Dijkstra prend le métro

Le but de ce TD est d'appliquer l'algorithme de Dijkstra sur le métro parisien.

On allégera le graphe du métro, mais on profitera de ce que fait l'algorithme de Dijkstra : les arêtes ont un poids.

> Rappel : une arête est un bout de ligne entre deux stations A et B. Le poids de cette arête pourra être le temps mis par un métro pour aller de A à B. La somme des poids des arêtes (=poids d'un trajet, à minimiser) serait donc le temps total pour le trajet (pour les correspondances, on verra plus tard).

Le plus logique dans un premier temps est d'estimer que le poids de l'arête de A à B est donc la distance euclidienne (mesurée par norme d'un vecteur ou théorème de Pythagore, c'est pareil) entre les deux points A et B (rappelons que notre « base de données » contient les coordonnées des stations : x de 0 à 800 et y de 0 à 600 environ).
Ce sera à vous de calculer (une fois pour toutes) le poids de ces arêtes.

Votre mission

Reprenez le programme **métro.py**. Quand le programme ouvre le fichier texte ratp_voisins.txt pour le transférer dans une liste **L** (liste de listes et **int** et **char**), ne recopiez que ceux dont le champ « arrondissement » contient la chaîne de caractère « Paris ».[1] Vérifiez que ne sont alors plus visibles que les stations intra-muros lors de la création du canevas Can.

Créez ensuite le tableau **LA** des longueurs des arêtes.
Ce tableau sera de taille **n** sur **n** [2]. A priori dans chaque cas **L[i][j]**, il y aura **float(« inf »)** (vous vous souvenez de ce que c'est?).

L'instruction sera donc déjà au choix :
LA = [[float(« inf ») for k in range(n)] for i in range(n)] ou **LA = [[float(« inf »)]*n for i in range(n)]** ou **LA = [float(« inf ») for k in range(n)]*n** ou **LA = [[float(« inf »)]*n]*n**
à vous de vous souvenir lesquelles sont valables, et lesquelles ne le sont pas.

Ensuite, prenez une par une les stations **L[i]** et un par un les voisins **j** (par leur code ou par leur numéro ?) de **L[i]**, calculez la distance euclidienne (avez vous pensé à importer **sqrt** du module **math** ?). Arrondissez à l'entier par int.

> Variante pour les plus informaticiens.
>
> Il y a des lignes qui roulent plus ou moins bien, vous le savez (lemme : c'est toujours celle que vous voulez prendre qui est la plus lente). Multipliez la distance par un facteur de 1 à 5 qui dépend du numéro de la ligne (mais comment savoir quel est le numéro de la ligne qui relie **L[i]** et **L[j]**?).

> Variante pour les plus infographistes.
>
> Arrangez vous pour que le poids de l'arête soit visible sur le canevas **Can**, juste entre les deux stations (il faudra utiliser **Can.create_text(…)** avec pour paramètre la moyenne des coordonnées des stations, et un texte égal à la conversion en string de **LA[i][j]**).

Appliquez l'algorithme de Dijkstra avec pour paramètre une station de départ **Si** et une destination **Sc**.
Les conventions de « couleur » des listes sont des conventions habituelles et peuvent servir de noms pour les listes justement (gris,noir, père).

Vous en avez des versions clefs en main si vous voulez (version sur le module **graphe.py** par exemple, où vous pouvez enlever ce qui sert juste à visualiser) sinon, je vous redonne la version pseudo code :

---

1 Les stations hors de Paris sont placées « pour que ça tienne sur le plan », leurs coordonnées sont donc erronées, et puis on en a assez comme ça.

2 n étant le nouveau nombre de stations, calculé automatiquement lors de la création de la liste L, sinon c'est len(L)

Algorithme de Dijkstra

On se donne un sommet initial Si, et un sommet cible Sc.

- Colorier Si en gris //*(à explorer)*, et tous les autres sommets en blanc //*(non explorés)*.
  Mettre Si dans une file de priorité initialement vide, avec la distance d(S)=0
- Tant qu'il reste des sommets gris :
  - Choisir un sommet gris S de priorité minimale.
  - Le colorier en noir //exploré
  - Si c'est Sc, l'algorithme se termine par un succès.
  - Pour chaque voisin S′ de S
    - Si S′ est blanc, le colorier en gris, l'ajouter à la file de priorité avec une distance d(S′)=d(S)+LA(SS′), où LA(SS′) désigne la longueur de l'arête reliant S à S′.
      Poser père(S′)=S. // *On a trouvé un chemin de Si à S′, passant par S*.
    - Si S′ est gris, comparer d(S′) et d(S)+LA(SS′). Si d(S)+LA(SS′)<d(S′), poser d(S′)=d(S)+LA(SS′) et père(S′)=S. //*On avait déjà trouvé un chemin de Si à S′, mais on vient d'en trouver un plus court passant par S*.
- L'algorithme se termine par un échec (*Sc n'a pas été trouvé*).

Appliquez l'algorithme à des stations classiques (de Saint-Paul à chez vous par exemple).

Vérifiez sur le plan sa cohérence.

Variante pour informaticien plus poussé : que se passe-t-il si on ferme totalement la ligne 1 par exemple (comment modifier **LA** ?).

Algorithme A*

L'algorithme de Dijkstra n'est pas forcément la meilleure idée pour le graphe du métro. En effet, partant par exemple de Saint-Paul, à destination de Porte d'Orléans, il teste tout « en rond à partir de Saint-Paul », en s'en éloignant de plus en plus. Mais il va finir par arriver à République et même à Père-Lachaise, alors qu'il vaudrait mieux qu'il se dirige « globalement » vers le Sud.

On va donc l'aider par l'algorithme A*[3], qui dispose d'une heuristique[4] qui favorise des choix plutôt que d'autres. Cet algorithme est totalement approprié pour les jeux vidéo, les explorations de labyrinthe.

https://fr.wikipedia.org/wiki/Algorithme_A*

https://www.youtube.com/watch?v=19h1g22hby8

https://www.youtube.com/watch?v=X3x7BlLgS-4

3 A étoile, ou A star, c'est son petit nom.

4 En informatique, une heuristique est ce qu'on peut appeler une intuition, une tendance, un « mon petit doigt m'a dit regarde plutôt vers le Sud ».

Algorithme A star.

On se donne un sommet initial Si, et un sommet cible Sc.

- Colorier Si en gris //*(à explorer)*, et tous les autres sommets en blanc //*(non explorés)*. Mettre Si dans une file de priorité initialement vide, avec la distance d(S)=0
- Tant qu'il reste des sommets gris :
    - Choisir un sommet gris S de priorité minimale.
    - Le colorier en noir (*exploré*)
    - Si c'est Sc, l'algorithme se termine par un succès.
    - Pour chaque voisin S′ de S
        - Si S′ est blanc, le colorier en gris, poser d(S′)=d(S)+LA(SS′) et ajouter S′ à la file de priorité avec une priorité d(S′)+dC(S′), où LA(SS′) désigne la longueur de l'arête reliant S à S′, et dC(S′) est la distance euclidienne entre S′ et Sc . Poser père(S′)=S. // *On a trouvé un chemin de Si à S′, passant par S.*
        - Si S′ est gris, comparer d(S′) et d(S)+l(SS′). Si d(S)+l(SS′)<d(S′), poser d(S′)=d(S)+l(SS′), mettre à jour la priorité de S′ dans la file (à la valeur d(S′) +dC(S′)) et père(S′)=S. //*On avait déjà trouvé un chemin de Si à S′, mais on vient d'en trouver un plus court passant par S.*
- L'algorithme se termine par un échec (*Sc n'a pas été trouvé*).

Comparez l’algorithme de Dijkstra et l’algorithme A* pour des trajets simples.

Sur le site de Mickael Pechaud (prof d’info C.P.G.E. à Montpellier) vous pouvez simuler Dijkstra et A* sur le même graphe (ouvrez deux fenêtres en simultané).

http://mpechaud.fr/scripts/parcours/index.html