INF 1573: PROGRAMMATION II

Devoir 2:

SYSTEME DE GUIDAGE DE ROUTES DANS UN RESÉAU

Noms des participants:

- Nemy Johanne A
- Kenmoe Ulrich D
- Laroche Samuel
- Nduhura Credo

Nom du professeur :

Mme Benyahia Iham

- **Spécification de l'interface**

L'interface utilisateur de cette application nécessite un panel qui a plusieurs composantes parmi lesquelles :

- La composante **Démarrer** qui est un JButton permet de lancer le graphe et le début du programme.
- La composante **GenererUnAccident** qui est un JButton permettant de simuler un accident entre deux nœuds sur le garph.
- La composante **GenererUnTraffic** qui est également un JButton permettant de simuler un Traffic entre deux nœuds du graphe.
- La composante **route d'arrivee**, un JLabel qui indique une zone de saisie, un JTextField (**entrer votre point d'arriver**) sur lequel nous allons entrer notre nœud d'arriver.
- La composante **route départ**, un JLabel qui indique une zone de saisie, un JTextField (**entrer votre point de départ**) sur lequel nous allons entrer notre nœud de départ.
- Un JLabel **Départ** pour afficher le nœud de départ saisir.
- UN JLabel **Destination** pour afficher le nœud d'arriver saisir.

- **Décomposition du problème**

Pour la résolution de ce problème nous avons utilisé l'architecture **MVC.** Cette dernière permet d'organiser son code en trois packages à savoir **M** pour modèle, **V** pour vue et **C** pour contrôleur. C'est une façon d'appliquer le principe de séparation des responsabilités, en occurrence celles du traitement de l'information et de sa mise en forme. Ainsi ils seront regroupés en trois package :

- Le package **modèle** contient les classes qui modélisent notre système.
- Le package **vue** qui contient tout ce qui sera visuel qui gère l'interface utilisateur et la simulation sur le graphe.
- Le package **contrôleur** qui gère la classe principale, celle qui contient la main et la classe **ActionEvennement** qui est le vrais contrôleur, c'est elle qui gère la logique de notre application.

Fonctionnement : Toutes les actions exercées sur la vue sont transmises au contrôleur qui les communique au modèle, le modèle à son tour se charge de traiter ses informations et de retransmettre les résultats correspondants au contrôleur. De ce fait le contrôleur étant un intermédiaire entre la modèle8 et la vue, va donc recevoir ces informations et les renvoyer à la vue.

Prenons un exemple pour mieux expliquer le fonctionnement. Après la saisi des points de départ et d'arriver dans l'interface de simulation, un appuie sur le bouton démarré vas déclencher un événement qui sera transmis au contrôleur, plus précisément a la classe contrôleur **ActionEvennement**, ce dernier étant la classe qui implémente l'interface

ActionListenner. Pour permettre cette transmission, les composantes de la vue susceptible de créer des événements comportent des écouteurs d'événements.

Une fois l'événement du bouton démarrer transmis au contrôleur, celui-ci, grâce a sa méthode implémenté **ActionPerfomed**, exécute le code correspondant à cette demande de la vue, cette exécution va engendre une exécution des méthodes correspondantes du package modèle. Le morceau de code suivant montre la partie exécuté suite a un appui sur le bouton démarrer.

```
            } else if (nomButtonClique.equals("DÃ©marrer")  && !nbrClick) {

                  System.out.println("Click sur Demarrer");

                  nbrClick = true;// Empeche le reaffichage du panel en cas
d'un second appuis

                  /**
                   * Instance de la class Voiture, rpresente la voiture qui
va se deplacer
                   */
                  voiture = new Voiture();

                  /**
                   * Creer un Panel utilisateur a l'appuis sur demarrer
                   */
                  if (pointDepart != null && pointArriver != null) {

                        /*
                         * Calculer le chemin le plus court entre le noeud
de depart et le noeud
                         * d'arrive grace a la methode calculCheminCourt de
la classe graphe qui utilise
                         * l'algorithme de Dijkstra. Cette methode stocke
les noeuds correspondant au
                         * chemin le plus court dans l'attribut
NoeudCheminPlusCourt (ArrayList) de sa
                         * classe
                         */

      graphe.calculCheminCourt(graphe.getNoeud(pointDepart),
graphe.getNoeud(pointArriver));// calcul du


            // chemin le


            // plus court

                        trajet = graphe.getTrajet();
                        /*
                         * Stocker les points correspondant au chemin le
plus court calculer dans la
```

```
                             * classe voiture dans l'ArrayList
pointCheminVoiture
                             */

                            ActionEvenement.pointCheminVoiture =
voiture.deplacerPoint(trajet.getListeNoeuds());// deplacement


      // sur le


      // chemin le


      // plus


      // court

                            tempParcours =
voiture.dureeDuTrajet(trajet.getListeNoeuds());

                            System.out.println("Le temps du parcours est : " +
tempParcours);

                            ActionEvenement.iconesDirection =
voiture.directionAPrendreDansCheminPlusCourt(trajet.getListeNoeuds());

                    }

            }
```

Comme on peut le remarquer l'exécution des méthodes des classes du package modèle va traiter un certains nombres d'informations et les retransmettre au contrôleur qui a son tour le transmettra a la vue pour un affichage.

Parmi les méthodes exécuter dans cette exemple, il y a la méthode `graphe.calculCheminCourt(graphe.getNoeud(pointDepart), graphe.getNoeud(pointArriver))` qui se charge de calculer le chemin le plus court et la methode `voiture.dureeDuTrajet(trajet.getListeNoeuds());` qui se charge de deplacer la voiture sur le chemin le plus court.

On part du principe que la voiture doit connaitre le chemin sur lequel elle se déplace.

Une fois ces informations traitées et transmises a la vue, cette dernière se charge de les affiche a l'utilisateur.

- **Conception de la solution**

Le package modèle : il comprend les classes nœud, lien, graphe, trajet, évènements, accident, Traffic, voiture.

- **La classe nœud** c'est la toute première classe à laquelle on a pensé. Elle représente les différents points d'intersection entre les rues dans notre graphe. Elle a 5 variables d'instances tels que :
  - ✓ **Nom** qui représente le nom de chaque nœud de notre graphe.
  - ✓ **Coordonnée** qui est de **point** définis les coordonnes de chaque point représente sur le graphe (nous avons q un point est représenté par son abscisse et son ordonnée)
  - ✓ **Statut** qui définit si le plus court chemin vers le nœud est permanent ou temporel.
  - ✓ **LongueurChemin** qui définit le temps ou la longueur vers un nœud à partir du nœud source.
  - ✓ **Predecesseur** qui définit le dernier nœud dans la séquence du chemin le plus court à partir duquel on est arrivé à ce nœud (important pour connaitre la séquence qui définit le chemin le plus court).

- **La classe lien** représente la distance entre deux nœuds du graphe. Elle comprend les nœuds **a, b** ; le **poid** qui est la distance de chaque lien, un **evenement** de **Evenement.** Elle a une methode **getcongestion** qui retourne un facteur de congestion a la hausse si l'heure courante s approche des heures de pointes. Nous avons défini les heures de haute densité 8h et 17h. la densité suit une fonction cosinus entre 4h et 21h, la densité est constante à l extérieure de ces heures. Plus on s'approche de 8h et 17h, plus le poids (temps) retourne est élevé. Lors des pointes, le facteur maximum est 3x le poids normal.

- **La classe graphe** représente notre graphe qui est compose de nœuds et lien comprend la méthode **calculCheminCourt** qui permet de calculer le chenin le plus court en se basant sur l'algorithme de **Djikstra**. Cette méthode stocke les nœuds correspondant au chemin le plus court dans l'attribut **NoeudCheminPlusCourt** de sa classe.

- **La classe trajet** qui représente le trajet à suivre au cours d'un déplacement sur notre graphe elle est fournie par l'algorithme de Dijkstra.

- **La classe évènement** qui génère un évènement sur notre graphe. L'évènement peut être de deux sortes; soit un Traffic soit un accident.

- **La classe voiture** représente le point qui se déplace sur le trajet. Comprend la méthode **deplacerPoint** qui est chargée de définir le parcours de la voiture en définissant une suite de point représentant le parcours. Ses points seront ensuite transmis a la vue pour l'affichage.

- **Package vue :** il comprend les classes frmPanelSimulation, frmPanelUtilisateur, et PanelUtilsiateur.

    - **La classe frmPanelSimulation** represente l'écran qui permet de saisir des informations permettant de faire la simulation. Comprend des boutons qui servent a generer des evenements (accidents et traffic) et de saisir le point de départ et le point d'arrivée.

    - **La classe frmPanelUtilisateur** permet d'afficher le graphe et ses composants (nœuds et liens) et de montrer le chemin le plus court parcouru par la voiture.

    - **La classe PanelUtilisateur** permet de dessiner le graphe. Elle permet d'afficher toutes les informations relatives au l'execution de l'application.

- **Package contrôleur :** ce package comprend les classes ActionEvenement et Principale.
    - **La classe ActionEvenement** est la classe principale de notre application. Elle représente le contrôleur car elle est l'intermédiaire entre la vue et le modèle. Lorsque le contrôleur reçoit une information et il la communique au modèle qui va la traiter.

    - **La classe Principale** comprend la méthode **main(),** c'est dans cette méthode que le lancement de l'application se déroule. Autrement dit, elle initialise le graphe, les fenêtres de simulation et de l'utilisateur.

- **Implémentation des classes**

  On fournira l'implémentation des algorithmes clé des classes du package Modèle étant donné que toutes les méthodes de calculs se définissent dans ce package.

- Méthode **calculCheminCourt** qui permet de calculer le chemin le plus court en utilisant l'algorithme de Djikstra.

```
/**
         * Méthode qui permet de calculer le chemin le plus court en utilisant
         * l'algorithme de Dijkstra. Cette methode stocke les noeuds correspondant au chemin le plus court
         * dans l'attribut NoeudCheminPlusCourt (ArrayList) de sa classe
         *
         * @param depart le point de depart
         * @param destination le point d'arriver
         */
        public void calculCheminCourt(Noeud depart, Noeud destination) {
                Noeud courantTempo = depart;
                Noeud tempoVisite = null;


                depart.setLongueurChemin(0); // la distance du noeud de depart = 0
                depart.setStatu(true); // son statu est permanent


                /**
                 * On arrete l'algorithme si tout les noeuds du graphe sont a permanent ou si
                 * tout les noeuds temporaires restant on un chemin infini
                 */
                while (true) {
                        /**
                         *
                         */
                        if(arretAlgo1() || arretAlgo2())
                                break;
```

```
                    /**
                     * Visiter tout les noeud adjacent au noeud courant et
change leur labels s'il
                     * le faut
                     */
                    for (Lien lien : courantTempo.getVoisins()) {
                           try {


                                  if
(lien.getNoeudUn().getNom().equals(courantTempo.getNom())) {
                                         if
(!lien.getNoeudDeux().isStatu()) {
                                                tempoVisite =
lien.getNoeudDeux(); // si le 1er noeud de lien = noeud courant on recupere
le

                                         // 2eme noeud si celui ci a un
statu temporaire

       //System.out.println(tempoVisite.toString());
                                         }


                                  } else {
                                         if (!lien.getNoeudUn().isStatu())
                                                tempoVisite =
lien.getNoeudUn(); // sinon on recupere le 1er noeud si celui ci a un statu

                                         // temporaire

       //System.out.println(tempoVisite.toString());
                                  }


                                  /**
                                   * le premier noeud voisin au noeud
courant sera le noeud de reference pour la
                                   * recherche du noeud ayant le plus
petit chemin et sera dans l'attribut
                                   * tempoPlusPetit
                                   */
                                  tempoPlusPetit = tempoVisite;
```

```
                                    /**
                                     * determiner si le noeud adjacent a une
distance plus grand que celle du noeud
                                     * courant + le poid du lien et si il
est temporaire. Si oui, change les labels
                                     * du noeud adjacent.
                                     */
                                    if (courantTempo.getLongueurChemin() +
lien.getPoid() < tempoVisite.getLongueurChemin()
                                                    && !tempoVisite.isStatu())
{

       tempoVisite.setLongueurChemin(courantTempo.getLongueurChemin() +
lien.getPoid());

       tempoVisite.setPredecesseur(courantTempo.getNom());
                                    }


                             } catch (Exception e) {
                                    e.printStackTrace();
                             }


                     } // ------------------fin du for----------------------
--------

                     //System.out.println(courantTempo.toString());

                     /**
                      * chercher le noeud qui a le plus petit chemin parmi
tout les noeuds du graphe
                      * c'est lui qui devient le noeud courant
                      */
                     this.noeudCheminCourt();
                     courantTempo = tempoPlusPetit;
                     courantTempo.setStatu(true);


              } // -------------------------fin du while--------------------
-------
```

```
                /**
                 * On extrait les noeuds appartenant au chemin le plus court
        et on les stocke
                 * dans l'attribut noeudCheminPlusCourt
                 */
                cheminPlusCourt(depart, destination);


            }
```

- Méthode **noeudCheminCourt** qui cherche le nœud qui a le plus petit chemin parmi tous les nœuds du graphe.

```
private void
noeudCheminCourt()
{ // faire la
recherche du plus
petit parmi tout
les noeuds du
graphe
                            for (Entry<String, Noeud> noeud :
                    noeuds.entrySet()) {
                                    if (noeud.getValue().getLongueurChemin()
                    < tempoPlusPetit.getLongueurChemin()
                                                        &&
                    !noeud.getValue().isStatu()) {
                                                tempoPlusPetit = noeud.getValue();
                                    }
                            }
                        }
```

- Méthode **arretAlgo1** qui parcourt tout le tableau contenant les nœuds du graphe et vérifie si tous les nœuds sont permanent c'est à dire on arrête l'algorithme.

```
Private boolean
arretAlgo1() {
                                    boolean res = false;
                                    int nbrElement = 0;
```

```
                        for (Entry<String, Noeud> noeud :
                noeuds.entrySet()) {
                                if (noeud.getValue().isStatu())
                                        nbrElement++;

                        //System.out.println(noeud.getValue().isStatu());
                                }
                                if (nbrElement == noeuds.size())
                                        res = true;
                                //System.out.println("finaly : " + res);
                                return res;
                        }
```

- Méthode récursive **cheminPlusCourt** qui permet d'extraire les nœuds appartenant au chemin le plus court.

```
/**
                 * Methode qui permet d'extraire les noeuds appartenant au chemin le
        plus court
                 *
                 * @param depart point de depart
                 * @param destination point d'arriver
                 */
                private void cheminPlusCourt(Noeud depart, Noeud destination) {
                        Noeud destinationTempo = destination;
                        if
        (destinationTempo.getPredecesseur().equals(depart.getNom())) {
                                noeudCheminPlusCourt.add(destinationTempo);

                        noeudCheminPlusCourt.add(this.getNoeud(depart.getNom()));
                        } else {
                                noeudCheminPlusCourt.add(destinationTempo);
                                destinationTempo =
        this.getNoeud(destination.getPredecesseur());
                                cheminPlusCourt(depart, destinationTempo);// appel
        recursive
                        }
                }
```

- Methode **deplacerPoint** qui permet de déplacer la voiture sur le chemin le plus court

```
public ArrayList<Point> deplacerPoint(ArrayList<Noeud> cheminCourt) {
```

- Methode **dureeDuTrajet** qui permet de calculer la durée du trajet

```
/**
		 * Methode qui calcule la duree du trajet
		 */
		public double dureeDuTrajet(ArrayList<Noeud> cheminCourt) {
			double tempParcourt = 0;
			for(Noeud noeud : cheminCourt) {
				//tempParcourt += noeud.get
			}
			return tempParcourt;

		}

		/**
		 * Methode qui permet de definir les directions a prendre a partir des noeud du
		 * chemin le plus court
		 */
		public ArrayList<String> directionAPrendreDansCheminPlusCourt(ArrayList<Noeud> cheminCourt)
		{
			ArrayList<String> nomIconDirectionAPrendre = new ArrayList<>();


			for (int i = 0; i < cheminCourt.size()-1; i++) {
				if (cheminCourt.get(i).getX() < cheminCourt.get(i+1).getX()) {
				/**
				 * On compare les positions des noeuds appartenant au chemin le plus court
				 * et dependament du resultat on stocke le nom de l'icone de direction correspondante
				 *
				 * Ici le sens de depalcement est de la gauche de l'ecran vers la droite
				 */
```

```
                            if (cheminCourt.get(i).getY() <
cheminCourt.get(i + 1).getY()
                                                    ||
cheminCourt.get(i).getX() > cheminCourt.get(i + 1).getX()) {
                                        /**
                                         * La voiture se deplace dans
le sens inverse a celui des tests de comparaison
                                         * des noeuds
                                         */

        nomIconDirectionAPrendre.add("tournerDroite");


                            } else if (cheminCourt.get(i).getY()
> cheminCourt.get(i + 1).getY()
                                                    ||
cheminCourt.get(i).getX() < cheminCourt.get(i + 1).getX()) {



        nomIconDirectionAPrendre.add("tournerGauche");
                            } else {



        nomIconDirectionAPrendre.add("toutDroit");
                            }
                    }else {
                            /**
                             * On fait la meme chose mais dans
l'autre sens (de la droite de l'ecran vers la gauche)
                             */
                            if (cheminCourt.get(i).getY() >
cheminCourt.get(i + 1).getY()
                                                    ||
cheminCourt.get(i).getX() < cheminCourt.get(i + 1).getX()) {
                                        /**
                                         * La voiture se deplace dans
le sens inverse a celui des tests de comparaison
                                         * des noeuds
                                         */

        nomIconDirectionAPrendre.add("tournerDroite");
```

```
                                } else if (cheminCourt.get(i).getY()
< cheminCourt.get(i + 1).getY()
                                                        ||
cheminCourt.get(i).getX() > cheminCourt.get(i + 1).getX()) {



            nomIconDirectionAPrendre.add("tournerGauche");
                                } else {



            nomIconDirectionAPrendre.add("toutDroit");
                                }
                        }
                }
                return nomIconDirectionAPrendre;
        }
```

- La methode **getCongestion** permet de simuler la congestion par rapport à l'heure courant

```
public double
getCongestion(){
                                /**
                                 * La méthode retourne un facteur de congestion à
                            la hausse si l'heure courante s'approche
                                 * des heures de pointes. Les heures de haute
                            densité sont 8h et 17h. La
                                 * densité suit une fonction cosinus entre 4h et
                            21h, la densité est
                                 * constante à l'exterieur de ces heures. Plus on
                            s'approche de 8h et
                                 * 17h, plus le poid (temps) retourné est élevé.
                            Lors des pointes, le
                                 * facteur maximum est 3x le poid normal.
                                 *
                                 * Si un évènement (traffic ou accident) affecte
                            la circulation, le facteur
```

```
                 * est haussé.
                 *
                 * @author Samuel
                 */
                double congestion = 1; //Pour s'assurer que 1 est
retourne si jamais il y a une erreur avec la formule

                float time = (float) LocalTime.now().getHour() +
(float) LocalTime.now().getMinute() / 60;

                if (7.368 < time && time < 8.625) {
                        // poid = poid départ (distance) * facteur
heures de pointes *
                        // facteur evenement
                        congestion = (Math.cos(time / 0.2 + 4) +
2);
                } else if (15.536 < time && time < 18.05){
                        congestion = (Math.cos(time / 0.4 + 2) +
2);
                }
                if (evenement == null) {
                        return congestion;
                } else {
                        return congestion *
evenement.getCongestion();
                }
        }
```

- Test de qualité

Qualités recherchées	Observations	Corrections a apporter
La réutilisation	Il est réutilisable a cause du patron de conception MVC qui permet de grouper les classes ayant les memes rôles dans un même package. Donc, par exemple si nous voulons recréer notre système, nous avons juste à réutiliser le package Modèle qui	Revoir l'implémentation des différentes méthodes

	représente le comportement de notre système.	
La robustesse	Il n'est pas assez robuste	Gérer les exceptions et revoir la conception du système.
La portabilité	Il est utilisable partout (nous l'avons testé sur mac, et Windows)	Pas d'améliorations étant donne qu'il s'adapte a différents environnements

- Librairie JDK

1. Interface utilisateur
   - JFrame
   - JLabel
   - JPanel
2. Interface simulation
   - JFrame
   - JLabel
   - JButton
   - JTextfield

Nom de la classe	**Rôle selon l'API de java**	**Méthodes associées**	**Descriptions dans le contexte du code fourni**
JButton	Une implémentation d'un bouton pour cliquer.	JButton	Le constructeur de la classe avec le paramètre pour le contenu du bouton.
JPanel	C'est un conteneur léger. La méthode *PaintComponent* permet d'ajouter des composantes au conteneur.	JPanel PaintComponent	Le constructeur par défaut. Cette methode permet de dessiner sur le JPanel
JLabel	La méthode *getText* une composante légère qui permet l'édition d'une ligne de texte.	getText	Permet de recuperer le texte affiche sur le JLabel

JFrame	C'est une version étendue de *awt Frame* pour supporter des composantes *swing*. La méthode *add* permet d'ajouter des composantes dans la fenêtre.	add	Pour ajouter le panel et la boite de texte au Frame.
JTextfield	Représente la saisie de donnees au clavier	addActionListener	Pour ajouter un ecouteur d'evements
Graphics	La methode set color permet de definir la couleur actuelle de ce contexte graphique a la couleur specifiee La methode set font permet de definir la police d'un graphique sur la police specifiee La methode filloval permet de remplir un oval delimite par le rectangle specifie avec la couleur courante La methode drawstring permet quant a elle de dessiner un texte specifique en utlisant la couleur et police actuelles du graphique	Set color Set font Filloval Drawstring	On attribue la couleur rouge On attribue une police specifique de 19 Pour remplir l'oval specifie On dessine les differents points du graphique en utlisant les couleurs et polices definies

Note : Pour l'utilisation de l'application, voir le manuel d'utilisation dans le dossier artefacts.

Conclusion

Ce devoir a été une occasion pour nous d'approfondir les notions appris dans le cours de programmation 2, et d'apprendre d'autres notions à travers des recherches sur la programmation oriente objet en java et sur la conception et l'utilisation des patrons de conceptions. Par les sites internet ou forum que nous a vont utiliser, se trouver le forum sur la programmation **stack over flow** et les vidéos sur le site **YouTube**.

Parmi les points réussis de ce devoir, on peut citer l'implémentation de l'algorithme de calcul du chemin le plus court, le déplacement du pointeur sur le chemin le plus court calculer et la gestion des interfaces graphique. On peut aussi citer la gestion du graphe dans son ensemble et de ses composantes (liens, nœuds) et les associations entre elles. Nous sommes satisfaits de la maitrise de la complexité de notre application qui n'as pas été facile au début mais qu'on pu maitrise au fur du temps notamment l'implémentation des algorithme et la communication (envoie des messages) entre différentes partie (classes) qui compose notre application.

Parmi les points les moins réussi qui demande plus de travailles et d'amélioration, il y a la gestion des événements que nous n'avons pas pu finir a implémentée dans les temps, en effet celle-ci demande des modifications significatives sur l'ensemble de l'application et nous avions pas assez de temps pour apporter ses modifications vu le délai de dépôt du devoir. La complexité de cette gestion vient du fait que les évènements doivent être gérer en temps réel.

Nous vous remercions de cette occasion que vous nous avez offert de travailler sur un projet aussi complexe et aussi complet qui nous a appris beaucoup de notions sur la programmation oriente objet basée sur les évènements.