**Département d’informatique et d’ingénierie**

**Université du Québec en Outaouais**

**INF1573**

**Programmation II**

**Hiver 2015**

**Devoir II**

Diaw Cheikh Charles

Kossi Ahlin Cornelus MADJRI

Ngaido Mamadou Belly

**Travail présenté à**

Ilham Benyahia

*16-04-2015*

Table des matières

[Phase 1 : Analyse et conception des interfaces graphiques 3](#_Toc416986104)

[Fonctionnement du système à développer : 3](#_Toc416986105)

[Conception des interfaces 3](#_Toc416986106)

[Interface Principale 4](#_Toc416986107)

[Présentation des différentes composantes utilisées 5](#_Toc416986108)

[Phase 2 : Analyse et conception de la solution globale 6](#_Toc416986109)

[Rappel du fonctionnement du système à développer : 6](#_Toc416986110)

[Conception globale de la solution 6](#_Toc416986111)

[**Décomposition du problème :** 6](#_Toc416986112)

[**Conception de la solution** 7](#_Toc416986113)

[Classes identifiées : 9](#_Toc416986114)

[Phase 3 : Codage de la solution 10](#_Toc416986115)

[Phase 4 : Documentation du fonctionnement du système et analyse de sa qualité 13](#_Toc416986116)

[Analyse de la qualité 19](#_Toc416986117)

[Références 22](#_Toc416986118)

# Phase 1 : Analyse et conception des interfaces graphiques

## Fonctionnement du système à développer :

Ce système doit considérer un réseau routier et un écran GPS. A l’initialisation le réseau entier est affiché. L’utilisateur peut entrer un itinéraire en choisissant un point de départ et un point d’arrivée parmi les intersections du réseau (sommets). Ensuite, un calcul du chemin le plus court pour l’itinéraire (en tenant compte des contraintes) est effectué et ce chemin est identifiable (colorié) sur le réseau affiché. De même, les contraintes (routes impraticables) sont identifiables.

L’emplacement de l’utilisateur (sur son itinéraire) qui change en fonction du temps est aussi identifié.

Pendant que l’utilisateur se rend vers le point d’arrivée, des évènements tels qu’un accident ou une route congestionnée peuvent survenir et affecter le chemin le plus court. Ces évènements sont aussi identifiables sur le réseau (on peut voir visuellement qu’une route est impraticable).

## Conception des interfaces

Il y’a une seule interface, l’interface principale de l’application qui est une fenêtre constituée de plusieurs autres composantes :

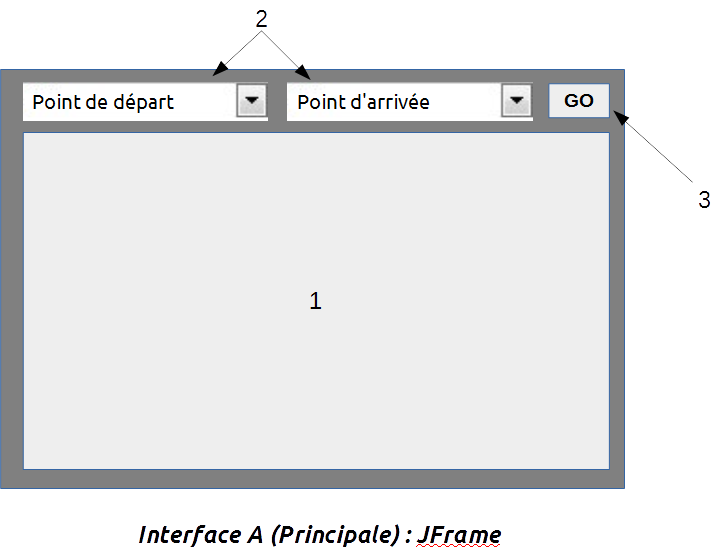
* Deux listes déroulantes composées des sommets du graphe : pour définir un itinéraire l’utilisateur devra choisir un sommet comme point de départ dans la première liste et un autre sommet comme point d’arrivée dans la deuxième.
* Un bouton GO qui déclenche la navigation : lorsqu’il sera cliqué on pourra observer sur le réseau (un panneau dans la fenêtre) le chemin le plus court colorié qui sera colorié, les nœuds de départ et d’arrivée, ainsi que la position actuelle de l’utilisateur qui va évoluer en fonction du temps et passer du nœud de départ au nœud d’arrivée.

Nous avons décidé d’opter vers le package Swing pour l’interface pour les raisons suivantes :

* Swing est plus récent qu’AWT, par conséquent plus optimisé et plus rapide.
* L’utilisation de composantes légères augmente la portabilité de l’application (les composantes ne dépendent que très légèrement du système d’exploitation utilisé)
* Préférence personnelle : les membres de l’équipe ont plus d’expérience avec ce package.

Les couleurs utilisées sont susceptibles de changer, de même que la taille de chaque composante.

## Interface Principale



1. Représentation du réseau routier et du chemin le plus court : JPanel
2. Listes déroulantes pour choisir l’itinéraire : JComboBox
3. Bouton pour lancer/réinitialiser la recherche du chemin le plus court : JButton

## Présentation des différentes composantes utilisées

|  |  |  |  |
| --- | --- | --- | --- |
| Nom de la composante (Classe) | Rôle selon l’API de java | Méthodes associées | Descriptions dans le contexte du code fourni |
| BorderLayout | Le BorderLayout sépare un container en cinq zones: NORTH,  SOUTH, EAST, WEST et CENTER | BorderLayout( )  BorderLayout (int hgap, int vgap) | Permet de séparer la fenêtre de l’interface principale en plusieurs zones |
| JFrame | Une JFrame est une fenêtre avec un titre et une bordure | public **JFrame** ();  public **JFrame** (String name);  **setTitle** (String titre) **setResizable** (boolean)  String **setVisible** (boolean) | L’interface de l’application est une fenêtre. |
| JButton | Crée un bouton que l’utilisateur peut cliquer pour déclencher une action. | public **JButton** ([String](http://java.sun.com/j2se/1.4.2/docs/api/java/lang/String.html) text)  public **JButton** ([Icon](http://java.sun.com/j2se/1.4.2/docs/api/javax/swing/Icon.html) icon)  **paramString**()  public **setEnabled** (boolean) | Le bouton « GO » sera un objet de cette classe. |
| JPanel | La classe JPanel est un conteneur utilisé pour regrouper et organiser des composants grâce à un gestionnaire de présentation (layout manager) | public **JPanel**();  public Component **add**(Component comp);  public void **setLayout**(LayoutManager lm); | Il nous permet de regrouper tous les composants de la fenêtre.  Le réseau routier est aussi un panneau à part entière. |
| JComboBox | Combine un bouton et une liste déroulante d’éléments. L’utilisateur peut choisir un élément dans la liste qui s’affiche lorsqu’il clique sur le bouton. | public **JComboBox**(String s);  public **JComboBox**(E[] items)  public void **addItem**(E item);  public void **removeItem**(Object object);  public int **getColumns** ( );  public void **setEditable** (boolean) | Représente les deux listes déroulantes utilisées pour permettre à l’utilisateur de choisir un itinéraire. |

# Phase 2 : Analyse et conception de la solution globale

## Rappel du fonctionnement du système à développer :

Voir Phase 1 : Fonctionnement du système à développer:

## Conception globale de la solution

### **Décomposition du problème :**

Partie 1 : Définition du réseau routier (graphe)

Le réseau routier est représenté par un graphe : les sommets sont les intersections et les routes sont les arcs. Le poids des arcs représente la longueur de la route, équivalent au temps requis pour traverser la route.

Un graphe G est composé d’un ensemble de sommets V et d’un ensemble d’arcs E, un arc étant un couple d’éléments de V.

Partie 2 : Récupérer les inputs du système : Point de départ, d’arrivée.

Le système prend comme input minimal un itinéraire de navigation : point de départ et point d’arrivée. Étant donné que nous avons un graphe, ces points représentent des sommets dans notre graphe.

Partie 3 : Calculer le chemin le plus court

Connaissant le graphe du réseau routier, le système devra calculer le chemin le plus court entre les deux sommets fournis en entrée en utilisant l’algorithme de Dijkstra.

Partie 4 : Afficher le résultat (chemin le plus court, routes à ignorer)

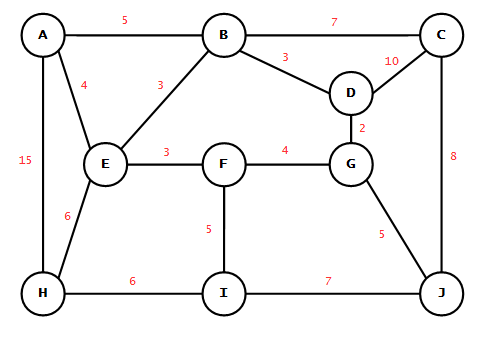
Il faut ensuite afficher le résultat du traitement à l’utilisateur :

* Afficher le réseau routier
* Mettre en évidence le chemin le plus court
* Mettre en évidence le nœud sur lequel on se trouve.
* Mettre en évidence les congestions aléatoires et le changement de chemin conséquent.

### **Conception de la solution**

Définition du réseau :

On considère pour notre application le graphe par défaut suivant (l’application pourra fonctionner avec un autre graphe qu’il faut définir avant exécution) :



* Les nœuds ont tous une étiquette (le nom du nœud) et chaque nœud a sa liste d’adjacence qui est une liste des arcs partant de ce nœud.
* Les arcs sont caractérisés par le nœud vers lequel ils pointent, ainsi que leur poids.

Calcul du chemin le plus court

Le calcul du chemin le plus court sera effectué en utilisant l’algorithme de Dijkstra [1].

Algorithme de Dijkstra

Le principe est de construire à partir du graphe initial un sous-graphe dans lequel les différents sommets sont classés par ordre croissant de leur distance minimale par rapport au sommet de départ. La distance correspond à la somme des poids des arêtes empruntées.

Le sous graphe est construit comme tel : Au début la distance du nœud de départ (par rapport à lui-même) est 0 et celle de tous les autres nœuds est à l’infini. Ce nœud est ajouté au sous-graphe. A chaque itération, on met à jour les distances des sommets reliés au dernier sommet du sous-graphe (la distance est mise à jour seulement si la nouvelle distance qui est égale à la distance du sommet de départ + le poids de l’arc est inférieur à l’ancienne). Ensuite on ajoute au sous-graphe le nœud qui n’en fait pas partie et dont la distance est minimale.

Solution

Notre résolution du chemin le plus court avec l’algorithme de Dijkstra est basée sur la réutilisation d’un code disponible en ligne [2]. En considérant des nœuds et arcs donnés, on peut établir le sous-graphe dans lequel tous les sommets ont la distance minimale pour y arriver en partant du sommet de départ. Par conséquent on peut établir le chemin le plus court entre deux sommets donnés.

Pour un plus court temps d’exécution un tas comme une file de priorités sera utilisé pour trouver rapidement le prochain sommet dont la distance est minimale.

Un tas est une structure de donnée qui permet de retrouver rapidement l’élément le plus petit (ou le plus grand) car vérifiant certaines propriétés [3].

Afficher le réseau routier

Le réseau doit être peint dans un panneau. Le réseau sera donc défini comme une composante graphique en héritant de la classe JPanel du package swing. Dans la redéfinition de la méthode *paintComponent* ( ) les différents nœuds et arcs du graphe seront dessinés. Le réseau aura donc besoin de :

* La liste des nœuds et arcs composant le graphe.
* Les coordonnées des différents nœuds.

L’affichage du réseau (la méthode *paint*) se fera de la façon suivante :

Pour i allant de 0 à liste\_noeuds.length :

Nœud n = liste\_noeuds (i);

dessiner tous les arcs partant de n; // tient compte des couleurs.

dessiner un cercle aux coordonnées de n; // coord\_x[i] et coord\_y[i]

écrire le nom de n au centre du cercle;

Sachant que plusieurs informations supplémentaires devront être affichées sur le réseau (chemin le plus court, emplacement du véhicule, congestions), on peut déjà établir que les nœuds et arcs doivent avoir un attribut supplémentaire : la couleur.

Récupérer les inputs du système : Point de départ, d’arrivée.

Le graphe étant déjà défini au moment de l’exécution, la liste des sommets est connue. L’utilisateur devra donc choisir les points de départ et d’arrivée de son itinéraire dans une liste déroulante de sommets qui lui sera fournie. La classe JComboBox de l’API de Java fera l’affaire. Après avoir choisi les points il faudra appuyer sur le bouton qui déclenche le calcul du chemin et la navigation. Ce bouton sera un objet de la classe JButton. Il faut gérer le cas où l’utilisateur choisit le même sommet dans les deux listes : lorsqu’il appuie sur le bouton une boite de dialogue s’affiche pour lui indiquer qu’il est déjà au point d’arrivée.

Gestion contraintes

Au cours de la navigation des évènements tels qu’un accident ou une congestion peuvent survenir. Le chemin doit être recalculé en tenant compte de ces contraintes et les arcs concernés identifiés (coloriés) sur le graphe.

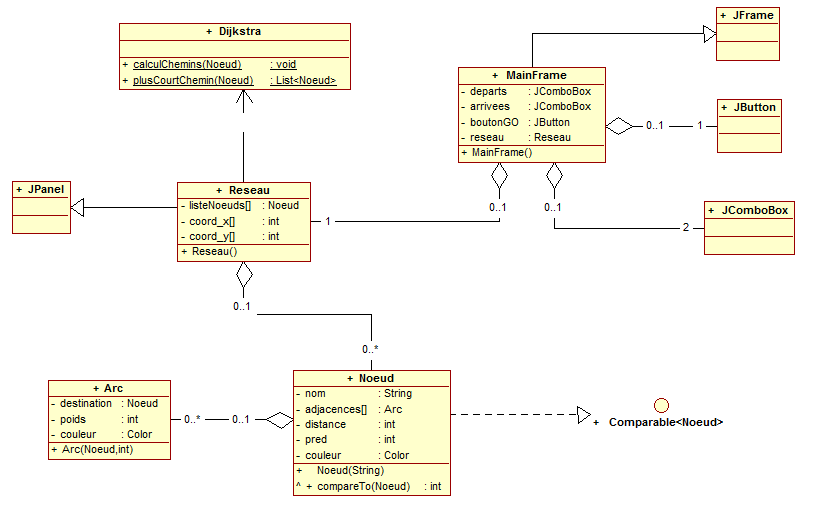
Aléatoirement, le programme principal déclenche une congestion sur une route (un arc). Cela équivaut à affecter une valeur très grande au poids de l’arc, modifier la couleur de l’arc en rouge par exemple (pour le rendre identifiable sur l’affichage), et calculer le chemin le plus court entre la position actuelle et le point d’arrivée. Ce chemin ne passera pas par la route bloquée du fait de son poids très grand.

Pour gérer l’aléatoire, considérons une liste des routes du réseau (liste d’arcs) implémentée avec la classe ArrayList pour que sa taille ne soit pas fixe : A l’initialisation on met tous les arcs du graphe dans la liste et on sauvegarde le nombre d’arcs dans une variable nombreArcs. On ajoute ensuite l’objet null correspondant à un non-évènement un certain nombre de fois. Ainsi la liste d’arcs sera composée des arcs du graphe de l’indice 0 à l’indice nombreArcs – 1 puis d’objets null (ou non-évenèments) de nombreArcs à la fin. A chaque étape de la navigation (chaque fois qu’on arrive sur un nœud), on utilise la fonction *random* ( ) sur la taille de la liste pour obtenir un entier (index). Si l’index est inférieur à nombreArcs on déclenche une congestion à l’arc correspondant, sinon c’est un non-évènement : on continue la navigation.

## Classes identifiées :

* MainFrame extends JFrame : Fenêtre de l’application, contient les différents composants. Contient 2 JComboBox, un JButton et un Réseau.
* JComboBox : Les deux listes déroulantes pour choisir l’itinéraire.
* JButton : Bouton « GO » pour lancer la navigation.
* Réseau extends JPanel : Panneau sur lequel on affiche le graphe. Ses attributs sont : Nœud [ ] liste\_noeuds, int [ ] coord\_x, int [ ] coord\_y.
* Nœud implements Comparable<Nœud> : String nom, Arc [ ] adjacences, int distance, Noeud pred (prédécesseur), Color couleur. Méthodes : int *compareTo* (Nœud autre). Constructeur : Nœud (String nom).
* Arc : Nœud destination, int poids. Constructeur : Arc (Nœud dest, int poids).
* Dijkstra : Méthodes : void calculChemins (Nœud depart), List<Nœud> plusCourtChemin (Nœud arrivee).

**Diagramme UML (à compléter)**



# Phase 3 : Codage de la solution

Voir aussi fichiers source dans l’archive du projet.

L’implémentation a été effectuée à partir de la conception de la solution décrite en phase 2. En complément des commentaires du code, on peut relever les détails d’implémentation suivants :

* Un arc est défini avec le nœud d’arrivée et le poids. Sa couleur par défaut est NOIR.
* Un nœud est défini avec un nom, sa distance par défaut est maximale et il n’a pas de prédécesseur. Sa couleur par défaut est BLANC, et sa liste d’adjacences nulle. Après l’initialisation il faut créer une liste d’arcs partant de ce nœud et affecter cette liste au nœud (*setAdjacences(Liste)*).
* Considérons deux nœuds A et B reliés par un arc. S’il y’a dans la liste d’adjacences de A un arc X de destination B et de poids 10, il doit forcément exister dans la liste d’adjacences de B un arc Y de destination A et de même poids 10. Les arcs X et Y sont dits *réciproques*.
* Les classes Arc et Nœud ont toutes leurs variables d’instance déclarées privées, et disposent par conséquent d’accesseurs et de mutateurs.
* La classe Nœud implémente l’interface Comparable<Nœud>. Cette interface définit comment deux objets de la classe Nœud sont comparés, classés, en se basant sur la méthode *compareTo(Nœud)* qu’il faut implémenter. Dans notre cas l’attribut distance des nœuds sera comparé. Il est nécessaire de définir la manière de comparer deux nœuds car la file de priorités utilisée pour l’algorithme de Dijkstra en a besoin.
* La classe Dijkstra est réutilisée [2] et a été adaptée pour être conforme au reste du code. On peut relever que la classe *PriorityQueue<Nœud>* de l’API sert de file à priorités et garde en tout temps une liste ordonnée des nœuds (ordre défini par l’interface Comparable), ce qui permet de retrouver rapidement le nœud de distance minimale.
* Un réseau est un panneau (dérivant de JPanel) servant à l’affichage du réseau. Il dispose comme attributs d’une liste de Nœuds composant le réseau, deux listes d’entiers contenant les coordonnées en X et Y des nœuds du réseau. Les 3 listes ont la même taille et les indices correspondent. Ainsi si on considère un nœud d’indice i dans la liste de nœuds, ses coordonnées X et Y dans le panneau seront dans les listes *coord\_x* et *coord\_y* à l’indice i.
* Toutes les couleurs utilisées sont définies avec la classe Color de l’API.

**La méthode paintComponent de la classe Réseau**

Conformément à l’algorithme décrit dans la conception, le dessin du graphe se fera de la façon suivante :

On parcourt la liste des nœuds du réseau (*this.listeNoeuds*) avec une boucle for pour garder le contrôle sur l’indice i :

* On récupère les coordonnées du nœud en cours qui se trouvent à l’indice i des tableaux *coord\_x* et *coord\_y* du réseau et on les sauvegarde dans des variables x et y.
* On initialise des variables x2, y2 et index2 qui représentent respectivement les coordonnées et l’indice dans *listeNoeuds* du nœud d’arrivée d’un arc à trouver.
* On récupère dans une variable n de type Nœud le nœud courant.
* Pour dessiner tous les arcs partant du nœud, on parcourt la liste d’adjacences de n, et pour chaque arc a de cette liste :
  + On met à jour les variables x2, y2 et index2. Pour cela on parcourt la liste des nœuds et on compare à chaque itération le nœud de la liste au nœud de destination de a.
  + On dessine l’arc seulement si index2 > i. Cela permet de dessiner l’arc une seule fois, sachant que si i > index2 l’arc courant sera en réalité le réciproque d’un arc déjà dessiné.
  + Pour dessiner l’arc a on dessine une ligne de couleur *a.getColor()* ayant comme coordonnées de départ x + 15 et y +15 (centre du nœud départ) et comme coordonnées d’arrivée x2 +15 et y2 + 15 (centre du nœud d’arrivée). La méthode correspondante est *drawLine* de la classe Graphics. Les lignes sont dessinées avec une bordure personnalisée.
* Après avoir dessiné tous les arcs sortant du nœud, on dessine un cercle de couleur *n.getColor()* et de rayon 15 aux coordonnées x et y. La méthode utilisée est *fillOval* de la classe Graphics qui dessine un cercle plein et cache donc la partie des arcs sortant se trouvant dans le cercle.
* On écrit en NOIR au centre du cercle le nom du nœud (*n.getNom()*) avec la méthode *drawString* la classe Graphics. La police est définie au début de la méthode paintComponent ().
* On dessine le contour du nœud, un cercle NOIR avec la méthode *drawOval* aux coordonnées x et y. Cette méthode dessine un cercle vide et ne cache donc pas le nom du nœud et le premier disque de couleur dessiné.

**Classe Dijkstra**

La classe a deux méthodes :

* calculChemin (Nœud source) : En considérant source comme le nœud de distance 0, cette méthode met à jour les attributs distance et pred de tous les nœuds du graphe (accessibles à partir de source), conformément à l’algorithme de Dijkstra. Elle ne retourne rien.
* plusCourtChemin (Nœud arrivee) : Cette méthode retourne une liste de nœuds ordonnée correspondant au chemin le plus court allant de la source (qui est déjà définie lors de l’appel à calculChemin) à l’arrivée. La méthode part du nœud arrivée et remonte en passant par le prédécesseur et ajoute chaque nœud rencontré à une liste de nœuds. On s’arrête lorsqu’on rencontre null, or seule la source a comme prédécesseur l’objet null. La liste contient donc les nœuds dans le chemin inverse, on fait appel à la méthode *reverse* de la classe Collections de l’API qui permet de gérer les structures de données. Cette méthode inverse l’ordre des éléments d’une collection et permet d’obtenir un chemin dans le bon ordre.

# Phase 4 : Documentation du fonctionnement du système et analyse de sa qualité

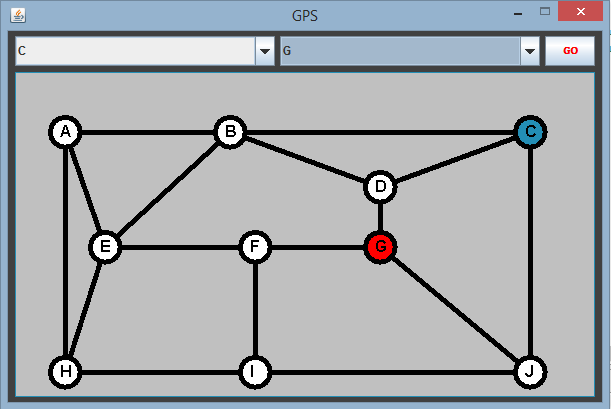
**Jeux de test :**

Fonctionnement du système :

L’utilisateur choisit un point de départ dans la première liste déroulante et un point d’arrivée dans la deuxième liste. Le système colorie le nœud de départ en bleu et le nœud d’arrivée en rouge.

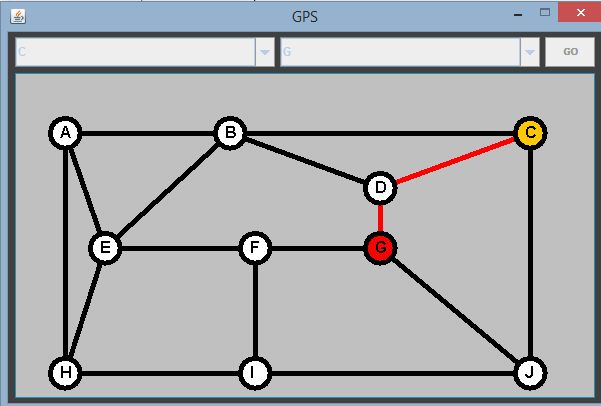
Le bouton GO de l’application permet de démarrer la navigation. Une fois la navigation lancée un algorithme (DIJKSTRA) calcule le chemin le plus court entre les deux nœuds et le colorie en rouge. Des évènements peuvent survenir lors de la navigation (congestions et accidents). Ils sont matérialisés sur le réseau par le coloriage de l’arc en jaune. Si l’arc colorié se trouve sur le chemin que l’algorithme a déjà tracé, un nouvel itinéraire est affiché, la navigation est alors poursuivie.

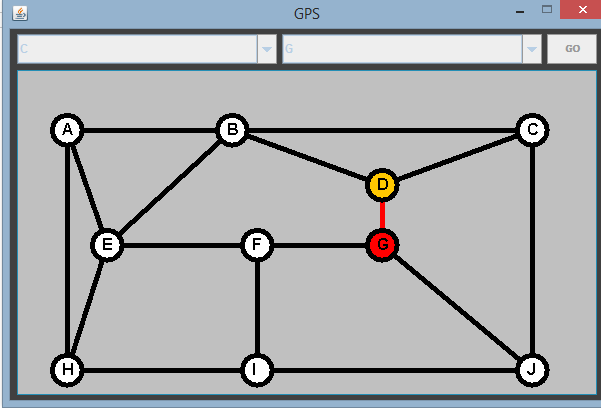
*\*Les couleurs utilisées pour les tests sont susceptibles de changer dans la version finale du produit.*



Voici quelques tests effectués avec le système :

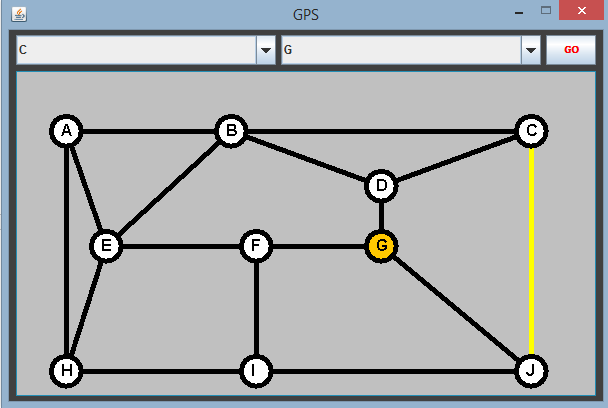
**1er test : C vers G**





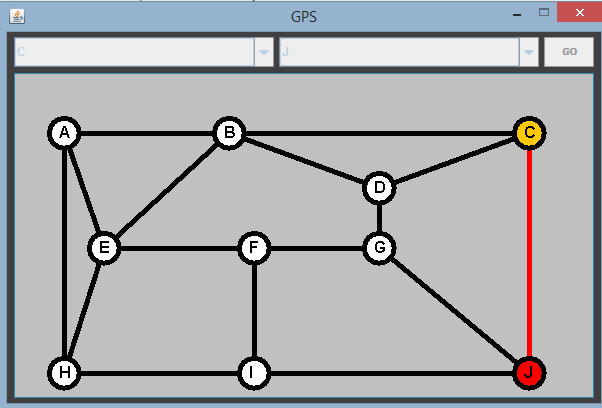
A la fin de la navigation un tronçon aléatoire (C J) est marqué en jaune.

Chaque fois qu’on arrive sur un nœud il y’a 75% de chances qu’une congestion se déclenche, et si c’est le cas l’emplacement de la congestion est aléatoire.

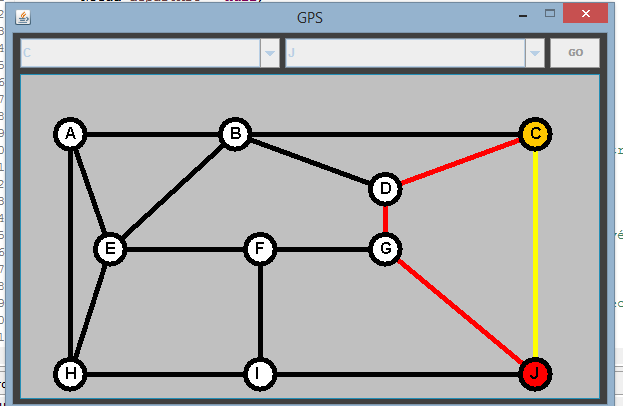


**2eme test : C vers J**

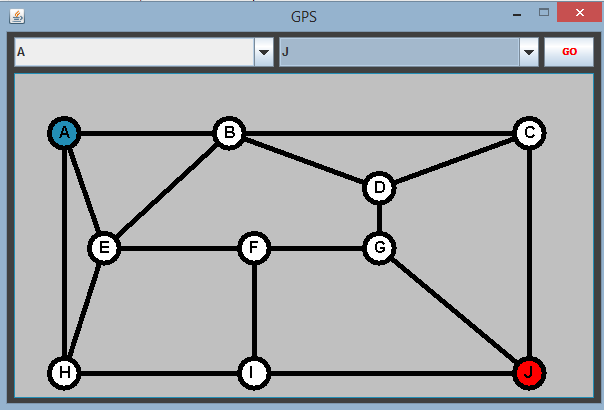
* Chemin normal

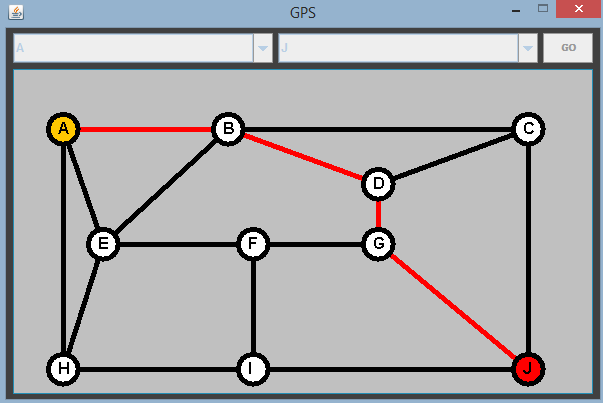


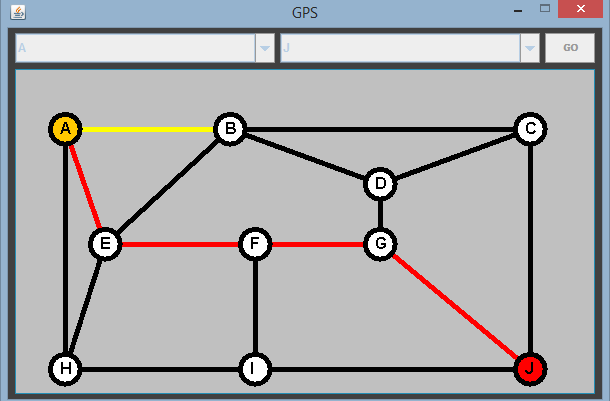
* Lorsqu’il y’a congestion entre C et J un nouveau itinéraire est retracé.

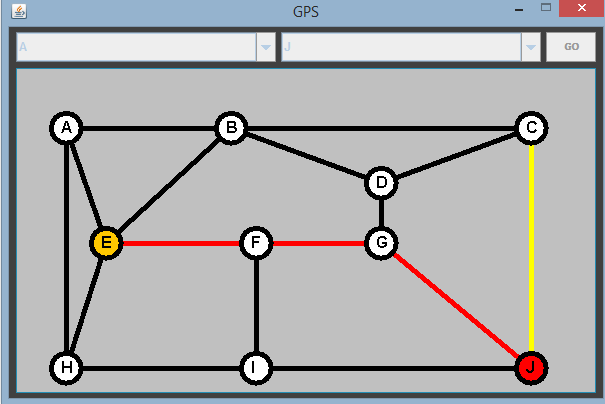


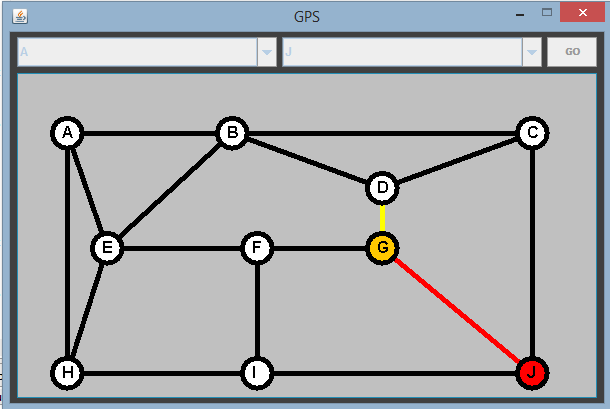
**Un dernier test : A vers J**







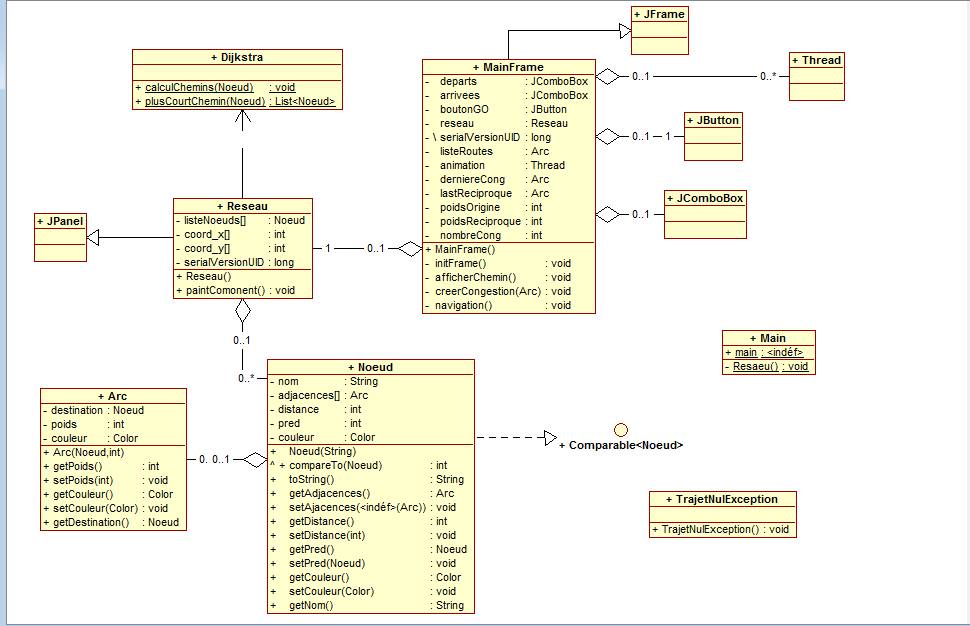




## Analyse de la qualité

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Type de critères** | **Qualité recherchée** | **Référence au code (no ligne ou nom méthode,etc)** | **Observation positive** | **Commentaire insuffisance de qualité** | **Correction à apporter (cas d’insuffisance)** |
| Concept objet :  Héritage | Réutilisation | Reseau extends JPanel,  MainFrame extends JFrame | Plusieurs composants de l’API ont été réutilisés grâce à l’héritage. | - | - |
| Concept objet : Encapsulation | Protection | Attributs d’instance, dans toutes les classes | Tous les attributs d’instance ont été déclarés privés. | - | - |
| Concept objet : Modularité | Compréhension du code | - | Le code du programme est subdivisé en plusieurs modules et le rôle de chaque classe est bien défini. | - | - |
| Conception | Respect des spécifications | - | Tout le réseau est affiché. Le chemin le plus court est calculé selon Dijkstra et affiché. Les congestions sont aléatoires, dynamiques et identifiables. La congestion peut provoquer un changement de chemin. | - | - |
| Conception | Robustesse | Classe MainFrame |  | Le cas où l’itinéraire est indéfini n’est pas pris en compte.  Le cas où depart et arrivee sont identiques n’est pas pris en compte. | Ajouter des exceptions pour les deux cas dans le code. |
| Lecture | Présence Commentaires | Toutes les classes. | Toutes les classes sont bien commentées. La JavaDoc est aussi utilisée. | Le code de Dijkstra réutilisé n’est pas commenté | Ajout de commentaires et restructuration du code dans la classe Dijkstra. |
| Lecture | Organisation du code | Toutes les classes. | Le code est bien structuré, selon la convention. | - | - |
| Lecture | Indentation | Toutes les classes. | Les règles d’indentation sont respectées. | - | - |
| Lecture | Déclarations | Toutes les classes. | Les conventions de déclarations sont respectées. Les variables qui doivent être initialisées le sont. | Le nom du package ne respecte pas la convention. Il n’y a pas de tab entre le type des objets déclarés et leur nom. | Ajouter com, edu, gov, net, mil ou org au nom du package.  La tabulation n’est pas nécessaire, le code reste lisible. |

**Diagramme UML Final :**



# Références

|  |  |
| --- | --- |
| [1] | Wikipédia, l'encyclopédie libre, «Algorithme de Dijkstra,» 06 Janvier 2015. [En ligne]. Available: http://fr.wikipedia.org/wiki/Algorithme\_de\_Dijkstra. [Accès le 25 03 2015]. |
| [2] | Algolist, «Dijkstra's algorithm in Java,» [En ligne]. Available: http://www.algolist.com/code/java/Dijkstra's\_algorithm. [Accès le 25 03 2015]. |
| [3] | Wikipédia, l'enclyclopédie libre, «Tas (informatique),» 06 Mars 2015. [En ligne]. Available: http://fr.wikipedia.org/wiki/Tas\_(informatique). [Accès le 25 Mars 2015]. |