TP 14
Création et parcours de labyrinthe – Sujet

Activité préparatoire : Génération d'une grille

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 $(1, 0)$).

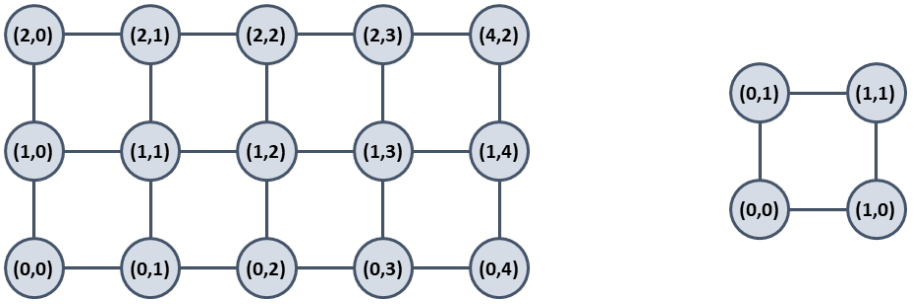


FIGURE 1 – Grille (5,3) et grille (2,2)

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.

Question 1 Écrire la fonction `creer_graphe(n:int, p:int) -> dict` permettant de créer le graphe d'une grille de `n` colonnes et `p` lignes.

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)]}
```

On souhaite afficher ce graphe en utilisant `matplotlib`. Pour cela, on va commencer par tracer chacune des arêtes puis chacun des sommets.

Question 2 Écrire la fonction `get_sommets(G:dict) -> (list,list)` renvoyant deux listes `les_x` et `les_y` contenant respectivement les abscisses des sommets et les ordonnées des sommets.

Exemple –

Dans l'exemple qui suit, les coordonnées de sommets peuvent être dans un ordre différent.

```
>>> les_sx, les_sy = get_sommets(G2)
>>> les_sx, les_sy
        ([0, 1, 0, 1], [0, 0, 1, 1])
```

Question 3 Écrire la fonction `trace_sommets(G:dict, couleur : str) -> None` qui trace sur la figure courante les sommets en utilisant un point coloré.

Exemples : pour tracer avec des points rouge on utilise la fonction suivante : `plt.plot(x,y,'r.')` ; en bleu : `plt.plot(x,y,'b.')`, en noir : `plt.plot(x,y,'k.')` .

Question 4 Écrire la fonction `get_aretes(G:dict) -> list` renvoyant la liste des arêtes du graphe sous la forme d'une liste de listes de tuples. Une arête est donc une liste de sommets où les sommets sont des tuples. Les arêtes ne devront être présentes qu'une fois.

Exemple –

Dans l'exemple qui suit, l'ordre des arêtes peut être dans un ordre différent. Pour une arête donnée, les sommets peuvent aussi être dans un ordre différent.

```
>>> get_aretes(G2)
    [[(0, 0), (1, 0)], [(0, 0), (0, 1)], [(1, 0), (1, 1)], [(0, 1), (1, 1)
    ]]
```

Question 5 Écrire la fonction `trace_arete(s1:tuple, s2:tuple, couleur : str , epaisseur : int ) -> None` qui trace une arête reliant les sommets `s1` et `s2` sur la figure courante.

Exemple : pour tracer l'arête [(0,2),(1,2)] en bleu avec une épaisseur de 2, il faut utiliser l'instruction : `plt.plot([0,1],[2,2],'b',linewidth=2)`.

Question 6 Écrire la fonction `trace_graphe(G:dict,couleur : str,epaisseur : int) -> None` qui permet de tracer les sommets et les arêtes du graphe `G` sur la figure courante. Tracer le graphe en rouge avec une épaissuer de 1 pour obtenir la figure ci-dessous.

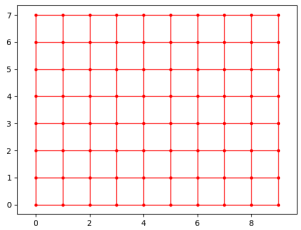


FIGURE 2 – Tracé d'un graphe grille de 10 colonnes et 8 lignes

TP : Génération d'un labyrinthe

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 obtenue dans l'activité préparatoire. 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.

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.

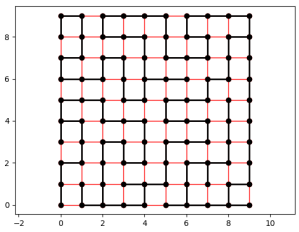


FIGURE 3 – Exemple de labyrinthe obtenu sur une grille 10*10

Ajouter une arête dans un graphe

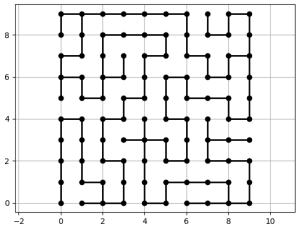
Pour générer un labyrinthe il faut construire un graphe vide puis ajouter une arête à chaque fois que l'on découvre un sommet.

Question 7 Écrire la fonction `ajouter_arete(G:dict, s1:tuple, s2:tuple) -> None` qui permet d'ajouter l'arête `([s1,s2])` au graphe `G`.

Exemple –

On reprend le graphe noté `G2` précédemment.

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



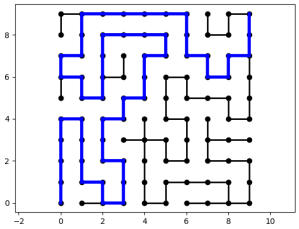


FIGURE 4 – Un labyrinthe et ce même labyrinthe résolu

Bien penser aux éléments suivants : ajouter une arête entre les sommets `s1` et `s2` consiste :

- à ajouter le sommet `s1` dans le graphe si ce sommet n'existait pas : dans ce cas il faut commencer une nouvelle liste de ses voisins avec son premier voisin `s2` ;
- à ajouter le sommet `s2` dans le graphe si ce sommet n'existait pas : dans ce cas il faut commencer une nouvelle liste de ses voisins avec son premier voisin `s1` ;
- à ajouter `s1` (respectivement `s2`) dans la listes des voisins de `s2` (resp. `s1`) dans le cas où le sommet `s2` (resp. `s1`) existait déjà.

Marquages des noeuds visités

On propose de marquer les sommets en utilisant un dictionnaire noté `visited` :

- 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 8 Créer un graphe `G` associé à une grille initiale de 5*5. L'afficher (épaisseur 1, en rouge).

Question 9 Créer un dictionnaire `visited` associé à cette grille ayant ses valeurs toutes à `'W'`.

Question 10 Créer une fonction `trace_visites` qui prend pour argument un dictionnaire `v` des sommets découverts et trace les noeuds découverts en gris ou en noir.

(*Rappel : tracé avec points noirs :* `plt.plot(x,y,'ko')` */ avec des points gris :* `plt.plot(x,y,'o',color='grey')` *)*

Question 11 Test : Après avoir modifié arbitrairement le dictionnaire `visited` pour le test (ajouter arbitrairement des noeuds en gris et en noir), le tracer sur la même figure que la grille.

Génération d'un labyrinthe par parcours en largeur

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.

Initialisation

Question 12 Ecrire une fonction `parcours_largeur_init`, qui initialise le dictionnaire des sommets découverts (tout à 'W'), crée une file ne contenant que le noeud de départ et trace l'état obtenu en utilisant votre fonction `trace_visites`.

Première étape : visite des voisins

Question 13 Modifier la fonction précédente pour obtenir 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.

Algorithme complet

Question 14 Modifier la fonction précédente pour obtenir la fonction `parcours_largeur_complet` , qui continuera à explorer les voisins tant que la file n'est pas vide. A chaque étape (à chaque tour de boucle) vous pouvez tracer l'état, tracer une arête pour indiquer le chemin parcouru, et ajouter l'instruction `plt.pause(0.5)` de façon à voir progressivement le parcours du graphe.

Vous pouvez utiliser la note en bas de page pour ajouter de l'aléatoire au parcours.

Algorithme complet avec création du labyrinthe

Il est maintenant possible de créer le labyrinthe.

Question 15 Modifier la fonction précédente pour obtenir la fonction `labyrinthe_largeur` , qui construit le graphe labyrinthe `L` au fur et à mesure du parcours. Cette fonction retourne le graphe labyrinthe. Vous pouvez tracer à chaque étape le labyrinthe en trait plus épais noir.

Note : Comme vous pouvez 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.

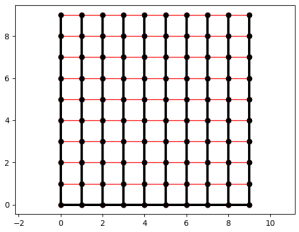


FIGURE 5 – Exemple de labyrinthe obtenu sur une grille 10*10 par parcours en largeur

Génération d'un labyrinthe par parcours en profondeur

Question 16 Modifier l'algorithme `parcours_profondeur` pour réaliser le marquage des sommets comme proposé (gris = sommet "découvert", noir = sommet dont tous les voisins ont été découverts).

Question 17 Etablir la fonction `labyrinthe_profondeur`, qui construit le graphe labyrinthe `L` par un parcours en profondeur. Vous pouvez tracer à chaque étape le labyrinthe en trait plus épais noir.

Résolution du labyrinthe

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

Question 18 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.

Question 19 Afficher en trait épais bleu la solution donnée par le parcours en largeur.

Question 20 Répondre aux mêmes questions en utilisant un parcours en profondeur.