Rapport Technique
Métro - Boulot - Dodo

Hugo LEROUX, Rémi GRIMAULT, Axel JACQUET

Rendu : 8 décembre 2024

Table des matières

1 Contexte et Objectif

Ce projet a pour but d'appliquer des concepts de théorie des graphes au contexte spécifique du réseau de métro parisien. L'objectif est de représenter ce réseau (sommets, arêtes, temps de trajet) sous forme de graphe, puis d'exploiter les algorithmes classiques de la théorie des graphes afin de :

— Vérifier si le graphe est connexe.

— Calculer les plus courts chemins entre stations.

— Déterminer un Arbre Couvrant de Poids Minimum (ACPM).

Le code fourni (en JavaScript) repose sur plusieurs classes essentielles :

— `Graph` : Représente le graphe (sommets, arêtes) et offre des méthodes d'analyse (connexité, plus courts chemins, ACPM).

— `Node`, `Edge` : Encapsulation des propriétés des sommets et arêtes.

— `Parser` : Chargement et parsing des données externes (fichiers texte).

— `MinHeap` : Structure de tas binaire minimum, utilisée notamment par l'algorithme de Dijkstra.

— `UnionFind` : Structure de gestion d'ensembles disjoints, utilisée par l'algorithme de Kruskal.

Chaque choix d'implémentation sera justifié d'un point de vue algorithmique et mathématique, avec des analyses de complexité (en $O(\cdot)$), et une mise en perspective des bornes théoriques. Toutes les sources du projets sont disponnible sur le dépot suivant : `https://github.com/KalinkaGit/efrei-maths-project/tree/dev`

2 Préparation de l'Environnement et Dépendances

Avant de lancer l'application, assurez-vous de disposer d'un environnement adéquat. Si vous le souhaitez il n'est pas nécéssaire de procéderaux différentes installation et vous pouvez vous rendre sur cette url la ou notre site est hébergé `https://kalinkagit.github.io/efrei-maths-project/`. Si vous accédez aux site hébergé vous pouvez directement passer à la prochaine partie du rapport.

2.1 Dépendances et Installation

Le projet utilise un environnement basé sur Node.js pour la gestion des dépendances et le lancement des scripts nécessaires. Les principales dépendances sont :

— `d3.js` : Librairie utilisée pour manipuler et afficher les données sous forme de graphiques SVG interactifs.

— `express` : Framework minimaliste pour le serveur HTTP.

— `body-parser` : Middleware pour parser les requêtes entrantes.

— `dotenv` : Permet de charger des variables d'environnement depuis un fichier `.env`.

2.2 Installation de Node.js

Pour installer Node.js, suivez les étapes ci-dessous :

1. **Télécharger Node.js** :

- Rendez-vous sur le site officiel de Node.js : `https://nodejs.org/`.
- Téléchargez la version **LTS** (recommandée pour la plupart des utilisateurs).

2. **Installer Node.js** :
   - Lancez l'installateur téléchargé.
   - Suivez les étapes, en conservant les options par défaut (notamment pour ajouter Node.js au `PATH`).
   - Une fois l'installation terminée, ouvrez une invite de commande (Windows PowerShell ou Command Prompt).
3. **Vérifier l'installation** :
   - Exécutez les commandes suivantes pour vérifier que Node.js et npm (Node Package Manager) sont correctement installés :

     ```
     node -v
     npm -v
     ```

   - Les versions respectives devraient s'afficher.

2.3 Préparation du Projet

1. **Accéder à la racine du projet** :
   - Ouvrez une invite de commande.
   - Utilisez la commande `cd` pour naviguer vers le dossier racine de votre projet. Exemple :

     ```
     cd C:\\chemin\\vers\\projet
     ```

2. **Installer les dépendances** :

   ```
   npm install
   ```

2.4 Lancer le Serveur

1. **Démarrez le serveur** :
   - Exécutez la commande suivante dans le dossier du projet :

     ```
     node server.js
     ```

2. **Vérifiez dans le navigateur** :
   - Ouvrez votre navigateur et accédez à l'adresse indiquée par le serveur :

     ```
     http://localhost:8080
     ```

3 Représentation du Graphe et Choix de la Structure de Données

3.1 Liste d'Adjacence

Le graphe $G = (V, E)$ est représenté par une liste d'adjacence. Concrètement, pour chaque sommet $v \in \{0, \ldots, V-1\}$, nous stockons une liste de ses voisins (u, w), où u est un sommet adjacent et w le poids (temps de trajet).

Formellement :

$$A : V \to \mathcal{P}(V \times \mathbb{R}^+)$$

Avec $A(v) = \{(u, w) \mid (v, u) \in E\}$.

3.2 Caractère Non Orienté du Graphe

Le code insère chaque arête dans les deux sens dans la liste d'adjacence :

```
this.dicoAdjency.get(vertex1).push({ node: vertex2, weight: travel_time });
this.dicoAdjency.get(vertex2).push({ node: vertex1, weight: travel_time });
```

Cela signifie que pour chaque arête entre $vertex1$ et $vertex2$, les deux sommets se référencent mutuellement. Ainsi, le graphe est traité comme non orienté.

3.3 Complexité en Espace

La liste d'adjacence utilise un espace $O(V + E)$. Étant donné que, dans un réseau de métro, le nombre d'arêtes E est en général proportionnel à V (chaque station a un nombre limité de connexions), cette représentation est très efficace, bien meilleure qu'une matrice d'adjacence qui nécessiterait $O(V^2)$ d'espace.

4 Vérification de la Connexité : Parcours en Profondeur (DFS)

La connexité du graphe est vérifiée en exécutant un DFS à partir d'un sommet, généralement le sommet 0. Le principe est simple :

1. Marquer le sommet de départ comme visité.
2. Explorer récursivement chaque voisin non visité.
3. À la fin, si tous les sommets sont marqués, le graphe est connexe.

4.1 Extrait de Code

```
isConnexe() {
    const visited = new Array(this.vertexCount).fill(false);

    const dfs = (node) => {
        visited[node] = true;
        (this.dicoAdjency.get(node) || []).forEach(({ node: neighbor }) => {
            if (!visited[neighbor]) dfs(neighbor);
        });
    };
```

```
    dfs(0);
    return visited.every(v => v);
}
```

4.2 Complexité

Le DFS explore chaque sommet et chaque arête au plus une fois. Sa complexité est donc :

$$O(V + E).$$

Cette complexité est optimale. Vérifier la connexité nécessite, dans le pire cas, de considérer la quasi-intégralité du graphe.

5 Calcul des Plus Courts Chemins : Algorithme de Dijkstra

L'algorithme de Dijkstra calcule les distances minimales d'un sommet source s vers tous les autres sommets dans un graphe à poids non négatifs. Il s'appuie sur une structure de priorité (un tas minimum) pour sélectionner à chaque étape le sommet le plus proche non encore fixé.

5.1 Principe Mathématique

Soit $d(v)$ la distance minimale connue pour atteindre v depuis s. L'algorithme maintient l'invariant suivant : lorsque le sommet u est extrait du tas avec la plus petite distance $d(u)$, cette distance est optimale. Ceci est démontré par récurrence : la plus petite valeur extraite ne peut pas être améliorée par une exploration future, garantissant la minimalité du chemin correspondant.

5.2 Extrait de Code

```
dijkstra(startNode) {
    const distances = Array(this.vertexCount).fill(Infinity);
    distances[startNode] = 0;
    const previousNodes = Array(this.vertexCount).fill(null);
    const heap = new MinHeap();
    heap.push({ node: startNode, distance: 0 });

    while (heap.size() > 0) {
        const { node: closestNode, distance: currentDist } = heap.pop();
        if (currentDist > distances[closestNode]) continue;

        for (let { node: neighbor, weight } of (this.dicoAdjency.get(closestNode) || [])
            const newDist = currentDist + weight;
            if (newDist < distances[neighbor]) {
                distances[neighbor] = newDist;
                previousNodes[neighbor] = closestNode;
                heap.push({ node: neighbor, distance: newDist });
            }
        }
    }
```

```
    return { distances, previousNodes };
}
```

5.3 Complexité

Avec un tas binaire (MinHeap), chaque opération `push` et `pop` est en $O(\log V)$. Comme on effectue $O(V+E)$ opérations (insertion initiale, extractions, relaxations), la complexité globale est :

$$O((V+E)\log V).$$

5.4 Améliorations Potentielles

Un tas de Fibonacci pourrait réduire la complexité à $O(E+V\log V)$, mais l'implantation serait plus complexe. Dans le contexte du réseau de métro, l'amélioration serait pas forcément utile de plus cette méthode n'as pas forcément était vue en cours mais découvert lors de nos rechecrhe sur le sujet.

6 Arbre Couvrant de Poids Minimum : Algorithme de Kruskal

6.1 Principe Mathématique

L'ACPM (Arbre Couvrant de Poids Minimum) est un sous-ensemble d'arêtes qui connecte tous les sommets avec un poids total minimal. L'algorithme de Kruskal :

1. Trie les arêtes par ordre croissant de poids.
2. Parcourt ces arêtes, et les ajoute si elles ne forment pas de cycle dans l'arbre en construction.

6.2 Union-Find

Pour détecter les cycles efficacement, on utilise la structure Union-Find :

— `find(x)` : Trouve la racine (représentant) de l'ensemble contenant x.

— `union(x,y)` : Fusionne les ensembles contenant x et y.

Avec l'optimisation *path compression* et *union by rank*, chaque opération est en amorti $O(\alpha(V))$, pratiquement constant.

6.3 Extrait de Code

```
kruskal() {
    const edgesSorted = [...this.edges].sort((a, b) => a.travel_time - b.travel_time);
    const unionFind = new UnionFind(this.vertexCount);
    const mst = [];
    let totalWeight = 0;

    for (const edge of edgesSorted) {
        const { vertex1_id, vertex2_id, travel_time } = edge;
        const root1 = unionFind.find(vertex1_id);
```

```
        const root2 = unionFind.find(vertex2_id);

        if (root1 !== root2) {
            unionFind.union(root1, root2);
            mst.push(edge);
            totalWeight += travel_time;
        }
    }

    return { mst, totalWeight };
}
```

6.4 Complexité

Le tri des arêtes domine la complexité :

$$O(E \log E) + O(E\alpha(V)) \approx O(E \log E).$$

Comme $E \approx V$ dans un graphe peu dense, on a $O(V \log V)$ en ordre de grandeur.

7 Complexité et Optimalité

7.1 Connexité via DFS

— Complexité : $O(V + E)$

— Optimalité : Une borne inférieure $\Omega(V + E)$ s'applique, donc c'est optimal.

c.f : Annexe - Figure 1

7.2 Dijkstra (Tas Binaire)

— Complexité Dijkstra (sans tas) : $O(V^2)$

— Complexité Dijkstra (tas binaire) : $O((V + E) \log V) \approx O(V \log V)$ pour un graphe peu dense.

c.f : Annexe - Figure 2

7.3 Kruskal et Union-Find

— Complexité : $O(E \log E)$, dominée par le tri.

— Opérations d'Union-Find en $O(\alpha(V))$, quasi constant.

c.f : Annexe - Figure 3

8 Conclusion

Nous avons présenté les choix d'implémentation, justifiés d'un point de vue à la fois pratique et mathématique :

— La liste d'adjacence assure une mémoire efficace ($O(V + E)$) et des parcours linéaires.

— Le DFS permet de vérifier la connexité en $O(V + E)$, ce qui est optimal.

— Dijkstra, avec un tas binaire, tourne en $O((V + E) \log V)$, ce qui est optimal.

— Kruskal, combiné à Union-Find, offre un calcul de l'ACPM en $O(E \log E)$, proche de l'optimal.

Toutes ces complexités correspondent aux limites théoriques établies en, cours. Les algorithmes utilisés (DFS, Dijkstra, Kruskal) sont standard. Dans le contexte du réseau de métro, ces approches sont suffisantes pour une exécution rapide.

En somme, l'implémentation réalisée est un compromis optimal entre simplicité, efficacité et robustesse théorique.

9 Sources

— https://fr.wikipedia.org/wiki/Algorithme_de_parcours_en_profondeur#Complexit%C3%A9

— https://fr.wikipedia.org/wiki/Algorithme_de_Dijkstra#Complexit%C3%A9_de_l'algorithme

— https://fr.wikipedia.org/wiki/Tas_de_Fibonacci

— https://fr.wikipedia.org/wiki/Union-find

— Cours II - Arbre couvrant de poids minimum

— Cours III - Recherche du plus court chemin

10 Annexes

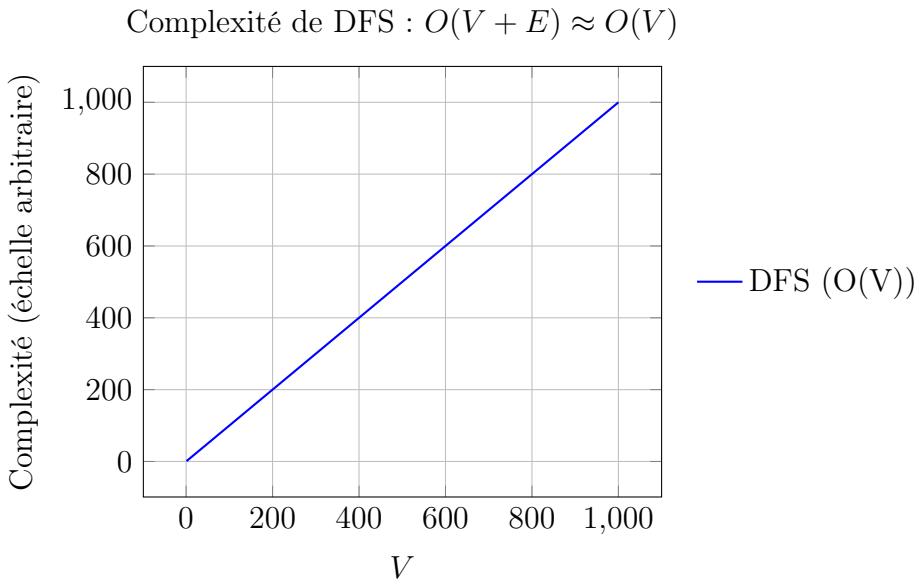

FIGURE 1 – Croissance asymptotique de la complexité de DFS

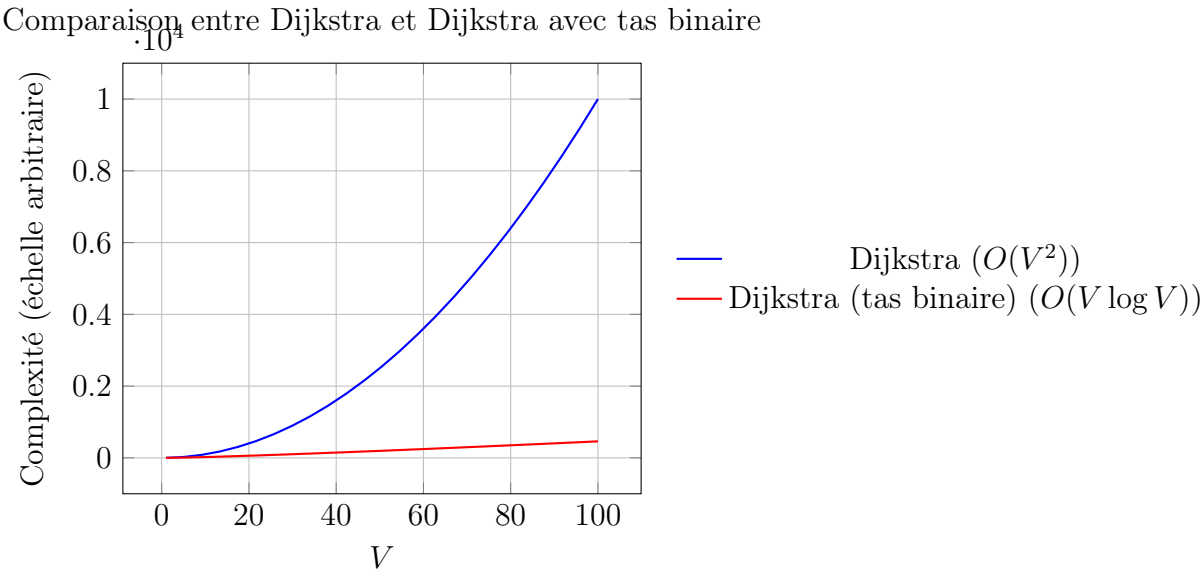

FIGURE 2 – Croissance asymptotique de la complexité : Dijkstra vs Dijkstra avec tas binaire

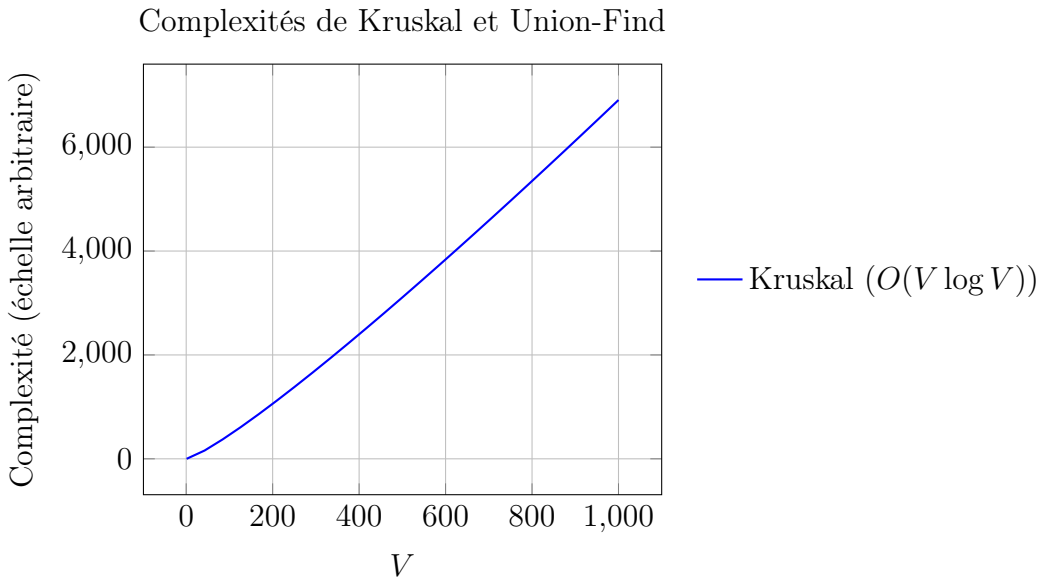

FIGURE 3 – Croissance asymptotique de la complexité pour Kruskal et Union-Find