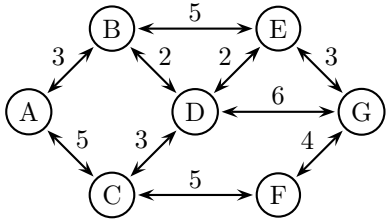
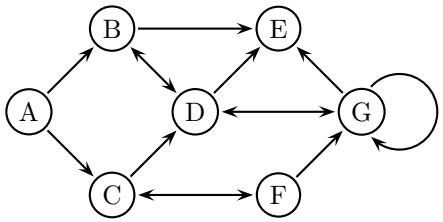
Informatique - Python - TP6
Graphes Pondérés : Algorithme de Dijkstra

1 Graphes pondérés et matrice d'adjacence



Graphe 1 : pondéré, non orienté



Graphe 2 : non pondéré, orienté

Quelques définitions supplémentaires :

- A chaque arête on peut affecter un nombre réel appelé **poids** : dans ce cas le graphe est dit **valué** ou **pondéré** (un graphe non pondéré peut être considéré comme un graphe où tous les poids sont égaux, à 1 par exemple).
- Pour un sommet X donné, les sommets Y pour lesquels il existe une arête (X,Y) sont dits **adjacents** à X et sont appelés les **voisins** de X.
- Un graphe pondéré peut aussi être représenté par une **matrice d'adjacence** : en ordonnant les sommets de 0 à $n-1$, le terme de place $(i;j)$ de la matrice est **le poids de l'arête** entre le sommet numéro i et le sommet numéro j (0 s'il n'y a pas d'arête entre les deux).

  Ex. Les matrices d'adjacence des graphes 1 et 2 sont respectivement :

$$\begin{pmatrix} 0 & 3 & 5 & 0 & 0 & 0 & 0 \\ 3 & 0 & 0 & 2 & 5 & 0 & 0 \\ 5 & 0 & 0 & 3 & 0 & 5 & 0 \\ 0 & 2 & 3 & 0 & 2 & 0 & 6 \\ 0 & 5 & 0 & 2 & 0 & 0 & 3 \\ 0 & 0 & 5 & 0 & 0 & 0 & 4 \\ 0 & 0 & 0 & 6 & 3 & 4 & 0 \end{pmatrix} \quad \text{et} \quad \begin{pmatrix} 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 & 1 \end{pmatrix}.$$

  G1 : matrice symétrique (gr. non orienté) G2 : matrice non-symétrique.

- Un **chemin** est une succession de sommets, telle qu'il existe une arête entre chaque sommet et le suivant. Par exemple, ABDGE est un chemin du graphe 2, mais BDGF n'en est pas un (il n'y a pas d'arête (G,F))
- Le **poids d'un chemin** est la somme des poids des arêtes entre un sommet du chemin et le suivant. Par exemple : Le chemin ABDG du graphe 1 a pour poids 11 : poids de (A,B) + poids de (B,D) + poids de (D,G).

Question (Préambule).
Récupérez et sauvegardez sous un autre nom le fichier dijkstra.py joint, qui sera à compléter. Il contient un début de programme principal avec les listes de listes `G1` et `G2` définissant les graphes ci-contre par matrices d'adjacence.

Remarque : ici pas besoin de numpy : avec des graphes pondérés les puissances de la matrice d'adjacence n'auront pas d'intérêt.

Question 1 (Dictionnaire des listes d'adjacence).
Récupérez et modifiez la fonction `dic_adjacence` du TP précédent : elle doit renvoyer la liste des voisins de chaque sommet mais aussi le poids de l'arête correspondante, les valeurs seront donc des listes de tuples (sommet, poids de l'arête). Par exemple le dictionnaire d'adjacence du graphe G1 est :

```
{'A': [('B', 3), ('C', 5)], 'B': [('A', 3), ('D', 2),
↪  ('E', 5)], 'C': [('A', 5), ('D', 3), ('F', 5)], 'D':
↪  [('B', 2), ('C', 3), ('E', 2), ('G', 6)], 'E':
↪  [('B', 5), ('D', 2), ('G', 3)], 'F': [('C', 5),
↪  ('G', 4)], 'G': [('D', 6), ('E', 3), ('F', 4)]}
```

Question 2.
Utiliser la fonction précédente pour créer les deux dictionnaires d'adjacences des graphes donnés par les matrices $G1$ et $G2$. On notera $V1$ et $V2$ ces dictionnaires. Puis recopier les lignes suivantes

```
for (v,p) in V1['A']:
        print(v)
        print(p)
```

Ce type de boucle permet de parcourir un dictionnaire en obtenant à chaque fois la clé et la valeur associée.

Question 3 (Adjacence).
Écrire une fonction `adjacence(voisins,S1,S2)` qui pour un sommet `S1` et un sommet `S2` dans un graphe donné par son dictionnaire d'adjacence `voisins` renvoie le poids de l'arête partant de `A` et arrivant à `B` si elle existe et -1 sinon.

Question 4 (Poids d'un chemin).
Écrire une fonction `poids_chemin(voisins,chemin)` qui prend en paramètres un dictionnaire d'adjacence `voisin` et un `chemin`, et qui renvoie le poids du chemin. On pourra renvoyer -1 si le chemin n'existe pas. Un chemin sera donné sous la forme d'une liste des sommets parcourus dans l'ordre, par exemple ['A','B','D','G'].

2 Algorithme de Dijkstra

2.1 Présentation

Il s'agit de déterminer efficacement un chemin de **poids minimal**, dit "plus court chemin" entre un sommet et un autre dans un graphe **pondéré** dont tous les poids sont **positifs**.
Exemple : lorsqu'un logiciel utilise des données GPS (et cartographiques) pour construire un itinéraire, il résout un problème de recherche de plus court chemin sur un graphe dont les sommets sont divers points géographiques (villes, intersection de routes,...) et dont les arêtes représentent les liaisons routières entre ces points. Les poids des arêtes seront les distances, les temps, ou les coûts de ces liaisons, selon que l'on souhaite trouver l'itinéraire le plus court, le plus rapide, ou le plus économique.

L'algorithme de Dijkstra (1959) est un algorithme de recherche d'un plus court chemin entre deux sommets d'un graphe **pondéré**, appelés **E** ('Entrée' : sommet de départ) et **S** ('Sortie' : sommet d'arrivée).

C'est un algorithme **glouton** : il choisit à chaque étape i le chemin **dont le poids depuis E** jusqu'au sommet considéré à l'étape i est **minimal**, ce poids étant éventuellement réactualisé en fonction des sommets explorés.

Algorithme de Dijkstra :

▶ Pour chaque sommet on doit manipuler les valeurs suivantes

- **choisi** : booléen qui indique si le sommet à déjà été visité.
- **poids** qui indique le poids du chemin le plus court, trouvé jusqu'à l'étape actuelle, partant du départ et arrivant au sommet. Si on n'a pas encore trouvé de chemin, on utilisera une valeur infini $+\infty$.
- **pere** le père provisoire du sommet dans le chemin le plus court trouvé à l'étape en court.

▶ Initialisation :

- on marque tous les sommets comme non encore choisis.
- on initialise la valeurs des pères avec une valeur arbitraire (par exemple None ou $\oslash$), indiquant qu'aucun sommet n'a encore de père attribué.
- on affecte le poids 0 au sommet origine et, provisoirement, le poids ∞ aux autres sommets :
- X est le sommet origine E, qu'on marque comme choisi.

▶ **Tant que le sommet de sortie S n'a pas été choisi** :

- **mise à jour** des "poids à partir de E" :
  pour chaque sommet Y **non encore choisi** et **voisin de X**, on calcule son nouveau **poids à partir de E** : c'est la somme **s** du poids de X à partir de E et du poids de l'arête X → Y .
  Si ce nouveau poids **s** est strictement inférieur au poids actuel du sommet Y :
  - ☐ on affecte à Y son nouveau poids provisoire : c'est **s** ('meilleur' que l'ancien) ;
  - ☐ on attribue à Y son 'meilleur' père provisoire : c'est X.

  Sinon... on ne change rien pour ce sommet Y.
- parmi les sommets **non encore choisis**, on choisit un nouveau sommet X de poids **minimal** , on le marque comme choisi, et on recommence.

▶ Résultat : un plus court chemin de E à S s'obtient, en fin d'algorithme, en 'remontant' la chaîne des pères à partir de la fin S, puis en le réécrivant à l'endroit.

Voici une simulation de cet algorithme pour obtenir un plus court chemin de A à G dans le graphe 1 :
(la notation $\oslash$ signifie que le sommet n'a pas encore de père)

INIT	**A(0,$\oslash$)**	B(∞,$\oslash$)	C(∞,$\oslash$)	D(∞,$\oslash$)	E(∞,$\oslash$)	F(∞,$\oslash$)	G(∞,$\oslash$)
mise à jour choix mini	/	B(3,A) **B**	C(5,A)	D(∞,$\oslash$)	E(∞,$\oslash$)	F(∞,$\oslash$)	G(∞,$\oslash$)
mise à jour choix mini	/	/	C(5,A) **C**	D(5,B)	E(8,B)	F(∞,$\oslash$)	G(∞,$\oslash$)
mise à jour choix mini	/	/	/	D(5,B) **D**	E(8,B)	F(10,C)	G(∞,$\oslash$)
mise à jour choix mini	/	/	/	/	E(7,D) **E**	F(10,C)	G(11,D)
mise à jour choix mini	/	/	/	/	/	F(10,C) **F**	G(10,E)
mise à jour choix mini	/	/	/	/	/	/	G(10,E) **G**
FIN : G sélectionné	On remonte depuis G en suivant les prédécesseurs, et on remet à l'endroit : G,E,D,B,A → plus court chemin **ABDEG**						

Question 5 (Un exemple)**.**
À l'aide du tableau ci-dessous, simuler **à la main** cet algorithme pour obtenir un plus court chemin de B à F dans le graphe 1.

INIT	A( )	**B( )**	C( )	D( )	E( )	F( )	G( )
mise à jour choix mini							
mise à jour choix mini							
mise à jour choix mini							
mise à jour choix mini							
mise à jour choix mini							
mise à jour choix mini							
FIN : F sélectionné	plus court chemin :						

2.2 Mise en place des objets, fonctions auxiliaires

1. On cherche un plus court chemin entre un sommet deb et un sommet fin.
2. On travaille avec trois dictionnaires dont les clés sont les sommets :
   - **choisi**, dont les valeurs sont des booléens, qui permet de savoir rapidement si un sommet a déjà été choisi : choisi[s] vaut True si le sommet s a déjà été choisi, False sinon.
     Marquer un sommet s comme choisi consiste donc à affecter la valeur True à choisi[s]. Il est alors très simple de tester si un sommet a été choisi :
     if choisi[s] : ... ou if not choisi[s] : ...
   - **poids** : poids[s] est le poids (provisoire) entre le sommet deb et le sommet s ;
   - **voisins**, passé en paramètre, qui contient les voisins de chaque sommet du graphe (avec le poids de l'arête correspondante).
3. Il faut un nombre jouant le rôle de $+\infty$ on choisit un nombre suffisamment grand par rapport aux autres grandeurs en jeu.

Question 6 (Fonctions auxiliaires)**.**
Écrire les deux fonctions suivantes, pour pouvoir ensuite les utiliser dans la fonction principale **dijkstra** :

- une fonction `choix_mini(voisins,choisi,poids)` qui trouve et renvoie un sommet X de **poids minimal**.
- une fonction `mise_a_jour(X,voisins,choisi,poids)` qui, étant donné un sommet X, met à jour (en les modifiant en place) les dictionnaires choisi, poids pour tous les voisins de X non encore choisis. Remarque : les dictionnaires sont *mutables*, cette fonction ne renvoie rien mais modifie "en place" deux des dictionnaires passés en paramètre.
- Définir dans le programme principal la constante `INFINI = 10**12`

2.3 Fonction principale

Question 7 (Début de la fonction principale)**.**
Écrire le début de la fonction principale `dijkstra(voisins,deb,fin)` avec l'initialisation et les constructions des dictionnaires.
Pour cela :

- Initialiser les dictionnaires **choisi, poids**.
- Affecter le poids **0** au sommet deb et marquer comme choisi le sommet deb.

Question 8 (Fin de la fonction principale)**.**
Après avoir tester la version précédente :

1. Compléter finalement la fonction `dijkstra(voisins,deb,fin)` qui renvoie un plus court chemin du sommet deb au sommet fin d'un graphe donné, en suivant l'algorithme.
   Pour cela : Écrire la boucle correspondant à chaque étape répétée dans l'algorithme : tant que le sommet fin n'a pas été choisi, mettre à jour poids, puis choisir le sommet de plus petit poids parmi les non choisis.
2. Tester cela par un appel de la fonction **dijkstra** dans le programme principal, faire afficher le poids du chemin le plus court entre les sommets A et G dans le graphe 1.

2.4 Calcul du chemin le plus court

Pour le moment la fonction ne renvoie que le poids du chemin optimal. Nous allons modifier les fonctions précédentes pour obtenir le chemin.
Nous allons utiliser un nouveau dictionnaires `pere` dont les clés sont les sommets et tel que la valeur de `pere[s]` est, à chaque instant :

- `None` si le sommet `s` n'a pas encore été atteint
- Le sommet qui précède `s` dans le chemin de poids minimal trouvé jusqu'à présent si `s` a été visité

Question 9 (Modification)**.**
Vous devez modifier les fonctions déjà écrites

- Compléter les constructions de dictionnaires dans la fonction `dijkstra` pour construire ce dictionnaire
- Modifier la fonction `mise_a_jour`, elle doit maintenant recevoir en argument le dictionnaire `pere` et le modifier.
- Modifier la fonction principale pour mettre à jour le dictionnaire père

Question 10 (Construction du chemin)**.**
Chaque sommet est maintenant associé à son père optimal. Compléter la fonction principale pour construire le chemin optimal et le renvoyer
Comme on part de la fin et on "remonte" de père en père ; à la fin l'instruction `nom_de_la_liste.reverse()` permettra d'inverser l'ordre des éléments de la liste obtenue.

2.5 Complément

Prévoir une sortie de l'algorithme dans le cas d'un graphe où il n'y aurait aucun chemin reliant l'entrée à la sortie.
Tester sur des chemins 'impossibles' dans le graphe 2.

Aide : cela se produit lorsque, la sortie n'étant pas encore atteinte, la fonction choix_mini n'arrive pas à sélectionner un chemin de plus petit poids : tous les sommets non-choisis ont un poids infini, indiquant qu'aucun chemin depuis l'entrée n'a pu les atteindre. Dans ce cas, faire renvoyer à la fonction choix_mini la valeur None par exemple, et traiter ce cas dans la fonction dijkstra : on peut alors renvoyer une liste vide.