TP13.2 - Graphes, parcours

→ *Q.1) Créez un fichier pour sauvegarder votre code. Attention, on se servira de ce code dans le TP suivant.*

A) Chemin dans le plan

Dans certains jeux vidéos, le personnage doit se déplacer d'un point de départ à un point d'arrivée donné par le joueur. Toutefois, il peut y avoir des obstacles sur son chemin, et il faut alors un moyen de trouver un chemin le plus court possible pour aller du point de départ à l'arrivée en évitant les obstacles.
Pour simplifier, on va supposer que le personnage ne peut se déplacer que sur un échiquier rectangulaire, d'une unité vers la droite, la gauche, le haut ou le bas, mais sans sortir de l'échiquier. On peut alors représenter ce rectangle par une matrice (c'est-à-dire une liste de listes) de 0 et de 1 : un 0 correspond à un obstacle et un 1 à une case disponible. Le personnage veut aller de la case en haut à gauche de la matrice (qui est donc forcément un 1) à la case en bas à droite (qui est aussi un 1).

→ *Q.2) Écrire une fonction `visualise_terrain(M)` qui prend une matrice représentant un terrain et affiche une image en noir et blanc : les obstacles sont des pixels noirs, les autres sont blancs.*

A partir d'une matrice de 0 et de 1, on construit un graphe de la façon suivante :

- on créé un dictionnaire vide `G` ;
- on parcourt tous les éléments de la matrice à l'aide d'une double boucle sur les indices `i` et `j` ;
- si une case contient un 0, on ne fait rien ;
- si une case contient un 1, ajoute une clé `(i,j)` à `G` dont la valeur associée est la liste contenant les couples `(k,l)` où `(k,l)` parcourt les indices des cases disponibles qui sont adjacentes à `(i,j)` : il faut faire attention lorsque `(i,j)` est au bord de la matrice !

On obtient ainsi un graphe dans lequel on va pouvoir faire une recherche de plus court chemin (voir paragraphe suivant).

→ *Q.3) Écrire une fonction `taille(M)` qui prend une matrice (supposée non vide) et qui renvoie le couple du nombre de lignes et du nombre de colonnes dans `M`.*

On pourra remarquer dans la suite qui si `M` est une matrice, `n, m = taille(M)` stocke le nombre de lignes de `M` dans `n` et le nombre de colonnes de `M` dans `m`.
De plus, pour accéder à l'élément en position `(i,j)` dans la matrice `M`, on utilise `M[i][j]`.

→ *Q.4) Écrire une fonction `liste_disponibles(M, i, j)` qui prend en paramètres une matrice et deux indices et qui renvoie la liste des couples parmi `(i-1,j)`, `(i+1,j)`, `(i,j-1)` et `(i,j+1)` des cases adjacentes à `(i,j)` qui contiennent un* 1.
Attention à ne pas sortir de `M` !
La liste renvoyée est donc de taille au maximum 4.

→ *Q.5) Écrire une fonction `graphe(M)` qui prend une matrice en paramètre et renvoie le graphe correspondant.*
Vérifier sur une petite matrice que le graphe obtenu est correct.

B) Parcours en largeur

On va maintenant programmer un parcours en largeur qui permet de trouver un chemin de longueur minimale entre deux sommets dans un graphe.
Pour cela on va utiliser le type `deque` codé dans le module `collections` :

- on importe donc `deque` du module `collections`
- pour créer une file `f` vide : `f = deque()`
- pour ajouter un élément `e` à la file `f` : `f.append(e)`
- pour enlever le sommet de la file : `f.popleft()`
- pour récupérer la longueur de la file : `len(f)`

→ *Q.6) Écrire une fonction `jeu(n)` qui prend en paramètre un entier `n` et qui :*

- *créé un file vide `F` ;*
- *répète `n` fois les actions suivantes :*
  - *affiche `F` ;*
  - *choisit un entier au hasard entre 1 et 6 ;*

- *si l'entier n'est pas égal à 1, on l'ajoute à F;*
- *sinon on enlève le sommet de F et on l'affiche;*

— *renvoie F.*

On revient au parcours dans le graphe : on a deux sommets `depart` et `arrivee` du graphe. Voici le plan de l'algorithme :

- on créé un dictionnaire `visites` dont les clés sont les sommets du graphe et toutes les valeurs sont `(np.infty, None)`, sauf celle associée à `depart` qui est `(0, None)` : à la fin de l'algorithme, les valeurs de ce dictionnaire seront des couples `(distance_au_depart, predecesseur)`;
- on créé une file `atraiter` des sommets à traiter qui contient au départ le sommet `depart`;
- tant que la file `atraiter` est non vide :
  - on défile le sommet `sommet` de la file;
  - si ce sommet est `arrivee` on renvoie le chemin utilisé pour l'atteindre;
  - sinon, pour chaque voisin de ce sommet :
    - si le voisin est associé à `(None, np.infty)` dans `visites`, on l'ajoute à la file `atraiter`, et on lui associe le couple `(sommet, dist)` dans `visites`, où `dist` est la distance trouvée entre l'origine et le voisin;

Le résultat de cet algorithme est une liste de sommets de `G` commençant par `depart` et finissant par `arrivee`.

→ *Q.7) Écrire une fonction `init_dico(d, x)` qui prend en paramètres un dictionnaire `d` et un objet `x` et qui renvoie un nouveau dictionnaire ayant les mêmes clés que `d`, toutes associées à `x`.*

Une fois qu'on a atteint le sommet `arrivee` dans l'algorithme principal, il faut lire le dictionnaire `visites` correctement pour renvoyer le chemin voulu. On va écrire une fonction `chemin(d, depart, arrivee)` qui prend en paramètres un dictionnaire dont les clés sont les sommets de `G` et les valeurs associées sont les couples `(distance, predecesseur)`, et deux sommets de `G` et qui renvoie le chemin pour aller de `depart` à `arrivee`. Pour cela :

- on créé une liste `C` contenant le sommet `arrivee`;
- tant que le dernier élément de la liste `C` n'est pas `depart`, on récupère le prédecesseur du dernier élément de `C` et on l'ajoute à la fin de `C`;
- juste avant de la renvoyer, on renverse la liste `C` (avec un slicing bien choisi!).

→ *Q.8) Écrire la fonction `chemin(d, depart, arrivee)`.*

→ *Q.9) Écrire une fonction `parcours(G, depart, arrivee)` qui renvoie un chemin de longueur minimale entre les sommets `depart` et `arrivee` dans le graphe `G` en utilisant toutes les fonctions programmées précédemment.*

On peut maintenant faire la recherche de chemin pour notre personnage.

→ *Q.10) Créer votre matrice de taille `(n, m)` et visualisez la (elle doit avoir un `1` en haut à gauche et en bas à droite). À l'aide des fonctions `graphe` et `parcours`, déterminer le plus court chemin allant de `(0,0)` à `(n-1, m-1)`.*

→ *Q.11) Écrire une fonction `visualise_chemin` qui permet de voir en gris sur le terrain le chemin trouvé.*

C) Ralentisseurs

On va maintenant considérer qu'on peut traverser certains obstacles en prenant un peu plus de temps que sur le terrain normal : on dispose d'une matrice d'entiers naturels

- on a un 0 si l'obstacle est infranchissable;
- pour une case avec un entier $n > 0$, on prend n secondes pour aller sur la case.

On va donc adapter notre graphe pour avoir un graphe pondéré et faire un parcours dans un graphe pondéré (ce sera le TP 13.3).

- on créé un dictionnaire vide `G`;
- on parcourt tous les éléments de la matrice à l'aide d'une double boucle sur les indices `i` et `j`;
- si une case contient un 0, on ne fait rien;
- si une case contient un entier strictement positif, ajoute une clé `(i,j)` à `G` dont la valeur associée est la liste contenant les couples `((k,l), poids)` où `(k,l)` parcourt les indices des cases disponibles qui sont adjacentes à `(i,j)` et `poids` est le temps mis pour aller sur la case `(k, l)`.

→ *Q.12) Écrire une fonction `graphe_pondere(M)` qui prend une matrice en paramètre et renvoie le graphe correspondant.*

On fournit la fonction `visualise_terrain_bis` pour faire apparaître les obstacles en bleu.