Algorithmes et Pensée Computationnelle

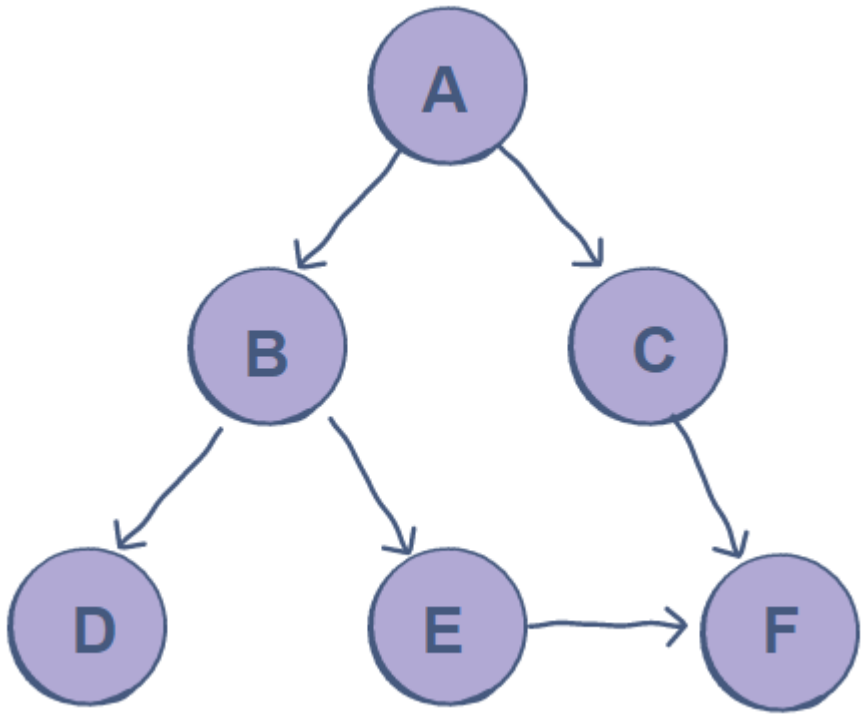
Algorithmes de graphes

Le but de cette séance est de comprendre le fonctionnement des graphes et d'appliquer des algorithmes courants sur des graphes simples.

Le code présenté dans les énoncés se trouve sur Moodle, dans le dossier `Code`. Le temps indiqué (🕓) est à titre indicatif.

1 Breadth-First Search

Question 1: (🕓 *10 minutes*) **Adjacency list et adjacency matrix : Python**
L'adjacency list (ou liste de contiguïté) et l'adjacency matrix (ou matrice de contiguïté) sont les 2 méthodes dont nous disposons pour représenter un graphe. Utilisez ces 2 méthodes de représentations pour stocker le graphe ci-dessous dans un programme Python :



Conseil

Pour l'adjacency list, utilisez un dictionnaire Python. Pour l'adjacency matrix, utilisez une liste multidimensionnelle (ou liste contenant une ou plusieurs listes).

Solution

```
#Question 1 solution

adjacency_list_graph = {
  'A' : ['B','C'],
  'B' : ['D', 'E'],
  'C' : ['F'],
  'D' : [],
  'E' : ['F'],
  'F' : []}

adjacency_matrix_graphe = [[0,1,1,0,0,0],
                           [0,0,0,1,1,0],
                           [0,0,0,0,0,1],
                           [0,0,0,0,0,0],
                           [0,0,0,0,0,1],
                           [0,0,0,0,0,0]]
```

Le fait que le graphe soit dirigé joue un rôle important pour la construction de ces représentations. Par exemple, pour l'adjacency list, 'B' apparaît dans la liste correspondant à la clé 'A' mais l'inverse n'est pas vrai. Cela signifie que l'on peut aller du sommet A au sommet B mais pas du sommet B au sommet A.

Question 2: (*20 minutes*) **Breadth-First Search algorithm : Python**

Nous allons maintenant nous intéresser au premier algorithme portant sur les graphes : **Breadth-First Search**. Le but du Breadth-First Search est de trouver tous les sommets atteignables à partir d'un sommet de départ.
Implémentez l'algorithme en suivant les étapes suivantes :

1. Partez du sommet initial, visitez les sommets adjacents, sauvegardez-les comme **visités**, insérez-les dans une **queue**.
2. Parcourez la **queue**. Pour chaque élément de la queue, visitez les sommets adjacents. S'ils ne sont pas dans la liste des sommets visités, ajoutez-le à cette dernière et ajoutez-le à la queue. Une fois que cela est fait, supprimez l'élément parcouru de la queue.
3. Répétez l'étape 2 jusqu'à ce que la queue soit vide.

Conseil

Quelques conseils pour l'implémentation de votre algorithme :

1. Utilisez l'adjacency list.
2. Pour le point 3), utilisez une boucle `while` avec la condition appropriée.
3. Pour parcourir les sommets adjacents, utilisez une boucle for .
4. L'algorithme devrait retourner une liste contenant l'ensemble des sommets atteignables.
5. Vous pouvez utiliser l'image du graphe pour déterminer si l'output de votre l'algorithme est correct.
6. Pensez à utiliser deux listes qui contiendront la liste des nœuds à visiter et ceux qui restent à visiter.
7. Vous pouvez utiliser la méthode `.pop()` pour supprimer un élément d'une liste et retourner l'élément supprimé. Cet élément peut être stocké dans une variable.

Solution

```
#Question 2

adjacency_list_graph = {
  'A' : ['B','C'],
  'B' : ['D', 'E'],
  'C' : ['F'],
  'D' : [],
  'E' : ['F'],
  'F' : []
}

def BFS(graph, start):
    visited = list() # liste des sommets visités
    queue = [start] # liste des sommets à visiter
    while len(queue) > 0: # tant que la queue n'est pas vide
        node = queue.pop(0) # on stocke le premier élément de la queue, puis on l'enlève de la queue
        if node not in visited: # si le sommet n'a pas déjà été visité
            visited.append(node) # on l'ajoute à la liste des sommets visités
            neighbors = graph[node] # on récupère la liste des sommets adjacents au sommet courant
        for neighbor in neighbors: # Puis on parcourt la liste de ceux-ci
            queue.append(neighbor) # on les ajoute à la queue
    return visited # on retourne la liste des sommets visités (=atteignables)


# Vérifions que l'algorithme fonctionne correctement
print(BFS(adjacency_list_graph, 'B'))
print(BFS(adjacency_list_graph, 'A'))
```

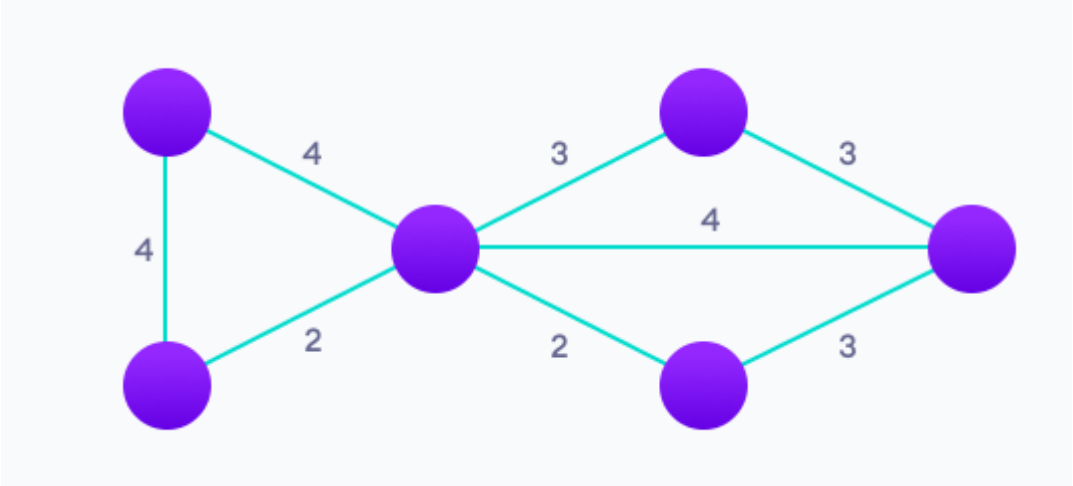
Si vous avez correctement écrit votre algorithme, l'output de celui-ci avec comme input notre graphe et le sommet 'A' devrait être ['A', 'B', 'C', 'F', 'D', 'E']

2 Minimum Spanning tree

Nous allons maintenant nous intéresser à **l'algorithme de Kruskal**. Ce dernier s'applique uniquement aux **weighted graphs** (ou graphes pondérés). Ces derniers sont des graphes où les arêtes ont des poids, représentant par exemple une distance. L'algorithme de Kruskal a pour but de trouver un **minimum spanning tree**. Un minimum spanning tree S de G est un sous-graphe connexe de G tel que :

1. V' = V, c'est-à-dire que tous les sommets de G sont aussi dans S
2. (V',E') ne contient pas de cycle (pas de cycle dans S)
3. S est le graphe satisfaisant 1) et 2) et ayant la plus petite somme des poids

Question 3: (🕒 *5 minutes*) **Algorithme de Kruskal : Papier**
Appliquez l'algorithme de Kruskal au graphe suivant :



Conseil

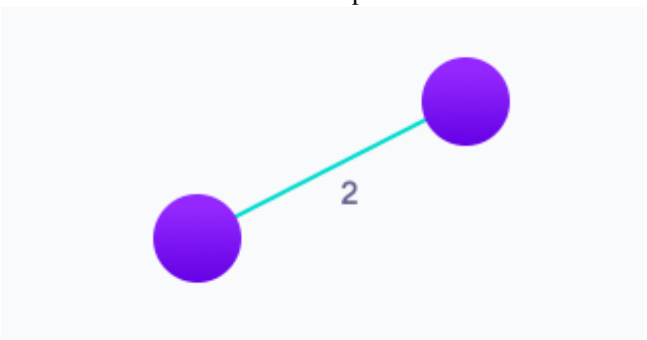
L'algorithme de Kruskal fonctionne de la façon suivante :

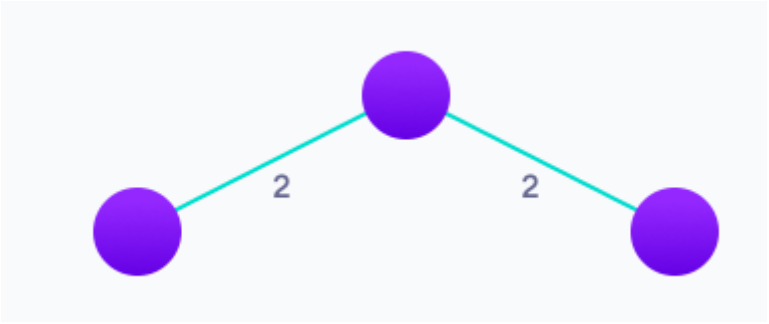
1. Classer les arêtes par ordre croissant de poids.
2. Prendre l'arête avec le poids le plus faible et l'ajouter à l'arbre (si 2 arêtes ont le même poids, choisir arbitrairement une des 2).
3. Vérifiez que l'arête ajoutée ne crée pas de cycle, si c'est le cas, supprimez la.
4. Répétez les étapes 2) et 3) jusqu'à ce que tous les sommets aient été atteints.

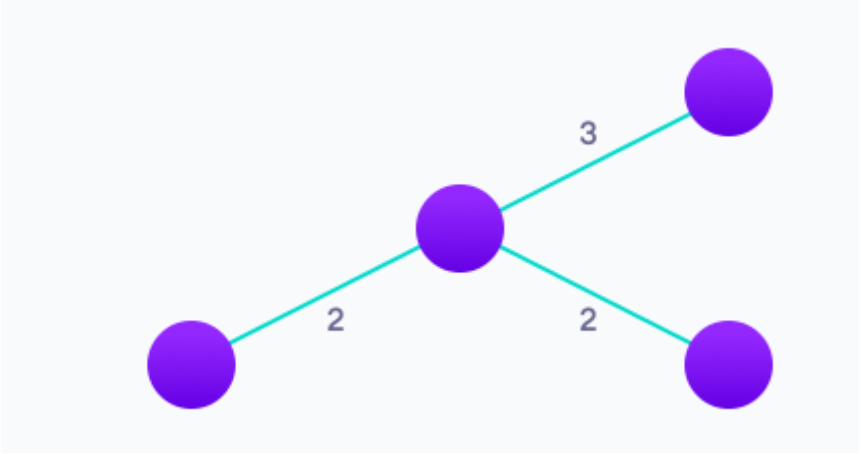
Un Minimum Spanning Tree, s'il existe, a toujours un nombre d'arêtes égal au nombre de sommets moins un. Par exemple, ici notre graphe a 6 sommets. L'algorithme devrait donc s'arrêter lorsque 5 arêtes ont été choisies.

Solution

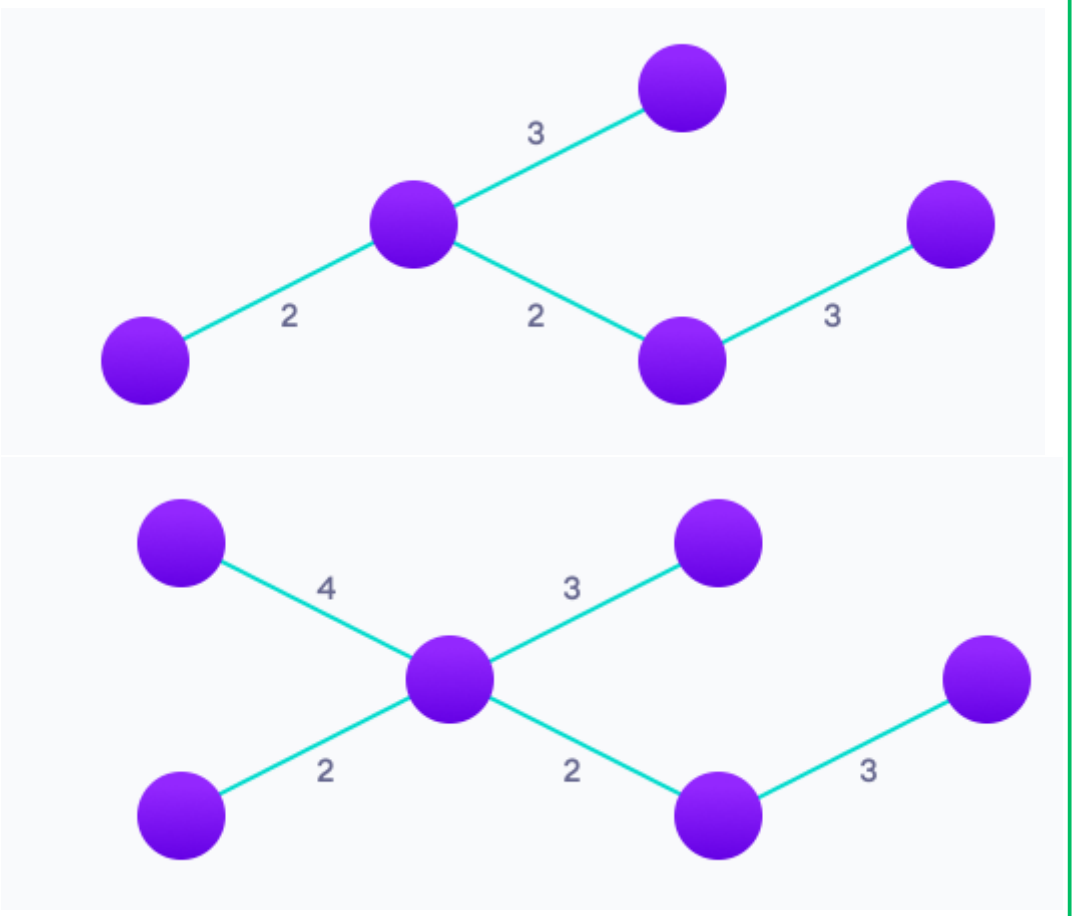
Vous trouverez ci-dessous les étapes de la construction du Minimum spanning tree :







Solution



L'algorithme s'arrête car tous les sommets ont été atteints. On voit bien que seules 5 arêtes ont été nécessaires.

Question 4: (*15 minutes*) **Algorithme de Kruskal : Python**
Vous trouverez ci-dessous l'algorithme de Kruskal implémenté en Python (disponible sur Moodle). Parcourez la fonction **Kruskal_algo(Graph)** afin de vous assurez que vous ayez bien compris le fonctionnement :

```
1  #Question 3
2
3  class Graph:
4      def __init__(self, vertices): # permet de créer un graphe lorsqu'on écrit
       p.ex Graph(6), il faut notamment indiquer le nombre de sommets
5          self.V = vertices
6          self.graph = []
7
8      def add_edge(self, u, v, w): # ajoute une arête entre le sommet u et v
       avec un poids w
9          self.graph.append([u, v, w])
10
11     def find(self, parent, i): # Correspond à la fonction Find-set(x)
       présentée dans le cours
12         if parent[i] == i:
13             return i
14         return self.find(parent, parent[i])
15
16     def apply_union(self, parent, rank, x, y): # Correspond à la fonction
       Union(x,y) présentée dans le cours
17         xroot = self.find(parent, x)
```

```
18          yroot = self.find(parent, y)
19          if rank[xroot] < rank[yroot]:
20              parent[xroot] = yroot
21          elif rank[xroot] > rank[yroot]:
22              parent[yroot] = xroot
23          else:
24              parent[yroot] = xroot
25              rank[xroot] += 1
26
27  g = Graph(6)
28  g.add_edge(0, 1, 4)
29  g.add_edge(0, 2, 4)
30  g.add_edge(1, 2, 2)
31  g.add_edge(1, 0, 4)
32  g.add_edge(2, 0, 4)
33  g.add_edge(2, 1, 2)
34  g.add_edge(2, 3, 3)
35  g.add_edge(2, 5, 2)
36  g.add_edge(2, 4, 4)
37  g.add_edge(3, 2, 3)
38  g.add_edge(3, 4, 3)
39  g.add_edge(4, 2, 4)
40  g.add_edge(4, 3, 3)
41  g.add_edge(5, 2, 2)
42  g.add_edge(5, 4, 3)
43
44  def kruskal_algo(Graph):
45          result = [] # Permettra de stocker le résultat
46          i, e = 0, 0 # Index utilisé dans l'algorithme
47
48          Graph.graph = sorted(Graph.graph, key=lambda item: item[2]) # Trie les
    arêtes par poids croissant, étape 1)
49          parent = []
50          rank = []
51
52
53          for node in range(Graph.V): # Cette boucle parcourt tous les sommets
    du graphe et crée un ensemble pour chacun d'entre eux
54              parent.append(node)
55              rank.append(0)
56
57          # Tant que le nombre d'arêtes est inférieur à V-1, notre sous-graphe
    n'atteint pas tous les sommets -> on continue
58          while e < Graph.V - 1:
59
60              u, v, w = Graph.graph[i] # self.graph contient les arêtes par
    ordre croissant de poids, on commence avec i = 0
61              i = i + 1               # puis à l'itération suivante on voudra
    avoir la 2ème arête la plus légère, donc on incrémente.
62
63              x = Graph.find(parent, u) # Ces 2 lignes des codes permettent de
    rechercher et de stocker à quel ensemble
64              y = Graph.find(parent, v) # appartiennent u et v.
65
66              if x != y: # Si u et v font déjà parti du Minimum Spanning Tree,
    i.e. u et v appartiennent au même ensemble
67                      # Alors on ne veut pas ajouter cette arête au minimum
    spanning-tree, d'ou le x!=y
68                  e = e + 1 # Si u et v sont d'ensemble différent, on a atteint
    un sommet de plus donc on incrémente
69                  result.append([u, v, w]) # On ajoute la nouvelle arête au
    résultat
70                  Graph.apply_union(parent, rank, x, y) # On fusionne l'ensemble
```

```
    auquel appartient v à celui auquel appartient u
71          for u, v, weight in result:
72              print("%d - %d: %d" % (u, v, weight)) # méthode permettant
    d'afficher le résultat
73
74          return result
75
76  kruskal_algo(g)
```

Conseil

L'output de l'algorithme est :
1 - 2 : 2
2 - 5 : 2
2 - 3 : 3
3 - 4 : 3
0 - 1 : 4
Il se lit comme une liste d'arêtes et de poids à l'arête correspondante.

Question 5: (*10 minutes*) **Social Network Analysis : Papier**

Les graphes peuvent être utilisés pour représenter une multitude de choses. L'une d'entre elles est la représentation de votre réseau d'amis. Imaginez que vous possédiez une liste de vos amis ainsi que des amis de vos amis (qui ne sont pas nécessairement vos amis). Cette liste peut-être représentée sous forme de graphe. Dans ce graphe :

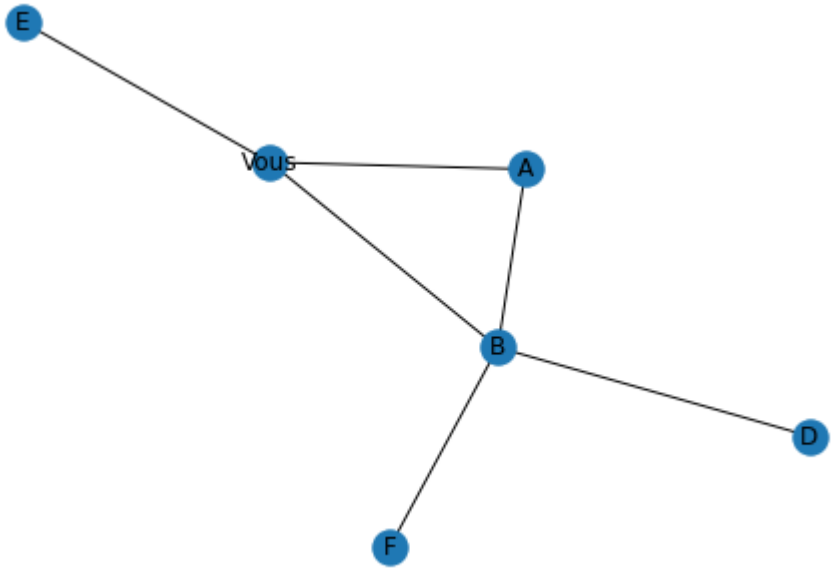
1. À quoi correspondent les arêtes et les nœuds ?
2. Faut-il utiliser un graphe dirigé ?

Supposez que vous disposiez d'un graphe des relations sociales. Décrivez comment retrouver les éléments suivants :

1. Votre ami qui a le plus d'ami
2. Découvrir quels amis à vous se connaissent
3. Listez vos amis qui pourraient vous présenter quelqu'un que vous ne connaissez pas (ami d'ami qui n'est pas votre ami)

Vous voudriez désormais ajouter une nouvelle personne sur ce graphe, mais cette dernière n'est ni votre ami, ni l'ami d'un de vos amis. Quel sera son degré dans le graphe ?

Retrouvez les éléments 1) à 3) dans le graphe ci-dessous :



Conseil

Réfléchissez en terme d'arêtes, de sommets, de degrés et de cycles.

Solution

Dans le graphe des relations sociales, les arêtes correspondent à un lien d'amitié et les nœuds représentent les personnes. Il n'est pas nécessaire d'utiliser un graphe dirigé si l'on considère qu'une relation d'amitié est toujours réciproque.

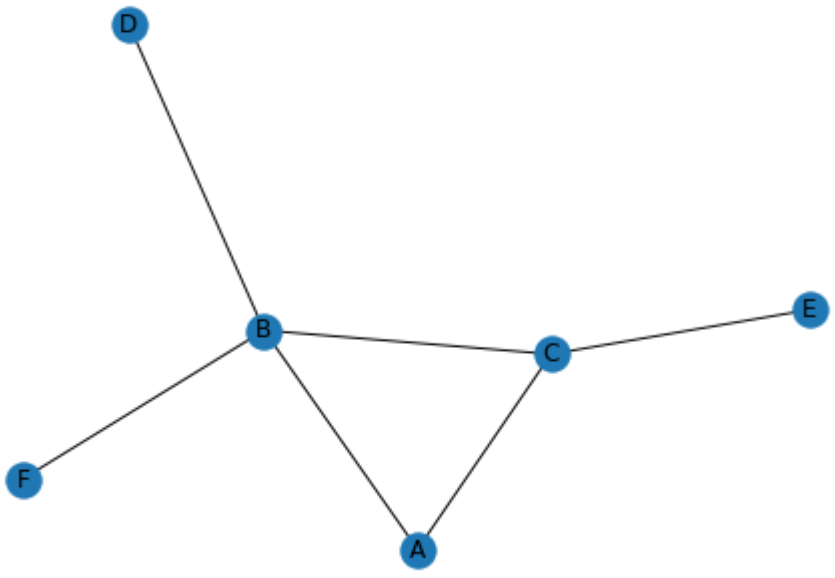
Pour trouver les éléments 1) à 3), il faut raisonner de la façon suivante :

1. Trouvez le sommet relié à vous qui a le degré le plus élevé. Sommet B dans le graphe.
2. Premièrement, les 2 personnes doivent être mes amis donc reliées à moi, mais de plus elles doivent être reliées entre elles. Par conséquent, cela correspond à un cycle dans le graphe. Il y a autant d'amis qui se connaissent que de cycle dans le graphe. Ami A et B dans le graphe.
3. Il ne doit pas y avoir d'arêtes me reliant avec l'ami de mon ami. Sommet B dans le graphe.

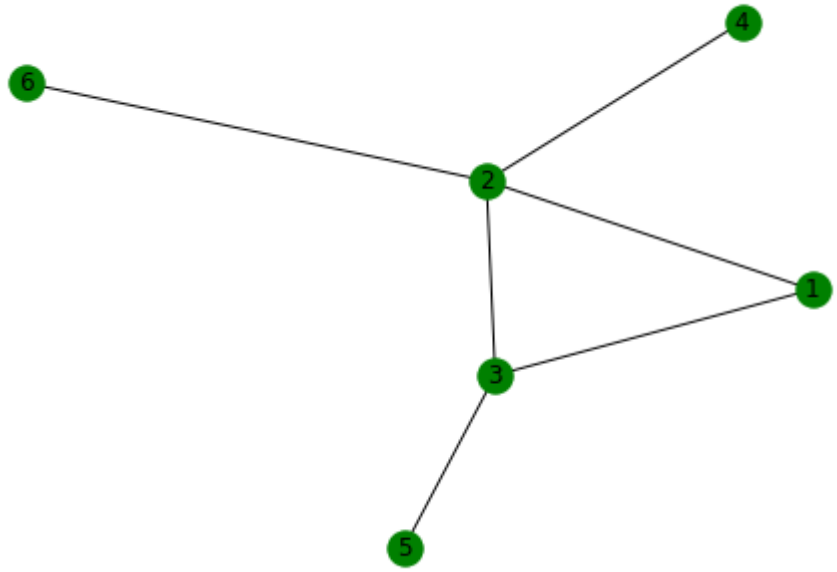
Si l'on ajoute une personne qui n'est ni un ami, ni l'ami d'un ami, alors aucune arête n'est reliée à ce sommet. Par conséquent, ce sommet a un degré 0.

Question 6: (*5 minutes*) **Reconnaître des réseaux semblables**
Supposons maintenant que vous travaillez chez Meta, qui a racheté Whatsapp, et que vous disposiez des 2 graphes représentant respectivement Facebook et Whatsapp. Pouvez-vous déterminer visuellement à qui correspondent les individus de Facebook sur Whatsapp ?

Graphe de Facebook :



Graphe de Whatsapp :



Conseil

Quelques conseils pour vous aider à résoudre cet exercice :

1. Identifiez les sommets des 2 graphes avec des caractéristiques semblables.
2. Il est possible que les sommets ne soient pas tous identifiables.

Solution

1. B correspond à 2 (seul sommet de degré 4)
2. A correspond à 1 (seul sommet de degré 1 relié au sommet de degré 4)
3. C correspond à 3 (seul sommet de degré 3)
4. E correspond à 5 (seul sommet de degré 1 relié au sommet de degré 2)

Les sommets D et F et 4 et 6 ne peuvent pas être dissociés. On ne peut donc pas savoir qui correspond à qui.