CALCUL DE PLUS COURTS CHEMINS DANS DES GRAPHES

Ce TP porte sur la manipulation de graphes orientés valués, en particulier l'implantation et la comparaison de différents algorithmes de calcul de plus courts chemins. L'application étudiée est celle d'un réseau de transports en commun.

Les principaux objectifs de ce TP sont :
- choisir une représentation d'un graphe adaptée à un problème donné, et l'implanter efficacement ;
- programmer différents algorithmes de calcul de plus courts chemins ;
- tester et comparer ces algorithmes.

D'un point de vue Programmation Orientée Objet en Java, les points importants sont :
- le respect des notions d'encapsulation sur toutes les classes écrites ;
- l'utilisation intensive de conteneurs Java, en justifiant systématiquement vos choix quant aux conteneurs utilisés ;
- éventuellement la mise en place du patron de conception *Stratégie*.

Une archive `.tar.gz` devra être rendue sur *Teide*, contenant :
- l'ensemble de vos fichiers sources, lisibles et commentés ;
- un rapport de **4 pages maximum, au format PDF**, contenant l'ensemble des choix de conception et leur justification, les tests effectués et l'analyse des résultats obtenus.

Le TP est à réaliser en **binôme**. La date de rendu est fixée au **mercredi 14 mars 2012**.

1 Graphe d'un réseau de transports en commun

La notion de graphe a été vue en Recherche Opérationnelle en période 1, et elle sera reprise dans le cours d'Algorithmique Avançée en début de 2ème année. Les définitions de base relatives aux graphes ne sont pas rappellées ici ; elles sont disponibles dans le polycopié de Recherche Opérationnelle qui vous a été distribué (également disponible sur le Kiosk).

Les lignes d'un service de transports en commun, par exemple la RATP ou la TAG, sont représentées sous forme de graphe. Les nœuds du graphe correspondent aux stations de métro/RER/tramway, tandis que les arcs connectent des stations voisines (c'est-à-dire joignables, sans arrêt intermédiaire, par un service de transport en commun). Pour une station donnée, il y a **autant de nœuds dans le graphe que de lignes passant par cette station**. Par exemple, trois nœuds correspondent à la station "Bastille" pour le graphe du métro parisien. Les nœuds correspondant à la même station seront reliés par des arcs, pour modéliser les correspondances entre lignes.

Les graphes étudiés dans ce TP sont des graphes **orientés** : en effet, bien que de nombreuses connexions entre stations voisines soient possibles dans les deux sens, certaines se font à sens unique (voir par exemple la liaison Chardon Lagache - Mirabeau de la ligne 10 du métro parisien). Lorsqu'une ligne relie deux stations dans les deux sens, deux arcs existent.

Enfin, les arcs des graphes sont ici valués d'un **coût positif ou nul** représentant le coût du transport du nœud source au nœud destination d'un arc. Ce coût a une interprétation qui varie selon le contexte : il peut correspondre à la durée du trajet ou au coût tarifaire, par exemple. Il peut être modulé en fonction du mode de transport : métro, bus, à pied (pour les stations comportant des correspondances).

Les nœuds des graphes étudiés sont munis des informations suivantes :
- un entier `id` identifiant de manière unique le nœud étudié ;
- une chaîne de caractères `station` indiquant le nom de la station.
  Cette chaîne n'identifie pas de manière unique un nœud, car tous les nœuds correspondant à la même station partagent la même chaîne de caractères.

Les arcs des graphes étudiés seront eux munis de quatre informations :
- un nœud `source` et un nœud `destination`
- une chaîne de caractères `ligne` représentant le numéro de la ligne (ou "0" pour une correspondance) ;
- un réel `cout` positif ou nul.

Documents fournis

Les documents fournis sont les suivants :
- un fichier `reseauMetroRERTramParis2009.graph` [1], décrivant le graphe du réseau parisien de métro, RER et tramway ;
- un fichier `reseauTramGrenoble2010.graph`, décrivant le graphe du réseau grenoblois de tramway ;
- le source `GenerateurGraphe.java` d'un programme permettant de créer des fichiers de graphe au format `.graph`. Ces graphes sont générés à partir du nombre de nœuds et d'arcs souhaités, en déterminant aléatoirement les arcs existants.

Description du format `.graph`

Les graphes fournis ou générés sont stockés dans des fichiers au format `.graph`. Un fichier `.graph` comporte, dans l'ordre :
- une ligne avec le nombre n de nœuds et le nombre m d'arcs du graphe ;
- n lignes avec, sur chacune, un entier identifiant de manière unique un nœud et une chaîne de caractères indiquant le nom de la station correspondant à ce nœud ;

1. *courtoisie* Michel Desvignes & Franck Hetroy

– m lignes avec, sur chacune, deux entiers indiquant les identifiants des nœuds source et destination d'un arc, un réel indiquant le coût associé à cet arc, et une chaîne de caractères indiquant le nom de la ligne de métro/RER/tramway associée (le nombre 0 est indiqué dans le cas d'une correspondance).

Pour information, dans `reseauMetroRERTramParis2009.graph` le coût d'un arc correspond à la distance euclidienne entre les stations (calculées à partir de leurs coordonnées GPS), sauf quelques cas particuliers :
– le coût d'une correspondance (même nom de station, mais changement de ligne) est de 120.0 ;
– le coût d'un trajet à pied (changement de ligne avec changement de nom de station) est de 240.0 ;
– le coût d'un trajet en tramway est de 40.0.

2 Algorithmes de plus court chemin

Trois algorithmes sont présentés pour calculer le plus court chemin depuis un nœud source vers *tous les autres nœuds du graphe*. Les versions proposées, écrites en pseudo-code, sont adaptées à des graphes orientés avec circuit (structures de marquage, pas de cycle infini).

2.1 Bellman-Ford

Très général, cet algorithme serait aussi adapté à la présence de coûts négatifs sur certains arcs (hors propos ici). L'algorithme peut aussi être optimisé en arrêtant la boucle principale dès que plus aucune distance n'est modifiée lors d'une itération.

Algorithm 1 Bellman-Ford

```
for i ← 1 to n do
    distance(i) ← +∞
end for
distance(source) ← 0
for i ← 2 to n do
    for each arc(u, v) do                    ▷ pour chaque arc du graphe
        d ← distance(u) + cout(u,v)
        if d < distance(v) then
            distance(v) ← d
            predecesseur(v) ← u
        end if
    end for
end for
```

2.2 Dijkstra

L'algorithme de Dijkstra utilise une *file de priorité* des nœuds à parcourir : à chaque étape, le parcours se poursuit à partir du nœud le plus proche de la source.

Algorithm 2 Dijkstra

```
for i ← 1 to n do
    distance(i) ← +∞
end for
distance(source) ← 0
INSERER(source,file_attente)
while not EST_VIDE(file_attente) do
    u ← NOEUD_DE_DISTANCE_MINIMUM(file_attente)
    SUPPRIMER(u, file_attente)
    for each arc(u,v) do                          ▷ pour chaque arc issu de u
        d ← distance(u) + cout(u,v)
        if d < distance(v) then
            distance(v) ← d
            predecesseur(v) ← u
            METTRE_A_JOUR_OU_INSERER(v, file_attente)
        end if
    end for
end while
```

2.3 Exploration arborescente

Une autre possibilité est de parcourir le graphe à partir du nœud source à la manière d'un arbre, par exemple en profondeur. Pour affiner cet algorithme, une heuristique peut être ajoutée sur l'ordre de parcours des successeurs `v` du nœud `u`.

3 Travail demandé

3.1 Représentation du graphe

Les seules spécifications portant sur le graphe sont les suivantes :

- un graphe sera chargé à partir d'un fichier au format `.graph` décrit ci-dessus. Il doit pouvoir être affiché en totalité, par exemple ligne par ligne.
- les méthodes obligatoires sont :
  - `public void afficheLigne(String ligne)` : affiche toutes les stations d'une ligne, d'un terminus à l'autre. Par exemple :

Algorithm 3 Exploration arborescente

```
for i ← 1 to n do
    distance(i) ← +∞
end for
distance(source) ← 0
PARCOURS_PROF(source)

procedure PARCOURS_PROF(u)
    for each arc(u,v) do                          ▷ pour chaque arc issu de u
        d ← distance(u) + cout(u,v)
        if d < distance(v) then
            distance(v) ← d
            predecesseur(v) ← u
            PARCOURS_PROF(v)
        end if
    end for
end procedure
```

```
Ligne 4: Porte de Clignancourt, Simplon, ..., Alésia, Porte d'Orléans
(26 stations)
```

- `public void afficheCorrespondances(String station)` : affiche les lignes passant par une station donnée.
- `public void affichePlusCourtChemin(String stationDepart, String stationArrive)` : affiche l'itinéraire le plus court entre deux stations. Par exemple :

```
Itinéraire le plus court entre <Sentier> et <Glacière> :
Ligne  3: de <Sentier> a <République>  (4 stations)
 => Correspondance
Ligne  5: de <République> a <Place d'Italie>  (9 stations)
 => Correspondance
Ligne  6: de <Place d'Italie> a <Glacière>  (2 stations)
Cout total: 311.3154
```

  Si plusieurs itinéraires sont de coût identique, le premier trouvé sera retenu. (optimisation : celui avec le moins de correspondances ?)
- bien entendu d'autres méthodes pourront être ajoutées, selon vos besoins !

Le choix d'une structure adéquate pour la représentation du graphe est entièrement libre. Elle peut reposer sur toute structure de donnée, usuelle (matrice d'adjacence, liste de successeurs, ...) ou non. Certaines informations pourront être stockées ou bien calculées à la demande. Le point important est de **justifier vos choix** en fonction des opérations nécessaires sur le graphe et de leurs caractéristiques (coûts, compromis coûts/mémoire, fréquence d'utilisation des opérations, caractère critique ou non, ...).

3.2 Algorithmes

Les trois algorithmes décrits (de nombreuses optimisations sont possibles, mais non demandées dans ce TP) devront être implantés.

Dans un premier temps au moins, ils peuvent l'être directement dans des méthodes de la classe `Graphe`. Si vous le souhaitez, une approche plus "orientée objet" peut aussi être utilisée via l'utilisation du patron de conception *Stratégie*. Dans ce cas, les algorithmes seront délégués à une hiérarchie de classe, et pourront être sélectionnés dynamiquement.

3.3 Tests : validation & comparaisons

La première partie des tests a pour but de *valider* les opérations sur le graphe :
- plus courts chemins calculés avec les différents algorithmes
- autres opérations : affichage du graphe, des lignes, ...

Expliciter les tests effectués, sur les graphes fournis ou éventuellement d'autres que vous aurez pu générer vous même, pour valider vos méthodes.

La seconde partie des tests vise à *comparer* l'efficacité des différents algorithmes de calcul de plus courts chemins.
- Tester et comparer les temps de calcul sur les graphes fournis puis sur d'autres graphes plus importants générés aléatoirement.
- Donner la complexité théorique (pire et meilleur cas) des trois algorithmes, en fonction du nombre de nœud n et du nombre d'arcs m dans le graphe. Quel méthode est théoriquement la plus rapide ?
- Discuter les résultats de vos tests : la pratique est-elle conforme à la théorie ? Comment l'expliquez-vous ? Qu'en pensez-vous ?
- Conclure : que pensez-vous de ces algorithmes ? Avantages et inconvénients ?

Tous les tests devront être écrits dans un package spécifique.

Tout y est. Bon travail à vous tous !