Plus courts chemins dans un graphe pondéré

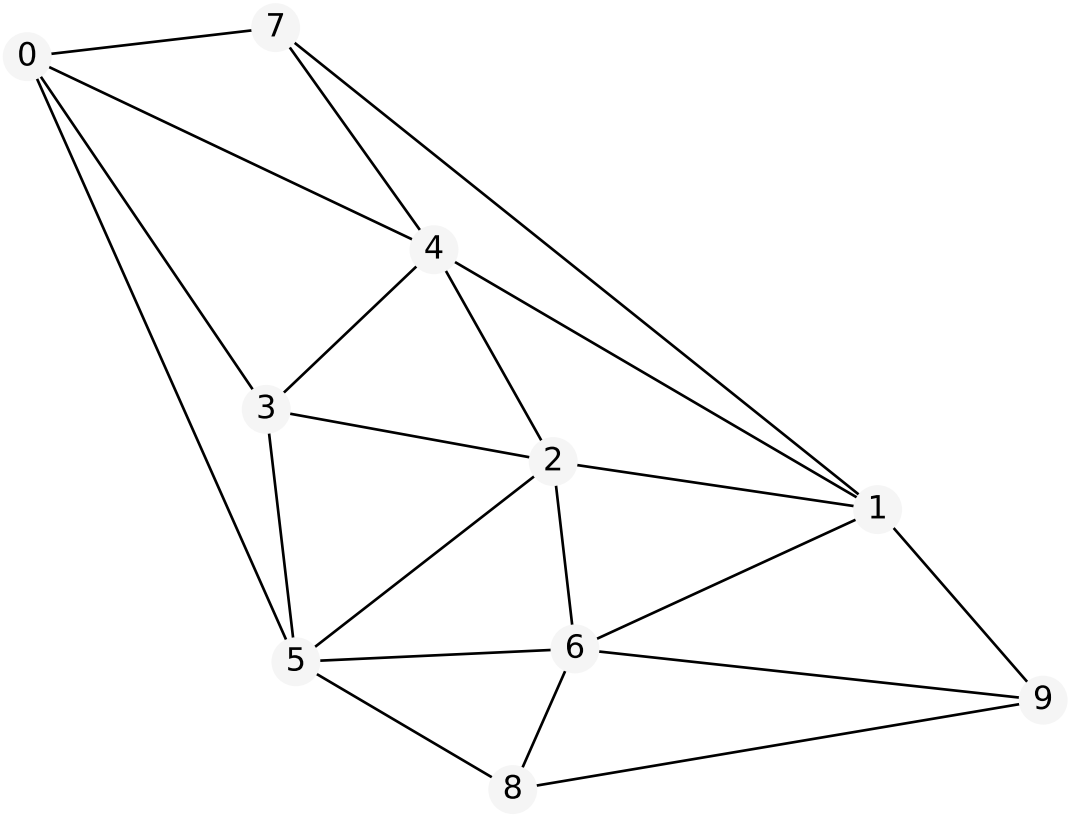
Comme d'habitude, on lance spyder. On commencera un nouveau script qu'on pourra nommer `TP13.py` et qu'on sauvegardera dans son répertoire personnel. Téléchargez depuis `https://cahier-de-prepa.fr/pcsi-bdb/docs?rep=90` le script `exemples.py` que vous enregistrerez dans le même répertoire que `TP13.py`, et on importera les exemples de celui-ci en ajoutant :

`from exemples.py import *` à votre script.

On s'intéresse dans ce TP à des graphes dits « pondérés », c'est-à-dire pour lesquels chaque arête est munie d'un poids (qu'on peut dans certains cas interpréter comme une distance, ou le coût du parcours de celle-ci). Tous ces poids seront supposés positifs, et les graphes considérés pourront être orientés ou non, même si les exemples étudiés dans ce TP ne le sont pas. L'objectif sera de parcourir un graphe depuis un sommet initial, ou bien pour connaître le plus court chemin depuis celui-ci vers tous les sommets accessibles, ou juste le plus court chemin depuis un sommet initial jusqu'à un sommet final.

L'algorithme de Dijkstra donne une réponse élémentaire, et optimale, au premier des deux problèmes. S'il permet également de donner une réponse au second, pour un graphe de grande taille, on lui préfère souvent l'algorithme A* qui en donne une solution approchée, de manière plus efficace.

Voici l'un des graphes (c'est celui nommé `G1`) sur lesquels nous testerons nos algorithmes :



Les poids des arêtes n'apparaissent pas ici, mais dans les graphes sur lesquels nous travaillerons, nous considèrerons les positions des sommets fixées, et les poids de chaque arête seront les distances (euclidiennes) entre les deux sommets qu'elles rejoignent. De ce fait, on devine sur l'exemple précédent qu'un plus petit chemin du sommet 7 au sommet 8 est donné par `[7, 4, 2, 6, 8]` formé de 4 arêtes, et non par exemple `[7, 1, 9, 8]` qui n'est constitué que de 3 arêtes.

Les graphes proposés en exemples (issus du fichier `exemples.py`) sont `G1`,`G2` et `G3`. Avec `len(G*)` vous obtenez le nombre de sommets du graphe, `G*.adj` est une liste d'ajacence (une liste de listes) : `G*.adj[i]` est donc une liste de la forme `[(s1, p1), (s2, p2), ..., (sk, pk)]` indiquant que `i` a `k` voisins, et que l'arête reliant `i` à `s1` a un poids de `p1` et ainsi de suite.

Ce ne sera pas utile dans un premier temps, mais avec `G*.pos` vous avez un dictionnaire où à chaque sommet est associé sa position sous la forme d'un couple `(x, y)` de flottants (ce dictionnaire pourrait nous permettre, si cela était nécessaire, de reconstituer les poids de chaque arête puisque dans le cas présent, le poids d'une arête entre les sommets `i` et `j` est la distance entre donc `G*.pos[i]` et `G*.pos[j]`.

1 Algorithme de Dijkstra

Voici une présentation possible de l'algorithme de Dijkstra, adaptée à une implémentation élémentaire (mais pas de complexité optimale) en python :

```
En entrée : un graphe G donné par une liste d'adjacence, un sommet de départ s_depart
En sortie : une liste dis des distances de s_depart aux autres sommets,
et la liste pere où pour tout sommet s, pere[s] donne le sommet depuis lequel s est atteint
   On initialise dist telle que pour tout s autre que s_depart, dist[s] vaut +oo
   et pere tel que pour tout s, pere[s] = None
   et on introduit une liste aTraiter initialisée à [s_depart] et une dernière traités

   Tant que la liste aTraiter n'est pas vide :
       Déterminer dans aTraiter le sommet s1 le plus proche de s_depart et l'extraire
       Ajouter s1 à traités
       Pour tout voisin s2 de s1 et non déjà dans traités:
           si dist[s2] = +oo alors
               ajouter s2 à aTraiter
           si dist[s1] + poids(s1, s2) < dist[s2]:
               dis[s2] <- dis[s1] + poids(s1, s2)
               pere[s2] <- s1
```

Question 1. Ecrire une première fonction d'entête `def plusProcheSommet(dist, aTraiter)` qui renvoie parmi les sommets de la liste `aTraiter` celui qui minimise `dist`.

Question 2. Ecrire alors une fonction d'entête `dijkstra(G, s_depart)` qui réalise le travail présenté ci-dessus et qui renvoie le couple `(dist, pere)`. (Pour retirer un élément d'indice `i` d'une liste, voir l'annexe)

Question 3. Ecrire une fonction d'entête `def chemin(G, s1, s2)` qui renvoie le couple `(distance, L)` formée de la distance de `s1` à `s2` et une liste de la forme `[s1,...,s2]` d'un chemin minimal de `s1` à `s2`. S'il n'existe pas de chemin de `s1` à `s2` on renverra `None`. Bien sûr, on fera appel à la fonction précédente.

2 Algorithme A*

L'algorithme de Dijkstra est difficile à améliorer (à part l'implementation naïve qui en a été proposée) s'il s'agit de déterminer les chemins minimaux d'un sommet donné à tous les sommets d'un graphe, mais si par exemple on ne cherche qu'un chemin aussi optimal que possible d'un point A à un point B, on lui préfèrera une approche qui, si possible, nous évitera de parcourir l'intégralité du graphe. Il ne paraît pas rentable par exemple sur un trajet routier de Paris à Marseille d'étudier tous les trajets optimaux de Paris aux villes du nord de la France, voire de la Bretagne, or c'est pourtant ce que nous inviterait à faire l'algorithme de Dijkstra, puisqu'on étendrait notre parcours de manière concentrique autour de Paris, jusqu'à atteindre enfin Marseille.

L'idée de l'algorithme A* est alors de rajouter une heuristique qui permettra d'orienter notre recherche, afin de parvenir plus rapidement à notre but. Cette heuristique s'appuie sur une estimation de la distance attendue entre deux sommets du graphe, qui vient compléter les choix locaux liés aux poids de chacune des arêtes de celui-ci. Pour le problème posé, connaissant les positions des sommets dans le plan, on prendra comme heuristique la distance euclidienne entre deux sommets, qui certes sous-estime la distance réelle à parcourir, mais est tout de même un bon point de départ.

Si l'algorithme A* reprend les grandes idées de l'algorithme de Dijkstra, il y a quelques adaptations à faire :

- Pour déterminer le prochain sommet à traiter, on tient bien sûr compte et des distances calculées et de l'heuristique qui s'ajoute à celles-ci (l'heuristique sera pour nous la distance à vol d'oiseau entre le sommet étudié et le sommet final, mais on pourrait aussi ajouter un coefficient multiplicatif à celle-ci)

- Contrairement à l'algorithme de Dijkstra, lorsqu'on aborde un nouveau sommet, on n'est pas certain que la distance calculée depuis le sommet initial est bien la plus petite, à cause de l'heuristique. De ce fait, on ne peut s'interdire de le parcourir à nouveau.

- Ici bien sûr, on arrête le parcours du graphe dès lors que le sommet final est atteint, et on reconstruira alors le chemin trouvé.

L'algorithme devient, où `h` est notre heuristique (en pratique `h[s]` mesurera la distance à vol d'oiseau entre `s` et le sommet final, à un coefficient multiplicatif près) :

```
En entrée : un graphe G, un sommet de départ s_dep et d'arrivée s_fin
En sortie : un couple (distance, chemin) formé d'un chemin de s_dep à s_fin et de la longueur (somme
des poids) de celui-ci.

    On initialise dist telle que pour tout s autre que s_dep, dist[s] vaut +oo
    et pere tel que pere[s] vaut None pour tout s.
    On initialise une liste aTraiter avec [s].

    Tant que aTraiter n'est pas vide:
        On détermine parmi ses éléments celui s qui minimise dist[s] + h[s]
        Si s est le noeud final:
            Reconstruire le chemin depuis s_dep et s'arrêter.
        Pour chaque voisin t de s:
            si dist[t] > dist[s] + poids(s, t):
                dist[t] = dist[s] + poids(s, t)
                pere[t] = s
                if dist n'est pas dans aTraiter:
                     l'ajouter !
    Si on arrive ici, c'est que le sommet final n'est pas atteignable !
```

Question 4. On rappelle que pour les graphes `G*` proposés en exemple, on accède avec `G*.pos[i]` à la position (sous la forme d'un couple de flottants) du sommet `i`. En déduire alors une fonction d'entête `def distance(G, i, j)` qui renvoie la distance entre les sommets `i` et `j` de `G`.

Question 5. Ecrire alors une fonction d'entête `def aStar(G, s_dep, s_fin)` qui prend pour arguments un graphe `G`, un sommet initial `s_dep` et final `s_fin`, et qui renvoie le couple `(distance, chemin)` si un chemin existe, et `None` sinon.

3 Comparaison de Dijkstra et A*

Sur les exemples proposés, je ne pense pas qu'il existe d'exemple de chemin à rechercher pour lequel l'algorithme A* ne renverrait pas le chemin de plus courte distance. L'algorithme A* est, on le devine néanmoins, plus efficace au sens où, si notre heuristique est bonne, la partie du graphe parcouru sera bien moindre.

Question 6. En reprenant les fonctions précédentes, écrire une fonction d'entête `def cheminDijkstra(G, s_dep, s_fin)` qui prend pour arguments un graphe `G`, un sommet initial `s_dep` et final `s_fin` et qui renvoie, s'il existe un chemin au moins, le couple `(chemin, p)` formé d'un chemin minimal de `s_dep` à `s_fin` et du nombre `p` de sommets parcourus. Bien sûr, on reprend ici le code de la fonction Dijkstra pour s'interrompre dès que le sommet final est atteint.

Ajouter de même un compteur dans la fonction `astar` afin de comparer le nombre de sommets parcourus.

Un exemple de ce que j'ai obtenu :

```
>>> cheminDijkstra(G3, 8, 86)
([8, 55, 57, 15, 70, 56, 77, 25, 86], 82)
>>> astar(G3, 8, 86)
([8, 55, 57, 15, 70, 56, 77, 25, 86], 12)
```

4 Quelques pistes pour améliorer la complexité de nos implémentations

- Tester l'appartenance d'un objet à une liste est une opération qui prend du temps, a priori proportionnel à la longueur de celle-ci, puisque sans hypothèse sur celle-ci, on est bien obligé d'en passer en revue tous les termes.

  Proposer d'autres structures qu'une liste permettant de tester plus efficacement l'appartenance d'un objet à celles-ci. On peut aussi tenir compte de ce que les sommets sont numérotés à partir de 0 pour proposer un autre moyen de tester si un sommet a ou non déjà été parcouru.

- Lorsqu'on souhaite trouver le plus petit ou le plus grand élément d'une liste (ce qu'on a besoin de faire, aussi bien pour Dijkstra que pour A*, à chaque étape pour choisir le sommet optimal), là aussi l'algorithme naïf conduit à une complexité linéaire, puisque nous oblige à passer en revue tous les termes de cette liste.

  Faire en sorte que notre liste soit à tout moment triée n'est pas vraiment une solution car c'est au moment de l'insertion d'un nouvel élément que l'on retrouverait une opération agissant en temps linéaire. Une solution est d'utiliser un tas qui permet d'extraire (et retirer) l'élément optimal souhaité avec une complexité de `log(n)` où `n` est le nombre d'éléments stockés, ainsi que d'ajouter un nouvel élément avec la même complexité de `log(n)`.

Pour les plus curieux, vous pourrez aller voir la page Tas(informatique) de wikipédia. A noter que le module `heapq` de python implémente la structure de tas.

(Une remarque : utiliser un tas pour l'algorithme de Dijkstra en réduit significativement la complexité, mais seulement pour les arbres pas trop denses, c'est-à-dire pour ceux dont le nombre d'arêtes est plus proche de n que de n^2 (un graphe complet de n sommets sans arêtes qui bouclent sur un sommet comporte exactement $\frac{n\,(n-1)}{2}$ arêtes.) Les graphes proposés en exemple rentrent bien dans cette catégorie, car sont planaires, or on peut établir qu'un graphe planaire de n sommets ne peut compter, au plus, que $3\,n-6$ arêtes.

5 Annexe : quelques éléments de syntaxe utiles

- Pour définir $+\infty$ en tant que nombre en virgule flottante, on peut utiliser `float('inf')`. On obtient alors un nombre en virgule flottante qui est plus grand que tout autre et qui joue donc bien ce rôle de nombre infini.

- On rappelle qu'un moyen commode de retourner une liste (plus exactement d'obtenir à partir d'une liste la liste de ses termes en sens inverse) est d'utiliser un slice : `L[::-1]` renvoie ainsi la liste formée des mêmes éléments que `L`, mais rangés en sens inverse.

- Aussi bien pour Dijkstra que pour A*, il est un moment où on se trouve confronté au problème suivant : retirer un élément d'une liste qui ne se trouve pas à la fin de celle-ci, mais à un indice `i`. Une manière pythonesque serait d'utiliser la syntaxe `L.pop(i)`, mais d'une part vous n'êtes pas supposés connaître cette utilisation de `pop`, d'autre part cette opération présente un coût non négligeable, car proportionnel au nombre de termes de `L` (alors que retirer le dernier terme de `L` s'effectue en temps constant).

  L'ordre des éléments de notre liste étant de peu d'importance ici, une meilleure méthode consiste à échanger le terme d'indice `i` de `L` avec son dernier terme, et alors seulement en extraire le dernier terme avec la commande ordinaire `L.pop()`.