Labyrinthes et structures de données

INFORMATIQUE COMMUNE - TP n° 3.7 - Olivier Reynet

À la fin de ce chapitre, je sais :

☞ utiliser les listes imbriquées,
☞ utiliser des dictionnaires,
☞ utiliser un tableau bidimensionnel sous la forme de liste imbriquées,
☞ représenter graphiquement un labyrinthe,
☞ générer un labyrinthe de deux manières différentes.
☞ explorer un labyrinthe.

Ce TP est construit dans l'esprit d'une épreuve de concours et de manière progressive. Il permet de réviser de nombreux éléments du programme de première année.

A Contexte

Le contexte de ce TP est l'exploration de labyrinthes. On considère des labyrinthes parfaits, c'est à dire dont deux cases quelconques peuvent toujours être reliées par un unique chemin. Par convention, ces labyrinthes sont des pièges et on ne peut pas en sortir : leur périphérie est constituée de murs (cf. figure 1). On choisit arbitrairement que la case de départ est en bas à gauche et l'arrivée en haut à droite (cf. figure 2).

On cherche à :

1. modéliser un tel labyrinthe,
2. afficher à l'écran un labyrinthe,
3. générer des labyrinthes,
4. explorer un labyrinthe, c'est à dire à en sortir **sans savoir où se trouve la sortie**.

B Modéliser et implémenter un labyrinthe

On choisit de modéliser un labyrinthe sous la forme d'un tableau bidimensionnel de taille fixe. La constante `N_COLS` désigne le nombre de colonnes et `N_ROWS` le nombre de lignes de ce tableau. Ce tableau bidimensionnel est **implémenté** par une liste imbriquée en Python, c'est à dire une liste de listes. Par exemple, `maze[0][2]` renvoie la cellule de la première ligne et troisième colonne du labyrinthe.

Chaque case du tableau modélise une cellule du labyrinthe. Une cellule possède quatre points cardinaux North, East, South, West que l'on notera `"N"`, `"E"`, `"S"`, `"W"` en Python. Selon ces directions, on peut construire ou abattre un mur autour de la cellule. On choisit d'**implémenter** une cellule par un

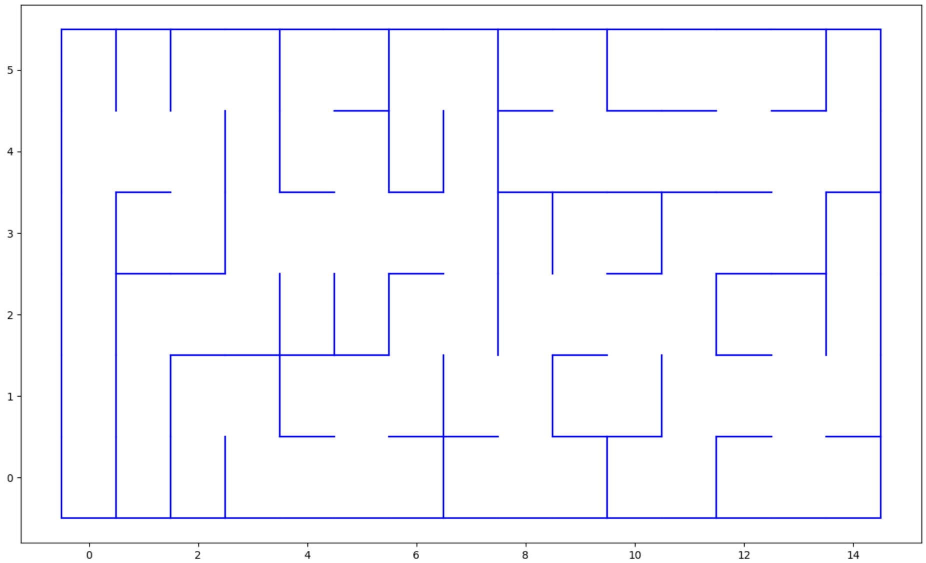


FIGURE 1 – Labyrinthe parfait rectangulaire

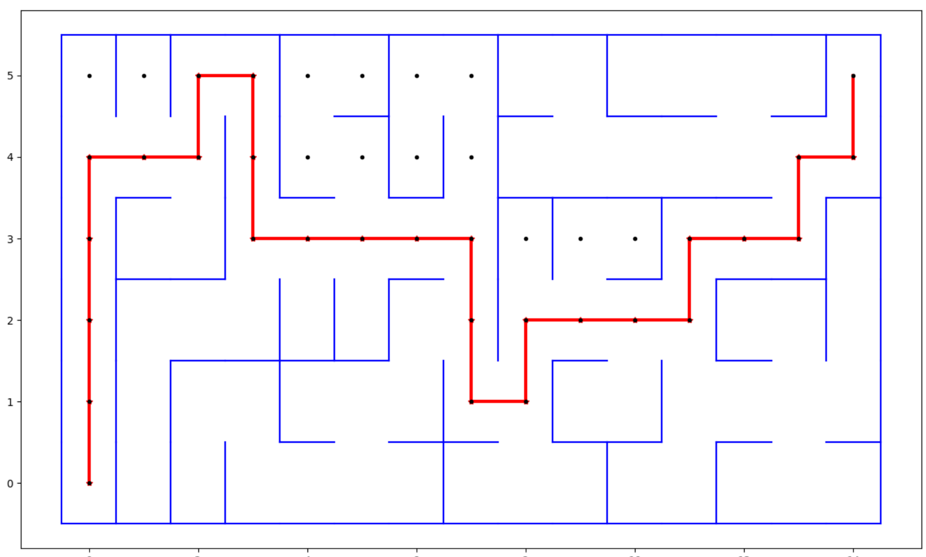


FIGURE 2 – Exploration d'un labyrinthe parfait rectangulaire. Le départ est effectué en bas à gauche et l'arrivée est en haut à droite.

dictionnaire dont les clefs sont les quatre points cardinaux et les valeurs un booléen. **La convention adoptée est qu'un mur existe si ce booléen vaut True.**

B1. En considérant le labyrinthe de la figure 1, créer un dictionnaire Python qui modélise les cellules d'indice (ligne, colonne) :

(a) (0,0),

(b) (1,4),

(c) (4,3),

(d) (14,3)

Solution :

```python
cell_0_0 = {'N': False, 'S': True, 'W': True, 'E': True}
cell_1_4 = {'N': False, 'S': True, 'W': False, 'E': False}
cell_4_3 = {'N': True, 'S': False, 'W': False, 'E': False}
cell_14_3 = {'N': True, 'S': False, 'W': True, 'E': True}
```

B2. Écrire une fonction de prototype `create_closed_cell()` qui renvoie une cellule complètement murée, c'est à dire qui possède des murs dans toutes les directions.

Solution :

```python
def create_closed_cell():
  return {"N": True, "S": True, "W": True, "E": True}
  # True means there is a wall !
```

B3. Écrire une fonction de prototype `is_closed(cell)` où le paramètre `cell` est un dictionnaire représentant une cellule du labyrinthe. Cette fonction renvoie un booléen qui atteste que la cellule possède un mur dans chaque direction, c'est à dire qu'elle est complètement close.

Solution :

```python
def is_closed(cell):
  # Simple way
  for k in cell:
    if not cell[k]:
      return False
  return True
  # Hard way
  # return not (False in cell.values())
```

B4. Écrire une fonction de prototype `create_all_closed_maze(n_rows, n_cols)` qui renvoie un labyrinthe de taille `n_rows`×`n_cols` et dont les cellules sont complètement murées. Le tester en utilisant une assertion et la fonction précédente, c'est à dire vérifier que toutes les cellules sont closes.

Solution :

```
N_ROWS = 6
N_COLS = 15

def create_all_closed_maze(n_rows, n_cols):
  # Simple way
  maze = []
  for i in range(n_rows):
    maze.append([])
    for j in range(n_cols):
      maze[i].append(create_closed_cell())
  return maze
  # Hard way
  # return [[create_closed_cell() for j in range(n_cols)] for i in range(
      n_rows)]


def B4():
  maze = create_all_closed_maze(N_ROWS, N_COLS)
  for i in range(N_ROWS):
    for j in range(N_COLS):
      assert is_closed(maze[i][j])
```

B5. Écrire une fonction qui permette d'abattre un mur d'une cellule selon une certaine direction. Son prototype est `tear_down(maze,i,j,direction)`, où `maze` est le labyrinthe, `i` et `j` les indices de ligne de colonne de la cellule considérée et `direction` la direction `"N"`, `"E"`, `"S"` ou `"W"` selon laquelle on souhaite abattre le mur [1]. Cette fonction ne renvoie rien et il faut être capable d'expliquer pourquoi ;-) Tester cette fonction avec des assertions et la fonction `is_closed`. Cette fonction sera utile à la section E.

Solution : La fonction `tear_down(maze,i,j,direction)` ne renvoie rien car le labyrinthe est passé en paramètre sous la forme d'un objet muable de type `list`. Il est modifiable et modifié par la fonction. On n'a donc pas besoin de le renvoyer.

```
N_ROWS = 6
N_COLS = 15

def tear_down(maze, i, j, direction):
  maze[i][j][direction] = False
  if direction == "N" and i < len(maze) - 1:
    maze[i + 1][j]["S"] = False
  elif direction == "S" and i > 0:
    maze[i - 1][j]["N"] = False
  elif direction == "W" and j > 0:
    maze[i][j - 1]["E"] = False
  elif direction == "E" and j < len(maze[0]) - 1:
    maze[i][j + 1]["W"] = False
  else:
    raise Exception("Unknown direction --> " + direction)
```

1. Il faut remarquer qu'un mur est constitué de deux cloisons...

```python
def B5():
  maze = create_all_closed_maze(N_ROWS, N_COLS)
  tear_down(maze, 2, 2, "N")
  tear_down(maze, 2, 2, "W")
  tear_down(maze, 2, 2, "S")
  tear_down(maze, 2, 2, "E")
  # Simple test
  assert not is_closed(maze[2][2]), "this cell should be open !"
  assert not is_closed(maze[3][2]), "this cell should be open !"
  # Full test
  for i in range(N_ROWS):
    for j in range(N_COLS):
      # print(i, j, maze[i][j])
      if (i, j) in [(2, 2), (2, 1), (3, 2), (2, 3), (1, 2)]:
        assert not is_closed(maze[i][j]), f"({i},{j}) should be open !"
      else:
        assert is_closed(maze[i][j]), f"({i},{j}) should be closed !"
```

B6. Créer le tableau bidimensionnel de booléens nommé `visited` de la même dimension que le labyrinthe sous la forme d'une liste imbriquée. Lorqu'on explore le labyrinthe, ce tableau représente le fait d'être passé sur une cellule : si le booléen vaut True, c'est qu'on est déjà passé sur la cellule. On initialisera à `False` toutes les cases du tableau. Pour savoir si on est passé sur la cellule (3,4) , on testera la valeur de `visited[3][4]`.

Solution :

```python
def B6():
  # Simple way
  visited = []
  for i in range(N_ROWS):
    visited.append([])
    for j in range(N_COLS):
      visited[i].append(False)
  # Hard way
  visited = [[False for j in range(N_COLS)] for i in range(N_ROWS)]
```

B7. Écrire une fonction de prototype `dir_unvisited_neighbours(visited, i, j)` qui renvoie les directions des cellules voisines de la case (i,j) qui n'ont pas été visitées sous la forme d'une liste, par exemple `["N", "E"]`. Pour bien tester cette fonction, certaines cases de `visited` doivent être à `True` . Cette fonction sera utile à la section E.

Solution :

```python
def dir_unvisited_neighbours(visited, i, j):
  unvisited = []
  if i < len(visited) - 1 and not visited[i + 1][j]:
    unvisited.append("N")
  if i > 0 and not visited[i - 1][j]:
```

```python
      unvisited.append("S")
    if j > 0 and not visited[i][j - 1]:
      unvisited.append("W")
    if j < len(visited[0]) - 1 and not visited[i][j + 1]:
      unvisited.append("E")
    return unvisited

  def B7():
    visited = [[False for j in range(N_COLS)] for i in range(N_ROWS)]
    visited[3][2] = True
    assert ["S", "W", "E"] == dir_unvisited_neighbours(visited, 2, 2)
    assert ["N", "S", "W", "E"] == dir_unvisited_neighbours(visited, 1, 1)
    assert ["N", "W", "E"] != dir_unvisited_neighbours(visited, 1,3)
```

C Afficher un labyrinthe

On suppose pour l'instant qu'on a réussi à générer un labyrinthe, à l'explorer et à récupérer toutes ces informations. Pour répondre aux questions qui suivent, il est nécessaire de télécharger le fichier Pyhon maze_example.py. Il contient le labyrinthe de l'exemple dessiné sur la figure 1, le chemin d'exploration trouvé et les cellules visitées. L'objectif est de programmer les fonctions nécessaires pour afficher le labyrinthe et les résultats comme sur la figure 2. Les constantes du labyrinthe sont pour cet exemple :

```python
N_ROWS = 6  # Maze's width
N_COLS = 15  # Maze's height
```

Les fonctions à programmer dans cette section ne renvoient rien, elles tracent seulement. Elles s'appuient sur la bibliothèque **matplotlib** et la **convention suivante** :

- les coordonnées (i,j) de la cellule dans le tableau correspondent au centre de la case telle qu'elle est dessinée sur la figure,
- les points d'exploration sont matérialisés au centre de la case,
- les chemins passent par le centre de cette case,
- les murs se situent à égale distance du centre de la case et ne débordent pas sur l'autre case,
- les cloisons d'un mur entre deux cellules sont superposées.

C1. À l'aide d'une directive d'importation de bibliothèque, importer tous les éléments du fichier téléchargé.

Solution :

```python
from maze_example import *
```

C2. Vérifier que la valeur des constantes `N_ROWS` et `N_COLS` définies au début de votre programme sont cohérentes avec les dimensions du labyrinthe `maze_ex` donné en exemple.

Solution :

```
from maze_example import *
assert N_ROWS == len(maze_ex)
assert N_COLS == len(maze_ex[0])
```

C3. À l'aide d'une directive d'importation de bibliothèque, importer le module `pyplot` de la bibliothèque `matplotlib` sous le nom `plt`.

Solution :

```
from matplotlib import pyplot as plt
```

C4. Écrire une fonction de prototype `draw_visited(maze, visited)` qui marque les cellules visitées lors de l'exploration **d'un point noir au centre de la cellule**. Dans le programme principal, effectuer un `plt.show()` pour visualiser le résultat.

Solution :

```
def draw_visited(maze, visited):  # mark visited cells
  for row in range(N_ROWS):
    for col in range(N_COLS):
      if visited[row][col]:
        plt.plot(col, row, marker=".", color="black")
```

C5. Écrire une fonction de prototype `draw_path(maze, path)` qui trace le chemin `path` passé en paramètre sur le labyrinthe `maze`. Ce chemin a l'apparence d'**une ligne rouge**. Les puristes ajouteront une étoile rouge au centre de la case traversée.

Solution :

```
def draw_path(maze, path): # draw path to exit
  if path:    # same as : if len(path) > 0
    start_y, start_x = path[0]
  for i in range(1, len(path)):
    next_y, next_x = path[i]
    plt.plot(start_x, start_y, marker="*", color="red")
    plt.plot([start_x, next_x], [start_y, next_y], color="red", linewidth=3)
    start_x, start_y = next_x, next_y
  # on peut faire plus court avec zip...
```

C6. Écrire une fonction de prototype `draw_maze(maze)` qui trace le labyrinthe passé en paramètre. Les murs sont bleus.

Solution :

```python
def draw_maze(maze):
plt.clf()  # erase previous maze
for row in range(N_ROWS):
  for col in range(N_COLS):
    if maze[row][col]["E"]:
      plt.plot([col + 0.5, col + 0.5], [row - 0.5, row + 0.5], 'b-')
    if maze[row][col]["S"]:
      plt.plot([col - 0.5, col + 0.5], [row - 0.5, row - 0.5], 'b-')
    if maze[row][col]["N"]:
      plt.plot([col - 0.5, col + 0.5], [row + 0.5, row + 0.5], 'b-')
    if maze[row][col]["W"]:
      plt.plot([col - 0.5, col - 0.5], [row - 0.5, row + 0.5], 'b-')
```

D Explorer et sortir d'un labyrinthe

On souhaite maintenant pouvoir trouver la sortie du labyrinthe. A priori, on doit pouvoir définir n'importe quelle cellule comme cellule de sortie. Au niveau de l'implémentation, pour distinguer cette cellule des autres, on la matérialise **en ajoutant à son dictionnaire une clef** `"EXIT"` dont la valeur est `True`.

On dispose de l'algorithme 1 qui permet de sortir d'un labyrinthe en l'explorant récursivement[2].

Algorithme 1 Trouver la sortie d'un labyrinthe en explorant récursivement

1 : **Fonction** EXPLORER_SORTIR(maze, i, j, visited, path) ▷ au premier appel, path est une liste vide.
2 : visited[i][j] ← Vrai ▷ au premier appel, visited est totalement à Faux.
3 : Ajouter (i,j) à path
4 : **si** maze[i][j] est la sortie **alors** ▷
5 : **renvoyer** Vrai
6 : **sinon**
7 : **pour** chaque direction `"N"`, `"S"`,`"E"`, `"W"` **répéter** ▷ On ne traduira pas par une boucle.
8 : **si** un case voisine non visitée existe dans cette direction **alors**
9 : p,q ← les coordonnées de ce voisin non visité dans cette direction
10 : **si** EXPLORER_SORTIR(maze, p, q, visited, path) **alors**
11 : **renvoyer** Vrai ▷ On a trouvé la sortie !
12 : **sinon**
13 : Retirer (p,q) du path ▷ C'est la dernière case visitée et ajoutée qu'on retire.
14 : **renvoyer** Faux ▷ Pas de sortie par ici...

D1. À quoi sert la ligne 4 de l'algorithme 1 ? Comment pouvez-vous la traduire en Python ?

Solution : C'est elle qui permet de repérer la sortie. Grâce à notre choix pour marquer la cellule de sortie, on peut écrire : `if "EXIT"in maze[i][j]:`.

2. Cette technique est un algorithme classique d'exploration nommé retour sur trace ou backtracking.

D2. Au cours de l'algorithme 1, passe-t-on deux fois sur la même case ?

Solution : Non, grâce à l'usage tableau `visited`.

D3. L'algorithme 1 parcourt-il toutes les cases ?

Solution : Non, dès qu'il trouve la sortie, il renvoie les informations en interrompant l'exploration.

D4. Écrire une fonction Python qui implémente l'algorithme 1. Vérifier que le chemin obtenu correspond avec celui donné à la section précédente.

Solution :

```python
def explore_exit(maze, i, j, visited, path):  # BACKTRACKING
  #print("(i,j) --> ", i, j)
  visited[i][j] = True
  path.append((i, j))
  n_rows = len(maze)
  n_cols = len(maze[0])
  if "EXIT" in maze[i][j]:
    #print("EXIT FOUND (i,j) --> ", i, j)
    return True
  else:
    if i < n_rows - 1 and not maze[i][j]["N"] and not visited[i + 1][j]:
      if explore_exit(maze, i + 1, j, visited, path):
        return True
      else:
        path.pop()  # backtrack, no way
    if j < n_cols - 1 and not maze[i][j]["E"] and not visited[i][j + 1]:
      if explore_exit(maze, i, j + 1, visited, path):
        return True
      else:
        path.pop()  # backtrack, no way
    if i > 0 and not maze[i][j]["S"] and not visited[i - 1][j]:
      if explore_exit(maze, i - 1, j, visited, path):
        return True
      else:
        path.pop()  # backtrack, no way
    if j > 0 and not maze[i][j]["W"] and not visited[i][j - 1]:
      if explore_exit(maze, i, j - 1, visited, path):
        return True
      else:
        path.pop()  # backtrack, no way
  return False


def D4():
  from maze_example import maze_ex
  n_rows = len(maze_ex)
  n_cols = len(maze_ex[0])
  v = [[False for j in range(n_cols)] for i in range(n_rows)]
  p = []
```

```
explore_exit(maze_ex, 0, 0, v, p)
draw_maze(maze_ex)
draw_path(maze_ex, p)
draw_visited(maze_ex, v)
plt.show()
```

D5. Pourrait-on définir plusieurs sorties dans le labyrinthe ?

Solution : Oui, il suffit d'ajouter une clef `"EXIT"` aux cellules concernées. L'algorithme sortira à la première rencontrée.

E Générer un labyrinthe par exploration exhaustive

Les labyrinthes parfaits, c'est à dire qui relient deux cellules quelconques par un unique chemin, peuvent être générés par exploration exhaustive de toutes les cellules. La situation de départ est un labyrinthe dont toutes les cellules sont closes. Puis, une cellule est aléatoirement choisie comme point de départ et on la marque comme étant *visitée*. Parmi les cellules voisines, on en choisie une non visitée et on abat la cloison entre les deux cellules. On réitère la procédure avec cette cellule voisine comme nouvelle cellule. Le labyrinthe est fini de construire lorsque l'exploration est finie, c'est à dire lorsqu'il n'y a plus de cellules non visitées.

L'algorithme 2 décrit cette procédure de manière itérative à l'aide d'une pile pour garder la trace des cellules dont il faut chercher les voisins : tant qu'on n'a pas visité tous les voisins d'une cellule, celle-ci doit être présente dans la pile.

Pour l'implémentation, on utilisera une `list` Python pour implémenter une pile, en utilisant les méthodes `append((i,j))` pour empiler et `pop()` pour dépiler. Il faudra également se servir des fonctions programmées à la section B, `create_all_closed_maze`, `tear_down` et `dir_unvisited_neighbours`.

E1. Pour choisir une cellule au hasard, on choisit d'utiliser les fonctions `randrange` et `choice` de la bibliothèque `random`. En utilisant une directive d'importation, importer ces fonctions de cette biliothèque en une seule ligne de code.

Solution :

```
from random import randrange, choice
```

E2. Écrire la fonction `explore_generate(n_rows, n_cols)` qui implémente l'algorithme 2.

Solution :

```
def generate_maze(n_rows, n_cols):
  maze = create_all_closed_maze()
  visited = [[False for j in range(n_cols)] for i in range(n_rows)]
  i, j = randrange(n_rows), randrange(n_cols)  # select a start cell
  pile = []
  pile.append((i, j))
```

Algorithme 2 Générer un labyrinthe en explorant

```
Fonction EXPLORER_GÉNÉRER(n_lignes, n_cols)
    maze ← un labyrinthe dont les cellules sont fermées de dimension (n_lignes, n_cols)
    visited ← un tableau de booléens initialisés à faux de même dimension que maze
    (i,j) ← une cellule de départ choisie au hasard
    pile ← une pile vide                                          ▷ On l'implémentera par une list Python
    Ajouter (i,j) à pile
    visited[i][j] ← Vrai
    tant que la pile n'est pas vide répéter                       ▷ Il y a encore des cellules voisines à explorer.
        (i,j) ← dépiler(pile)                                     ▷ On prend la dernière qu'on a visité.
        n_dir ← les directions des voisins non visités
        si n_dir possède au moins deux éléments alors             ▷ Il y a plusieurs voisins à considérer.
            empiler(pile, (i,j))                                  ▷ Il faudra donc revenir sur cette cellule.
        si il y a des voisins non visités alors
            dir ← une direction choisie aléatoirement dans n_dir  ▷ un voisin à la fois.
            selon cette direction dir, on a la cellule (p,q)
            abattre le mur reliant (p,q) à (i,j)
            empiler(pile, (p,q))                                  ▷ C'est notre nouvelle cellule !
            visited[p][q] ← Vrai
    renvoyer maze
```

```python
    visited[i][j] = True
    # print("Starting --> ", i, j)
    while len(pile) > 0:
      (i, j) = pile.pop()
      n_dir = dir_unvisited_neighbours(visited, i, j)
      if len(n_dir) > 1:
        pile.append((i, j))  # do not forget to deal with the others
             neighbours later
      if len(n_dir) > 0:  # if there are neighbours
        dir = choice(n_dir)  # pick a random direction among possible
             neighbours
        # print("Direction -> ", dir)
        if dir == "N":
          tear_down(maze, i, j, "N")
          pile.append((i + 1, j))
          visited[i + 1][j] = True
        elif dir == "S":
          tear_down(maze, i, j, "S")
          pile.append((i - 1, j))
          visited[i - 1][j] = True
        elif dir == "W":
          tear_down(maze, i, j, "W")
          pile.append((i, j - 1))
          visited[i][j - 1] = True
        else:
          tear_down(maze, i, j, "E")
          pile.append((i, j + 1))
          visited[i][j + 1] = True
```

```python
    return maze
```

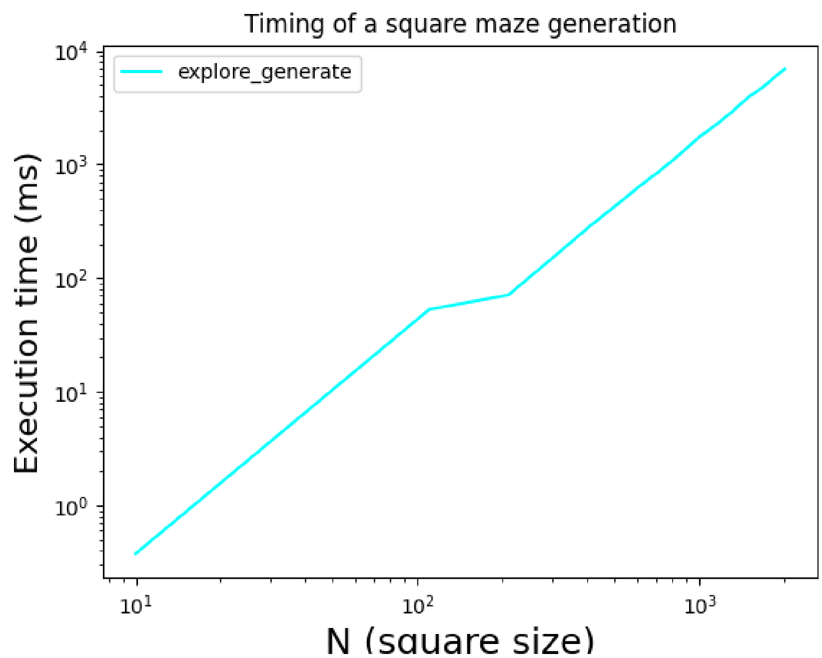
E3. Tester l'algorithme sur des dimensions quelconques et visualiser les résultats. Que peut-on dire de la complexité temporelle de cet algorithme ?

Solution : Dans le pire des cas, une case est mise quatre fois dans la pile, car à chaque tour de boucle, on n'explore qu'un seul voisin. Donc, pour créer tout le labyrinthe et donc visiter toutes les cases, il faut dans le pire des cas $4\times$`N_ROWS`$\times$`N_COLS` tours de boucle. On peut donc en conclure que la complexité temporelle de cet algorithme est en $O($`N_ROWS`$\times$`N_COLS`$)$.

E4. Tracer l'évolution du temps d'exécution de la fonction `explore_generate` en fonction des dimensions du labyrinthe. On pourra supposer que celui-ci est carré.

Solution : On vérifie bien qu'on a une complexité quadratique en fonction du côté du labyrinthe carré.

```python
from time import perf_counter_ns
N = 2011
timing = {}
for i in range(10, N, 100):
  tic = perf_counter_ns()
  maze = explore_generate(i, i)
  toc = perf_counter_ns()
  timing[i] = (toc - tic) * 1e-6
  print(f"{i} --> {timing[i]} ms")
plt.figure()
plt.plot(timing.keys(), timing.values(), color='cyan', label='
    explore_generate')
plt.xlabel('N (square size)', fontsize=18)
plt.ylabel("Execution time (ms)", fontsize=16)
plt.title('Timing of a square maze generation')
plt.legend()
plt.show()
```



E5. Peut-on trouver la sortie de n'importe quel labyrinthe ainsi généré ? Y-t-a-il une limite de dimension du labyrinthe ? Pourquoi ? Comment pourrait-on y remédier ?

Solution : En augmentant la taille du labyrinthe, on va multiplier les appels récursifs dans l'algorithme 1. À partir d'une certaine taille de labyrinthe, la taille de la pile d'exécution allouée par la machine virtuelle Python explose, une exception est levée et le calcul échoue :

```
RecursionError: maximum recursion depth exceeded while calling a Python object
```

On pourrait essayer d'écrire une version itérative de cet algorithme. Par exemple, on pourrait parcourir en profondeur le labyrinthe et s'arrêter dès qu'on trouve la sortie :

```
def ite_explore_exit(maze, i=0, j=0):
  visited = [[False for j in range(N_COLS)] for i in range(N_ROWS)]
  visited[i][j] = True
  path = [(i, j)]
  i, j = path.pop()
  while not "EXIT" in maze[i][j]:
    if i < N_ROWS - 1 and not maze[i][j]["N"] and not visited[i + 1][j]:  # North is available
      path.append((i, j))
      path.append((i + 1, j))
      visited[i + 1][j] = True
    elif j < N_COLS - 1 and not maze[i][j]["E"] and not visited[i][j + 1]:  # East is available
      path.append((i, j))
      path.append((i, j + 1))
      visited[i][j + 1] = True
    elif i > 0 and not maze[i][j]["S"] and not visited[i - 1][j]:  # South is available
```

```python
            path.append((i, j))
            path.append((i - 1, j))
            visited[i - 1][j] = True
        elif j > 0 and not maze[i][j]["W"] and not visited[i][j - 1]:  # West is
            available
            path.append((i, j))
            path.append((i, j - 1))
            visited[i][j - 1] = True
        else:
            pass  # there is no way
        # print("cell --> ", i, j, "path --> ", path)
        i, j = path.pop()
    path.append((i, j))  # add exit cell to path
    return path, visited
```

F Générer un labyrinthe par fusion de chemins

Une seconde approche pour générer un labyrinthe est de créer des chemins et de les fusionner comme indiqué sur l'algorithme 3. La situation de départ est un labyrinthe dont toutes les cellules sont closes. Les cellules sont donc toutes isolées les unes des autres : le labyrinthe est constitué de `N_ROWS` ×`N_COLS` parties non connexes, des débuts de chemins.

Tant qu'il existe au moins deux composantes non connexes distinctes dans le labyrinthe, c'est à dire tant qu'il existe au moins deux chemins qui ne se rejoignent pas, on choisit au hasard un mur entre deux cellules n'appartenant pas au même chemin et on abat ce mur. On a ainsi relié deux chemins qui formaient deux composantes non connexes du labyrinthe.

L'algorithme s'achève lorsqu'il n'existe plus qu'un seul chemin. Comme à chaque tour de boucle on diminue d'un élément le nombre de composantes non connexes du labyrinthe, il est nécessaire de répéter ceci `N_ROWS`×`N_COLS` fois.

On choisit de représenter une composante connexe (un chemin) d'après la cellule qui l'a démarré. À chaque fois qu'on ajoute une cellule au chemin, celle-ci a pour parent la cellule qui l'a intégré dans le chemin. **En remontant les parents de chaque cellule de proche en proche, on finit par trouver la cellule qui a initié le chemin : celle-ci est son propre parent**.

Dans le code, cela se traduit par l'utilisation d'un dictionnaire nommé `parents` dont les clefs sont des cellule (i,j) et les valeurs d'autres cellules (p,q). Par exemple, si (p,q) a été intégrée au chemin par (i,j), alors `parents[(p,q)]` vaut (i,j). **Au démarrage, comme toutes les cellules sont closes, elles sont toutes leur propre parent.**

F1. Écrire les lignes de code pour initialiser le dictionnaire `parents` correctement au début du programme.

Solution :

```python
parents = {(i, j): (i, j) for i in range(n_rows) for j in range(n_cols)}  #
    no connection at all
```

F2. On a extrait un chemin d'un dictionnaire `parents` :

Algorithme 3 Générer un labyrinthe par fusion de chemin

Fonction FUSIONNER_GÉNÉRER(n_lignes, n_cols)
  maze ← un labyrinthe dont les cellules sont fermées de dimension (n_lignes, n_cols)
  parents ← un dictionnaire qui recense le parent de chaque cellule.
  tant que il y a des chemins non connexes **répéter**
    Choisir au hasard une cellule (i,j) puis une seconde (p,q) **voisine** de celle-ci **et non connexe**
    La cellule à l'origine du chemin de (i,j) devient la cellule à l'origine du chemin de (p,q).
    Abattre le mur reliant les cellules (i,j) et (p,q).
  renvoyer maze

```
path = {(0, 0): (0, 0), (0, 1): (0, 0), (0, 2): (0, 1), (0, 3): (0, 2)}
```

(a) Quel chemin `path` représente-t-il? Quelle est la cellule qui a démarré le chemin?

(b) Écrire une fonction de prototype `find_origin(parents, c)` où `parents` est un dictionnaire comme on l'a décrit plus haut, `c` une cellule de type (i,j) et qui renvoie la cellule à l'origine du chemin auquel appartient `c`.

Solution : Le chemin `path` a démarré en (0,0).

```
def find_origin(parents, c):
  while parents[c] != c:
    c = parents[c]
  return c
```

F3. Écrire une fonction de prototype `fusion_generate(n_rows, n_cols)` qui implémente l'algorithme 3.

Solution :

```
def fusion_generate(n_rows, n_cols):  # path fusion
  maze = create_all_closed_maze(n_rows, n_cols)
  parents = {(i, j): (i, j) for i in range(n_rows) for j in range(n_cols)}
      # no connection at all
  for _ in range(n_rows * n_cols - 1):  # at the end, only one connected
      component remains.
    while True:  # take one component which has a neighbour not connected
      i, j = randrange(n_rows), randrange(n_cols)
      p, q = choice([(i + 1, j), (i - 1, j), (i, j + 1), (i, j - 1)])  #
          random neighbour cell
      if 0 <= p < n_rows and 0 <= q < n_cols:
        ok = find_origin(parents, (p, q))  # Who is your parent ?
        ol = find_origin(parents, (i, j))  # Who is your parent ?
        if ok != ol: break  # are they different ?
    parents[ok] = ol  # Union of disjoint sets
    if p == i + 1:  # Let's tear down the wall !
      tear_down(maze, i, j, "N")  # connect cells
    elif p == i - 1:
      tear_down(maze, i, j, "S")  # connect cells
    elif q == j + 1:
```

```
        tear_down(maze, i, j, "E")  # connect cells
      elif q == j - 1:
        tear_down(maze, i, j, "W")  # connect cells
      else:
        raise Exception("Unknown direction !")
    # How many disjoint sets are there ?
    # print("# path connected --> ", len(set(parents.values())))
    return maze
```

G Comparer les méthodes de génération

Les deux méthodes de génération de labyrinthe (par exploration et par fusion de chemins) engendre des résultats très différents.

G1. Comparer la longueur des chemins nécessaires à la découverte de la sortie d'un labyrinthe selon que le labyrinthe a été engendré par la première ou la seconde méthode. On choisira des labyrinthes de taille 30×50 et on répètera l'expérience une centaine de fois.

Solution :

```
def G1():
  n_rows = 30
  n_cols = 50
  L = []
  for _ in range(100):
    m = explore_generate(n_rows, n_cols)
    m[n_rows - 1][n_cols - 1]["EXIT"] = True
    v = [[False for j in range(n_cols)] for i in range(n_rows)]
    p = []
    explore_exit(m, 0, 0, v, p)
    L.append(len(p))
  print("exploration generation --> Mean length path --> ", sum(L) / len(L))
  L = []
  for _ in range(100):
    maze = fusion_generate(n_rows, n_cols)
    maze[n_rows - 1][n_cols - 1]["EXIT"] = True
    v = [[False for j in range(n_cols)] for i in range(n_rows)]
    p = []
    explore_exit(maze, 0, 0, v, p)
    L.append(len(p))
  print("fusion path generation --> Mean length path --> ", sum(L) / len(L))
```

G2. Interpréter les résultats.

Solution : Les chemins générés en mode exploratoire sont d'une longueur moyenne de l'ordre de 385. Ceux générés par la fusion de chemins sont d'une longueur moyenne de 138. La seconde méthode permet d'obtenir un réseau de chemins plus courts et plus interconnectés.