S2.02
Boudot Nathan Couturier Quentin
Ferreira Iannis

Rapport SAE 2.02

Partie 1 : Connaissance des deux algorithmes
(Djikstra et Bellman Ford)

Partie 2 : Représentation graphique d'un graphe à partir de sa matrice.

Partie 3 : Génération aléatoire de matrices de graphes pondérés

Partie 4 : Codage de l'algorithme de Dijkstra

Partie 5 : Codage de l'algorithme de Bellman-Ford

Partie 6 : Influence du choix de la liste des fl`eches pour l'algorithme de Bellman-Ford

Partie 7 : Comparaison expérimentale des complexités

Partie 1 : Connaissance des deux algorithmes (Dijkstra et Bellman Ford)

1. **Liste de flèche ordonnée : a -> b, a -> c, a -> d, c -> b, d -> b, d -> c.**
2. **Principe de "BellMan Ford" : L'algorithme a pour objectif de trouver le plus court chemin d'un point donné dans un graphe orienté et de ressortir les prédécesseurs. L'algorithme permet de détecter une erreur en cas de boucle de poids négatif dans le graphe. Le principe est de prendre le point de départ et de calculer la flèche de poids la plus basse, après avoir trouvé la flèche la plus courte on refait le procédé avec le nouveau point jusqu'à trouver le chemin le plus court.**

Somme t	a	b	c	d
initialisation	0, ⌀	(+∞, ⌀)	(+∞, ⌀)	(+∞, ⌀)
Tour 1	0, ⌀	(7,d)	(3,d)	(2,a)
Tour 2	0, ⌀	(6,c)	(3,d)	(2,a)
Tour 3	0, ⌀	(5,c)	(3,d)	(2,a)
Tour 4	0, ⌀	(5,c)	(3,d)	(2,a)

Tour 1 :

Flèches (s,t)	dist(s)+poids(s,t) ? dist(t)	Modification des variables en t
ab	0+8 < +∞	Dist(b)=8 pred(b)=a)
ac	0+6 < +∞	Dist(c)=6 pred(c)=a
ad	0+2 < +∞	Dist(d)=2 pred(d)=a
cb	6+3 > 8	/ /
db	2+5 < 8	Dist(b)=7 pred(b)=d
dc	2+1 < 6	Dist(c)=3 pred(c)=d

Tour 2 :

Flèches (s,t)	dist(s)+poids(s,t) ? dist(t)	Modification des variables en t
ab	0+8 > 7	
ac	0+6 > 3	
ad	0+2 = 2	
cb	3+3 < 7	Dist(b)=6 pred(b)=c
db	2+5 > 6	
dc	2 +1=3	

Tour 3 :

Flèches (s,t)	dist(s)+poids(s,t) ? dist(t)	Modification des variables en t
ab	0 + 8 > 6	
ac	0 + 6 > 3	
ad	0 + 2 = 2	
cb	2 + 3 < 6	Dist(b)=5 pred(b)=c
db	2 + 5 > 5	
dc	2 + 1 = 3	

Tour 4 :

Flèches (s,t)	dist(s)+poids(s,t) ? dist(t)	Modification des variables en t
ab	0 + 8 > 5	
ac	0 + 6 > 2	
ad	0 + 2 = 2	
cb	3 + 3 > 5	
db	2 + 5 > 5	
dc	2 + 1 = 3	

maintenant nous ajoutons la flèche (c,a) de poids -4 :

Flèches (s,t)	dist(s)+poids(s,t) ? dist(t)	Modification des variables en t
ab	0 + 8 < +∞	Dist(b)=8 pred(b)=a
ac	0+6 < +∞	Dist(c)=6 pred(c)=a
ad	0+2 <+∞	Dist(d)=2 pred(d)=a
ca	6 - 4 > +∞	/ /
cb	6 + 3 > 8	/ /
db	2 + 5 < 8	Dist(b)=7 pred(b)=d
dc	2 + 1 < 6	Dist(c)=3 pred(c)=d

Le résultat est le même cela prouve qu'il n'y a pas de cycle de poids p négatif

3. **Principe Dijkstra : L'algorithme a pour objectif de trouver le plus court chemin entre 1 sommet de départ et les autres sommets d'un graphe. A chaque sommet correspond un couple « antécédant », « distance » en partant du sommet de départ nous mettons à jour les couples de c'est successeur puis les successeurs de ces successeurs et ainsi de suite. Une fois tous les sommets atteints et mis à jour il est possible de connaître le plus court chemin entre n'importe quel sommet et le sommet de départ. Un couple est mis à jour si est seulement si la distance du sommet (s) plus le poids du chemin (s,t) est inférieur à la distance du sommet (t).**

3. On peut utiliser l'algorithme de Dijkstra sur cet exemple car il ne possède aucun poids négatif.

	a	b	c	d
initialisation	(a,0)	(∅,+∞)	(∅,+∞)	(∅,+∞)
Tour 1	/	(a , 8)	(a , 6)	(a , 2)

- Dans le cadre de Dijkstra on ne peut pas utiliser de poids négatif.

Partie 2 : Représentation graphique d'un graphe à partir de sa matrice.

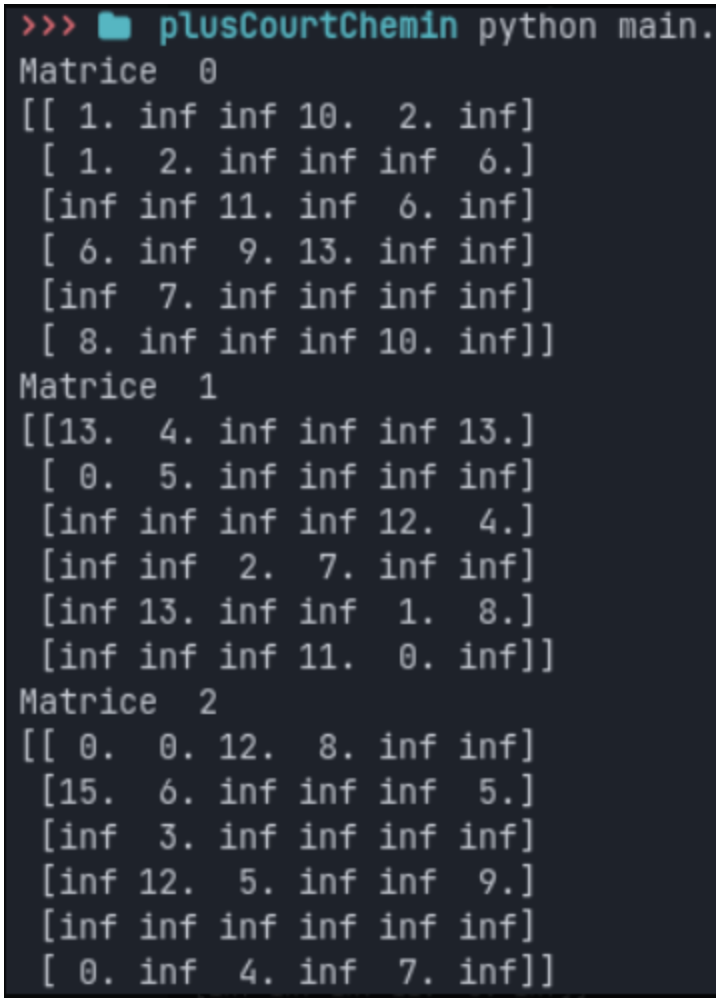
Exemple :

On peut représenter un graphique à l'aide de la matrice d'incidence en Java, on pourrait aussi tenter en python d'utiliser la bibliothèque plotly pour représenter graphiquement un graphe à l'aide de la matrice.

⚠

Partie 3 : Génération aléatoire de matrices de graphes pondérés

Exemple



Partie 4 : Codage de l'algorithme de Dijkstra

Test :

```
>>> plusCourtChemin python main.py
Matrice aléatoire Générée :
[[ 8. inf inf inf inf  1.]
 [ 4. inf inf  3. inf 11.]
 [inf  9. inf  5. inf inf]
 [13. inf 11.  5. inf  8.]
 [ 5. inf inf inf  5. 13.]
 [inf  0.  1.  9.  3. inf]]
Résultat avec l'algorithme de BellmanFord :
Le chemin d'accés à 'a' est ['a'], sa taille totale est: 0.0
Le chemin d'accés à 'b' est ['a', 'f', 'b'], sa taille totale est: 1.0
Le chemin d'accés à 'c' est ['a', 'f', 'c'], sa taille totale est: 2.0
Le chemin d'accés à 'd' est ['a', 'f', 'b', 'd'], sa taille totale est: 4.0
Le chemin d'accés à 'e' est ['a', 'f', 'e'], sa taille totale est: 4.0
Le chemin d'accés à 'f' est ['a', 'f'], sa taille totale est: 1.0
```

```
>>> plusCourtChemin python main.py
Matrice aléatoire Générée :
[[ 4. 11. inf 12. inf  2.]
 [14. inf inf inf  2. inf]
 [ 9.  0. inf  1.  8. inf]
 [15.  4. inf inf inf inf]
 [inf inf inf inf  3.  4.]
 [11. inf 15.  5.  4. inf]]
Résultat avec l'algorithme de BellmanFord :
Le chemin d'accés à 'a' est ['a'], sa taille totale est: 0.0
Le chemin d'accés à 'b' est ['a', 'b'], sa taille totale est: 11.0
Le chemin d'accés à 'c' est ['a', 'f', 'c'], sa taille totale est: 17.0
Le chemin d'accés à 'd' est ['a', 'f', 'd'], sa taille totale est: 7.0
Le chemin d'accés à 'e' est ['a', 'f', 'e'], sa taille totale est: 6.0
Le chemin d'accés à 'f' est ['a', 'f'], sa taille totale est: 2.0
```

```
Djikstra:
[0, 3, 0, 0, 1, 2]
Le chemin d'accés à 'a' est ['a'], sa taille totale est: 0
Le chemin d'accés à 'b' est ['a', 'd', 'b'], sa taille totale est: 9.0
Le chemin d'accés à 'c' est ['a', 'c'], sa taille totale est: 8.0
Le chemin d'accés à 'd' est ['a', 'd'], sa taille totale est: 8.0
Le chemin d'accés à 'e' est ['a', 'd', 'b', 'e'], sa taille totale est: 9.0
Le chemin d'accés à 'f' est ['a', 'c', 'f'], sa taille totale est: 8.0
```

Partie 5 : Codage de l'algorithme de Dijkstra

Test :

```
>>> plusCourtChemin python main.py
Matrice aléatoire Générée :
[[10.  4.  8. inf inf  0.]
 [inf  4. inf 10.  0. inf]
 [inf  9. inf  2.  4. 12.]
 [ 0. 15. inf inf inf  5.]
 [inf inf 12. inf  5. inf]
 [inf  3. inf inf 12. inf]]
Résultat avec l'algorithme de Djikstra :
Le chemin d'accés à 'a' est ['a'], sa taille totale est: 0
Le chemin d'accés à 'b' est ['a', 'f', 'b'], sa taille totale est: 3.0
Le chemin d'accés à 'c' est ['a', 'c'], sa taille totale est: 8.0
Le chemin d'accés à 'd' est ['a', 'c', 'd'], sa taille totale est: 10.0
Le chemin d'accés à 'e' est ['a', 'f', 'b', 'e'], sa taille totale est: 3.0
Le chemin d'accés à 'f' est ['a', 'f'], sa taille totale est: 0.0
```

```
>>> plusCourtChemin python main.py
Matrice aléatoire Générée :
[[ 6.  8. 15. inf inf 12.]
 [inf inf  5.  2.  8.  5.]
 [11.  3. inf 11.  1.  8.]
 [inf inf  0. inf  8. 12.]
 [11. inf inf inf inf inf]
 [inf inf  8. inf inf inf]]
Résultat avec l'algorithme de Djikstra :
Le chemin d'accés à 'a' est ['a'], sa taille totale est: 0
Le chemin d'accés à 'b' est ['a', 'b'], sa taille totale est: 8.0
Le chemin d'accés à 'c' est ['a', 'b', 'd', 'c'], sa taille totale est: 10.0
Le chemin d'accés à 'd' est ['a', 'b', 'd'], sa taille totale est: 10.0
Le chemin d'accés à 'e' est ['a', 'b', 'd', 'c', 'e'], sa taille totale est: 11.0
Le chemin d'accés à 'f' est ['a', 'f'], sa taille totale est: 12.0
```

```
>>> plusCourtChemin python main.py
[[inf  8.  6. inf]
 [inf inf inf inf]
 [inf  3. inf inf]
 [inf  5.  1. inf]]
BellmanFord:
[0, 0, 0, None]
Le chemin d'accés à 'a' est ['a'], sa taille totale est: 0.0
Le chemin d'accés à 'b' est ['a', 'b'], sa taille totale est: 8.0
Le chemin d'accés à 'c' est ['a', 'c'], sa taille totale est: 6.0
Le sommet 'd' est inaccessible
```

Partie 6 : Influence du choix de la liste des flèches pour l'algorithme de Bellman-Ford

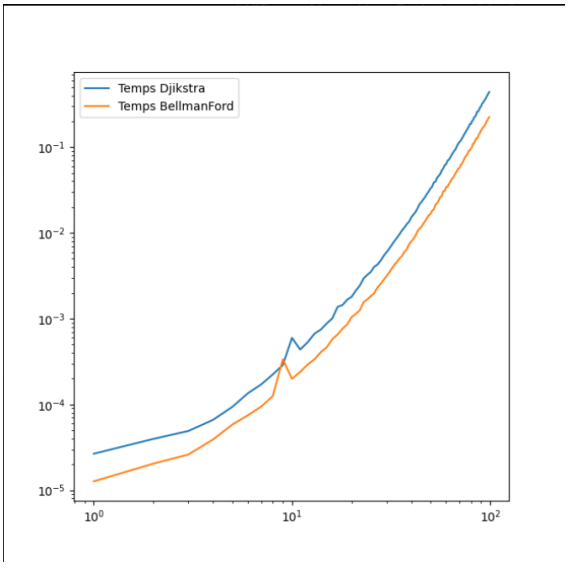
Partie 7 : Comparaison expérimentale des complexités

Les fonctions Dij(n) et BF(n) qui calcule le temps de calcul utilisé par ces dernières

```python
def Dij(n):
        M = gm.genMatrix(n, 0, 15, 50)
        return dj.Djikstra(M, 0)
def BF(n):
        M = gm.genMatrix(n, 0, 15, 50)
        return bf.BellmanFord(M, 0)
```

Ainsi pour un n allant de 1 à 100 nous obtenons les résultats suivants :

- Graphiquement :



- Temps de calcule :

```
>>> plusCourtChemin python main.py
9.150002245000906
4.692998300001818
```

- On obtient donc 9.15 avec l'algorithme Dijkstra et 4,69 avec l'algorithme de BellMan Ford

On remarque que L'algorithme de BellMan Ford est le plus rapide.
La complexité de Dijkstra est : O(a+n*log(n))

La complexité de Bellman Ford est : 0(|S||A|) ou |S| est le nombre de sommet et |A| le nombre de branches.

Ainsi avec ces 2 informations on conclut que Bellman Ford est plus intéressant pour des cas avec peu de sommets et de branches, il faut donc pour des un nombre petit de sommets choisir BellMan Ford et pour un grand nombre Dijkstra, bien sur il ne faut pas de poids négatif.

Voies d'améliorations :
Notre représentation de Dijkstra admet un défaut, en effet celui-ci met fin au programme lorsque la fermeture transitive de la matrice en entrée n'est pas complète sur la ligne du sommet de départ. Afin de remédier à cela, il faudrait que nous adaptions un algorithme de fermeture transitive pour les matrices en nombres flottants et en tester la fermeture au début de l'exécution de Dijkstra afin de pouvoir retourner une erreur "propre".