UE Programmation 3

Projet Mazes

1 Présentation du projet

Dans ce projet, nous allons générer et afficher 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 rédiger des fonctions permettant la gestion d'un graphe pondéré non-orienté.
— 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'implémenter un algorithme de parcours en profondeur pour générer la solution d'un labyrinthe.
— D'utiliser la librairie matplotlib pour afficher un labyrinthe et sa solution sous forme de diagramme.
— 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 ferez 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.

Enfin, il est fortement conseillé d'organiser le code en plusieurs fichiers `.py`, par exemple un par tâche, et de les importer lorsque nécessaire.

1.1 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 Tâche 1 : graphes pondérés non-orientés

Avant toute chose, il est important de programmer les fonctions qui nous permettent de structurer un graphe pondéré non-orienté. Pour rappel, un graphe non-orienté est un graphe dont toutes les arcs $u \to v$ sont accompagnés de leur symmétrique $v \to u$. Une telle paire d'arcs est une arête. Ainsi, un graphe non-orienté est composé d'arêtes (edge), là ou un graphe orienté est composé d'arêtes et d'arcs (arcs). Un graphe pondéré non-orienté est un graphe non-orienté dans lequel un nombre est associé à chaque arête.

Pour ce projet, nous vous conseillons de rédiger un module `graph.py` réservé aux fonctions sur nos graphes. Pour encoder un graphe, nous vous conseillons de les représenter sous la forme d'une liste d'adjacence. Plus particulièrement, cette liste pourra être encodée par un dictionnaire, qui à chaque sommet du graphe associe la liste de ses successeurs.

Pour représenter les poids des arêtes du graphe, on pourra utiliser un second dictionnaire qui à chaque arête dans le graphe associe son poids. Pour rappel, les dictionnaires python peuvent utiliser n'importe quel type immutable comme clé. Un tuple qui représente une paire de sommets est un tel exemple de type immutable.

Pour réaliser cette tâche, nommer, programmer et tester les méthodes suivantes. L'ensemble de ces fonctions doivent prendre en paramètre la liste d'adjacence et/ou le dictionnaire des poids du graphe, selon leurs besoins.

- Une méthode qui renvoie la liste d'adjacence d'un graphe vide.
- Une méthode qui ajoute un sommet donné au graphe. Un sommet fraichement ajouté dans un graphe est initialisé avec un liste vide de successeurs. Attention, si le sommet est déjà présent avant l'ajout il ne faut pas effacer les successeurs déjà présents.
- Une méthode qui ajoute un arc au graphe. Cet arc est donné comme une paire de sommets. Assurez vous que les deux sommets font bien partie du graphe avant de les connecter dans la liste d'adjacence.
- Une méthode qui ajoute une arête au graphe. Une arête entre u et v est composée de deux arcs, $u \to v$ et $v \to u$.
- Une méthode qui pour une paire de sommets u, v retourne vrai si et seulement si il existe un arc $u \to v$.
- Une méthode qui pour une paire de sommets u, v retourne vrai si et seulement si il existe une arête entre u et v.
- Une méthode qui renvoie une copie de la liste des sommets. Rappel : pour d un dictionnaire, on peut récupérer une liste de ses clés avec `list(graph.keys)`.
- Une méthode qui renvoie la liste des successeurs d'un sommet.
- Une méthode qui affecte le poids d'un arc.
- Une méthode qui affecte le poids d'une arête. Cette méthode doit donc appliquer ce poids aux deux directions.
- Une méthode qui renvoie le poids d'un arc.
- Une méthode qui combine la création d'une arête et l'affectation de son poids.

3 Tâche 2 : affichage simple

Maintenant que nous disposons de quoi gérer un graphe pondéré non-orienté, nous allons mettre tout cela en pratique en affichant nos graphes à l'écran à l'aide de la librairie matplotlib.pyplot. Il est préférable de commencer à écrire l'ensemble des fonctions de cette tâche sous leur propre module (render.py, par exemple). Ce module commence en important la librairie, avec la ligne :

```
import matplotlib.pyplot as plot
```

Cette librairie est principalement employée pour l'affichage de graphes dans le sens plus numérique du terme, c'est à dire pour faire le graphe de fonctions ou pour illustrer des statistiques. Il s'agit cependant d'une librairie bien plus générale dans ses possibilités, et nous l'utiliserons dans ce projet pour afficher nos labyrinthes.

Pour afficher quelque chose, on ajoute des éléments à notre diagramme avec des appels de la forme :

```
# Exemple qui ajoute au digramme un segment entre les points (x1, y1)
# et (x2, y2). Notez que les coordonnées ne sont pas dans l'ordre naturel
# dans l'appel à la fonction. On peut changer la couleur avec le paramètre color,
# pour plus de détails voir la documentation :
# https://matplotlib.org/2.0.2/users/colors.html
plot.gca().add_line( plot.Line2D( (x1, x2), (y1, y2), color = 'r') )
```

Une fois satisfaits de notre diagramme, on l'affiche avec :

```
plot.axis('scaled') # permet d'être sûrs de bien cadrer notre diagramme
plot.show()
```

Affichage d'un graphe. Afin de pouvoir afficher nos graphes en utilisant ces primitives, nous allons utiliser des coordonnées comme noms de nos sommets. Par exemple, on pourra ajouter une arête entre le sommet $(0, 0)$ et le sommet $(0, 1)$. Cela fonctionne car on peut utiliser tout type immutable (qui ne peut pas être modifié après

initialisation) comme clé d'un dictionnaire ; utiliser un tuple de coordonnées comme clé est donc parfaitement valable. Cela permettra de rendre l'affichage plus pratique à rédiger.

Pour réaliser cette tâche, nommez, rédigez et testez les méthodes suivantes :

— Une méthode qui ajoute un segment à l'image. Cette méthode doit prendre deux points, qui sont des tuples de la forme (x, y), et une couleur.
— Une méthode qui affiche le diagramme à l'écran.
— Une méthode qui prend un graphe en paramètre, et qui affiche ce graphe à l'écran. Pour cela, ajouter un segment pour chaque arête dans le graphe.

Ces méthodes simples ne nous permettent pas d'afficher un labyrinthe pour le moment, mais elles vont nous permettre de se faire une idée rapide de ce qu'il se passe lorsque nous générons des arbres couvrants.

4 Tâche 3 : génération d'une grille

Comme d'habitude, les fonctions de cette tâche sont à rédiger dans leur propre module.

Dans cette tâche nous allons générer des grilles ; une grille peut être vue comme un graphe dont les coordonnées sont des points du plan. Nous allons donc créer des grilles aux dimensions données sous la forme de graphes.

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 d'affichage ne pourront pas fonctionner.

Cette tâche vous demande de rédiger :

— Une méthode qui prend en paramètre une hauteur et une largeur (en anglais width et height), et qui génère puis retourne la liste d'adjancence d'un graphe non-orienté et un dictionnaire de poids correspondant à la grille aux dimensions spécifiées. Pour l'instant, le poids de chaque arête sera de 1.
— Une méthode qui prend en paramètre une hauteur et une largeur et qui génère puis retourne la liste d'adjancence d'un graphe non-orienté et un dictionnaire de poids correspondant à la grille aux dimensions spécifiées, cette fois en associant à chaque arête un poids compris entre 0 et 1. Une telle génération se fait en utilisant `random.uniform(0, 1)`, avec `random` une variable précédemment initialisée avec `random = Random()`. Pensez également à inclure `from random import Random` en début de module.

Testez vos méthodes en utilisant matplotlib pour plusieurs dimensions de grilles.

5 Tâche 4 : l'algorithme de Prim

Maintenant que nous disposons d'une grille vierge, nous allons générer un arbre couvrant cette grille. Pour cette tâche, créez un module qui hébergera les méthodes autour de l'implémentation et l'application de l'agorithme de Prim. Son pseudo-code est présenté par l'algorithme 1. Cette tâche est de loin la partie la plus technique de ce projet.

À propos du nommage des variables d'un pseudocode. L'algorithme de Prim vous est présenté sous la forme d'un pseudocode ; ces pseudocodes sont une forme de version abstraite d'un langague de programmation. Ils trouvent leur place sur un sujet d'examen ou dans un livre d'algorithmique. C'est pour cela que les variables ont une nomenclature proche des mathématiques : des variables aux noms abbréviés, généralement composés d'une lettre. **Ce ne sont pas des noms valables pour un algorithme lorsqu'il est rédigé dans un vrai language de programmation**. En effet, les noms de variables d'un programme doivent être facilement compris à la lecture. Il vous faut donc renommer les variables de ce pseudocode sous une forme correcte pour un programme python.

Avant de l'implémenter, exécutez-le sur le papier sur une grille simple de taille 2 par 2, avec des poids aléatoires sur les arêtes.

Remarquez que cette algorithme ne renvoie pas un graphe, mais un dictionnaire qui associe son parent à chaque sommet.

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

Pour cette tâche, implémentez :

— Une méthode qui extrait l'élément de coût minimal parmis une liste de choix. Cette méthode est utilisée à la première ligne du while, dans le pseudocode.
— Une méthode qui réalise l'algorithme de Prim tel que fourni.
— Une méthode qui prend en paramètre un graphe et un dictionnaire de poids, et qui retourne un graphe représentant un arbre couvrant le graphe passé en paramètre. Cette fonction doit évidemment faire appel à la précédente, et doit construire l'arbre obtenu à partir du dictionnaire des parents.

Pour tester cette méthode, utilisez les fonctions d'affichages pour afficher l'arbre couvrant obtenu sous la forme d'un diagramme. Cet arbre doit être connexe (il ne doit pas être découpé en plusieurs sections) et ne doit pas comporter de cycles.

6 Tâche 5 : affichage d'un labyrinthe

Maintenant que tout est prêt, nous allons implémenter un affichage de notre labyrinthe à proprement parler en ajoutant des méthodes au module entamé en tâche 2.

Pour cela, plutôt que simplement afficher les arêtes du graphe, nous allons afficher l'ensemble des cellules qui composent le labyrinthe. Étant donné que le centre de chaque cellule est à une coordonnée entière, nous allons dessiner les murs qui séparent les cellules comme des segments intermédiaires. Ainsi, si la cellule $(0, 1)$ n'est pas connectée à la cellule $(1, 1)$, on dessinera un segment de $(0.5, 0.5)$ vers $(0.5, 1.5)$.

Implémentez :

— Une méthode qui pour un deux coordonnées affiche le mur qui les sépare.
— Une méthode qui pour un sommet donné retourne l'ensemble des coordonnées vers lesquels il doit y avoir un mur.
— Une méthode qui pour un sommet donné affiche la cellule de ce sommet.
— Une méthode qui affiche l'ensemble des cellules d'un labyrinthe donné.

7 Tâche 5 : un main.py propre

Dans cette section, créez un module main.py qui utilise tout le code fait jusqu'ici pour générer et afficher un labyrinthe.

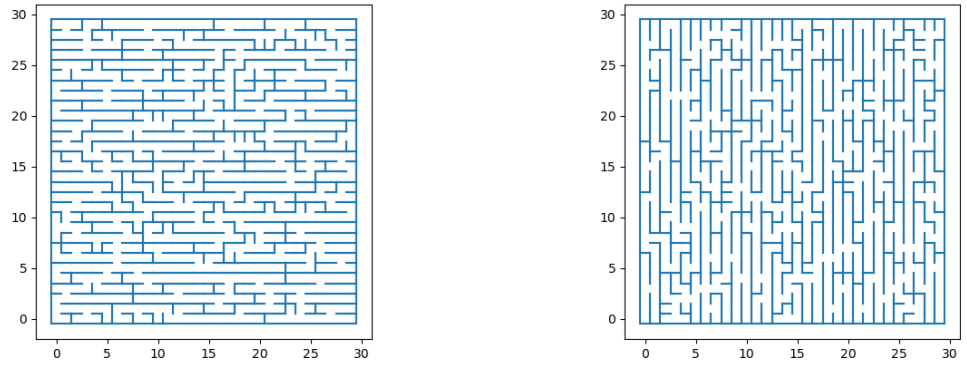

Figure 1: Deux labyrinthes dont les poids ont été biaisés pour être plus coûteux dans une direction. Les arêtes verticales du labyrinthe de gauche et les arêtes horizontales du labyrinthe de droite ont été alourdies de 0.5.

8 Tâche 6 : labyrinthe biaisés

Dans cette tâche nous allons ajouter des méthodes au module implémenté en tâche 3.

Ces méthodes permettent l'affichage de labyrinthes dans lesquels les directions horizontale ou verticales seront privilégiées. Implémentez :

— Une méthode qui prend en paramètre des dimensions et un biais, qui est une valeur entre 0 et 1. Cette méthode génère une grille aux dimensions données dont les poids sont générés de façon uniforme entre 0 et 1, sauf pour les arêtes horizontales qui sont générées par la somme entre une valeur prise entre 0 et 1, et le biais.

— Une méthode équivalente à la précédente, mais en vertical.

9 Tâche 7 : résolution

Pour un labyrinthe donné, on peut facilement générer le chemin qui le résoud à l'aide d'un parcours en profondeur. On considérera que le chemin résolvant le labyrinthe est celui qui relie la coordonnée `(0, 0)` à la coordonnée `(width - 1, height - 1)`. Implémentez cette résolution et modifiez votre affichage pour afficher cette résolution comme en figure 2.

10 Tâche Bonus : création de labyrinthes hexagonaux

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 dont le rôle est pratiquement identique à celui de votre imlémentation de la tâche 3 ; cependant, plutôt que de génerer des graphes supports de la forme de grilles carrées, générez 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.

Il faudra également créer de nouvelles fonctions d'affichages. Un exemple de résultat est illustré en figure 3.

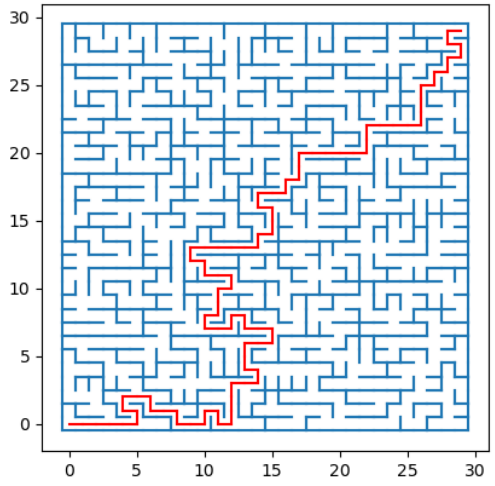

Figure 2: Un labyrinthe affiché avec sa résolution.

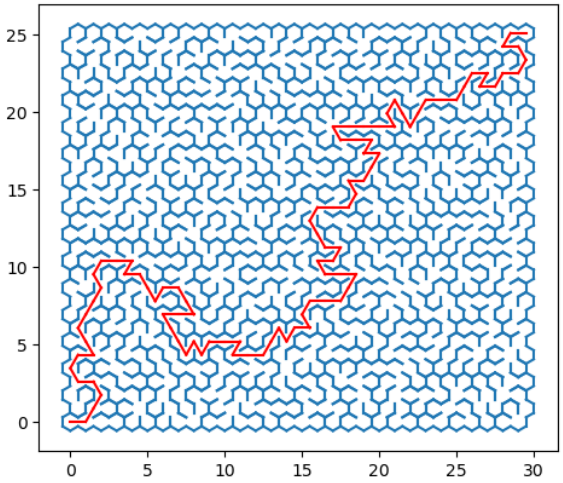

Figure 3: Un labyrinthe hexagonal affiché avec sa solution.