SAE 2.02 : Exploration algorithmique d’un problème

Groupe A1

ROS Natalia
HELBERT Titouan
VON LUNEN Shun

Première partie

1) Connaissance des algorithmes de plus courts chemins

1.1 Présentation de l’algorithme de Dijkstra

L'algorithme de Dijkstra est une méthode traditionnelle dans le domaine de l'informatique, conçue pour résoudre le défi du chemin le plus court dans un réseau orienté et doté de poids. Il est largement appliqué dans divers secteurs tels que les réseaux de télécommunication, la cartographie de routes, la robotique, etc.

Voici comment fonctionne l'algorithme de Dijkstra :

Initialisation : On commence par créer un tableau de distances depuis le nœud de départ vers tous les autres nœuds du réseau. La distance du nœud de départ est fixée à zéro, tandis que celles des autres nœuds sont définies à l'infini.

Sélection du nœud : Ensuite, on choisit le nœud non encore traité ayant la plus petite distance parmi ceux qui n'ont pas encore été visités. Ce nœud est souvent désigné comme le "nœud actuel".

Mise à jour des distances : Pour chaque nœud adjacent au nœud actuel, on calcule une distance temporaire en ajoutant la distance entre le nœud actuel et cet adjacent à la distance actuelle du nœud actuel. Si cette distance temporaire est inférieure à la distance actuellement connue pour cet adjacent, on met à jour la distance de cet adjacent avec cette nouvelle valeur.

Marquage du nœud actuel : Après avoir mis à jour toutes les distances, on marque le nœud actuel comme visité pour indiquer que sa distance minimale a été calculée.

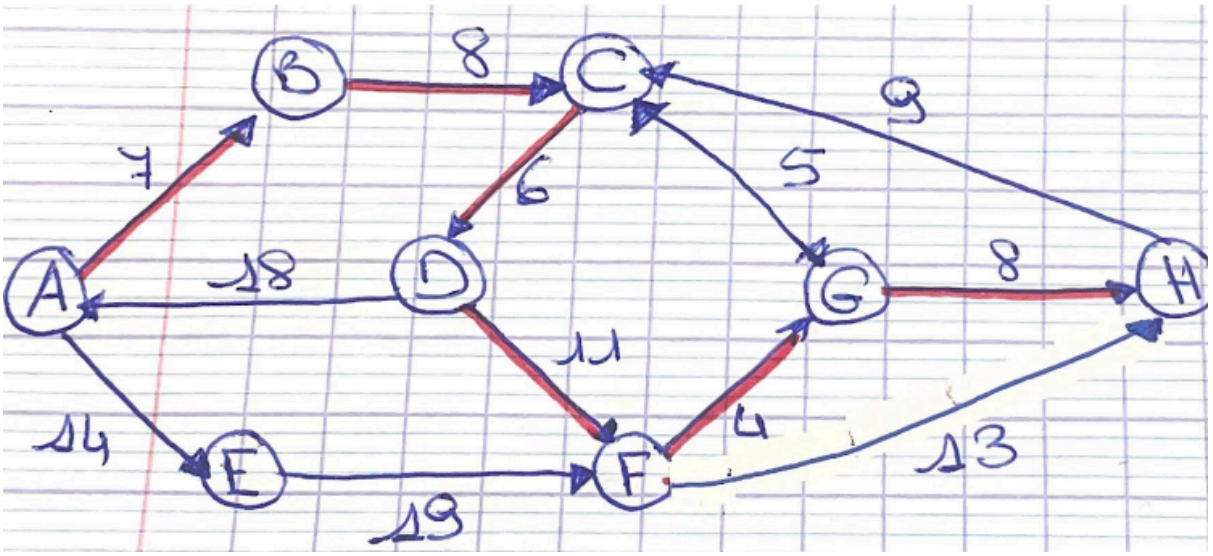
Répétition : On répète les étapes 2 à 4 jusqu'à ce que tous les nœuds du réseau aient été visités.

Au terme de l'exécution de l'algorithme, on dispose des distances minimales de tous les nœuds par rapport au nœud de départ, ainsi que des chemins les plus courts menant à chaque nœud. Cela permet de déterminer le chemin le plus court entre le nœud de départ et n'importe quel autre nœud du réseau.

L'algorithme de Dijkstra est particulièrement efficace pour les réseaux avec des poids positifs, mais il peut conduire à des résultats erronés si le réseau comporte des arêtes avec des poids négatifs. Dans de tels cas, d'autres méthodes comme l'algorithme de Bellman-Ford sont préférables.

Exemple :

Le graphe ci-dessous représente le réseau routier d'une région qui prend en compte le sens de la circulation, chaque arc représente une route à sens unique dont le poids est la distance en kilomètre entre deux sommets. Quel est l'itinéraire le plus court qui relie A à H ?



Le chemin le plus court(en rouge) est : A-B-C-D-F-G-H

1.2 Présentation de l'algorithme de Bellman-Ford

L'algorithme de Bellman-Ford permet de trouver le chemin le plus court entre un nœud source et tous les autres nœuds d'un graphe pondéré, même avec des arêtes de poids négatif. Voici comment il fonctionne :

1. **Initialisation** : La distance de chaque nœud par rapport au nœud source est initialisée à l'infini, sauf pour le nœud source qui est initialisé à zéro.

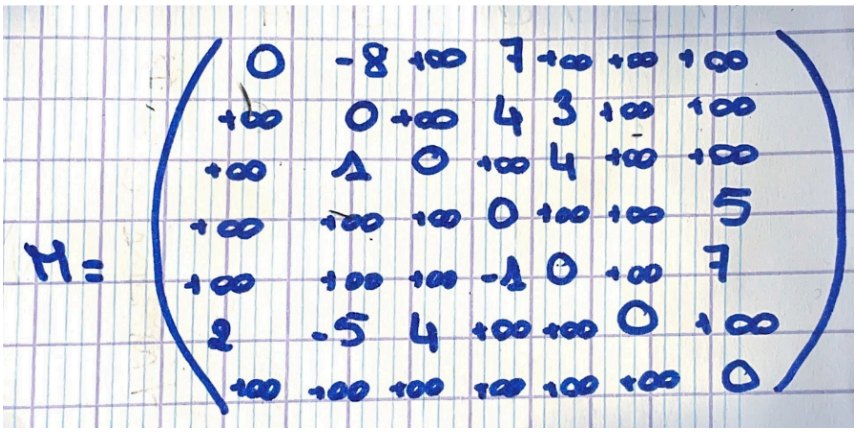
2. **Relaxation des arêtes** : Pour chaque arête du graphe, cette étape est répétée |V| - 1 fois (où |V| est le nombre de nœuds dans le graphe). Pour chaque arête (u, v) de poids w, on vérifie si la distance actuelle de u + w est inférieure à la distance de v. Si c'est le cas, la distance de v est mise à jour à la distance de u + w.

3. **Vérification des cycles de poids négatif** : Après avoir effectué l'étape précédente, on peut vérifier la présence d'un cycle de poids négatif dans le graphe. Si, après |V| - 1 itérations, une mise à jour de distance se produit encore, cela indique la présence d'un cycle de poids négatif, car le plus court chemin peut continuer à être amélioré à chaque itération. L'algorithme de Bellman-Ford détecte ce cas et indique qu'il n'y a pas de solution, car il n'y a pas de chemin le plus court dans la présence de cycles de poids négatif.

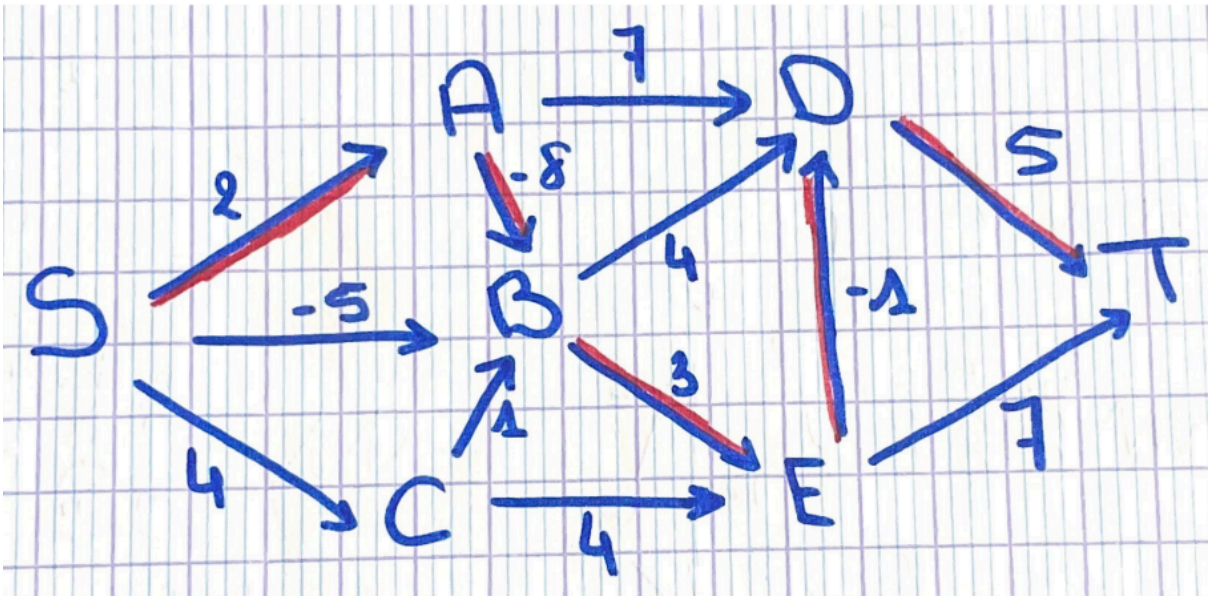
4. **Résultats** : Une fois l'algorithme terminé, les distances les plus courtes de chaque nœud au nœud source sont calculées et peuvent être utilisées pour retracer le chemin le plus court de n'importe quel nœud au nœud source.

L'algorithme de Bellman-Ford est moins efficace que l'algorithme de Dijkstra, mais il peut gérer les graphes avec des arêtes de poids négatif, ce que Dijkstra ne peut pas faire.

Exemple:



Il existe un chemin plus court (chemin en rouge) : SABEDT.



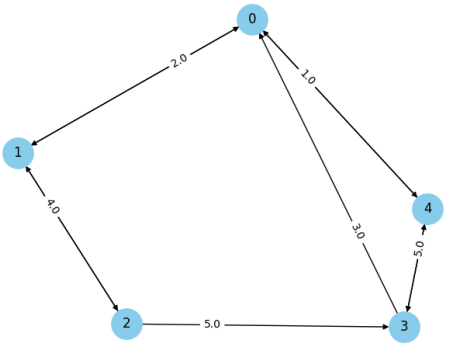
2) Dessin d'un graphe et d'un chemin à partir de sa matrice

2.1 Dessin d'un graphe

NetworkX est une bibliothèque Python conçue pour la création, la manipulation et l'étude de la structure, de la dynamique et des fonctions des réseaux complexes.

Exemple :

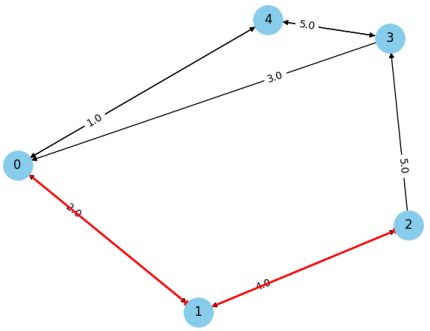
```
M = np.array([
    [0, 2, float("inf"), float("inf"), 1],
    [3, float("inf"), 4, 0, float("inf")],
    [float("inf"), 4, 0, 5, float("inf")],
    [3, 0, float("inf"), 0, 2],
    [6, 0, float("inf"), 5, 0]
])
```



2.2 Dessin d'un chemin

Exemple:

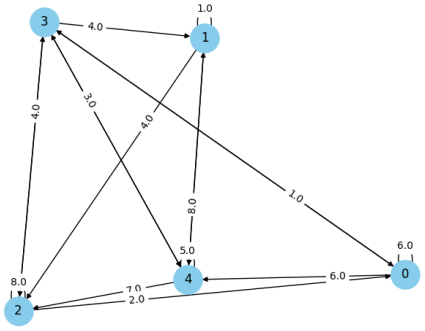
```
M = np.array([
    [0, 2, float("inf"), float("inf"), 1],
    [3, float("inf"), 4, 0, float("inf")],
    [float("inf"), 4, 0, 5, float("inf")],
    [3, 0, float("inf"), 0, 2],
    [6, 0, float("inf"), 5, 0]
])
```



3) Génération aléatoire de matrices de graphes pondérés

3.1 Graphes avec 50% de flèches

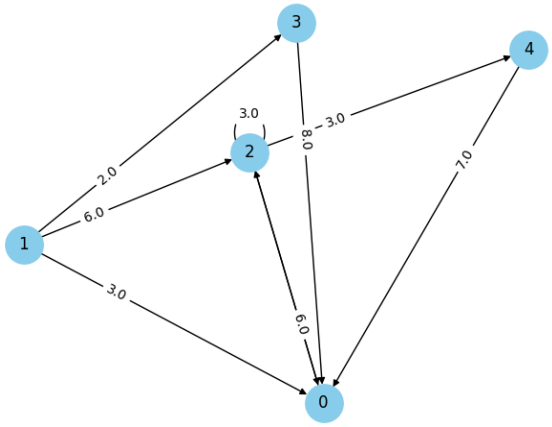
Exemple (avec n=5, a=1 et b=10):




```
[[ 6. inf inf  1.  6.]
 [inf  1.  4. inf  7.]
 [ 2. inf  8.  7. inf]
 [ 1.  4.  4. inf  3.]
 [inf  8.  7.  1.  5.]]
```

3.2 Graphes avec une proportion variable p de flèches

Exemple (avec n=5, p=0.3, a=1 et b=10):

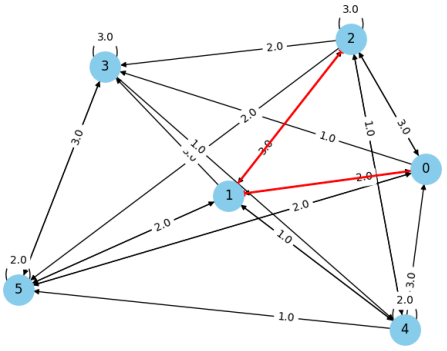



```
[[0. 2. 3. 1. 0. 2.]
 [3. 0. 3. 3. 1. 2.]
 [1. 2. 3. 2. 1. 2.]
 [0. 0. 0. 3. 1. 3.]
 [3. 3. 2. 0. 2. 1.]
 [1. 2. 0. 2. 0. 2.]]
```

4) Codage des algorithmes de plus court chemin

4.1 Codage de l'algorithme de Dijkstra

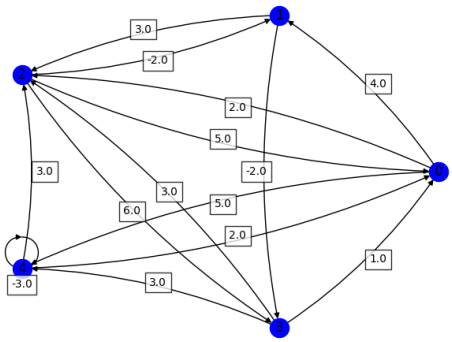
Exemple:



```
[[0. 2. 3. 1. 0. 2.]
 [3. 0. 3. 3. 1. 2.]
 [1. 2. 3. 2. 1. 2.]
 [0. 0. 0. 3. 1. 3.]
 [3. 3. 2. 0. 2. 1.]
 [1. 2. 0. 2. 0. 2.]]
```

4.2 Codage de l'algorithme de Bellman-Ford

Exemple:



```
M = np.array([
  [0, 4, 2,float("inf") , 5],
  [float("inf"), float("inf"), 3, -2, float("inf")],
  [5, -2, float("inf"), 6, 0],
  [1, float("inf"), 3, float("inf"), 3],
  [2, float("inf"), 3, float("inf"), -3]
])
```

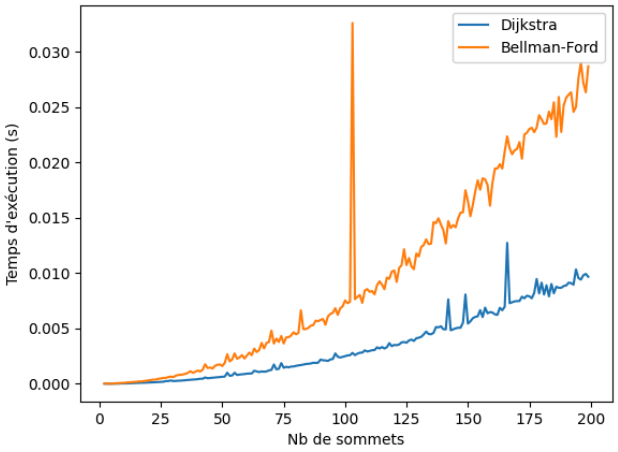
5) Influence du choix de la liste ordonnée des flèches pour l'algorithme de Bellman-Ford

6) Comparaison expérimentale des complexités

6.1 Deux fonctions "temps de calcul"

6.2 Comparaison et identification des deux fonctions temps

L'algorithme de Dijkstra est plus rapide que celui de Bellman-Ford comme nous pouvons le voir dans la représentation graphique ci-dessous.



Supposons que la fonction de temps t soit approximativement $t(n) = cn^a$ pour des grandes valeurs de n, où c et a sont des constantes.

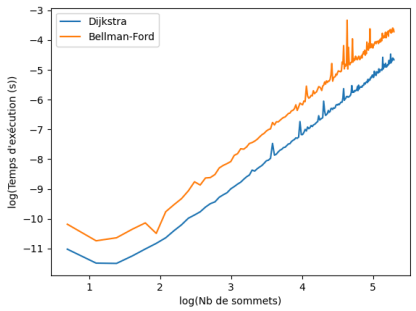
Prenons le logarithme de la fonction t des deux côtés : log(t(n))=log(cn^a)

On utilise la **propriété du logarithme**:
log(t(n))=log(c)+log(n^a)
log(t(n))=log(c)+a*log(n)

On obtient donc une **équation de la forme linéaire** y = ax + b
avec a=a (pente), b=log(c), x=log(n) et y=log(t(n))

Donc, la relation **log(t(n))=alog(n)+log(c)** montre que le logarithme de la fonction de temps est une **fonction linéaire** du logarithme du nombre de sommets n. La pente de cette ligne est a de la croissance polynomiale. Par conséquent, si la fonction de temps t croît comme cn^a, sa représentation en coordonnées log-log sera une **droite de pente a**.

Pour voir si les fonctions temps TempsDij et TempsBF sont de la forme polynomiale, on met sous la forme loglog et il faut que les courbes se rapprochent le plus possible d'une droite avec une pente.



L’exposant des fonctions Temps sont:

```
Exposant pour Dijkstra: 1.7921383803943192
Exposant pour Bellman-Ford: 1.8449182992452662
```

6.3 Conclusion

Généralement, l'algorithme de Dijkstra est plus rapide pour des graphes avec des poids positifs, tandis que Bellman-Ford est plus polyvalent mais plus lent.

7) Test de forte connexité

Pourquoi un graphe est fortement connexe **si et seulement** si la matrice de sa fermeture transitive n’a que des 1 ?

D’abord, il faut définir les termes de la question :

Fortement connexe :
Une composante fortement connexe d’un graphe G orienté est une partie du graphe qui vérifie la propriété suivante :
“Deux sommets distincts s et s’ sont dans un même composante fortement connexe s’il existe un chemin de s vers s’ et un chemin de s’ vers s dans G”

Remarque 1 :
Un graphe orienté est **fortement connexe** lorsqu’il n’a qu’une composante fortement connexe.

Fermeture transitive :
La fermeture transitive d’un graphe G est **le plus petit** graphe G(barre) qui “contient” G et est transitif (“contient” est relatif à l’ensemble des arêtes, l’ensemble des sommets restant identique).

Remarque 2 :
Il y a un chemin de i vers j dans G ⇔ il y a une flèche de i vers j dans G(barre) ⇔ M(barre)[i, j] = 1.

Raisonnement :
Si la fermeture transitive ne contient que des 1, cela signifie que chaque sommet est accessible depuis chaque autre sommet et vice-versa. Donc, pour chaque paire de

sommets (u,v), il existe un chemin de u à v et un chemin de v à u. Par conséquent, le graphe est fortement connexe.

Réciproquement :
La forte connexité garantit l'existence de chemins entre tous les couples de sommets Cela signifie que, dans la fermeture transitive T, chaque élément $T[i][j]=1$ car il y a un chemin de i à j.

8) Forte connexité pour un graphe avec p = 50% de flèches

"Lorqu'on teste cette fonction fc(M) sur des matrices de taille n avec n grand, avec une proportion p = 50% de 1 (et 50% de 0), on obtient presque toujours un graphe fortement connexe."

L'affirmation est vraie à partir de n=31.

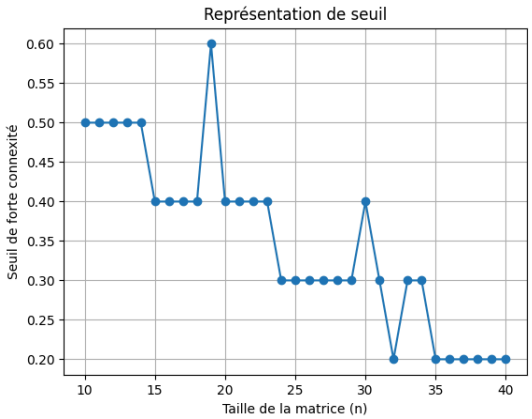
```
L'affirmation est vraie à partir de n=31
```

9) Détermination du seuil de forte connexité

10) Etude et identification de la fonction seuil

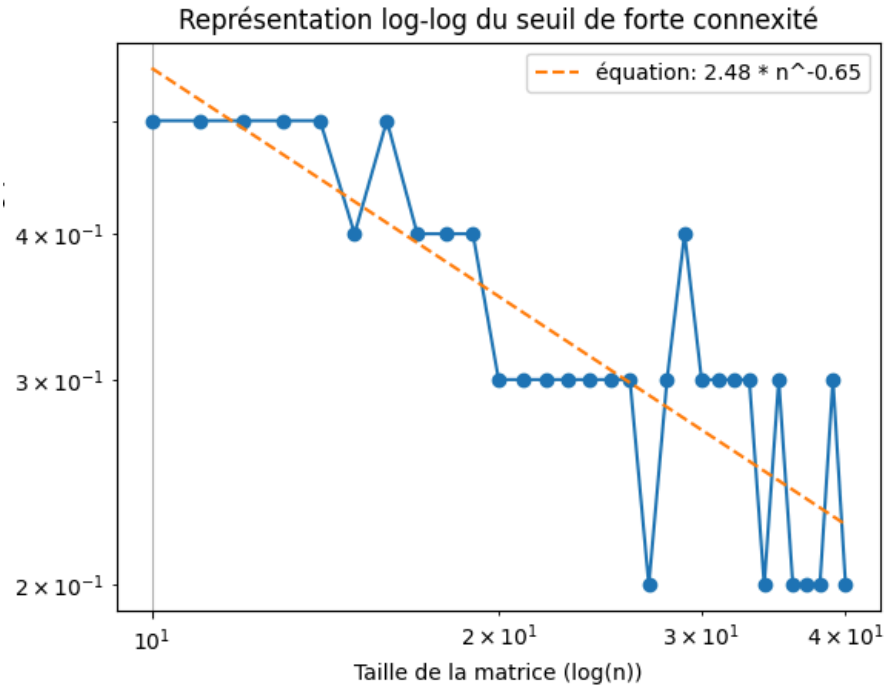
10.1 Représentation graphique de seuil(n)

La fonction du seuil pour n entre 10 et 40 est bien décroissante.



10.2 Identification de la fonction seuil(n)

La fonction seuil(n) est effectivement asymptotiquement une fonction puissance car en analysant la variation du seuil en fonction de la taille de la matrice, on peut s'apercevoir que nous pouvons réduire les variations du seuil en une droite d'équation: 2.48 * x^-0.65, soit une fonction puissance.



Ou sinon, nous pouvons faire une régression linéaire sur Python.

```
Les paramètres de la fonction puissance sont : a = -0.6497029560500076, c = 2.475976206709868
La fonction seuil(n) est approximativement 2.475976206709868 * n^-0.6497029560500076
```