Travaux pratiques 11

Graphe (2) - Recherche d'un plus court chemin

Informatique tronc commun MPSI et PCSI

Rappel : Utilisation de *deque*

On rappelle qu'il est plus efficace pour implémenter les piles ou files d'utiliser les *deque* fournis par le module collections. Il faudra alors précharger cette méthode en préambule du script.

```
from collections import deque
```

I Algorithme du plus court chemin sur un graphe orienté et non pondéré

On considère le graphe orienté et non pondéré de la figure 1. Si on considère l'arc qui relie les sommets 0 et 2, nous dirons dans la suite que « 0 est le père de 2 ».

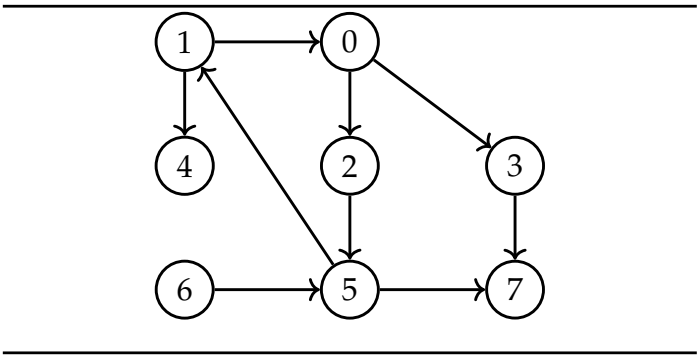

FIGURE 1 –

Exercice 1 - Père
On effectue un parcours en largeur du graphe depuis le sommet 0. On rappelle que cela nécessite de construire une file de sommets à visiter. Dans ce cas, quel sera le premier père du sommet 7 ?

Exercice 2 - Dictionnaire des pères
On représente dans cette partie les graphes orientés par un dictionnaire d'adjacence. Écrire un algorithme `parcours_largeur(dico, debut)` qui prend en argument un dictionnaire d'adjacence d'un graphe orienté et non pondéré et un sommet `debut` et qui renvoie un dictionnaire des pères obtenu par un parcours en largeur. Dans l'exemple de la figure, cela devra donner :

```
dico1 = {0:[2,3],1:[0,4],2:[5],3:[7],4:[],5:[7],6:[5],7:[]}
print(parcours_largeur(dico1, 0))
# renvoie {2: 0, 3: 0, 5: 2, 7: 3}
```

ce qui signifie que le père de 8 est 5.

Principe de l'algorithme du plus court chemin :

Pour obtenir ce chemin [a] entre les sommets `debut` et `fin`, il faut remonter la « filiation » obtenue par le parcours en largeur depuis `fin` jusqu'à `debut`.

a. ou chaîne car le graphe est orienté.

Exercice 3 - Plus court chemin - version itérative
Écrire une fonction `plus_court_chemin(dico, debut, fin)` qui prend en argument un dictionnaire d'adjacence d'un graphe non pondéré et orienté et deux entiers `debut` et `fin` et qui renvoie une liste des sommets correspondant aux plus court chemin entre `debut` et `fin`.
Par exemple, `plus_court_chemin(dico1, 0, 7)` devra renvoyer `[0, 3, 7]` dans le cas du graphe de la figure 1.

Exercice 4 - Plus court chemin - version récursive
Proposer une version récursive de l'algorithme précédent avec les même paramètres.

II Algorithme de Dijkstra sur un graphe avec pondérations positives

On considère le graphe non orienté et pondéré de la figure 2. On souhaite, par exemple, trouver le plus court chemin entre le sommet 0 et le sommet 4. La pondération des arêtes représente la distance entre deux sommets. On représentera ce graphe par la matrice d'adjacence :

$$M_2 = \begin{pmatrix} \infty & 8 & 3 & \infty & \infty \\ 8 & \infty & \infty & \infty & 6 \\ 3 & \infty & \infty & 2 & 9 \\ \infty & \infty & 2 & \infty & 4 \\ \infty & 6 & 9 & 4 & \infty \end{pmatrix}$$

où l'on considère que deux sommets qui ne sont pas reliés par une arête sont séparés par une « distance infinie ». On rappelle que le module `math` fournit un flottant `inf` pour représenter cet infini.

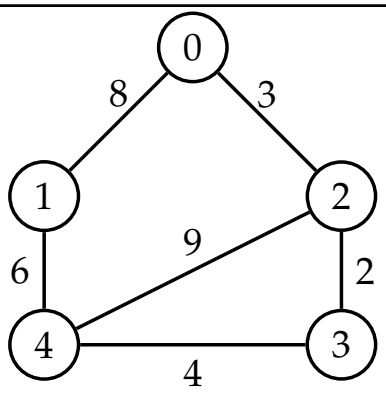

FIGURE 2 –

Rappel sur l'algorithme de Dijkstra :

Soit n le nombre de sommets du graphe. On cherche le chemin de longueur minimale entre deux sommets `depart` et `arrivee`.

- On initialise une liste de taille n appelée `distances` qui représente la distance parcoure entre le sommet de départ et chacun des sommets libres. Cette distance est initialisée à 0 pour le sommet de départ et à `inf` pour les autres sommets.
- On construit une liste `peres` qui permet de connaître le prédécesseur de chaque sommet.
- On notera qu'on ne passe qu'une seule fois par chaque sommet. Il faut donc consigner la liste des sommets disponibles.
- Enfin, on crée une file `a_visiter` initialement composée de `depart`. Tant que la file n'est pas vide et qu'on n'a pas encore atteint `arrivee` :
  - On appelle k_1 le sommet actuel. On commence évidemment par k_1 =`depart`.
  - phase d'actualisation : Parmi les sommets encore disponibles, on vérifie si la route passant par k_1 n'est pas plus rapide qu'une éventuelle route précédemment choisie. On oublie pas d'actualiser alors la distance parcourue jusque k_1 et de consigner que k_1 est le nouveau père des sommets disponibles et atteignables.
  - phase d'exploration : On fait à présent sortir le sommet k_2 le plus proche du sommet actuel k_1.
    - si k_1 et k_2 sont identiques, alors l'arrivée n'est pas atteignable.
    - sinon, on rend k_2 indisponible pour la suite et k_1 prend la valeur de k_2.
- phase de remontée : on remonte la filiation établie précédemment.

Exercice 5 - Sommet le plus proche dans une liste de visite
Écrire une fonction `sortir_min(liste_visite, M, depart)` qui prend en argument une liste d'entiers, une matrice d'adjacence et un entiers `depart` et qui renvoie le sommet le plus proche de `depart`, qui l'enlève de `liste_visite` et qui la renvoie.
Par exemple, dans le cas du graphe de la figure 2, `sortir_min([1,2,3,4], M2, 0)` renvoie `(2, [1, 3, 4])` puisque 2 est le sommet le plus proche de 0 dans dans la liste de visite proposée.

Exercice 6 - Dijkstra
Écrire une fonction `dijkstra(M, depart, arrivee)` qui prend en arguments une matrice d'adjacence et deux sommets `depart` et `arrivee` et qui renvoie le chemin le plus court et la distance totale parcourue selon l'algorithme de Dijkstra.
Par exemple, `dijkstra(M2, 0, 4)` renvoie `([0, 2, 3, 4], 9)`.
Proposer un autre exemple pour tester le cas d'un graphe non connexe et d'un sommet non atteignable.

Question supplémentaire :

Pourquoi peut-on dire que l'algorithme de Dijkstra est un algorithme glouton ?

III Algorithme A* - heuristique

On considère le graphe orienté et pondéré de la figure 3. Celui-ci est orienté et on le représente par la matrice d'adjacence :

$$M_3 = \begin{pmatrix} \infty & 3 & 2 & \infty & \infty & \infty \\ \infty & \infty & 3 & 3 & 2 & \infty \\ \infty & 3 & \infty & 5 & 4 & \infty \\ \infty & \infty & \infty & \infty & \infty & 2 \\ \infty & \infty & 4 & 2 & \infty & \infty \\ \infty & \infty & \infty & 2 & 2 & \infty \end{pmatrix}$$

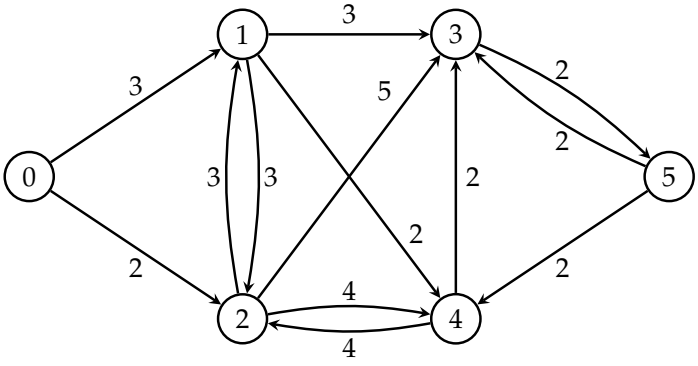

FIGURE 3 – Réseau routier où les routes entre deux sommets sont représentés par des arcs.

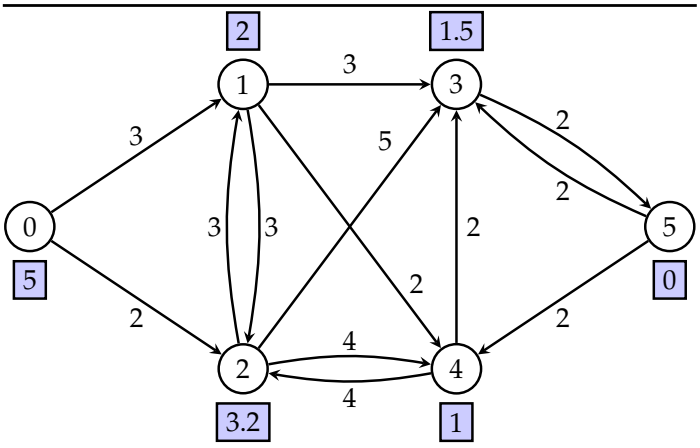

FIGURE 4 – Le même réseau routier. On a rajouté dans les carrés la distance à vol d'oiseau entre un sommet i et le sommet 5.

Exercice 7 - Dijkstra sur un graphe orienté

1. Quel est le plus court chemin entre les sommets 0 et 5 selon l'algorithme de Dijkstra ?
2. Vérifier-le sur le programme python.

Indication :

Une grosse source d'erreur ici est de confondre `M[i][j]` et `M[j][i]`.

Algorithme A* :

L'algorithme de Dijkstra donne une solution optimale mais a un coût d'exploration des sommets disponibles qui peut être très élevée. On rajoute alors une information supplémentaire : ici, la distance à vol d'oiseau entre un sommet i et le sommet 5 (voir figure 4). Notons :

- $h(i)$ la distance à vol d'oiseau entre l'arrivée 5 et le sommet i ;
- $g(i)$ la distance déja parcourue jusqu'au sommet i.

L'algorithme A* est une variante de l'algorithme de Dijkstra où lors de la phase d'exploration, on minimise la somme $g(i) + h(i)$.

Exercice 8 - Heuristique et algorithme A*

1. Expliquer en une phrase simple la notion d'heuristique.
2. Si on représente l'évaluation heuristique par la liste `H = [5, 2, 3.2, 1.5, 1, 0]`, écrire une fonction `aetoile` qui renvoie le plus court chemin.
3. Justifier qu'on obtienne une différence avec Dijkstra dans cet exemple ?
4. Qu'obtient-on si `H = [0, 0, 0, 0, 0, 0]` ?

IV Labyrinthe.

On considère un labyrinthe (voir figure 5) formés de carrés blancs et de carrés gris. Les carrés blancs représentent les zones accessibles et les carrés noirs les zones interdites. On peut lui associer un graphe non orienté (voir figure 5). La pondération n'est pas importante mais on pourra même une pondération 1 si deux sommets sont voisins. Enfin, on peut aussi représenter le plan du labyrinthe en se donnant un repère $(0, \overrightarrow{u_x}, \overrightarrow{u_y})$. Chaque sommet du graphe est alors réprésenté par un point de coordonnées $x = i$ et $y = j$ (voir figure 6).

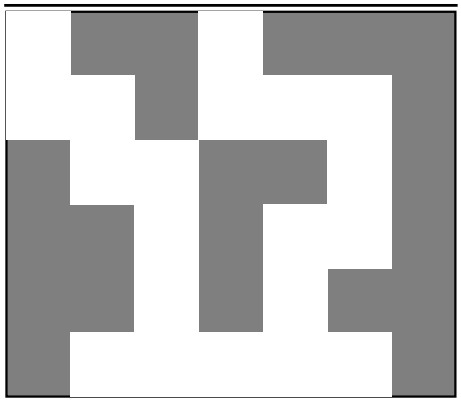

FIGURE 5 – Exemple de labyrinthe. Les parties grises représentent les zones inaccessibles.

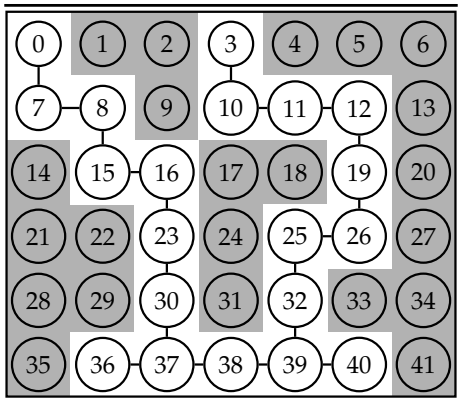

FIGURE 6 – Représentation du même labyrinthe par un graphe.

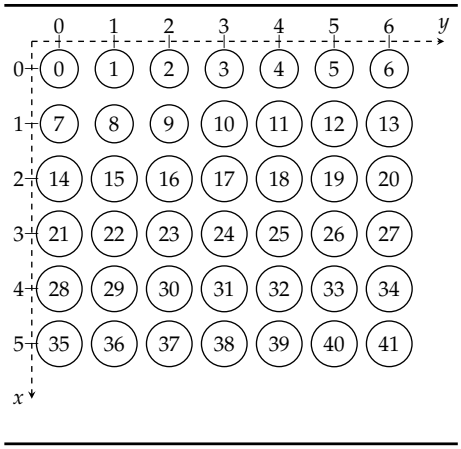

FIGURE 7 – Paramètre du plan du labyrinthe.

Visualiser le labyrinthe :

On peut visualiser le labyrinthe de la figure 5 à l'aide du programme :

```
import numpy as np
import matplotlib.pyplot as plt

L1 = [[1, 0, 0, 1, 0, 0, 0],
      [1, 1, 0, 1, 1, 1, 0],
      [0, 1, 1, 0, 0, 1, 0],
      [0, 0, 1, 0, 1, 1, 0],
      [0, 0, 1, 0, 1, 0, 0],
      [0, 1, 1, 1, 1, 1, 0]]
def affiche_labyrinthe(L):
    M = np.array(L)
    image = plt.imshow(M)
    plt.show()
affiche_labyrinthe(L1)
```

Exercice 9 - conversion en matrice d'adjacence

On note respectivement n et p le nombre de lignes et le nombre de colonnes du labyrinthe.

1. Quel est le nombre total de sommets du graphes ? Comment passe-t-on du numéro du sommet à ces coordonnées ?
2. Écrire une fonction `conversion(L)` qui prend en argument une liste de liste composée de 0 et 1 (comme sur l'exemple) et qui renvoie la matrice d'adjacence associée.
3. Tester sur un exemple simple.

Exercice 10 - Dijkstra sur la labyrinthe

On se donne un départ et une arrivée que l'on représente par leur coordonnées respectives (x_1, y_1) et (x_2, y_2). On souhaite réutiliser la fonction `dijkstra` implémentée précédemment.
Écrire une fonction `labyrinthe(L, x1, y1, x2, y2)` qui prend en argument une liste de liste représentant le labyrinthe et quatre entiers représentant les coordonnées des départ et arrivées et qui renvoie le chemin entre ceux-ci ainsi que la distance parcourue. Tester votre programme par exemple entre le sommet 0 et le sommet 3.

Exercice 11 - Amélioration possible

Parmi les améliorations possibles, on peut :

1. Générer aléatoirement un labyrinthe de grande taille.
2. Ajouter une évaluation heuristique (par exemple, la distance euclidienne qui nous sépare de l'arrivée).
3. introduire des pondérations différentes (représentant des « pièges » dans le labyrinthe).
4. Générer une animation de la résolution pas à pas du labyrinthe (Vous pouvez chercher *maze solver A** sur internet).