SAÉ 2.02 - Exploration algorithmique d'un problème

Binôme : ATTENOT / RODRIGUES NETO
Semestre 2 - BUT Informatique

1. Représentation d'un graphe

Objectif

Dans cette première partie, l'objectif était de représenter un graphe orienté pondéré permettant de modéliser un réseau de stations de métro. Nous devions construire une structure de graphe efficace, réutilisable par les algorithmes de recherche de chemins, en utilisant une structure adaptée comme la liste d'adjacence.

Travail réalisé

Nous avons commencé par définir une interface `Graphe` avec les méthodes suivantes :

```
public interface Graphe {
    List<String> listeNoeuds();
    List<Arc> suivants(String n);
}
```

Ensuite, nous avons codé les classes :

- `Arc` : pour représenter un arc orienté avec destination, coût, et ligne.
- `GrapheListe` : pour gérer les nœuds et arcs via une map associant chaque sommet à sa liste d'arcs sortants.

Méthodes ajoutées

- `ajouterNoeud(String)` : pour ajouter un sommet au graphe
- `ajouterArc(String, String, double, String)` : pour créer un arc entre deux stations
- `toString()` : pour visualiser facilement le graphe

Code significatif

```
public void ajouterArc(String depart, String dest, double cout, String
ligne) {
    ajouterNoeud(depart);
    ajouterNoeud(dest);
    adjacence.get(depart).add(new Arc(dest, cout, ligne));
}
```

Tests effectués

Nous avons vérifié notre implémentation avec de petits graphes manuels avant d'utiliser le fichier réel `metro-paris.txt`.

[Question 3]

Nous avons utilisé une structure `Map<String, List<Arc>>` pour que la recherche de voisins soit efficace, et que l'ajout d'un arc soit simple à implémenter.

Difficultés rencontrées

Le traitement du fichier de données a été la principale difficulté : il fallait faire correspondre les ID numériques des stations à leurs noms, et repérer les sections du fichier correctement. Nous avons corrigé cela en créant une fonction robuste dans la classe `LireReseau`, capable d'identifier les en-têtes même si la casse ou les espaces changeaient.

2. Point fixe (Bellman-Ford)

Objectif

Cette partie portait sur l'implémentation de l'algorithme de Bellman-Ford, fondé sur le principe de propagation de valeurs jusqu'à stabilisation (point fixe). Il permet de trouver le plus court chemin même avec des poids négatifs (ce que nous n'avions pas ici).

Travail réalisé

Nous avons créé deux classes :

- `Valeurs` pour stocker les distances, parents et lignes
- `BellmanFord` pour résoudre le problème avec et sans pénalité de changement de ligne

Méthodes importantes

- `resoudre(Graphe g, String depart)` : version classique
- `resoudre2(Graphe g, String depart)` : version avec pénalité
- `calculerChemin(String destination)` dans `Valeurs` pour remonter les parents

Code significatif

```
if (newDist < v.getValeur(w)) {
    v.setValeur(w, newDist);
    v.setParent(w, u);
    v.setLigne(w, ligneArc);
}
```

Tests effectués

Nous avons d'abord utilisé un petit graphe (A → B → C → D) pour vérifier le bon fonctionnement. Ensuite, nous avons chargé le fichier `metro-paris.txt` pour tester sur un vrai réseau de transport.

[Question 11]

Bellman-Ford est plus long que Dijkstra en moyenne car il traite tous les arcs à chaque itération, alors que Dijkstra ne parcourt que les sommets utiles.

Difficultés rencontrées

Au début, les chemins retournés étaient vides. Cela venait du fait que certaines stations n'étaient pas correctement ajoutées au graphe (oubli d'appel à `ajouterNoeud`). Une fois corrigé, les chemins sont apparus normalement.

3. Dijkstra (glouton)

Objectif

L'objectif ici était de coder l'algorithme glouton de Dijkstra, qui à chaque étape choisit le sommet ayant la distance minimale connue et met à jour ses voisins. Cet algorithme est plus rapide que Bellman-Ford dans les graphes sans poids négatifs.

Travail réalisé

Nous avons implémenté deux méthodes :

- `resoudre()` : version classique de Dijkstra
- `resoudre2()` : version avec pénalité de changement de ligne

Méthode ajoutée

```
private String trouverNoeudValeurMinimale(List<String> Q, Valeurs valeurs)
```

Elle sélectionne le sommet de `Q` avec la distance la plus faible.

Code significatif

```
if (d < valeurs.getValeur(v)) {
    valeurs.setValeur(v, d);
    valeurs.setParent(v, u);
    valeurs.setLigne(v, arc.getLigne());
}
```

Tests effectués

Même logique de test que Bellman-Ford : petits graphes d'abord, puis `metro-paris.txt`. On a aussi comparé les résultats entre les deux versions pour chaque trajet.

[Question 14]

Dijkstra est plus rapide car il ne considère que les sommets utiles et évite de répéter inutilement les relaxations.

Difficultés rencontrées

La logique de pénalité entre lignes a nécessité de bien stocker les lignes dans `Valeurs`. Nous avons aussi dû bien gérer la file Q et éviter d'y propager des nœuds déjà traités.

4. Validation

Objectif

Nous devions tester nos algorithmes sur un cas réel : le métro parisien. Pour cela, nous avons analysé plusieurs trajets réels et mesuré les performances.

Travail réalisé

Nous avons construit une classe `MainMetro` avec un tableau de trajets :

```
String[][] trajets = {
    {"Châtelet", "Nation"},
    {"Porte de Clignancourt", "Gare de Lyon"},
    {"Place d'Italie", "République"},
    {"Odéon", "Gare du Nord"},
    {"Montparnasse Bienvenue", "Bastille"}
};
```

Chaque trajet est testé avec :

- BellmanFord (simple + pénalité)
- Dijkstra (simple + pénalité)

Ce qui a été affiché

- Chemin trouvé (via `calculerChemin()`)
- Temps de traitement (via `System.nanoTime()`)

Résultats observés

Les chemins étaient vides tant que les stations n'étaient pas bien chargées. Une fois corrigé, nous avons obtenu des chemins cohérents. Dijkstra est toujours plus rapide.

[Question 21 - 22]

Les différences de chemin avec et sans pénalité étaient visibles dans les trajets ayant un ou plusieurs changements de ligne. L'algorithme sans pénalité minimise la durée uniquement, tandis que la version 2 tend à privilégier les lignes continues.

5. Comparaison avec la RATP

[Question 23]

Nous avons comparé les trajets proposés par nos algorithmes à ceux du site de la RATP (ou IDFM). Dans de nombreux cas, les résultats sont similaires. Parfois, notre version avec pénalité privilégie un trajet plus long mais avec moins de correspondances.

Exemple :

- Châtelet → Nation : Ligne 1 directe (identique RATP)
- Montparnasse → Bastille : notre version propose un trajet sans changement, mais plus long

Ces différences sont logiques car la RATP optimise souvent pour le temps, tandis que notre version pénalisée peut favoriser le confort.

6. Conclusion générale

Cette SAE a été très formatrice sur plusieurs plans. Elle nous a permis de :

Ce que nous avons appris

- Manipuler les structures de graphes et les algorithmes associés
- Lire et analyser des fichiers de données complexes
- Implémenter des algorithmes classiques dans un contexte réel

Difficultés rencontrées

- Lecture du fichier `metro-paris.txt` : erreurs de tabulations et détection des sections
- Reconstruction de chemins vides à cause de noms incorrects ou nœuds absents
- Mise en place de la pénalité de ligne dans les algorithmes

Bilan

Nous avons produit tout le code demandé, testé sur des cas simples et réels, et documenté les écarts observés. Le rapport que nous rendons aujourd’hui est le fruit d’une itération entre développement, tests, et corrections, qui nous a permis de bien comprendre les forces et faiblesses de chaque algorithme.