite[s] <- vrai
  rang <- rang + 1
  rangs[s] <- rang
  theta[s] <- rang

  Pour tous les successeurs k de s dans g faire
    Si visite[k] = faux alors
      nb_cfc <- CFCDESC(g,k,nb_cfc,numero_cfc,rangs,
                        theta,rang,liste,n,visite,empile)
      theta[s] <- min(theta[s], theta[k])
    Sinon Si empile[k] = vrai Alors
      theta[s] <- min(theta[s], theta[k])
    Finsi
  Finpour

  Si theta[s] = rangs[s]
      Répéter
        top <- retirer_premier(liste)
        empile[top] <- faux
        numero_cfc[top] <- nb_cfc
      Tant que top <> s

      nb_cfc <- nb_cfc + 1
  Fsi
  Retourne nb_cfc
```

```
Fin
```

3 L'implémentation JAVA

3.1 L'interface

Le type de données abstrait est représenté en Java par une interface, qui sera implémentée trois fois : pour la représentation par matrice d'adjacence, pour la représentation en Fs/Aps et pour la représentation en liste chaînée. Les trois classes qui implémentent cette interface permettent de réaliser des graphes orientés. Les classes permettant de réaliser les graphes non orientés seront détaillées plus loin.

Cette interface est présente dans le fichier `Graphe.java`.

```
interface Graphe
{
    public void    ajouter_sommet(Sommet s);
    public void    supprimer_sommet(Sommet s);
    public void    lister_sommet();
    public void    lister_arc();
    public boolean est_vide();
    public boolean est_sommet(Sommet s);
    public boolean est_arc(Sommet s1, Sommet s2);
    public int     degre_plus(Sommet s);
    public int     degre_moins(Sommet s);
    public void    ajouter_arc(Sommet s1, Sommet s2, int val);
    public void    supprimer_arc(Sommet s1, Sommet s2);
    public Sommet  ieme_successeur(Sommet s, int i);
    public Sommet  ieme_predecesseur(Sommet s, int i);
    public Arc     ieme_arc(int i);
    public Sommet  premier_successeur(Sommet s);
    public Sommet  successeur_suivant(Sommet s1, Sommet s2);
    public int     cout(Sommet s1, Sommet s2);
    public int     trouver_num_sommet(Sommet s);
    public Sommet  trouver_sommet(int i);
    public Sommet  trouver_sommet_par_ident(int ident);
    public Arc     trouver_arc(Sommet src, Sommet Dst);
    public int     getNbSommet();
}
```

3.2 Les implémentations de l'interface

3.2.1 La représentation matricielle

La représentation matricielle est simple : on maintient une matrice carrée (n x n) de références vers des objets Arc. Si la référence ne pointe vers rien, alors il n'y a pas d'arc, sinon elle pointe vers un objet Arc donnant diverses informations quant à l'Arc (valuation, sommet source, sommet destination). Cette matrice est définie par *private Arc[][] matrix* dans la classe.

L'ajout et la suppression de sommets nécessitent donc de redimensionner la matrice, et donc d'effectuer des recopies couteuses en temps. Cette représentation est donc optimale pour les accès (aucun parcours pour trouver si un arc existe entre deux sommets), mais peu optimale en ce qui concerne les insertion/suppression de sommets. En revanche, l'insertion et la suppression d'arc sont des opérations très simples.

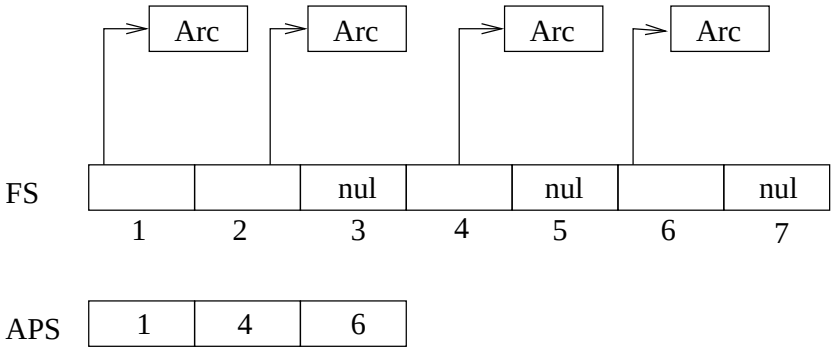
Cette représentation est implémentée par la classe *GrapheMatrix*, dans le fichier `GrapheMatrix.java`. Cette classe implémente toutes les méthodes de l'interface *Graphe*.

3.2.2 La représentation FS-APS

La représentation Fs/Aps consiste à maintenir deux tableaux : un tableaux de références vers des objets de type Arc, représentant la liste des successeurs de chaque sommet et un tableau des adresses des premiers successeurs dans le tableau précédent.

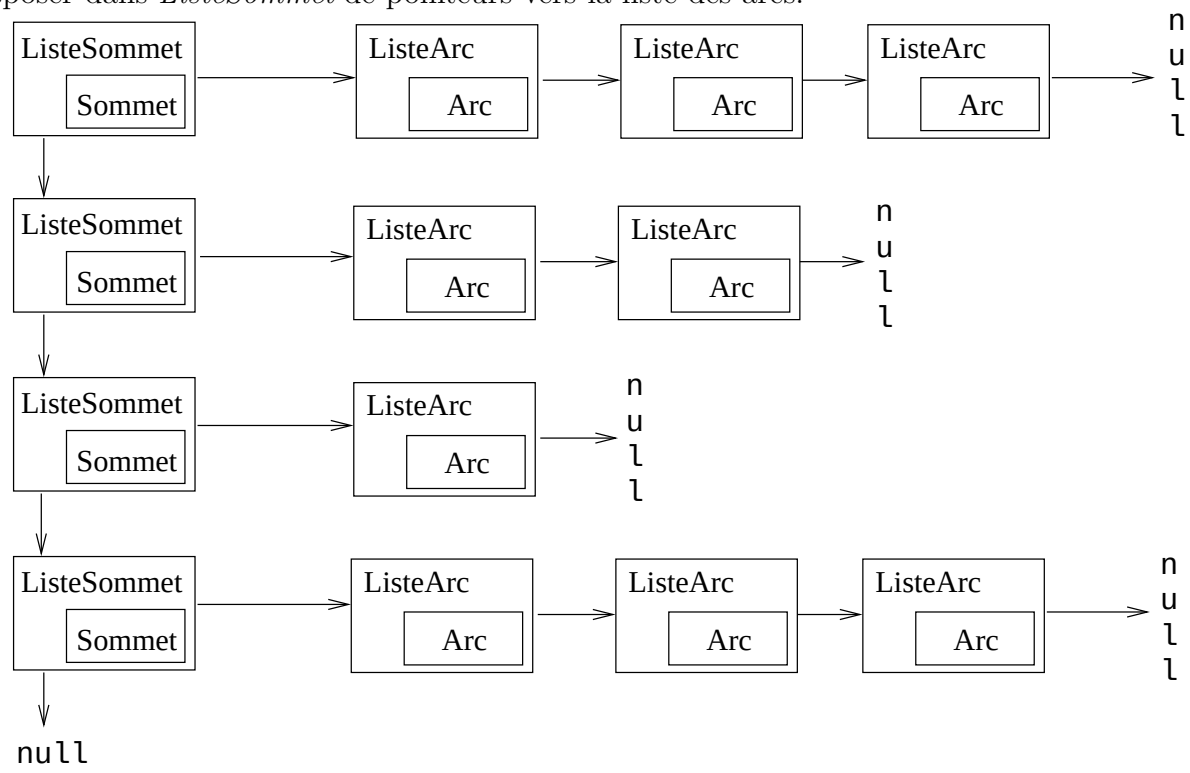
Cette représentation est optimale lorsque l'on souhaite obtenir la liste des successeurs d'un sommet. Par contre la suppression, l'ajout de sommets et d'arcs nécessite des allocations de tableaux et des recopies de tableaux couteuses en temps. D'autre part, la recherche des prédécesseurs est une opération lourde avec cette représentation.

Le tableau des successeurs, *FS* est défini par *private Arc[] fs* et le tableau des premiers successeurs, *APS* est défini par *private int[] aps*.



3.2.3 La représentation structure chaînée

La représentation en liste chaînée est sans aucun doute la plus simple car elle ne nécessite pas de redimensionnement de tableaux. Par contre, on doit définir deux nouvelles classes : *ListeSommet* et *ListeArc*, qui encapsulent respectivement les classes *Sommet* et *Arc*, afin de pouvoir chaîner ces éléments entre eux. Cette encapsalution permet d'ajouter un pointeur suivant et de disposer dans *ListeSommet* de pointeurs vers la liste des arcs.

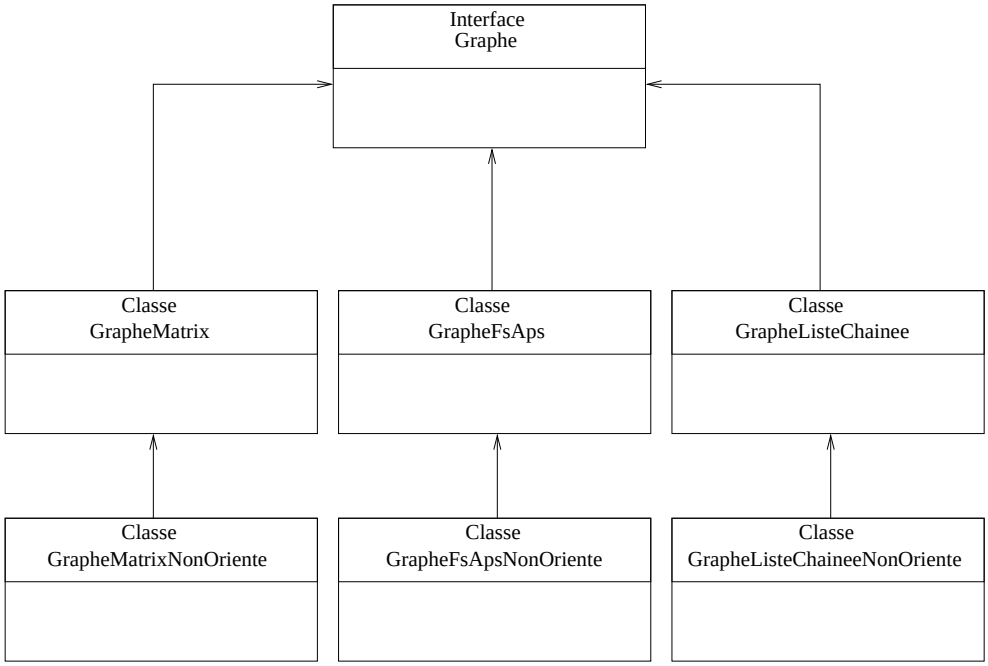


3.2.4 Les graphes non orientés

L'implémentation des graphes non orientés à partir des graphes orientés est une opération simple. Il suffit lorsque l'on ajoute un arc dans un graphe non orienté d'ajouter deux arcs dans le graphe orienté, et de faire de même pour la suppression.

Les graphes non orientés sont donc implémentés dans trois classes : *GrapheMatrixNonOriente*, *GrapheFsApsNonOriente*, *GrapheListeChaineeNonOriente*. Chacune d'entre elle hérite respectivement de *GrapheMatrix*, *GrapheFsAps*, *GrapheListeChainee*. Le constructeur de ces trois classes appelle le constructeur de la classe mère, et on redéfinit les méthodes *ajouter_arc* et *supprimer_arc* pour qu'elles appellent deux fois les méthodes de la classe mère.

3.3 Diagramme des classes



3.4 L'implémentation des algorithmes

Les différents algorithmes sur les graphes ont été implémentées sous forme de méthodes statiques dans la classe *GrapheAlgo*, ce qui permet des les utiliser sans instancier cette classe.

4 L'interface graphique

L'interface graphique a été réalisée avec l'API Swing proposée par le JDK. Pour l'affichage du graphe en lui même, nous avons utilisé la librairie sous license GPL *OpenJGraph*, que l'on peut trouver à l'adresse `http://openjgraph.sourceforge.net`. Cette librairie permet de réaliser de nombreuses opérations avec les graphes. Nous n'avons utilisé que les possibilités de représentation d'un graphe avec ses sommets et ses arcs.

Pour celà, la librairie OpenJGraph défini la classe *GraphScrollPane*, qui hérite de *JScrollPane*, qui permet de représenter un graphe à l'écran.

Nous avons modifié sur quelques points la librairie OpenJGraph pour éliminer les menus contextuels qui ne nous intéressaient pas, pour améliorer le positionnement des textes pour les arcs et pour permettre de redessiner facilement les arcs et les sommets.

Le fichier principal du programme est le fichier `MainWindow.java`, qui définit la classe *MainWindow* qui est la fenetre principale de notre programme. Elle permet via des menus de créer des nouveaux graphes en utilisant les différentes implémentations, d'ajouter/supprimer des sommets, d'ajouter/supprimer/modifier des arcs et d'appliquer différents algorithmes sur le graphe.

Presque toutes ces options requierent l'affichage d'une boite de dialogue. Celles-ci sont implémentées dans tous les fichiers se terminant par `Dialog.java`.

L'interface graphique est bien séparée des algorithmes. L'application du parcours en profondeur nous renvoie une liste chaînée des objets dans l'ordre du parcours, ce qui nous permet de réaliser l'affichage.

5 Améliorations possibles

Nous aurions souhaité pouvoir apporter diverses améliorations à notre travail :

- Supporter le format de fichier GML, qui permettrait de sauvegarder et de charger des graphes à partir de fichiers.
- Trouver un autre algorithme de recherche du plus long chemin qui supporte les graphes avec cycles.
- Réaliser des animations pour le parcours en largeur et le parcours en profondeur.

Conclusion

Le projet de LO42 nous a permis d'appliquer des connaissances acquises en cours en ce qui concerne la représentation des graphes. D'autre part, tous les algorithmes à implémenter n'ayant pas été vu en cours, ce projet nous a permis de nous familiariser avec la compréhension des algorithmes proposés dans les ouvrages spécialisés.

Par ailleurs, ce projet nous a permis de nous initier au langage Java et de comprendre le fonctionnement général des langages orientés objet. D'autre part, l'utilisation et la modification de la librairie OpenJGraph nous a permis de nous familiariser avec la réutilisation de code existant.