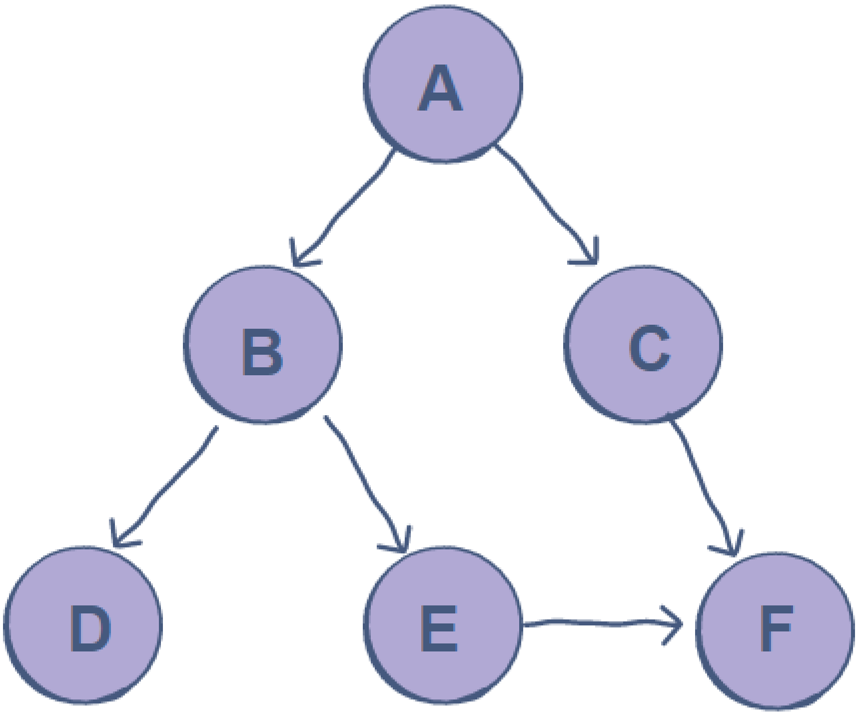
Algorithmes et Pensée Computationnelle

Graph Algorithms

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

1 Breadth-First Search

Question 1: (*5 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 une graphe. Utilisez ces 2 méthodes de représentations pour stockez le graphe ci-dessous dans un programme python :



1.PNG

Conseil

Pour l'adjacency list, utilisez un dictionnaire Python. Pour l'adjacency matrix, utilisez une liste de liste.

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: (*15 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 tout 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 adjascents, sauvegardez-les comme **visités**, insérez-les dans une **queue**.
2. Parcourez la **queue**. Pour chaque éléments de la queue, visitez les sommets adjacents. Si 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 utilisez l'image du graphe pour déterminer si l'output de votre l'agorithme est correct.

Solution

```python
#Question 2

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


def BFS(graphe,s):
    queue = [s] #on initialise la queue
    visited = [s]#on initialise la liste des sommets visités

    while len(queue) != 0: #Aussi longtemps que la queue n'est pas vide, répéter l'étape 2
        for i in queue: #On parcourt les éléments de la queue
            for k in graphe[i]:#Pour chaque éléments de la queue, on parcout tout les voisins
                if k not in visited:#Si le voisin n'est pas déjà visité, on l'ajoute à la queue, et on le marque comme visité
                    queue.append(k)
                    visited.append(k)
            queue.remove(i) #Une fois que l'élément de la queue a été parcouru, on le supprime de 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'))
```

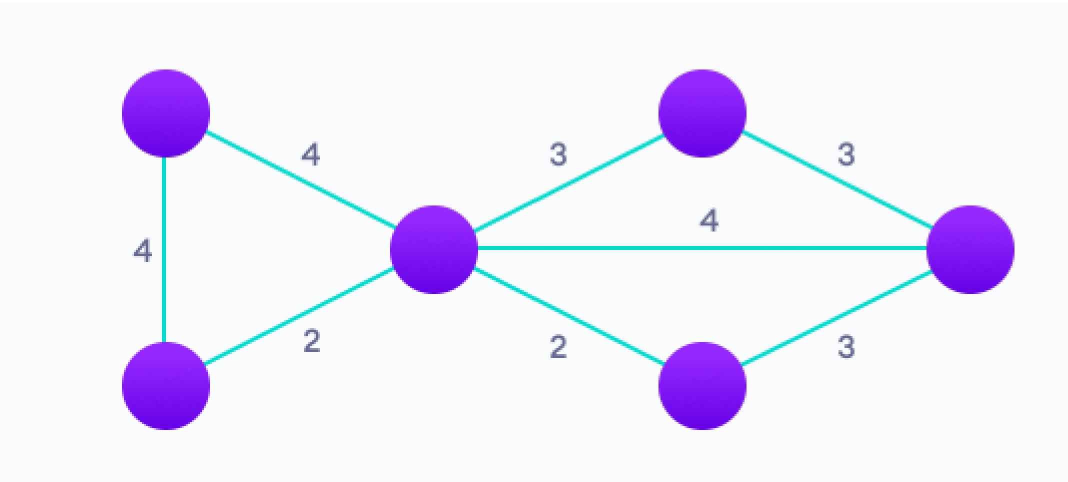
Si vous avez correctement codé votre algorithme, l'output de celui-ci avec comme input notre graphe et le sommet 'A' devrait être ['A', 'B', 'C', 'F', 'D', 'E']

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 arrêts 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 tout 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

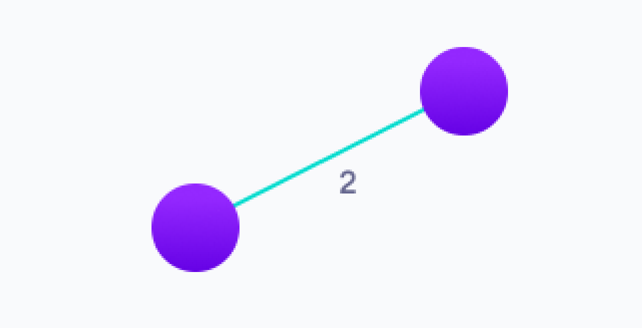
L'algorithme de Kruskal fonctionne de la façon suivante :

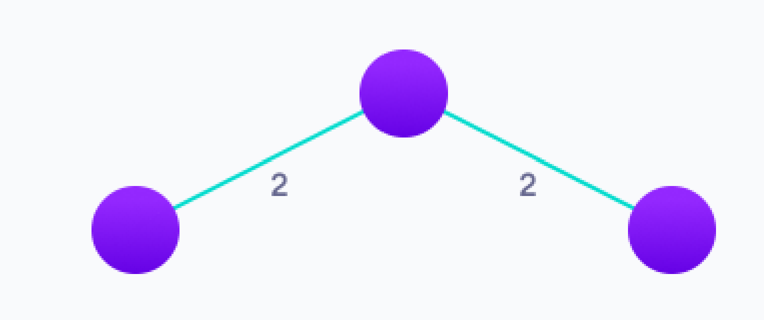
1. Classer les arêtes par ordre croissant de poids.
2. Prendre l'arête avec le poids le plus faible et l'ajouter à l'arbre (si 2 arêtes ont le même poids, choisir arbitrairement une des 2).
3. Vérifiez que l'arête ajouté ne crée pas de cycle, si c'est le cas, supprimez la.
4. Répétez les étapes 2) et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

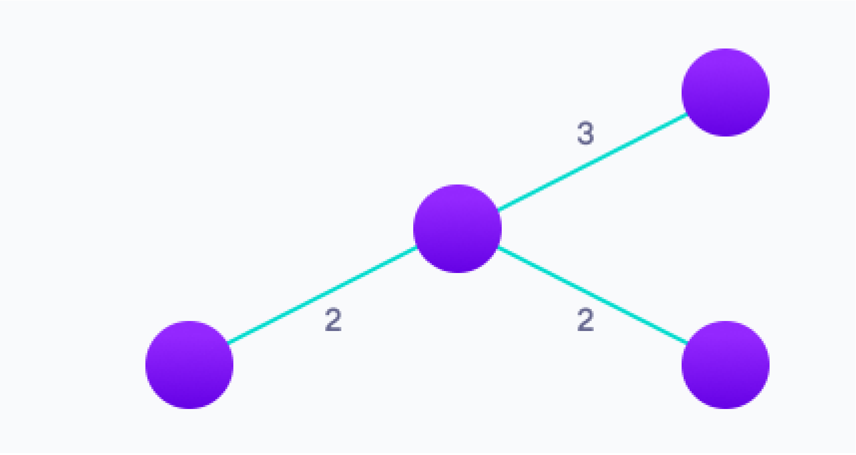
Un minimum spanning tree, s'il existe, a toujours un nombre d'arêtes égal au nombre de sommets moins un. Par exemple, ici notre graphe a 6 sommets. L'algorithme devrait donc s'arrêter lorsque 5 arêtes ont été choisies.

Solution

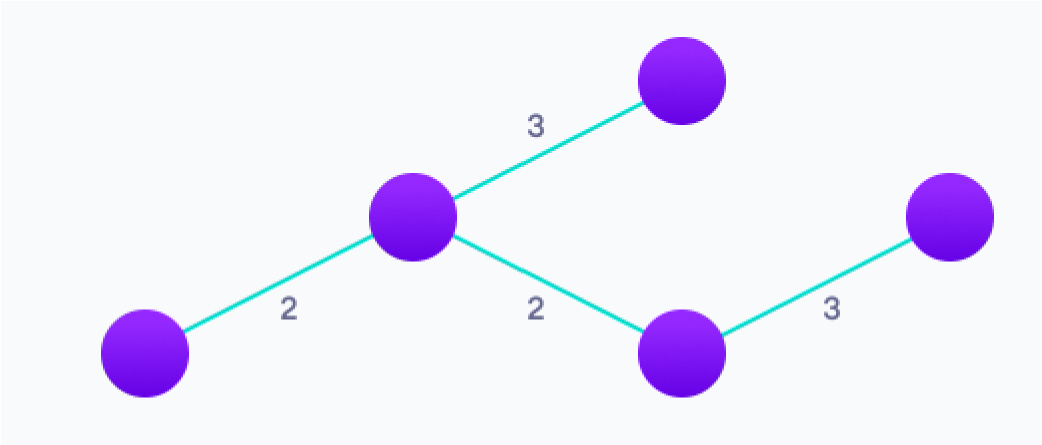
Vous trouverez ci-dessous les étapes de la construction du MST(Minimum spanning tree) :

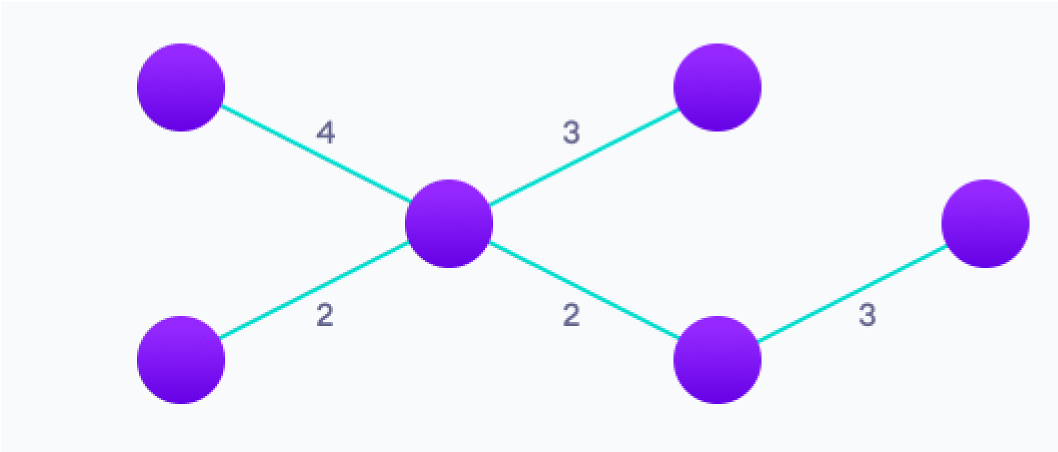






>_ Solution





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

Question 4: (🕒 *15 minutes*) **Algorithme de Kruskal : Python**

Vous trouverez ci-dessous L'algorithme de Kruskal implémenté en Python. Parcourez la fonction **kruskal_algo(Graph)** afin de vous assurez que vous ayez bien compris le fonctionnement :

```
#Question 3

class Graph:
    def __init__(self, vertices):#permet de créer un graphe lorsqu'on écrit p.ex Graph(6), il faut notamment indiquer le nb de sommet
        self.V = vertices
        self.graph = []

    def add_edge(self, u, v, w):#ajoute une arête entre le sommet u et v avec un poids w
        self.graph.append([u, v, w])

    def find(self, parent, i):#Correspond à la fonction Find-set(x) du cours
        if parent[i] == i:
            return i
        return self.find(parent, parent[i])

    def apply_union(self, parent, rank, x, y):#Correspond à la fonction Union(x,y) du cours
        xroot = self.find(parent, x)
        yroot = self.find(parent, y)
        if rank[xroot] < rank[yroot]:
            parent[xroot] = yroot
        elif rank[xroot] > rank[yroot]:
            parent[yroot] = xroot
        else:
            parent[yroot] = xroot
```

```
            rank[xroot] += 1

g = Graph(6)
g.add_edge(0, 1, 4)
g.add_edge(0, 2, 4)
g.add_edge(1, 2, 2)
g.add_edge(1, 0, 4)
g.add_edge(2, 0, 4)
g.add_edge(2, 1, 2)
g.add_edge(2, 3, 3)
g.add_edge(2, 5, 2)
g.add_edge(2, 4, 4)
g.add_edge(3, 2, 3)
g.add_edge(3, 4, 3)
g.add_edge(4, 2, 4)
g.add_edge(4, 3, 3)
g.add_edge(5, 2, 2)
g.add_edge(5, 4, 3)

def kruskal_algo(Graph):
    result = []#Permettra de stocker le résultat
    i, e = 0, 0 #Index utilisé dans l'algorithme

    Graph.graph = sorted(Graph.graph, key=lambda item: item[2]) #Trie les arêtes par poids croissant, étape 1)
    parent = []
    rank = []


    for node in range(Graph.V):#Cette boucle parcourt tout les sommets du graphes et crée un ensemble pour chacun d'entre eux
        parent.append(node)
        rank.append(0)

    #Tant que le nombre d'arêtes est inférieur à V-1, notre sous-graphe n'atteint pas tout les sommets -> on continue
    while e < Graph.V - 1:

        u, v, w = Graph.graph[i] #self.graph contient les arêtes par ordre croissant de poids, on commence avec i = 0
        i = i + 1           #puis à l'itération suivante on voudra avoir la 2ème arête la plus légère, donc on
                            #incrémente.

        x = Graph.find(parent, u)#Ces 2 lignes des codes permettent de rechercher et de stocker à quel ensemble
        y = Graph.find(parent, v)#appartiennent u et v.

        if x != y: #Si u et v font déjà parti du minimum spanning tree, i.e. u et v appartiennent au même ensemble
                #Alors on ne veut pas ajouter cette arête au minimum spanning-tree, d'ou le x!=y
            e = e + 1 #Si u et v sont d'ensemble différent, on a atteint un sommet de plus donc on incrémente
            result.append([u, v, w])#On ajoute la nouvelle arête au résultat
            Graph.apply_union(parent, rank, x, y)#On fusionne l'ensemble auquel appartient v à celui auquel apparient u
    for u, v, weight in result:
        print("%d - %d: %d" % (u, v, weight))#méthode permettant d'imprimer le résultat

    return result

kruskal_algo(g)
```

Conseil

La partie class **Graph** est une notion que vous verrez dans les chapitres dédiés à la programmation orientée objet. Pour le moment, il n'est pas important de comprendre son fonctionnement. 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 :

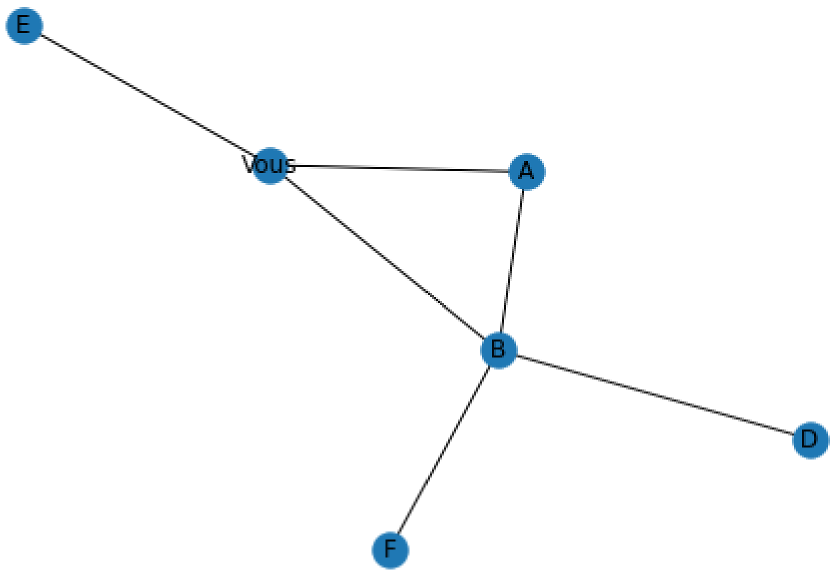
1. A quoi correspondent les arêtes et les nœuds ?
2. Faut-il utiliser un graphe dirigé ?

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.

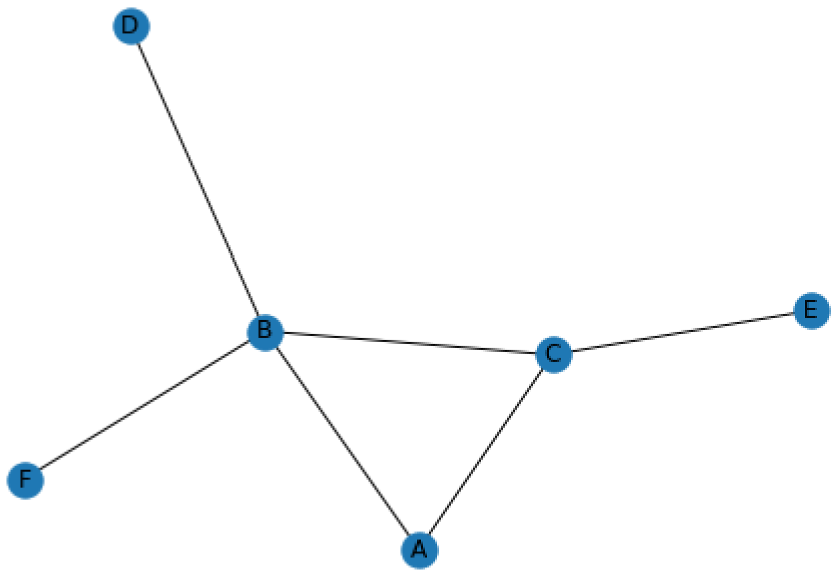
Pour trouvez les éléments 1) à 3), il faut raisonner de la façon suivante :

1. Trouvez le sommet relié à vous qui à 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.

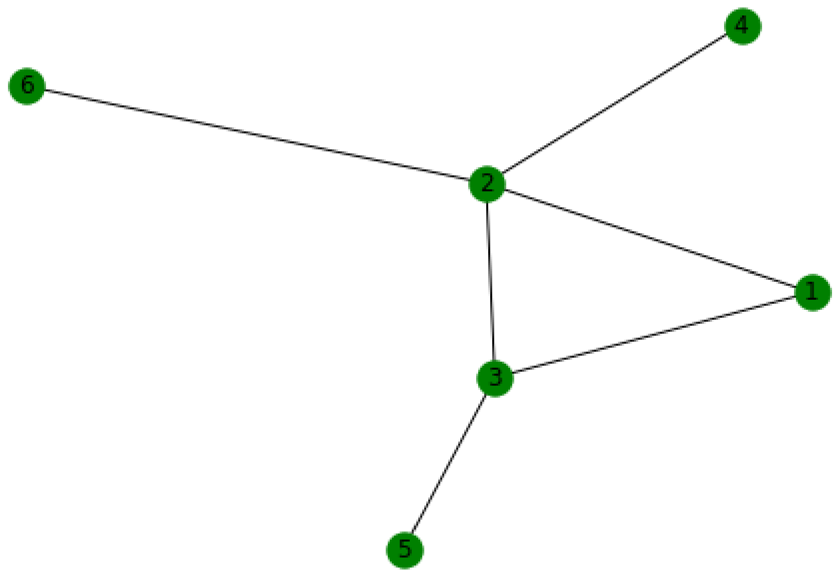
Si l'on ajoute un personne qui est ni un ami, ni l'ami d'un ami, alors aucune arête n'est reliée avec ce sommet. Par conséquent, ce sommet a un degré 0.

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

```
Graphe de Facebook :
```



```
Graphe de Whatsapp :
```



Conseil

Quelques hints 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)

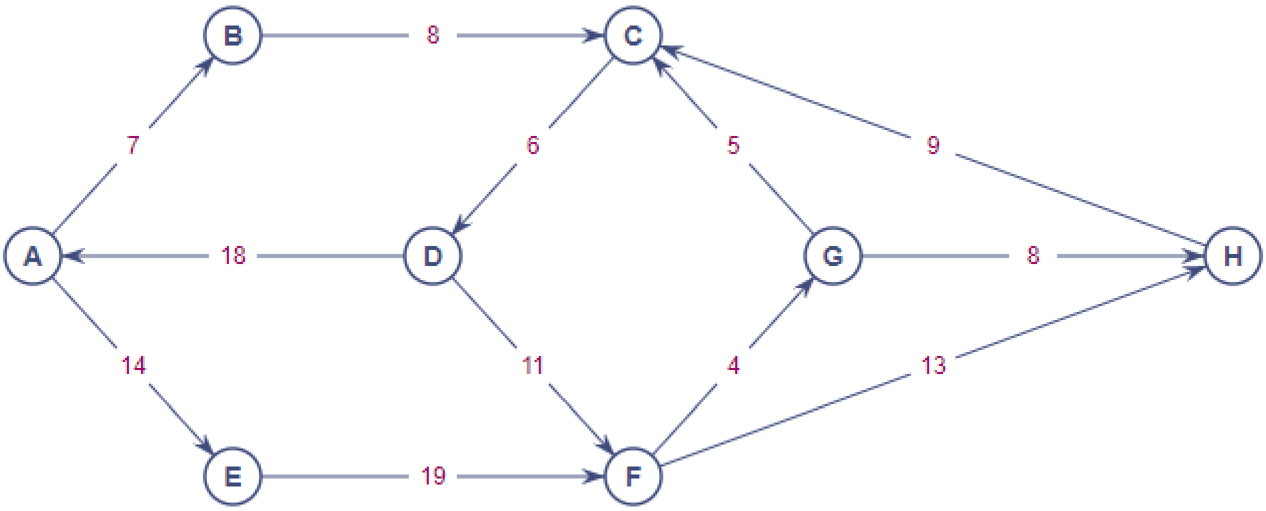
Les sommets D et F et 4 et 6 ne peuvent pas être dissociés. On ne peut donc pas savoir qui correspond à qui.

3 Algorithme de Dijkstra : Python

Nous allons nous intéresser à l'algorithme de Dijkstra qui permet de calculer le chemin le plus court entre 2 sommets d'un graphe. Cet algorithme est par exemple utilisé par les systèmes GPS. Les questions de cet exercice sont à remplir sur le fichier Exercice3.py disponible sur Moodle dans le dossier Ressources.

Question 7: (🕒 *10 minutes*) **Un petit échauffement**

Avant de rentrer dans le vif du sujet, nous allons devoir passer par quelques étapes préliminaires afin que vous puissiez coder l'algorithme par vous même. Considérez le graphe suivant :



Ouvrez le fichier Exercice3.py, et prenez connaissance du code, votre objectif sera de le compléter. Premièrement, représentez le graphe sous la forme d'une **adjacency matrix**.

Conseil

Représentez la matrice avec les colonnes et les lignes par ordre alphabétique.

Solution

```
#Question 7

Adjacency_matrix = [[0,7,i,i,14,i,i,i ],
               [i,0,8,i,i,i,i,i  ],
               [i,i,0,6,i,i,i,i  ],
               [18,i,i,0,i,11,i,i],
               [i,i,i,i,0,19,i,i ],
               [i,i,i,i,i,0,4,13 ],
               [i,i,5,i,i,i,0,9  ],
               [i,i,9,i,i,i,i,0  ]]
```

Question 8: (🕒 *10 minutes*) **Des outils utiles**
La première étape étant complétée, nous allons maintenant nous intéresser à comment récupérer des informations de notre graphe. Voici une liste non-exhaustive d'opérations que l'on peut effectuer :

1. get_node() permet d'accéder à un nœud. Par exemple, en faisant graphe.get_node('A'), j'obtiens des informations concernant le nœud A.
2. Lorsque l'on accède à un nœud par la fonction get_node(), on peut ensuite accéder à la liste de toutes les arêtes qui s'y connecte par l'attribut **relationships**. On peut par la suite distinguer les arêtes partant du nœud et les arêtes y arrivant à l'aide des attributs **relationship._from** et **relationship.to**.

L'exemple de code ci-dessous devrait vous aider à mieux comprendre :

```
#Question 8

A = graphe.get_node('A')
Arrete_liee_noeud_A = A.relationships #Cette variable continent une liste d'arrête
#A ce stade, vous ne savez pas encore comment "lire" ce que contient la variable Arrete_liee_noeud_A

print("Le nombre d'arrête liée à A est : {}.".format(len(Arrete_liee_noeud_A)))

#Cependant, vous pouvez voir que le sommet A est bien lié à 7 autres sommets. (Pour rappel, le sommet A est virtuellement
#lié à 7 sommets dans la matrice d'adjascence)

vertice = Arrete_liee_noeud_A[1] #On sélectionne la 2ème arrête liée au sommet A (pour rappele les indice d'une liste
#python commence à 0)

#Pour déterminer d'ou vient l'arrête et ou elle se termine, utilisez .to.value et _from.value :
print("L'arrête part du point : {}".format(vertice._from.value))
print("Et arrive au point : {}".format(vertice.to.value))

#Pour obtenir le poids de cette arrête, faites : vertice.value :
print("Le poids de l'arrête est : {}".format(vertice.value))
```

Pour vous assurez que vous avez bien compris cette partie avant de commencer, complétez la fonction linked du fichier Exercice3.py de sorte à ce que la fonction permette d'afficher tout les nœuds **partant** d'un sommet donné et d'afficher le poids de l'arête reliant ces 2 sommets.

Conseil

Quelques hints pour écrire ce programme :

1. Vous devrez retirez les sommets reliés par une arête avec un poids de 99999
2. Utilisez une boucle for pour parcourir les arêtes

Votre output devrait être A 18, F 11.

Solution

```
# Question 8

def linked(graph,N):
    N = graph.get_node(N)#Accède au noeud N, permet par la suite d'en récupérer les infos

    relationships = N.relationships #Accède à toutes les relations du point N

    for rel in relationships:#Parcout les relations du point N
        if rel.value == 99999:#Si le poids est de 99999 il n'y pas de relation dans le graphe --> itération suivante
            continue
        else :
            print(rel.to.value, rel.value)#Sinon on imprime la destination, puis le poids de l'arrête.

    return None
```

Question 9: (*15 minutes*) **Algorithme de Dijkstra** **Optionnel**

L'algorithme de Dijkstra permet de calculer le chemin le plus court entre 2 sommets d'un graphe. L'algorithme de Dijkstra que l'on va utiliser se construit de façon récursive. On initialise l'algorithme en partant au point de départ. Puis l'on va se déplacer vers tout les sommets atteignable depuis notre point de départ et appliquer l'algorithme de dijkstra à ces voisins. Ainsi de suite jusqu'à ce que l'on atteigne le sommet de destination. Pour éviter de créer une boucle infinie à cause des cycles, nous appliquerons uniquement Dijkstra aux voisins qui n'ont pas encore été visité.

Complétez la fonction dijkstra du fichier Exercice3.py.

Conseil

Quelques conseils pour l'implémentation de l'algorithme :

1. Le chemin le plus court entre le sommet A et H devrait être ABCDFGH.
2. Lorsque vous appliquerez Dijkstra aux voisins d'un sommet, pour choisir le chemin optimal vous devrez additionez la distance entre le sommet de départ et le poids du chemin fourni par Dijkstra.
3. Attention, lorsque vous devez déterminer quels voisins sont atteignables, ceux dont le poids est de 99999 ne sont pas atteignables.
4. L'algorithme doit retourner un tuble contenant la distance totale du trajet et le trajet.

Solution

```python
#Question 9

from math import inf

def dijkstra(origin,destination,visited = None):

  if visited is None:#A l'initialisation, l'ensemble des sommets atteints est vide
    visited = set()

  if origin.value == destination: #Si on est arrivé à destination, terminer l'algorithme => return
    return(0, origin.value)

  distance = inf
  path = origin.value
  visited.add(origin.value)#Ajoute le point à l'enseble des sommets visités

  for relationship in origin.relationships:

    if relationship.value == 99999:
      continue

    neighbour = relationship.to #Les voisins atteignables


    if neighbour.value not in visited:#Si un des voisins n'a pas encore été visités
      distance_temp, path_temp = dijkstra(neighbour, destination, visited) #On se déplace sur ce point et fait l'algo à partir de ce points.


      total_distance = distance_temp + relationship.value #Distance du chemin optimal à partir du neighbor + distance entre le point de départ et le neighbour.



      if total_distance < distance: #Si le chemin en question est meilleur que les précédents, on le sauvegarde.
        distance = total_distance
        path = origin.value + path_temp

  return (distance, path)
```