EC836 – Théorie des graphes
TP1 – Le réseau métropolitain de Springfield

Objectifs :

— Savoir reconnaître un problème de théorie des graphes ;
— Formaliser les graphes associés ;
— Programmer le graphe et l'algorithme de résolution du problème en Java.

---

1 Le problème initial

La ville de Springfield vient de se doter d'un réseau métropolitain moderne. La Régie Autonome des Transports de Springfield (RATS), en charge de ce réseau, souhaiterait mettre à disposition des citoyens une application permettant de naviguer dans le réseau, et surtout, de conseiller des itinéraires entre deux stations afin d'optimiser leur temps de transport. Vous êtes nommés à la tête de ce projet logiciel pour lequel vous devrez effectuer une livraison dans les plus brefs délais.

Pour vous aider, la RATS met à votre disposition le plan de son réseau métropolitain (voir fig. 2).

Ce dernier est annoté avec les temps de parcours en minutes entre chaque station.

2 Cas des correspondances instantanées

Pour cette première partie, nous supposons que les correspondances s'effectuent de manière instantanée. Pour raisonner, nous n'allons, pour le moment, considérer que les lignes 1-2-3-5-6-8 et R. Nous prendrons le cas d'un voyageur qui descend à Railroad station et qui souhaite se rendre à Convention center. Notre application devrait lui suggérer le chemin prenant le moins de temps, via le réseau métropolitain.

2.1 Premier raisonnement sur papier

Travail à réaliser :

1. Déterminer à quelle famille de problème de théorie des graphes correspond notre situation ;
2. Tracer un graphe simplifié (ne faisant apparaître que les stations avec correspondances pour les lignes considérées) ;
3. En appliquant l'un des algorithmes vus en cours, déterminer le chemin le plus rapide pour notre voyageur.

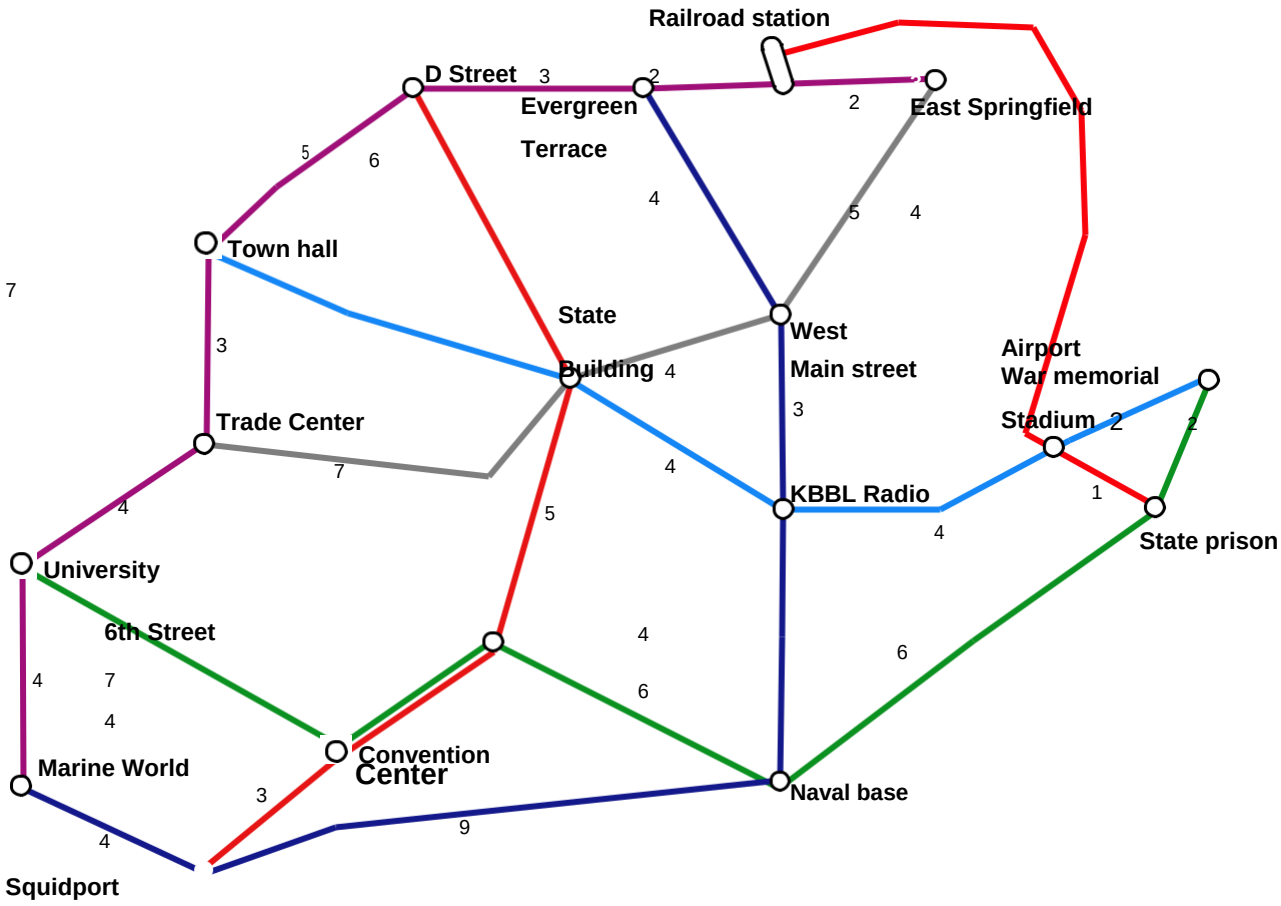

FIGURE 1 – Graphe ”simplifié” pour la première question

2.2 Résolution informatique

Une fois cette première étape réalisée, nous allons pouvoir passer à la réalisation informatique. Pour cette dernière, nous allons nous appuyer sur des classes fournies en exemple, et qui permettent de représenter un graphe dans un fichier, de le lire et de remplir une structure informatique représentant des graphes.
Parmi ces dernières, nous avons deux interfaces :

— IGraph décrit le comportement d’un graphe ;
— IVertex décrit le comportement d’un sommet.

Ainsi que 4 classes :

— Graph réalise le comportement d’un graphe et contient une collection de sommets ;
— Vertex réalise le comportement d’un sommet qui contient également sa liste d’adjacence ;
— GraphReader est une classe permettant de lire et des construire la structure d’un graphe à partir d’un fichier ;
— TestGraphReader est la classe principale du programme à laquelle on passe en paramètre le nom d’un fichier décrivant un graphe à charger.

Pour programmer votre algorithme de résolution, vous aurez essentiellement à modifier les classes Graph et Vertex.

Travail à réaliser

1. Déterminer quelles sont les informations à associer à vos sommets. Modifier la classe Vertex en conséquence ;
   Il faut penser à ajouter à la classe les membres suivants :
   — Une distance ;
   — Une référence au sommet d'où l'on vient ;
   — L'appartenance ou non à l'ensemble des sommets marqués.
2. Implémenter l'algorithme de résolution de votre programme dans la classe Graph
   — Il s'agit d'implémenter l'algorithme présenté en annexe ;
   — On peut séparer les différentes phases en autant de fonctions pour décomposer le déroulement de l'algorithme ;
   — La propagation peut être déléguée aux sommets.
   A noter, une annexe au TP présente la méthode pour implémenter l'algorithme de Dijkstra. (Voir à l'annexe A)
3. Vérifier que votre algorithme donne un résultat identique à ce que vous aviez déterminé à la main.

3 Prise en compte des correspondances

A présent, nous souhaitons prendre en compte les temps de correspondance. Dans notre cas, nous considérerons que le temps de correspondance est fixe et est de 2 minutes.

Travail à réaliser

1. Déterminer et indiquer quels types de changements vous devrez apporter à votre graphe pour prendre en compte les correspondances. Pour illustrer vos modifications, tracez votre graphe modifié pour uniquement les stations West Main Street, War memorial stadium, KBBL Radio, Evergreen terrace, Railroad station et East Springfield ;

FIGURE 2 – Plan du réseau métropolitain de Springfield

2. Modifier le fichier d'entrée de votre programme en conséquence et lancer l'analyse ;
3. Quel est, à présent, l'itinéraire le plus court en temps entre Railroad station et Convention Center ?

4 Finaliser l'application

Nous souhaitons incorporer une interface graphique pour conseiller les voyageurs. Cette interface devra permettre de :

— De choisir une station de départ ;
— De choisir une station d'arrivée ;
— Lorsque les choix ont été validés, proposer un itinéraire de voyage.

Pour la sélection des stations, vous pouvez envisager des listes déroulantes ou sélectionner la station directement sur le plan.

Travail à réaliser

1. Modifier votre programme en lui incorporant une interface graphique ;
2. Contrôler sa bonne exécution.

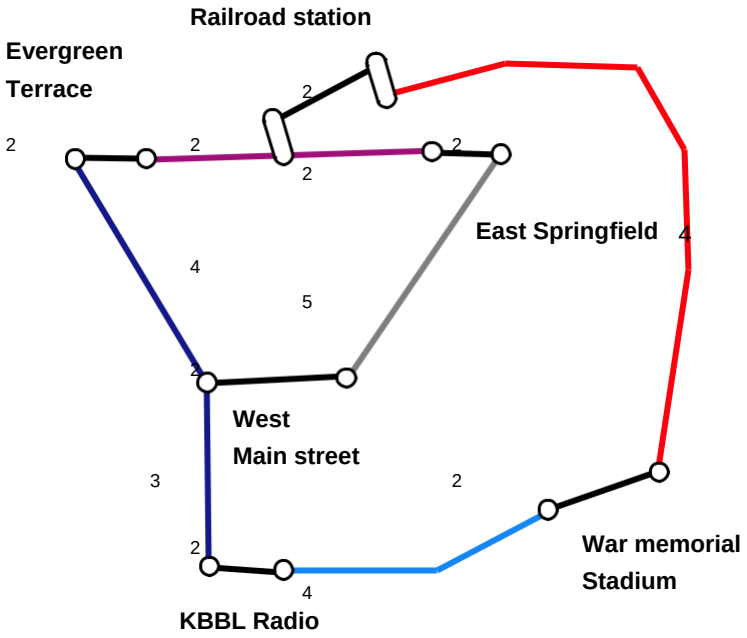

FIGURE 3 – Graphe modifié pour prendre en compte les correspondances

A Méthode pour programmer l'algorithme de Dijkstra

L'objectif est de décomposer le travail en un ensemble d'attributs et de méthodes à répartir entre les classes Graph et Vertex. A l'exception de la méthode principale qui implémente l'algorithme de Dijkstra, la plupart des méthodes auxiliaires font moins d'une dizaine de lignes de code.

A.1 Données à associer aux sommets

Pour exécuter l'algorithme de Dijkstra, nous avons besoin d'annoter les sommets avec un certain nombre d'informations qui sont :

— La distance du sommet au point de départ (distance, de type entier) ;

— Le sommet par lequel on est passé pour actualiser la distance (previous, de type Vertex) ;

— L'appartenance aux sommets marqués ou non (marked, de type booléen).

Il faut donc modifier la classe Vertex afin de bénéficier de ces attributs supplémentaires. En ce qui concerne les accesseurs, nous vous recommandons de ne les développer que s'ils sont nécessaires.

Il suffit d'ajouter, dans la classe Vertex, les déclarations suivantes :

```
Private     int distance ;
private     Vertex previous ;
private     boolean marked;
```

Remarque : Par la suite, le listing correspondant aux questions est simplement affiché à la suite des questions.

A.2 Méthodes auxiliaires préparatoires

Rechercher un sommet par son nom

Dans la classe Graph, les sommets sont stockés dans une table de hachage où ils sont recensés par le biais de leur identifiant. Toutefois, l'identifiant est différent du nom du sommet. Par exemple, pour les stations de métro de Springfield, la station nommée Town hall a pour identifiant A.

Le premier travail consiste à implémenter dans la classe Graph une méthode à la signature suivante : `Vertex findByName(String name)`. Elle fera le parcours de la collection de sommets (`vertices`), et retournera le sommet au nom correspondant à `name`. Si aucun sommet n'est trouvé, la méthode renverra alors la valeur **`null`**.

Remarque : en Java, comparer deux chaînes de caractères ne s'effectue pas avec l'opérateur == mais avec une méthode de la classe String qui est la méthode **`equals()`.**

```
private Vertex findByName ( String name )                {
        Iterator <Vertex > it = vertices.values( ) .iterato r ( ) ;
        while ( it.hasNext( )) {
                Vertex v = it.next( );
                if ( v.getName( ).equals(name))          {
                        return v ;
                }
        }
        return     null;
}
```

Remarque : Il existe d'autres manières de réaliser le parcours de la collection. Ainsi, le listing ci-dessous est également correct :

```
Private Vertex findByName ( String name )                {
        Enumeration <Vertex> en = vertices.elements( ) ;
        while ( en.hasMoreElements ( ) ) {
                Vertex v = en.nextElement ( ) ;
                i f ( v.getName ( ).equals ( name ) )    f
                        return v ;
                }
        }
        return null ;
}
```

Initialiser l'algorithme

Cette phase consiste à parcourir l'ensemble des sommets du graphe et à les initialiser. Pour la réaliser, nous nous proposons d'utiliser une combinaison de méthodes dans les classes Graph et Vertex.

Dans la classe Graph, vous développerez une méthode à la signature suivante : `void initDijkstra(String startName)`
qui parcourra les sommets de la collection et réalisera les actions suivantes :

— Pour chaque sommet, appel d'une méthode visant à le réinitialiser ;
— Si le sommet a pour nom `startName`, alors la distance associée au sommet sera initialisée à 0.

La classe Vertex possédera deux méthodes :

— `void reset()` qui retire le marquage, met la distance à l'infini (pour les valeurs entières, on utilisera la valeur `Integer.MAX_VALUE`, et indique que le sommet précédent est `null`
(pas de sommet precédent) ;
— `void setAsStart()` qui confère une distance 0 au sommet.

A.3 Fonctions et méthodes auxiliaires de la boucle principale de l'algorithme

Avant d'écrire la boucle principale de l'algorithme de Dijkstra, nous pouvons écrire quelques fonctions et méthodes secondaires que l'on appellera depuis la boucle principale.

Rechercher le sommet de poids minimal

Dans la classe Graph, on introduira une méthode à la signature ci-dessous `Vertex findMinimalVertex()` qui aura pour fonction de :

— Parcourir la collection de sommets ;
— Devra trouver le sommet à la distance minimale et le retourner.

Remarque : Cette façon de procéder implique l'ajout d'une méthode à la classe Vertex également (en réalité un accesseur pour retrouver la valeur de la distance).

Marquage des sommets

Dans la classe Vertex :

— ajouter une opération `mark()` qui permet de marquer le sommet ;
— ajouter une opération ayant pour signature `boolean isMarked()` permettant de déterminer si un sommet est marqué.
Ici, il s'agit de rajouter quelques accesseurs particuliers à la classe Vertex

Propagation des distances

Cette opération peut se réaliser sur chaque sommet. En effet, ces derniers connaissant la distance à laquelle se trouvent leurs successeurs, ils sont à même de modifier le calcul des distances.

Dans la classe Vertex, écrire une méthode qui aura pour signature `void propagate()` qui prendra en charge la propagation des distances. Cette méthode fera le parcours des successeurs du sommet considéré et mettra à jour les distances si ces dernières sont plus petites que la distance déjà associée à un successeur particulier.

Calculer et afficher le plus court chemin

Le plus court chemin est obtenu en remontant de prédécesseur en prédécesseur à partir du sommet d'arrivée. Toutefois, en faisant cela, on obtient le chemin à l'envers. Pour afficher le chemin dans le bon ordre, nous allons utiliser une pile (classe Stack en Java), empiler les prédécesseurs jusqu'à ce qu'il n'y ait plus de prédécesseur, puis nous allons les dépiler pour les afficher.
Dans la classe Vertex, écrire une méthode ayant la signature `void displayPath()` permettant de construire et d'afficher le chemin obtenu.

A.4 Ecriture de la fonction finale

Dans la classe Graph, écrire une méthode ayant la signature `void applyDijkstra(String startName, String endName)` permettant de calculer le plus court chemin entre un sommet de départ dont le nom est donné par le paramètre `startName` et un sommet d'arrivée dont le nom est donné par `endName`. Pour l'écriture de cette méthode, appuyez-vous le plus possible sur les fonctions auxiliaires développées précédemment.

A.5 Récapitulatif des méthodes à développer

La figure 4 donne une vue synthétique des méthodes à développer dans les classes Graph et Vertex. Les membres apparaissant en gras sont les membres que vous aurez ajouté au cours de votre processus de développement.

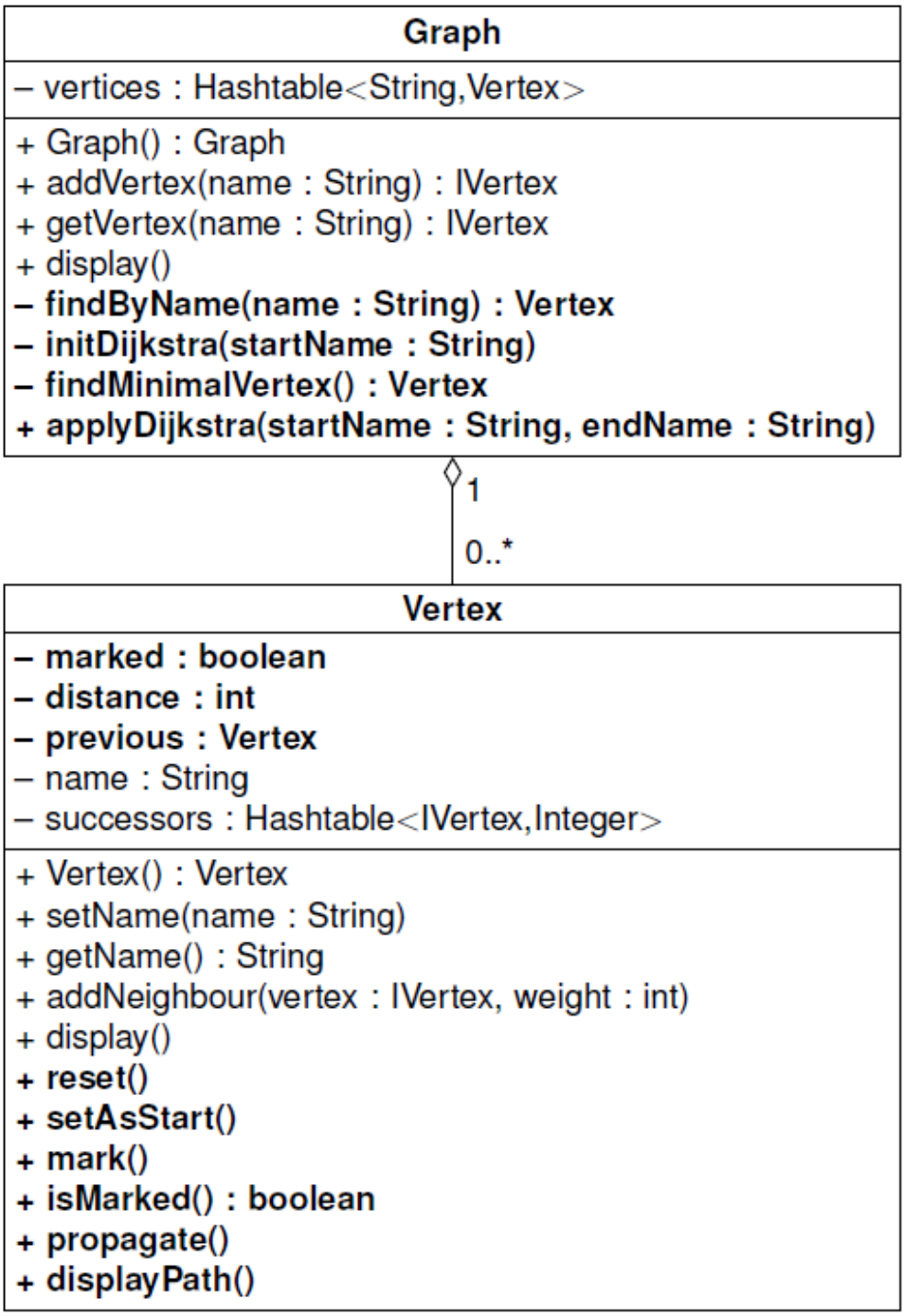

FIGURE 4 – Documentation des classes à compléter pour implémenter l'algorithme de Dijkstra