TP d'informatique n°14
Laissez moi sortir* ! ! !

PCSI 2024 – 2025

I Objectifs :

- utiliser les parcours de graphe pour trouver la sortie/le chemin le plus court jusqu'à la sortie ;
- déterminer le caractère connexe ou non ;
- détecter la présence de cycles.

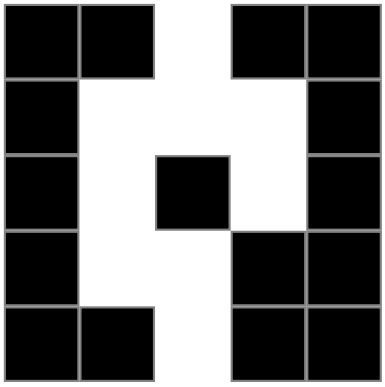
II Introduction

Dans ce TP, les labyrinthes seront représentés par des tableau numpy carré de taille $N \times N$. La présence d'un "0" indique un couloir, c'est-à-dire une case accessible, la présence d'un "1" indique un mur, c'est à dire une case inaccessible.

Ainsi, par exemple, la matrice

```
[[1, 1, 0, 1, 1],
 [1, 0, 0, 0, 1],
 [1, 0, 1, 0, 1],
 [1, 0, 0, 1, 1],
 [1, 1, 0, 1, 1]]
```

Représenterait le labyrinthe ci-contre (noir = mur, blanc = couloir).



Dans la suite du TP, on considèrera que l'entrée du labyrinthe est en haut «au milieu» $(0, N//2)$ et la sortie en bas au milieu $(N-1, N//2)$.

Un avantage d'un tableau numpy par rapport à une liste de liste pour notre TP d'aujourd'hui est la possibilité de prendre en indice un tuple. La convention est la même que pour les matrices : ligne, colonne (mais les indices commencent à 0).

III Mise en œuvre

Exercice n°1 Rappeler la différence entre pile et file. Comment les manipuler en python ? Tester.

Exercice n°2 Écrire une fonction `voisins(s:tuple, N:int)->list[tuple]` qui prend en argument un tuple $s = (ligne,col)$ représentant une position dans le labyrinthe et N la taille du labyrinthe et renvoie la liste des cases voisines où l'on pourrait se déplacer. On considère qu'on ne peut se déplacer que verticalement ou horizontalement d'une case (pas de déplacement en diagonale). On ne tient pas compte de la présence de mur ou non pour le moment. Peu importe l'ordre dans lequel vous renvoyez les cases possibles.

Exemples :

```
voisins( (0,0), 5 )-> [(1, 0), (0, 1)]
voisins( (2,0), 5 )-> [(1, 0), (3, 0), (2, 1)]
voisins( (2,2), 5 )-> [(1, 2), (3, 2), (2, 1), (2, 3)]
voisins( (4,2), 5 )-> [(3, 2), (4, 1), (4, 3)]
```

*du labyrinthe, pas de la prépa ... juste pour être 100% d'accord.

Exercice n°3 Plusieurs labyrinthes sont récupérables sur internet (sur mon site http://grenard.dyndns.org/info.html) pour ce TP. Une fois les fichiers textes sauvegardés dans le même répertoire que votre programme, vous pourrez les charger via la commande `laby1 = np.loadtxt("laby1.txt")` (après avoir importé numpy et en adaptant pour les autres labyrinthes).

Exercice n°4 Après avoir importé matplotlib.pyplot, afficher les labyrinthes via les commandes : plt.figure() puis plt.imshow(laby1) et enfin plt.show().

Exercice n°5 Répondez aux questions suivantes pour chaque labyrinthe : existe-t-il un chemin entre l'entrée et la sortie ? Le labyrinthe est-il connexe ? Y-a-t'il un cycle ?

Exercice n°6 Si l'on souhaite trouver le chemin le plus court jusqu'à la sortie, faut-il plutôt implémenter BFS ou DFS ? Quelle structure (pile ou file) sera alors nécessaire ?

Exercice n°7 Écrire une fonction `existe_sortie(laby:np.array)->bool` qui, via un parcours en largeur[1], teste si le labyrinthe passé en argument possède un chemin (une chaine) entre l'entrée du labyrinthe et la sortie du labyrinthe. Testez avec les différents labyrinthes donné en exemple (vous pouvez les modifier si vous le souhaiter pour rajouter ou enlever un accès à la sortie). Pour «couleur», vous pouvez créer un tableau de même taille que le labyrinthe en faisant `coul = laby*0` et prendre en convention «0=blanc», «1=gris», «2=noir» par exemple.

Exercice n°8 Améliorer votre fonction pour, s'il existe un chemin jusqu'à la sortie, renvoyer un chemin dans l'ordre. Par exemple avec le labyrinthe du début, il faudrait renvoyer [[0,2],[1,2],[1,1],[2,1],[3,1],[3,2],[4,2]] On pourra garder un **dictionnaire** des sommets «parents/antécédents» pour remonter depuis la sortie jusqu'au début à la fin de l'algorithme.

Exercice n°9 Créer une fonction `est_connexe(laby:np.array)->bool` qui renvoie `True` ou `False` en fonction de si le labyrinthe est connexe ou non.

Exercice n°10 Créer une fonction `a_cycle(laby:np.array)->bool` qui renvoie `True` ou `False` en fonction de si le labyrinthe présente un cycle ou non. Il est «conseillé» de faire un parcours en profondeur (facilite la prochaine question).

Exercice n°11 Améliorer la fonction précédente pour renvoyer le cycle en question (liste des sommets dans le cycle).

Exercice n°12 Créer vous même des labyrinthes avec un parcours en profondeur avec choix aléatoire du voisin.

IV Pour la prochaine fois.

Exercice n°13 (Il est possible de partir d'un des programmes que vous avez fait et de le modifier plutôt que de partir de 0) Écrire une fonction `distance_min(laby:np.array)->np.array` qui prend en argument un labyrinthe et renvoie un tableau numpy de même taille contenant les distances minimales depuis l'entrée jusqu'à chaque point du labyrinthe. On prendra comme convention -1 lorsque le point n'est pas accessible (mur ou composante connexe différente). Si N est la taille du labyrinthe, le résultat pourra être initialisée par la commande suivante : `res = -1 + np.zeros( (N,N), dtype = int)`.

1. Utilisez votre cours d'info pour vous aider au besoin... même si ça ne devrait pas être nécessaire !