جامعة سيدي محمد بن عبد الله بفاس
UNIVERSITÉ SIDI MOHAMED BEN ABDELLAH DE FES
كلية العلوم ظهر المهراز فاس
FACULTE DES SCIENCES DHAR EL MAHRAZ F

Master Big Data Analytics & Smart System

Intitulé :

Algorithme Dijkstra.

Réalisé par :

EL JARI Nassima

EL BOUSSADANY Chaymae

CHATER Najoua

Année Universitaire : ***2023/2024***

Dédicace

A Dieu le tout puissant, le Majestueux, le glorieux et le bienveillant

A nos chers parents

A nos professeurs

A nos amies et nos camarades.

REMERCIEMENT

Avant tout, nous tenons à

Remercier toute personne qui

Nous a aidé à la réalisation du projet et de

Ce rapport tout au long de ce

Dernier mois du près ou de loin.

Particulièrement, nous adressons nos remerciements à notre encadrant

Mr. Bennani Mohamed Taj

Qui n’a pas cessé de nous encourager pendant la durée du projet, et

Nous le remercions pour son encadrement,

Pour sa modestie,

Pour ses conseils et ses soutiens.

Finalement,

Nous tenons à remercier l’ensemble du corps professoral

Du département informatique spécialement

A nos professeurs et membres des jurés présents qui

Nous

Font l’honneur d’examiner notre travail

RÉSUMÉ

Le projet consiste à mettre en œuvre l'algorithme de Dijkstra en utilisant Python pour déterminer le chemin le plus court entre un état initial et un état cible. L'algorithme prend en entrée une matrice d'adjacence et des coordonnées GPS pour représenter le graphe, et son objectif est de calculer les chemins les plus courts depuis le nœud de départ vers tous les autres nœuds du graphe.

Table de matières

Introduction Générale

La quête du plus court chemin demeure un défi central et intensément exploré dans le cadre de la théorie des graphes. L'objectif consiste à dénicher les trajets les plus courts entre deux points au sein d'une structure de données nommée graphe, dépeignant les connexions entre divers éléments. Cette problématique revêt une pertinence pratique étendue, de l'optimisation des itinéraires au sein des réseaux de transport à l'élaboration de circuits pour les véhicules de livraison. En outre, elle est mobilisée pour définir les voies les plus rapides pour les transmissions de données au sein des réseaux informatiques.

Plusieurs algorithmes ont été développés pour résoudre le problème du plus court chemin, chacun présentant ses propres atouts et limitations. Parmi les algorithmes les plus fréquemment employés figurent l'algorithme de Dijkstra, l'algorithme Bellman-Ford, et l'algorithme A*. Ces méthodes sont adaptées à la résolution de problèmes de plus court chemin dans des graphes pondérés et non pondérés. Un autre exemple d'algorithme significatif est l'algorithme de Floyd-Warshall, qui se distingue par sa capacité à résoudre le problème du plus court chemin entre tous les pairs de sommets dans un graphe, offrant ainsi une alternative robuste dans divers contextes.

En synthèse, la problématique du trajet optimal demeure un secteur de recherche essentiel et largement appliqué dans divers domaines. L'efficacité résolutive de cette question revêt une importance cruciale pour optimiser les performances et la rentabilité de multiples systèmes et processus. Les méthodes dédiées au plus court itinéraire sont mises en œuvre pour définir les parcours les plus véloces entre deux points dans une structure de données, et elles présentent de nombreuses applications dans les domaines des réseaux de déplacement, des communications, des réseaux informatiques et des systèmes de guidage.

Chapitre 1: GENERALITES

1. Définition et exploitation de l'algorithme de Dijkstra

1.1. Definition

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 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.

L'algorithme porte le nom de son inventeur, l'informaticien néerlandais Edsger Dijkstra, et a été publié en 1959.

1.2. Description de Dijkstra

Les étapes de l'algorithme de Dijkstra pour résoudre le problème du plus court chemin dans un graphe pondéré sont les suivantes :

1. Initialisation des distances : Attribuer une distance initiale à chaque nœud du graphe. La distance du nœud source est fixée à zéro, tandis que les distances des autres nœuds sont initialement mises à une valeur infinie.

2. Sélection du nœud source : Choisir le nœud source comme point de départ.

3. Exploration des nœuds adjacents : Pour chaque nœud adjacent au nœud source, calculer la distance temporaire en passant par le nœud source. Si cette distance est plus courte que la distance actuellement attribuée au nœud adjacent, mettre à jour la distance.

4. Marquage du nœud visité : Marquer le nœud source comme visité, indiquant que la distance la plus courte de la source à ce nœud est trouvée.

5. Sélection du nœud non visité le plus proche : Parmi les nœuds non visités, sélectionner celui avec la distance temporaire la plus courte et le marquer comme visité.

6. Répétition des étapes 3 à 5 : Répéter les étapes d'exploration des nœuds adjacents, de marquage des nœuds visités et de sélection du nœud non visité le plus proche jusqu'à ce que tous les nœuds aient été visités.

7. Fin de l'algorithme : Une fois tous les nœuds visités et que les distances les plus courtes de la source à tous les autres nœuds sont déterminées, l'algorithme est terminé.

À la fin de l'algorithme, les distances les plus courtes et les chemins associés peuvent être extraits pour chaque nœud du graphe par rapport au nœud source initial.

2. Avantage et inconvénient de l'algorithme de Johnson :

2.1. Les avantages

Avantages de l'algorithme de Dijkstra

- ✓ Optimalité garantie : L'algorithme assure la découverte du chemin le plus court entre un nœud source et tous les autres nœuds dans un graphe pondéré.
- ✓ Simplicité d'implémentation : L'algorithme est relativement simple à comprendre et à mettre en œuvre, ce qui en facilite l'utilisation.
- ✓ Adaptabilité à différents types de graphes : Il peut être appliqué avec succès à des graphes dirigés et non dirigés, ainsi qu'à des graphes avec des poids positifs.
- ✓ Principe de fonctionnement intuitif : La logique de l'algorithme, qui explore progressivement les nœuds les plus proches, est intuitive et facile à conceptualiser.

2.2. Les Inconvénients

Inconvénients de l'algorithme de Dijkstra :

- ✓ Sensibilité aux poids négatifs : L'algorithme ne fonctionne pas correctement avec des graphes comportant des poids négatifs, car il suppose que l'addition de distances ne diminue jamais.
- ✓ Complexité temporelle : La complexité temporelle de l'algorithme de Dijkstra est relativement élevée, surtout dans des graphes de grande taille, ce qui peut limiter son efficacité dans certains contextes.
- ✓ Sélection des priorités : Le choix du nœud prioritaire à chaque étape peut être coûteux en termes de temps, surtout sans l'utilisation de structures de données optimisées comme les tas binaires.

- ✓ Inefficace pour les graphes denses : Dans le cas de graphes très denses, où la plupart des nœuds sont connectés les uns aux autres, l'algorithme peut devenir moins efficace en raison du nombre élevé d'explorations nécessaires.

3. Analyse de la complexité :

En général, la complexité temporelle de l'algorithme de Dijkstra est de l'ordre de O((V + E) * log(V)), où V est le nombre de nœuds (sommets) dans le graphe et E est le nombre d'arêtes. Cette complexité est typiquement applicable lorsque l'algorithme utilise une structure de données efficace, telle qu'une file de priorité basée sur un tas binaire.

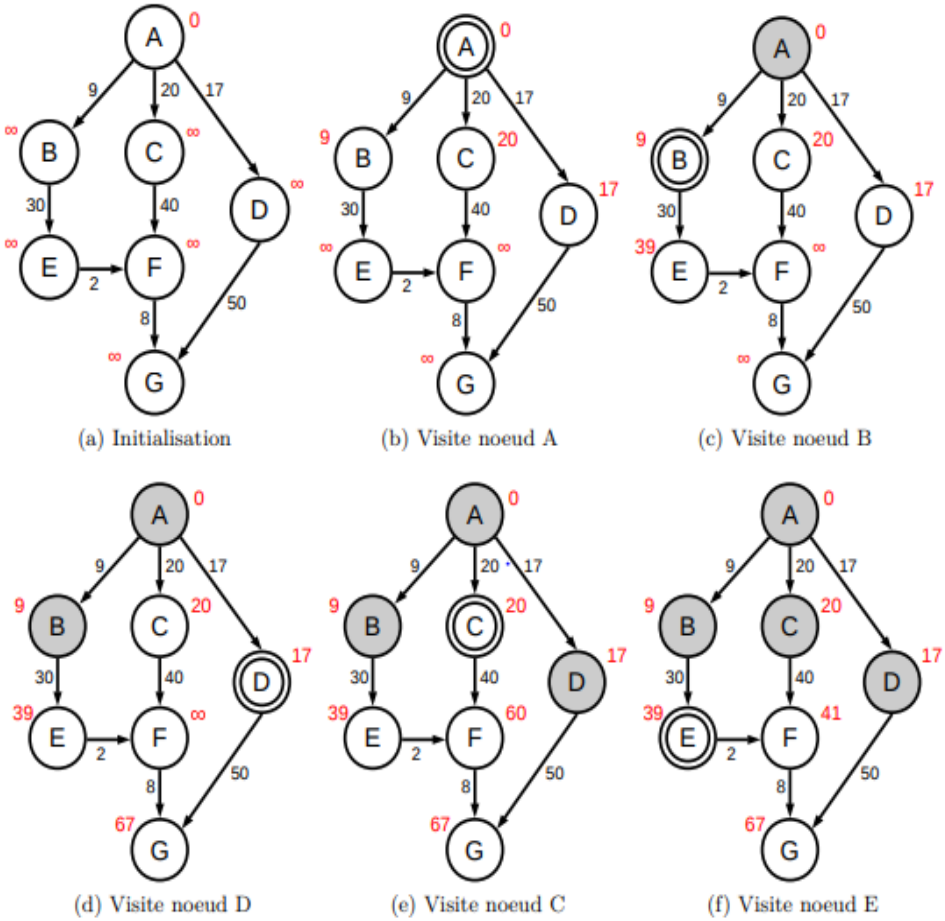
Il est important de noter que dans le pire des cas, lorsque le graphe est dense, c'est-à-dire que le nombre d'arêtes (E) est proche de V^2, la complexité peut être approximativement O(V^2 * log(V)). Cependant, en pratique, l'algorithme de Dijkstra est souvent efficace pour les graphes peu denses ou moyennement denses. L'utilisation de structures de données optimisées, comme les tas binaires ou les tas de Fibonacci, contribue à améliorer la performance de l'algorithme.

Chapitre 2 : Présentation de l'algorithme

1. Introduction

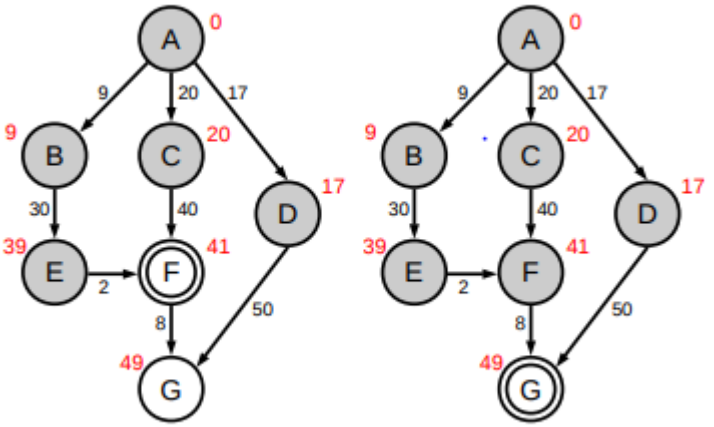
Cette section propose un exemple d'application de l'algorithme de Johnson. Pour appréhender le mécanisme de l'algorithme de Johnson, il est essentiel de se familiariser avec deux autres algorithmes de plus court chemin : l'algorithme de Dijkstra et un autre algorithme.

1.1. Exemple



(a) Initialisation (b) Visite noeud A (c) Visite noeud B

(d) Visite noeud D (e) Visite noeud C (f) Visite noeud E



(a) Visite noeud F (b) Visite noeud d'arrivée : arrêt

La matrice d'adjacence correspondant au graphe est la suivante.

$$\begin{pmatrix} -1 & 9 & 20 & 17 & -1 & -1 & -1 \\ -1 & -1 & -1 & -1 & 30 & -1 & -1 \\ -1 & -1 & -1 & -1 & -1 & 40 & -1 \\ -1 & -1 & -1 & -1 & -1 & -1 & 50 \\ -1 & -1 & -1 & -1 & -1 & 2 & -1 \\ -1 & -1 & -1 & -1 & -1 & -1 & 8 \\ -1 & -1 & -1 & -1 & -1 & -1 & -1 \end{pmatrix}$$

On donne ci-dessous la liste des arcs sortants de chaque nœud.

```
graphe = [[(1,9), (2,20), (3,17)],
[(4,30)],
[(5,40)],
[(6,50)],
[(5,2)], [(6,9)], []]
```

Chapitre 3: implémentation de l’algorithme

1. Introduction ·

L'algorithme de **Dijkstra** est un algorithme classique pour résoudre le problème du plus court chemin entre un nœud de départ et tous les autres nœuds dans un graphe pondéré. L'implémentation de cet algorithme repose sur une approche gourmande qui explore itérativement les nœuds adjacents à partir du nœud source, mettant à jour les distances les plus courtes connues à chaque étape. L'utilisation d'une matrice d'adjacence pondérée représente les poids des arêtes entre les nœuds. La procédure commence par initialiser toutes les distances à l'infini, sauf la distance du nœud source à lui-même, qui est fixée à zéro. Ensuite, elle sélectionne le nœud non visité avec la distance minimale, marque ce nœud comme visité, et met à jour les distances des nœuds adjacents si un chemin plus court est découvert. Ce processus se répète jusqu'à ce que tous les nœuds soient visités. L'implémentation en Python utilise une liste de distances, un tableau booléen pour suivre les nœuds visités, et une boucle principale pour itérer à travers les nœuds du graphe. Enfin, l'algorithme renvoie un tableau des distances les plus courtes du nœud source à tous les autres nœuds. ·

L’application utilise l'algorithme de Dijkstra pour calculer le plus court chemin entre un point de départ et un point d'arrivée, puis trace ce trajet sur la carte GPS en utilisant plusieurs bibliothèques Python.

2. GPS

Le Système de Positionnement Global (GPS) est devenu omniprésent dans nos vies, offrant un moyen précis de déterminer la position géographique sur la Terre. Il fonctionne en utilisant un réseau de satellites qui transmettent des signaux aux récepteurs GPS pour calculer la position en fonction du temps de voyage de ces signaux.

Utilisation de Python pour l'Analyse GPS :

Python offre des bibliothèques puissantes pour travailler avec des données géographiques et des signaux GPS. Des bibliothèques telles que geopy, folium, et gpsd peuvent être utilisées pour extraire, analyser et visualiser des informations de localisation.

Importance pour Identifier la Localisation Actuelle :

Navigation : Le GPS est essentiel à la navigation moderne. Que ce soit pour guider les conducteurs sur la route, les randonneurs sur un sentier, ou même pour les navires en mer, le GPS fournit des informations cruciales sur la position actuelle.

Applications Mobiles : De nombreuses applications mobiles utilisent le GPS pour des services de localisation, que ce soit pour des recommandations géographiques, le suivi de fitness, ou la recherche de points d'intérêt à proximité.

Sécurité : Le GPS joue un rôle important dans les services d'urgence et de secours, permettant une localisation précise des personnes nécessitant de l'aide.

En conclusion Intégrer le GPS avec Python offre des opportunités considérables pour l'analyse et la compréhension de la localisation. Que ce soit pour des applications pratiques du quotidien ou des projets plus avancés, l'utilisation du GPS avec Python ouvre la porte à une gamme diversifiée d'innovations et d'améliorations.

3. Implémentationn

Les bibliothèques utilisées.

osmnx : Cette bibliothèque est utilisée pour récupérer des données OSM, construire et analyser des graphes de réseau routier.

networkx : Une bibliothèque Python pour la création, la manipulation et l'étude de la structure, la dynamique et les fonctions de réseaux complexes.

folium : Une bibliothèque de visualisation de cartes interactives.

heapq : Un module Python pour implémenter des files de priorité sous forme de tas binaires.

geopy : Une bibliothèque pour géocoder et calculer les distances géographiques

```
Entrée [1]: #importation des modèles nécessaires
import osmnx as ox
import networkx as nx
import folium
from heapq import heappop, heappush
from geopy.distance import geodesic
import time
```

Utilisation de la fonction ox.graph_from_place() pour télécharger le graphe du réseau routier pour une région spécifique (ici, Fès, Maroc) à partir des données OpenStreetMap.

```
Entrée [2]: # Téléchargement du graphe du réseau routier pour une région spécifique (ici, Fès, Maroc)
place_name = "Fès, Morocco"
graph = ox.graph_from_place(place_name, network_type='drive')
```

Dans ce code on a stocker les nœuds de notre graphe dans une variable nommée: **tous_les_noeuds**

```
Entrée [3]: #Obtenir tous les nœuds :
tous_les_noeuds = graph.nodes
print("Tous les nœuds :", tous_les_noeuds)
```

```
0, 260148091, 260148404, 260148405, 260148407, 260148408, 260148409, 260148441, 260148457, 260148543, 260148545, 260148547, 2
60148607, 260148609, 260148739, 260148740, 260148741, 260148742, 260148743, 260148811, 260148903, 260148968, 260148969, 26014
9038, 260149039, 260149043, 260149049, 260149130, 260149304, 260149740, 260151170, 260151365, 260151366, 260151668, 26015167
4, 260151701, 260151704, 260151705, 260151709, 260151710, 260151711, 260152121, 260152235, 260152244, 260152563, 260152564, 2
60157217, 260157237, 260157738, 260157757, 260157768, 260157940, 260158280, 260158281, 260158283, 260158719, 260158725, 26015
8727, 260257373, 260257440, 260257443, 260257457, 260257464, 260257521, 260257532, 260257583, 260257823, 260257981, 26025798
2, 260258022, 260258194, 260258195, 260258197, 260258266, 260258290, 260258714, 260258859, 260258902, 260258928, 260258933, 2
60258985, 260258986, 260259012, 260259036, 260259038, 260259078, 260259079, 260259089, 260259657, 260259931, 260259932, 26025
9935, 260260720, 260260721, 260261030, 260261031, 260261032, 260261034, 260261037, 260261038, 260261043, 260261049, 26026105
5, 260261250, 260261252, 260261253, 260261256, 260261375, 260261488, 260261934, 260261944, 260262053, 260262492, 260262497, 2
60262500, 260262502, 260262503, 260262511, 260262567, 260262577, 260262663, 260262667, 260262673, 260262871, 260263278, 26026
3279, 260263939, 260266024, 260266027, 260266029, 260266053, 260634517, 260808470, 260808473, 260808475, 260808478, 26080847
9, 262199760, 262199761, 262199762, 262199763, 262199779, 262199853, 262199854, 262199855, 262199859, 262200104, 262200105, 2
62200106, 262200175, 262200177, 262200178, 262200179, 262200183, 262200258, 262200363, 262200464, 262200512, 262200513, 26220
0515, 262200523, 262200524, 262200525, 262200591, 262200603, 262200604, 262200655, 262200656, 262200657, 262200658, 26220065
9, 262200675, 262200676, 262200707, 262200731, 262200732, 262200733, 262200797, 262200798, 262200799, 262200801, 262200939, 2
62200940, 262200941, 262200945, 262200946, 262200948, 262200998, 262201000, 262201206, 262201207, 262201208, 262201221, 26220
1479, 262201627, 262201628, 262201629, 262201630, 262201632, 262201636, 262201666, 262201676, 262201677, 262201678, 26220167
9, 262201680, 262201681, 262201682, 262201683, 262201731, 262201733, 262201749, 262201786, 262201787, 262201795, 262201837, 2
```

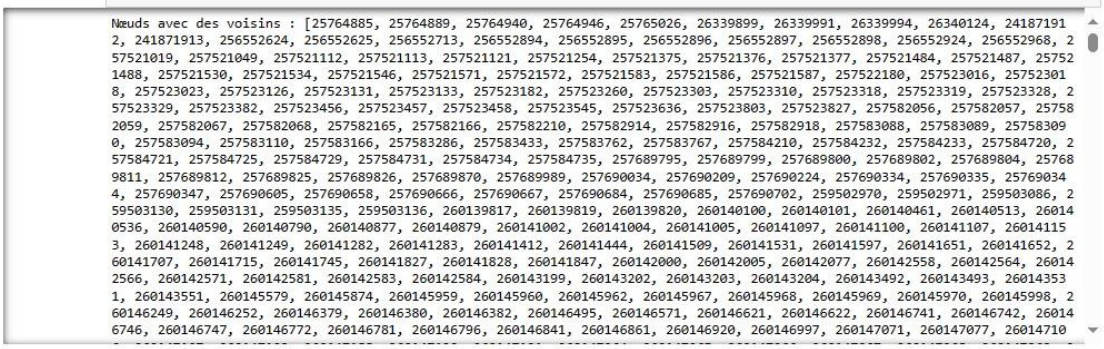
Pour ce code, nous avons récupéré les liens de notre graphe dans une variable nommée corriger.: **tous_les_liens**

```
Entrée [4]: #Obtenir tous les liens :
tous_les_liens = graph.edges
print("Tous les liens :", tous_les_liens)
```

```
Tous les liens : [(25764885, 1037919515, 0), (25764889, 1584322396, 0), (25764889, 1037919694, 0), (25764940, 2924737137, 0),
(25764940, 6840622011, 0), (25764946, 1098650628, 0), (25764946, 3468119373, 0), (25765026, 6838083998, 0), (25765026, 573765
8124, 0), (26339899, 1584708271, 0), (26339899, 1586999941, 0), (26339991, 1373970631, 0), (26339991, 262474151, 0), (2633999
4, 549305756, 0), (26339994, 26339991, 0), (26340124, 6586561192, 0), (26340124, 5528379672, 0), (26340124, 3364810564, 0),
(241871912, 1037923780, 0), (241871912, 8395496781, 0), (241871912, 241871913, 0), (241871912, 1700097074, 0), (241871913, 10
37923911, 0), (241871913, 241871912, 0), (256552624, 256552894, 0), (256552624, 6802211068, 0), (256552624, 864885870, 0), (2
56552625, 5659650870, 0), (256552625, 257521121, 0), (256552625, 5742127325, 0), (256552713, 1605921989, 0), (256552713, 2576
90605, 0), (256552894, 5743656676, 0), (256552894, 256552624, 0), (256552894, 5743656671, 0), (256552895, 864594225, 0), (256
552895, 5743656671, 0), (256552895, 6802211021, 0), (256552895, 256552896, 0), (256552896, 864594251, 0), (256552896, 8648858
27, 0), (256552896, 256552895, 0), (256552897, 7108311941, 0), (256552897, 257690034, 0), (256552897, 864885951, 0), (2565528
98, 6932774755, 0), (256552898, 864912918, 0), (256552898, 864594241, 0), (256552924, 864610970, 0), (256552924, 864610981,
0), (256552924, 257689826, 0), (256552924, 6802135536, 0), (256552968, 864594203, 0), (256552968, 864885976, 0), (256552968,
864594256, 0), (257521019, 5743898567, 0), (257521049, 257690685, 0), (257521049, 864912453, 0), (257521049, 4579373737, 0),
(257521049, 864912464, 0), (257521112, 864611100, 0), (257521112, 6805659340, 0), (257521113, 4579374323, 0), (257521113, 257
521112, 0), (257521121, 256552625, 0), (257521121, 2924735532, 0), (257521254, 864912621, 0), (257521254, 4423465933, 0), (25
7521254, 4760768540, 0), (257521375, 864912515, 0), (257521375, 257521530, 0), (257521375, 7108311918, 0), (257521376, 864912
709, 0), (257521376, 864912391, 0), (257521376, 257521377, 0), (257521377, 4760768530, 0), (257521377, 4806325706, 0), (25752
1377, 257521376, 0), (257521377, 9708890163, 0), (257521484, 4760768550, 0), (257521484, 5743898467, 0), (257521487, 47607685
53, 0), (257521487, 9708982787, 0), (257521487, 864594419, 0), (257521487, 4760768579, 0), (257521488, 4579373928, 0), (25752
```

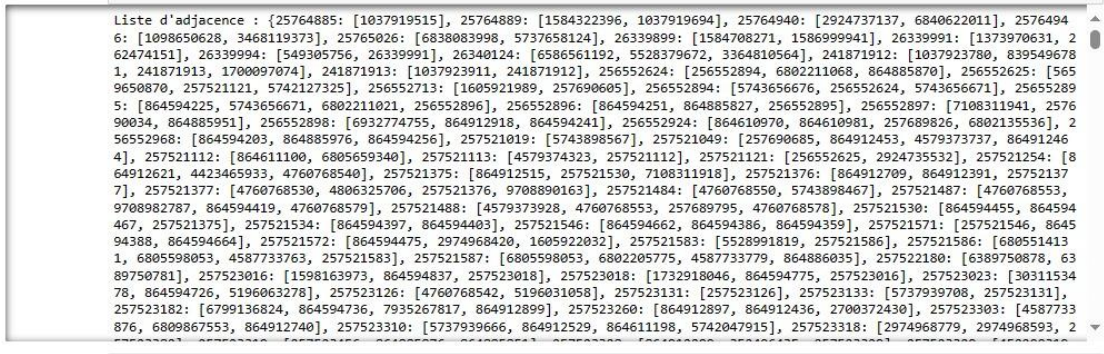
Dans cette partie de code on a comme resultat de cette boucle une variable qui contient les nœuds du graph avec ces voisins

```
Entrée [5]: #Obtenir les nœuds avec des voisins :
noeuds_avec_voisins = [noeud for noeud, degre in graph.degree() if degre > 0]
print("Nœuds avec des voisins :", noeuds_avec_voisins)
```



Le resultat de ce code est une liste d'adjacence .La liste d'adjacence associe chaque nœud du graphe à la liste de ses nœuds adjacents

```
Entrée [6]: #Obtenir la liste d'adjacence :
liste_adjacence = {noeud: list(graph.neighbors(noeud)) for noeud in graph.nodes}
print("Liste d'adjacence :", liste_adjacence)
```



Implémentation de l'algorithme de Dijkstra pour trouver le chemin le plus court entre deux nœuds dans le graphe

Distances : Un dictionnaire pour stocker la distance minimale du nœud de départ à chaque nœud. Initialisé avec l'infini pour tous les nœuds sauf le nœud de départ, qui est fixé à 0.

Tas : Une file de priorité (implémentée comme un tas min) pour suivre les nœuds et leurs distances actuelles depuis le nœud de départ.

previous : Un dictionnaire pour stocker le nœud précédent dans le chemin le plus court du début à chaque nœud. Initialisé avec None pour tous les nœuds.

Boucle principale :

L'algorithme utilise une file de priorité pour sélectionner de manière vorace le nœud avec la distance minimale et explore ses voisins.

Dépiler le nœud avec la plus petite distance actuelle du tas.

Si le nœud déplié est le nœud de destination (fin), reconstruire le chemin du début à la fin en utilisant le dictionnaire previous et le renvoyer.

Pour chaque voisin du nœud actuel, calculer la distance au voisin depuis le début et mettre à jour la distance si elle est plus courte que la distance connue actuelle. Mettre également à jour le dictionnaire previous.

Ajouter le voisin au tas avec sa distance mise à jour.

Reconstruction du chemin :

Si le nœud de destination est atteint, l'algorithme reconstruit le chemin le plus court en remontant du nœud de destination au nœud de départ en utilisant le dictionnaire previous.

Entrée [42]:

```python
def my_dijkstra(graph, start, end):
    # Initialisation des distances à l'infini pour chaque nœud
    distances = {node: float('inf') for node in graph.nodes}
    distances[start] = 0  # Distance de départ à 0
    heap = [(0, start)]  # Tas min pour stocker les nœuds non visités et leurs distances
    previous = {node: None for node in graph.nodes}  # Dictionnaire pour stocker les prédécesseurs de chaque nœud dans le ch

    # Tant que le tas n'est pas vide
    while heap:
        current_distance, current_node = heappop(heap)  # Extraire le nœud avec la distance minimale actuelle

        # Si le nœud actuel est le nœud de destination, reconstruire et retourner le chemin
        if current_node == end:
            path = []
            while current_node is not None:
                path.append(current_node)  # Ajouter le nœud au chemin
                current_node = previous[current_node]  # Remonter au nœud précédent

            return path[::-1]  # Retourner le chemin dans le bon ordre (du départ à la destination)

        # Parcourir tous les voisins du nœud actuel
        for neighbor, _ in graph[current_node].items():
            # Calcul de la distance entre les nœuds voisins en utilisant leurs coordonnées géographiques
            point1 = (graph.nodes[current_node]['y'], graph.nodes[current_node]['x'])
            point2 = (graph.nodes[neighbor]['y'], graph.nodes[neighbor]['x'])
            distance = current_distance + geodesic(point1, point2).meters
            # Mettre à jour la distance si le nouveau chemin est plus court que l'ancien
            if distance < distances[neighbor]:
                distances[neighbor] = distance
                previous[neighbor] = current_node  # Mettre à jour le prédécesseur du voisin
                heappush(heap, (distance, neighbor))  # Ajouter le voisin et sa nouvelle distance au tas

    # Si aucun chemin n'a été trouvé
    return None
```

Le code ci dessous contient le point de départ et point d'arrivée.et le temps d’exécution de notre algorithme et aussi le calcul de plus court chemin par la fonction de Dijkstra et l’affichage de ces deux information

```
: #(34.0136774, -5.0029647)
start_node =5754364428

# (34.0200014, -4.9749313)
end_node =  256552894
```

```
: # Mesurer le temps d'exécution de l'algorithme
start_time = time.time()

# Calcul du chemin le plus court (utilisation de l'algorithme de Dijkstra)
shortest_path = my_dijkstra(graph, start_node, end_node)
end_time = time.time()

# Afficher le chemin le plus court et le temps d'exécution
print("Chemin le plus court :", shortest_path)
print("\n\nTemps d'exécution de l'algorithme :", end_time - start_time, "secondes")
```

```
Chemin le plus court : [5754364428, 5754364424, 1587596369, 3364810700, 1587596374, 3968626723, 6586561194, 3968626718, 3968626
720, 5741838729, 5752437138, 5752437137, 5752437136, 5752437135, 5752437134, 5752437133, 5752437132, 5752437131, 5741402750, 57
52552869, 5752552866, 5752437129, 5752437128, 5752437127, 5752437126, 5752436909, 6928685607, 2924736995, 1688028933, 168802824
1, 1688028844, 2924735760, 6805659321, 2924736467, 6805659336, 6805630959, 5754813800, 5754813802, 2924737194, 2924735758, 2575
21121, 256552625, 5742127325, 5742127324, 257690702, 5659650464, 6802062989, 864886136, 864610963, 352406429, 864886048, 864610
972, 864885870, 256552624, 256552894]


Temps d'exécution de l'algorithme : 0.08605217933654785 secondes
```

À partir du chemin le plus court trouvé par la fonction de Dijkstra, nous procédons maintenant à la récupération des coordonnées des nœuds de ce chemin.

```
Entrée [10]: # Récupération des coordonnées des nœuds du chemin le plus court
coordinates = [(graph.nodes[node]['y'], graph.nodes[node]['x']) for node in shortest_path]

print("coordinates: ",coordinates)
```

```
coordinates:  [(34.0136774, -5.0029647), (34.013741, -5.0029316), (34.0141826, -5.0013992), (34.0143649, -5.0006461), (34.01469
24, -4.9991324), (34.0152652, -4.9966613), (34.0152607, -4.996655), (34.0152597, -4.9964852), (34.0152868, -4.9964551), (34.015
5563, -4.9957186), (34.0157136, -4.9952233), (34.0158124, -4.9949059), (34.0159949, -4.9943396), (34.016174, -4.9937677), (34.0
162629, -4.9934741), (34.0164052, -4.9929978), (34.0164827, -4.9927233), (34.0166514, -4.9920255), (34.0168854, -4.9910141), (3
4.0169731, -4.9906318), (34.0171187, -4.9899638), (34.0171697, -4.9897452), (34.0172445, -4.9894251), (34.0173188, -4.9891033),
(34.0173897, -4.9887991), (34.0175054, -4.9883861), (34.0176373, -4.9882153), (34.0191803, -4.9867622), (34.019175, -4.986714
7), (34.0192778, -4.9865214), (34.0193474, -4.9865041), (34.0197115, -4.9857499), (34.0199297, -4.985254), (34.0201036, -4.9848
359), (34.020448, -4.9840083), (34.0206176, -4.9835913), (34.0209281, -4.9823765), (34.0209275, -4.9823118), (34.020727, -4.981
5135), (34.0206874, -4.9814762), (34.0203435, -4.9812975), (34.0202517, -4.9813055), (34.0202296, -4.9809412), (34.020247, -4.9
804323), (34.02025, -4.9800473), (34.0202733, -4.9792803), (34.0202821, -4.9787836), (34.0202835, -4.9787089), (34.0202892, -4.
9784322), (34.0202985, -4.9779011), (34.0203116, -4.9769972), (34.0203185, -4.976287), (34.0203242, -4.9760476), (34.0203353, -
4.974938), (34.0200014, -4.9749313)]
```

Ce code calcule la distance du chemin à partir des coordonnées trouvées pour ce chemin et On a utiliser la methode snapping pour tracer le chemin sur les routes reelles.

```python
# Calculate the distance between consecutive points in the path
total_distance = 0
for i in range(len(coordinates) - 1):
    point1 = coordinates[i]
    point2 = coordinates[i + 1]
    total_distance += geodesic(point1, point2).meters

print("Distance totale du chemin :", total_distance, "mètres")
```

```
Distance totale du chemin : 2877.187587336111 mètres
```

```python
# Création de la carte Folium centrée sur Fès, Maroc
m = folium.Map(location=(34.0339, -5.0003), zoom_start=13)
```

```python
# Tracer le chemin le plus court sur la carte

# Utiliser la méthode 'snapping' pour obtenir le chemin sur les routes réelles
snapped_coordinates = ox.plot_route_folium(graph, shortest_path, route_map=m, popup_attribute=None)
```

Pour cette partie, nous avons tracé le chemin obtenu sur la carte et marqué le point de départ et le point d'arrivée avec des couleurs différentes.

```python
# Ajouter un marqueur pour le point de départ (bleu)
start_point = graph.nodes[start_node]['y'], graph.nodes[start_node]['x']
folium.Marker(location=start_point, popup='Point de départ', icon=folium.Icon(color='blue')).add_to(m)
```

```
<folium.map.Marker at 0x1c5df0ee310>
```

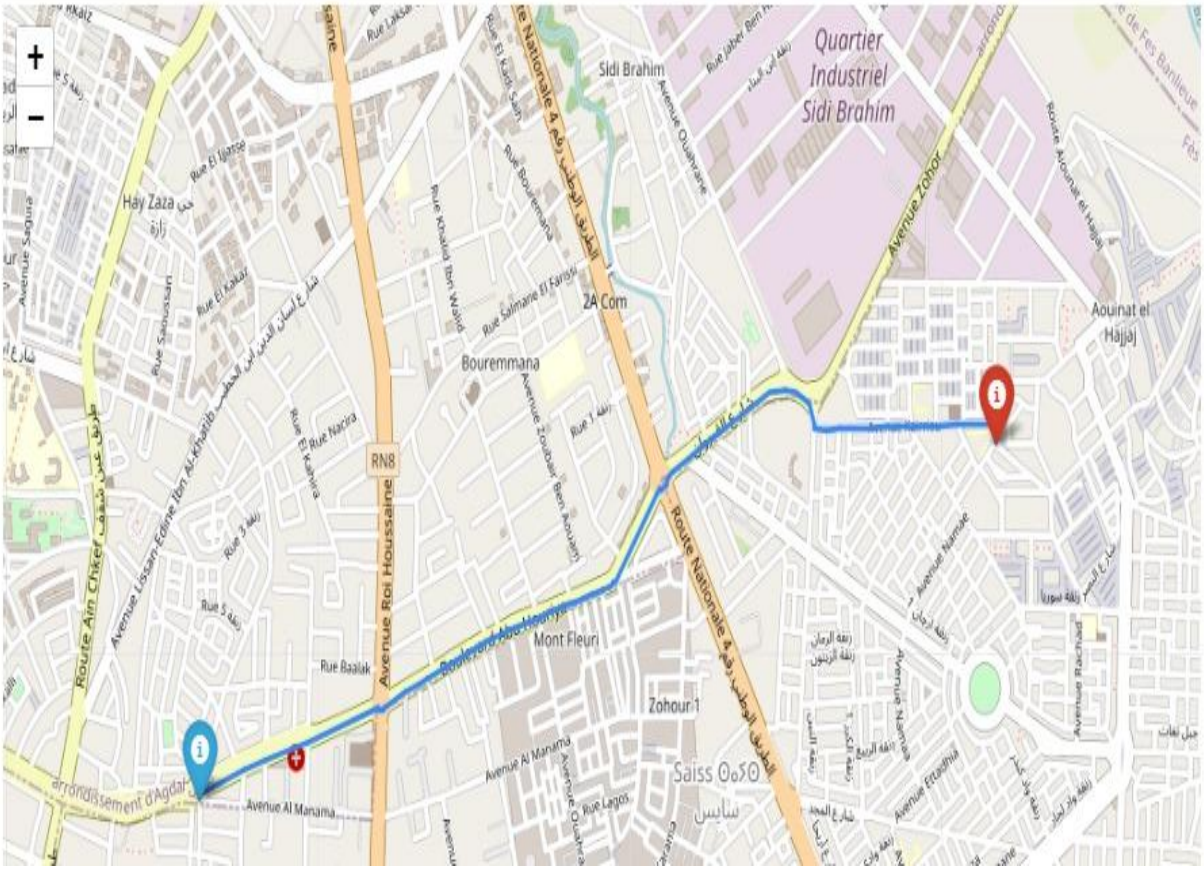
```python
# Ajouter un marqueur pour le point d'arrivée (rouge)
end_point = graph.nodes[end_node]['y'], graph.nodes[end_node]['x']
folium.Marker(location=end_point, popup='Point d\'arrivée', icon=folium.Icon(color='red')).add_to(m)
```

```
<folium.map.Marker at 0x1c5d07b66d0>
```

Voici le résultat sur la carte :

Entrée [17]:
```
# Affichage de la carte avec le chemin le plus court, le point de départ et le point d'arrivée
m
```

Out[17]:



Conclusion générale

L'algorithme de Dijkstra, salué pour sa simplicité et son efficacité, est un joyau de la résolution de problèmes de plus courts chemins dans les graphes pondérés. Son approche gourmande et itérative assure l'optimalité des résultats, ce qui le rend incontournable pour la planification d'itinéraires, la gestion logistique, et d'autres domaines nécessitant des chemins les plus courts. Sa capacité à s'adapter à différentes structures de graphes, qu'ils soient dirigés ou non, avec des poids positifs, en fait un outil polyvalent. L'algorithme offre une solution rapide et fiable pour trouver les chemins optimaux, ce qui le rend précieux dans des applications allant de la navigation automobile à la conception de réseaux de communication. En résumé, l'algorithme de Dijkstra se distingue par sa simplicité d'implémentation, son adaptabilité et son impact significatif dans des domaines variés, en apportant des solutions optimales aux défis des plus courts chemins.

Outre l'algorithme de Dijkstra, plusieurs autres algorithmes sont utilisés pour calculer le plus court chemin dans un graphe. Voici quelques-uns d'entre eux :

Algorithme de Bellman-Ford : Similaire à Dijkstra, il fonctionne même avec des graphes contenant des arêtes de poids négatif. Cependant, sa complexité est supérieure à celle de Dijkstra.

Algorithme A* : Utilisé principalement dans la recherche de chemins dans les domaines de l'intelligence artificielle et de la robotique, A* combine la recherche heuristique et la recherche de coût pour trouver un chemin optimal.

Algorithme de Floyd-Warshall : Conçu pour trouver les chemins les plus courts entre tous les paires de nœuds dans un graphe, il fonctionne également avec des arêtes de poids négatif, mais sa complexité est plus élevée que celle de Dijkstra.

Algorithme de Johnson : Utilise la transformation de Johnson pour gérer les poids négatifs et applique ensuite Dijkstra ou Bellman-Ford pour trouver les plus courts chemins.

Algorithme de Yen : Une extension de Dijkstra ou A* qui permet de trouver les K plus courts chemins entre deux nœuds dans un graphe.

Algorithme de Bidirectional Dijkstra : Une variante de Dijkstra qui explore simultanément à partir du nœud source et du nœud destination, réduisant ainsi le nombre d'explorations nécessaires.

Algorithme de Suurballe :Modifie l'algorithme de Dijkstra pour gérer les graphes dirigés avec des arêtes négatives en doublant les arêtes et en les transformant en arêtes positives.

Chaque algorithme a ses avantages et ses inconvénients, et le choix dépend des spécificités du problème à résoudre et des caractéristiques du graphe.