Le Bellman-Prim Express

Elias Khallouk

Wissam Shaaban

Yiannis Leblanc

Sommaire

Introduction

Ce projet met en œuvre, en Python, une application d'exploration du graphe du métro parisien (1998–2002). Il a trois objectifs principaux : (1) charger et représenter un graphe réel issu d'un fichier d'échange, (2) proposer un service d'interrogation via une API + interface web légère, et (3) implémenter et visualiser deux algorithmes fondamentaux de théorie des graphes — le plus court chemin (Bellman–Ford) et l'arbre couvrant de poids minimum (Prim). Le code fournit une base claire, documentée et exécutable localement.

I/ Expérience utilisateur

A- Lancer le programme

L'application peut être exécutée de deux manières :

1. Via Python directement,
2. Via le script `run.sh`, dédié aux systèmes Unix-like (Linux, macOS).

1. Lancement classique via Python

Prérequis : Python 3.x (3.10+ conseillé) et les dépendances listées dans requirements.txt.

1. (optionnel) créer et activer un environnement virtuel :

```
python -m venv .venv

source .venv/bin/activate
```

2. Installer les dépendances :

```
pip install -r requirements.txt
```

3. Démarrer le serveur Flask :

```
python run.py
```

4. Ouvrir un navigateur sur : http://127.0.0.1:5000/

Le serveur expose des endpoints API pour diagnostics et requêtes (par ex. `/health, /stations, /path?start=<id>&end=<id>`). L'interface web principale (`static/index.html`) offre un canevas de visualisation et des contrôles simples.

2. Lancement via `run.sh` (Unix-like uniquement)

Un script d'automatisation (`run.sh`) est fourni à la racine du projet.

Il effectue automatiquement :

- la création (si nécessaire) et l'activation d'un environnement virtuel,
- l'installation des dépendances,
- puis le lancement du serveur Flask.

Exécution :

```
chmod +x run.sh

./run.sh
```

Ce script repose sur des fonctionnalités spécifiques aux environnements Unix-like (bash, gestion des permissions, structure du système), il n'est donc pas compatible Windows sans adaptation préalable.

B- Rechercher le chemin le plus court

Interface :

- Deux menus déroulants (départ / arrivée), triés par identifiant de station.
- Un bouton « Calculer le plus court chemin ».

Comportement :

- Lorsque l'utilisateur lance le calcul, l'API `/path?start=<id>&end=<id>` retourne un JSON contenant le temps total (en secondes), la séquence d'identifiants et la liste des noms de stations sur l'itinéraire.
- L'interface affiche :
    - Le détail textuel de l'itinéraire (liste ordonnée de stations, durée approximative en minutes),
    - La représentation graphique sur le canevas : toutes les stations connues (points) et l'itinéraire surligné (couleur distinctive).

Exemple de réponse API (illustratif) :

```
{
```

```
  "total_time_seconds": 200,

  "path": [0, 238, 12, 10],

  "stations": ["Abbesses", "Pigalle", "Châtelet", "Auber"]

}
```

C- Tracer l’ACPM

Interface :

- Un bouton ou contrôle « Calculer l’ACPM » sur la carte.

Comportement :

- L’application exécute Prim sur le graphe (en considérant les poids = temps de parcours) et retourne l’ensemble d’arêtes de l’ACPM et son poids total.
- L’ACPM est affiché sur le canevas (arêtes mises en évidence) et le poids total (somme des temps) est présenté à l’écran.
- L’affichage permet d’identifier visuellement la structure de connexion minimale reliant toutes les stations présentes.

II/ Fonctionnement du programme

A- Descriptions des classes et structures de données utilisées

Le modèle de données suit une représentation classique adaptée aux algorithmes demandés.

Principales entités :

- **Station** (classe ou dict) :
    - id : identifiant entier unique,
    - name : nom (chaîne, espaces normalisés),
    - x, y : coordonnées (optionnelles — renseignées si présentes dans pospoints.txt),
    - autres métadonnées éventuelles (terminus, branche).
- **Edge / Arête** :
    - paire d'identifiants (u, v) et weight (temps en secondes).
- **Graphe** :
    - représentation en liste d'adjacence : pour chaque sommet, liste de voisins (voisin_id, poids) — structure choisie pour l'efficacité des parcours et de l'itération sur voisins.
    - stockage supplémentaire d'un tableau d'arêtes [(u, v, w)] pour l'exécution de Prim ou de procédures qui souhaitent parcourir les arêtes.

Justification :

- La liste d'adjacence est adaptée à des graphes sparses comme le métro. Elle permet BFS/DFS en O(V+E), et itérations efficaces pour Bellman–Ford et Prim.
- Un dictionnaire stations_by_id facilite la conversion ID ↔ métadonnées (nom, coordonnées).

B- Chargement de fichiers

Fichiers sources :

- Data/metro.txt : format original (lignes V pour sommets, E pour arêtes). Les lignes V définissent les stations ; les lignes E définissent des arrêtes bidirectionnelles avec temps en secondes.
- Data/pospoints.txt : positions approximatives des stations ; les noms y utilisent @ comme séparateur d'espaces et sont normalisés au chargement.

Procédure de parsing :

1. Parsing des stations : lecture ligne par ligne ; les lignes commençant par V sont décodées pour extraire id, name, flag terminus éventuel et autres champs. Les doublons apparents (stations avec même nom mais identifiants différents) sont conservés tels qu'ils figurent dans le fichier d'origine — ceci reflète la réalité de la base de données fournie.
2. Parsing des arêtes : pour chaque ligne E, extraire u, v, weight et ajouter dans la liste d'adjacence :

```
adj[u].append((v, weight))

adj[v].append((u, weight))
```

3. Parsing des positions : associer coordonnées x,y aux stations correspondantes en normalisant les noms (@ → espace). Toutes les stations sans coordonnées restent traçables uniquement par leur présence dans la liste.
4. Vérification de cohérence : détecter références d'ID inexistants et les signaler.

Remarques pratiques :

- Les noms normalisés facilitent la recherche par nom.
- Les correspondances (transferts) sont modélisées comme des arêtes ordinaires (poids = temps de transfert) conformément aux hypothèses du projet.

C- Implémentation des algorithmes

3. Vérification de la connectivité

Objectif : s'assurer que le graphe est connexe (accessible de n'importe quel sommet vers n'importe quel autre).

Algorithme utilisé :

- Méthode : parcours en profondeur (DFS) ou largeur (BFS) à partir d'un sommet arbitraire ; compter le nombre de sommets atteints.
- Si le nombre de sommets atteints < |V|, le graphe n'est pas connexe. Dans ce cas, le module signale les composantes connexes isolées et propose (dans le rapport) d'ajouter les liaisons manquantes identifiées (ou, côté travaux pratiques, l'équipe ajoute manuellement des arêtes justificatives).

Justification :

- BFS/DFS sont optimaux pour tester la connectivité en O(V + E) et faciles à expliquer dans le rapport.

4. Plus court chemin : Bellman–Ford

Objectif : minimiser la somme des temps entre deux stations (poids non négatifs).

Implémentation :

- Adaptation classique de l'algorithme de Bellman–Ford :
  - Initialisation : `dist[start] = 0, dist[others] = +inf, pred[.] = None`.
  - Relaxation : répéter |V|-1 fois : pour chaque arête (u, v, w), si `dist[u] + w < dist[v]` alors mettre à jour `dist[v] et pred[v] = u`, et réciproquement (graphe non orienté).
  - Optionnel : détection de cycles de poids négatif (non applicable ici car poids = temps ≥ 0).
- Reconstruction du chemin : partant de end, suivre pred jusqu'à start.
- Complexité : O(V*E) — acceptable pour la taille du graphe métro fourni.

Remarque sur l'usage de `heapq` :

- Le README mentionne l'option d'un `heapq` ; traditionnellement Dijkstra utilise un tas tandis que Bellman–Ford n'en a pas besoin. Dans notre implémentation principale nous utilisons bellman–ford pour respecter le cahier des charges ; si l'on souhaite optimiser pour poids non négatifs, une variante basée sur Dijkstra (avec `heapq`) est envisageable et peut être proposée en extension.

5. Arbre couvrant de poids minimum : Prim

Objectif : calculer l'ACPM en considérant les poids (temps).

Implémentation :

- Prim avec tas (complexité O(E log V)) :
    - Démarrer sur un sommet arbitraire, pousser ses arêtes incidentes dans un tas prioritaire trié par poids.
    - Extraire l'arête minimale reliant un sommet marqué à un sommet non marqué ; l'ajouter à l'ACPM ; marquer le nouveau sommet ; ajouter ses arêtes incidentes au tas ; répéter jusqu'à couvrir tous les sommets.
- Résultat : liste d'arêtes de l'ACPM et poids total (somme des poids des arêtes sélectionnées).

Justification :

- Prim est un algorithme standard, intuitif pour la visualisation (on voit l'arbre se construire) et adapté à la représentation par liste d'adjacence + structure de tas.

Conclusion

Le livrable fourni est un prototype satisfaisant les exigences du sujet « Le Bellman–Prim Express » : il charge les données historiques du métro, assure une représentation exploitable, propose des API simples et une interface web minimale pour interroger et visualiser les résultats des algorithmes. Les choix techniques (liste d'adjacence, Bellman–Ford, Prim) sont motivés par la clarté, la conformité au sujet et la facilité d'explication dans le rapport. Des extensions naturelles (optimisation via Dijkstra pour poids non négatifs, pénalités de correspondance, filtrage par ligne, mise en cache d'itinéraires fréquents, tests unitaires) sont identifiées et peuvent être implémentées ultérieurement.