

## 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 point 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.

**SOLUTION:**

```
from queue import PriorityQueue
from math import sqrt
import random
from labyrinthes import liste_labyrinthes
import timeit

# g n rer un labyrinthe alatoire
def generer(x, y):
    return [[random.randint(0, 1) for i in range(x)] for y in range(y)]

def affiche_labyrinthe_murs(l):
    print('\n'.join(''.join('#' if item else '.' for item in row) for row in l))
    print()

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

*# question 1.2*

```
def voisins(l, x, y):
    return ((x+dx, y+dy)
            for dx, dy in ((1,-1), (1,0), (1,1), (0,-1), (0,1), (-1,-1), (-1,0), (-1,1))
            if 0 <= x+dx < len(l[0]) and 0 <= y+dy < len(l) and l[y+dy][x+dx] == 0)
"""
```

*IMPORTANT : chaque appel recursif ne doit pas modifier les chemins des autres, donc chacun a une copie non modifiable du chemin, ce pourquoi on utilise un tuple plutot que liste*

```
def existe_profondeur(l, x0=0, y0=0, chemin=()):
    # print(x0, y0)
    if (x0, y0) == (len(l[0])-1, len(l)-1): # condition de terminaison
        return True
    chemin += ((x0, y0),) # on cree un nouveau chemin avec la position courante
    for x, y in voisins(l, x0, y0):
        if (x, y) not in chemin: # on ignore le voisin si deja visite par le chemin courant
            return existe_profondeur(l, x, y, chemin)
    return False # aucun des voisins ne mene a la sortie, donc l actuelle non plus
```

```
def existe_largeur(l):
    todo = [(0,0)] # file
    dejaVu = [[False] * len(ligne) for ligne in l] # matrice des cellules qu'on a deja ajoutes
    dejaVu[0][0] = True
    while todo and todo[0] != (len(l[0])-1, len(l)-1): # sortie si trouve ou + de cellules a explorer
        x0, y0 = todo.pop(0) # on retire au debut pour un parcours en largeur (fin pour profondeur)
        for x, y in voisins(l, x0, y0):
            if not dejaVu[y][x]:
                dejaVu[y][x] = True
                todo.append((x, y)) # ajout en fin de todo list
    return len(todo) > 0 # il existe un chemin SSI on a quitte la boucle en trouvant la sortie
```

*# question 1.3*

```
def solution_largeur(l):
    todo = [(0,0)] # file

    debug = [row[:] for row in l] # copie du labyrinthe

    antecedente = [[None] * len(ligne) for ligne in l] # matrice des cellules dont on vient
    antecedente[0][0] = (0, 0) # important pour que la cellule soit consideree comme deja visitee
    nb_traitements = 1
    debug[0][0] = 1

    while todo and todo[0] != (len(l[0])-1, len(l)-1): # sortie si trouve ou + de cellules a explorer
        x0, y0 = todo.pop(0) # on retire au debut pour un parcours en largeur (fin pour profondeur)
        for x, y in voisins(l, x0, y0):
```

```

        if antecedente[y][x] == None:
            antecedente[y][x] = (x0, y0)
            todo.append((x, y)) # ajout en fin de todo list
            nb_traitements += 1
            debug[y][x] = nb_traitements
    if not todo: # S'il n'existe aucun chemin atteignant la sortie on renvoie None
        return None
    chemin = [todo.pop(0)] # on recupere la position d arrivee en tete de todo
    while (x, y) != (0, 0): # condition importante car antecedente[0][0] boucle sur elle-meme
        x, y = antecedente[y][x]
        chemin.append((x, y))
    chemin.reverse()
    affiche_labyrinthe(debug)
    # affiche_labyrinthe(l) # si on veut afficher la solution dessin e
    return chemin, nb_traitements

def heuristique(a, b):
    return abs(a[0] - b[0]) + abs(a[1] - b[1])

# question 1.4
def solution.largeur_prio(l):

    file_prio_todo = PriorityQueue()

    debug = [row[:] for row in l] # copie du labyrinthe

    file_prio_todo.put((heuristique((0,0), (len(l[0])-1, len(l)-1)), (0,0)), (0,0))
    antecedente = [[None] * len(ligne) for ligne in l] # matrice des cellules dont on vient
    antecedente[0][0] = (0, 0) # important pour que la cellule soit consideree comme deja visitee
    nb_traitements = 1
    debug[0][0] = 1

    while not file_prio_todo.empty() and file_prio_todo.queue[0][1] != (len(l[0])-1, len(l)-1): # so
        x0, y0 = file_prio_todo.get()[1] # on retire au debut pour un parcours en largeur (fin pour p
        for x, y in voisins(l, x0, y0):
            if antecedente[y][x] == None:
                antecedente[y][x] = (x0, y0)
                file_prio_todo.put((heuristique((x,y), (len(l[0])-1, len(l)-1)), (x,y)))
                nb_traitements += 1
                debug[y][x] = nb_traitements # heuristique((x,y), (len(l[0])-1, len(l)-1))

                #todo.append((x, y)) # ajout en fin de todo list
    if file_prio_todo.empty(): # S'il n'existe aucun chemin atteignant la sortie on renvoie None
        return None
    chemin = [file_prio_todo.get()[1]] # on recupere la position d arrivee en tete de todo

```

---

```

while (x, y) != (0, 0): # condition importante car antecedente[0][0] boucle sur elle-meme
    x, y = antecedente[y][x]
    chemin.append((x, y))
chemin.reverse()

affiche.labyrinthe(debug)
# affiche_labyrinthe(l) # si on veut afficher la solution dessin e
return chemin, nb_traitements

if __name__=="__main__":

    n = 7 # taille du labyrinthe
    l2 = [[0 for i in range(n)] for j in range(n)]
    affiche.labyrinthe(l2)

    print(existe_profondeur(l2))

    # print(existe_profondeur(l2))
    # print(existe_largeur(l2))
    # print(solution_largeur(l2))
    # print(solution_largeur_prio(l2))
    # i = 0
    ## tests sur les labyrinthes import s
    # for l, s in liste_labyrinthes:
    #     starttime = timeit.default_timer()
    #     i+=1
    #     r = existe_profondeur(l)

    #     if s is None: # pas de solution
    #         assert r == False
    #     else:
    #         assert r == True
    #         print("test OK pour l{} temps d'excution :".format(i), timeit.default_timer() - starttime)

```

---