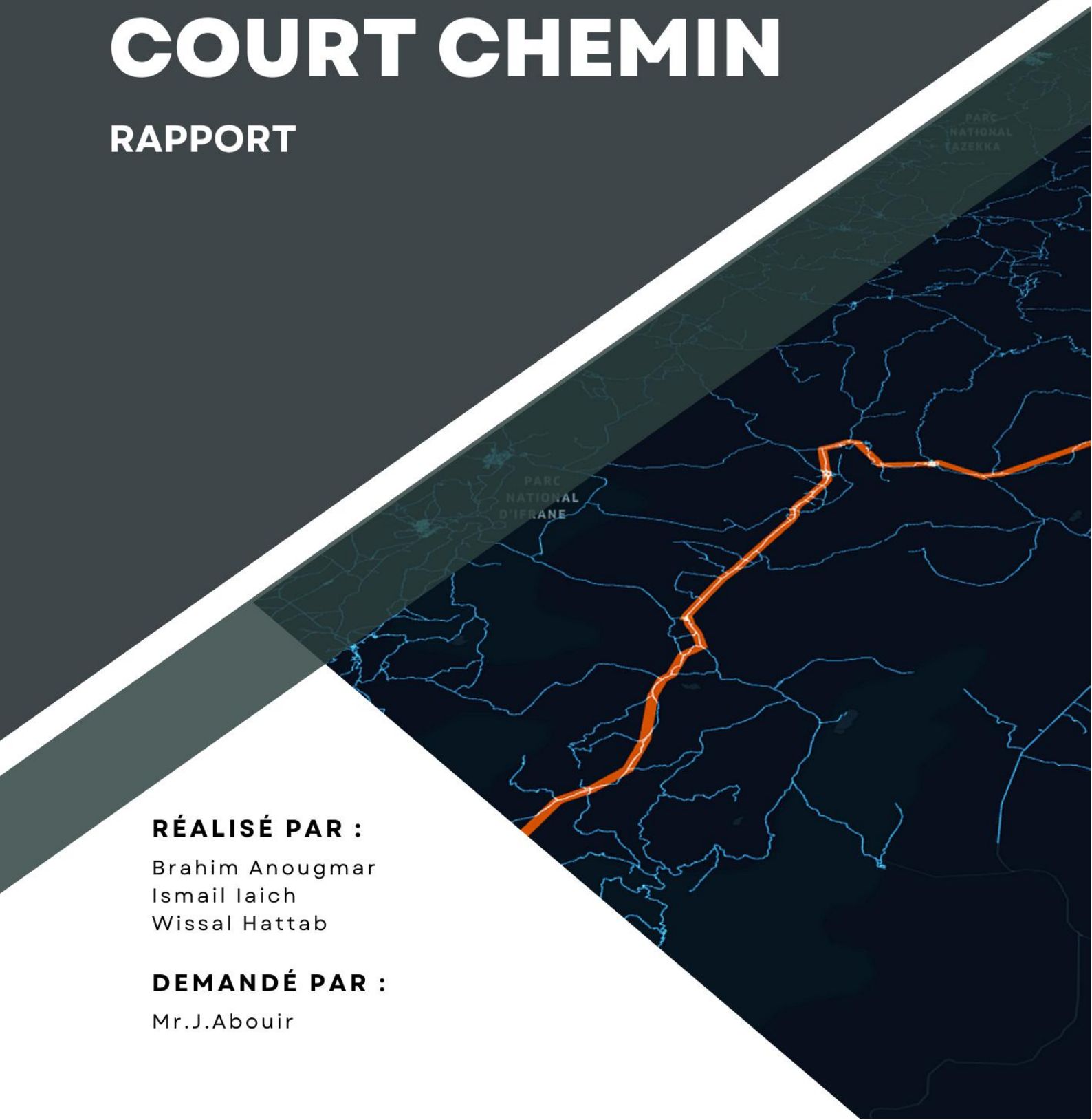
FACULTÉ DES SCIENCES ET TECHNIQUES
DE MOHAMMEDIA
UNIVERSITÉ HASSAN II DE CASABLANCA

PLUS COURT CHEMIN

RAPPORT



RÉALISÉ PAR :

Brahim Anougmar
Ismail Iaich
Wissal Hattab

DEMANDÉ PAR :

Mr.J.Abouir

Remerciement

Avant de commencer la présentation de notre rapport

Nous tenons à exprimer notre profonde gratitude envers notre enseignant, **M. J. Abouir**, pour nous avoir donné l'opportunité d'étudier et de mettre en pratique les concepts des algorithmes du plus court chemin, notamment l'algorithme de Dijkstra et l'algorithme de Bellman-Ford. Son soutien et ses explications détaillées ont grandement contribué à notre compréhension de ces sujets complexes. Son dévouement à notre apprentissage est sincèrement apprécié et a enrichi notre expérience académique en tant que groupe.

Nous lui sommes reconnaissants pour son encouragement constant et son engagement envers notre succès académique.

Table des matières

Liste Des Figures

Introduction

La résolution du problème du plus court chemin est essentielle dans de nombreux domaines, de la logistique à l'ingénierie des réseaux. Elle consiste à trouver le chemin optimal entre deux points dans un réseau ou un graphe, en tenant compte des coûts associés à chaque arête ou lien.

La recherche opérationnelle, apparue dans les années 1940, vise à maximiser l'efficacité de l'utilisation des ressources dans divers domaines, de l'industrie à la planification militaire. Son développement s'est accéléré avec l'avènement de l'ordinateur, permettant une résolution plus rapide et efficace des problèmes complexes.

Dans cette étude, nous examinerons deux algorithmes clés pour résoudre le problème du plus court chemin : l'algorithme de **Dijkstra** et l'algorithme de **Bellman-Ford**. Nous explorerons leur fonctionnement, leurs avantages et leurs limitations, ainsi que leur mise en œuvre pratique en Python.

Chapitre 1

Introduction aux Graphes

Les graphes, en informatique et en mathématiques, sont des structures de données qui représentent des relations entre des objets. Ils sont largement utilisés pour modéliser des problèmes dans divers domaines, de la logistique à la biologie en passant par les réseaux sociaux.

1 Définitions de base :

Un graphe est composé de nœuds (ou sommets) et d'arêtes (ou liens) qui relient ces nœuds. Ces liens peuvent représenter des connexions ou des relations entre les nœuds. Le graphe est souvent noté G = (S, A), où S représente l'ensemble des sommets (ou nœuds) du graphe et A représente l'ensemble des arêtes (ou liens) qui les relient.

Les graphes peuvent être dirigés, où les arêtes ont une direction, ou non dirigés, où les arêtes ne sont pas directionnelles.

Dans un graphe orienté, on appelle circuit une suite d'arcs consécutifs (chemin) dont les deux sommets extrémités sont identiques. La notion correspondante dans les graphes non orientés est celle de cycle. On parle parfois de cycle orienté. Un circuit constitué d'un seul arc est une boucle.

Un circuit absorbant dans un graphe est un circuit dans lequel la somme totale des poids des arêtes est négative. En d'autres termes, c'est un chemin qui forme une

boucle fermée et dont la somme des poids des arêtes le long de ce chemin est négative. Un tel circuit est également appelé circuit de poids négatif. Les circuits absorbants peuvent poser des problèmes dans certaines applications, notamment dans le cadre de l'algorithme de recherche du plus court chemin, car ils peuvent induire en erreur les calculs de chemins optimaux en présence de poids négatifs.

2 Types de graphes :

Les graphes peuvent être simples, où chaque paire de nœuds est reliée par au plus une arête, ou multigraphes, où des paires de nœuds peuvent être reliées par plusieurs arêtes.

Ils peuvent également être pondérés, où chaque arête a un poids associé, ou non pondérés, où les arêtes n'ont pas de poids.

Les graphes peuvent être orientés, où les arêtes ont une direction définie, ou non orientés, où les arêtes ne sont pas directionnelles.

On distingue également les graphes avec des circuits absorbants, qui contiennent au moins un circuit avec un poids total négatif.

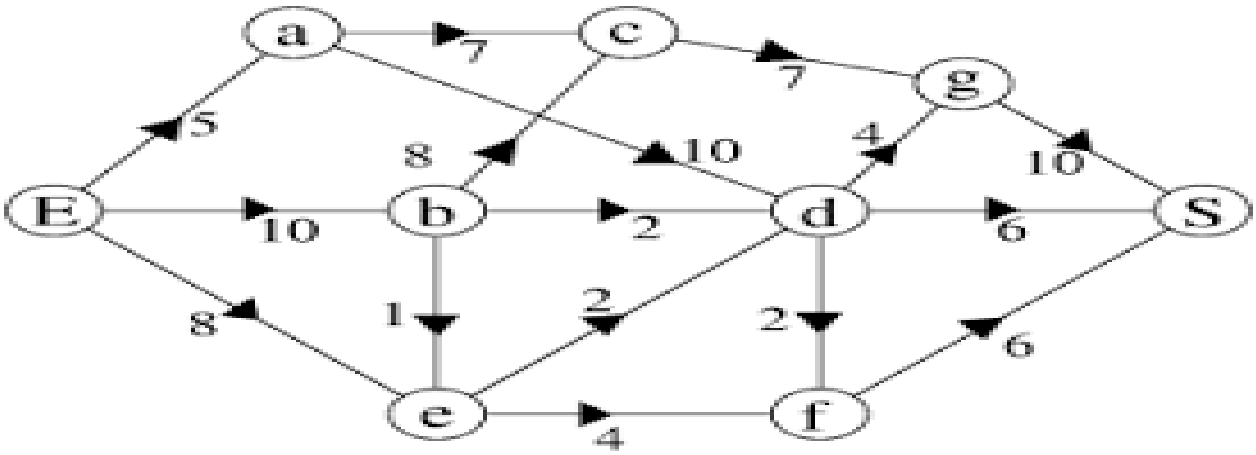


Figure 1: Graphe orienté pondéré

3 Les Graphes dans python :

Les graphes sont des structures de données fondamentales en informatique, utilisées pour représenter des relations entre des objets. Python offre plusieurs bibliothèques puissantes pour travailler avec des graphes, notamment NetworkX, igraph et graph-tool. Dans ce chapitre, nous explorerons l'utilisation de la bibliothèque NetworkX pour la manipulation et l'analyse de graphes. Nous présenterons les fonctionnalités de base de NetworkX, ainsi que des exemples d'utilisation pour créer, visualiser et analyser des graphes.

➢ Fonctionnalités de NetworkX :

NetworkX est une bibliothèque Python utilisée pour la création, la manipulation et l'analyse de structures de réseau complexes. Parmi ses fonctionnalités principales, on retrouve la possibilité de représenter des graphes non orientés et orientés, des graphes pondérés et non pondérés, ainsi que des multigraphes. De plus, NetworkX propose des outils pour générer des graphes aléatoires selon différents modèles et des algorithmes pour résoudre des problèmes courants liés aux graphes, tels que le parcours de graphes, la recherche de chemins les plus courts, et bien d'autres.

➢ Création et Manipulation de Graphes :

Grâce à NetworkX, il est possible de créer des graphes de manière intuitive en ajoutant des nœuds et des arêtes, ainsi que de manipuler ces structures en ajoutant, supprimant ou modifiant des éléments. De plus, NetworkX offre des fonctions permettant d'importer et d'exporter des graphes à partir de divers formats de fichiers, facilitant ainsi l'échange de données avec d'autres outils ou systèmes.

Pour représenter un graphe avec NetworkX, vous pouvez utiliser les fonctions suivantes :

- nx.Graph(): Pour créer un graphe non orienté.
- nx.DiGraph(): Pour créer un graphe orienté.

Vous pouvez ajouter des nœuds et des arêtes au graphe en utilisant les méthodes suivantes :

- G.add_node(): Ajoute un nœud au graphe.
- G.add_edge(): Ajoute une arête entre deux nœuds.

Pour manipuler les graphes, vous disposez des fonctions suivantes :

- G.remove_node(): Supprime un nœud du graphe.
- G.remove_edge(): Supprime une arête du graphe.

➢ Visualisation de Graphes :

NetworkX inclut des outils de visualisation pour représenter graphiquement les graphes créés. Ces fonctionnalités de visualisation permettent d'explorer et de comprendre la structure des graphes de manière visuelle, facilitant ainsi l'analyse des données et la communication des résultats.

Pour visualiser un graphe avec NetworkX, vous pouvez utiliser la fonction suivante :

- nx.draw(): Dessine le graphe.

➢ Exemple d'utilisation :

```python
import networkx as nx
import matplotlib.pyplot as plt

# Création d'un graphe orienté et pondéré
G = nx.DiGraph()

# Ajout de nœuds
G.add_nodes_from([1, 2, 3, 4])

# Ajout d'arêtes avec des poids
G.add_weighted_edges_from([(1, 2, 5), (2, 3, 3), (3, 4, 7), (4, 1, 2)])

# Dessin du graphe avec les poids des arêtes
pos = nx.circular_layout(G)  # Positionnement des nœuds
labels = nx.get_edge_attributes(G, 'weight')  # Récupération des poids des arêtes
nx.draw(G, pos, with_labels=True, node_size=1000, node_color="skyblue")  # Dessin des nœuds
nx.draw_networkx_edge_labels(G, pos, edge_labels=labels)  # Dessin des poids des arêtes
plt.show()
```

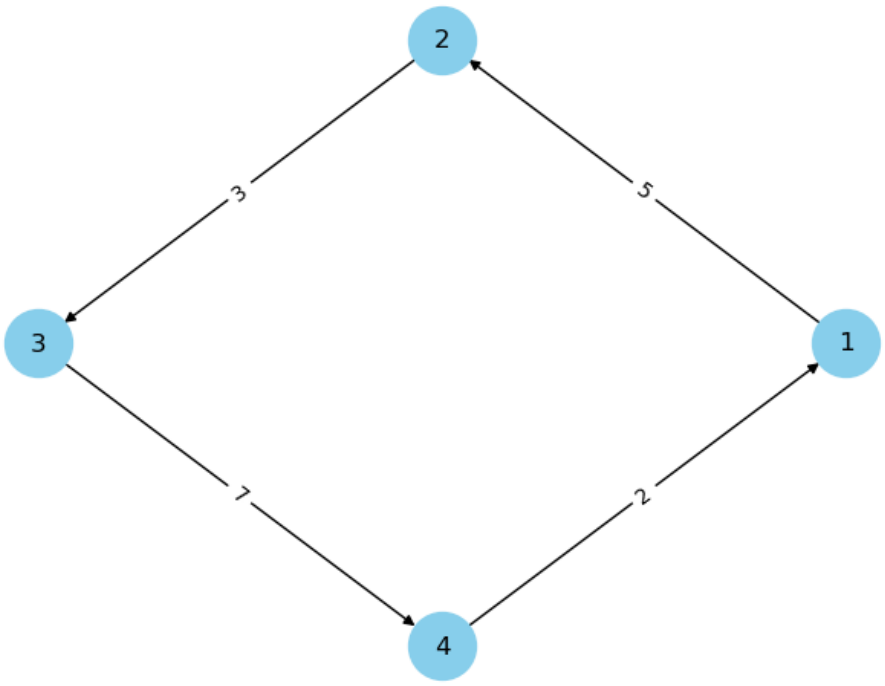


Figure 2: Définir un graphe dans python

Chapitre 2

Algorithme de Dijkstra

L'algorithme de Dijkstra, proposé par l'informaticien néerlandais Edsger W. Dijkstra en 1956, est un algorithme classique de recherche de plus court chemin dans un graphe pondéré, non orienté ou orienté, avec des poids non négatifs sur les arêtes.

L'algorithme de Dijkstra est largement utilisé dans les réseaux de télécommunication, les systèmes de cartographie, et d'autres applications nécessitant la recherche de chemins optimaux dans un graphe. Son efficacité réside dans sa capacité à trouver rapidement le plus court chemin entre deux nœuds dans un graphe pondéré. Cet algorithme calcule le chemin le plus court entre un nœud source et tous les autres nœuds du graphe.

Dans ce chapitre, nous allons présenter l'algorithme de Dijkstra, détailler son fonctionnement étape par étape, et fournir des exemples d'implémentation en Python.

1 Fonctionnement de l'algorithme :

L'algorithme de Dijkstra fonctionne selon une approche itérative pour trouver le chemin le plus court entre un nœud source et tous les autres nœuds dans un graphe pondéré. Voici un aperçu détaillé de son fonctionnement :

1) Initialisation :

-L'algorithme commence par initialiser la distance de chaque nœud depuis le nœud source à l'infini, sauf pour le nœud source lui-même dont la distance est mise à zéro.

-Il initialise également un ensemble de nœuds visités vide.

2) Sélection du Nœud Courant :

-À chaque itération, l'algorithme sélectionne le nœud non visité avec la plus petite distance depuis le nœud source.

3) Mise à Jour des Distances :

-Pour chaque nœud voisin du nœud courant, l'algorithme calcule la distance minimale jusqu'à ce voisin en passant par le nœud courant.

-Si cette distance calculée est plus petite que la distance actuellement enregistrée pour le voisin, la distance est mise à jour.

4) Marquage du Nœud Courant comme Visité :

-Une fois que toutes les distances des nœuds voisins ont été mises à jour, le nœud courant est marqué comme visité et est retiré de l'ensemble des nœuds non visités.

5) Répétition :

-Les étapes 2 à 4 sont répétées jusqu'à ce que tous les nœuds aient été visités ou jusqu'à ce que la destination souhaitée soit atteinte.

6) Retour des Résultats :

-Une fois que l'algorithme a terminé l'exploration du graphe, il retourne les distances les plus courtes depuis le nœud source vers tous les autres nœuds.

2 Pseudo-Code :

```
Algorithme Dijkstra(Graphe, s)
    # Début
    p(s) ← 0

    # Initialisation
    Sconnu ← {s}
    Sinconnu ← Sommets(Graphe) - Sconnu

    # Pour chaque sommet u non encore exploré
    Pour chaque u ∈ Sinconnu faire:
        p(u) ← ∞
        précédent(u) ← None

    # Pour chaque sommet u adjacent à s
    Pour chaque u ∈ Successeurs(s) faire:
        p(u) ← poids(s, u)
        précédent(u) ← s

    # Tant qu'il y a des sommets non explorés
    Tant que ∃ u ∈ Sinconnu tel que p(u) ≠ ∞ faire:
        choisir u ∈ Sinconnu tel que p(u) soit minimal
        Pour chaque v ∈ Successeurs(u) ∩ Sinconnu faire:
            Si p(u) + poids(u, v) < p(v) alors:
                p(v) ← p(u) + poids(u, v)
                précédent(v) ← u
            FinSi
        FinPour
        Sconnu ← Sconnu ∪ {u}
        Sinconnu ← Sinconnu - {u}
    FinTantQue
Fin
```

Figure 3 : Pseudo-code Dijkstra

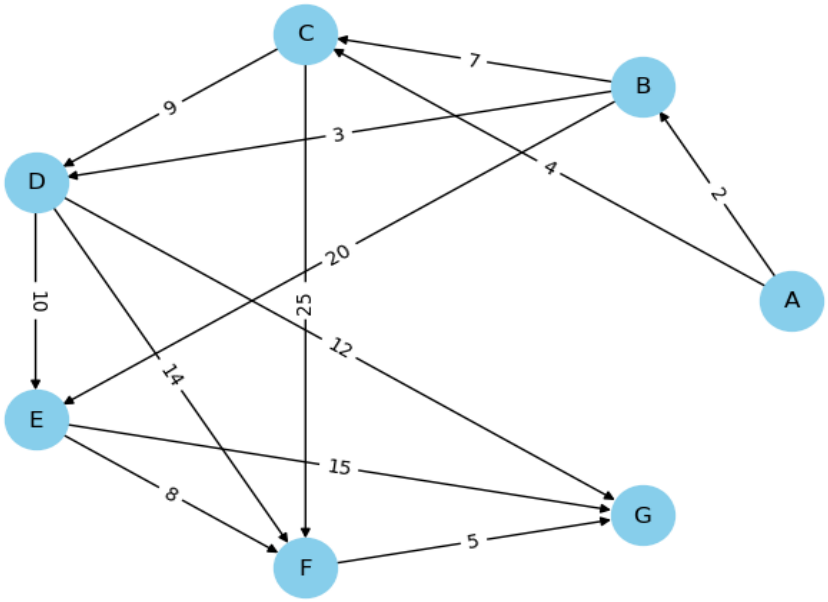
3 Fonction python de Dijkstra :

```python
import math
import networkx as nx

def dijkstra(graphe, depart):
    # Initialisation des distances
    distances = {sommet: math.inf for sommet in graphe.nodes()}
    distances[depart] = 0

    # Initialisation des chemins
    chemins_plus_courts = {sommet: [depart] for sommet in graphe.nodes()}

    # Initialisation de l'ensemble P et de l'ensemble Q
    P = {depart}
    Q = set(graphe.nodes()) - P

    while Q:
        # Étape 1 : Mettre à jour les distances des voisins de P
        for sommet in P:
            for voisin in graphe.neighbors(sommet):
                distance_voisin = distances[sommet] + graphe[sommet][voisin]['weight']
                if distance_voisin < distances[voisin]:
                    distances[voisin] = distance_voisin
                    chemins_plus_courts[voisin] = chemins_plus_courts[sommet] + [voisin]

        # Recherche du prochain nœud à ajouter à P
        min_distance = math.inf
        prochain_sommet = None
        for sommet in Q:
            if distances[sommet] < min_distance:
                min_distance = distances[sommet]
                prochain_sommet = sommet

        # Ajouter le prochain nœud à P et le retirer de Q
        P.add(prochain_sommet)
        Q.remove(prochain_sommet)

    return distances, chemins_plus_courts
```

Figure 4 : Fonction python de Dijkstra

4 Test de Dijkstra :

```python
# Création du graphe pondéré orienté
G = nx.DiGraph()

# Ajout des arêtes avec les poids spécifiés
G.add_weighted_edges_from([
    ('A', 'B', 2), ('A', 'C', 4),
    ('B', 'C', 7), ('B', 'D', 3), ('B', 'E', 20),
    ('C', 'D', 9), ('C', 'F', 25),
    ('D', 'E', 10), ('D', 'F', 14), ('D', 'G', 12),
    ('E', 'F', 8), ('E', 'G', 15),
    ('F', 'G', 5)
])

# Dessin du graphe avec les poids des arêtes
pos = nx.circular_layout(G)  # Positionnement des nœuds
labels = nx.get_edge_attributes(G, 'weight')  # Récupération des poids des arêtes
nx.draw(G, pos, with_labels=True, node_size=1000, node_color="skyblue")  # Dessin des nœuds
nx.draw_networkx_edge_labels(G, pos, edge_labels=labels)  # Dessin des poids des arêtes
plt.show()
```




```python
# Sommet de départ
depart = 'A'

# Utilisation de l'algorithme de Dijkstra pour trouver les distances les plus courtes et les chemins les plus courts
distances, chemins_plus_courts = dijkstra(G, depart)

# Affichage des résultats
print("Distances les plus courtes et chemins les plus courts:")
for sommet in distances:
    print(f"Chemin vers {sommet} : {' -> '.join(chemins_plus_courts[sommet])}, distance = {distances[sommet]}")
```

```
Distances les plus courtes et chemins les plus courts:
Chemin vers A : A, distance = 0
Chemin vers B : A -> B, distance = 2
Chemin vers C : A -> C, distance = 4
Chemin vers D : A -> B -> D, distance = 5
Chemin vers E : A -> B -> D -> E, distance = 15
Chemin vers F : A -> B -> D -> F, distance = 19
Chemin vers G : A -> B -> D -> G, distance = 17
```

Figure 5 : Test python de Dijkstra

Chapitre 3

Algorithme de Bellman-Ford

L'algorithme de Bellman-Ford, nommé d'après les mathématiciens Richard Bellman et Lester Ford, est un algorithme classique utilisé dans le domaine de la théorie des graphes pour résoudre le problème du plus court chemin. Contrairement à l'algorithme de Dijkstra, l'algorithme de Bellman-Ford peut gérer les graphes pondérés comportant des arêtes de poids négatif, ce qui le rend plus polyvalent dans divers scénarios.

1 Différence entre Bellman-Ford et Dijkstra:

L'algorithme de Bellman Ford et l'algorithme de Dijkstra sont tous deux des algorithmes de chemin le plus court à source unique, c'est-àdire qu'ils déterminent tous les deux la distance la plus courte entre chaque sommet d'un graphe et un sommet source unique. Cependant, l'algorithme de Bellman-Ford autorise la présence de certains arcs de poids négatif et permet de détecter l'existence d'un circuit absorbant. Alors que l'algorithme de Dijkstra peut fonctionner ou non lorsqu'il y a un bord de poids négatif. Mais ne fonctionnera certainement pas en cas de cycle de poids négatif. En termes de complexité, l'algorithme de Bellman prend plus de temps que celui de Dijkstra. Sa complexité en temps est O(VE), mais pour celui de Dijkstra c'est O (E log V).

2 Fonctionnement de l'Algorithme de Bellman-Ford :

L'algorithme de Bellman-Ford fonctionne selon une approche itérative pour trouver le chemin le plus court entre un nœud source et tous les autres nœuds dans un graphe pondéré, même en présence d'arêtes de poids négatif. Voici comment cet algorithme fonctionne en détail :

1) Initialisation :

-L'algorithme commence par initialiser la distance du nœud initial (I) à zéro et la distance de tous les autres nœuds à l'infini, sauf le nœud initial lui-même.

2) Boucle Principale :

-L'algorithme effectue une boucle principale de |S|-1 itérations, où |S| est le nombre de sommets dans le graphe.

-Pour chaque itération de la boucle principale, l'algorithme explore chaque arc (u, v) dans le graphe et met à jour les distances des nœuds en vérifiant si la distance actuelle de v est plus grande que la somme de la distance de u et le poids de l'arc (u, v). Si c'est le cas, la distance de v est mise à jour avec cette nouvelle valeur.

3) Détection de Circuit Absorbant :

-Après avoir effectué |S|-1 itérations, l'algorithme effectue une autre boucle pour vérifier s'il existe un circuit absorbant dans le graphe. Pour chaque arc (u, v), il vérifie à nouveau si la distance de v peut être améliorée en passant par u. Si c'est le cas, cela signifie qu'il y a un circuit absorbant dans le graphe.

4) Résultats :

-Si aucune amélioration n'est possible après |S|-1 itérations, l'algorithme termine avec succès et retourne les distances les plus courtes de chaque nœud source à tous les autres nœuds.

-Si un circuit absorbant est détecté, cela indique qu'il existe une boucle de poids négatif dans le graphe, ce qui signifie qu'il n'y a pas de solution aux chemins les plus courts.

3 Pseudo-Code :

```
Algorithme BellmanFord(Graphe, I)

    # Initialisation
    d[I] ← 0
    Pour chaque v ∈ S sauf I faire
        d[v] ← ∞
    FinPour

    # Itérations
    Pour i de 1 jusqu'à |S| - 1 faire
        Pour chaque arc (u, v) ∈ A faire
            Si d[v] > d[u] + C(u, v) alors
                d[v] ← d[u] + C(u, v)
            FinSi
        FinPour
    FinPour

    # Détection de Circuit Absorbant
    Pour chaque arc (u, v) ∈ A faire
        Si d[v] > d[u] + C(u, v) alors
            Afficher "Existence d'un circuit absorbant"
        FinSi
    FinPour
FinAlgorithme
```

Figure 6 : Pseudo-code Bellman-Ford

4 Fonction python de Bellman-Ford :

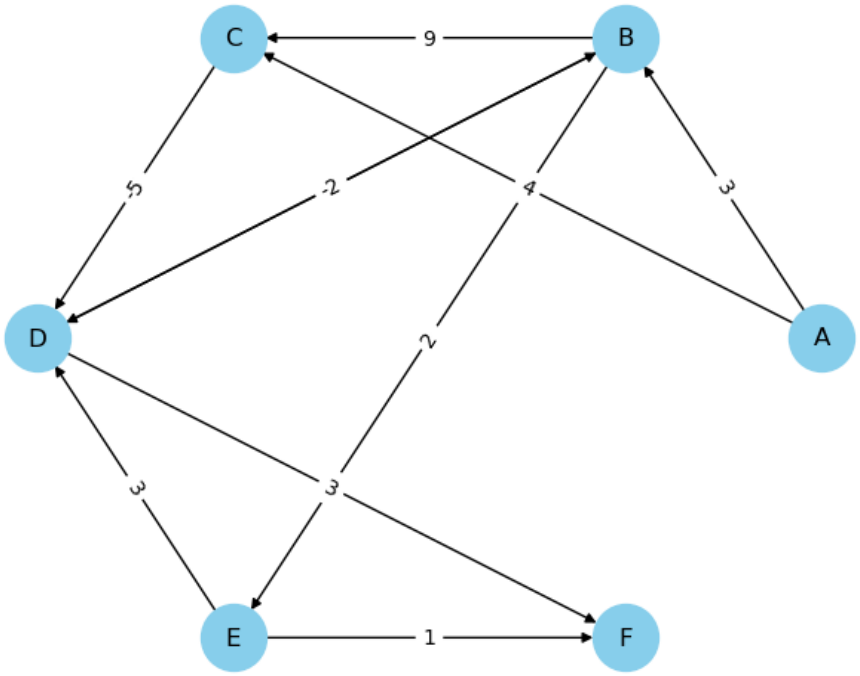
```python
# INF représente l'infini pour initialiser les distances
INF = float('inf')

def bellman_ford(nx_graphe, debut):
    # Initialisation des distances à l'infini sauf pour le sommet de départ
    distances = {sommet: INF for sommet in nx_graphe.nodes}
    distances[debut] = 0

    # Initialisation des prédécesseurs à None pour chaque sommet
    precedent = {sommet: None for sommet in nx_graphe.nodes}

    # Boucle principale pour itérer sur tous les sommets sauf le dernier
    for i in range(len(nx_graphe.nodes) - 1):
        # Parcours de toutes les arêtes avec leurs poids dans le graphe
        for u, v, poids in nx_graphe.edges(data='weight', default=1):
            # Mise à jour des distances si un chemin plus court est trouvé
            if distances[u] + poids < distances[v]:
                distances[v] = distances[u] + poids
                precedent[v] = u

    # Vérification de l'existence d'un circuit de poids négatif
    for u, v, poids in nx_graphe.edges(data='weight', default=1):
        if distances[u] + poids < distances[v]:
            print("Le graphe contient un cycle de poids négatif")
            exit()

    # Calcul des chemins les plus courts et de leurs distances pour chaque destination
    resultats = {}
    for destination in nx_graphe.nodes:
        if destination != debut:
            chemin = []
            sommet_courant = destination
            # Reconstruction du chemin à partir des prédécesseurs
            while sommet_courant != debut:
                chemin.insert(0, sommet_courant)
                sommet_courant = precedent[sommet_courant]
            chemin.insert(0, debut)
            resultats[destination] = {'chemin': chemin, 'distance': distances[destination]}

    return resultats
```

Figure 7 : Fonction python de Bellman-Ford

5 Test de Bellman-Ford :

-On va tester avec l'exemple faite dans le cours.

```python
# Création du graphe
G = nx.DiGraph()
G.add_weighted_edges_from([
    ('A', 'B', 3), ('A', 'C', 4),
    ('B', 'C', 9), ('B', 'D', 2), ('B', 'E', 2),
    ('C', 'D', -5), ('D', 'B', -2),
    ('D', 'F', 3), ('E', 'D', 3), ('E', 'F', 1)
])
# Dessin du graphe avec les poids des arêtes
pos = nx.circular_layout(G)  # Positionnement des nœuds
labels = nx.get_edge_attributes(G, 'weight')  # Récupération des poids des arêtes
nx.draw(G, pos, with_labels=True, node_size=1000, node_color="skyblue")  # Dessin des nœuds
nx.draw_networkx_edge_labels(G, pos, edge_labels=labels)  # Dessin des poids des arêtes
plt.show()
```




```python
# Sommet de départ
debut = 'A'

# Calcul des plus courts chemins
resultats = bellman_ford(G, debut)

# Affichage des résultats
print("Distances les plus courtes et chemins les plus courts:")
for destination, resultat in resultats.items():
    chemin_str = ' -> '.join(resultat['chemin'])
    print(f"Chemin vers {destination}: {chemin_str}, distance = {resultat['distance']}")
```

```
Distances les plus courtes et chemins les plus courts:
Chemin vers B: A -> C -> D -> B, distance = -3
Chemin vers C: A -> C, distance = 4
Chemin vers D: A -> C -> D, distance = -1
Chemin vers E: A -> C -> D -> B -> E, distance = -1
Chemin vers F: A -> C -> D -> B -> E -> F, distance = 0
```

Figure 8 : Test python de Bellman-Ford

Chapitre 4

Problème réel de plus court chemin

Système de navigation GPS

1 Description du problème :

Un service de navigation GPS doit être développé pour les voyageurs au Maroc, permettant de trouver le chemin le plus court entre deux villes. Le réseau routier marocain est représenté sous forme d'un graphe pondéré, où chaque nœud représente une ville et chaque arête représente une route entre deux villes, avec un poids représentant la distance entre ces villes en kilomètres.

Le service doit permettre à un utilisateur de spécifier une ville de départ et une ville de destination, puis de calculer le chemin le plus court entre ces deux villes en utilisant l'un des deux algorithmes : Dijkstra ou Bellman-Ford. Le chemin le plus court trouvé doit être affiché, ainsi que la distance totale parcourue.

2 Exemple de graphe pondéré pour le réseau routier marocain :

```python
G = nx.DiGraph()
G.add_weighted_edges_from([
    ('Rabat', 'Casablanca', 90),
    ('Casablanca', 'Mohammedia', 30),
    ('Casablanca', 'Tanger', 320),
    ('Mohammedia', 'Tanger', 270),
    ('Tanger', 'Fes', 190),
    ('Fes', 'Meknes', 60),
    ('Meknes', 'Rabat', 160),
    ('Casablanca', 'El Jadida', 110),
    ('El Jadida', 'Safi', 60),
    ('Safi', 'Essaouira', 150)
])
# Dessin du graphe avec les poids des arêtes
pos = nx.circular_layout(G)  # Positionnement des nœuds
labels = nx.get_edge_attributes(G, 'weight')  # Récupération des poids des arêtes
nx.draw(G, pos, with_labels=True, node_size=1000, node_color="skyblue")  # Dessin des nœuds
nx.draw_networkx_edge_labels(G, pos, edge_labels=labels)  # Dessin des poids des arêtes
plt.show()
```

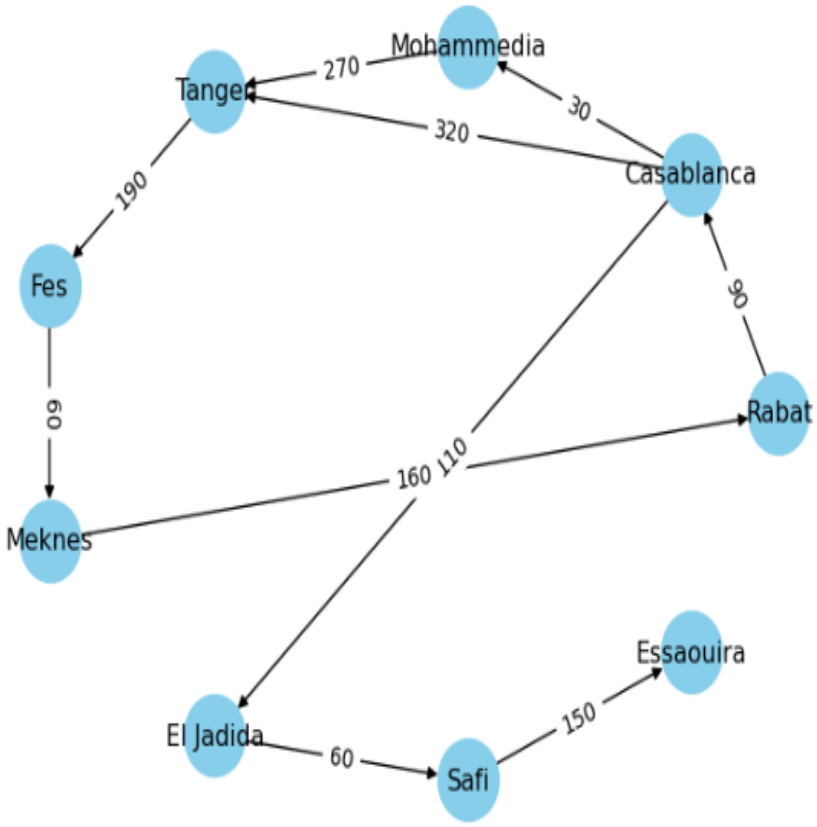


Figure 9 : Graphe de réseau routier marocain

3 Résolution du problème de plus court chemin :

- Voici l'exécution des algorithmes de Dijkstra et de Bellman-Ford pour trouver le chemin le plus court de Mohammedia vers les autres villes dans le graphe donné :

Résolution avce Dijkstra:

```
# Sommet de départ
depart = 'Mohammedia'

# Utilisation de l'algorithme de Dijkstra pour trouver les distances les plus courtes et les chemins les plus courts
distances, chemins_plus_courts = dijkstra(G, depart)

# Affichage des résultats
print("Distances les plus courtes et chemins les plus courts:")
for sommet in distances:
    print(f"Chemin vers {sommet} : {' -> '.join(chemins_plus_courts[sommet])}, distance = {distances[sommet]}")
```

```
Distances les plus courtes et chemins les plus courts:
Chemin vers Rabat : Mohammedia -> Tanger -> Fes -> Meknes -> Rabat, distance = 680
Chemin vers Casablanca : Mohammedia -> Tanger -> Fes -> Meknes -> Rabat -> Casablanca, distance = 770
Chemin vers Mohammedia : Mohammedia, distance = 0
Chemin vers Tanger : Mohammedia -> Tanger, distance = 270
Chemin vers Fes : Mohammedia -> Tanger -> Fes, distance = 460
Chemin vers Meknes : Mohammedia -> Tanger -> Fes -> Meknes, distance = 520
Chemin vers El Jadida : Mohammedia -> Tanger -> Fes -> Meknes -> Rabat -> Casablanca -> El Jadida, distance = 880
Chemin vers Safi : Mohammedia -> Tanger -> Fes -> Meknes -> Rabat -> Casablanca -> El Jadida -> Safi, distance = 940
Chemin vers Essaouira : Mohammedia -> Tanger -> Fes -> Meknes -> Rabat -> Casablanca -> El Jadida -> Safi -> Essaouira, distanc
e = 1090
```

Résolution avce Bellman-Ford:

```
# Sommet de départ
debut = 'Mohammedia'

# Calcul des plus courts chemins
resultats = bellman_ford(G, debut)

# Affichage des résultats
print("Distances les plus courtes et chemins les plus courts:")
for destination, resultat in resultats.items():
    chemin_str = ' -> '.join(resultat['chemin'])
    print(f"Chemin vers {destination}: {chemin_str}, distance = {resultat['distance']}")
```

```
Distances les plus courtes et chemins les plus courts:
Chemin vers Rabat: Mohammedia -> Tanger -> Fes -> Meknes -> Rabat, distance = 680
Chemin vers Casablanca: Mohammedia -> Tanger -> Fes -> Meknes -> Rabat -> Casablanca, distance = 770
Chemin vers Tanger: Mohammedia -> Tanger, distance = 270
Chemin vers Fes: Mohammedia -> Tanger -> Fes, distance = 460
Chemin vers Meknes: Mohammedia -> Tanger -> Fes -> Meknes, distance = 520
Chemin vers El Jadida: Mohammedia -> Tanger -> Fes -> Meknes -> Rabat -> Casablanca -> El Jadida, distance = 880
Chemin vers Safi: Mohammedia -> Tanger -> Fes -> Meknes -> Rabat -> Casablanca -> El Jadida -> Safi, distance = 940
Chemin vers Essaouira: Mohammedia -> Tanger -> Fes -> Meknes -> Rabat -> Casablanca -> El Jadida -> Safi -> Essaouira, distance
= 1090
```

Figure 10 : La résolution de problème

Conclusion

L'algorithme de **Bellman-Ford**, développé par Richard Bellman, a révolutionné la façon dont nous abordons le problème du plus court chemin dans les graphes pondérés. Son utilisation de la programmation dynamique offre une approche élégante pour traiter les cycles absorbants, rendant l'algorithme adaptable à une variété de situations. Cependant, sa complexité temporelle plus élevée le rend moins efficace que l'algorithme de **Dijkstra** dans de nombreux cas, en particulier pour les graphes de grande taille. Il est donc crucial de choisir judicieusement l'algorithme en fonction des caractéristiques spécifiques du graphe et des contraintes de performance de l'application. En définitive, bien que l'algorithme de **Bellman-Ford** soit moins efficace dans certains scénarios, sa polyvalence et sa capacité à traiter les cycles absorbants en font un outil précieux dans la boîte à outils des algorithmes de plus court chemin.