Rapport de projet – Algorithmes et théorie des Graphes

Classe : I1 APP LSI Groupe 1

Promotion : 2028

Sarah ARNAUD – Alexandre POISSONNEAU

Table des matières

1. Introduction

Ce projet a pour objectif de mettre en œuvre et de comparer plusieurs **algorithmes de graphes** : BFS, DFS, Dijkstra, Kruskal, Prim, Bellman-Ford et Floyd-Warshall.
Une **interface interactive** permet de visualiser les parcours et les résultats pour certains d'entre eux.

Le graphe choisi représente un **réseau de villes françaises** reliées par des routes pondérées.
L'application a été développée en **Python** pour la logique algorithmique et en **HTML/JavaScript** pour la visualisation dynamique.

2. Explication théorique des algorithmes

2.1 Parcours en largeur (BFS – *Breadth-First Search*)

Le BFS explore un graphe **couche par couche** à partir d'un nœud source, en utilisant une **file (queue)**.
Il garantit le plus court chemin en **nombre d'arêtes** pour les graphes non pondérés.

Complexité :

- Temps : O(V + E)
- Mémoire : O(V)

2.2 Parcours en profondeur (DFS – *Depth-First Search*)

Le DFS explore le graphe en **profondeur** avant de revenir en arrière.
Il utilise une **pile (stack)** ou la **récursion**.

Principe :

1. On part du sommet de départ.
2. On visite récursivement chaque voisin non visité avant de revenir en arrière.
3. L'ordre obtenu dépend du chemin de récursion.

Complexité :

- Temps : O(V + E)
- Mémoire : O(V)

Le DFS a été implémenté dans le projet Python, mais **non intégré à la version web** actuelle.

2.3 Algorithme de Dijkstra

Dijkstra calcule le **plus court chemin pondéré** depuis un nœud source vers tous les autres.
Il fonctionne uniquement avec des poids **positifs**.

Principe :

- On choisit le sommet non visité le plus proche.
- On met à jour les distances de ses voisins.
- On répète jusqu'à avoir visité tous les sommets.

Complexité :

- O((V + E) log V) avec une file de priorité.

Implémenté et visualisable dans la version web.

2.4 Algorithme de Kruskal (Arbre couvrant minimal)

Kruskal construit un **arbre couvrant minimal (MST)** en ajoutant progressivement les arêtes les plus légères sans créer de cycle, à l'aide d'une **structure Union-Find (DSU)**.

Complexité :

- O(E log E)

Implémenté et visualisable dans la version web.

2.5 Algorithme de Prim (Arbre couvrant minimal)

Prim est une alternative à Kruskal.
Il part d'un sommet initial et ajoute progressivement l'arête la plus légère connectant un nouveau sommet à l'arbre déjà construit.

Complexité :

- O(E log V) avec une file de priorité.

mplémenté dans la version Python, **non intégré à l'interface web**.

2.6 Algorithme de Bellman-Ford

Bellman-Ford calcule les plus courts chemins depuis une source, **même en présence de poids négatifs**, tant qu'il n'existe pas de cycle négatif.

Principe :

- On met à jour toutes les arêtes **V-1 fois**.
- Si une distance peut encore être améliorée à la V^e itération, il y a un **cycle négatif**.

Complexité :

- O(V × E)

Implémenté côté Python uniquement, **non visualisé en web**.

2.7 Algorithme de Floyd-Warshall

Floyd-Warshall trouve les **plus courts chemins entre toutes les paires de sommets**.
Il repose sur une programmation dynamique :
on considère progressivement tous les nœuds comme points intermédiaires.

Complexité :

- $O(V^3)$

Implémenté dans le code Python, **non disponible dans la version web** (trop lourd pour le navigateur).

3. Description de l'implémentation et choix techniques

3.1 Structure de données

Le graphe est représenté par une **liste d'adjacence**, stockée dans la classe Graph :

```
class Graph:
  def __init__(self, directed=False):
    self.directed = directed
    self.nodes = {}
    self.adj = {}
```

Chaque arête est stockée comme un tuple (voisin, poids).

Ce choix est optimal pour les graphes clairsemés, où le nombre d'arêtes est bien inférieur à V^2.

3.2 Organisation du code

Le projet est organisé de manière modulaire :

/ParcoursArbres

├— graph.py

├— algo_bfs.py

├— algo_dfs.py

├— algo_dijkstra.py

├— algo_kruskal.py

├— algo_prim.py

├— algo_bellman_ford.py

├— algo_floyd_warshall.py

├— main.py

├— gui_html.py

Chaque fichier correspond à un algorithme ou à une structure de donnée spécifique.

3.3 Interface web interactive

L'interface graphique repose sur un export HTML/JS via **Vis.js**.
Elle offre :

- des **boutons** pour exécuter les algorithmes ;
- une **visualisation dynamique** des arêtes (rouge pour Dijkstra, vert pour Kruskal, orange pour BFS) ;
- une interaction fluide, sans dépendance locale.

Exemple de boutons :

<button onclick="runBFS()">BFS (Largeur)</button>

<button onclick="runDijkstra()">Dijkstra</button>

<button onclick="runKruskal()">Kruskal (MST)</button>

Les résultats calculés par Python (chemins, coûts) sont insérés dans le fichier HTML sous forme de données JSON.

3.4 Choix techniques

- **Langage principal :** Python 3.11
- **Affichage :** Vis.js (JavaScript côté client)
- **Structures :** dictionnaires, deque, heapq, Union-Find
- **Pas de dépendances externes lourdes** (aucun framework web ni Tkinter).

4. Résultats expérimentaux et interprétation

4.1 Graphe de test

Le graphe représente dix villes françaises connectées par des routes pondérées.

Ville A	Ville B	Distance
Rennes	Paris	110
Paris	Dijon	60
Lyon	Grenoble	40
Dijon	Nancy	75
Caen	Lille	65
Nantes	Bordeaux	90

4.2 Exécution et visualisation

- **BFS (depuis Paris)** :
  Paris → Caen → Rennes → Dijon → Lille → Nantes → Nancy → Lyon → Grenoble → Bordeaux
  🟠 Arêtes surlignées en orange.
- **Dijkstra (Paris → Grenoble)** :
  Paris → Dijon → Lyon → Grenoble
  🔴 Chemin optimal (distance = 140 unités).

- **Kruskal (MST)** :
  🟢 Relie toutes les villes avec un coût minimal de 640 unités.

4.3 Algorithmes non visualisés mais testés

Algorithme	Résumé	Statut
DFS	Parcours en profondeur (utile pour détection de cycles)	Implémenté, non visualisé
Prim	Variante de Kruskal pour MST	Implémenté, non visualisé
Bellman-Ford	Gère les poids négatifs	Implémenté, non visualisé
Floyd-Warshall	Chemins entre toutes les paires	Implémenté, non visualisé

Ces algorithmes ont été **testés dans le code Python**, mais **non intégrés à l'interface HTML** afin de conserver une expérience fluide et éviter les calculs coûteux côté navigateur.

4.4 Interprétation

- BFS et DFS servent à explorer la structure du graphe.
- Dijkstra et Bellman-Ford permettent des **optimisations de trajets**.
- Kruskal et Prim minimisent le **coût global de connexion**.
- Floyd-Warshall généralise la recherche de chemins à l'ensemble du graphe.

La visualisation web (Vis.js) s'est révélée idéale pour **illustrer visuellement le raisonnement algorithmique**, et aide à comprendre les étapes de chaque méthode.