Algorithmes et Pensée Computationnelle

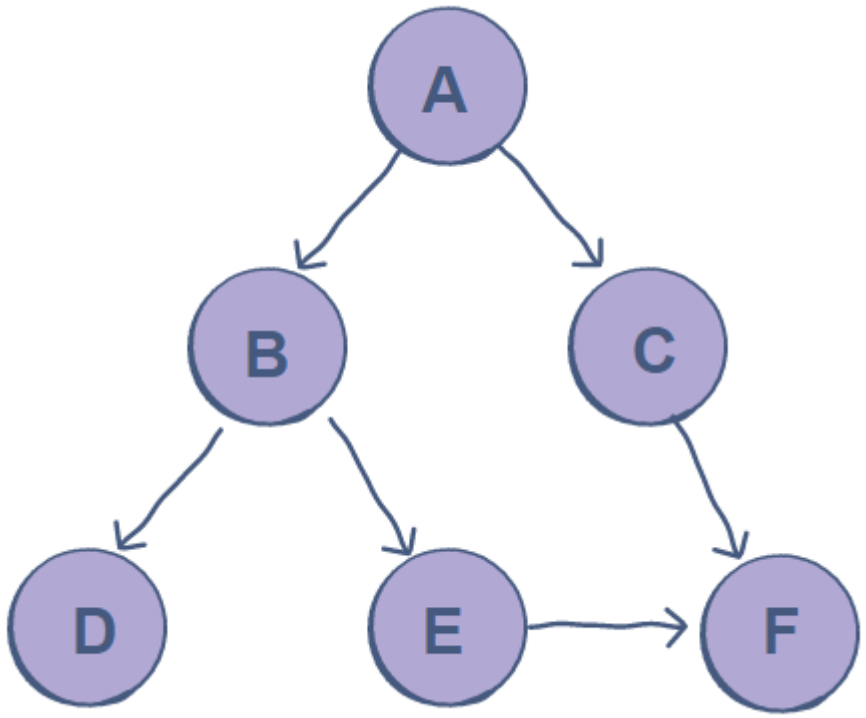
Algorithmes de graphes

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).

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 :

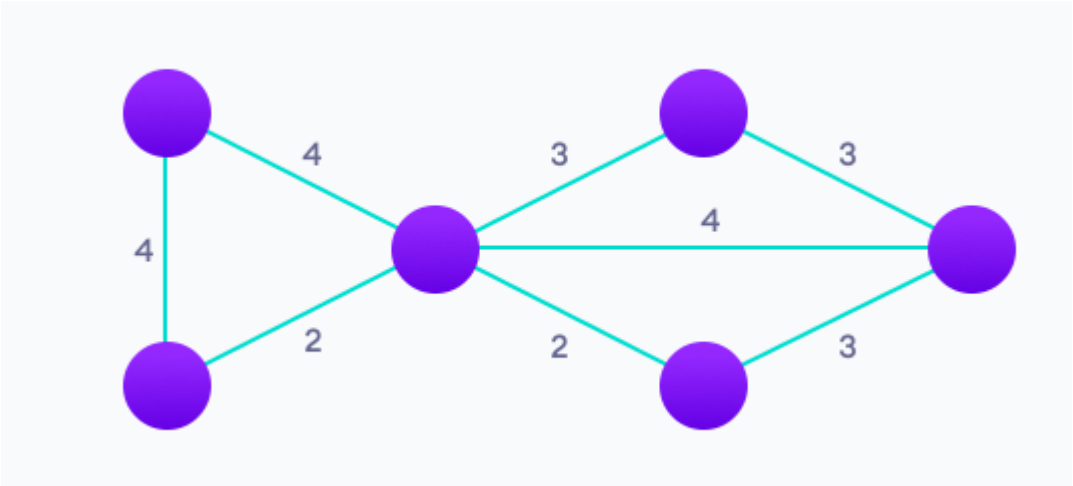
1. Utilisez l'adjacency list.
2. Pour le point 3), utilisez une boucle **while** avec la condition appropriée.
3. Pour parcourir les sommets adjacents, utilisez une boucle for .
4. L'algorithme devrait retourner une liste contenant l'ensemble des sommets atteignables.
5. Vous pouvez utiliser l'image 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.

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.

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 :

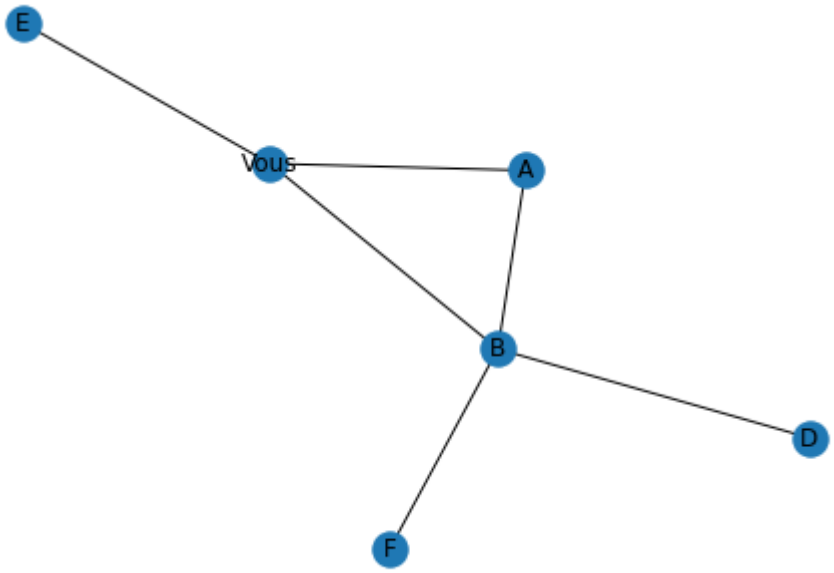
1. À quoi correspondent les arêtes et les nœuds ?
2. Faut-il utiliser un graphe dirigé ?

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 :

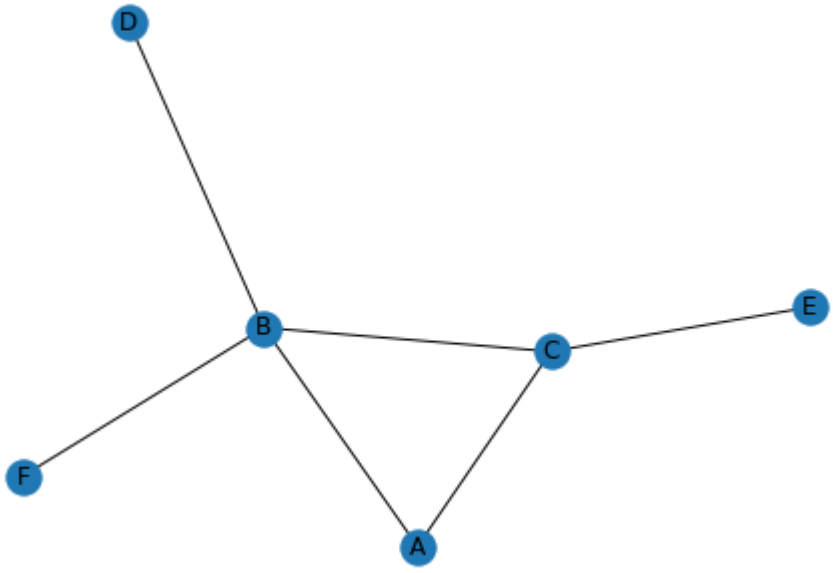


Conseil

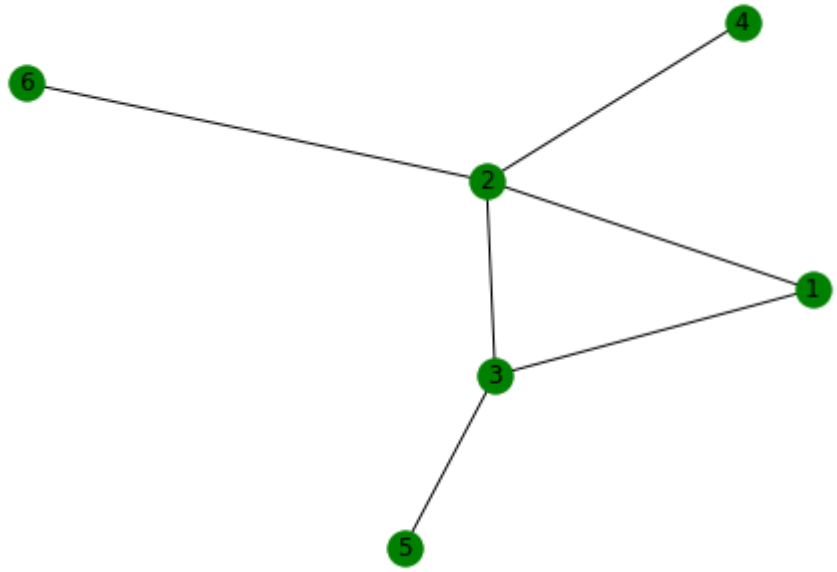
Réfléchissez en terme d'arêtes, de sommets, de degrés et de cycles.

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 :

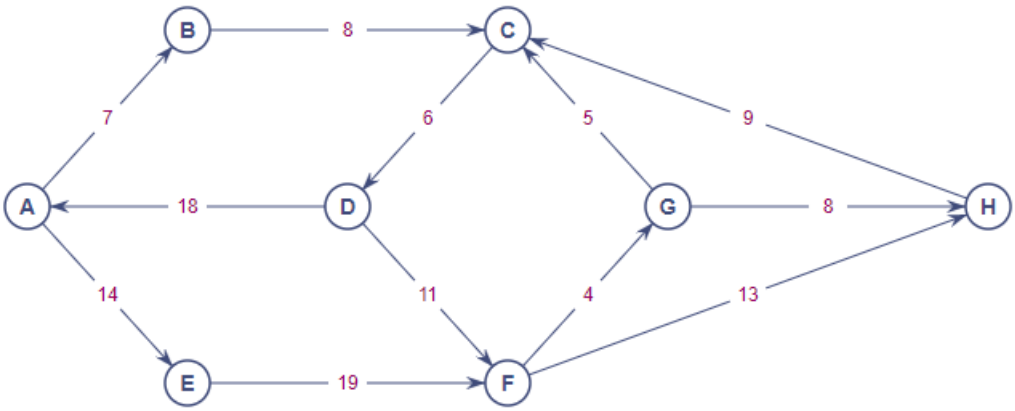
1. Identifiez les sommets des 2 graphes avec des caractéristiques semblables.
2. Il est possible que les sommets ne soient pas tous identifiables.

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 de navigation (GPS par exemple) pour trouver le chemin le plus court ou le moins coûteux entre 2 points. Les questions de cet exercice sont à remplir sur le fichier tp12.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 tp12.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.

Question 8: (*10 minutes*) **Outils pratiques**
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')
Arete_liee_noeud_A = A.relationships #Cette variable contient une liste d'arêtes

print("Le nombre d'arêtes liées à A est : {}.".format(len(Arete_liee_noeud_A)))

# 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'adjacence)

vertice = Arete_liee_noeud_A[1] #On sélectionne la 2ème arête liée au sommet A (pour rappel les indices d'une liste Python
    commence à 0)

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

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

Pour vous assurez que vous avez bien compris cette partie avant de commencer, complétez la fonction **linked** du fichier **tp12.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 conseils 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
>
> En faisant appel à votre fonction **linked()** avec les paramètres **graphe** (définie dans le fichier **tp12.py**) et **'D'**, vous devriez obtenir **A 18** et **F 11** qui correspondent aux nœuds liés à **D** et à leur poids.

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 à partir du point de départ. Puis, l'on va se déplacer vers tous les sommets atteignables depuis notre point de départ et appliquer l'algorithme de Dijkstra à ses 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 **tp12.py**. Pour y arriver, vous pouvez suivre les étapes suivantes :

1. L'algorithme de Dijkstra est un algorithme récursif. Nous l'avons vu lors des dernières semaines, il est nécessaire d'avoir une condition d'arrêt. L'algorithme appelle la fonction **dijkstra** de façon récursive en spécifiant l'origine (qui changera à chaque appel de la fonction **dijkstra**) et la destination. Quelle condition semble appropriée pour terminer l'appel récursif ? Implémentez cette condition dans la partie **Question 9.1**
2. Maintenant, nous allons avoir besoin d'une boucle parcourant toutes les relations voisines à notre point d'origine. Pour rappel, *origin.relationships* donne la liste des relations (sommet d'arrivée et poids de l'arête) à partir d'un sommet de départ. Implémentez cette boucle dans la partie **Question 9.2**.
3. À présent, nous pouvons sauter certaines itérations. Pour rappel, nous avons utilisé une matrice d'adjacence pour représenter notre graphe. Mais certaines relations ont vu leur poids arbitrairement assigné à 99999, pour signifier que 2 sommets n'étaient pas reliés. Implémentez une condition qui permet de passer à l'itération suivante de la boucle dans le cas où le poids de l'arête entre l'origine et le voisin considéré est égal à 99999. Pour accéder au poids d'une relation, utilisez la syntaxe *relationship.value*. Implémentez cette condition sous la section **Question 9.3**.
4. La partie *Appel récursif de la fonction* va faire un appel récursif à la fonction Dijkstra et va retourner le chemin le plus court entre le voisin du point d'origine et le point de destination. Dans les variables *distance_temp* et *path_temp* sont contenues respectivement le poids total du chemin le plus court entre le voisin considéré et la destination et l'intitulé de ce chemin (par exemple : ['BCDHG']). Dans la variable *total_distance*, on stocke la distance du chemin qui partirait de notre point d'origine, passerait par le voisin considéré et ensuite suivrait le chemin le plus court contenu dans la variable *path_temp* et dont le poids correspond à *distance_temp*. Votre mission est maintenant d'implémenter un bout de code qui permet de stocker le poids total du meilleur chemin, ainsi que son intitulé (par exemple : ['ABDJF'] Pour cela utilisez la variable distance qui a été initialisée à une valeur infinie. Faites cela dans la partie **Question 9.4**.

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 additionner 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 tuple contenant la distance totale du trajet et le trajet.