

Bernard-Nicod Vivien
Goujon Maël
Höffler Marie-Ange
Koh You Chen

Groupe C
Équipe 1

S.A.E. S2.02 - EXPLORATION ALGORITHMIQUE D'UN PROBLÈME

TABLE DES MATIÈRES

| | |
|--|----|
| I Comparaison d'algorithmes de plus courts chemins..... | 4 |
| 1 Connaissance des algorithmes de plus courts chemins..... | 4 |
| 1.1 <i>Présentation de l'algorithme de Djikstra</i> | 4 |
| 1.2 <i>Présentation de l'algorithme de Bellman-Ford</i> | 6 |
| 2 Dessin d'un graphe et d'un chemin à partir de sa matrice..... | 6 |
| 2.1 <i>Dessin d'un graphe</i> | 8 |
| 2.2 <i>Dessin d'un chemin</i> | 8 |
| 3 Génération aléatoire de matrices de graphes pondérées..... | 9 |
| 3.1 <i>Graphes avec 50% de flèches</i> | 9 |
| 3.2 <i>Graphes avec une proportion variables p de flèches</i> | 10 |
| 4 Codage des algorithmes de plus court chemin..... | 11 |
| 4.1 <i>Codage de l'algorithme de Dijkstra</i> | 11 |
| 4.2 <i>Codage de l'algorithme de Belman-Ford</i> | 12 |
| 5 Influence du choix de la liste ordonnée des flèches pour l'algorithme de Bellman-Ford..... | 13 |

| | |
|--|----|
| 6 Comparaison expérimentale des complexités..... | 14 |
| <i>6.1 Deux fonctions "temps de calcul"</i> | 14 |
| <i>6.2 Comparaison et identification des deux fonctions temps.....</i> | 15 |
| <i>6.3 Conclusion.....</i> | 16 |
| II Seuil de forte connexité d'un graphe orienté..... | 17 |
| 7 Test de forte connexité..... | 17 |
| 8 Forte connexité pour un graphe avec p=50% de flèches..... | 18 |
| 9 Détermination du seuil de forte connexité..... | 19 |
| 10 Etude et identification de la fonction seuil..... | 20 |
| <i>10.1 Représentation graphique de seuil(n)</i> | 20 |
| <i>10.2 Identification de la fonction seuil(n)</i> | 20 |

COMPARAISON ALGORITHMES PLUS COURT CHEMIN

Connaissance des algorithmes de plus courts chemins

Présentation de l'algorithme de Djikstra

Explications :

Cet algorithme permet de trouver le chemin le plus court (s'il existe), d'un sommet de départ jusqu'à n'importe quel sommet du graphe. Il s'appuie sur une matrice pondérée du graphe représentant les sommets et les arrêtes du graphe.

Initialisation :

La distance du sommet de départ est initialisée à 0 et son prédecesseur à lui-même. Pour les autres sommets, s'ils sont successeurs du sommet de départ, la distance est initialisée à la valeur du poids de la flèche entre le départ et sommet en cours tandis que le prédecesseur est initialisé au sommet de départ.

Les sommets qui ne sont pas successeurs de d sont initialisés à une distance de $+\infty$ et un prédecesseur nul.

Itération :

Pour chaque sommet du graphe, on répète les actions suivantes :

1. Sélection du sommet non visité le plus proche
2. Calcul des nouvelles distances des sommets adjacents au sommet sélectionné
3. Actualisation des distances et prédecesseur pour chaque sommet si la nouvelle distance calculée est plus courte que l'ancienne
4. Marquer le sommet adjacent en cours comme étant visité et passer au prochain sommet non visité

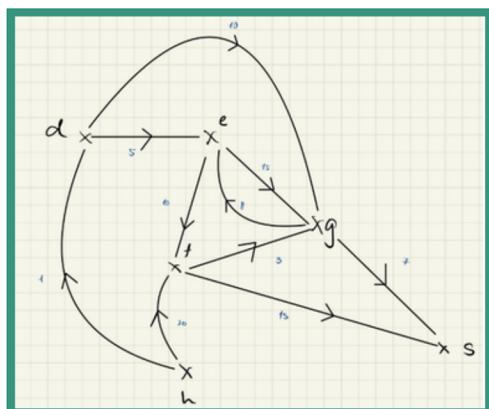
Connaissance des algorithmes de plus courts chemins

Présentation de l'algorithme de Djikstra

Résultat :

Une fois que tous les sommets ont été visités, les distances les plus courtes depuis le sommet de départ vers tous les autres sommets ont été déterminées durant la phase d'itération. Les chemins les plus courts peuvent être alors reconstruits à l'aide du dictionnaire des prédecesseurs en remontant depuis le sommet de destination jusqu'au sommet de départ.

Exemple de traitement :



Graphe et matrice correspondante utilisés pour l'exemple.

| | d | e | f | g | h | s |
|---|----------|----------|----------|----------|----------|----------|
| d | ∞ | 5 | ∞ | 10 | ∞ | ∞ |
| e | ∞ | ∞ | 10 | 15 | ∞ | ∞ |
| f | ∞ | ∞ | ∞ | ∞ | 3 | 15 |
| g | ∞ | 8 | ∞ | ∞ | ∞ | 7 |
| h | 1 | ∞ | 20 | ∞ | ∞ | ∞ |
| s | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ |

| sommet de départ : d | | | | | | |
|----------------------|------|--------|------------------------|---------|------------------------|------------------------|
| | d | e | f | g | h | s |
| Initialisation | 0, d | 5, v d | ∞ , \emptyset | 10, v d | ∞ , \emptyset | ∞ , \emptyset |
| Itération 1 | | 5, v d | ∞ , \emptyset | 10, v d | ∞ , \emptyset | ∞ , \emptyset |
| Itération 2 | | | 15, v e | 10, v d | ∞ , \emptyset | ∞ , \emptyset |
| Itération 3 | | | | 15, v e | ∞ , \emptyset | 7, v g |
| Itération 4 | | | | | ∞ , \emptyset | f (15) |
| Itération 5 | | | | | | ∞ , \emptyset |

Se lit "poids = 5 venant du sommet d"
Symbolise qu'il n'y a pas de flèche entre d et f
Plus petite valeur trouvée sur la ligne. Ici 5, qui correspond au sommet e (colonne de e). Ainsi, on ne revient plus sur le sommet e.
Si 2 valeurs sont identiques (par exemple si l'on avait deux fois 5, il aurait fallu choisir l'une ou l'autre)
10 (flèche ef) + 5 (flèche d vers e) = 15 < ∞
Plus petite valeur entre 10 venant de d, trouvé précédemment, et 20 venant de e (obtenu avec 5 (e (5) choisi précédemment) + 15 (flèche eg))

Explication :

Additionner les poids et lire le "venant de" dans les cellules encadrées en rouge

Interprétation :

Le plus court chemin entre d et s est de poids 17. Il s'agit de d-g-s.

Connaissance des algorithmes de plus courts chemins

Présentation de l'algorithme de Bellman-Ford

Explication :

À l'instar de l'algorithme de Dijkstra, celui-ci permet de trouver le plus court chemin dans un graphe pondéré d'un sommet à un autre. Toutefois, contrairement à l'algorithme de Dijkstra, l'algorithme de Bellman-Ford permet de trouver les chemins les plus courts sur des graphes pondérés avec des valeurs négatives. Il peut être intéressant de l'utiliser par exemple si l'on cherche à déterminer le chemin le plus froid pour aller d'une ville A à une ville B au Canada.

Initialisation :

Au départ, nous initialisons 3 listes de données :

1. Une liste des flèches du graphe
2. Un dictionnaire de prédécesseurs de chaque sommet
3. Un dictionnaire de distance de chaque sommet

Le dictionnaire de prédécesseurs affecte à chaque sommet la valeur *None* sauf pour le sommet de départ qui a lui même comme prédécesseur

Le dictionnaire de distance affecte à chaque sommet la valeur infinie sauf pour le sommet de départ qui a une distance de 0 par rapport à lui-même

Itération :

Tant que des modifications ont lieu ou que l'on n'a pas fait plus d'itérations qu'il n'y a de sommets dans le graphe faire :

- Parcourir la liste de flèches et pour chaque flèche effectuer cette vérification :
 - Si le poids de la flèche additionné à la distance du sommet de départ de la flèche est inférieur à la distance du sommet d'arrivée de la flèche, changer la distance du sommet d'arrivée de la flèche pour la valeur de la somme utilisée pour la comparaison.

Connaissance des algorithmes de plus courts chemins

Présentation de l'algorithme de Bellman-Ford

Résultat :

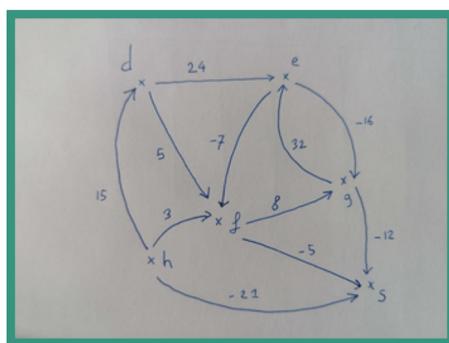
Dès lors qu'il n'y a plus de modifications dans les dictionnaires de distances et de prédécesseurs ou que le nombre de parcours de la liste de flèches a été itéré autant de fois qu'il y a de sommets dans le graphe, nous pouvons traiter les résultats.

Si le nombre d'itération est égal au nombre de sommets, cela signifie qu'il y a un cycle absorbant dans le graphe. C'est un cycle qui boucle sur lui-même à cause des poids négatifs.

Sinon, pour chaque sommet du graphe nous pouvons déterminer en regardant sa distance dans le dictionnaire de distance s'il existe un chemin allant du sommet de départ à lui.

Si ce chemin existe (c'est-à-dire que la distance n'est pas infini) nous pouvons le reconstruire à partir du dictionnaire de prédécesseurs en le lisant à l'envers en partant du sommet d'arrivée.

Exemple de traitement :



| | d | e | f | g | h | s |
|----------------|------|-------|-------|-------|-------|-------|
| Initialisation | 0, d | +∞, ∅ | +∞, ∅ | +∞, ∅ | +∞, ∅ | +∞, ∅ |
| T ₁ | 0, d | 24, d | 5, d | 8, e | +∞, ∅ | -4, g |
| T ₂ | 0, d | 24, d | 5, d | 8, e | +∞, ∅ | -4, g |

Remarque : fin de parcours, fin de modifications.

Ici, en observant le tableau représentant l'actualisation des dictionnaires de prédécesseurs et de distance, nous pouvons reconstruire le plus court chemin pour aller à **s** en partant de **d**.

Pour ce faire, il suffit de chercher le prédécesseur de **s**, puis le prédécesseur de ce prédécesseur et ainsi de suite jusqu'à ce que nous retrouvions le sommet de départ.

Nous obtenons ainsi comme résultat :

- Plus court chemin de **d** à **s** : inverse de [**s, g, e, d**] soit [**d, e, g, s**]
- Distance du plus court chemin de **d** à **s** : -4

Dessin d'un graphe et d'un chemin à partir de sa matrice

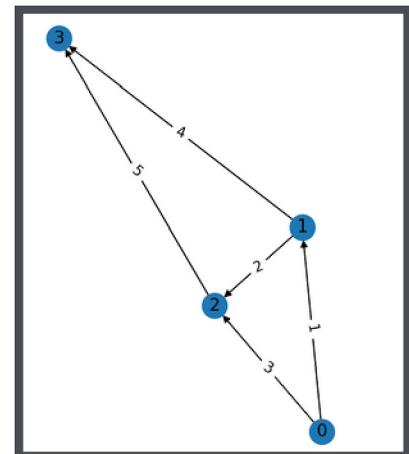
Dessin d'un graphe

Afin de réaliser un tracé visuel du graphe, nous avons utilisé diverses bibliothèques python :

- **networkx** pour la création d'un graphe orienté pondéré à partir de la matrice d'incidence et le dessin du graphe et des poids
- **matplotlib.pyplot** pour afficher le graphe généré par networkx

À l'aide de ces deux bibliothèques voici les principales fonctions que nous avons utilisées :

- `nx.DiGraph()` : permet d'avoir un graphe orienté
- `G.add_edge()` : ajoute l'arête au graphe G
- `nx.spring_layout()` : détermine la position de chaque noeud dans le graphe
- `nx.draw()` : dessine le graphe G avec les coordonnées de pos
- `nx.get_edge_attributes()` : Récupère les poids des arêtes
- `nx.draw_networkx_edge_labels()` : Ajoute les labels sur le graphe
- `plt.show()` : Affichage du graphe dessiné

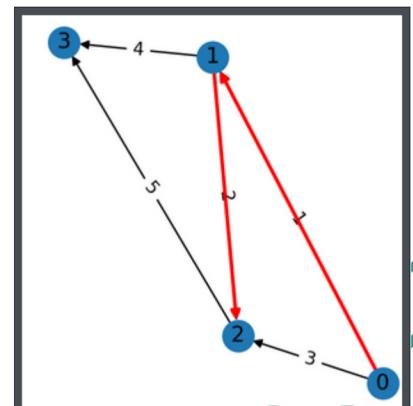


Dessin d'un chemin

Le code utilisé afin de réaliser le dessin d'un chemin est similaire à celui du dessin du graphe, la seule différence est l'ajout de la coloration d'un chemin.

Principaux ajouts :

- Définition d'un chemin à colorer avec une liste de sommets ordonnés
- Définition des arêtes de ce chemin
- `nx.draw_networkx_edges()` : Colorisation des arêtes



Génération aléatoire de matrices de graphes pondérées

Graphes avec 50% de flèches

Afin de produire des graphes aléatoires, nous avons réalisé une fonction graphe ayant pour signature : `def graphe(taille, borneInf, borneSup)`

- taille représente la taille de la matrice à générer.
- borneInf représente la valeur minimale de la fourchette des poids à renseigner.
- borneSup représente la valeur maximale de la fourchette des poids à renseigner.

Voici son fonctionnement :

- 1.Création d'une matrice binaire de taille renseignée ayant 50% de 0 et de 1.
- 2.Parcours de la matrice et remplacement des 1 par une valeur dans la fourchette renseignée. Remplacement des 0 par la valeur infini.
- 3.Retour de la matrice.

Exemple de matrices produites avec la fonction graphe :

```
print(graphe(10,0,100))
```

```
[[inf  8. 32.  inf 12. 25.  inf inf inf 48.]  
[55. 60. 44.  inf 17. 19.  inf 54. 8. inf]  
[20. 26.  inf inf 26.  inf 64. 42.  inf inf]  
[61.  inf inf 8.  inf 81. 68. 78.  inf inf]  
[inf 35. 2.  inf inf 16.  inf 51.  inf 1.]  
[inf 76. 51.  inf 18. 87.  inf inf inf inf]  
[71.  inf inf inf inf 18.  inf 85. 20. 98.]  
[43.  inf 70. 75. 74.  inf 17. 23. 8. 80.]  
[inf 38.  inf inf 94. 90. 27.  inf inf inf]  
[ 8.  inf inf inf inf 5.  inf 33. 34.]]
```

```
print(graphe(5,-100,100))
```

```
[[ 4.  inf 70.  inf inf  
[-88.  inf 37.  inf 97.]  
[ 15. -89.  inf 67. -49.]  
[ inf inf inf 12. inf]  
[ inf 45. -74. -76. 44.]]
```

```
[[inf 20. 73. 66. 79.  inf 47.  inf 84. inf]  
[15. 32. 8.  inf 17. 9. 50.  inf inf 83.]  
[inf 12. 33. 69.  inf 10.  inf 97. 88. 59.]  
[inf 82.  inf 29. 68.  inf inf 13.  inf inf]  
[55. 6.  inf 63. 27.  inf inf 86. 31. 51.]  
[inf inf inf 16. 97. 84.  inf inf 51. inf]  
[ 2.  inf inf 20. 19.  inf 14.  inf inf inf]  
[inf 88.  inf inf 79. 96.  inf 32. 11. 84.]  
[inf inf 13. 23.  inf 31. 46. 56.  inf 5.]  
[inf inf inf 9.  inf inf 8.  inf 5. inf]]
```

```
[[ inf 57. 34. 26.  inf]  
[ inf 66.  inf inf inf]  
[ inf inf inf 10. -88.]  
[ inf inf inf inf 69.]  
[ -1.  inf -92.  inf inf]]
```

```
[[ -85.  inf inf -73.  inf]  
[ -68.  inf inf -38.  inf]  
[ inf inf -27. 86.  inf]  
[ -93. -18. 46. -49.  inf]  
[ -19. -28.  inf -18. 51.]]
```

Génération aléatoire de matrices de graphes pondérées

Graphes avec une proportion variables p de flèches.

Afin de produire des graphes pondérés aléatoires avec une proportion variable de flèches nous avons réalisé une fonction graphe ayant pour signature :

```
def graphe (taille, proportion, bornelInf, borneSup)
```

- taille représente la taille de la matrice à générer.
- proportion (de 0 à 1) représente la proportion de flèches dans la matrice.
- bornelInf représente la valeur minimale de la fourchette des poids à renseigner.
- borneSup représente la valeur maximale de la fourchette des poids à renseigner.

Voici son fonctionnement :

- 1.Création d'une matrice binaire de taille renseignée ayant une proportion indiquée de 1.
- 2.Parcours de la matrice et remplacement des 1 par une valeur dans la fourchette renseignée. Remplacement des 0 par la valeur infini.
- 3.Retour de la matrice.

Exemple de matrices produites avec la fonction graphe2 :

```
print(graphe2(10,0.8,0,100))
```

```
[[28. inf 26. 80. 94. 99. 38. 25. inf 49.]
 [77. 32. 91. 88. 57. 20. 29. 95. 82. 73.]
 [80. 30. 37. inf inf inf 96. 10. 54. 57.]
 [90. 74. inf 98. 91. 83. inf 43. 3. 70.]
 [23. 25. 76. 10. 83. inf inf 74. 73. 9.]
 [81. 40. inf inf inf 86. 42. 64. 63. 69.]
 [15. 29. 17. 62. 80. inf 23. inf 99. 71.]
 [59. 52. 8. 55. 58. inf 38. 74. 41. 4.]
 [66. 61. 36. inf 50. 19. 84. 55. 85. 12.]
 [76. 57. 89. 56. 58. 92. 21. 56. 63. 50.]]
```

```
print(graphe2(10,0.1,0,100))
```

```
[[14. inf inf inf inf inf inf inf inf]
 [inf inf inf inf inf inf inf inf 55. inf]
 [inf inf inf inf inf inf inf inf 23.]
 [inf inf inf inf inf inf inf inf inf]
 [inf inf inf inf inf inf inf inf inf]
 [inf inf inf inf inf inf inf inf inf]
 [inf inf inf inf inf inf inf inf 95. inf inf]
 [inf inf inf inf 27. inf inf inf inf inf]
 [inf inf inf inf inf inf inf inf inf]
 [inf inf inf inf inf inf inf inf inf 82.]
 [inf inf inf inf inf inf inf inf inf inf]]
```

Codage des algorithmes de plus court chemin

Codage de l'algorithme de Dijkstra

Afin de trouver le plus court chemin d'un graphique de poids pondérés, nous avons créé une fonction reprenant l'algorithme de Dijkstra ayant pour signature :

def Dijkstra (M, d)

- M : matrice représente la matrice pondérée dans laquelle nous allons rechercher les plus courts chemins.
- d : Sommet représentent le sommet de départ qui cherche à joindre les autres sommets.

Voici son fonctionnement :

- 1.Création d'un dictionnaire de distances et d'un dictionnaire de prédécesseurs initialisés respectivement à **infini** et **None** pour tous les sommets du graphe (sauf pour le sommet de départ qui est initialisé à 0 en distance).
- 2.Création d'un ensemble de sommets non choisis initialisé avec tous les sommets du graphe.
- 3.Tant qu'il reste des sommets non choisis dans la liste des sommets non choisis, on recherche le sommet ayant la distance minimale parmi les sommets restants dans la liste des sommets non choisis.
- 4.On compare ce sommet aux autres sommets et on actualise si nécessaire les prédécesseurs et distances des sommets adjacents au sommet choisi puis on retire le sommet initial de la liste des sommets non choisis.
- 5.Avec les dictionnaires finaux de distance et prédécesseurs nous construisons les plus courts chemins du sommet de départ aux autres sommets s'ils existent.

Exemple de parcours produit pour une matrice générée aléatoirement :

M=graphe2(5,0.4,0,100)

Dijkstra(M,0)

```
{0: [0, [0]],  
 1: [inf, 'sommet non joignable à d par un chemin dans le graphe G'],  
 2: [36.0, [0, 2]],  
 3: [78.0, [0, 3]],  
 4: [52.0, [0, 2, 4]]}
```

Codage des algorithmes de plus court chemin

Codage de l'algorithme de Bellman-Ford

Afin de trouver le plus court chemin d'un graphique de poids pondérés prenant en compte les valeurs négatives, nous avons créé une fonction reprenant l'algorithme de Bellman-Ford ayant pour signature :

```
def BellmanFord (matrice, Sommet, ParcoursType)
```

- matrice représente la matrice pondérée dans laquelle nous allons rechercher les plus courts chemins.
- Sommet représente le sommet de départ qui cherche à joindre les autres sommets.
- ParcoursType représente le type de parcours que l'on veut utiliser pour initialiser la liste des sommets à visiter.

Voici son fonctionnement :

1. Crédit d'une liste de sommets de la matrice selon le parcours renseigné
2. Crédit d'une liste de flèches de la matrice représentées par un tuple (sommet de départ - sommet d'arrivée - poids de la flèche)
3. Initialisation de dictionnaires enregistrant pour chaque sommet son prédécesseur et sa distance avec le sommet de départ
4. Boucle de parcours de la liste de flèches et d'actualisation des distances et prédécesseurs de chaque sommet lorsque le poids de la flèche en cours additionné au poids du sommet de la flèche est inférieur au poids de la pointe de la flèche.
5. Calcul et retour des parcours pour chaque sommet

Exemple de parcours produit pour une matrice générée aléatoirement :

```
M=graphe2(5,0.4,-100,100)
BellmanFord(M,0, "LARGEUR")
```

```
('Chemin le plus court du sommet 0 pour aller à 3 : 22.0',
'itinéraire du sommet 0 au sommet 3 : [0, 3]')
('Chemin le plus court du sommet 0 pour aller à 4 : -9.0',
'itinéraire du sommet 0 au sommet 4 : [0, 3, 1, 4]')
('Chemin le plus court du sommet 0 pour aller à 1 : -7.0',
'itinéraire du sommet 0 au sommet 1 : [0, 3, 1]')
Le sommet 2 n'est pas joignable à 0
```

Algorithme de Bellman-Ford

Influence du choix de la liste ordonnée des flèches

L'algorithme de Bellman Ford étant dépendant de l'ordre de la liste de flèches qu'il parcours, nous avons adapté notre fonction `BellmanFord` afin de prendre en compte un type de parcours à renseigner en attribut de fonction.

Ainsi, selon le choix de l'utilisateur la liste de flèches peut être construite de 3 façons différentes :

- A partir d'un **parcours linéaire** : les flèches sont construites à l'aide d'une liste de sommets ordonnée d'un point de vue alphabétique.
- A partir d'un **parcours en largeur** complété : les flèches sont construites à l'aide d'une liste ordonnée de sommet produite via le parcours en largeur pour laquelle nous avons rajouté les sommets manquants si le parcours ne mène pas à tous les sommets.
- A partir d'un **parcours en profondeur** : même processus que pour le parcours en largeur mais avec cette fois-ci un algorithme de parcours en profondeur.

Nous avons aussi modifié notre programme afin que la fonction renvoie le nombre de cycles effectués avant de trouver l'ensemble des chemins.

Exemple de résultat du nombre de cycles selon les 3 différents parcours pour une matrice de taille 500 et un sommet de départ à 0 :

```
M=graphe2(500,0.01,1,100)
TestsBellmanFordParcours(M, 0)

Nombre de cycle avec parcours lineaire : 9
Nombre de cycle avec parcours en profondeur : 8
Nombre de cycle avec parcours en largeur : 5
```

*Les résultats suite à multiples exécutions tendent à montrer que Bellman Ford est plus performant lorsqu'il s'agit d'utiliser un parcours en largeur.
Cela peut s'expliquer vis à vis du fait que le parcours en largeur nous indique directement les sommets adjacents au sommet de départ. Ainsi l'algorithme à plus de chances d'actualiser les sommets dans le bon ordre (car il utilise une liste de flèches adjacentes) et ainsi de réduire le nombre de cycles à effectuer.*

Comparaison expérimentale des complexités

Fonctions de temps de calcul

Afin d'analyser la différence de complexité entre l'algorithme de Dijkstra et celui de Bellman-Ford, nous avons programmé deux fonctions identiques permettant de calculer le temps d'exécution de l'algorithme pour une matrice de taille donnée.

Leurs signatures respectives sont les suivantes :

```
def TempsBF (taille, affichage)
def TempsDij (taille, affichage)
```

- Taille : la taille de la matrice testée
- Affichage : Boolean précisant si nous voulons ou non un affichage dans la console du temps d'exécution

Voici leur fonctionnement :

- 1.Création d'une matrice de graphe pondéré, de taille renseignée à l'appel de la fonction.
- 2.Lancement d'un chronomètre.
- 3.Application de l'algorithme de Dijkstra / Bellman-Ford (selon la fonction) sur la matrice.
- 4.Arrêt du chronomètre.
- 5.Renvoie de la valeur du chronomètre.

Exemple de résultats d'exécution :

TempsBF (5000, True)

Le temps d'execution avec BellmanFord pour une matrice de taille 5000 est : 33.77164689998608

TempsDij (5000, True)

Le temps d'execution avec dijkstra pour une matrice de taille 5000 est : 9.936726799991447

Comparaison expérimentale des complexités

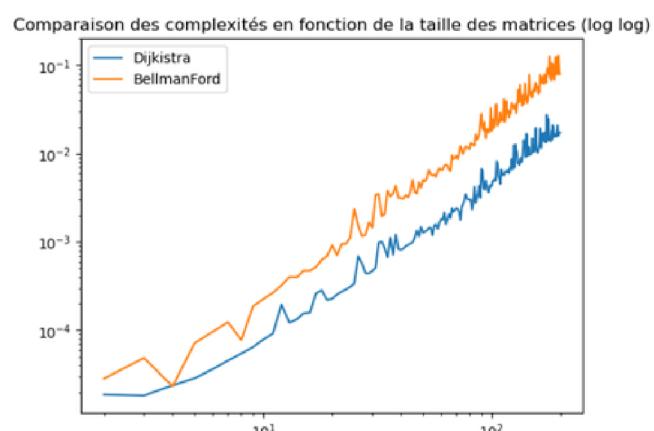
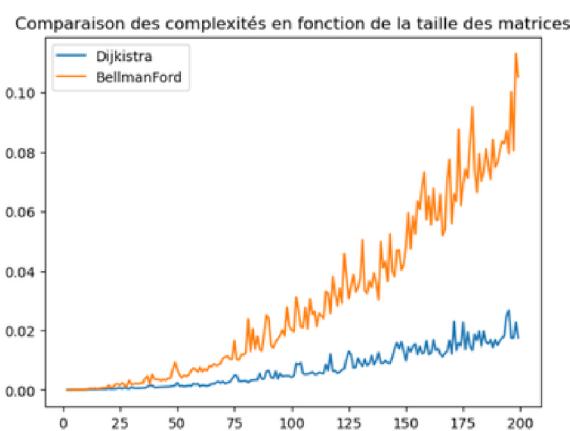
Comparaison des temps de calculs

À l'aide des fonctions de calculs de temps d'exécution des différents algorithmes, nous avons créé une fonction plus générale réalisant une série d'exécution de ces algorithmes afin de déterminer l'évolution du temps de calcul pour chaque algorithme en fonction de la taille de la matrice.

Voici l'algorithme de la fonction :

1. Initialisation de listes vides stockant les temps d'exécution pour Bellman-Ford et Dijkstra, et les tailles de matrices correspondantes
2. Boucle d'appel des fonctions de calcul de temps d'exécution avec une taille de matrice augmentant. Les résultats sont ajoutés aux listes correspondantes.
3. Affichage du graphique avec 2 courbes : temps d'exécution de Bellman-Ford en fonction de la taille de la matrice et temps d'exécution de Dijkstra en fonction de la taille de la matrice.

Résultats d'exécution pour des matrices de proportion de 50% de flèches :

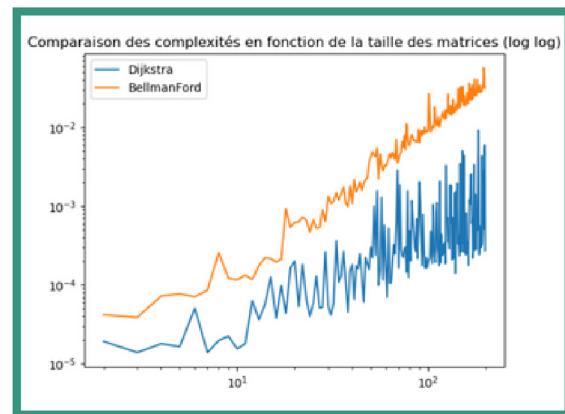
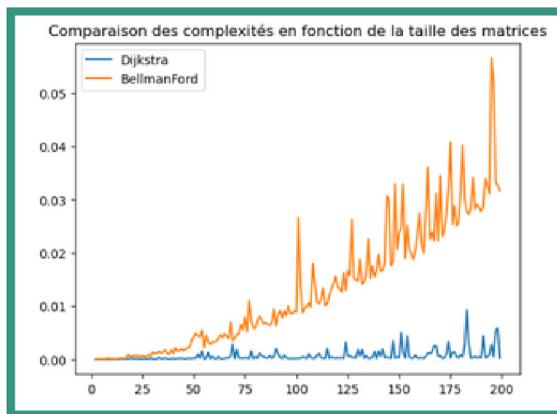


Nous pouvons observer graphiquement que l'algorithme de Dijkstra est le plus rapide des deux, et ce, proportionnellement à la taille de la matrice. Sa complexité est par extension plus faible que celle de l'algorithme de Bellman Ford.

Si une fonction $t(n)$ est approximativement de type cn^a pour n grand alors sa représentation graphique en coordonnées log-log est approximativement une droite de pente a , car $\log(t(n)) = \log(c) + a * \log(n)$ et cette équation est de la forme $y = mx + c$. Ici, nous pouvons observer sur le graphique en log log que les deux fonctions sont polynomiales. Il est alors possible de calculer le coefficient directeur des deux droites : a vaut environ 2.064 pour Dijkstra et environ 1.941 pour Bellman-Ford.

Comparaison expérimentale des complexités

Comparaison des temps de calculs



Les résultats de performance sont d'autant plus probants avec une proportion de flèches proportionnelle à la taille de la matrice (ici une proportion de flèches de 1/Taille).

Conclusion

Nos différents résultats montrent que l'algorithme de Dijkstra a une complexité plus faible que celui de Bellman-Ford. Il est donc plus intéressant d'utiliser cet algorithme pour la recherche d'un plus court chemin. Néanmoins, l'algorithme de Dijkstra ne fonctionne pas avec les valeurs négatives tandis que celui de Bellman-Ford est capable de calculer un plus court chemin dans un graphe à poids négatifs.

Selon le type de graphe à analyser nous pouvons conclure ainsi :

- Si le graphe est un graphe à poids positifs, afin de trouver un plus court chemin il faut utiliser l'algorithme de Dijkstra
- Si le graphe est un graphe à poids variables (positifs et négatifs) il faut utiliser l'algorithme de Bellman-Ford

Pour être plus précis au regard de l'algorithme de Bellman-Ford, nous compléterons en disant qu'il faut l'utiliser avec une liste de flèches établie avec un parcours en largeur afin de réduire sa complexité.

GRAPHE ORIENTÉ ET SEUIL DE FORTE CONNEXITÉ

Test de forte connexité

Dans un graphe **G**, deux sommets **S1** et **S2** sont dans une même composante fortement connexe s'il existe un chemin de **S1** vers **S2** et un chemin de **S2** vers **S1** dans G. On dit que dans **G**, **S1** voit **S2** et **S2** voit **S1**.

Un graphe est fortement connexe s'il n'est composé que d'une seule composante fortement connexe.

Si un graphe est fortement connexe, cela signifie que chaque nœud peut atteindre tous les autres nœuds du graphe. D'un point de vue matricielle, cela signifie que la fermeture transitive de la matrice d'incidence de ce graphe est une matrice composée entièrement de 1.

Afin de déterminer si un graphe est fortement connexe, nous avons programmé une fonction déterminant la forte connexité de sa matrice dont la signature est la suivante : **def fc (M)** :

- M : matrice d'incidence du graphe à vérifier

Voici l'algorithme de la fonction :

1. Parcours de la matrice et création de connexion (mettre un 1) entre deux sommets R et T s'il existe une arête RS et une arête ST dans la matrice.
2. Comparaison de la matrice à son homologue constitué uniquement de 1
3. Retour du résultat de la comparaison : si elles sont identique cela signifie que la fermeture transitive est composée uniquement de 1 et par extension, le graphe est fortement connexe.

Exemple de résultats d'exécution :

```
matrice indidence :   fermeture transitive :
[[1. 0. 0. 0. 0.]
 [0. 0. 1. 0. 0.]
 [0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0.]]
 [[1. 0. 0. 0. 0.]
 [0. 0. 1. 0. 0.]
 [0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]
 [0. 1. 1. 0. 0.]]
```

RETOUR : False

```
matrice indidence :   fermeture transitive :
[[0. 1. 0. 0. 0.]
 [0. 1. 1. 1. 0.]
 [1. 0. 1. 0. 1.]
 [0. 1. 0. 1. 0.]
 [0. 1. 0. 0. 0.]]
 [[1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1.]]
```

RETOUR : True

Forte connexité sur un graphe de 50% de flèches

Fonctions de calcul de pourcentage de graphes fortement connexes

On souhaite vérifier que lorsque l'on teste la fonction $fc(M)$ sur des matrices de taille n avec n grand, avec une proportion $p = 50\%$ de 1, on obtient presque toujours un graphe fortement connexe. Afin de calculer cela, nous avons implémenté une fonction dont la signature est la suivante :

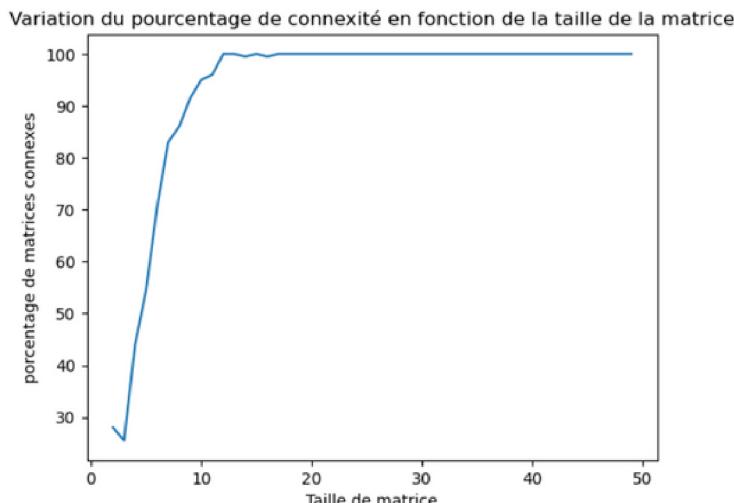
```
def test_stat_fc (n)
```

- n : la taille de matrice testée

Voici son fonctionnement :

1. Elle prend en entrée la taille n à tester.
2. Dans une boucle elle construit une matrice d'incidence aléatoire de 50% de flèches et teste sa forte connexité à l'aide de la fonction $fc(M)$ qui renvoie vrai ou faux selon la forte connexité de la matrice indiquée en paramètre.
3. Ce processus est répété 200 fois afin d'établir un résultat statistique pour la taille renseignée.
4. La fonction renvoie finalement le pourcentage de matrices fortement connexes sur le jeu d'essai réalisé.

A partir de quel taille de matrice il y a plus de 99% de chance que le graphe associé soit fortement connexe ?



Nous avons réalisé une seconde fonction testant cette fonction sur des tailles de matrices croissantes et traçant le graphique des résultats succincts. Nous pouvons ainsi observer sur ce graphique qu'à partir d'une taille de graphe supérieure à 13 composée à 50% de flèches, nous avons plus de 99% de chances d'obtenir un graphe fortement connexe.

Détermination du seuil de forte connexité

Afin de déterminer le seuil de forte connexité des matrices selon les pourcentages de flèches nous avons modifié la fonction `test_stat_fc` afin qu'elle prenne en paramètre la proportion de flèches que l'on veut tester.

Voici sa nouvelle signature :

```
def test_stat_fc2 (n, p)
```

- n : la taille de matrice testée.
- p : proportion de flèches du graphe. ex : 0.3 pour 30%.

Cette fonction est identique à la fonction initiale à la seule différence que lors de la création aléatoire de la matrice d'incidence, la proportion de 1 (flèches) n'est pas de 0.5, mais correspond à la proportion p indiquée en paramètre.

Par la suite, nous avons créé la fonction `seuil(n)` qui nous renvoie pour un pourcentage de flèches donné le seuil de forte connexité.

Voici sa signature :

```
def seuil(n)
```

- n : la taille de matrice testée

Voici son fonctionnement :

La fonction `seuil` va utiliser la fonction `def test_stat_fc2 (n, p)` afin de déterminer le seuil de forte connexité pour une taille de matrice donnée. Pour ce faire elle va tester le pourcentage de forte connexité en commençant avec une proportion de 100% de flèches.

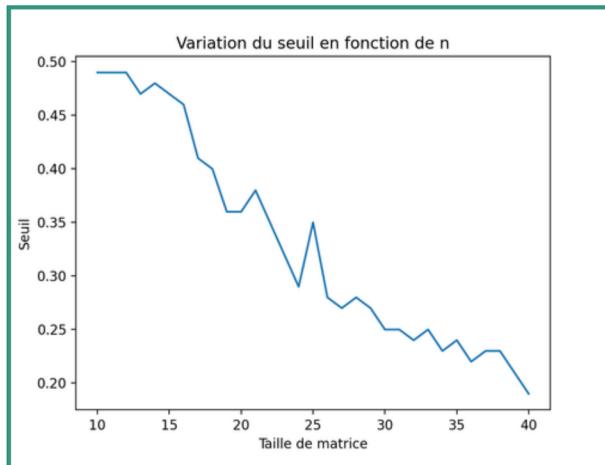
1. Tant que le pourcentage est au-dessus de 99%, elle teste avec une proportion diminuée de 1%.
2. Dès que le pourcentage de forte connexité est en dessous de 99%, elle renvoie le pourcentage de flèches testé.

Etude et identification de la fonction seuil

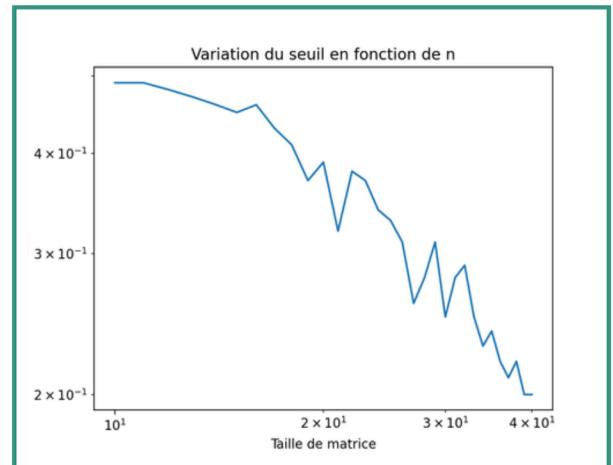
Représentation graphique du seuil

En reprenant les mêmes méthodes utilisées précédemment afin de construire les graphiques précédents, nous avons créé une nouvelle fonction permettant de représenter la suite seuil(n) sur l'intervalle [10, 40]

Représentation linéaire



Représentation logarithmique



Nous pouvons, à l'aide de ces graphiques, constater que la suite seuil est décroissante.

Identification de la fonction seuil

Au regard de la courbe du premier graphique, la fonction seuil a un comportement se rapprochant fortement d'une fonction puissance à exposant négatif.

Afin de déterminer l'exposant de cette fonction, nous avons transposé les données en données logarithmiques afin d'avoir une fonction asymptotiquement polynomiale de forme $y = ax + b$.

Par la suite, nous avons cherché la pente de cette fonction en calculant le rapport de division entre :

- la covariance de x et y.
- la variance de x..

Nous avons ainsi trouvé le coefficient directeur a valant environ -0.707, ce qui confirme notre hypothèse de départ qui supposait que la fonction seuil était asymptotiquement une fonction puissance à exposant négatif.