Premières manipulations sur les graphes

Dans tout ce TP, nous ne considèrerons que des graphes non pondérés et non orientés.

On rappelle qu'un graphe $G = (S, A)$ est la donnée d'un ensemble S de sommets et d'un ensemble A d'arêtes. Le degré d'un sommet S est le nombre d'arêtes issues de S, on pourra parler de «son nombre de voisins».

On rappelle que pour créer une matrice nulle de taille (n, p) on peut utiliser une liste de listes avec l'instruction `M = [ [0 for j in range(p)] for i in range(n)]`.

1 Implémentations de graphes

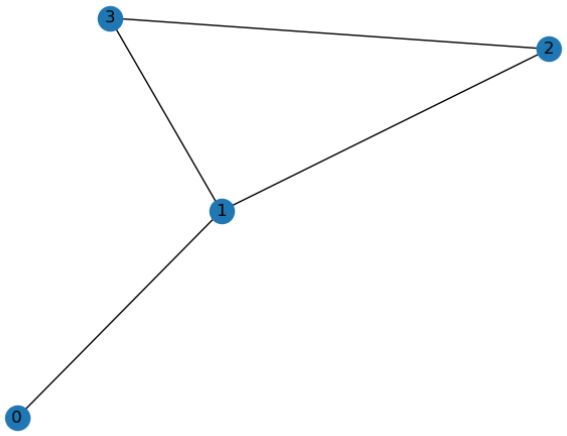

FIGURE 1 – Graphe 1, non pondéré, non orienté et connexe

Exercice 1 (Premiers exemples) On se familiarise ici avec matrice d'adjacence, liste d'adjacence et dictionnaire d'adjacence.

1. Écrire la matrice d'adjacence M du graphe 1 ci-contre et l'implémenter en Python.
2. Implémenter aussi ce graphe à l'aide d'une liste d'adjacence `adj`. Par exemple `adj[2]` sera égal à `[1, 3]` la liste des voisins du sommet 2.
3. Implémenter le graphe 2 à l'aide d'un dictionnaire d'adjacence `dico`. Par exemple `dico['a']` sera égal à `['b', 'c']` la liste des voisins du sommet `'a'` (on initialise un dictionnaire à l'aide de l'instruction `dico = dict()`).

Exercice 2 (D'une représentation à l'autre)

1. Écrire une fonction `liste_vers_matrice(adj)` qui prend en entrée la liste d'adjacence d'un graphe et renvoie sa matrice d'adjacence associée.
2. Écrire une fonction `matrice_vers_liste(mat)` qui prend en entrée la matrice d'adjacence d'un graphe et renvoie sa liste d'adjacence associée.

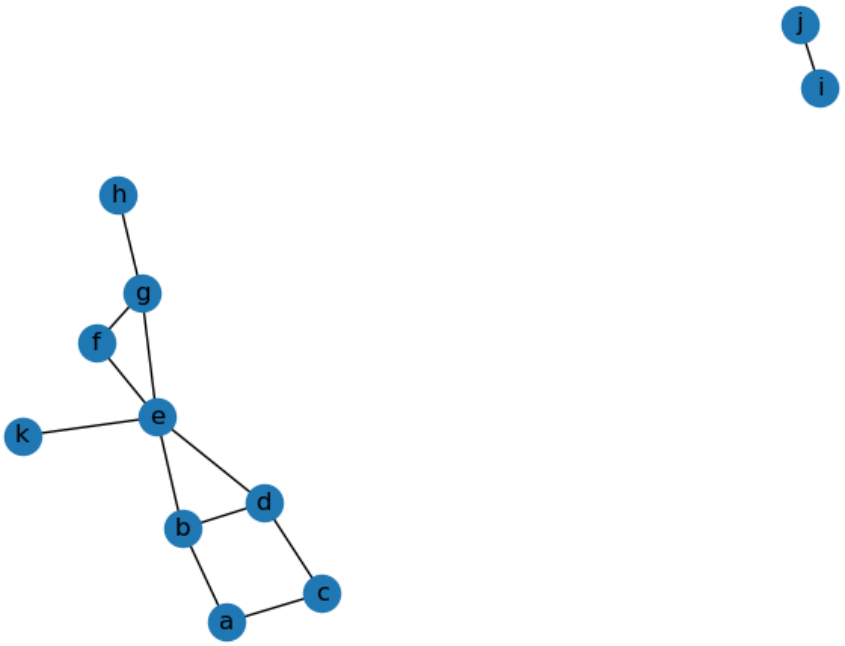

FIGURE 2 – Graphe 2, non connexe

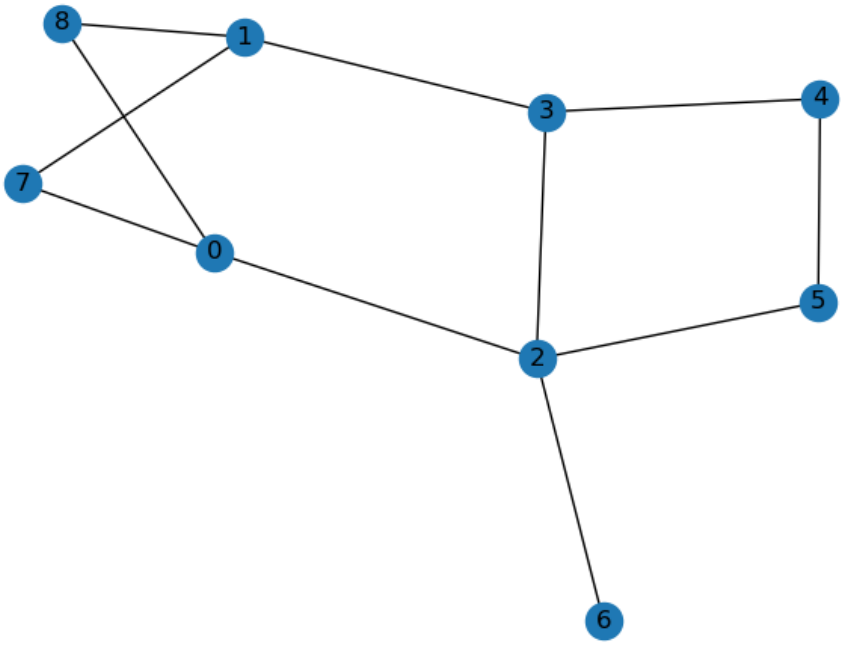

FIGURE 3 – Graphe 3

3. Écrire la liste d'adjacence `adj3` du graphe 3, en déduire sa matrice d'adjacence.

Exercice 3 (Densité d'un graphe) Soit $G = (S, A)$ un graphe non orienté. On appelle densité de G le nombre $\frac{2|A|}{|S|(|S|-1)}$.

1. Si un graphe possède n sommets. Quel est le nombre maximal d'arêtes que peut posséder un graphe à n sommets ? Un tel graphe sera dit complet, tous ses sommets sont connectés entre eux. Quelle est la densité d'un graphe complet ?

2. Programmer une fonction `degre(graphe, sommet)` qui renvoie le degré d'un sommet dans un graphe. On supposera que le graphe est implémenté par une liste d'adjacence.

3. En déduire une fonction `densite(graphe)` qui renvoie la densité d'un graphe. Vérifier que les graphes 1 et 3 ont pour densité respective $0,666$ et 0.305.

2 Une tentative de modélisation d'un réseau social

On considère le mini réseau social suivant implémenté dans votre fichier `.py` **par un dictionnaire d'adjacence**.

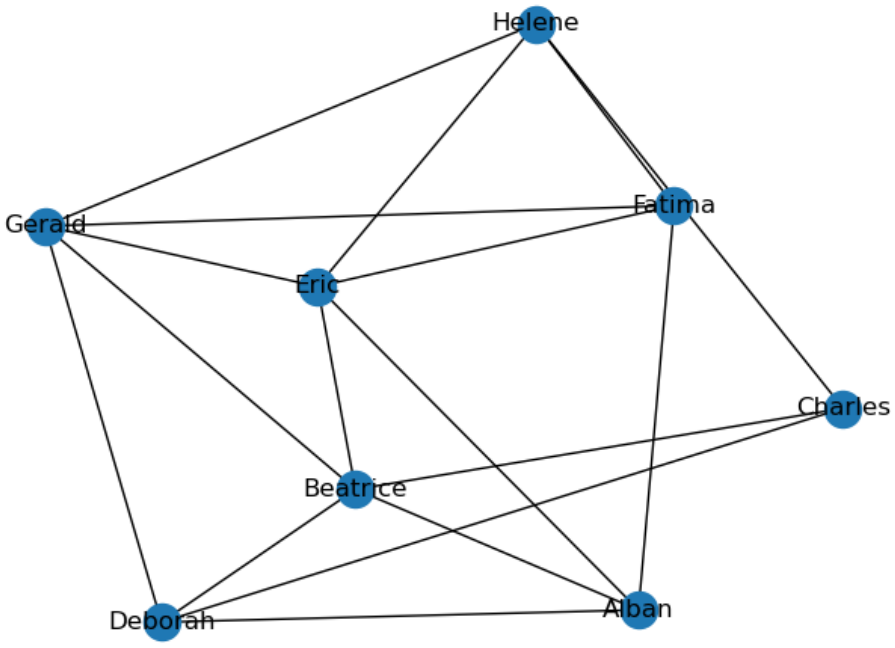

FIGURE 4 – Mini réseau social

1. Écrire une fonction `nombre_amis(personne, graphe)` qui prend en argument une personne du réseau social et renvoie son nombre d'amis.

2. Écrire une fonction `nombre_moyen_amis(graphe)` qui prend en argument un réseau social appelé `graphe` et renvoie le nombre moyen d'amis que possède une personne du réseau. On trouve 4.25 pour le mini réseau social.

3. Écrire une fonction `plus_populaire(graphe)` qui prend en argument un réseau social appelé `graphe` et renvoie la liste des personnes les plus populaires, c'est-à-dire ayant le plus d'amis, ainsi que leur nombre d'amis. On trouve `(['Beatrice', 'Eric', 'Gerald'], 5)` pour le mini réseau social.

3 Parcours aléatoire sans mémoire sur un graphe

On suppose dans cette section que les graphes que l'on considère sont non pondérés, non orientés et **connexes**.

On désire parcourir nos graphes de manière aléatoire, c'est-à-dire que lorsque qu'on est sur un sommet, on choisit au hasard un de ses voisins et on y va.

1. Écrire une fonction `est_chemin_sortie(G, source, cible)` qui possède trois paramètres : un graphe `G` défini par liste ou dictionnaire d'adjacence, ainsi que `source` et `cible` deux sommets du graphe. La fonction doit générer un parcours aléatoire depuis le sommet `source` et renvoyer 1 si on arrive à la `cible` sans repasser par `source`. Elle doit renvoyer 0 sinon. On pourra utiliser la fonction `random.choice` qui choisit un élément au hasard d'une liste.

2. En déduire une fonction `proba_de_ne_pas_repasser_par_source(G, source , cible, nb_simul = 1000)` qui renvoie la probabilité que lors d'un parcours aléatoire, on part de la source et on arrive à la cible sans repasser par la source.

3. Application : on prend le graphe 3. On suppose que le bar est le sommet d'indice 0. Quelle est la probabilité qu'une personne saoûle parte du bar et retrouve son domicile le sommet 4 sans repasser par le bar.

4. Coder une fonction `graphe_ligne(n:int)-> dict` qui permet de créer un «graphe-ligne» à $n+1$ sommets numérotés de 0 à n. On souhaite que le graphe soit implémenté par dictionnaire d'adjacence.

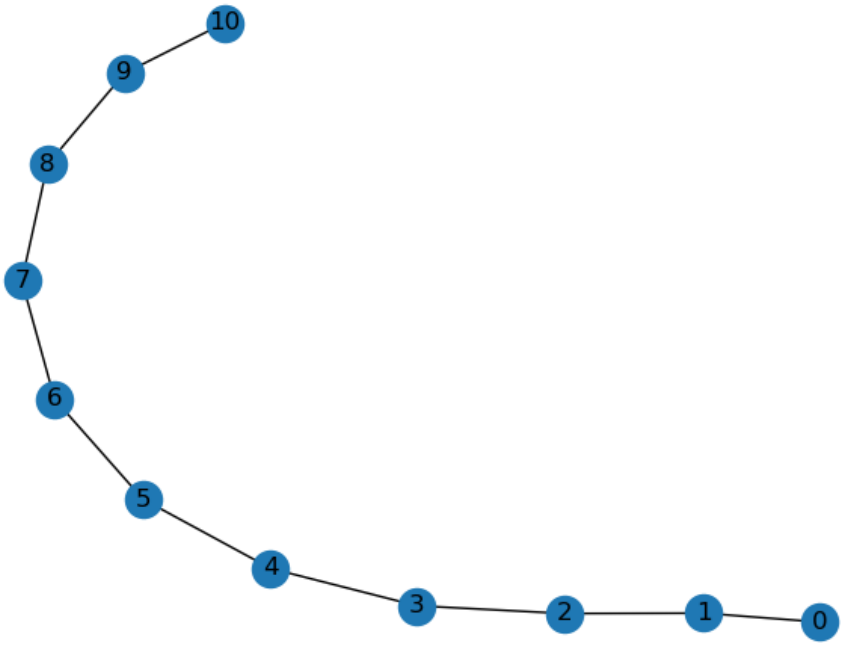

FIGURE 5 – Graphe ligne avec $n = 10$.

5. Estimer la probabilité que lors d'une marche aléatoire sur un «graphe ligne» au départ du sommet 0, on atteigne le sommet n sans repasser par le sommet 0. Regarder sur Internet le lien avec le théorème de Polya sur les marches aléatoires.

4 Quelques utilisations de la matrice d'adjacence

Exercice 4 (Puissances de la matrice d'adjacence d'un graphe non pondéré) On considère un graphe non pondéré représenté par sa matrice adjacence M. Nous allons obtenir un algorithme de plus court chemin, ou un test d'accessibilité en calculant les puissances k-ièmes de M.

1. Écrire une fonction `produit(A, B)` qui prend en argument A et B deux matrices et renvoie leur produit AB lorsque celui-ci est possible.

On admet pour la suite la propriété mathématique suivante : soit $k \in \mathbb{N}^*$, le coefficient d'indice (i, j) de la matrice M^k est égal au nombre de chemins de longueur k entre i et j.

Il existe donc un chemin de i vers j s'il existe un entier k tel que le coefficient $M^k_{i,j}$ soit strictement positif. Si le graphe possède n sommets, un tel chemin s'il existe est forcément de longueur inférieure ou égale à $n-1$. On en déduit un algorithme (assez coûteux) naïf qui teste l'accessibilité entre deux sommets ou la connexité d'un graphe : on part de $k = 1$ et on calcule M^k pour $k \leqslant n-1$. Si le coefficient $(M^k)_{i,j}$ est non nul, alors on a trouvé un chemine de longueur k...

2. Implémenter un tel algorithme en écrivant une fonction `est_accessible(mat, i, j)` qui teste si l'on peut accéder du sommet i au sommet j dans le graphe implémenté par la matrice d'adjacence `mat`. Si l'on trouve un chemin de longueur k, on renverra k, s'il n'y a pas de chemin on renverra `None`.

3. Tester avec la matrice d'adjacence fourni dans le fichier «élève».

4. Démontrer la propriété mathématique.