TP 15. Parcours de graphe : génération d'un labyrinthe.

1 Principe du TP

L'objectif de ce TP est de générer un labyrinthe. Un labyrinthe est un graphe qui contient tous les sommets de la grille et un certain nombre d'arêtes pour les relier. Pour obtenir un labyrinthe aléatoirement on réalise un parcours de la grille. Pour cela il faut visiter l'ensemble des sommets de la grille et conserver les chemins qui ont permis cette découverte. Sommets et arêtes seront stockés dans un graphe appelé labyrinthe.

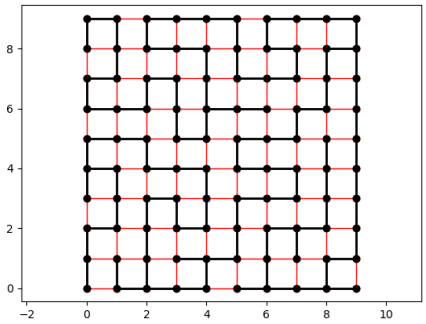

FIGURE 1 – Exemple de labyrinthe obtenu sur une grille 10*10

Viendra ensuite le moment de résoudre ce labyrinthe : ce sera l'objectif de la dernière partie. Il faudra alors être capable de trouver le chemin qui permet d'aller du coin inférieur gauche (départ) au coin supérieur droit (arrivée) en n'empruntant que les lignes (arêtes) du labyrinthe.

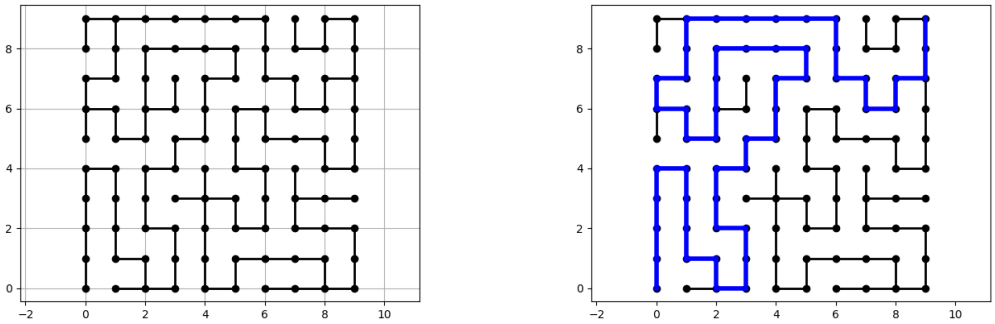

FIGURE 2 – Un labyrinthe et ce même labyrinthe résolu

2 Préambule : génération d'une grille, affichage, test des fonctions proposées

Le fichier "TP15.py" contient des fonctions permettant de créer une grille, tracer une grille, ajouter des arrêtes à une grille... L'objectif de cette partie est de prendre connaissance de ces fonctions, d'apprendre à les utiliser, et de comprendre la structure des données manipulées.

Soit une grille rectangulaire $n \times p$ constituée de n colonnes et de p lignes contenant toutes les arêtes possibles. On modélise cette grille par un graphe dont l'ensemble des sommets est donné par les couples (i, j) tels que : $i \in [\![0, n[\![$ et $j \in [\![0, p[\![$.
Les voisins d'un sommet (i, j) sont ceux situés en haut, en bas, à droite et à gauche s'ils existent (par exemple, le sommet $(0, 0)$ a comme voisin les sommets $(0, 1)$ et $(1, 0)$).

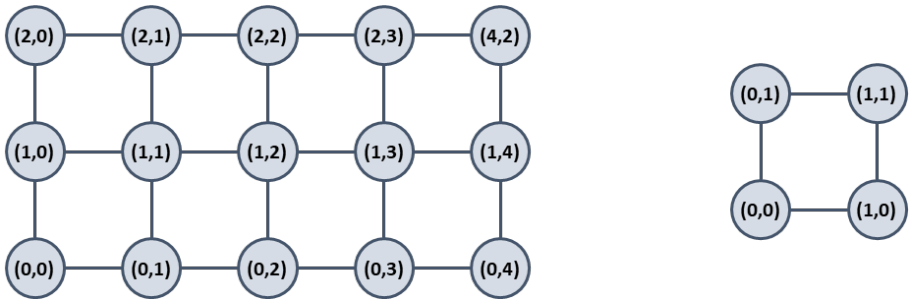

FIGURE 3 – Grille (5,3) et grille (2,2)

Le graphe est implémenté par un dictionnaire d'adjacence où les clés sont les tuples, coordonnées d'un sommet. La valeur associée est une liste des sommets voisins.

Exemple

La grille 2×2 sera modélisée par le graphe suivant :

```
>>> G2 = creer_graphe(2,2)
>>> G2
       {(0, 0): [(1, 0), (0, 1)],
       (1, 0): [(1, 1), (0, 0)],
       (0, 1): [(1, 1), (0, 0)],
       (1, 1): [(0, 1), (1, 0)]}
```

L'affichage de ce graphe est réalisé en utilisant `matplotlib`. A l'éxécution du script "TP15.png" vous devez voir apparaitre dans le shell, le dictionnaire d'adjacence d'un graphe représentant une grille de 4 colonnes et 3 lignes. Ce graphe s'affiche aussi dans la fenêtre **"Tracé 1"**. Vous pouvez analyser rapidement les fonctions qui permettent d'obtenir ce tracé. Prenez le temps aussi d'analyser la structure de donnée du graphe **G1**.

Ajouter une arête

Question 1. En utilisant la fonction `ajouter_arete` fournie, ajouter une arête au graphe `G1` qui part du sommet `(0,0)` et va au sommet `(1,1)`. Dans une nouvelle fenêtre de tracé nommée **"Tracé 2"**, afficher le graphe obtenu.

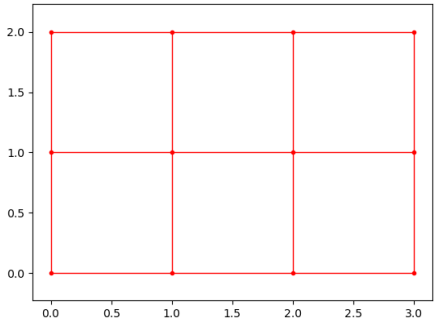

FIGURE 4 – Tracé 1 : grille de 4 colonnes et 3 lignes (graphe G1)

Marquage des noeuds visités

Dans ce TP différents parcours de graphe vont être réalisés. Pour savoir si un sommet a déjà été découvert on utilise un dictionnaire spécifique noté `visited` pour le marquage des sommets suivant les règles suivantes :

- Initialement, tous les sommets sont blancs. On dira qu'un sommet blanc n'a pas encore été découvert.
- Lorsqu'un sommet est "découvert" (autrement dit, quand on arrive pour la première fois sur ce sommet), il est colorié en gris. Le sommet reste gris tant qu'il reste des successeurs de ce sommet qui sont blancs (autrement dit, qui n'ont pas encore été découverts).
- Un sommet est colorié en noir lorsque tous ses successeurs sont gris ou noirs (autrement dit, lorsqu'ils ont tous été découverts).

Ce dictionnaire contiendra tous les sommets de la grille.
On leur associera le caractère **'W'** pour blanc (**W**hite), **'G'** pour gris et **'K'** pour noir (blac**K**).

Question 2. Créer un dictionnaire `visited` associé à la grille `G1` ayant toutes ses valeurs à `'W'`.

Question 3. Modifier ce dictionnaire pour faire un test : on propose de marquer arbitrairement en gris le sommet en haut à gauche ('G') et en noir le sommet en haut à droite ('K'). Utiliser enuite la fonction `trace_visites` fournie pour visualiser l'effet obtenu, toujours sur la même figure **"Tracé 2"**.

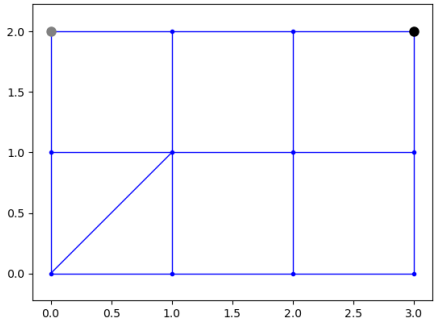

FIGURE 5 – Résultat du tracé 2, marquage de sommets arbitraire

3 Génération d'un labyrinthe par parcours en largeur

L'objectif est de **créer le labyrinthe (schéma à gauche de la figure 2)**. On propose de travailler par étape en modifiant successivement la fonction pour aboutir à l'agorithme complet du parcours en largeur. La fonction aura pour argument systématiquement le graphe à parcourir `G` et un noeud de départ `depart`. Vous penserez à modifier la coloration des sommets via le dictionnaire `visited` (= dictionnaire des sommets découverts) au fur et à mesure.

3.a Initialisation

Question 4. Compléter la fonction `parcours_largeur_init`, qui initialise le dictionnaire des sommets découverts (tout à 'W'), crée un dictionnaire vide `labyrinthe`, crée une file ne contenant que le sommet de départ et trace l'état obtenu.

3.b Première étape : visite des voisins

Question 5. Compléter la fonction `parcours_largeur_etape1`, qui explore le premier sommet (la tête de file). On entend ici par "explorer", découvrir ses voisins, les ajouter dans la file s'ils n'ont pas été déjà découverts. Tracer aussi l'état obtenu.

3.c Algorithme complet

Question 6. Compléter la fonction `parcours_largeur_laby` qui continuera à explorer les voisins tant que la file n'est pas vide. A chaque étape (à chaque tour de boucle) vous ajouterez l'arête dans le labyrinthe, tracerez le graphe `labyrinthe`, afficherez le marquage des noeuds via `visited` et ajouterez l'instruction `plt.pause(0.5)` de façon à voir progressivement le parcours du graphe.

Note : Comme vous pourrez le constater, le coté aléatoire de ce labyrinthe est discutable :). Il est possible de mélanger une liste en utilisant le module `random` : `random.shuffle(voisins)` ce qui permet de mélanger la liste de tuples voisins.

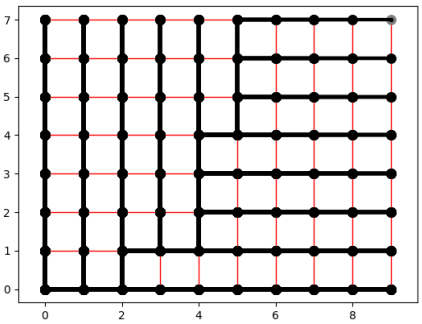

FIGURE 6 – Exemple de labyrinthe obtenu sur une grille 10*10 par parcours en largeur

4 Génération d'un labyrinthe par parcours en profondeur

On rappelle les différentes étapes d'un parcours en profondeur.

On initialise :

- le labyrinthe avec un dictionnaire vide,
- le dictionnaire des sommets visités `visited` ; initialement tous les sommets sont blancs,
- une pile avec le noeud de départ.

A chaque tour de boucle :

- On prend le sommet situé au dessus de la pile, on identifie parmi ses voisins les voisins qui n'ont jamais été visités (on peut les stocker dans une liste `voisins_blancs`)
- S'il reste encore des voisins non découverts (=blancs), on va visiter l'un de ces voisins, on le marque en gris, et on l'ajoute à la pile.
- S'il ne reste plus de voisins blancs, alors on marque le sommet en noir (tous ses successeurs ont été découverts), on l'enlève de la pile : on continuera donc avec le sommet situé juste en dessous dans la pile.

On répète ces instructions tant que la pile n'est pas vide.

Question 7. Compléter l'algorithme `parcours_profondeur_laby`, qui construit le graphe labyrinthe `L` par un parcours en profondeur.

5 Résolution du labyrinthe

Il est possible de résoudre le labyrinthe en utilisant un parcours en largeur ou un parcours en profondeur.

Question 8. Écrire la fonction `resolution_largeur(G:dict, s:tuple) -> list` qui permet de résoudre le labyrinthe en utilisant un parcours en largeur. Cette fonction renvoie la liste des sommets permettant d'atteindre le sommet en haut à droite depuis le sommet en bas à gauche (chemin).

Question 9. Afficher en trait épais bleu la solution donnée par le parcours en largeur.

Question 10. Répondre aux mêmes questions en utilisant un parcours en profondeur.