Aix-Marseille Université
L3 Mathématiques

UE Programmation 3

Projet Mazes

1 Modalités de rendu

Ce projet d'informatique conclut les séances de Programmation 3. Il est à rendre pour le Jeudi 16 décembre à 23h59 et peut-être fait en monôme ou en binôme. Le rendu se fait par Amétice, sous la forme d'une archive. Cette archive doit contenir votre code, ainsi qu'un léger compte-rendu qui précise comment utiliser votre programme.

Nous vous rappellons que tout plagiat de code d'autres groupes ou de code trouvé en ligne impliquera une note de 0 pour tous les groupes impliqués. Tout code copié doit faire référence à la source, que ce soit un autre élève, un site web, ou autre.

2 Présentation du projet

Dans ce projet, nous allons générer des labyrinthes. Plutôt que d'utiliser la méthode la plus simple reposant sur un simple parcours en profondeur du graphe support, nous allons utiliser un algorithme plus sophistiqué, et générer des arbres couvrants de poids minimal.

Dans sa totalité, ce projet vous demande :

— De générer le graphe non-orienté correspondant à une grille carrée, et de pouvoir la générer avec des poids aléatoires sur les arêtes.
— D'implémenter l'algorithme de Prim, qui génère un arbre couvrant de poids minimal sur un graphe.
— D'utiliser le code fourni pour afficher l'arbre couvrant obtenu sous la forme d'un labyrinthe.
— De faire varier la génération des poids de la grille initiale pour obtenir des labyrinthes plus horizontaux, ou plus verticaux.

Une fois ces tâches réalisées et s'il vous reste du temps, vous pourrez faire de même pour générer des labyrinthes de formes hexagonales ; l'algorithme de génération est le même que pour un labyrinthe carré, seule varie la structure de la grille qui supporte le graphe du labyrinthe.

Dans l'intégralité de ce projet, c'est à vous de choisir comment nommer vos variables et vos méthodes ; c'est en général à vous de choisir comment présenter votre code. Un code difficile à lire est un code qui donne moins de points qu'un code facile à lire ; comme vous le savez, les meilleurs noms de variables et de fonctions nous informent sur leur rôle dans le programme, et sont généralement choisis en anglais pour correspondre au reste du language.

2.1 Un labyrinthe est un arbre couvrant une grille

Les labyrinthes sont souvent générés sur une grille ; il existe cependant d'autres types de labyrinthes, comme des labyrinthes circulaires ou générés depuis des grilles hexagonales, ou même des graphes quelconques.

Dans tout les cas, un labyrinthe est généré depuis un graphe support. Dans le cas d'un labyrinthe carré, ce support est une grille. Pour générer le labyrinthe, nous allons générer un arbre couvrant du graphe support. Un arbre couvrant est un graphe connexe défini sur les mêmes sommets que le graphe support, mais qui ne dispose d'aucun cycle.

Dans ce projet, nous allons générer des arbres couvrant de poids minimal, ce qui nous permettra de jouer avec la génération de labyrinthe en faisant varier la distribution initiale des poids. L'algorithme que nous allons implémenter est l'algorithme de Prim.

2.2 Code fourni

Pour vous aider à faire ce projet, nous vous fournissons deux modules Python :

— graph.py est un module définissant une classe modélisant un graphe pondéré (c'est à dire avec des poids sur les arêtes), sous la forme d'une liste d'adjacence. Il s'agit en substance de la correction du TD2. Vous êtes libres d'ajouter des méthodes à cette classe lorsque cela vous semble utile.

— render.py est un module qui vous donne accès à des fonctions d'affichage, qui utilisent la librairie pyplot. Pour savoir comment s'en servir, lisez les commentaires présent dans le code ; ils vous serviront de documentation.

Graphes orientés vs graphes non-orientés : la classe Graph vue en TD, et rédigée dans le module graph.py, encode les arêtes du graphe sous la forme d'une liste d'adjancence. Cela permet d'encoder des graphes orientés (où une arête de a vers b ne signifie par forcément qu'il existe une arête de b vers a). Dans le reste de ce projet, nous considérons nos graphes non-orientés ; c'est à dire que pour toute arête de a vers b, il existe également une arête de b vers a, et les deux arêtes ont le même poids. C'est à vous de vous assurer que les grilles et les arbres que vous générez sont bien non-orientés.

3 Tâche 1 : classe Maze (2 points)

Pour démarrer, créez le module maze.py qui va contenir la classe Maze. Cette classe aura pour responsabilité de représenter un labyrinthe ; elle doit donc contenir le graphe du labyrinthe, et permettre son affichage.

Pour l'instant nous ne générons pas de labyrinthes. Pour cette première tâche, écrivez la classe Maze de façon à ce que :

— Elle dispose d'une méthode qui prend en paramètre une hauteur et une largeur (en anglais width et height), et qui génère puis retourne un graphe non-orienté (en utilisant la classe Graph) correspondant à la grille avec ces dimensions. Pour l'instant, on laissera le poids des arêtes à leur valeur par défaut.

— Elle dispose d'un constructeur qui prend en paramètre une largeur (width) et une hauteur (height), et qui initialise l'attribut correspondant au graphe du labyrinthe sous la forme d'une grille.

— Elle dispose d'une fonction qui affiche le graphe du labyrinthe à l'écran ; pour cela, faites appel à la fonction correspondante du module render.py .

Les sommets de la grille sont les coordonnées entières de la grille. Ainsi, pour générer une grille de dimensions 2 par 2, les sommets du graphe seront $(0,0)$, $(0,1)$, $(1,0)$ et $(1,1)$. Le sommet $(0,0)$ est connecté aux sommets $(0,1)$ et $(1,0)$, le sommet $(0,1)$ aux sommets $(0,0)$ et $(1,1)$, le sommet $(1,0)$ aux sommets $(1,1)$ et $(0,0)$, et le sommet $(1,1)$ aux sommets $(1,0)$ et $(0,1)$. Il est important de suivre cette nomenclature, car sinon les fonctions du module render.py ne fonctionneront pas correctement.

Testez votre classe pour plusieurs dimensions de grille. L'affichage à l'écran devrait donner un grand rectangle qui entoure votre labyrinthe ; pensez à activer l'affichage des coordonnées du labyrinthe pour y voir plus clair. Pour afficher les coordonnées, lisez les commentaires de documentation du module render.py .

4 Tâche 2 : l'algorithme de Prim (6 points)

Maintenant que nous disposons d'une grille vierge, nous allons générer un arbre couvrant. Pour cette tâche, créez un module coveringtree.py, dans lequel vous devez créer une méthode qui implémente l'algorithme de Prim. Son pseudo-code est représenté par l'algorithme 1.

5 Tâche 3 : génération d'un labyrinthe (2 points)

Ajouter une méthode à la classe Maze qui génère un labyrinthe. Pour cela, elle doit suivre les étapes suivantes :

1. Faire appel à l'algorithme de Prim, avec pour paramètre le graphe support du labyrinthe, donné par l'attribut initialisé dans le constructeur de la classe.
2. Utiliser le dictionnaire obtenu à l'étape précédente pour créer un nouveau graphe (de la classe Graph) qui représente cet arbre. Ce graphe doit être non-orienté, mais les poids des arêtes ne sont pas importants.
3. Remplacer l'attribut contenant le graphe du labyrinthe pas le graphe construit à l'étape précédente.

Pour tester votre travail jusqu'ici, créez un module main.py qui crée un objet maze (une instance de la classe Maze) d'une taille 10 par 10, génère son labyrinthe et l'affiche à l'écran. Si le résultat ne semble pas juste, vous

Algorithm 1 Algorithme de Prim, pour $G = (S, A)$ un graphe.

$C \leftarrow$ un dictionnaire qui pour chaque sommet v associera le coût de connexion de v à l'arbre
$P \leftarrow$ un dictionnaire qui pour chaque sommet v associera le parent de v dans l'arbre généré
for $v \in S$ **do**
  $C[v] = \inf$
  $P[v] = \emptyset$
end for
$Q \leftarrow S$
while Q n'est pas vide **do**
  $u \leftarrow$ un sommet dans Q tel que $C[u]$ est minimal
  $Q \leftarrow Q \setminus \{u\}$
  for tout v successeur de u dans G **do**
    if $v \in Q$ et $weight(u, v) < C[v]$ **then**
      $C[v] \leftarrow weight(u, v)$
      $P[v] \leftarrow u$
    end if
  end for
end while
Retourner P

pouvez suivre la documentation du module render.py pour afficher les coordonnées des sommets du graphe ainsi que d'afficher directement l'arbre plutôt que le labyrinthe.

6 Tâche 4 : génération d'un labyrinthe aléatoire (3 points)

La tâche précédente permet de générer un labyrinthe. Félicitations ! Cependant, pour des dimensions fixes, le labyrinthe obtenu sera toujours le même ; nous n'avons jamais inclu l'aléatoire dans sa génération.

Pour corriger ce souci, ajoutez une méthode dans la classe Maze qui génère une grille de largeur et hauteur passées en paramètres, telle que les poids des arêtes soient aléatoires. Nous allons considérer ce poids comme une valeur comprise entre 0 et 1 ; en Python, on obtient une telle valeur grâce à à l'appel de la méthode uniform(0, 1), du module standard random. **Attention !** Pour toute paire de sommets a et b, le poids de l'arête (a, b), si elle existe, doit toujours être le même que celui de l'arête (b, a).

Pour tester cette tâche, modifiez le constructeur de la classe Maze pour utiliser cette nouvelle méthode de génération de grille, plutôt que l'ancienne. L'exécution de votre module main.py devrait maintenant générer un labyrinthe différent à chaque exécution.

7 Tâche 5 : résolution d'un labyrinthe (4 points)

Maintenant que nous disposons de labyrinthes à la demande, nous pouvons facilement les résoudre ! 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 $(width - 1, height - 1)$ la solution de notre labyrinthe.

Dans cette tâche, ajoutez au constructeur de la classe Maze la définition d'un attribut qui représentera la solution de notre labyrinthe. Cet attribut sera initialisé comme la liste vide.

Ajoutez une méthode dans la classe Maze qui résout le labyrinthe, et stocke la solution dans l'attribut correspondant. Résoudre un labyrinthe repose sur une simple exploration en profondeur, jusqu'à arriver au sommet final.

Modifiez la fonction d'affichage de votre classe Maze pour prendre en compte cette solution. Pour ce faire, consultez les commentaires de la documentation du module render.py . Un exemple d'affichage est présenté en figure 1.

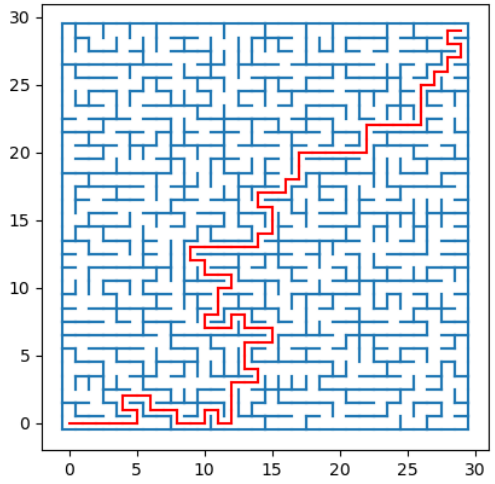

Figure 1: Un labyrinthe affiché par le module render.py, avec sa résolution.

8 Tâche 6 : génération d'un labyrinthe biaisé (3 points)

Générer un labyrinthe se fait très facilement ; en vérité, il suffit de procéder à une exploration aléatoire en profondeur du graphe support. Notre méthode plus compliquée a cependant un avantage : nous allons pouvoir facilement modifier notre génération pour transformer le rendu de nos labyrinthes.

Pour accomplir cette tâche, rajouter deux fonctions à la classe 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 (resp. horizontales) seront privilégiées par l'algorithme de Prim. Ainsi, un biais de 0 ne fera aucune différence, mais un biais vertical de 0.5 rendra les transition verticales bien plus chères par rapport au arêtes horizontales, et donc choisies moins souvent. Un biais de 1 rendra toutes les transitions de la direction choisie nécessairement plus chères que les transitions de l'autre direction ; ainsi, un biais horizontal de 1 nous donnera un labyrinthe avec un nombre minimal de transitions horizontales.

Un exemple d'affichage est présenté en figure 2.

9 Tâche Bonus : création de labyrinthes hexagonaux (+5 points)

Dans cette tâche bonus, nous allons utiliser l'avantage d'avoir travaillé sur des graphes (et non sur des grilles), et changer la topologie du graphe support de notre labyrinthe.

Pour faire cette tâche, rédigez un module hexagonalmaze.py qui définit une classe HexagonalMaze. Son comportement est pratiquement identique à celui de la classe Maze. Cependant, plutôt que de génerer des graphes supports de la forme de grilles carrées, 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 $x \in \{0, \ldots, width - 1\}$ et $y \in \{0, \ldots, height - 1\}$. Seul leur voisinage change. Pour pouvoir savoir quel voisinage donner à un sommet, créez une grille vide (sans arêtes) et affichez là avec les coordonnées sous forme hexagonale à l'aide du module render.py .

Une fois la grille générée, et l'affichage adapté pour utiliser les fonctions affichant les grilles hexagonales, modifiez votre module main.py pour utiliser la classe HexagonalMaze plutôt que Maze, et admirez le résultat. Un exemple d'affichage est présenté en figure 3.

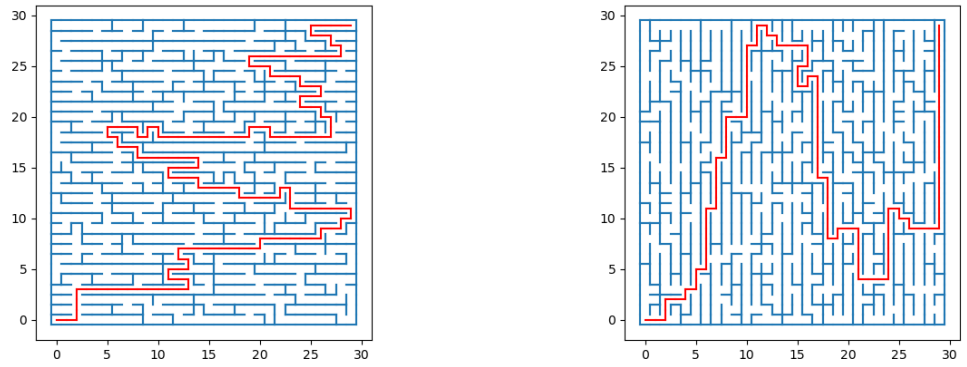

Figure 2: Deux labyrinthes dont les poids ont été biaisés pour être plus coûteux dans une direction. Les arêtes verticales du labyrinthe de gauche et les arêtes horizontales du labyrinthe de droite ont été alourdies de 0.5.

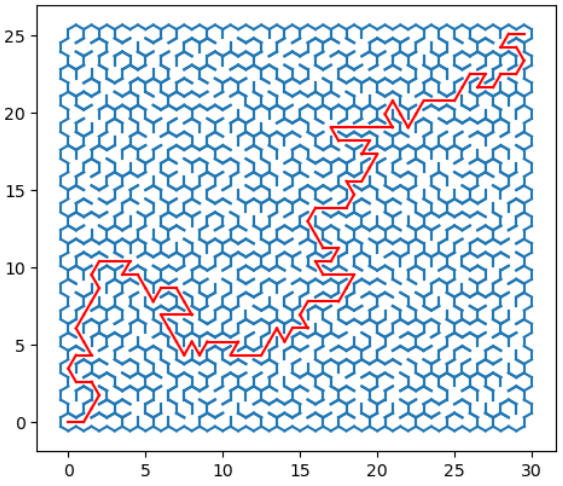

Figure 3: Un labyrinthe hexagonal affiché par le module render.py, avec sa solution.