

## Comment sortir d'un labyrinthe ?

Dans ce TD vous allez écrire un algorithme permettant de sortir d'un labyrinthe. Ce labyrinthe sera représenté sous forme de matrice 2D (coordonnées discrètes) qui servira de structure de donnée de stockage et de parcours. Le point de départ du parcours sera toujours de coordonnées (0, 0) en haut à gauche, et le point d'arrivée le points en bas à droite (dans le cas de l'exemple donné ci-dessous (9, 9)). Les murs de valeur 1 sont infranchissables; on ne peut pas non plus sortir de la matrice. On peut aller dans 8 directions possibles : haut, bas, gauche, droite, et leurs diagonales correspondantes. Dans ce TD vous allez explorer trois méthodes de parcours, à noter qu'il peut y avoir plusieurs résultats différents mais tous valides.

**Exercice 1.1** – Dans un premier temps recopiez le programme ci-dessous qui charge le labyrinthe et une fonction d'affichage; identifiez un des chemins conduisant à la sortie (fichier `code/load.py`) :

---

```
labyrinthe = [[0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
               [0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
               [0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
               [0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
               [0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
               [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
               [0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
               [0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
               [0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
               [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]]

def affiche_labyrinthe(l):
    print('\n'.join([''.join(['{:4}'.format(item) for item in row])
                     for row in l]))
```

---

**Exercice 1.2** – Proposez un algorithme de décision qui indique si il existe un chemin reliant l'entrée et la sortie. Dans cette question vous utiliserez une approche de *parcours en profondeur* et une implémentation *récursive* comme suit :

- Ecrivez une fonction `voisin` qui renvoie tous les voisins d'une cellule (autrement dit toutes les autres cellules accessibles depuis celle-ci)
- Utilisez cette fonction `voisin` pour effectuer les appels récursifs de votre parcours en profondeur
- Le cas d'arrêt de votre algorithme de parcours est atteint si on se trouve sur la cellule de sortie ou si tous les voisins ont été visités

L'algorithme doit renvoyer `True` si un chemin existe, ou bien `False` si le chemin n'existe pas.

**Exercice 1.3** – Nous allons désormais chercher le chemin permettant de sortir (si il possède une sortie). Proposez tout d'abord un algorithme de *parcours en largeur* avec une implémentation

*itérative* afin de déterminer si il existe une sortie. Rajoutez ensuite une structure de données permettant de mémoriser le chemin parcouru, et de retourner ce chemin. Voici un exemple de chemin pour le labyrinthe ci dessus.

[(0, 0), (1, 0), (2, 1), (3, 2), (4, 3), (5, 4), (5, 5), (6, 6), (7, 7),  
(8, 8), (9, 9)]

A noter que vous pouvez obtenir plusieurs chemins (légèrement) différents.

**Exercice 1.4** – Nous allons maintenant optimiser le parcours car trop de sommets sont explorés, parfois de manière inutile. Une optimisation est possible en utilisant une *heuristique*, à savoir une connaissance a priori de la solution. En effet, étant donné que la cellule de destination est connue, il est possible de ne choisir que les voisins qui se rapprochent de cette solution, au détriment des autres. Afin d'illustrer cela, prenez l'exemple ci-dessous (il s'agit d'un labyrinthe 7x7 qui ne contient aucun mur, donc que des 0) : 49 traitements de cellules sont nécessaires afin d'arriver au coin en bas à droite (les numéros indiquent si le sommet est le  $n$ -ème traité). La matrice de droite indique un parcours utilisant cette heuristique, et comme vous le voyez seulement 29 sommets ont été parcourus (les 0 indiquent les sommets non parcourus).

1	2	5	10	17	26	37
3	4	6	11	18	27	38
7	8	9	12	19	28	39
13	14	15	16	20	29	40
21	22	23	24	25	30	41
31	32	33	34	35	36	42
43	44	45	46	47	48	49

1	2	5	0	0	0	0
3	4	6	10	0	0	0
7	8	9	11	15	0	0
0	12	13	14	16	20	0
0	0	17	18	19	21	25
0	0	0	22	23	24	26
0	0	0	0	27	28	29

- Ecrivez une fonction **heuristique** qui calcule la distance d'une cellule vers une autre en utilisant par exemple la distance de Manhattan pour calculer la distance vers la sortie (il s'agit de la distance séparant deux points en suivant le quadripage).
- Utilisez une file de priorité qui choisira le sommet ayant la distance minimale vers la sortie (inspirez-vous du code donné ci-dessous).

Vous pourrez utiliser le module Python **PriorityQueue** pour gérer une file de priorité qui renvoie la valeur minimale insérée avec votre information (fichier `code/priorite.py`) :

---

```
from queue import PriorityQueue
```

```
# initialisation de la file
file_prio = PriorityQueue()
```

```
# remplissage
```

```
file_prio.put((2, "Bob"))
file_prio.put((1, "Alice"))
file_prio.put((6, "Nat"))

# permet d'accéder au premier element de la file
# (sans le supprimer)
print(file_prio.queue[0])

# tant que non vide, affiche par ordre de priorite
# (mais supprimer chaque element accede)
while not file_prio.empty():
    print(file_prio.get()[1])
```

---

Testez par exemple sur de grands labyrinthes vides afin de montrer le gain de temps de la méthode avec heuristique :

```
n = 10
l = [[0 for i in range(n)] for j in range(n)]
```

**Exercice 1.5** – Illustrez vos algorithmes avec différents labyrinthes afin de les valider (avec un labyrinthe vide, sans issue, sans mur, etc.) et justifiez leur efficacité (minimisation du nombre de cases visitées, etc.). Illustrez vos résultats comme avec les matrices au dessus en stockant les étapes de traitement (ou la distance) en effectuant une copie de votre labyrinthe dans laquelle vous stockerez soit les étapes de traitement, soit les valeurs de l'heuristique.