

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). Le point de départ sera toujours de coordonnées (0, 0) 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.

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, 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. Une approche simple est le parcours en largeur, avec un mécanisme de mémorisation afin d'éviter de re-parcourir les noeuds déjà parcouru.

Exercice 1.3 – On sait qu'il existe un ou plusieurs chemins pour sortir, mais pas forcément lequel. Rajoutez une structure de données permettant de mémoriser le chemin parcouru, et une fonction d'affichage de ce chemin parcouru. Un chemin ressemble à la suite de couples comme suit :

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

étapes : 11
```

Exercice 1.4 – Enfin on a trouvé un chemin.. mais pas forcément le plus court ! Pour cela, améliorez votre algorithme afin de prendre en compte la minimisation de 1) la distance vers la sortie, et 2) la distance déjà parcourue. Conseil : utilisez la distance de Manhattan pour calculer la distance vers la sortie, pour la distance parcourue il s'agit

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

```
from queue import PriorityQueue

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

while not file_prio.empty():
    print(file_prio.get()[1])
```

Exercice 1.5 – Illustrez vos différents algorithmes avec différents labyrinthes afin de tester (un labyrinthe vite, sans issue, sans mur, etc.) et de montrer leur efficacité (minimisation du nombre de cases visitées, parcours optimal).

SOLUTION:

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

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

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) in chemin: # on ignore le voisin si deja visite par le chemin courant
        continue
    if existe_profondeur(l, x, y, chemin): # appel recursif a partir de la position voisine
        return True # on a trouve un chemin par la position voisine, donc aussi par l actuelle
    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 ajoutees
    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

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

affiche_labyrinthe(labyrinthe)
print(existe_profondeur(labyrinthe))
print(existe_largeur(labyrinthe))
print(solution_largeur(labyrinthe))

```
