TP 12 : Manipulation de graphes, parcours

Exercice 1 – Manipulation de graphes non pondérés

▶ **Question 1** Pour chacun des deux graphes suivants, déterminer leur représentation en Python par matrice d'adjacence et par liste d'adjacence :

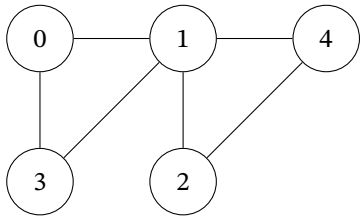


FIGURE 1 – Graphe $\mathcal{G}_1$

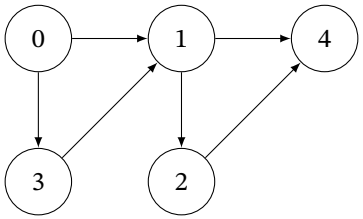


FIGURE 2 – Graphe $\mathcal{G}_2$

▶ **Question 2** Dessiner les graphes décrits par les matrices d'adjacence suivantes :

$$M_1 = \begin{pmatrix} \texttt{faux} & \texttt{vrai} & \texttt{vrai} & \texttt{faux} \\ \texttt{vrai} & \texttt{faux} & \texttt{vrai} & \texttt{vrai} \\ \texttt{vrai} & \texttt{vrai} & \texttt{faux} & \texttt{vrai} \\ \texttt{faux} & \texttt{vrai} & \texttt{vrai} & \texttt{faux} \end{pmatrix} \quad M_2 = \begin{pmatrix} \texttt{faux} & \texttt{vrai} & \texttt{faux} & \texttt{faux} & \texttt{faux} \\ \texttt{faux} & \texttt{faux} & \texttt{vrai} & \texttt{vrai} & \texttt{faux} \\ \texttt{faux} & \texttt{faux} & \texttt{faux} & \texttt{faux} & \texttt{vrai} \\ \texttt{faux} & \texttt{faux} & \texttt{vrai} & \texttt{faux} & \texttt{faux} \\ \texttt{faux} & \texttt{faux} & \texttt{faux} & \texttt{faux} & \texttt{faux} \end{pmatrix}$$

▶ **Question 3** Écrire une fonction `matrice_vers_liste(mat_g)` qui prend en argument un graphe orienté représenté par une matrice d'adjacence et renvoie la liste d'adjacence correspondante.

▶ **Question 4** Écrire une fonction `liste_vers_matrice(lst_g)` qui prend en argument un graphe orienté représenté par une liste d'adjacence et renvoie la matrice d'adjacence correspondante.

▶ **Question 5** Écrire une fonction `degre_NO(lst_g, s)` qui prend en argument un graphe non orienté représenté par une liste d'adjacence et un sommet $s \in [\![0, len(lst\_g) - 1]\!]$ et renvoie le degré de s.

▶ **Question 6** Appliquée à un graphe orienté, quel degré la fonction `degre_NO` renvoie-t-elle ?

▶ **Question 7** Écrire une fonction `degre_O_entrant(lst_g, s)` qui renvoie le degré entrant d'un sommet s.

★ **Question 8** Implémenter ces fonctions sur les graphes représentés par matrice d'adjacence.

▶ **Question 9** Déterminer la complexité des fonctions écrites jusqu'ici en fonction de n le nombre de sommets du graphe et p le nombre d'arêtes. ★ Déterminer la complexité des fonctions sur les graphes représentés par matrice d'adjacence.

Exercice 2 – Parcours de graphes

▶ **Question 1** En reprenant les exemples des Questions 1 et 2 de l'Exercice 1, déterminer leur parcours en profondeur à la main. On parcourra les sommets dans l'ordre croissant.

▶ **Question 2** Implémenter une fonction `parcours_profondeur(lst_g, s)` qui prend en argument un graphe orienté représenté par une liste d'adjacence et un sommet $s \in [\![0, len(lst\_g) - 1]\!]$ et qui parcoure en profondeur le graphe à partir du sommet s, en affichant avec un `print` les sommets dans l'ordre de leur visite.

La première application d'un parcours de graphe est de tester la connexité d'un graphe non orienté. Pour cela, il suffit de parcourir le graphe en profondeur à partir d'un sommet quelconque. Si tous les sommets sont visités par le parcours, le graphe est connexe.

▶ **Question 3** Implémenter une fonction `est_connexe(lst_g)` qui prend en argument un graphe non orienté représenté par une liste d'adjacence et qui renvoie `True` si le graphe est connexe et `False` sinon.

Un parcours de graphe permet également de déterminer les composantes connexes d'un graphe non orienté. Pour cela, on effectue un parcours en profondeur à partir de chaque sommet non visité. Chaque fois qu'un parcours est terminé, on a trouvé une composante connexe.

★ **Question 4** Implémenter une fonction `composantes_connexes(lst_g)` qui prend en argument un graphe non orienté représenté par une liste d'adjacence et qui renvoie la liste des composantes connexes du graphe, chaque composante étant représentée par une liste d'entiers.

★ **Question 5** Déterminer la complexité de ces fonctions.

Exercice 3 – Exemple d'application : parcours de labyrinthe

On représente dans la figure suivante un labyrinthe constitué de 30×30 cases, numérotées lignes par lignes puis colonnes par colonnes de 0 à 899. L'objectif de cet exercice est de trouver des chemins permettant d'atteindre différents objectifs dans le labyrinthe : ma sortie est au niveau de la case 899, il y a un trésor dans la case 777 et un monstre sur la case 666.

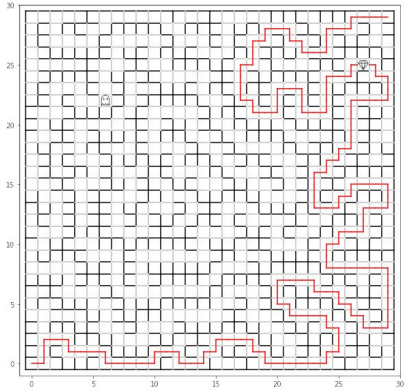


FIGURE 3 – Labyrinthe et chemin du départ vers la sortie

Une représentation possible du labyrinthe est par un graphe non orienté : chaque case est identifiée à un sommet (par son numéro) et deux sommets sont reliés par une arête si les cases correspondantes sont adjacentes et qu'il n'y a pas de mur entre elles. La liste d'adjacence d'un labyrinthe de taille 900×900 est stockée dans le fichier `labyrinthe.py` sur le cahier de prépas. L'entrée est en 0, la sortie en $900^2 - 1$ et le trésor en 777 et le monstre en 666.

▶ **Question 1** Est-il raisonnable de représenter ce labyrinthe par matrice d'adjacence ?

▶ **Question 2** En partant du départ, à l'aide des fonctions écrites précédemment, répondez aux questions suivantes :

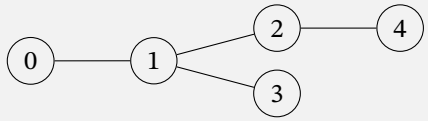
- Peut-on atteindre la sortie ?
- Peut-on récupérer le trésor ?
- Risquez-vous de vous faire attraper par le monstre s'il peut se déplacer ?
- ★ pouvez-vous atteindre toutes les cases du labyrinthe ? Si non, combien pouvez-vous en atteindre ?
- ★ (après l'exercice suivant) quelle est la liste des cases empruntées pour les deux premières questions ?

Exercice 4 – Reconstitution de chemins

Une fois qu'un parcours en profondeur se termine, il n'y a plus aucune information sur le chemin parcouru pour atteindre un sommet donné. Pour reconstituer un chemin, on va commencer par stocker en mémoire l'arbre de parcours en profondeur à l'aide d'un *liste de pères* : pour chaque sommet visité s, on stocke dans $l[s]$ le sommet p qui a permis de visiter s pour la première fois (-1 s'il n'a pas encore été visité, lui-même si c'est le point de départ du parcours). On peut alors reconstituer le chemin en remontant de la sortie au départ.

Exemple 1

Le parcours en profondeur du graphe $\mathcal{G}_1$ est représenté par l'arbre de parcours suivant (représenté de gauche à droite) :



La liste de pères correspondante est alors : $[0, 0, 1, 1, 2]$. 0 est la racine de l'arbre de parcours, donc $l[0] = 0$. On a parcouru le sommet 4 depuis le sommet 2, donc $l[4] = 2$.

▶ **Question 1** Adapter le parcours en profondeur en une fonction `parcours_profondeur_pere(lst_g, s)` qui renvoie la liste des pères des sommets visités lors du parcours en profondeur depuis s.

À partir de cette liste de prédécesseurs, il est possible de reconstituer un chemin du sommet initial s vers un sommet accessible x en « remontant » les prédécesseurs depuis x jusqu'à s.

▶ **Question 2** En déduire une fonction `chemin(lst_g, s, x)` qui prend en argument un graphe, un sommet initial s et un sommet accessible x et qui renvoie la liste des sommets du chemin de s à x. *Indication : la syntaxe* `L[::-1]` *produit une copie de la liste* `L` *retournée.*

Le chemin produit par cette méthode pour un parcours en profondeur **n'est pas un plus court chemin** au sens de la longueur. Une propriété intéressante du *parcours en largeur* est qu'il parcourt le graphe par distance successive au sommet initial, ce qui permet d'adapter la méthode de listes des pères de cet exercice pour produire un plus court chemin.