Informatique — MPSI/PCSI

TP n° 13 – Graphes - Partie 3/3

CONSIGNE IMPORTANTE :

— Pour ce TP, l'IDE SPYDER sera IMPERATIVEMENT utilisé.

1 Implémentation d'un graphe pondéré

Lors des deux TP précédents, nous avons représenté un graphe non pondéré par un dictionnaire dont les clefs étaient les sommets du graphe, la valeur de chaque clef étant à son tour un dictionnaire dont la clef `'adj'` contenait la liste des sommets adjacents.

Pour manipuler un graphe pondéré, il faut maintenant préciser pour chaque sommet non seulement l'ensemble des sommets adjacents, mais pour chacun de ces sommets le poids de l'arête associée.

On modifie donc la structure de donnée précédente de façon à ce que la clef `'adj'` associée à un sommet `A` ait pour valeur un dictionnaire dont les clefs sont les sommets adjacents à `A` et la valeur associée à chaque clef le poids de l'arête correspondante.

Si on considère par exemple le graphe de la figure 1 ci-dessous.

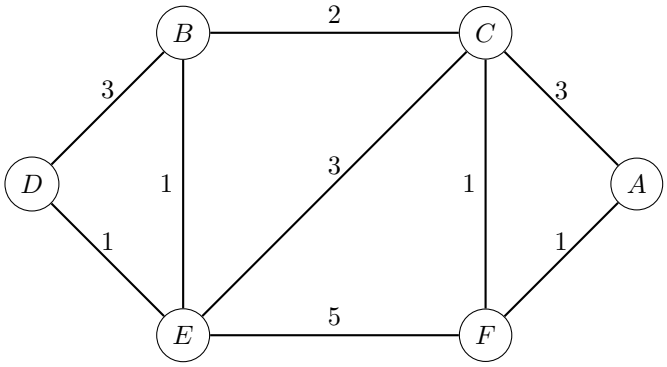

FIGURE 1 – graphe G_1

La clef associée au sommet `A` est initialement `'A': {'adj': {'C': 3, 'F': 1}}`.

Récupérez sur le site cahier de prépa le fichier `info-TP13-cadeau.py`, qui contient une implémentation de ce graphe exemple ainsi que d'autres graphes et fonctions utiles pour la suite.

Ce fichier comprends notamment un certains nombre de fonctions permettant de construire des exemples de graphes :

— la fonction `sommet` qui prend en entrée un graphe G et un sommet S et ajoute à G un sommet isolé S

— la fonction `sommets` qui prend comme argument une séquence de sommets et qui renvoie le dictionnaire initialisé associé

— la fonction `arete` qui prend en entrée un graphe G, deux sommets s et t et un poids de type flottant p et ajoute au graphe une arête de poids p reliant les deux sommets

— la fonction `aretes` qui prend en entrée un graphe G et une liste d'arêtes A, chaque arête étant codée par un triplet s, t, p où s et t sont des sommets et p un poids de type flottant, et ajoute toute les arêtes de la liste au graphe.

2 Algorithme de Dijkstra

2.1 Rappel et premier exemple

On rappelle le principe de l'algorithme de Dijkstra, vu en cours, qui prend en entrée un graphe pondéré G et un sommet racine r et détermine pour chaque sommet du graphe le chemin le plus court de la racine au sommet.

On initialise le graphe en marquant le sommet racine comme visité et tout les autres sommets comme non visités, et en attribuant à chaque sommet s une étiquette $e(s)$ égale à 0 pour le sommet racine, et à plus l'infini pour tous les autres. Ensuite, tant qu'il reste des sommets non visités, on choisit le sommet non visité s d'étiquette la plus faible possible, on le marque comme visité, et on teste s'il est possible de faire diminuer l'étiquette des voisins de s non encore visité. Lorsque l'algorithme termine, l'étiquette de chaque sommet est sa distance au sommet racine. De plus, si l'on a attribué à chaque sommet son prédécesseur, il est possible de « remonter » le chemin aboutissant à ce sommet jusqu'à la racine.

L'algorithme suppose d'associer à chaque sommet une étiquette e qui vaut en fin d'exécution la distance de ce sommet au sommet racine, ainsi qu'une étiquette "vu" qui vaudra 0 si le sommet n'a pas encore été visité et 1 sinon. Ces étiquettes n'étant pas des caractéristiques intrinsèques du graphe, on utilisera une copie profonde P du graphe G, c'est-à-dire une copie occupant un emplacement différent en mémoire et que l'on peut pourra modifier sans modifier le graphe initial. On peut pour cela utiliser la fonction `copy.deepcopy` du module `copy`.

Pour modéliser l'infini, on peut associer à une variable de type flottant la valeur `float('inf')`.

Exercice 13.1

1. Ecrire une fonction `init_parcours` qui prend en entrée un graphe G et un sommet racine r et renvoie une copie profonde de G initialisée, dans laquelle chaque sommet s'est vu attribuer les étiquettes e et "vu" avec leurs valeurs de départ.
2. Ecrire une fonction `plus_faible` qui prend en entrée un graphe *étiqueté* G et une liste de sommets S et retourne le sommet de S d'étiquette minimum.
3. Ecrire une fonction `dijkstra` qui prend en entrée un graphe G et un sommet racine r et applique l'algorithme de Dijkstra pour calculer la distance à la racine de chaque sommet.

   La fonction devra retourner le graphe étiqueté P, à chaque sommet de P étant associé une clef e contenant la distance au sommet racine, ainsi qu'une clef *parent* indiquant le parent éventuel du sommet dans le plus court chemin trouvé par l'algorithme.

On désire maintenant écrire une fonction qui prend entrée un graphe et deux sommets r et s et retourne le plus court chemin entre les deux sommets. L'algorithme de Dijkstra de la question précédente a pour défaut de calculer le plus court chemin à la racine de tous les sommets du graphe.

Exercice 13.2

Ecrire une fonction `distance` qui implémente une variante de cet algorithme qui s'arrête dès que le sommet s a été visité et retourne la chemin de r à s sous forme d'une liste de sommets.

2.2 Visualisation des parcours

Le fichier `Info-TP13-cadeau.py` contient notamment les deux procédures suivante : une procédure `trace_graphe_adj`, prenant en argument le dictionnaire associé au graphe, qui permet de visualiser le graphe et une procédure `trace_graphe_adj_cc`, prenant en argument le dictionnaire associé au graphe, la couleur de chaque noeud étant la valeur associée à la clé `'coul'` du dictionnaire associé à l'étiquette de chaque sommet, qui permet de faire cette visualisation en couleur. Nous allons utiliser ces deux procédures pour visualiser étape par étape l'exécution des algorithmes de Dijkstra et $A*$.

Exercice 13.3
Ecrire une fonction `distance_col` appliquant l'algorithme de Dijkstra à un graphe G pour déterminer le plus court chemin d'un sommet r à un sommet s en affichant à chaque étape l'état du graphe.

On coloriera en jaune les sommets non encore vus, en bleu les sommets vus mais non encore visités, et en rouge les sommets visités.

3 L'algorithme $A*$

3.1 Rappels et premier exemple

L'algorithme $A*$ est une variante de l'algorithme de Dijkstra dans laquelle certains sommets sont explorés préférentiellement à d'autres afin d'accélérer l'exécution.

On utilise pour cela une heuristique en associant à chaque sommet un poids d'autant plus important qu'on suppose qu'il s'agit d'un choix prometteur. Plus formellement, soit $G = (S, A)$ un graphe muni d'une fonction de poids g, c'est-à-dire d'une fonction définie dans A qui associe à chaque arête $\{s, t\}$ la distance $g(s, t)$. Une heuristique est une fonction $h : S \to \mathbb{R}$, qui associe à chaque sommet v un nombre $h(v)$ (idéalement d'autant plus petit que v est proche de l'arrivée).

Pour trouver le plus petit chemin du sommet r et un sommet s, on applique l'algorithme de Dijkstra à une variante du graphe G dans laquelle on associe à chaque arête $\{s, t\}$ la pseudodistance $h(s, t) = g(s, t) + h(s) - h(t)$.

On peut montrer que si la fonction h est choisie de façon à ce que toutes les pseudo-distances entre deux sommets soit positive, le chemin obtenu est effectivement le plus court chemin entre les deux sommets.

Donnons nous par exemple le graphe de la figure 2, dans lequel toute les arêtes sont de longueur 1. On désire utiliser l'algorithme $A*$ pour trouver le plus petit chemin entre les sommets A_{12} et A_{55}.

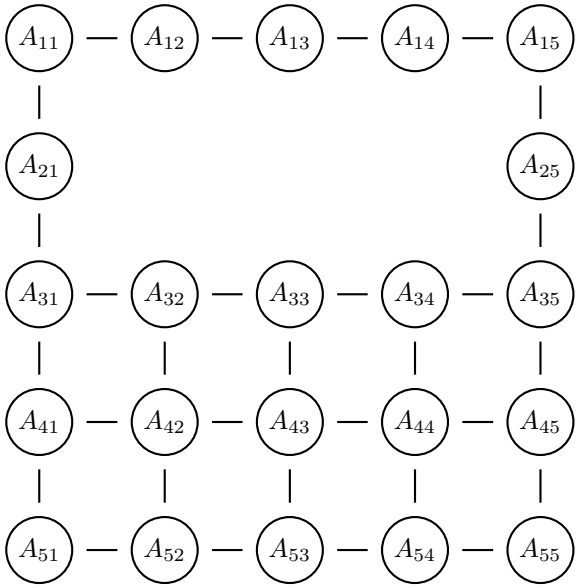

FIGURE 2 – graphe G_2

On utilise pour cela une distance appelée distance de Manhattan. Si (i, j) et (k, l) sont deux couples d'entiers compris entre 1 et 5, la distance de Manhattan entre les sommets A_{ij} et A_{kl} est définie par

$d(A_{ij}, A_{kl}) = |k-i|+|l-j|$. (Cette distance correspond à la distance minimale à parcourir pour aller d'un sommet à l'autre en se déplaçant horizontalement et verticalement le long des arêtes.)

On définit ensuite une heuristique en associent à chaque sommet s sa distance de Manhattan au sommet arrivée, multipliée par un réel C : $h(s) = Cd(s, A_{55})$. L'emploi de cette heuristique avec une constante $C \in]0;1[$ permet d'employer l'algorithme $A*$ en veillant à ce que toutes les pseudodistances restent positives.

Le fichier `info-TP13-cadeau.py` contient un graphe `G2` modélisant le graphe ci-dessus, dans lequel à chaque sommet est associé une clef `XY` contenant le couple d'indices `i,j` associé.

Exercice 13.4

1. Ecrire une fonction `Manhattan` qui prend en entrée un graphe G du type ci-dessus et deux sommet s et t et retourne la distance de Manhattan entre les deux sommets.
2. Ecrire une fonction `heuristique` qui prend en entrée un graphe étiqueté G et une fonction f et associe à chaque sommet du graphe son heuristique $f(s)$.
3. Ecrire une fonction `pseudo_distance` qui prend en entrée un graphe étiqueté G, une fonction f et remplace la distance de chaque arête par la pseudo-distance calculée à l'aide de f.
4. Ecrire une fonction `Astar` qui prend en entrée un graphe G, une fonction f, et deux sommets s et t et détermine le plus court chemin entre ces sommets à l'aide de l'algorithme $A*$.

   On commencera par faire une copie profonde du graphe et par l'initialiser comme pour l'algorithme de Dijkstra, avant de modifier la copie à l'aide de la fonction `pseudo_distance`.

   Tester l'algorithme sur le graphe $G2$ pour l'heuristique définie ci-dessus et vérifier qu'il donne le plus court chemin entre les sommets A_{12} et A_{55}.
5. Ecrire une fonction `Astar_col` appliquant l'algorithme $A*$ à un graphe G pour déterminer le plus court chemin d'un somme r à un sommet s en affichant à chaque étape l'état du graphe.
6. Appliquer ces deux fonctions au graphe $G2$ pour déterminer le chemin le plus court de $A21$ à $A55$ en utilisant différentes valeur de la constante C pour l'algorithme $A*$ et comparer les résultats.

3.2 Graphe Routier

Le fichier `reseauisere.csv` est un ficher au format `csv` décrivant les principaux axes routiers du département de l'Isère[1]. Pour chaque axe est précisé notamment :

— les coordonnées géographiques (latitude et longitudes) d'une borne réelle ou fictive (désignée sous le nom de "point routier") de départ
— une liste des couples de coordonnées des points routiers situés le long de l'axe
— l'abscisse cumulée du point de dépar et du point d'arrivée depuis un point de référence (colonnes F et G)
— les références du point de départ et du point d'arrivée (colonnes N et O).

Vous pouvez ouvrir et parcourir le fichier au moyen d'un tableur, par exemple LibreOfficeCalc.

Le fichier `graphe_routier.py` permet de générer le graphe Gis associé au réseau routier de l'Isère à partir de ce fichier, à chaque borne étant associé un sommet du graphe portant un nom du type $Bnum$ où num est le numéro de la borne. A chaque sommet est associé notamment une clef XY dont le contenu est le couple de coordonnées du point correspondant.

Le but de cet exercice est d'appliquer l'algorithme de Dijkstra pour trouver le chemin le plus court entre deux noeuds routier, puis l'algorithme $A*$ en utilisant comme heuristique la distance euclidienne entre deux sommets.

Exercice 13.5

1. Ouvrir le fichier `graphe_routier.py`, modifier le chemin d'accès au répertoire passé en argument à `os.chdir` pour s'adapter à votre répertoire courant et générez le graphe Gis.
2. Visualiser la graphe à l'aide de la fonction `trace_graphe_adj`. Pour plus de lisibilité, on peut utiliser les paramètres `d` et `etiquettes` pour modifier la taille des noeuds et supprimer éventuellement les étiquettes.

1. Source :https://opendata.isere.fr

On cherche maintenant à déterminer le plus court chemin entre les sommets $B1222$ et $B363$. Pour cela, on peut utiliser l'algorithme de Dijkstra, ou l'algorithme $A\star$ en utilisant comme heuristique la distance entre deux bornes.

Pour cela, il est nécessaire, connaissant la latitude et la longitude de deux points, de déterminer la longueur de l'arc de cercle joignant les deux (voir figure 3). On utilise le résultat mathématiques suivant : si A et B sont deux points de la sphère terrestre de latitudes ϕ_A et ϕ_B et de longitudes λ_A et λ_B, et $d\lambda$ désigne la différence $\lambda_B - \lambda_A$, la distance angulaire en radians $S_{A,B}$ entre A et B est donnée par la relation fondamentale de trigonométrie sphérique

$$S_{A,B} = \arccos(\sin\phi_A \sin\phi_B + \cos\phi_A \cos\phi_B \cos d\lambda)$$

On obtient ensuite la longueur de l'arc entre les deux point en multipliant cette distance angulaire par le rayon de la terre, soit environ 6 378 137 mètres [2].

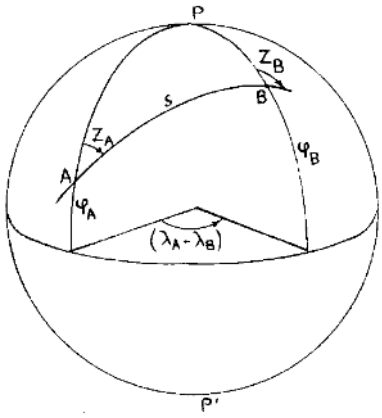

FIGURE 3 – Distance angulaire entre deux points sur une sphère

Exercice 13.6

1. Ecrire une fonction `euclidienne` prenant en paramètres un graphe G et deux sommets s et t et calculant la distance euclidienne entre les sommets s et t.
2. Déterminer le plus court chemin entre les sommets $B1222$ et $B363$ du graphe correspondant au réseau routier de l'Isère avec la fonction `distance`.
3. Déterminer le plus court chemin avec l'algorithme $A*$ en utilisant comme heuristique la distance euclidienne.
4. Faire afficher l'état final du graphe dans chacun des deux cas à l'aide des fonctions `distance_col` et `Astar_col`. Le nombres d'étapes étant important, on pourra modifier ces fonctions afin de ne pas faire afficher toutes celles-ci.

* *
*

2. Source : https://geodesie.ign.fr/contenu/fichiers/Distance_longitude_latitude.pdf