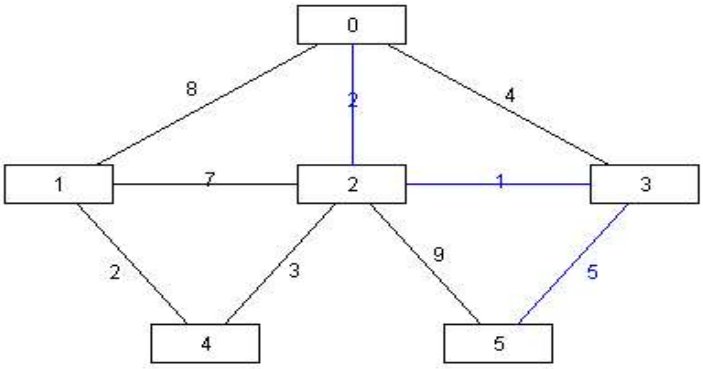
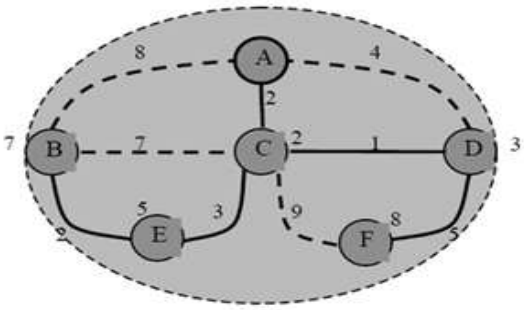
LO42 – TP5
Recherche de chemin de poids minimal
Et utilisation de jgraph





ROLLAND Martin
COUVIDOU Laurent

Introduction

Le but de ce TP (très complet) est la réalisation d'une classe java utilisant jgraph (librairie libre pour gérer des graphes), et permettant le calcul et l'affichage du plus court chemin entre deux sommets, grâce à l'algorithme de Dijkstra.

Nous utilisons un modèle (`DefaultGraphModel`), permettant d'effectuer des opérations vues en cours : *degré+* et *ièmeSucc*, développé lors du TP4. De plus, les distances seront assimilées aux valuations des arêtes (les « poids »).

Ce rapport expose l'algorithme utilisé, la méthode d'intégration dans jgraph, les jeux d'essais utilisés et utilisables et enfin les sources du TP.

Table des matières

1.Algorithme de Dijkstra et jgraph

1.1.L'algorithme en bref

Sans entrer dans les détails (le principe ayant déjà été exposé dans l'énoncé), voici le fonctionnement de l'algorithme de Dijkstra.

Il s'agit de calculer le plus court chemin entre un sommet d'un graphe et tout les autres. Or on ne souhaite obtenir que le chemin entre *deux* sommets ; le résultat obtenu devra donc être ignoré en grande partie.

Pour cela, on attribue à chaque sommet une *étiquette*, qui indique la distance de ce sommet au sommet initial. Elles sont initialisées à l'∞, et modifiées à l'exécution à chaque *relaxation d'arête* (cf. énoncé), c'est-à-dire à chaque fois que l'on traite un des sommets qui leur est contigü.

1.2.Intégration dans jgraph

Par ignorance de la programmation orienté objet, et par soucis de simplification, nous avons choisit de coder les sommets par des numéros et de stocker nos valeurs dans des tableaux.

Ce choix pose problème pour la traduction de l'algorithme en java : en effet il s'est averé difficile de récupérer un objet sommet ou arc à partir de son numéro. Pour résoudre ce problème, il nous a fallu passer par deux tableaux statiques (`sommet` et `arc`, cf. source). Ainsi, le programme fonctionne, mais les principes de la programmation *orientée objet* sont « mis à mal ».

1.3.Affichage du résultat

Nous avons choisit d'afficher trois choses.

En console :
- Les « logs » de l'algorithme : à chaque passage sur un sommet, l'opération effectuée est indiquée ;
- Les valeurs finales des étiquettes (ce qui évitait une opération fastidieuse d'ajout sur le graphe) ;

Sur le graphe :
- Le chemin le plus court calculé en couleur bleue (comme demandé par l'énoncé).

Nota Bene : L'affichage des couleurs semble avoir un petit bug. Pour le voir, il faut parfois cliquer plusieurs fois sur le chemin.

2.Jeu d'essai

2.1.Données

On travaille dans le code source fourni avec 6 sommets, définis comme en page 3 de l'énoncé (avec sommet 1 = sommet A ; sommet 2 = sommet B ; etc.).

Cf. illustration de couverture.

2.2.Résultats

```
Successeur du sommet 0 en cours : 1
   Nouvelle etiquette : 8

Successeur du sommet 0 en cours : 2
   Nouvelle etiquette : 2

Successeur du sommet 0 en cours : 3
   Nouvelle etiquette : 4

Successeur du sommet 2 en cours : 1
   Rien à changer : le chemin est plus long

Successeur du sommet 2 en cours : 4
   Nouvelle etiquette : 5

Successeur du sommet 2 en cours : 5
   Nouvelle etiquette : 11

Successeur du sommet 2 en cours : 3
   Nouvelle etiquette : 3

Successeur du sommet 2 en cours : 0
   Rien à changer : sommet déjà visité

Successeur du sommet 3 en cours : 5
   Nouvelle etiquette : 8

Successeur du sommet 3 en cours : 2
   Rien à changer : sommet déjà visité

Successeur du sommet 3 en cours : 0
   Rien à changer : sommet déjà visité

Successeur du sommet 4 en cours : 1
   Nouvelle etiquette : 7

Successeur du sommet 4 en cours : 2
   Rien à changer : sommet déjà visité

Successeur du sommet 1 en cours : 0
   Rien à changer : sommet déjà visité
(...)
Successeur du sommet 5 en cours : 2
   Rien à changer : sommet déjà visité

Etiquette de 0 : 0
Etiquette de 1 : 7
```

```
Etiquette de 2 : 2
Etiquette de 3 : 3
Etiquette de 4 : 5
Etiquette de 5 : 8
Chemin = [0, 2, 3, 5]
```

On a bien le déroulement de l'énoncé.
Le chemin est affiché correctement (cf. couverture une nouvelle fois).

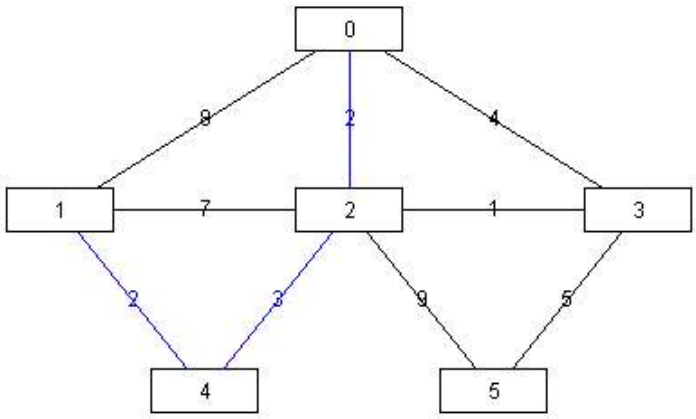
2.3.Autres données

Pour effectuer d'autres essais sur les mêmes données, il suffit de modifier la ligne du main où est appelée la méthode `plusCourtChemin`.

Par exemple, on peut décider d'afficher le chemin des sommets 0 à 1 :

On remplace :
```
// Calcul du plus court chemin
List chemin = graph.plusCourtChemin(0, 1);
```

Résultat



Il est aussi possible d'utiliser d'autres jeux de données, il faut pour cela ajouter des sommets et des arcs avec les méthodes simplifiées fournies.

Nota Bene : ne pas oublier de modifier la variable statique nbSommets.

3.Sources

3.1.Interface du modèle

```
import org.jgraph.graph.GraphModel;
import org.jgraph.graph.*;
import java.util.*;

public interface MonGraphModel extends GraphModel {

        public int degrePlus(DefaultGraphCell c);
        public DefaultGraphCell iemeSucc(int i, DefaultGraphCell c);
        public List listeSucc(DefaultGraphCell c);
        public int ordre();
        // ....

}
```

3.2.Modèle

```
import org.jgraph.graph.DefaultGraphModel;
import org.jgraph.graph.*;
import java.util.*;

public class DefaultMonGraphModel
        extends DefaultGraphModel
        implements MonGraphModel {

        public int degrePlus(DefaultGraphCell c) {
                int i = 0;
                Port p = (Port) getChild(c, 0);
                Iterator it = p.edges();
                while (it.hasNext()) {
                        it.next();
                        i++;
                }
                return i;
        }

        public DefaultGraphCell iemeSucc(int i, DefaultGraphCell c) {
                Port p = (Port) getChild(c, 0);
                DefaultPort p1;
                Iterator it = p.edges();
                while (i > 0 && it.hasNext()) {
                        i--;
                        it.next();
                }
                Edge e = (Edge) (it.next());
                if ((p1 = (DefaultPort) e.getSource()) == p)
                        p1 = (DefaultPort) e.getTarget();
                return (DefaultGraphCell) p1.getParent();
                //return (DefaultGraphCell) getParent(p1);
        }

        public List listeSucc(DefaultGraphCell c) {
                // on suppose qu'il n'y a qu'un port par sommet
                List succs = new ArrayList();
                Port p = (Port) getChild(c, 0);
                DefaultPort p1;
                Iterator it = p.edges();
                while (it.hasNext()) {
                        Edge e = (Edge) (it.next());
                        if ((p1 = (DefaultPort) e.getSource()) == p)
                                p1 = (DefaultPort) e.getTarget();
                        succs.add((DefaultGraphCell) p1.getParent());
                }
                return succs;
        }

        public List listeSom() {
                Vector soms = new Vector(); //ou LinkedList
                Object o[] = DefaultGraphModel.getRoots(this);
```

```
                for (int i = 0; i < o.length - 1; i++)
                        if (o[i].getClass() == DefaultGraphCell.class)
                                soms.add(o[i]);
                return soms;
        }

        public int ordre() {
                Vector soms = new Vector(); //ou LinkedList
                Object o[] = DefaultGraphModel.getRoots(this);
                for (int i = 0; i < o.length - 1; i++)
                        if (o[i].getClass() == DefaultGraphCell.class)
                                soms.add(o[i]);
                return soms.size();
        }

}
```

3.1. Création du graphe

```
/**
 * Importation des bibliothèques
 */

import org.jgraph.JGraph;
import org.jgraph.graph.*;
import javax.swing.JFrame;
import javax.swing.JScrollPane;
import javax.swing.border.Border;
import javax.swing.undo.UndoableEdit;
import java.util.Hashtable;
import java.util.LinkedList;
import java.util.List;
import java.awt.Rectangle;
import java.awt.Color;
import java.util.*;

public class SimpleGraph extends JFrame {

        /**
         * Variables globales
         */
        // Nombre de sommets (attention ! à tenir à jour...)
        static int nbSommets = 6;

        // Tableau des sommets
        static DefaultGraphCell[] sommet = new DefaultGraphCell[nbSommets];

        // Tableau des arêtes
        static DefaultEdge[][] arc = new DefaultEdge[nbSommets][nbSommets];

        // Poids de chaque arête (distance)
        static int[][] poids = new int[nbSommets][nbSommets];

        /**
         * Déclaration du graphe, du modèle et de la map
         */
        JGraph myGraph;
        DefaultMonGraphModel myModel; // On utilise le modèle du TP4
        Map myAttribute;

        /**
         * Création du graphe, du modèle et de la map
         */
        SimpleGraph() {
                myModel = new DefaultMonGraphModel();
                myGraph = new JGraph(myModel);
                myAttribute = new Hashtable();
        }

        /**
         * Création d'un sommet (cellule rectangulaire)
         */
        DefaultGraphCell setVertix(String name, Rectangle bound, Color bgColor,
                        boolean isOpaque, Border border, Color borderColor) {
                DefaultGraphCell myCell = new DefaultGraphCell(name);
                //Map cellAttribute = GraphConstants.createMap();
```

```
        Map cellAttribute = new AttributeMap();
        myAttribute.put(myCell, cellAttribute);
        //cellAttribute.put("valeur",new Integer(2));
        GraphConstants.setBounds(cellAttribute, bound);
        Object[] cells = new Object[] { myCell };
        myModel.insert(cells, myAttribute, null, null, null);

        if (bgColor != null)
                GraphConstants.setBackground(cellAttribute, bgColor);
        if (isOpaque)
                GraphConstants.setOpaque(cellAttribute, isOpaque);
        if (border != null)
                GraphConstants.setBorder(cellAttribute, border);
        if (borderColor != null)
                GraphConstants.setBorderColor(cellAttribute, borderColor);
        return myCell;
}


/**
 * Création d'une arête (entre 2 sommets)
 */
DefaultEdge setEdge(String name, int lineEnd, boolean isEndFill) {
        DefaultEdge myEdge = new DefaultEdge(name);
        //Map edgeAttribute = GraphConstants. createMap();
        Map edgeAttribute = new AttributeMap();
        myAttribute.put(myEdge, edgeAttribute);
        edgeAttribute.put("value", new Integer(4));
        //GraphConstants.setValue(edgeAttribute, new Integer(4));
        GraphConstants.setLineEnd(edgeAttribute, lineEnd);
        GraphConstants.setEndFill(edgeAttribute, isEndFill);
        GraphConstants.setLineColor(edgeAttribute, Color.blue);
        return myEdge;
}

/**
 * Création d'une arête (entre deux cellules) avec valeur et couleur en
 * paramètres
 */
DefaultEdge setEdge(String name, int lineEnd, boolean isEndFill,
                int valeur, Color couleur) {
        DefaultEdge myEdge = new DefaultEdge(name);
        //Map edgeAttribute = GraphConstants. createMap();
        Map edgeAttribute = new AttributeMap();
        myAttribute.put(myEdge, edgeAttribute);
        Integer val = new Integer(valeur);
        edgeAttribute.put("value", val);
        //GraphConstants.setValue(edgeAttribute, new Integer(4));
        GraphConstants.setLineEnd(edgeAttribute, lineEnd);
        GraphConstants.setEndFill(edgeAttribute, isEndFill);
        GraphConstants.setLineColor(edgeAttribute, couleur);
        return myEdge;
}

/**
 * Connexions
 */
void insertModel(Edge edge, Port source, Port target, ParentMap pm,
                UndoableEdit[] e) {
        ConnectionSet cs = new ConnectionSet(edge, source, target);
        Object[] cells = new Object[] { edge };
        myModel.insert(cells, myAttribute, cs, pm, e);
}

/**
 * Affichage du graphe
 */
void showFrame() {
        this.getContentPane().add(new JScrollPane(myGraph));
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        this.pack();
        this.setVisible(true);
}

/**
 * Création simplifiée d'un noeud
 * @param nom
 * @param x
 * @param y
 * @return
 */
DefaultGraphCell creerNoeud(String nom, int x, int y) {
```

```
        //System.out.println("Creation d'un noeud appelé " + nom + "... ");
        DefaultGraphCell cell = setVertix(nom, new Rectangle(x, y, 60, 25),
                        null, false, null, Color.black);
        //System.out.println("OK");
        return cell;
}

/**
 * Création simplifiée d'une arête
 * @param port1
 * @param port2
 * @param s1
 * @param s2
 * @param valeur
 * @param couleur
 * @return
 */
DefaultEdge creerArete(DefaultPort port1, DefaultPort port2, int s1,
                int s2, int valeur, Color couleur) {
        DefaultEdge edge = setEdge("arc" + s1 + "+" + s2,
                        GraphConstants.ARROW_NONE, true, valeur, couleur);
        insertModel(edge, port1, port2, null, null);
        poids[s1][s2] = valeur;
        poids[s2][s1] = valeur;
        arc[s1][s2] = edge;
        arc[s2][s1] = edge;
        return edge;
}

/**
 * Passage en bleu d'une arête
 * Nota : BUG -> cliquer plusieurs fois pour faire apparaître la couleur
 * @param edge
 */
void marquerArete(DefaultEdge edge) {
        Map edgeAttribute = new AttributeMap();
        myAttribute.put(edge, edgeAttribute);
        GraphConstants.setLineColor(edgeAttribute, Color.blue);
        edge.changeAttributes(edgeAttribute);
}

/**
 * Entier -> Sommet
 * @param s
 * @return
 */
DefaultGraphCell donnerSommet(int s) {
        return sommet[s];
}

/**
 * // Sommet -> Entier
 * @param cell
 * @return
 */
int donnerEntier(DefaultGraphCell cell) {
        int som = -1;
        for (int i = 0; i < nbSommets; i++) {
                if (sommet[i] == cell)
                        som = i;
        }
        return som;
}

/**
 * Plus court chemin entre deux sommets
 * On utilise l'algo. de Dijskra (cf. schéma), puis on sélectionne le sommet
 * d'arrivée voulu parmi ceux trouvés à l'arrivée
 * Nota : sur le schéma, en continu : les plus courts sous-chemins
 * @param s1 s2
 * @return chemin
 */
public List plusCourtChemin(int s1, int s2) {

        final int infini = Integer.MAX_VALUE;       // Valeur initiale des étiquettes
        int sommetEnCours = s1;                     // Dernier sommet integré au nuage
        boolean[] sommetVisite = new boolean[nbSommets]; // Tableau indiquant pour chaque
                                                         // sommet s'il a été visité ou non
        int[] etiquette = new int[nbSommets];       // Étiquette ("d(u)") de chaque sommet
        int[] predecesseur = new int[nbSommets];    // Prédesseur de chaque sommet sur le
                                                    // plus court sous-chemin
```

```
        int sommet;                                     // Variable de traitement
        int i;                                          // Idem
        boolean continuer = true;                       // Idem

        // Initialisations
        for (i = 0; i < nbSommets; i++)      etiquette[i] = infini;
        etiquette[s1] = 0;
        for (i = 0; i < nbSommets; i++)      sommetVisite[i] = false;

        // Tant que tout n'a pas été visité
        while (continuer) {

                List liste = myModel.listeSucc(donnerSommet(sommetEnCours));
                Iterator it = liste.iterator();

                // Pour chaque successeur non visité du sommet en cours de traitement
                // -> mise à jour des étiquettes et du prédecesseur
                while (it.hasNext()) {
                        i = Integer.parseInt(it.next().toString());
                        System.out.println("Successeur du sommet "
                                        + sommetEnCours + " en cours : " + i);
                        // On élimine les sommets visités
                        if (!sommetVisite[i]) {
                        // Si le chemin en cours est plus court on met à jour l'étiquette
                                if (etiquette[i] > etiquette[sommetEnCours]
                                                + poids[i][sommetEnCours]) {
                                        etiquette[i] = etiquette[sommetEnCours]
                                                        + poids[i][sommetEnCours];
                                        System.out.println("   Nouvelle etiquette : "
                                                        + etiquette[i]);
                                        // Successeurs des successeurs
                                        // -> mise à jour des prédecesseurs
                                        List liste2 = myModel.listeSucc(donnerSommet(i));
                                        Iterator it2 = liste2.iterator();
                                        int j;
                                        while (it2.hasNext()) {
                                                j = Integer.parseInt(it2.next().toString());
                                                // Le prédecesseur du successeur est
                                                // le sommet en cours
                                                if (j == sommetEnCours)
                                                        predecesseur[i] = j;
                                        }
                                }
                                else System.out.println("   Rien à changer : chemin + long");
                        }
                        else System.out.println("   Rien à changer : sommet déjà visité");
                }

                // Le sommet en cours est visité
                sommetVisite[sommetEnCours] = true;

                // On cherche le sommet de plus petite étiquette
                // C'est le sommet suivant à traiter
                int minEtiquette = infini;
                sommetEnCours = -1;
                for (i = 0; i < nbSommets; i++) {
                        if ((etiquette[i] < minEtiquette) && (sommetVisite[i] == false)) {
                                minEtiquette = etiquette[i];
                                sommetEnCours = i;
                        }
                }
                if (sommetEnCours == -1)
                        continuer = false;
        }

        // Affichage pour info
        for (i = 0; i < nbSommets; i++)
                System.out.println("Etiquette de " + i + " : " + etiquette[i]);

        // On reconstitue le chemin à retourner (s1->s2)
        LinkedList chemin = new LinkedList();
        sommet = s2;
        chemin.add(donnerSommet(sommet));
        while (sommet != s1) {
                sommet = predecesseur[sommet];
                chemin.addFirst(donnerSommet(sommet));
        }
        return chemin;
}

/**
 * Programme principal
```

```
         * @param arg
         */
        public static void main(String[] arg) {

                // *** Création du graphe
                SimpleGraph graph = new SimpleGraph();

                // *** Création des sommets
                // Création de 0
                sommet[0] = graph.creerNoeud("0", 200, 20);
                DefaultPort port0 = new DefaultPort();
                sommet[0].add(port0);
                // Création de 1
                sommet[1] = graph.creerNoeud("1", 40, 120);
                DefaultPort port1 = new DefaultPort();
                sommet[1].add(port1);
                // Création de 2
                sommet[2] = graph.creerNoeud("2", 200, 120);
                DefaultPort port2 = new DefaultPort();
                sommet[2].add(port2);
                // Création de 3
                sommet[3] = graph.creerNoeud("3", 360, 120);
                DefaultPort port3 = new DefaultPort();
                sommet[3].add(port3);
                // Création de 4
                sommet[4] = graph.creerNoeud("4", 120, 220);
                DefaultPort port4 = new DefaultPort();
                sommet[4].add(port4);
                // Création de 5
                sommet[5] = graph.creerNoeud("5", 280, 220);
                DefaultPort port5 = new DefaultPort();
                sommet[5].add(port5);

                // *** Création des arêtes
                graph.creerArete(port0, port1, 0, 1, 8, Color.black);
                graph.creerArete(port0, port2, 0, 2, 2, Color.black);
                graph.creerArete(port0, port3, 0, 3, 4, Color.black);
                graph.creerArete(port1, port2, 1, 2, 7, Color.black);
                graph.creerArete(port2, port3, 2, 3, 1, Color.black);
                graph.creerArete(port1, port4, 1, 4, 2, Color.black);
                graph.creerArete(port2, port4, 2, 4, 3, Color.black);
                graph.creerArete(port2, port5, 2, 5, 9, Color.black);
                graph.creerArete(port3, port5, 3, 5, 5, Color.black);

                // Calcul du plus court chemin
                List chemin = graph.plusCourtChemin(0, 5);

                // Affichage en console pour info
                System.out.println("Chemin = " + chemin);

                // Marquage du plus court chemin
                // Nota : BUG -> cliquer plusieurs fois pour faire apparaître la couleur
                Iterator it = chemin.iterator();
                DefaultGraphCell sommet = (DefaultGraphCell) it.next();
                while (it.hasNext()) {
                        graph.marquerArete(arc[graph.donnerEntier(sommet)][graph
                                        .donnerEntier(sommet = (DefaultGraphCell) it.next())]);
                }

                // Affichage du graphe
                graph.showFrame();
        }

} // Fin de SimpleGraph
```

Conclusion

Ce TP nous a permis de mieux comprendre la théorie des graphes, notamment les opérations de TDA vues en cours (*ièmeSucc* et *degré+*, utilisées indirectement).

L' utilisation de jgraph permet une visualisation des graphes très pratique.

Notre code serait améliorable, par exemple en utilisant plus les notions de programmation orientée objet, ou en permettant des modifications plus souples du graphe.