TP d'algorithmique : Itinéraire de bus avec Dijkstra

Lilian PATTIER ISN-1A

11 juin 2017

Compte-rendu de TP

Table des matières

1 Analyse du problème

1.1 Description du problème

Ce problème traite de l'implémentation d'un programme de calcul d'itinéraires sur un réseau de bus. Il porte principalement sur la notion de recherche d'un chemin optimal dans un graphe. On retrouve le même genre de problèmes dans plusieurs domaines (réseau internet, itinéraire gps...).

Dans notre cas, le graphe représente un réseau de trois lignes de bus (A,B,C). Les sommets représentent les stations tandis que les arêtes représentent les routes. Chaque arête est valuée par un nombre qui représente le temps de parcours entre les deux stations adjacentes. Le but de ce TP est de trouver, à partir de n'importe quelles stations de départ et d'arrivée, l'itinéraire le plus court en matière de temps.

Enfin, ce TP à également pour but de gérer les flux d'entrées/sorties en JAVA. Nous disposons de deux fichiers textes représentant respectivement les arrêts et lignes correspondantes puis les coordonnées des différents arrêts. La première partie de notre travail consiste à implémenter ces informations pour construire le graphe correspondant. Ensuite, nous devons créer le système de recherche de chemin optimal. Finalement, une interface graphique doit être conçu avec l'affichage du chemin précedemment trouvé.

1.2 Solutions proposées

La solution de force brute n'est pas envisageable dans un problème de recherche de plus court chemin dans un graphe en raison de la complexité. Il existe deux algorithmes déterministes principaux pour répondre à ce problème :

- L'algorithme de Dijkstra qui permet de trouver un chemin optimal.
- L'algorithme A^* plus rapide mais dont l'optimalité n'est garantie que sous certaines conditions.

1.3 Choix d'une solution

Dans ce sujet, le choix de l'algorithme de Dijkstra nous est imposé. Cependant nous verrons que nous devons modifier l'algorithme de façon à répondre à notre problématique (Mémorisation du précédent élement, valeur variable pour les arêtes sous certaines conditions). Nous reviendrons à ces modifications plus tard.

2 Présentation du programme

2.1 Classes préalables

Les classes `Liste` et `Element` (Annexes 5.4 et 5.3) fournies et nécessaires au fonctionnement de l'algorithme de recherche de chemin optimal ont dû être modifiées pour répondre à nos besoins. L'algorithme de Dijsktra calcule un chemin optimal mais ne permet pas dans sa forme initiale d'expliciter le chemin choisi. Pour ce faire, en sachant que chaque sommet a un unique prédécesseur dans la solution de Dijsktra, il a été rajouté une variable d'instance `int preced` dans la classe `Element`. De cette façon, nous pourrons plus tard exploiter la solution de l'algorithme en partant du sommet d'arrivée pour remonter jusqu'à la source et ainsi disposer de l'itinéraire correspondant.

Une variable d'instance `String ligne` a également été rajoutée dans la classe `Liste` pour disposer de la ligne correspondante au tronçon entre l'arrêt `num_noeud` et l'arrêt suivant. Cette variable sera utile pour déterminer s'il y a un changement de ligne (correspondance) dans la solution et ainsi y ajouter le temps de correspondance.

Enfin, une classe `Arret` (Annexe 5.2) disposant des attributs `String nom`, `int coordX` et `int coordY` a été implémentée de façon à instancier les stations et pouvoir les situer géographiquement. Cette classe dispose d'une méthode `distance2Arets(Arret A)` qui renvoie la distance entre l'arrêt instancié et l'arrêt `A`.

2.2 Création du graphe

2.2.1 Méthodes préalables

La récupération des données des fichiers textes se fait à l'aide de la classe `Fichier` (Annexe 5.5). Les deux premières méthodes qui nous intéressent sont les méthodes `listeArrets()` et `listeNomArrets()` de la classe `Methodes` (Annexe 5.6). Ces deux classes lisent dans le fichier `kelkonk_bus.geo` (Annexe 5.10).

La première renvoie un vecteur `vecArets` contenant les différents objets `Arret` dont la classe a été défini dans le fichier `Arret.java`. Cette fonction lit les différentes lignes du fichier texte et y récupère les arrêts et leurs coordonnées (x et y) à l'aide de la méthode `split` associée au délimiteur de tabulation `\t`. Pour chaque ligne lue (différente d'un commentaire), un arrêt est instancié puis rajouté au vecteur.

La seconde méthode renvoie simplement une liste `listeNomArrets` contenant les noms des différents arrêts de façon à associer chaque arrêt avec son numéro correspondant. En effet, les fonctions liées au graphe fonctionnent avec des entiers et non des chaînes de charactères. Cependant l'utilisateur doit pouvoir avoir accès aux noms des arrêts. On peut ainsi récupérer le nom d'un arrêt à partir de son numéro (par exemple : `listeNomArrets.get(0)` renverra `A1`) et vice-versa (`listeNomArrets.indexOf(A1)` renverra `0`).

2.2.2 Classe GrapheParListe

Une fois ces méthodes créées, nous pouvons nous intéresser à la création du graphe. La classe `GrapheParListe` (Annexe 5.1) dispose d'une variable d'instance `adj` du type `Liste` qui correspond à la liste d'adjacence du graphe.
Le constructeur de cette classe prend en argument un `Fichier` (que l'on instanciera dans la classe `Principale` et qui correpsond au fichier `kelkonk_bus.graph`), ainsi que le vecteur `ListeArrets` et la liste `listeNomArrets`.

Concrètement, ce constructeur lit dans le fichier les lignes deux à deux en récupérant les arrêts, ainsi que la ligne de bus correspondante : pour chaque arrêt courant `numAret` et arrêt précédememnt lu `numAretPrec`, on calcule la distance entre ces deux arrêts à l'aide de la méthode `distance2Arets` de la classe `Arret`. Puis, on calcule le temps de trajet correspondant à ce tronçon à l'aide de la méthode `tempsTrajet2Arets` de la classe `Methodes` qui prend en argument la ligne de bus correspondante et la distance précédemment calculée.

Finalement, on instancie la liste `Liste(numAret, adj[numAretPrec], tempsTrajet, ligneBus)` dans `adj[numAretPrec]`. Il est a noté que dans la liste d'adjacence, on a finalement des temps de parcours et non des distances ce qui est conforme à l'optimalité temporelle que l'on souhaite.

2.3 Recherche du plus court chemin

2.3.1 Méthodes préalables

Le calcul du plus court chemin doit également prendre en compte les temps correspondants aux changements de lignes. Comme nous le verrons en 4.1, il n'est pas possible de prendre en compte ces temps en amont du calcul de chemin optimal, ni à posteriori. Nous devons donc considérer ces temps supplémentaires pendant l'algorithme de Dijkstra.

Pour ce faire, une méthode `changementLigne` a été implémentée dans la classe `GrapheParListe`. Elle prend en argument trois arrêts `sommet1, sommet2` et `sommet3` et renvoie un booléen égal à `true` s'il y a un changement de ligne entre le tronçon `sommet1-sommet2` et le tronçon `sommet2-sommet3`.

Les méthodes `arc` et `valeurArc` de la classe `GrapheParListe` n'ont pas été modifiées.

2.3.2 Algorithme de Dijkstra

L'algorithme de Dijkstra correspond à la méthode `plusCourtChemin` de la classe `GrapheParListe`. Elle prend en argument un numéro de sommet source `num_sommet` et renvoie un vecteur solution `S`. L'algorithme en lui-même a été un petit peu modifié pour pouvoir répondre à notre problématique. Outre l'ajout du sommet précédent à chaque objet `Element`, la principale problématique consistait à ajouter (ou non) les 300 secondes correspondant à un changement de ligne effectif. Cela dépend de la ligne sur laquelle on circulait avant d'arriver à un sommet de changement.

Pour ce faire, lors du recalcul des distances de tout sommet `x` qui possède un arc avec le sommet courant `m`, on vérifie à l'aide de la méthode `chagementLigne` si le tronçon `m.preced - m.sommet` correspond à la même ligne que le tronçon `m.sommet - x.sommet`. Si tel est le cas, on ajoute simplement 60 secondes (temps d'arrêt du bus). Sinon, on ajoute les 300 secondes de correspondance.

2.3.3 Itinéraire du chemin optimal

Une fois que nous disposons de la solution `S` de l'algorithme de Dijkstra et en sachant que chaque `Element` dispose d'un attribut `preced`, on peut remonter la solution en partant du noeud de destination. La méthode correspondant est `cheminOptimal`. Elle prend en argument un sommet source `source` et un sommet de destination `dest` ainsi que la solution `S` de l'algorithme de Dijkstra. Cette méthode renvoie un vecteur `cheminOptimal` comportant :

- Une ArrayList `ListChemin` qui contient la liste des numéros des sommets successifs du chemin.
- Un entier `tempsTrajet` qui correspond au temps de parcours entre la source et la destination.

Cette méthode calcule dans un premier temps, le temps de parcours en cherchant dans le vecteur `S` l'élement de destination et en récupérant l'attribut `distance` de cet élement. On y ajoute les 180 secondes correspondant au temps d'attente de l'arrivée du bus. Enfin, on enlève les 60 secondes qui correspondent au temps d'arrêt du bus au dernier sommet. En effet, on considère que dès lors que le bus arrive à sa destination, le trajet s'achève.

2.4 Méthode principale

La méthode `Principale` (Annexe 5.7) reprend toutes les méthodes précédemment définie pour implémenter la recherche d'itinéraire. On a trois objectifs :

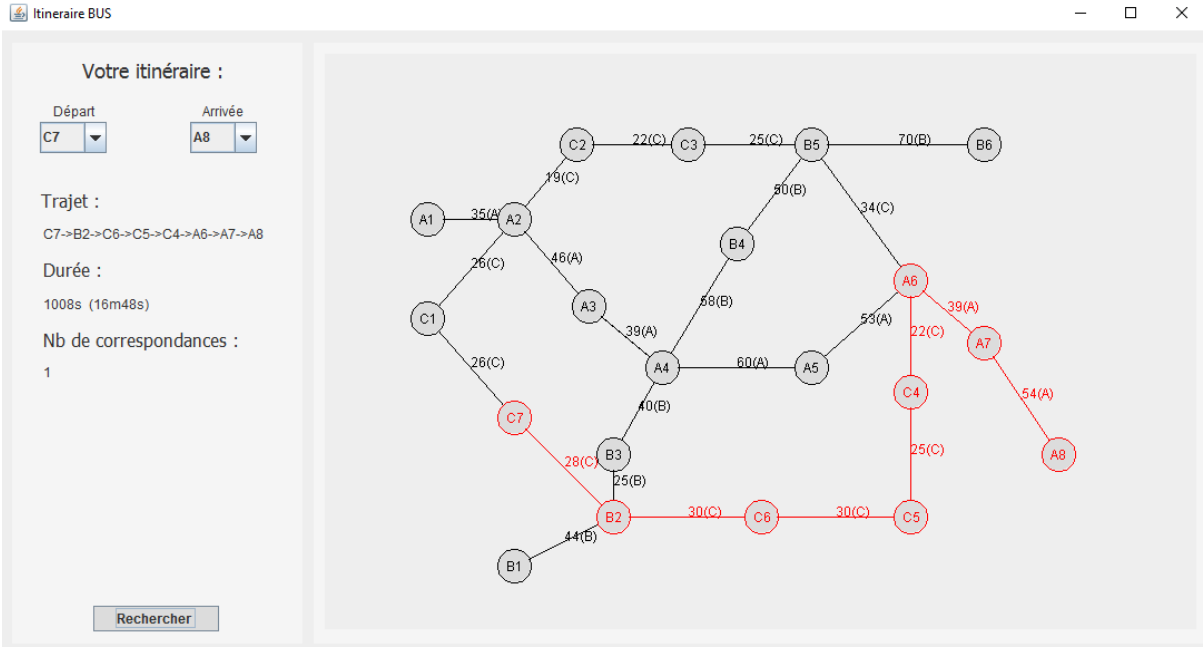
- Afficher le trajet détaillé.
- Afficher le temps de parcours.
- Afficher le nombre de correspondances.

Pour cela on dispose de trois méthodes :

- La méthode `afficherTrajet` de la classe `Methodes` prend en argument `ListChemin` la liste des arrêts de la solution fournie par la méthode `cheminOptimal` et la liste `listeNomArrets`. Elle renvoie une chaîne de charactères sous la forme `A1->A2->...->B5` correspondant au trajet.
- La méthode `tempsTrajetMinute` de la classe `Methodes` prend en argument le temps de trajet fourni par la méthode `cheminOptimal` et renvoie une chaîne de charactère correspondant au temps de trajet en minutes.
- Enfin, la méthode `nombreCorrespondance` de la classe `GrapheParListe` prend en argument `ListChemin` et reparcourt cette liste à l'aide de la méthode `changementLigne` pour compter le nombre de correspondances.

On peut finalement implémenter la méthode principale.

2.5 Affichage graphique



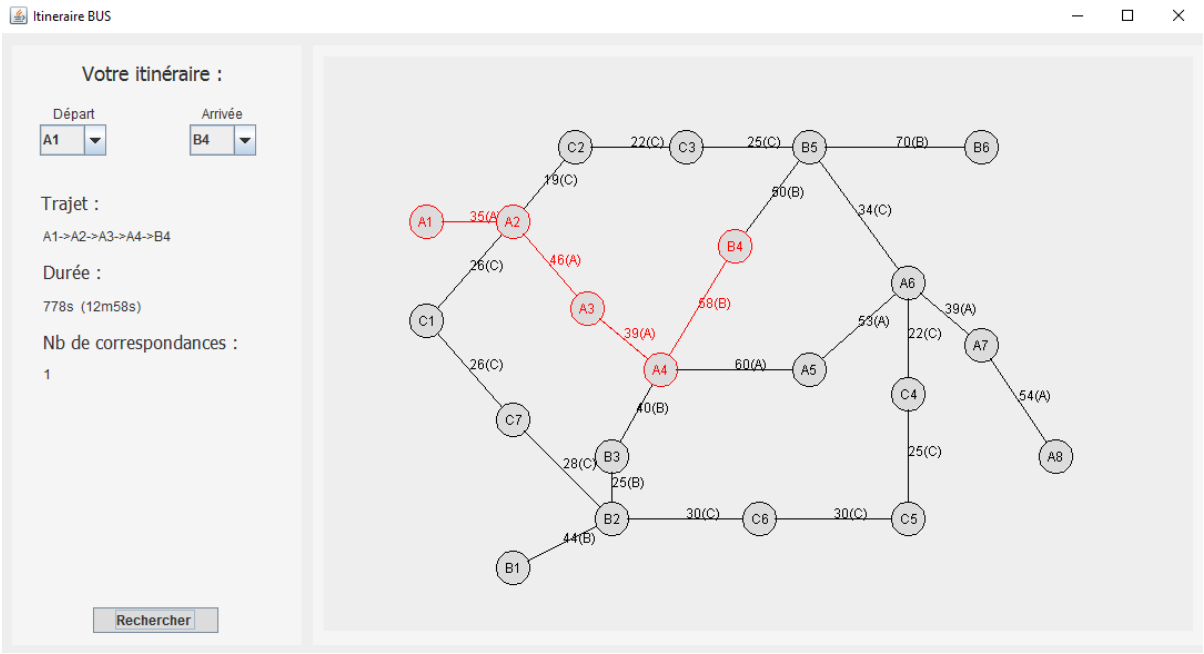
L'interface graphique est implémentée à l'aide de deux classes :

La première nommée `planGraphique` (Annexe 5.8) correspond à l'affichage du plan (partie droite de la fenêtre). Elle se décompose en deux parties, l'affichage des arêtes et l'affichage des sommets. On commence par récupérer les arrêts et leurs coordonnées. Puis, on vérifie si ces arrêts font partie du trajet. Si c'est le cas, on colore les arêtes adjacentes en rouge, sinon en noir puis on les places dans le plan. On procède de même avec les sommets. Dans les deux cas, on place les éléments en fonction des coordonnées réelles des sommets (que l'on a divisé par 4.5 de façon à ce que le plan soit à l'échelle de la fenêtre.)

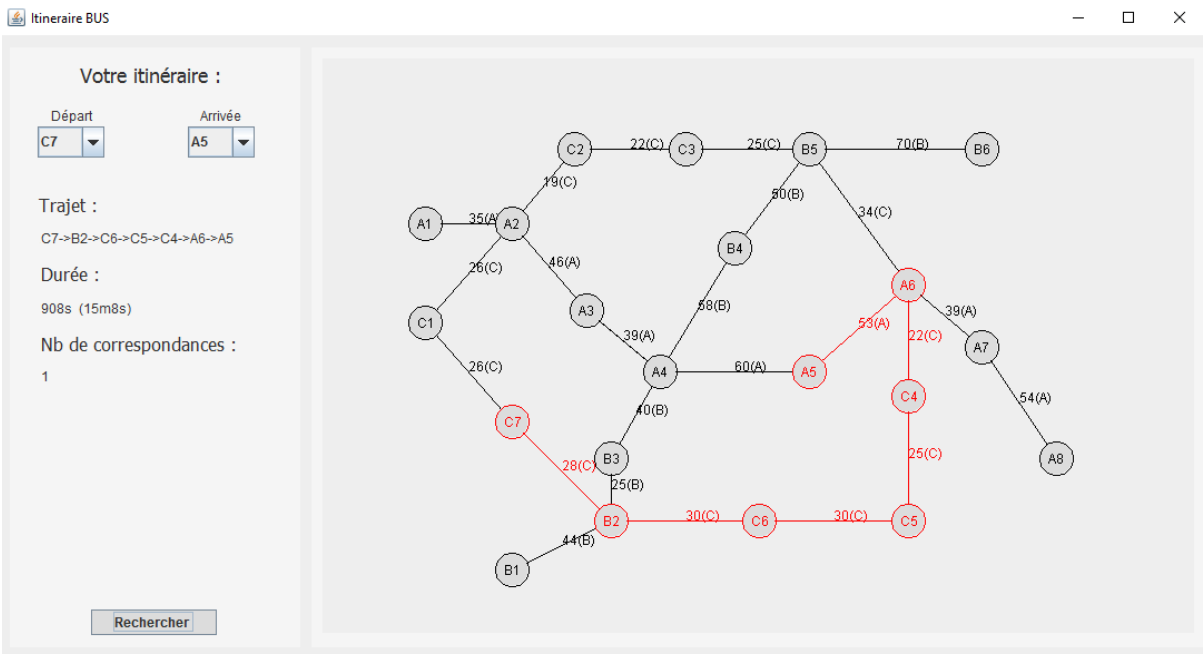
La seconde classe `interfaceGraphique` (Annexe 5.9) correspond à la fenêtre globale. On y incorpore l'algorithme de recherche du chemin optimal dans le listener du bouton de recherche. On commence par récupérer les donnéees des combobox des arrêts de départ et de destination que l'on convertit ensuite en les entiers correspondants. On applique l'algorithme puis on réactualise les élements de la fenêtre pour afficher les différentes informations.

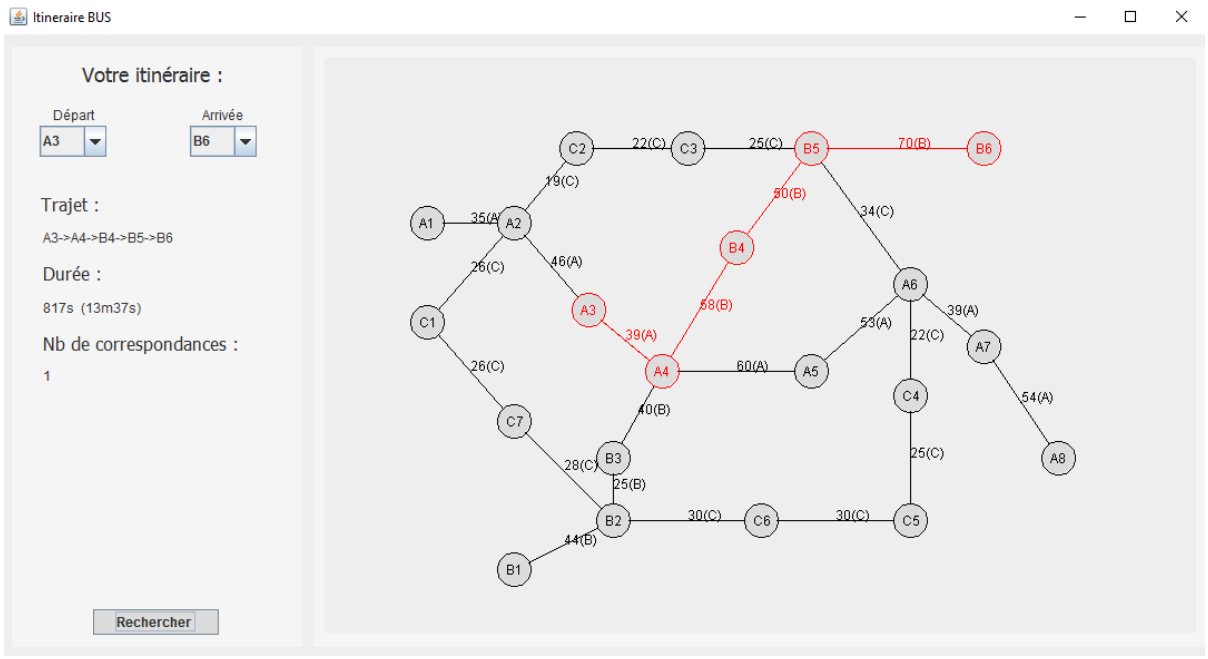
3 Jeu de tests

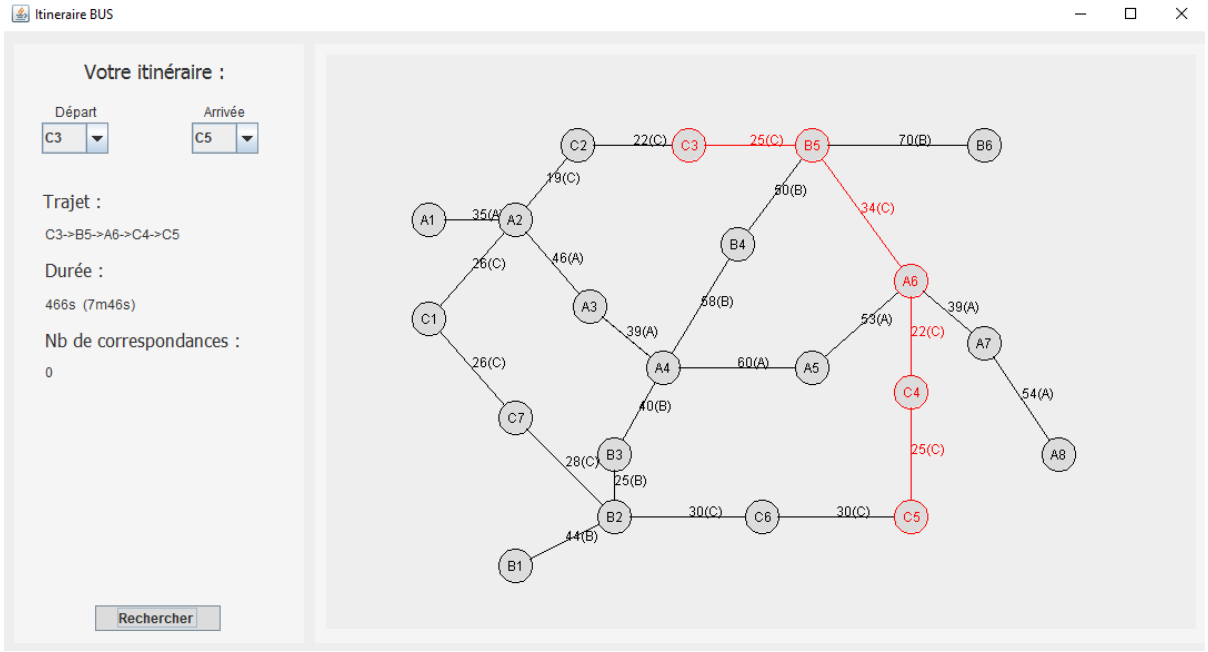
Quelques exemples de tests du programme :

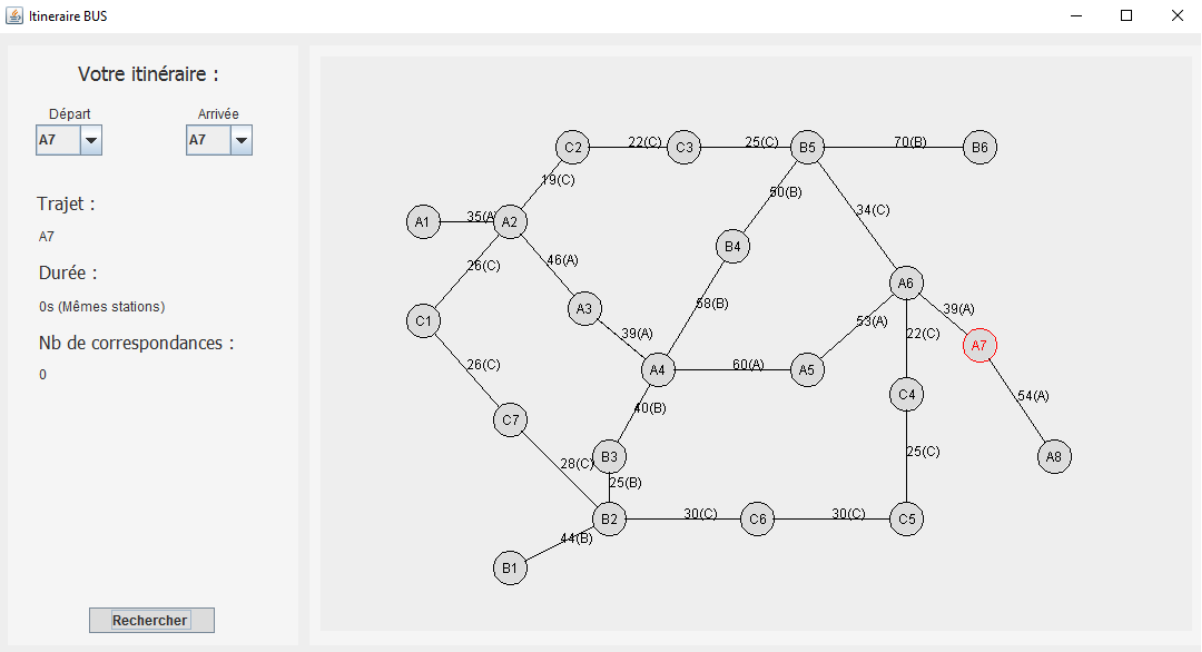


Calcul effectif pour le premier cas :
180 secondes d'attente à la première station.
60 secondes de temps d'arret en `A2,A3`.
300 secondes de correspondance en `A4`.
$35, 46, 39$ et 58 secondes de temps de trajet.
Total : $180 + 60 \times 2 + 300 + 35 + 46 + 39 + 58 = 778s$ soit 12 minutes et 58 secondes.




Itineraire BUS
Votre itinéraire :
Départ
A3
Arrivée
B6
Trajet :
A3->A4->B4->B5->B6
Durée :
817s (13m37s)
Nb de correspondances :
1
Rechercher
C2
22(C)
C3
25(C)
B5
70(B)
B6
19(C)
80(B)
34(C)
A1
35(A)
A2
B4
26(C)
46(A)
A6
A3
58(B)
39(A)
C1
53(A)
22(C)
A7
39(A)
A4
60(A)
A5
26(C)
C4
54(A)
C7
40(B)
25(C)
28(C)
B3
A8
25(B)
B2
30(C)
C6
30(C)
C5
44(B)
B1

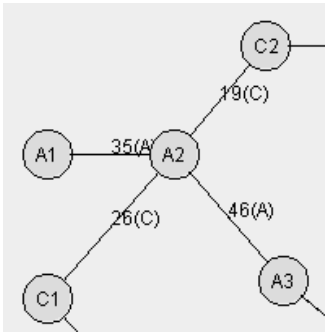

Itineraire BUS
Votre itinéraire :
Départ
C3
Arrivée
C5
Trajet :
C3->B5->A6->C4->C5
Durée :
466s (7m46s)
Nb de correspondances :
0
Rechercher
C2
22(C)
C3
25(C)
B5
70(B)
B6
19(C)
80(B)
34(C)
A1
35(A)
A2
B4
26(C)
46(A)
A6
A3
58(B)
C1
39(A)
53(A)
22(C)
A7
39(A)
A4
60(A)
A5
26(C)
C4
54(A)
C7
40(B)
25(C)
28(C)
B3
A8
25(B)
B2
30(C)
C6
30(C)
C5
44(B)
B1



4 Bilan

4.1 Problèmes rencontrés

Le principal problème rencontré concerne les temps de correspondance entre deux lignes. Ces temps ne peuvent pas être inclus directement sur le graphe. Prenons pour exemple l'image suivante :



Supposons que l'on souhaite arriver en `A1`. Si l'on vient de `A3`, il suffit de rajouter 60 secondes à l'arête `(A2-A1)` correspondant au temps d'attente en `A2`. Cependant, si l'on vient de `C1` ou de `C2`, il faut non plus rajouter 60 secondes à l'arête `(A2-A1)` mais 300 secondes. En d'autres termes, la valeur de l'arête dépend du trajet emprunté. On ne peut donc pas inscrire les temps de correspondance sans connaître le trajet emprunté.

On pourrait dans un premier temps penser qu'il faille alors ajouter les temps d'arrêt et de correspondance après la recherche du chemin optimal. Cependant, on se rend vite compte que ces temps ont une influence sur l'optimalité. En effet si l'on prend l'exemple du trajet `C7-A5` et que l'on ignore les temps de correspondance dans un premiers temps, l'algorithme nous renvoie l'itinéraire `C7-B2-B3-A4-A5`. Or ce chemin qui est à priori plus court emprunte 3 lignes différentes et rajoute donc 600 secondes à la solution finale. Tandis que le trajet `C7-B2-C6-C5-C4-A6-A5` est plus rapide (bien que comportant plus de stations) car il ne rajoute que 300 secondes à la solution finale.

4.2 Solutions trouvées

Nous faisons alors face à un problème important : La solution qui consiste à rajouter les temps de correspondance ultérieurement à la recherche du chemin n'est pas viable. Cependant, nous ne pouvons pas non plus ajouter ces temps sans connaître le chemin emprunté.

En d'autres termes, les temps de trajet ne peuvent être rajoutés qu'une fois le trajet optimal connu ! Or ces temps de correspondances sont nécessaires pour pouvoir calculer le chemin optimal.

La seule solution envisageable est donc de rajouter ces temps supplémentaires pendant le déroulement de l'algorithme de recherche du chemin optimal. C'est cette solution que nous avons mis en place grâce à la méthode `changementLigne` qui, pendant l'algorithme de Dijkstra, recalcule les valeurs des arêtes en prenant en compte les correspondances en fonction du potentiel sommet suivant, du sommet courant et du sommet précédent.

On pourrait penser que cette solution assure l'optimalité mais ce n'est pas le cas :

4.3 Limites du programme

La solution précédemment évoquée est satisfaisante pour une majorité de trajets mais pas pour la totalité. Prenons l'exemple du trajet de `A1` à `B2`. L'algorithme nous renvoie le trajet (`A1-A2-C1-C7-B2`, 715s, 1 correspondance). Ce trajet est effectivement le chemin optimal. Si l'on recherche maintenant le trajet `A1-B1` on pourrait croire qu'il suffit de rajouter la branche `B2-B1` au trajet précédent car ce dernier était déja optimal et qu'il ne reste que cette possibilité. Or ce n'est pas le cas. On se rend compte que le trajet optimal pour `A1-B1` est : (`A1-A2-A3-A4-B3-B2-B1`, 949s, 1 correspondance), alors que notre algorithme nous renvoie : (`A1-A2-C1-C7-B2-B1`, 1059s, 2 correspondances).

Le problème est que l'algorithme ne rajoute les 300 secondes que lorsque le sommet en cours d'analyse précède juste le sommet de correspondance. Ainsi, l'algorithme a déja calculé les chemins optimaux précédents avant de savoir qu'il y aurait des arêtes plus importantes que prévues ce qui remet cause l'optimalité des précédents calculs.
En résumé, le fait d'avoir des valeurs variables pour les arêtes dans l'algorithme de Dijkstra ne garantit pas l'optimalité de la solution.

4.4 Améliorations

L'idéal pour garantir l'optimalité serait de savoir à chaque embranchement de plusieurs lignes s'il y aura ou non des correspondances lors des prochains croisements. Nous pourrions imaginer un programme qui à chaque croisement, vérifie de proche en proche s'il y a un ajout de correspondance à un croisement ultérieur. Mais cette solution s'avère sûrement couteuse en matière de complexité.

5 Annexes

5.1 GrapheParListe.java

```
import utilensemjava.Lecture;
import java.io.IOException;
import java.util.*;

public class GrapheParListe {

    public Liste adj []; //Liste d'adjacence.

    //Initialisation du graphe :
    GrapheParListe(Fichier graphe, ArrayList listeNomArrets, Vector
       ListeArets) throws IOException{

        String l="";
        String []s;
        String lprec ="";
        String [] sprec;
        String ligneBus;
        int numAret, numAretPrec,tempsTrajet;
        double distance;

        l = graphe.lire();
        adj = new   Liste[listeNomArrets.size()];
        for(int i = 0; i < listeNomArrets.size(); i++)
            adj[i] = null;
        while(l != null)
        {
            lprec = l;
            l = graphe.lire();
            if(l != null && !(l.equals("")) && !(l.substring(0,1).
               equals("#")))
            {
                s = l.split(":");
                if(!(lprec.equals("")) && !(lprec.substring(0,1).equals
                   ("#")))
                {
                    sprec = lprec.split(":");

                    numAret = listeNomArrets.indexOf(s[1]); //Numéro de
                        l'arrêt courant.
                    numAretPrec = listeNomArrets.indexOf(sprec[1]); //
                       Numéro de l'arrêt précédent.

                    ligneBus = s[0].substring(0, 1); //Ligne de bus
                       entre l'arrêt courant et l'arrêt précédent.
                    distance = ((Arret)ListeArets.elementAt(numAretPrec
                       )).distance2Arets((Arret)ListeArets.elementAt(
                       numAret));
                    //On calcule la distance entre les deux arrêts.

                    tempsTrajet = Methodes.tempsTrajet2Arets(distance,
                       ligneBus);
```

```
                //On calcule le temps de trajet entre les deux
                    arrêts en fonction de la ligne de bus sur
                    laquelle on est.

                adj[numAretPrec] = new Liste(numAret,adj[
                    numAretPrec], tempsTrajet, ligneBus);
            }
        }
    }

}

//affichage du graphe: liste de noeuds connectés à partir d'un
    sommet sortant
public void afficherGraphe(){
    for(int i = 0; i <  adj.length; i++){
        System.out.print("sommet "+i+": ");
        if(adj[i]!= null) {
            Liste a = adj[i];
            while(a!=null) {
                System.out.print("s"+a.num_noeud+ " " + a.valeur+ "
                    " + a.ligne );

                if(a.suivant!= null )
                    System.out.print("|"+a. suivant.num_noeud+" ->
                        ");
                a = a.suivant;
            }
        }
        System.out.println(" null");
    }
}

//teste s'il existe un arc entre deux sommets.
public boolean arc (int source, int dest){
    boolean arcExiste= false;
    if( adj[source]!=null) {
        Liste a = adj[source];
        while(a !=null) {
            if(a.num_noeud== dest) arcExiste =true;
            if(arcExiste) a=null;
            else
                a = a.suivant;
        }
    }
    return  arcExiste;
}

//retourne la valeur de l'arc entre deux sommets.
public  int valeurArc(int source,int    dest){
    int val = 9999;
    boolean arcExiste=false;
    Liste a = adj[source];
    while(a !=null) {
        if(a.num_noeud== dest) {
            val = a.valeur;
```

```
                arcExiste =true;
            }
            if(arcExiste) a=null;
            else    a = a.suivant;
        }
        return  val;
    }

    //Renvoi true s'il y a un changement de ligne (correspondance)
       entre sommet1->sommet2 et sommet2->sommet3
    public boolean changementLigne(int sommet1, int sommet2, int
       sommet3)
    {
        if(sommet1 == 0 && sommet2 != 1)
            return false;
        //Cette condition est nécessaire car au début de dijkstra, le
           sommet s0 est à l'origine de tous les autres sommets.
        //Pour éviter des erreurs on s'assure que si sommet1 = 0 et
           sommet2 != 1, alors on ne prend pas en compte de changement
           de ligne.
        String ligne1="";
        String ligne2="";
        Liste a = adj[sommet1];
        while(a!=null) {
            if(a.num_noeud == sommet2)
                ligne1 = a.ligne;
            a = a.suivant;
        }
        a = adj[sommet2];
        while(a!=null) {
            if(a.num_noeud == sommet3)
                ligne2 = a.ligne;
            a = a.suivant;
        }
        return (!ligne1.equals(ligne2));
    }

    //recherche le plus court chemin d'un sommet source vers tous les
       autres noeuds.
    public  Vector plusCourtChemin( int num_sommet){
        final int INFINI = Integer.MAX_VALUE;
        Vector S = new  Vector(); //vector de solution
        Vector D = new Vector(); //vector du départ

        //initialiser l'ensemble D
        for(int i = 0; i <adj.length; i++){
            if(i!=num_sommet)
                D.addElement(new Element(i,INFINI,0));
            else
                D.addElement(new Element(i,0,0));
        }

        while(D.size()!=0){
            //on cherchel'élément qui a la plus petite distance/
            int indice_min=0;
            int dm=INFINI;
```

```
        int sm=((Element) D.elementAt(0)).sommet;
        int pm=((Element) D.elementAt(0)).preced;

        /*int changementLigne = 0;
        if(!adj[sm].ligne.equals(adj[pm].ligne))
            changementLigne = 300;
         */
        for (int i = 0; i < D.size(); i++){
            if(dm > ((Element) D.elementAt(i)). distance){
                dm = ((Element) D.elementAt(i)).distance;
                sm = ((Element) D.elementAt(i)).sommet;
                pm=((Element) D.elementAt(i)).preced;
                indice_min = i;
            }
        }
        Element m = new Element(sm,dm,pm);

        //on l'ajoute dans S puis on le supprime de D
        S.addElement(m);
        D.removeElementAt(indice_min);

        //on recalcule dx pour tout sommet x de D qui possède un
           arc avec m
        for (int i = 0; i < D.size(); i++) {
            Element x = (Element) D.elementAt(i);
            if (arc(m.sommet, x.sommet)) {
                int tempsCgt = 0;

                //On vérifie s'il y a ou non un changement de ligne
                    entre l'arrêt précédent m, l'arrêt m et l'arrêt
                    x.
                if(m.preced != m.sommet && changementLigne(m.preced
                   , m.sommet, x.sommet))
                    tempsCgt = 300; //Si tel est le cas, on rajoute
                        un temps de correspondance
                else
                    tempsCgt = 60; //Sinon on rajoute simplement le
                        temps d'arrêt du bus

                int d = m.distance + valeurArc(m.sommet, x.sommet)
                   + tempsCgt;
                if (d < x.distance) {
                    x.distance = d;
                    x.preced = m.sommet;
                    D.setElementAt(x, i);
                }
            }
        }
    }
    return  S;
}


//cheminOptimal selectionne le parcours à effectuer à partir du
   résultat de la méthode précédente et renvoi un vecteur
   comportant :
```

```
//- ListChemin : La liste des numéros des sommets successifs du
   trajet
//- tempsTrajet : Le temps de parcours entre la source et la
   destination.

public Vector cheminOptimal(int source, int dest, Vector S/*, List
   nomArets*/)
{
    Vector cheminEtTemp = new Vector();
    int tempsTrajet=0;
    List ListChemin = new ArrayList();
    ListChemin.add(0,dest); //On ajoute la destination à la liste
       des arrêts du parcours.

    //Calucl du temps de trajet :
    for(int i=0; i<S.size(); i++)
    {
        if(((Element)S.elementAt(i)).sommet == dest)
        {
            //On calcule le temps de trajet (en rajoutant les 180
               secondes d'attente au départ et en substituant les
               60 secondes à l'arrivée.
            //(On considère que dès que l'on arrive, le trajet est
               terminé)
            tempsTrajet = ((Element)S.elementAt(i)).distance
               +180-60;
        }
    }

    //Calcul des arrêts successifs: On remonte la liste S tant qu'
       on est pas à la source.
    while(source != dest)
    {
        for(int i=0; i<S.size(); i++)
        {
            if(((Element)S.elementAt(i)).sommet == dest)
            {
                dest = ((Element)S.elementAt(i)).preced;
                ListChemin.add(0,dest);
            }
        }
    }

    cheminEtTemp.addElement(ListChemin);
    cheminEtTemp.addElement(tempsTrajet);
    return cheminEtTemp;
}

//Calcul du nombre de correspondances (Méthode pûrement informative
   ) :

public int nombreCorrespondance(ArrayList ListChemin)
{
    int compteur=0;
    int a=0, b=0, c=0;
```

```
        if(ListChemin.size() <= 2) //Si le trajet est inférieur à 2
           stations, on sait qu'il n'y aura pas de changement de bus.
            return 0;
        else //Sinon, on vérifie les arrêts de la liste des sommets du
           trajet pour vérifier s'il y a des changements.
        {
            for(int i=0;i<ListChemin.size()-2;i++)
            {
                a=(int) ListChemin.get(i);
                b=(int) ListChemin.get(i+1);
                c=(int) ListChemin.get(i+2);

                if(changementLigne(a,b,c))
                    compteur++;
            }
        }
        return compteur;
    }

}
```

5.2 Arret.java

```
public class Arret {

    public String nom;
    public int coordX;
    public int coordY;

    Arret(String nom, int coordX, int coordY)
    {
        this.nom = nom;
        this.coordX = coordX;
        this.coordY = coordY;

    }

    //renvoi la distance entre l'arrêt instancié et l'arrêt A
    public double distance2Arets(Arret A)
    {
        double a = Math.sqrt(Math.pow(this.coordX - A.coordX,2)+Math.
           pow(this.coordY - A.coordY, 2));
        return a;
    }

}
```

5.3 Element.java

```
public class Element {

    int sommet; //numéro du sommet courant.
    int distance; //distance vers le sommet source.
    int preced; //numéro du sommet précédent.

    Element(int s, int d, int p){
        sommet=s;
```

```
        distance=d;
        preced=p;
    }
}
```

5.4 Liste.java

```
public class Liste {

    int num_noeud; //Numéro du sommet courant.
    int valeur;  //Valeur de l'arc.
    String ligne; //Ligne de bus correspondante.
    Liste suivant; //Liste des suivants.

    Liste ( int n, Liste t, int v, String l) {
        num_noeud=n;
        suivant= t;
        valeur=v;
        ligne=l;
    }
}
```

5.5 Fichier.java

```
import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.File;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
public class Fichier {
    private BufferedWriter fW;
    private BufferedReader fR;
    private char mode;
    public Fichier(String nomDuFichier, String s) throws IOException{
        mode = (s.toUpperCase()).charAt(0);
        File f = new File(nomDuFichier);
        if (mode == 'R' || mode == 'L')
            fR = new BufferedReader(new FileReader(f));
        else if (mode == 'W' || mode == 'E')
            fW = new BufferedWriter(new FileWriter(f));
    }
    public void fermer() throws IOException {
        if (mode == 'R' || mode == 'L') fR.close();
        else if (mode == 'W' || mode == 'E') fW.close();
    }
    public String lire() throws IOException {
        String chaine = fR.readLine();
        return chaine;
    }
    public void ecrire(int tmp) throws IOException {
        String chaine = "";
        chaine = chaine.valueOf(tmp);
        if (chaine != null) {
            fW.write(chaine,0,chaine.length());
            fW.newLine();
        }
```

```
    }
}
```

5.6 Methodes.java

```
import java.awt.List;
import java.io.IOException;
import java.util.ArrayList;
import java.util.Vector;

public class Methodes {

    //renvoi un vecteur comportant les objets Arrets du graphe
       correspondant.
    //pour rappel : Arret(nom, coordonnée x, coordonnée y)

    public static Vector listeArrets() throws IOException{
        Fichier graphe = new Fichier("kelkonk_bus.geo","R");
        String l;
        String s[];
        Vector vecArets = new Vector();
        while((l = graphe.lire()) != null){
            if(!(l.substring(0,1).equals("#")))
            {
                s = l.split("\t");
                vecArets.addElement(new Arret(s[0],Integer.parseInt(s
                   [1]),Integer.parseInt(s[2])));
            }
        }
        return vecArets;
    }

    //renvoi une liste comportant le nom des différents arrêts de façon
        à faire correspondre chaque arrêt avec son numéro :
    //Ex : listeNomArrets.get(0)->A1, ..get(1)->A2,...get(20)->C7

    public static ArrayList listeNomArrets() throws IOException{
        Fichier graphe = new Fichier("kelkonk_bus.geo","R");
        String l;
        String s[];
        ArrayList listeNomsArrets = new ArrayList();
        while((l = graphe.lire()) != null){
            if(!(l.substring(0,1).equals("#")))
            {
                s = l.split("\t");
                listeNomsArrets.add(s[0]);
            }
        }
        return listeNomsArrets;
    }

    //prend en argument une distance et une ligne de bus et renvoi le
       temps de trajet correspondant

    public static int tempsTrajet2Arets(double distance, String
       ligneBus)
```

```
    {
        int tempsTrajet = 0;
        int vitesse = 10;

        switch (ligneBus) {
        case "A":
            vitesse = 10;
            break;
        case "B":
            vitesse = 10;
            break;
        case "C":
            vitesse = 20;
            break;
        }

        tempsTrajet = (int) (distance/ vitesse);
        return tempsTrajet;
    }

    //Renvoi un String de la forme A0->A1->...->A5 correspondant à l'
       itinéraire calculé :

    public static String afficherTrajet(ArrayList ListChemin, ArrayList
        listeNomArrets)
    {
        String trajet="";

        for(int i=0; i < ListChemin.size()-1; i++)
        {
            trajet +=  listeNomArrets.get((int) ListChemin.get(i)) + "
               ->";
        }
        trajet +=  listeNomArrets.get((int) ListChemin.get( ListChemin.
           size()-1));

        return trajet;
    }

    //Prend en argument un temps en seconde et renvoi une chaîne de
       charactères correspondant au temps de trajet en minutes :

    public static String tempsTrajetMinute(int tempsTrajet)
    {
        String tempsMinutes = tempsTrajet/60+"m"+tempsTrajet % 60+"s";
        return tempsMinutes;
    }

}
```

5.7 Principale.java

```
import utilensemjava.Lecture;
import java.io.IOException;
import java.util.*;
```

```
public class Principale {
    public static void main(String[] args) throws IOException{

        ArrayList listeNomArrets = Methodes.listeNomArrets();
        //On crée la liste des noms des arrêts pour faire correspondre
           leur nom avec leur id
        //Ex : listeNomArets.get(0)->A1, ..get(1)->A2,...get(20)->C7

        Vector Q = Methodes.listeArrets();
        //Le vector Q contient les objets "Arrets" de spécifications (
           nom, coordx, coordy)

        //Création du graphe
        Fichier graphe = new Fichier("kelkonk_bus.graph","R");

        //creation du graphe a partir du Fichier .graph et du vecteur
           des arrets :
        GrapheParListe g = new  GrapheParListe(graphe, listeNomArrets,
           Q);

        // g.afficherGraphe(); //(affichage du graphe)

        /*----VALEURS A CHOISIR POUR CALCULER UN ITINERAIRE : -----*/
        String provenance = "B1";
        String destination = "A8";
        /*-----------------------------------------------------------*/

        int provId = listeNomArrets.indexOf(provenance);
        int destId = listeNomArrets.indexOf(destination);

        //On calcule le chemin optimal entre une source et tous les
           arrêts puis on stocke dans un vecteur (dijsktra)
        Vector S = g.plusCourtChemin(provId);

        //On remonte les élements du vecteur S à partir de la source
           pour trouver les différentes étapes du trajet.
        System.out.println("Itinéraire: ");
        Vector trajetFinal = g.cheminOptimal(provId, destId, S);

        //Affichage du trajet
        ArrayList stationSuccessive = (ArrayList) trajetFinal.get(0);
        String affichageTrajet = Methodes.afficherTrajet(
           stationSuccessive, listeNomArrets);
        System.out.println(affichageTrajet);

        //Affichage du temps de trajet :
        System.out.print("Temps de trajet: ");
        int temps = (int) trajetFinal.get(1);
        System.out.println(temps + "s");

        //Calcul du nombre de correspondances :
        System.out.print("Nombre de correspondances: ");
        int nombreCorresp = g.nombreCorrespondance(stationSuccessive);
        System.out.println(nombreCorresp);

    }
```

```
}
```

5.8 planGraphique.java

```
import java.awt.*;
import java.io.IOException;
import java.util.ArrayList;
import java.util.List;
import java.util.Vector;
import javax.swing.*;

public class planGraphique extends JPanel{

    GrapheParListe graph;
    ArrayList stationSuccessive;

    public planGraphique(GrapheParListe graph, ArrayList
       stationSuccessive) throws IOException {
        this.graph = graph;
        this.stationSuccessive = stationSuccessive;
    }


    public void paintComponent(Graphics g) {
        super.paintComponent(g);

        try {
            int size = 30; //Taille des arrêts dans la fenêtre
               graphique

            //On crée la liste des noms des arrêts telle que :
            //listeNomArrets.get(0)->A1, ..get(1)->A2,...get(20)->C7 :
            ArrayList listeNomArrets = Methodes.listeNomArrets();

            //Le vector Q contient les objets "Arrets" de spécification
                (nom, coordx, coordy) :
            Vector Q = Methodes.listeArrets();

            Fichier graphe = new Fichier("kelkonk_bus.graph","R");
            GrapheParListe graph = new  GrapheParListe(graphe,
               listeNomArrets, Q);
            //création du graphe à partir du Fichier .graph et du
               vecteur des arrêts

            //on affiche les arêtes :

            int numArret1, numArret2, distance12;
            for(int i = 0; i <  graph.adj.length; i++){
                if(graph.adj[i]!=   null) {
                    numArret1 = i;
                    Liste a = graph.adj[i];
                    while(a!=null) {

                        numArret2 = a.num_noeud;
                        distance12 =  a.valeur;
                        String lignebus = a.ligne;
```

```
                    int x1 = ((Arret)Q.elementAt(numArret1)).coordX
                        ;
                    int y1 = ((Arret)Q.elementAt(numArret1)).coordY
                        ;
                    int x2 = ((Arret)Q.elementAt(numArret2)).coordX
                        ;
                    int y2 = ((Arret)Q.elementAt(numArret2)).coordY
                        ;

                    //Si les arrêts adjacents sont dans la liste
                        des sommets du trajet, on peint l'arrête en
                        rouge
                    if(stationSuccessive.contains(numArret1) &&
                        stationSuccessive.contains(numArret2))
                         g.setColor(Color.RED);
                    else //Sinon on la peint en noir.
                         g.setColor(Color.BLACK);
                    g.drawLine((int)(x1/4.5)+15,(int) (y1/4.5)+15,(
                        int)(x2/4.5)+15,(int) (y2/4.5)+15);
                    g.drawString(Integer.toString(distance12) + "("
                        +lignebus+")", (int)(((x1/4.5)+15 + (x2/4.5)
                        +15)/2), (int) (((y1/4.5)+15 +(y2/4.5)+15)
                        /2));

                    a = a.suivant;
                }
            }
        }

        //on affiche les sommets:
        for(int i=0; i<Q.size(); i++)
        {
            int x = ((Arret)Q.elementAt(i)).coordX;
            int y = ((Arret)Q.elementAt(i)).coordY;
            String nom = ((Arret)Q.elementAt(i)).nom;
            int numArret = listeNomArrets.indexOf(nom);

            //On peint le fond des arrêts
            g.setColor(new Color(220, 220, 220));
            g.fillOval((int)(x/4.5), (int)(y/4.5), size-1, size-1);

            if(stationSuccessive.contains(numArret))
            {
                g.setColor(Color.RED);
            }
            else
                g.setColor(Color.BLACK);

            //On peint le contour des arrêts et on y inscrit le nom
                .
            g.drawOval((int)(x/4.5), (int)(y/4.5), size, size);
            g.drawString(nom, (int)(x/4.5) + 8, (int)(y/4.5) + 20);
        }

    } catch (IOException e) {
```

```
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }
}
```

5.9 interfaceGraphique.java

```
import java.awt.BorderLayout;
import java.awt.EventQueue;

import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.border.EmptyBorder;
import javax.swing.JButton;
import java.awt.event.ActionListener;
import java.io.IOException;
import java.util.ArrayList;
import java.util.List;
import java.util.Vector;
import java.awt.event.ActionEvent;
import java.awt.Font;
import java.awt.Rectangle;

import javax.swing.JDesktopPane;
import javax.swing.JSeparator;
import java.awt.Button;
import java.awt.Color;
import javax.swing.JTextField;
import javax.swing.JInternalFrame;
import javax.swing.JList;
import javax.swing.JComboBox;
import javax.swing.JTextPane;
import javax.swing.JSplitPane;
import javax.swing.UIManager;

public class interfaceGraphique extends JFrame {

    private JPanel contentPane;
    planGraphique plan;
    ArrayList listeNomArrets = Methodes.listeNomArrets();
    Vector Q = Methodes.listeArrets();
    ArrayList stationSuccessive = new ArrayList();
    Fichier graphe = new Fichier("kelkonk_bus.graph","R");
    GrapheParListe g = new  GrapheParListe(graphe, listeNomArrets, Q);

    //Launch the application.

    public static void main(String[] args) {
        EventQueue.invokeLater(new Runnable() {
            public void run() {
                try {
                    interfaceGraphique frame = new interfaceGraphique()
                        ;
                    frame.setVisible(true);
                } catch (Exception e) {
```

```
                e.printStackTrace();
            }
        }
    });
}

//Frame :

public interfaceGraphique() throws IOException {
    setTitle("Itineraire BUS");
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    setBounds(100, 100, 1100, 600);
    contentPane = new JPanel();
    contentPane.setLocation(0, -22);
    contentPane.setBorder(new EmptyBorder(5, 5, 5, 5));
    setContentPane(contentPane);
    contentPane.setLayout(null);

    JPanel panel = new JPanel();
    panel.setBackground(new Color(245, 245, 245));
    panel.setBounds(10, 11, 261, 539);
    contentPane.add(panel);
    panel.setLayout(null);

    JTextPane txtpnDepart = new JTextPane();
    txtpnDepart.setBounds(34, 50, 51, 20);
    panel.add(txtpnDepart);
    txtpnDepart.setEditable(false);
    txtpnDepart.setOpaque(false);
    txtpnDepart.setText("D\u00E9part");

    JTextPane txtpnArrive = new JTextPane();
    txtpnArrive.setBounds(168, 50, 41, 20);
    panel.add(txtpnArrive);
    txtpnArrive.setEditable(false);
    txtpnArrive.setOpaque(false);
    txtpnArrive.setText("Arriv\u00E9e");

    JComboBox comboBoxDepart = new JComboBox();
    comboBoxDepart.setBounds(25, 71, 60, 28);
    for(int i=0; i<listeNomArrets.size(); i++)
    {
        comboBoxDepart.addItem(listeNomArrets.get(i));
    }
    panel.add(comboBoxDepart);

    JComboBox comboBoxArrive = new JComboBox();
    comboBoxArrive.setBounds(160, 71, 60, 28);
    for(int i=0; i<listeNomArrets.size(); i++)
    {
        comboBoxArrive.addItem(listeNomArrets.get(i));
    }
    panel.add(comboBoxArrive);

    JTextPane txtpnVotreItinraire = new JTextPane();
    txtpnVotreItinraire.setBounds(60, 11, 139, 28);
```

```
panel.add(txtpnVotreItinraire);
txtpnVotreItinraire.setOpaque(false);
txtpnVotreItinraire.setFont(new Font("Tahoma", Font.PLAIN, 18))
    ;
txtpnVotreItinraire.setText("Votre itin\u00E9raire : ");
txtpnVotreItinraire.setEditable(false);

JTextPane txtTrajet = new JTextPane();
txtTrajet.setBounds(25, 160, 215, 20);
txtTrajet.setOpaque(false);
panel.add(txtTrajet);

JTextPane txtDuree = new JTextPane();
txtDuree.setOpaque(false);
txtDuree.setBounds(25, 223, 215, 20);
panel.add(txtDuree);

JTextPane txtCorr = new JTextPane();
txtCorr.setOpaque(false);
txtCorr.setBounds(25, 284, 215, 20);
panel.add(txtCorr);

JButton btnRechercher = new JButton("Rechercher ");
btnRechercher.setBounds(73, 505, 113, 23);
panel.add(btnRechercher);
btnRechercher.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {

        //On récupère les données Arret source et Arret départ
            des combobox.
        String source =  (String) comboBoxDepart.
            getSelectedItem();
        String arrivee = (String) comboBoxArrive.
            getSelectedItem();

        //On convertit les données en entiers correspondants
            aux sommets en question.
        int provId = listeNomArrets.indexOf(source);
        int destId = listeNomArrets.indexOf(arrivee);

        //On calcule le plus court chemin et on récupère le
            trajet :
        Vector S = g.plusCourtChemin(provId);
        Vector trajetFinal = g.cheminOptimal(provId, destId, S)
            ;
        stationSuccessive = (ArrayList)trajetFinal.get(0);

        //On récupère l'affichage du trajet
        String trajetAffiche = Methodes.afficherTrajet(
            stationSuccessive, listeNomArrets);

        //On récupère les temps de trajets (en seconde et en
            minute)
        int temps = (int) trajetFinal.get(1);
        String tempsfinal = Integer.toString(temps);
        String tempsMinute = Methodes.tempsTrajetMinute(temps);
```

```
            //On récupère le nombre de correspondances
            int nombreCorresp = g.nombreCorrespondance(
               stationSuccessive);
            String nbCorresp = Integer.toString(nombreCorresp);

            //On affiche les données précédement récupérées :
            //Affichage du trajet:
            txtTrajet.setText(trajetAffiche);

            //Affichage du temps de trajet:
            if(source == arrivee)
                //Si le départ et l'arrivée sont les mêmes:
                txtDuree.setText("0s (Mêmes stations)");
            else
                txtDuree.setText(tempsfinal + "s  (" +tempsMinute+
                   ")");

            //Affichage du nombre de correspondances :
            txtCorr.setText(nbCorresp);

            //Suppression du plan puis réactualisation du nouveau
               plan :
            contentPane.remove(plan);
            planGraphique plan;
            try {
                plan = new planGraphique(g, stationSuccessive);
                plan.setBounds(291, 21, 783, 516);
                contentPane.add(plan);
                plan.repaint();
            } catch (IOException e1) {

                e1.printStackTrace();
            }

        }
    });
    btnRechercher.setBackground(new Color(220, 220, 220));

    JTextPane txtpnTrajet = new JTextPane();
    txtpnTrajet.setFont(new Font("Tahoma", Font.PLAIN, 16));
    txtpnTrajet.setEditable(false);
    txtpnTrajet.setText("Trajet :");
    txtpnTrajet.setBounds(23, 129, 186, 28);
    txtpnTrajet.setOpaque(false);
    panel.add(txtpnTrajet);

    JTextPane txtpnDuree = new JTextPane();
    txtpnDuree.setText("Dur\u00E9e :");
    txtpnDuree.setOpaque(false);
    txtpnDuree.setFont(new Font("Tahoma", Font.PLAIN, 16));
    txtpnDuree.setEditable(false);
    txtpnDuree.setBounds(25, 191, 186, 28);
    panel.add(txtpnDuree);

    JTextPane txtpnNbCorr = new JTextPane();
```

```
        txtpnNbCorr.setText("Nb de correspondances :");
        txtpnNbCorr.setOpaque(false);
        txtpnNbCorr.setFont(new Font("Tahoma", Font.PLAIN, 16));
        txtpnNbCorr.setEditable(false);
        txtpnNbCorr.setBounds(25, 254, 186, 28);
        panel.add(txtpnNbCorr);


        plan = new planGraphique(g, stationSuccessive);
        plan.setBounds(291, 21, 783, 516);
        contentPane.add(plan);
        plan.repaint();//

        JPanel panel_1 = new JPanel();
        panel_1.setBackground(new Color(245, 245, 245));
        panel_1.setBounds(281, 11, 893, 539);
        contentPane.add(panel_1);

    }
}
```

5.10 kelkonkbus.geo

```
#### position des stations sur un plan 2D correspondant au plan de l'
   énoncé
#### format: nom de station x   y
#### les champs sont séparés par une tabulation
A1  350 600
A2  700 600
A3  1000    950
A4  1300    1200
A5  1900    1200
A6  2300    850
A7  2600    1100
A8  2900    1550
B1  700 2000
B2  1100    1800
B3  1100    1550
B4  1600    700
B5  1900    300
B6  2600    300
C1  350 1000
C2  950 300
C3  1400    300
C4  2300    1300
C5  2300    1800
C6  1700    1800
C7  700 1400
```

5.11 kelkonkbus.graph

```
#### Ligne A direction A1->A8, codé par A-a
A-a:A1
A-a:A2
A-a:A3
A-a:A4
A-a:A5
```

```
A-a:A6
A-a:A7
A-a:A8

#### Ligne A direction A8->A1, codé par A-b
A-b:A8
A-b:A7
A-b:A6
A-b:A5
A-b:A4
A-b:A3
A-b:A2
A-b:A1

#### Ligne B direction B1->B6, codé par B-a
B-a:B1
B-a:B2
B-a:B3
B-a:A4
B-a:B4
B-a:B5
B-a:B6

#### Ligne B direction B6->B1, codé par B-b
B-b:B6
B-b:B5
B-b:B4
B-b:A4
B-b:B3
B-b:B2
B-b:B1

#### Ligne C direction sens de rotation de montre C1->A2 -> ... C7,
   codé par C-a
C-a:C1
C-a:A2
C-a:C2
C-a:C3
C-a:B5
C-a:A6
C-a:C4
C-a:C5
C-a:C6
C-a:B2
C-a:C7
C-a:C1

#### Ligne C direction C7->B2-> ... C1, codé par C-b
C-b:C1
C-b:C7
C-b:B2
C-b:C6
C-b:C5
C-b:C4
C-b:A6
C-b:B5
```

```
C-b:C3
C-b:C2
C-b:A2
C-b:C1
```