Ministère de l'Enseignement Supérieur et de la Recherche Scientifique
Université des Sciences et de la Technologie Houari Boumediene

Faculté d'Electronique de l'Informatique
Département Informatique

Filière : Informatique

Spécialité : Big Data Analytics

Thème :

Problèmes de cheminement et applications

Travail présenté à M **BOUTICHE**

Module : Graphes et Big Data

Travail présenté par :
AISSANI Anouar
FELLAGUE Assia
GOUMEZIANE Kenza
BERGHOUT Mohamed Wassim

Table des matières

Introduction générale

En théorie des graphes, le problème de plus court chemin est le problème algorithmique qui consiste à trouver un chemin d'un sommet à un autre de façon que la somme des poids des arcs de ce chemin soit minimale.

Il existe de nombreuses variantes de ce problème suivant que le graphe est fini, orienté ou non, que chaque arc ou arête possède ou non une valeur (un poids).

Un chemin le plus court entre deux nœuds donnés est un chemin qui minimise la somme des valeurs des arcs traversés. Pour calculer un plus court chemin, il existe de nombreux algorithmes, selon la nature des valeurs et des contraintes supplémentaires qui peuvent être imposées. Par exemple, si:

- Les poids sont tous positifs : Dans ce cas, l'algorithme de Dijkstra permet de résoudre le problème du plus court chemin d'un sommet donné aux autres sommets.
- Les poids sont quelconques : Dans ce cas, l'algorithme de Bellman permet de tester s'il existe un circuit absorbant et, en absence d'un tel circuit, de trouver un plus court chemin.

Dans ce projet, nous allons programmer les deux algorithmes cités en dessus pour les appliquer dans un problème d'ordonnancement pour trouver une solution qui permet de terminer un projet le plus rapidement possible. Et enfin nous allons coder un autre algorithme comme bonus qui est l’algorithme de Prim.

Chapitre 01

Graphes valués

Matrice de poids:

Une matrice de poids est une matrice qui représente la distance d'un sommet à tous les autres d'un graphe évalué orienté ou non-orienté.

Construction d'un graphe valué :

Nous avons construit le graphe évalué avec l'algorithme suivant en Python

```
# Lire le nombre de sommet du graphe
n = int(input("donnez le nombre de sommets : \n"))

# Introduction de la matrice de poids M en lisant à chaque fois un poid
print("vous allez introduire votre matrice de poids\n")
M = [[inf] * (n) for i in range(n)]
print ("Initialisation de la matrice de poids \n")
print("M=", M)
for i in range(n):
    for j in range(n):
        e = input(
            "\n donnez l'élément[" + str(i + 1) +
            "][" + str(j + 1) + "] de la matrice "
        )
        if e == "inf" or e == "INF":
            M[i][j] = inf
        else:
            M[i][j] = int(e)

for i in range(n):
    for j in range(n):
        M[i][i] = 0

# L'affichage de la matrice de poids
print ("\n votre matrice de poids est : ")
for i in range(n):
    print((i + 1), "=", M[i]
```

Chapitre 02

Problème du plus court chemin

Algorithme de Bellman

Définition de l'algorithme :

L'algorithme de Bellman-Ford (Richard Bellman et Lester Ford) est un algorithme de programmation dynamique qui permet de trouver des plus courts chemins, depuis un sommet source donné, dans un graphe orienté pondéré.
Contrairement à l'algorithme de Dijkstra, qui ne peut être utilisé que lorsque tous les arcs ont des poids positifs ou nuls, l'algorithme de Bellman-Ford autorise la présence de certains arcs de poids négatif et permet de détecter l'existence d'un circuit absorbant, c'est-à-dire de poids total négatif, accessible depuis le sommet source.

Principe de l'algorithme de Bellman :

Nous résumons ici la logique de fonctionnement de l'algorithme :

1. Initialiser les distances de la source à tous les sommets en tant qu'infini (inf) et la distance à la source elle-même avec 0.
2. Créer un tableau dist[] de taille |V| où |V| est le nombre de sommets dans le graphe
3. Initialiser dist[] avec des valeurs infinies sauf pour dist[src] qu'on initialise à 0 où src est le sommet source.
4. Faire |V|-1 itérations suivant les instructions suivantes :

    Répéter pour chaque arête {u,v}

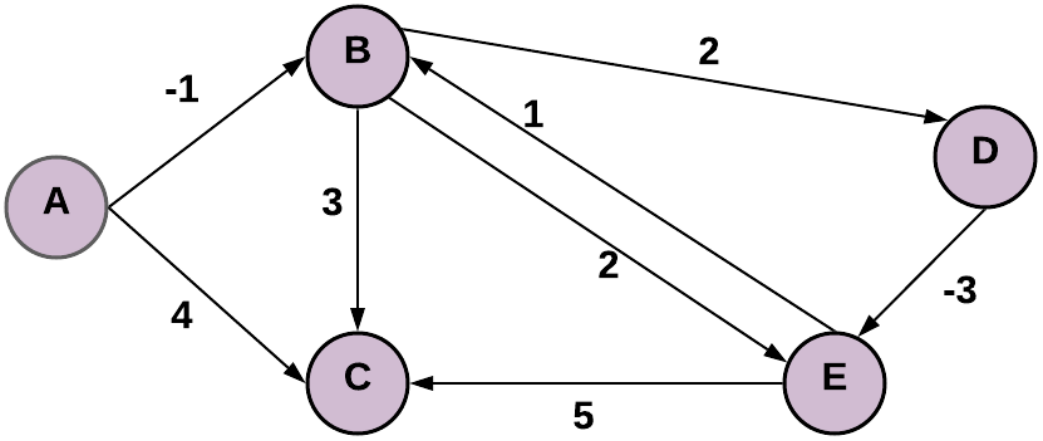
        Si dist[v]>dist[u]+poids l'arête {u,v}

            alors dist[v]=dist[u]+poid de l'arête {u,v}

5. Si dist[v] > dist [u] + poids de l'arête {u,v}, alors "le graphe contient un cycle de pondération négatif"

Exemple d'application de l'algorithme de Bellman :

Soit le graphe suivant :



(G)

Appliquons les étapes de l'algorithme de Bellman citées en haut sur le graphe (G) pour le sommet A.

1. Le Tableau des distances donne :

	A	B	C	D	E
Initialisation	0	∞	∞	∞	∞

2. Itération 1 :

➢ La distance entre A et B est égale à -1 et ∞ > (-1) + 0, alors on met à jour cette valeur.
➢ La distance entre A et C est égale à 4 et ∞ > 4 + 0, alors on met à jour cette valeur.
➢ Pour E et D, il n' y a pas d'arcs qui les relient à A et les autres valeurs sont toujours initialisées à l'infini alors on passe à l'itération suivante.

Le tableau des distances sera alors mis à jour comme suit :

	A	B	C	D	E
Itération 1	0	-1	4	∞	∞

3. Itération 2 :

➢ La distance entre B et C est égale à 3 et 4 > 3 + (-1), alors on met à jour la valeur 4 de C dans le tableau des distances.
➢ La distance entre B et C est égale à 2 et ∞ > 2 + (-1), alors on met à jour la valeur ∞ de C dans le tableau des distances.

➢ La distance entre B et E est égale à 2 et ∞ > 2 + (-1), alors on met à jour la valeur ∞ de E dans le tableau des distances.

	A	B	C	D	E
Itération 2	0	-1	2	1	1

En suivant la même logique, les deux dernières itérations donnent les deux tableau suivant :

4. Itération 3:

	A	B	C	D	E
Itération 3	0	-1	2	-2	1

5. Itération 4:

	A	B	C	D	E
Itération 4	0	-1	2	-2	1

Interprétation du résultat :

Le plus court chemin pour aller de A vers B vaut -1.
Le plus court chemin pour aller de A vers C vaut 2.
Le plus court chemin pour aller de A vers D vaut 1.
Le plus court chemin pour aller de A vers E vaut 1.
Le plus court chemin pour aller de A vers lui-même vaut 0.

Exécution :

A l'exécution nous obtenons le même résultat :

```
$ py bellman.py
[0, inf, inf, inf, inf]
[[0, 1, -1], [0, 2, 4], [1, 2, 3], [1, 3, 2], [1, 4, 2], [3, 2, 5], [3, 1, 1], [4, 3, -3]]
Vertex Distance from Source
0               0
1               -1
2               2
3               -2
4               1
```

Algorithme de Dijkstra

Définition de l'algorithme :

Il peut être utilisé pour trouver la distance la plus courte possible d'un sommet source à tout autre sommet possible existant dans un graphe pondéré, à condition que le sommet soit accessible depuis le sommet source.

Principe de l'algorithme de Dijkstra:

Le fonctionnement de l'algorithme de Dijkstra est le suivant.

- **Étape 1:** Étant donné des sommets non visités, sélectionnez le sommet avec la plus petite distance de la source et visitez-le.
- **Étape 2:** La distance pour chaque voisin est alors mise à jour. La même chose est faite pour le sommet visité, qui a une distance actuelle supérieure à la somme et au poids de l'arête donnée entre eux selon les instructions suivantes:

Si dist[u] + poids l'arête {u,v}< dist[v]

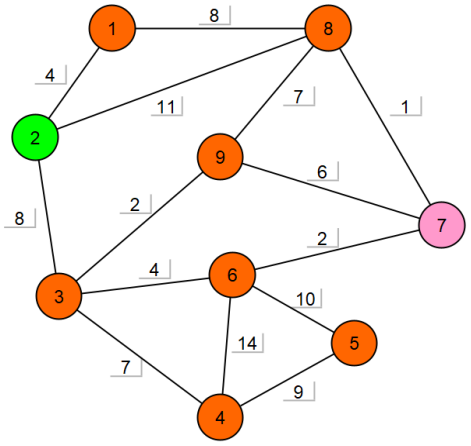
alors dist[v] = dist[u] + poids de l'arête {u,v}

- **Étape 3:** répéter les étapes 1 et 2 jusqu'à ce qu'il n'y ait plus de sommets non visités.

Exemple d'application de l'algorithme de Dijkstra:

Soit un graphe avec 9 sommets

On prend le sommet (2) et on calcule le plus court chemin qui mène à tous les sommets restants du graphe.

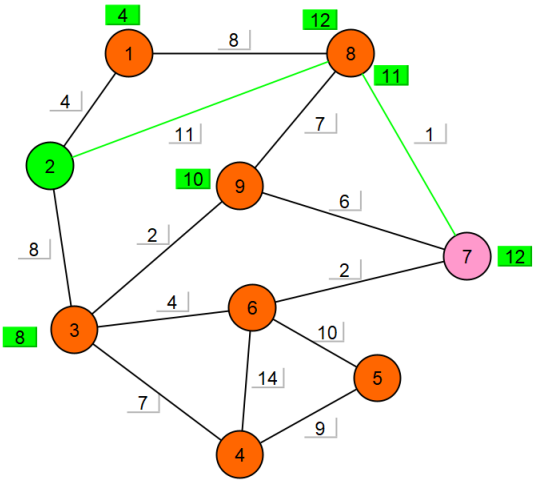


G(9)

On veut calculer le plus court chemin entre le sommet (2) et (7).

- On sélectionne les sommets voisins avec la plus petite distance du sommet (2).
- On met à jour les valeurs des sommets voisins comme suit:
  - (1) à 4.
  - (8) à 11.
  - (3) à 8.
  - (1)(8) à 12.
  - (1)(8)(7) à 13.

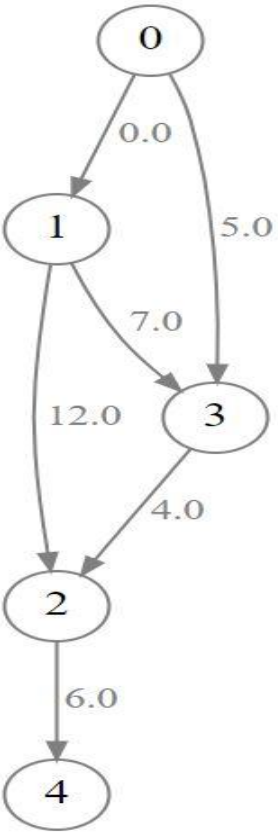
    …
  - (8)(7) à 12.
- On répète ce calcule sur tous les sommets qui ont un chemin sur le sommet (7) du graphe. On choisit la plus petite valeur qui est dans ce cas 12 avec le chemin (8)(7).



G(9)

Problème d'ordonnancement:

On parle d'un problème d'ordonnancement quand on a un ensemble de travaux à réaliser décomposable en tâches.
Le déroulement de ces tâches doit respecter une contrainte de succession stricte.
Le problème posé peut être schématisé sous forme d'un graphe tel que les sommets sont les tâches et un arc (i,j) a une valuation k ssi la tâche j commence au plus tôt k jours après le début de la tâche i.
Exemple:
Soit le graphe suivant :



L'objectif est de trouver les dates de début au plus tôt de chaque tâche, pour terminer le projet le plus rapidement possible. Il suffit alors de calculer les chemins les plus longs depuis le sommet 0 à tous les autres.
On sait que la recherche d'un plus long chemin dans graphe G=(E,V,P) (P est la matrice de poids) est équivalent à la recherche d'un plus court chemin dans le graphe G=(E,V,-P).
On appliquera donc un algorithme pour trouver la distance la plus courte possible d'un sommet source à tout autre sommet possible existant sur le graphe G=(E,V,-P).
On utilisera l'algorithme de bellman car il peut gérer plus de cas que les autres.

Chapitre 03

Algorithme de Prim

Algorithme de Prim

Définition :

L'algorithme de Prim est un algorithme qui calcule un arbre couvrant minimal dans un graphe connexe valué et non orienté.

Principe de l'algorithme de Prim :

1. Initialiser la matrice de poids.
2. Initialiser une matrice d''arbre couvrant vide.
3. Initialiser une liste de type booléen pour marquer les sommes utilisées à chaque fois.
4. Définir la fonction "`fullNodesList(nodesList)`" qui vérifie si la matrice de l'arbre couvrant de poids contient tous les sommets de la matrice de poids.
5. Définir la fonction "`addNode(M, nodeList, MAC)`" qui ajoute un nœud à l'arbre couvrant de poids.
6. Définir la fonction "`prim(M)`" qui retourne une matrice d''arbre couvrant vide à partir la matrice de poids M:
    - Une boucle qui ajoute un nœud minimal à l'arbre couvrant de poids à partir la matrice de poids.
    - Sortir de la boucle si tous les nœuds sont ajoutés.
    - retourner la matrice de l'arbre couvrant.

Exemple d'application de l'algorithme de Prim:

Déroulement:

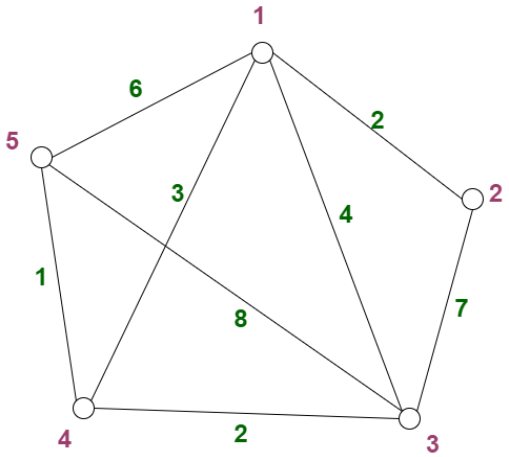


Figure 3.1: graphe connexe valué et non orienté

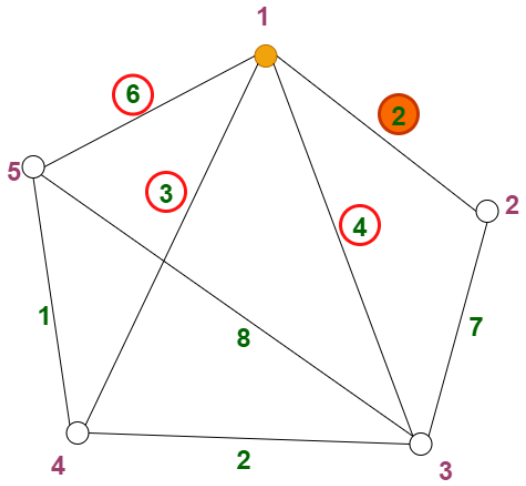


Figure 3.2: Marquage du premier sommet

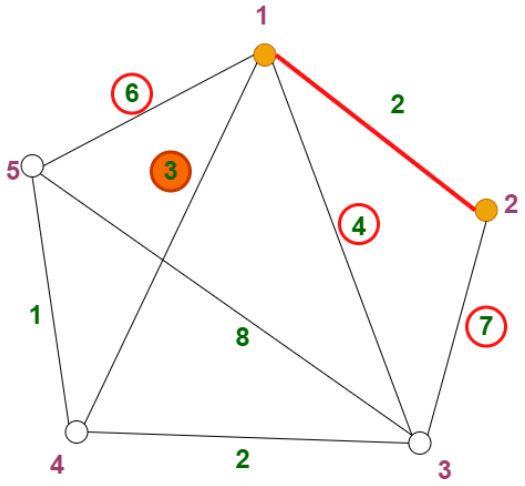


Figure 3.2:Marquage du deuxième sommet

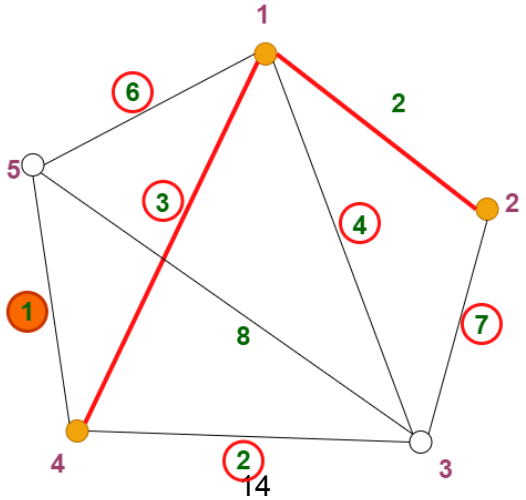


Figure 3.3: Marquage du troisième sommet

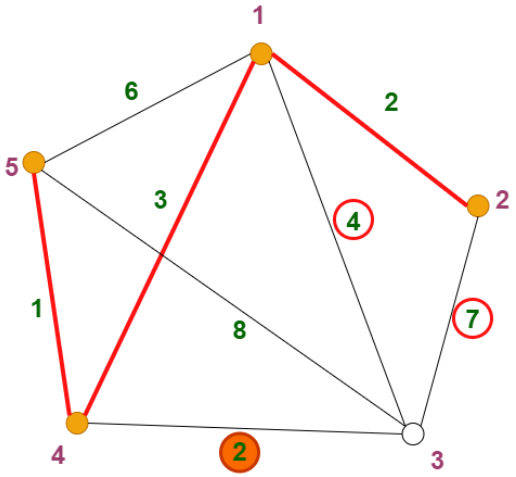


Figure 3.4: Marquage du quatrième sommet

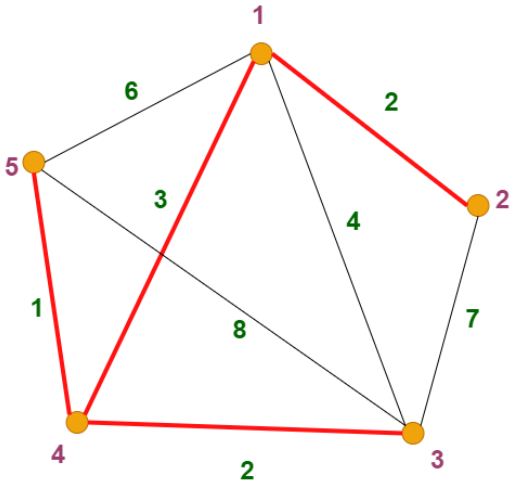


Figure 3.5: Marquage du cinquième sommet

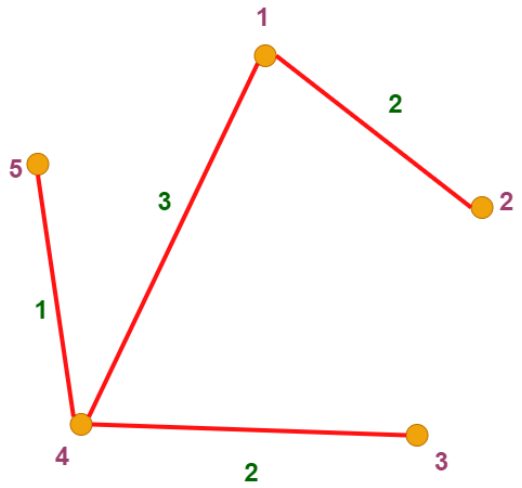


Figure 3.3: L' arbre couvrant minimal

Exécution:

```
votre matrice de poids est :
1 = [inf, 2, 4, 3, 6]
2 = [2, inf, 7, inf, inf]
3 = [4, 7, inf, 2, 8]
4 = [3, inf, 2, inf, 1]
5 = [6, inf, 8, 1, inf]

la matrice de l'arbre couvrant de poids
1 = [inf, 2, inf, 3, inf]
2 = [2, inf, inf, inf, inf]
3 = [inf, inf, inf, 2, inf]
4 = [3, inf, 2, inf, 1]
5 = [inf, inf, inf, 1, inf]
```

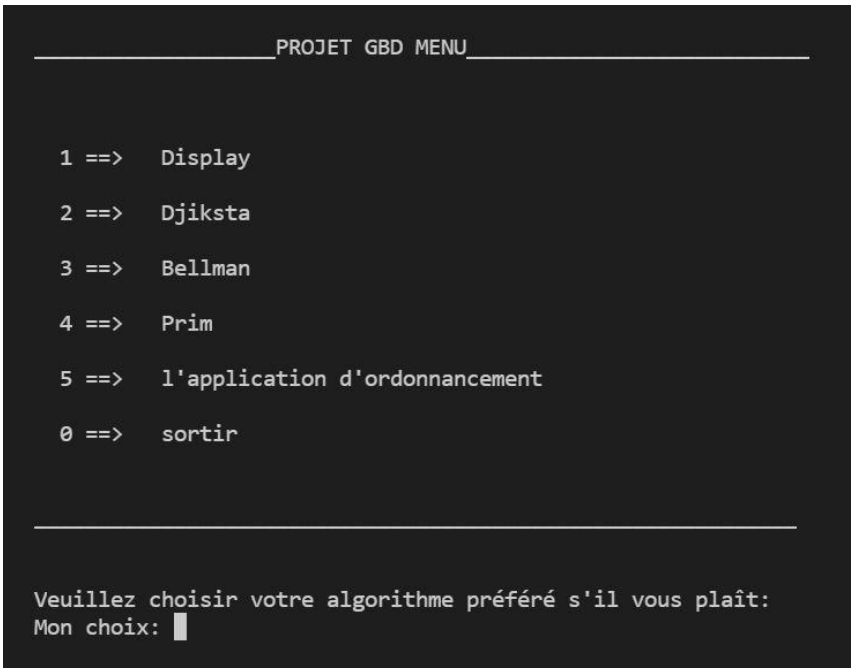
Chapitre 03

Réalisation

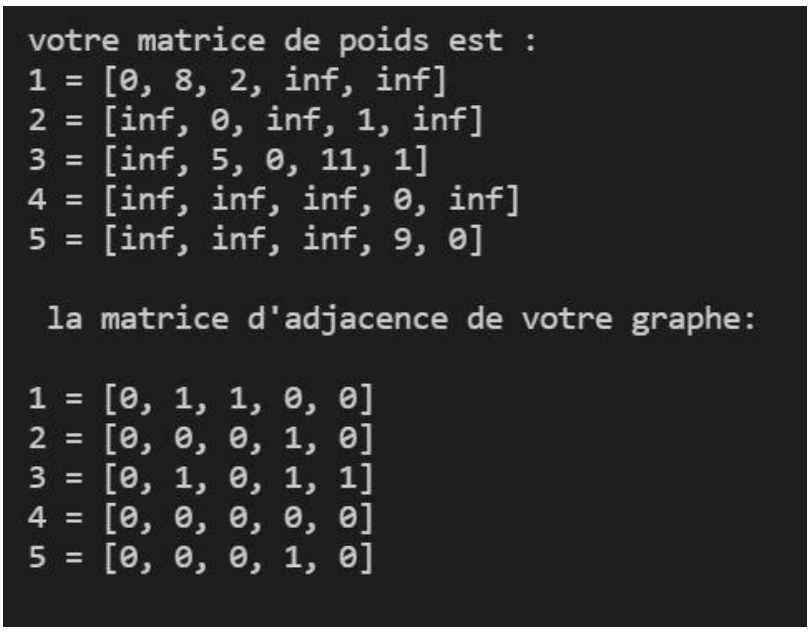
Réalisation du projet :

Voici des captures d'écrans sur les différentes parties du codes réalisé avec Python

1. **Menu du projet**



2. **Matrice d'adjacence de graphe non orienté**



3. Matrice d'adjacence de graphe orienté

```
votre matrice de poids est :
1 = [inf, 2, 4, 3, 6]
2 = [2, inf, 7, inf, inf]
3 = [4, 7, inf, 2, 8]
4 = [3, inf, 2, inf, 1]
5 = [6, inf, 8, 1, inf]

 la matrice d'adjacence de votre graphe:

1 = [0, 1, 1, 1, 1]
2 = [1, 0, 1, 0, 0]
3 = [1, 1, 0, 1, 1]
4 = [1, 0, 1, 0, 1]
5 = [1, 0, 1, 1, 0]
```

4. Algorithme de Djikstra

```
votre matrice de poids est :
1 = [0, 8, 2, inf, inf]
2 = [inf, 0, inf, 1, inf]
3 = [inf, 5, 0, 11, 1]
4 = [inf, inf, inf, 0, inf]
5 = [inf, inf, inf, 9, 0]

Distance depuis la source (Djiksta):

Sommet 1        ------------>    0
Sommet 2        ------------>    7
Sommet 3        ------------>    2
Sommet 4        ------------>    8
Sommet 5        ------------>    3
```

5. Algorithme de Bellman

```
votre matrice de poids est :
1 = [0, 8, 2, inf, inf]
2 = [inf, 0, inf, 1, inf]
3 = [inf, 5, 0, 11, 1]
4 = [inf, inf, inf, 0, inf]
5 = [inf, inf, inf, 9, 0]

Distance depuis la source (Bellman):

Sommet 1        ------------>    0
Sommet 2        ------------>    7
Sommet 3        ------------>    2
Sommet 4        ------------>    8
Sommet 5        ------------>    3
```

6. Algorithme de Prim

```
Donnez le poids entre les sommets 4 et 5: 1

votre matrice de poids est :
1 = [inf, 2, 4, 3, 6]
2 = [2, inf, 7, inf, inf]
3 = [4, 7, inf, 2, 8]
4 = [3, inf, 2, inf, 1]
5 = [6, inf, 8, 1, inf]

La matrice de l'arbre couvrant de poids:

1 = [inf, 2, inf, 3, inf]
2 = [2, inf, inf, inf, inf]
3 = [inf, inf, inf, 2, inf]
4 = [3, inf, 2, inf, 1]
5 = [inf, inf, inf, 1, inf]
```

7. solution du Problème d'ordonnancement

```
votre matrice de poids est :
1 = [0, 0, inf, 5, inf]
2 = [inf, 0, 12, 7, inf]
3 = [inf, inf, 0, inf, 6]
4 = [inf, inf, 4, 0, inf]
5 = [inf, inf, inf, inf, 0]
Distance depuis la source
Sommet 1        ------------>    0
Sommet 2        ------------>    0
Sommet 3        ------------>    12
Sommet 4        ------------>    7
Sommet 5        ------------>    18
```

8. Quitter l'application

```
Veuillez choisir votre algorithme préféré s'il vous plaît:
Mon choix: 0


~~~~ ~~~~ ~~~~ Merci (✿^‿^) ~~~~ ~~~~ ~~~~

C:\Users\Lyon\Downloads\STUDY\M1_BIGDATA\GBD\Code>
```

Conclusion générale

L’objectif de ce travail était de pouvoir résoudre un problème d’ordonnancement avec quelques algorithmes qui consistent à trouver le plus court chemin d’un point A à un point B à l’aide de la théorie des graphes.

On a résolu ce problème en utilisant l’algorithme de Bellman, où nous avons pu le modifier pour trouver le chemin le plus long du premier sommet à tous les autres et donc trouver les dates de début au plus tôt de chaque tâche, pour terminer le projet le plus rapidement possible.