



---

# 4M016 - Implémentation de l'algorithme Dijkstra

---

*Auteurs:*

Bailin CAI, Christain RITZ

*Responsable:*

Didier SMETS

25 Déc 2021

# Plan

<b>Introduction</b>	<b>3</b>
1 Algorithme de Dijkstra	3
2 Comparaison entre Matrice d'adjacente et Liste d'adjacente	5
3 Le file de priorité: min heap	6
4 Analyse le résultat	7
5 Lire le Graphe lié à Paris	9
<b>Conclusion</b>	<b>11</b>

# Introduction

Le but de ce projet était de réaliser une implémentation de l'algorithme de Dijkstra de recherche de chemin en utilisant des tas (file de priorité). Dans notre projet nous utilisons le min-heap pour réaliser le file de priorité.

Cette implémentation a été réalisée en langage C pour 2 types de présentation utilisée très souvent, la matrice d'adjacence et la liste d'adjacence. et pour tester la précision de l'algorithme Dijkstra avec min heap, on a aussi implémenté l'algorithme Dijkstra naïf pour les deux types de présentation du graphe pour assurer que on a le résultat correct.

## 1 Algorithme de Dijkstra

En théorie des graphes, l'algorithme de Dijkstra sert à résoudre le problème du plus court chemin. Il permet, par exemple, de déterminer un plus court chemin pour se rendre d'une ville à une autre connaissant le réseau routier d'une région. Plus précisément, il calcule des plus courts chemins à partir d'une source vers tous les autres sommets dans un graphe orienté pondéré ou non-orienté par des réels positifs. On peut aussi l'utiliser pour calculer un plus court chemin entre un sommet de départ et un sommet d'arrivée.

Cet algorithme réalisé avec file de priorité est de complexité polynomiale. Plus précisément, pour  $n$  sommets et  $a$  arcs, le temps est en  $O((a + n) \log n)$  voire en  $O(a + n \log n)$ . si on utilise le tas de fibonacci.

## 1.1 Le principe d'Algorithme Dijkstra

Le principe de l'algorithme est de parcourir progressivement tout le graphe. On initialise au début un tableau  $dist$  pour tous les noeuds associés à une distance infinie et un tableau  $prev$  pour tous les noeuds associés au père de chaque noeuds, une file de priorité(min heap). La source est attribuée à une distance nulle et son père est égale à lui même.

Ensuite on ajoute le noeud  $u = src$  dans le file de priorité, on itère ensuite sur la file de priorité jusqu'au ça soit vide, on parcourt tous les nodes  $v$  qui sont liés avec lui, et on les ajoute dans le file de priorité, chacune opération d'insertion prend temps en  $O(\log(n))$  avec la propriété de min heap, on peut extraire le noeud avec la valeur minimal en  $O(1)$ , et on calcule la distance du chemin de la source  $u$  en passant par  $v$ , S'il est plus faible que la distance de  $v$  à la source sans passer par  $v$ , on met à jour cette distance et on note que le prédécesseur de  $v$  est  $u$ .

## 1.2 Pseudo code

---

**Algorithm 1** Dijkstra minheap( $G, w, src, dist, prev$ )

---

```
 $Q \leftarrow \emptyset$ 
for each  $v \in V[G]$  do  $dist[v] \leftarrow +\infty, prev = \text{Not defined}$ 
Insert( $Q, src, 0$ )
while  $Q \neq \emptyset$  do
     $(u, k) \leftarrow \text{Extract-min}(Q)$ 
    if  $k \leq dist$  then
         $dist[u] \leftarrow k$ 
        for each  $(u, v) \in E[G]$  do
            if  $dist[u] + w[u, v] < dist[v]$  then
                Insert( $Q, v, dist[u] + w(u, v)$ )
                 $dist[v] \leftarrow w(u, v)$ 
                 $prev[v] \leftarrow u$ 
```

---

## 2 Comparaison entre Matrice d'adjacente et Liste d'adjacente

Un graphe présenté en matrice d'adjacente requies  $|V|^2$  pour stocker où  $|V|$  est le nombre de noeud dans le graph, on peut accéder le poid entre 2 noeud en temps  $O(1)$ .

Un graphe présenté en liste d'adjacente comme un tableau de liste chaînée, l'indice de tableau représente un noeud  $u$  et chaque élément de liste chaînée représente le noeud qui a un arête de  $u$  à  $v$

Et ci-dessous présente la complexité des opérations élémentaires.

Opération	Matrice d'adjacente	Liste d'adjacente
mémoire utilisé	$O( V ^2)$	$O( V  +  E )$
Accès	$O(1)$	$O( V )$
Ajoute un vertex	$O( V ^2)$	$O(1)$
Ajoute un arête	$O(1)$	$O(1)$
Supprime un arête	$O(1)$	$O( E )$
Supprime un noeud	$O(V)^2$	$O( V  +  E )$

**Donc pour la matrice d'adjacente si:**

- Le problème demande beaucoup de fois d'accès aux poids entre 2 arêtes.
- Ce n'est pas bien pour un graphe creuse i.e. un graphe avec  $O(|V|)$  arêtes.

**Donc pour la matrice d'adjacente si:**

- Quand on voudrais itérer sus les voisins d'un noeud.
- Ce n'est pas bien pour un graphe dense i.e. un graphe avec  $O(|V|^2)$  arêtes.

### 3 Le file de priorité: min heap

On commence dans cette section de parler le min heap, un min heap est une implémentation du file de priorité qui est un arbre binaire tel que au chaque noeud, leur fils est inférieure ou égale au noeud

#### 3.1 Insertion

Le procédure d'insérer un noeud dans un heap peut écrire comme suivant:

- Placer le nouveau élément dans la position le plus prochain possible dans le tableau.
- Comparer le nouveau élément avec son parent, si le nouveau élément est plus petit, alors échange lui avec son parent.
- Continue le procédure jusqu'au soit le parent nouveau élément est plus petit ou égale avec le nouveau élément, soit le nouveau élément arrive au racine de l'arbre i.e. index 0 dans le tableau.

#### 3.2 Extract-min

Le procédure d'éliminer un noeud dans un heap peut écrire comme suivant:

- Met l'élément racine dans une variable pour le retourner à la fin.
- Échanger le dernier élément avec la racine.
- Tant que l'élément placé est plus grand que un des fils, les échange
- Retourner la racine élément sauvegardé dans la variable déclaré au début.

## 4 Analyse le résultat

On a implémenté Dijkstra de façon naïve en matrice d'adjacente et en liste d'adjacente et avec min-heap, donc il y a en tout 4 algorithmes pour analyser la complexité, mais si on analyse un peu l'algorithme Dijkstra implémenté en Liste d'adjacente de façon naïve, déjà il faut parcourir tous les arête pour trouver le minimum distance à src, et ensuite parcourir encore une fois sur tous les arêtes non visités, c'est déjà  $O(|V|^2)$  et que pour accéder au poids, ça prend au pire encore  $O(|V|)$ , donc c'est de complexité en  $O(|V|^3)$ , donc ce n'est pas dans le cas considéré pour tester la complexité.

### 4.1 Graphe Dense

Un graphe Dense est un graphe avec le nombre d'arête qui est  $O(|V|^2)$ , et un graphe non-orienté connexe sans cycle sans plusieurs arête entre deux point a un nombre maximal de  $\frac{|V| \times (|V|-1)}{2}$  arêtes, la densité  $d = \frac{|E|}{|V| \times (|V|-1)}$  peut nous aider à générer un graphe dense

Premièrement on va travailler sur le même graphe dense pour tester le temps d'exécution pour les 3 algorithmes

- Dijkstra naïve en Matrice d'adjacente.
- Dijkstra avec min heap en Matrice d'adjacente.
- Dijkstra avec min heap en Liste d'adjacente.

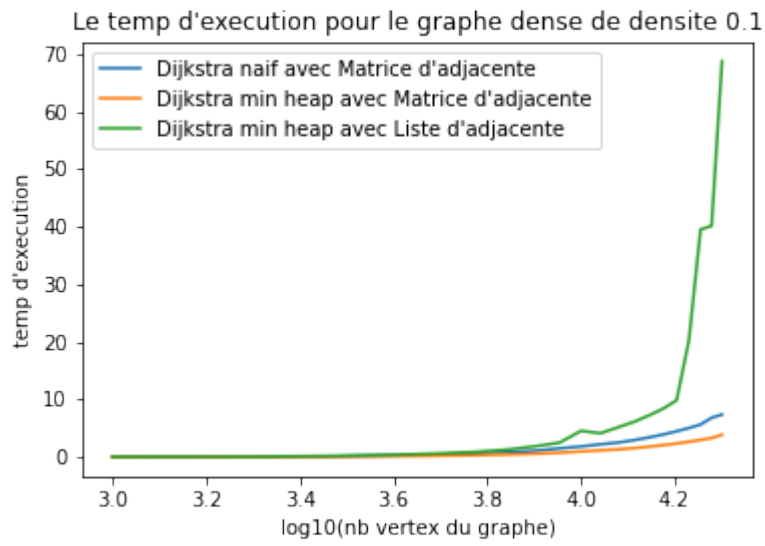


Figure 1: The caption

Le temp d'exécution pour  $d = 0.1$  est comme suivant:

Le temp d'exécution pour  $d = 0.5$  est comme suivant:



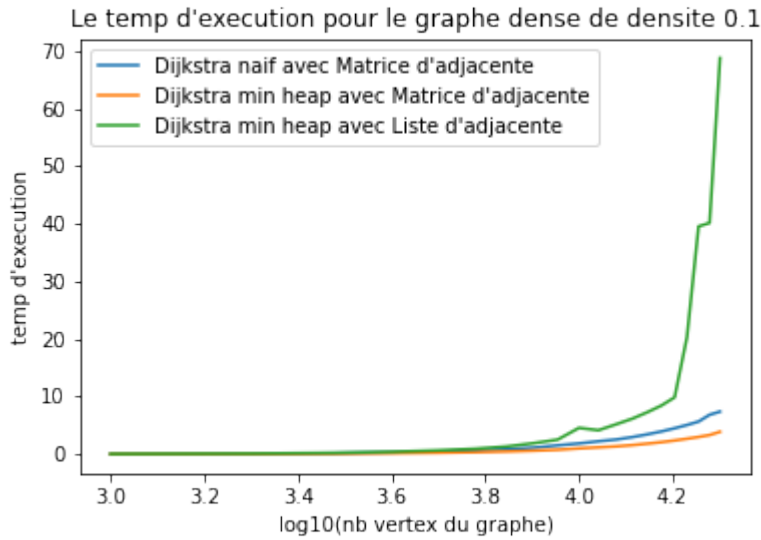


Figure 2: The caption

## 4.2 Graphe Creuse

Un graphe Creuse est un graphe avec le nombre d'arête qui est  $O(|V|)$ , on peut aussi définir la densité  $d = \frac{|E|}{|V|}$  peut nous aider à générer un graphe creuse facilement.

Ensuite on va travailler sur le même graphe creuse pour tester le temps d'exécution pour les 3 algorithmes

Le temp d'exécution pour  $d = 3$  est comme suivant:

Le temp d'exécution pour  $d = 5$  est comme suivant:

Le temp d'exécution pour  $d = 7$  est comme suivant:

## 4.3 Comparaison

Notons que la présentaion du graphe en liste d'adjacente pour le graphe dense est très mal c'est parce que

- Pour accéder au poids entre 2 noeuds, au pire il faut  $O(|V|)$  opération, et comme c'est un graphe dense, il faut en moyenne  $d$  opérations.

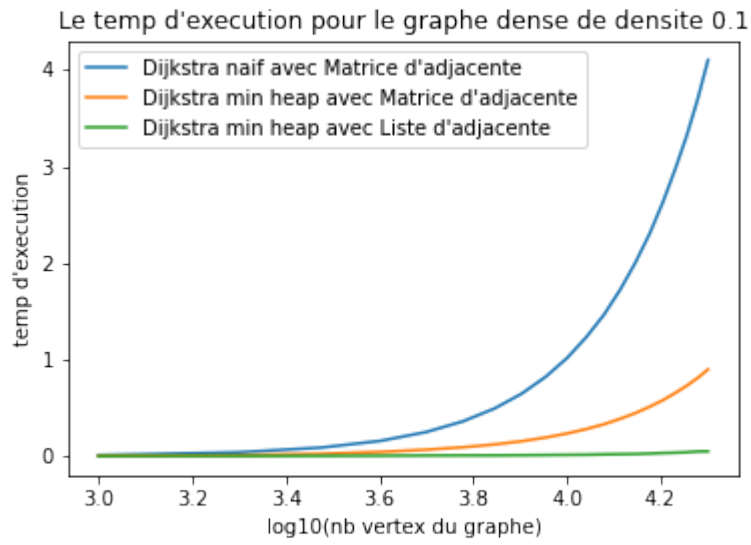


Figure 3: The caption

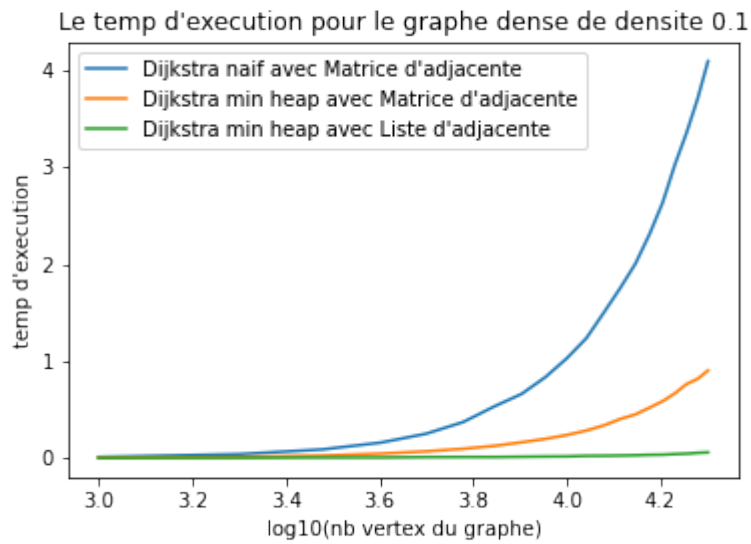


Figure 4: The caption

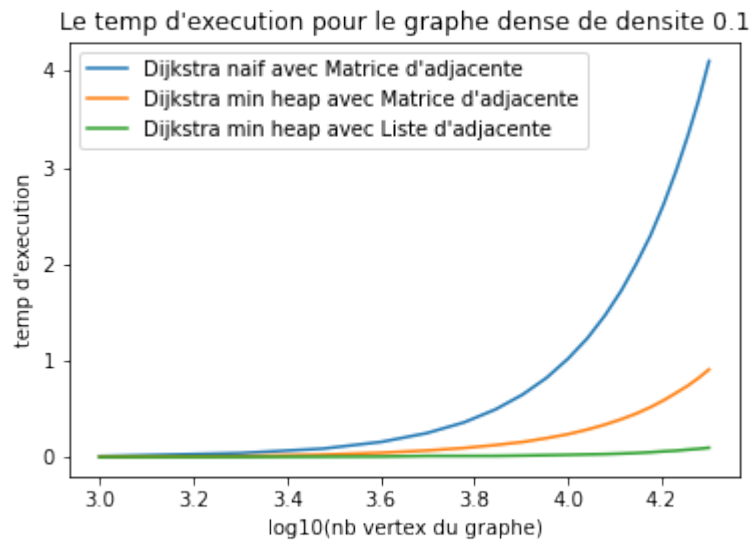


Figure 5: The caption

- Pour convertir le graphe de Matrice d'adjacente en graphe de la Liste d'adjacente, ça prend  $O(|V|^2)$  aussi.

## 5 Lire le Graphe lié à Paris

Pour Lire le graphe lié à paris, j'ai fait un peu de recherche sur Internet, j'ai trouvé sur la site Openstreetmap où qu'on peut récupérer le graphe en format .osm qui est en fait un fichier xml, il existe plusieurs librairies qui peuvent lire le fichier xml, par exemple les librairies libxml2 et expat, j'ai choisi libxml2 pour lire le fichier osm.

un fichier osm est commencé par la balise <osm> et dedans il y a node, way, relation, on s'intéresse que les informations sur les node et way avec tag = "highway", comme libxml2 nous propose d'extraire les informations par XPATH, on utilise l'expression XPATH = "/osm/node" pour extraire les nodes et xpath = "/osm/way[tag]/@k='highways']" pour extraire les ways.

### 5.1 node

```
<node id="25034333" visible="true" version="6" changeset="75240962"
  ↳ timestamp="2019-10-03T15:10:01Z" user="patman37" uid="311391
  ↳ " lat="48.8443883" lon="2.3535751"/>
<node id="25034334" visible="true" version="4" changeset="65643925
  ↳ " timestamp="2018-12-20T15:37:14Z" user="patman37" uid="
  ↳ 311391" lat="48.8445531" lon="2.3535496"/>
<node id="25034341" visible="true" version="3" changeset="4412077"
  ↳ timestamp="2010-04-13T09:47:35Z" user="Mawie" uid="150619"
  ↳ lat="48.8449881" lon="2.3538455"/>
<node id="25034342" visible="true" version="4" changeset="25655127
  ↳ " timestamp="2014-09-24T22:32:58Z" user="Goffredo" uid="
  ↳ 194751" lat="48.8450298" lon="2.3539405"/>
<node id="25034405" visible="true" version="8" changeset="66359980
  ↳ " timestamp="2019-01-16T11:36:33Z" user="patman37" uid="
  ↳ 311391" lat="48.8461163" lon="2.3545259">
  <tag k="highway" v="traffic_signals"/>
</node>
<node id="25034407" visible="true" version="6" changeset="65960848
  ↳ " timestamp="2019-01-02T14:54:59Z" user="patman37" uid="
  ↳ 311391" lat="48.8463956" lon="2.3517202"/>
```

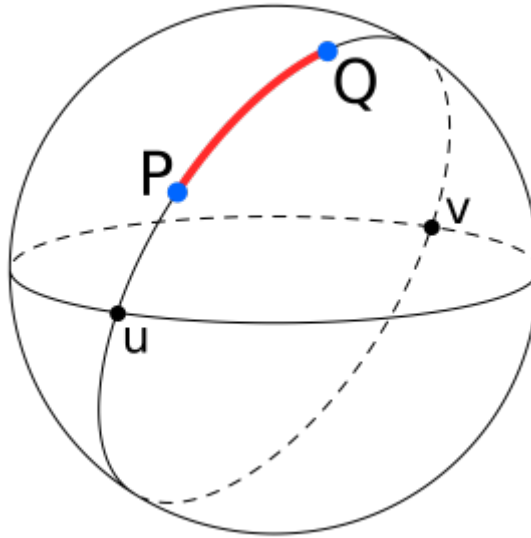


Figure 6: La distance entre 2 noeuds

Un node est écrit comme ci-dessus, on s'intéresse que sur les attributs id, lat, lon , et pour avoir la distance entre 2 nodes comme ci-dessus, on utilise Haversine formulaire

## 5.2 way

```
<way id="4217066" visible="true" version="4" changeset="4768311"
  ↪ timestamp="2010-05-21T18:24:00Z" user="Esperanza36" uid="
  ↪ 83557">
  <nd ref="25033531"/>
  <nd ref="25033533"/>
  <tag k="cycleway" v="opposite"/>
  <tag k="highway" v="residential"/>
  <tag k="maxspeed" v="30"/>
  <tag k="name" v="Rue des Ursulines"/>
  <tag k="oneway" v="yes"/>
</way>
```

On s'intéresse que pour les way avec le tag = "highway", après on ajoute les arêtes successivement dans le graphe et calculer leur distance en utilisant la formulaire Haversine

Le principe de générer un graphe à partir d'un fichier osm est de

- Parcourir le fichier pour trouver tous les noeuds
- Créer un table hashage pour accéder au index de noeud dans le graphe
- Récupérer tous les ways et en utilisant le table hashage crée, on peut accéder l'indice de noeud en temp  $O(1)$ . pour construire les arêtes pour ce graphe.

J'ai testé le temp d'exécution pour lire le fichier pas très grand, 13Mo est assez rapide mais quand je teste pour lire un fichier osm qui contient la carte de tout paris, mon ordinateur est planté. C'est la limite de fichier osm parce que c'est très grand et ce n'est pas comme d'autre fichiers basé sur .osm par exemple .osm.pbf qui est environ 10% de taille de fichier osm mais contient le même information.

## Conclusion

C'est un algorithme très très classique et connue et beaucoup utilisé dans le problème de recherche le plus court chemin, cependant pour avoir le plus vite possible le resultat, il faut bien choisir la methode de l'implementer et le structure de graphe

Pour aller plus loin, on peut essayer d'implementer le tas de fibonacci pour avoir le plus vite resultat et pour lire le fichier osm, il faut penser de trouver un autre librairie qui supporte le langage c a lire les donnees ou en plus de choisir une autre langage par exemple c++ pour traiter les donnees osm.pbf qui est le fichier compresse de fichier osm qui est beaucoup plus petit par rapport a fichier osm