Informatique — MPSI/PCSI, 2023–2024

TP n° 13 – Graphes - Partie 3/3

Objectifs :

— Implémenter et manipuler les graphes pondérés.
— Implémenter les algorithmes de Dijkstra et $A*$.

Rappel :

— Enregistrez dès le début du TP votre travail avec un nom de la forme `VOTRENOM_TPX.py` (évitez les accents et caractères spéciaux).

CONSIGNE IMPORTANTE :

— Pour ce TP, l'IDE SPYDER sera IMPÉRATIVEMENT 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'arrête correspondante.

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

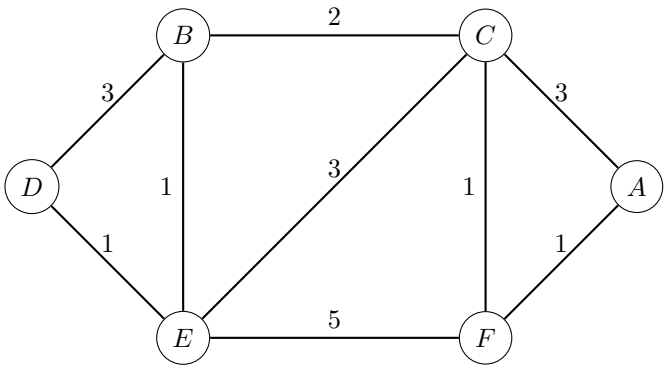

FIGURE 1 – graphe G_1

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

On rappelle 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 note g la fonction de poids qui associe à tout couple de sommet s et t relié par une arête le poids $g(s,t)$ de celle-ci (on supposera que tous les poids sont positifs). On associe à chaque sommet s une étiquette $e(s)$ initialisée à 0 pour le sommet racine et ∞ pour tous les autres.

```
Dijkstra(G,s):
  Initialiser les étiquettes (à 0 ou infini suivant les sommets)
  Tant qu'il reste des sommets non visités:
    Choisir un sommet non visité u d'étiquette la plus faible
    Marquer u comme visité
    Pour tout sommet v voisin de u et non visité:
      Si e(u) + g(u,v) < e(v):
        e(v) = e(u) + g(u,v)
```

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 vu 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. Écrire 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. Écrire 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. Écrire 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.
4. 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.

   Écrire 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.

3 L'algorithme $A*$

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 $h(s, t) = g(s, t) + C(h(s) - h(t))$, où C est un réel strictement positif fixé.

On peut montrer que si le réel C est choisi 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}.

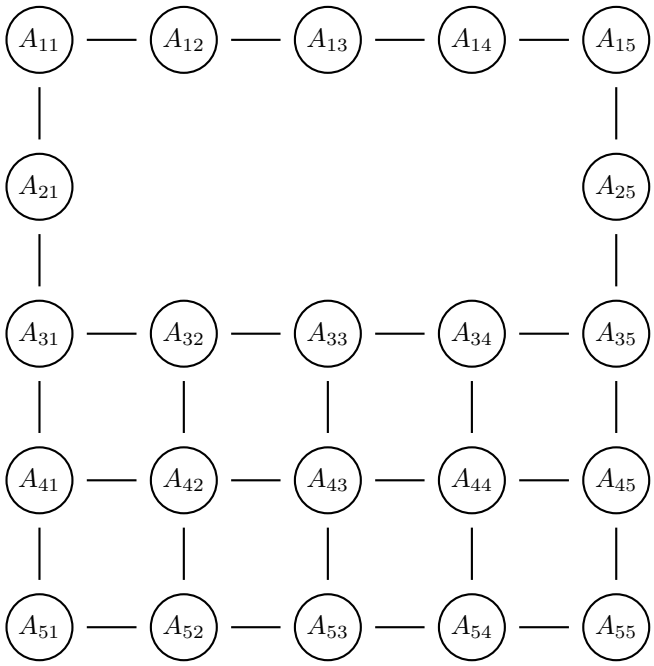

FIGURE 2 – graphe G_2

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 $d(A_{ij}, A_{kl}) = \lfloor k - i \rfloor + \lfloor l - j \rfloor$. (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 : $h(s) = d(s, A_{55})$. 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 13.2

1. Écrire 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. Écrire une fonction `heuristique` qui prend en entrée un graphe étiqueté `G` et une fonction `f` et associe à chaque sommet du graphe `G` son heuristique `f(s)`.

3. Écrire une fonction `pseudo_distance` qui prend en entrée un graphe étiqueté `G`, une fonction `f` et un nombre flottant `C` et remplace la distance de chaque arête par la pseudo-distance calculée à l'aide de `f` et `C`.
4. Écrire une fonction `Astar` qui prend en entrée un graphe `G`, une fonction `f`, un nombre flottant `C` et deux sommets `s` et `t` et détermine le plus court chemin entre `s` et `t` 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}.

4 Visualisation des parcours

Le fichier `cadeau.py` contient notamment les deux procédures suivantes.

1. Une procédure `trace_graphe_adj`, prenant en argument le dictionnaire associé au graphe, qui permet de visualiser le graphe.
2. Une procédure `trace_graphe_adj_cc`, prenant en argument le dictionnaire associé au graphe, la couleur de chaque nœud 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

1. Écrire une fonction `dijkstra_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.
2. Écrire 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.
3. 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.

* *
*