AIX MARSEILLE UNIVERSITE

Licence 3 Mathématiques

Programmation 3

PROJET MAZE

GEYER Rayane
KEBIR Younes

January 26, 2024

Contents

1 Le projet Maze

L'objectif du projet Maze est de générer des labyrinthes de différents aspects, carrés, hexagonaux... Pour ce faire nous utiliserons des notions de programmation orientée objet et nous travaillerons sur des algorithmes de génération et de résolution de labyrinthe.

On suppose dans tout le projet que les modules ci dessous ont été importés :

```
from graph import *
from render import *
from coveringtree import *
import random
```

2 La classe Maze

Dans cette première section on se propose de créer une classe Maze qui sera le receptacle de notre projet.

Cette classe est implanté dans le module maze.py et permet :

— de générer et retourner un graphe non-oriente en utilisant la classe Graph via une méthode muni de deux arguments : widht et height. On laisse les poids des arêtes à leurs valeurs par défaut.

— d'initialiser l'attribut correspondant au graphe du labyrinthe sous la forme d'une grille, utilisant les arguments width et height.

— d'afficher le graphe du labyrinthe à l'écran ; pour cela, elle fera appel au module render.py

L'idée est de générer notre graphe selon les quatre directions : haut, bas, gauche et droite. Pour assurer que le voisinage décrit naturellement le labyrinthe, on considère que si il existe un lien entre (a,b), il en est de même pour (b,a). On considérera donc que poids(a,b)=poids(b,a). Notre méthode revoie le graphe génération que l'on va passer à notre grille dans le constructeur.

```
def representlabyrinth (self, width, height) :
    graph = Graph()
    for x in range(width) :
        for y in range(height) :
            if y > 0 :
                down_link_direct = ((x,y),(x, y - 1))
                down_link_reciprocal = ((x, y - 1),(x,y))
                graph.add_arc(down_link_direct)
                graph.add_arc(down_link_reciprocal)
            if y < height - 1 :
                up_link_direct = ((x,y),(x, y + 1))
                up_link_reciprocal = ((x, y + 1),(x,y))
                graph.add_arc(up_link_direct)
                graph.add_arc(up_link_reciprocal)
            if x > 0 :
                left_link_direct = ((x,y),(x - 1, y))
                left_link_reciprocal = ((x - 1, y),(x,y))
                graph.add_arc(left_link_direct)
                graph.add_arc(left_link_reciprocal)
            if x < width - 1 :
                right_link_direct = ((x,y),(x + 1, y))
                right_link_reciprocal = ((x + 1,y),(x,y))
                graph.add_arc(right_link_direct)
                graph.add_arc(right_link_reciprocal)

    return graph
```

cette méthode revoie le graphe généré. qu'on va passer a notre grille self.mazegrid dans le constructeur.

```
def __init__(self, width, height) :

    self.width = width
    self.height = height
    self.mazegrid = self.representlabyrinth(width, height)
```

Pour l'affichage, on va utiliser les méthodes dans le module render

```
def showgraph (self, draw_coordinates) :

    draw_tree(self.mazegrid)

def drawlabyrinth (self) :

    draw_square_maze(self.mazegrid, self.path, draw_coordinates = False)
```

3 L'algorithme de Prim

Maintenant que nous disposons d'une grille vierge, nous allons générer un arbre couvrant. On vient créer un module coveringtree.py qui contiendra une méthode qui implémente l'algorithme de Prim. Cet algorithme calcule un arbre couvrant minimal dans un graphe connexe valué et non orienté.

Pour ce faire on utilisera la fonction 'extract_min' qu'on a étudié en TD pour obtenir le minimum.

```
def extract_min(F, distances):
    min_dst = math.inf
    min_s = None
    for s in F:
        if distances[s] <= min_dst:
            min_s = s
            min_dst = distances[s]
    F.remove(min_s)
    return min_s
```

Ensuite on implémente l'algorithme de Prim. Cet algorithme a pour but de calculer un arbre couvrant minimal pour notre graphe.

```
import math
from graph import *

def prim_algorithm (mazegraph):
    distances = {}
    parents   = {}
    for v in mazegraph.nodes() :
        distances[v] = math.inf
        parents[v] = None
    F = mazegraph.nodes()
    while ( len(F) != 0 ) :
        u = extract_min(F, distances)
        for v in mazegraph.successors(u):
            if v in F and mazegraph.arc_weight( (u, v) ) < distances[v] :
                distances[v] = mazegraph.arc_weight( (u, v) )
                parents[v] = u
    return parents
```

4 Génération d'un labyrinthe

Le but de cette partie est de compléter notre classe Maze afin qu'elle puisse générer un labyrinthe. Pour cela : On utilise l'algorithme de Prim avec le graphe support comme paramètre, donné par l'attribut initialisé dans le constructeur de la classe. On utilise le dictionnaire obtenu via Prim pour créer un nouveau graph qui représente cet arbre. Enfin on remplace l'attribut contenant le graphe du labyrinthe par le graphe construit précédemment.

```
def generatelabyrinth (self) :
    tree = prim_algorithm(self.mazegrid)
    maze = Graph()
    for node in tree.keys() :
        if node is not None and tree[node] is not None :
            maze.add_arc((node,tree[node]))
            maze.add_arc((tree[node], node))

    self.mazegrid = maze
```

5 Génération d'un labyrinthe aléatoire

A ce stade, nous pouvons générer un labyrinthe pour des dimensions fixes. Le but est maintenant de créer des labyrinthes aléatoires. Pour ce faire on instaure une nouvelle méthode dans la classe Maze qui sera chargée de générer via les paramètres height et width tel que les poids des arêtes soient aléatoires. Le poids sera compris entre 0 et 1 ; on utilisera la fonction uniform du module random pour prendre un poids aléatoire entre 0 et 1. En utilisant la méthode random.uniform, on fait en sorte qu'une variable reçoit random.uniform(0,1), qui est une valeur aléatoire comprise entre 0 et 1. On s'assure que pour toute arête (a,b) : poids(a,b)=poids(b,a).

```
def randomrepresentlabyrinth(self, width, height) :
    graph = Graph()
    for x in range(width) :
        for y in range(height) :
            if y > 0 :
                down_link_direct = ((x,y),(x, y - 1))
                down_link_reciprocal = ((x, y - 1),(x,y))
                weight = random.uniform(0, 1)
                graph.add_arc(down_link_direct, weight)
                graph.add_arc(down_link_reciprocal, weight)
            if y < height - 1 :
                up_link_direct = ((x,y),(x, y + 1))
                up_link_reciprocal = ((x, y + 1),(x,y))
                weight = random.uniform(0, 1)
                graph.add_arc(up_link_direct, weight)
                graph.add_arc(up_link_reciprocal,weight)
            if x > 0 :
                left_link_direct = ((x,y),(x - 1, y))
                left_link_reciprocal = ((x - 1, y),(x,y))
                weight = random.uniform(0, 1)
                graph.add_arc(left_link_direct,weight)
                graph.add_arc(left_link_reciprocal,weight)
            if x < width - 1 :
                right_link_direct = ((x,y),(x + 1, y))
                right_link_reciprocal = ((x + 1,y),(x,y))
                weight = random.uniform(0, 1)
                graph.add_arc(right_link_direct, weight)
                graph.add_arc(right_link_reciprocal, weight)

    return graph
```

Dans le constructeur, self.mazegrid reçoit maintenant le nouveau graph généré aléatoirement.

```
self.mazegrid = self.randomrepresentlabyrinth(width, height)
```

6 Résolution d'un labyrinthe

Maintenant que nous disposons de labyrinthes résolvons les. Bien que nos labyrinthes n'aient pas vraiment d'entrée ou de sortie, nous allons considérer le chemin du sommet (0, 0) vers le sommet (width − 1, height − 1) comme étant la solution de notre labyrinthe. Pour ce faire on incrémente notre constructeur de la classe Maze d'un attribut qui représentera la solution de notre labyrinthe. Cet attribut sera initialisé comme la liste vide. On ajoute ensuite une méthode dans la classe Maze qui résout le labyrinthe, et stocke la solution dans l'attribut correspondant. On modifie ensuite l'affichage afin d'y rajouter la solution en s'aidant du module render.py.

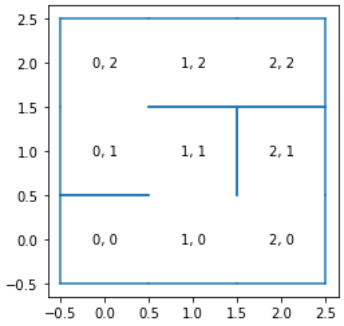
On utilise la méthode dfspath pour trouver le chemin entre le sommet $(0, 0)$ et $(width - 1, height - 1)$. L'idée est simple, on utilise l'algorithme du parcours en profondeur et une structure de données PILE (dernier arrivé, premier sorti).

```python
def dfspath(self, current, objective) :

    temp = [[current]]
    visited = []

    while temp :
        path = temp.pop(0)
        node = path[-1]
        if node == objective :
            return path
        for neighbor in self.mazegrid.successors(node) :
            if neighbor not in visited :
                newpath = path + [neighbor]
                temp.insert(0, newpath)
                visited.append(neighbor)
```

Prenons un exemple de $3 * 3$. pour le graphe suivant :



Une grille de taille $3 * 3$

La liste *temp* va avoir les valeurs suivantes :

```
[[(0, 0)]]
[[(0, 0), (1, 0)]]
[[(0, 0), (1, 0), (0, 0)], [(0, 0), (1, 0), (2, 0)], [(0, 0), (1, 0), (1, 1)]]
[[(0, 0), (1, 0), (2, 0)], [(0, 0), (1, 0), (1, 1)]]
[[(0, 0), (1, 0), (2, 0), (2, 1)], [(0, 0), (1, 0), (1, 1)]]
[[(0, 0), (1, 0), (1, 1)]]
[[(0, 0), (1, 0), (1, 1), (0, 1)]]
[[(0, 0), (1, 0), (1, 1), (0, 1), (0, 2)]]
[[(0, 0), (1, 0), (1, 1), (0, 1), (0, 2), (1, 2)]]
[[(0, 0), (1, 0), (1, 1), (0, 1), (0, 2), (1, 2), (2, 2)]]
```

Valeurs de temps pour la grille $3 * 3$

O a crée une autre méthode $defindpath$ qui a pour but de modifier $self.path$ afin qu'il reçoit $path$ généré par la méthode $dfspath$. (dans main.py,on fait appel à cette méthode).

```python
def defindpath(self) :
    current = (0,0)
    objective = (self.width - 1, self.height - 1)
    self.path = self.dfspath(current, objective))
```

Une autre méthode de le faire est d'utiliser une fonction récursive en construisant deux listes : la liste des sommets visités et La liste qui contient de chemin du sommet initial au sommet final. Cette méthode renvoie un booléen : True si le chemin a été trouvé, False sinon. (Méthode proposé par Monsieur PERROTIN Pacome).

```
def explorepath (self, current, objective, visited_list, path) :

    if current == objective :
        path.append(objective)
        return True
    if current in visited_list :
        return False
    visited_list.append(current)
    path.append(current)

    for neighbor in self.mazegrid.successors(current):
        if self.explorepath(neighbor, objective, visited_list, path) == True :
            print(neighbor)
            return True

    path.remove(current)
    return False
```

Dans une autre méthode, on passe $self.path$ comme paramètre et on aura notre chemin.

```
def calculpath(self) :
    current = (0,0)
    objective = (self.width - 1, self.height - 1)
    self.explorepath(current, objective, [], self.path)
```

7 Génération d'un labyrinthe biaisé

Le but de cette partie est de transformer le rendu de nos labyrinthes. Pour ce faire on crée deux nouvelles fonctions dans Maze. L'une génère une grille, et prend en paramètre supplémentaire une valeur entre 0 et 1, qui représente le biais qui sera rajouté au poids des arêtes verticales. La seconde fonction fait de même, mais rajoute ce biais au poids des arêtes horizontale. Par l'action de ce biais, les arêtes verticales (respectivement horizontales) seront privilégiées par l'algorithme de Prim.

Pour ce faire, on crée deux méthodes *verticallabyrinth* et *horizontallabyrinth* afin de modifier le poids entre deux sommets.

La méthode prends en paramètre *distortion* que l'on va ajouter aux poids selon les conditions suivantes :

On parcourt les sommets de notre graphe, et pour chaque sommet, on parcourt ses successeurs. Si la première composante du sommet et de ses voisins est la même et la deuxième composante est différente; on ajoute la distorsion au poids entre ces deux voisins.

Pareil pour l'autre méthode mais en considérant l'égalité pour la deuxième composante, et l'inégalité pour la première composante.

```
def verticallabyrinth (self, distortion) :
    self.generatelabyrinth()
    for node in self.mazegrid.nodes():
        for neighbor in self.mazegrid.successors(node) :
            if node[0] == neighbor[0] and node[1] != neighbor[1] :
                self.mazegrid.set_arc_weight(
                (node, neighbor), self.mazegrid.arc_weight(
                (node, neighbor)) + distortion)
```

```
def horizontallabyrinth (self, distortion) :
    self.generatelabyrinth()
    for node in self.mazegrid.nodes():
        for neighbor in self.mazegrid.successors(node) :
            if node[0] != neighbor[0] and node[1] == neighbor[1] :
                self.mazegrid.set_arc_weight(
                (node, neighbor), self.mazegrid.arc_weight(
                (node, neighbor)) + distortion)
```

8 Création de labyrinthe héxagonaux

Enfin dans cette partie, on tentera de faire apparaître un labyrinthe de forme hexagonal. Pour ce faire on crée un module hexagonalmaze.py qui définit une classe HexagonalMaze ayant un comportement similaire à Maze. La différence est qu'ici nous allons générer des graphes supports de forme hexagonale. Les coordonnées des sommets d'une grille hexagonales sont toujours les coordonnées de la forme (x, y), pour x $\in 0, ..., width - 1$ et $y \in 0, ..., height - 1$. Seul leur voisinage change. On modifie enfin notre module main.py pour utiliser la classe HexagonalMaze plutôt que Maze.

On importe les modules nécessaires.

La classe *Hexagonalmaze* hérite de la classe *Maze* donc hérite *width* et *height* ainsi que les différentes méthodes que l'on a crée.

Il nous reste qu'à crée la méthode *drawhexlabyrinth* qui a pour but d'afficher notre labyrinthe.

```
from maze import Maze
from graph import Graph
from render import draw_hex_maze

class Hexagonalmaze(Maze) :

    def __init__(self, width, height):
        super().__init__(width, height)

    def drawhexlabyrinth(self) :
        draw_hex_maze(self.mazegrid, self.path, draw_coordinates = False)
```

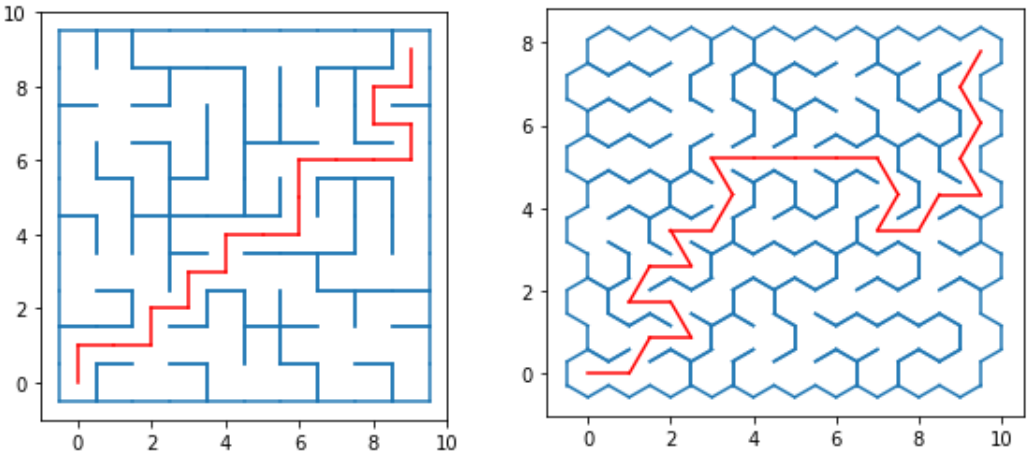
9 Application et Exemples

Dans notre fichier *main.py*, on fait appelle aux différentes classes et méthodes pour générer les labyrinthes.

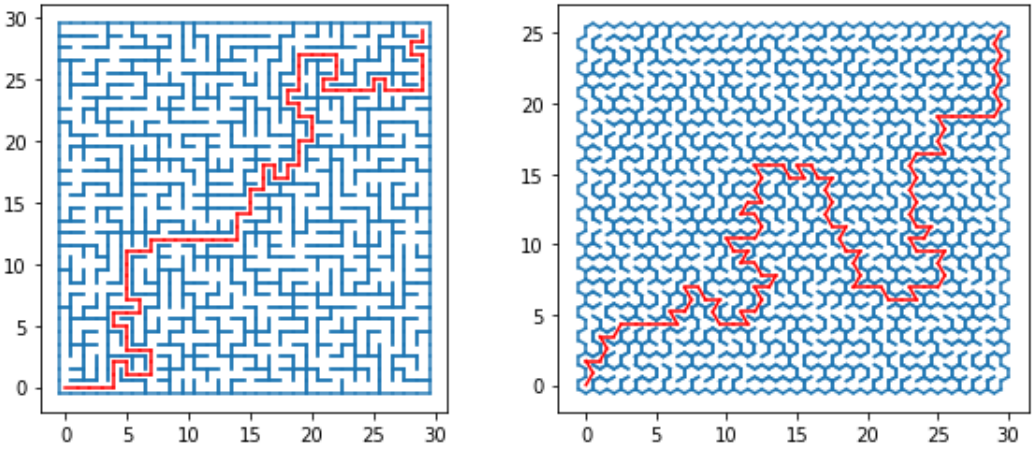
```
from maze import Maze
from hexagonalmaze import *

labyrinth = Maze(30, 30)
labyrinth.generatelabyrinth()
labyrinth.defindpath()
labyrinth.verticallabyrinth(0.5)
labyrinth.horizontallabyrinth(1)
labyrinth.drawlabyrinth()

labyrinthhex = Hexagonalmaze(30, 30)
labyrinthhex.generatelabyrinth()
labyrinthhex.defindpath()
labyrinthhex.drawhexlabyrinth()
```



Labyrinthes carré et hexagonal résolus de taille $10 * 10$



Labyrinthes carré et hexagonal résolus de taille $30 * 30$