Projet de modélisation du métropolitain

Aubier Jules
10821

Rossi Alessandro
10896

10 juin 2019

Table des matières

Introduction **3**

Graphe non pondéré **4**

0.1 Chemin le plus court 4

0.1.1 Algorithme pour le plus court chemin 4

0.1.2 Description de BFS(SubwayStation x) 5

Graphe pondéré **6**

0.2 Chemin le plus court 6

0.2.1 Algorithme pour le plus court chemin 6

0.2.2 Algorithme de Djikistra 7

0.2.3 Implémentation 7

Diamètre et rayon **8**

0.3 Rappels 8

0.4 Description de l'algorithme 8

Partitionnement de graphe **10**

0.5 Rappels 10

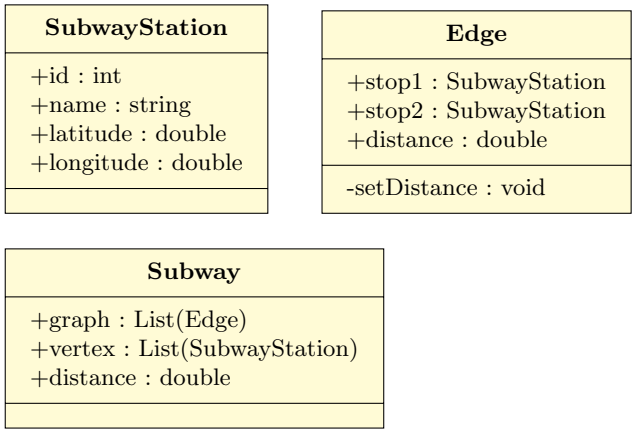
0.6 Description de l'algorithme et résultats 10

Conclusion **11**

Introduction

L'objectif de ce projet est de réaliser une application capable de donner le plus court ou le plus long chemin entre deux stations de métropolitain.Le second objectif est d'identifier les différents clusters du graphe pondéré.

Pour réaliser cet objectif nous allons modéliser le réseau parisien sous la forme d'un graphe non pondéré en premier lieu puis pondéré avec les distances entre les stations. Pour cela nous utilisons les données fournies par la RATP(`https://data.ratp.fr/explore/`) sur le réseau souterrain uniquement.
Ci-dessous la structure du graphe.



Dans la réalisation de ce projet, nous utiliserons java 8 et particulièrement l'API stream. Un stream est une séquence d'éléments sur laquelle on peut exécuter des opérations de manière séquentielle ou parallèle. Cette API permet d'une part de simplifier la syntaxe du code et de le rendre plus lisible. La grande force de cette API reste tout de même la parallélisation des streams permettant une exécution plus rapide du code mais à utiliser avec caution.

Graphe non pondéré

Dans cette partie, nous avons modélisé le métropolitain parisien comme un graphe non pondéré ou les stations sont les nœuds et les liens entre les stations les arêtes.

0.1 Chemin le plus court

Pour obtenir le chemin le plus court dans un graphe non pondéré nous allons utiliser le BFS (Breadth First Search). Il est évident que le graphe ne contient pas de boucle. Il n'y a pas de métro qui partent d'une station pour arriver à la même station. De plus, nous sommes dans un graphe non pondéré. Toutes les conditions d'utilisation du BFS sont remplies.

0.1.1 Algorithme pour le plus court chemin

L'objectif est de construire un arbre simple ayant pour racine une station de métro. Ainsi pour obtenir le chemin le plus rapide entre cette station racine et une autre station, il ne reste qu'à remonter l'arbre jusqu'à la source. Ci-dessous la description de l'algorithme.

L'algorithme implémente plusieurs méthodes qui sont :

— La méthode **getNeighbours(SubwayStation x)** qui permet de récupérer la liste des voisins directes d'une station et les lignes associées sous la forme d'une HashMap() `<SubwayStation,String>`.

— La méthode **BFS(SubwayStation x)** qui construit l'arbre à partir de la station racine x. Cette méthode renvoie une List `<Edge>` correspondant à l'arbre.

— La méthode **ShortestPath(SubwayStation a,SubwayStation b)** qui ne renvoie rien mais affiche le plus court chemin entre les stations a et b.

0.1.2 Description de BFS(SubwayStation x)

On initialise trois collections que sont
- List<Edge> **bfsTree** qui contiendra les arêtes de l'arbre construit après le BFS. Cette liste est initialisée comme vide.
- Map<SubwayStation,Boolean> **visited**. Cette map prends pour clé une station et pour valeur un booléen qui indique si ce nœud a déjà été visité.
- Map<SubwayStation,String> **voisins** qui contient les voisins obtenus par la fonction **getNeighbours(SubwayStation x)**.

On initialise également une queue<SubwayStation> qui contient initialement la station source.

Tant que cette queue n'est pas vide :
On initialise une station S en retirant la tête de la queue.
On update les voisins comme voisins de S.
Pour chaque voisin, on vérifie si il a été visité. Si le voisin a déjà été visité on passe au suivant.
Autrement on marque le voisin comme visité et on ajoute à **bfsTree** une nouvelle arête ayant pour source S et pour destination ce voisin.

Cette fonction est appelée pour calculer le plus court chemin entre deux stations. Cette fonction identifie la destination et remonte jusqu'à la source. En moyenne le temps d'exécution de cette méthode est de 18 millisecondes.

Graphe pondéré

Dans cette partie, nous avons modélisé le métropolitain parisien comme un graphe pondéré.

0.2 Chemin le plus court

0.2.1 Algorithme pour le plus court chemin

Dans le cas d'un graphe pondéré ou les poids des arêtes ne sont pas tous équivalents le BFS n'est pas utilisable. Nous allons davantage envisager l'utilisation de l'algorithme de Djikistra.

Les conditions d'utilisation de Djikistra sont comme suit :
— Un graphe pondéré et acyclique
— Aucun poids d'arête négatif ou nul
— Les distances entre deux stations sont des réels

Dans notre étude de cas, les poids des arêtes sera la distance séparant deux stations de métro. Cette distance **strictement positive** est la distance euclidienne entre deux stations représentées comme des points dans un plan en 2D ayant pour coordonnées (longitude,latitude). Les arêtes ne prennent donc pas en compte la route des rails de métro.

Une amélioration possible de cette modélisation est de pondérer le graphe avec le temps entre chaque station (fourni par l'API RATP) plutôt qu'avec la distance euclidienne qui n'est pas représentative de la réalité.

0.2.2 Algorithme de Djikistra

Ce qui suit est un rappel du fonctionnement de l'algorithme de Djikistra. L'algorithme de Djikistra permet d'obtenir un arbre simple à partir d'un nœud source de telle manière que le chemin de ce nœud source à un autre nœud dest soit la distance la plus courte possible.

Cet algorithme est de complexité temporelle polynomiale.Pour n noeuds et a arcs, la complexité est en O(a+nlog(n)).

0.2.3 Implémentation

De la même manière qu'avec le graphe non pondéré, Djikistra est implémenté de sorte que la fonction retourne un arbre connecté acyclique contenant les plus courts chemins à partir d'une station source. La différence est que cet arbre est pondéré.

Description de l'algorithme : On initialise toutes les stations à une distance de ∞ sauf la distance de la station source initialisée à 0.
On initialise deux set que sont settled et unsettled qui vont fonctionner comme des piles pour les stations. Dans settled on aura des stations dont les distances sont déja déterminées avec leurs stations précédentes et suivantes et dans unsettled seront les stations dont il reste à visiter les voisins autre que la station précédente.
On boucle tant que toutes les stations n'ont pas été totalement explorées. Pour chaque station, on compare la distance aux les stations suivantes et on détermine le minimum.
A la fin de cette phase, l'arbre à un arc en plus ayant pour station source la station actuelle et pour destination la station adjacente ayant la distance minimale.
L'algorithme fournit à la fin l'arbre des plus courts chemin.

Cette fonction est appelée pour calculer le plus court chemin entre deux stations. Cette fonction identifie la destination et remonte jusqu'à la source. En moyenne le temps d'exécution de cette méthode est de 18 millisecondes.

Diamètre et rayon

Dans cette partie nous allons déterminer le diamètre et le rayon du graphe. La méthode de calcul du diamètre ne change pas selon si le graphe est pondéré ou non.

0.3 Rappels

En théorie des graphes, le **diamètre** d'un graphe est la plus grande distance possible qui puisse exister entre deux de ses sommets ; la distance entre deux sommets étant définie par la longueur d'un plus court chemin entre ces deux sommets.
En d'autres termes, le diamètre est l'excentricité maximale de ses sommets.

En théorie des graphes, le **rayon** d'un graphe est l'excentricité minimale de ses sommets, c'est-à-dire la plus petite distance à laquelle puisse se trouver un sommet de tous les autres. Le centre d'un graphe est formé de l'ensemble de ses sommets d'excentricité minimale.

L'excentricité maximale est appelée diamètre.

La distance entre deux sommets dans un graphe est définie par la longueur d'un plus court chemin entre ces deux sommets. Source : `https://fr.wikipedia.org/wiki/Diam%C3%A8tre_(th%C3%A9orie_des_graphes)`

0.4 Description de l'algorithme

Pour obtenir le diamètre et le rayon du graphe il faut donc comparer la taille de tous les plus courts chemins soit tous les calculer. Nous avons donc pour chaque paire de nodes distinct, calculé la taille du chemin le plus court et tout comparé.‘

Dans le cas du métro parisien, il paraît évident que toutes les stations sont reliées. Une station isolée est une station non utilisée. Ainsi, il est impossible que le diamètre ou le rayon est une valeur infinie.

Pour le diamètre, on obtient un résultat de 40 stations. Le résultat n'est pas abbérant. Cela signifie que le plus long des courts chemins entre deux stations est de 40.

Pour le rayon, on récupère l'eccentricité maximale de chaque station et on garde la plus petite. Dans notre cas le rayon est de 2. Cela signifie que le plus petit nombre de stations possible entre une station et chaque station est de 2.

Partitionnement de graphe

0.5 Rappels

Le partitionnement est le fait de calculer une partition. Le plus souvent le partitionnement de graphe consiste à créer une subdivision de l'ensemble des sommets de S en k sous ensembles de tailles réduites de façon à minimiser un ou plusieurs critères.

Source : `https://fr.wikipedia.org/wiki/Partitionnement_de_graphe`

0.6 Description de l'algorithme et résultats

L'objectif de cette partie est premièrement d'identifier les arcs ou passent le plus de plus court chemins puis dans l'idéal d'identifier et d'expliquer les différentes partitions. Nous n'avons réalisé que la première partie. On réalise exactement le même processus que pour trouver le diamètre sauf qu'on ne s'interesse pas qu'à la taille des chemins les plus courts. On va ici crée une Map`<Edge,Tuple<Station,Station` qui va pour chaque arc listé les plus courts chemins passant par lui. A la fin de l'algorithme, on retourne cet objet trié dans l'ordre décroissant. Cela nous donne ainsi les arcs ayant le plus de passage et les chemins les plus courts qui y passent.

Pour récupérer les clusters , il faudrait construire des sous graphes (donc ici des List `<Edge>`) selon les arcs les plus visités.

Conclusion

En utilisent les informations du réseau souterrain du métro parisien fournis par la RATP, nous avons donc créé un programme permettant de connaitre le trajet à prendre pour se déplacer le plus rapidement possible d'un point à un autre, en prennent en compte le temps entre chaque station. Et l'utilisation de différents algorithmes nous a permis de réaliser des études sur les graphes, en s'appuient sur les données de la RATP.

Les algorithmes pour trouver le diamètre et pour identifier les clusters sont longs à l'exécution. En effet chacun prends 5 minutes pour s'exécuter. En effet on réalise une boucle qui tourne 302 fois et qui exécute notre algorithme qui prends en moyenne 18 millisecondes d'exécution. 5 minutes est plutôt long et chercher des voies d'améliorations peut être intéressant.