



MATHEMATIQUES DISCRETES  
PYTHON

# Algorithme de Kruskal

Enzo Flayac  
Gus Farine

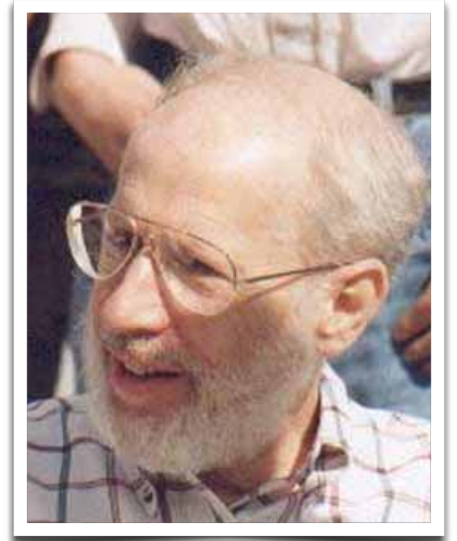
# Sommaire :

<b>Histoire :</b>	<b>3</b>
<b>Théorie des graphes :</b>	<b>3</b>
Les graphes	3
Graphe orientés et pondérés	3
Connexité des graphes	4
Algorithme de kruskal	4
<b>Application de l'algorithme</b>	<b>4</b>
Problématique	4
Application manuelle	5
Application Python	6
Résultat :	9
<b>Conclusion</b>	<b>9</b>

# Histoire :

Joseph Kruskal est un mathématicien statisticien et informaticien américain du 20<sup>e</sup> me siècle.

En 1956, il conçoit l'algorithme de Kruskal, un algorithme qui permet de trouver l'arbre couvrant minimum dans un graphe connexe non orienté et pondéré.

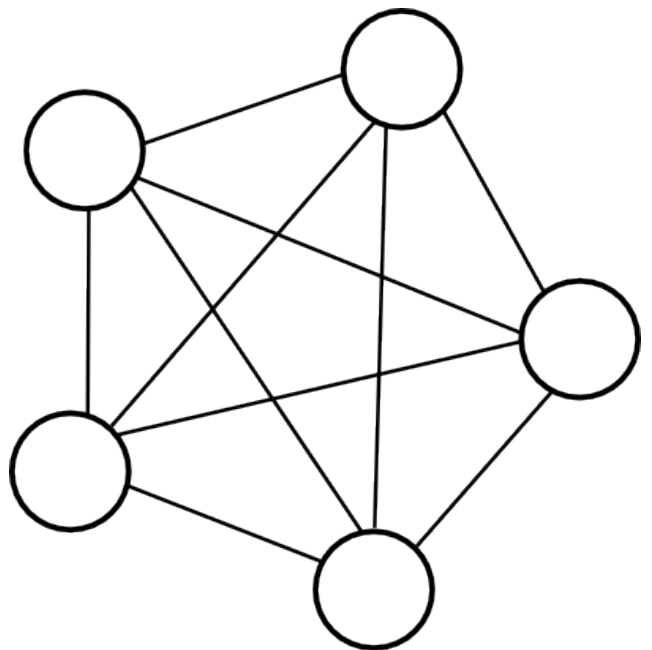


# Théorie des graphes :

## Les graphes

Un graphe est un modèle mathématique constitué de sommets et d'arêtes. Ils ont de multiples utilisations tel que dans les transport, les réseaux routier, le métro, réseau de transport et les réseaux ferroviaire ainsi que les télécommunications.

Par exemple dans un réseau ferroviaire, les sommets peuvent représenter des gare et les arrêtes représente les chemins de fer les reliant.

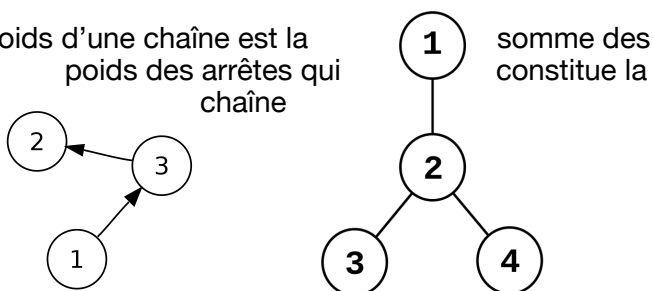


## Graphe orientés et pondérés

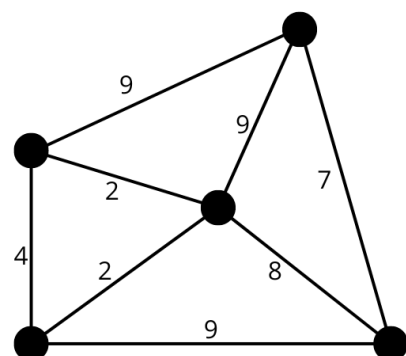
Un graphe est orienté si ses arêtes ne peuvent être parcourues que dans un sens. L'orientation des arêtes est indiquée par des flèches sur les arêtes.

Un graphe pondéré est un graphe étiqueté dont toutes les étiquettes sont des nombres réels positif ou non nul. On appelle ces nombres les poids des arrêtes.

Le poids d'une chaîne est la somme des poids des arrêtes qui constitue la chaîne



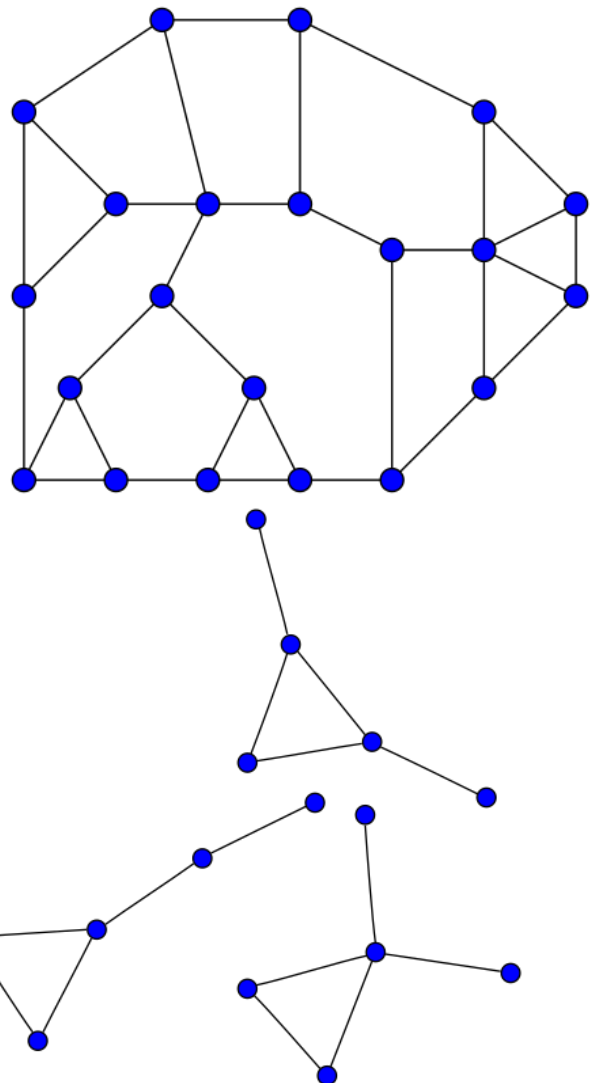
$$P(U,V) = 8+9+9 = 26$$



# Connexité des graphes

Un graphe non orienté est dit connexe si pour tout sommet U et V du graphe il existe une chaîne les reliant.

Pour un graphe non connexe, les sommets U et V ne pourront pas être reliés par une chaîne.



## Algorithme de kruskal

L'algorithme de Kruskal permet donc de trouver l'arbre couvrant minimum d'un graphe connexe non orienté et pondéré .

L'arbre couvrant minimum est un moyen de relier tout les sommets entre eux dont la somme des poids de chaque arrêtes est minimale . Dans une application pratique, il peut servir à établir un réseaux électrique ou de télécommunications en utilisant le moins câble possible et de réduire les coûts.

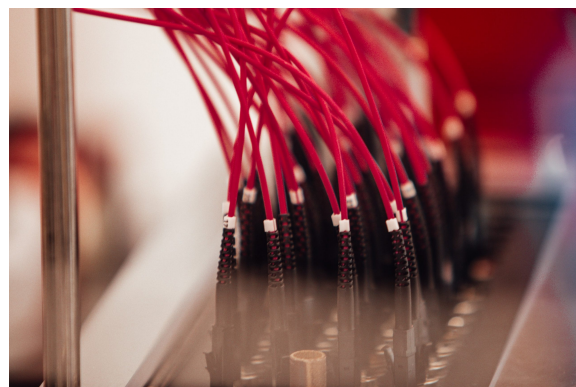
# Application de l'algorithme

## Problématique

De nos jours l'accès à internet est devenu primordial. Or celui ci devient de plus en plus rapide notamment grâce aux nouvelles technologies tel que la fibre optique.

Celle ci se présente sous forme de câble optique et permet d'atteindre des vitesse de connexion à très haut débit, la longueur du chemin parcouru n'influe donc plus sur la vitesse. Le chemin le plus court n'est donc pas le plus optimale dans ce cas si.

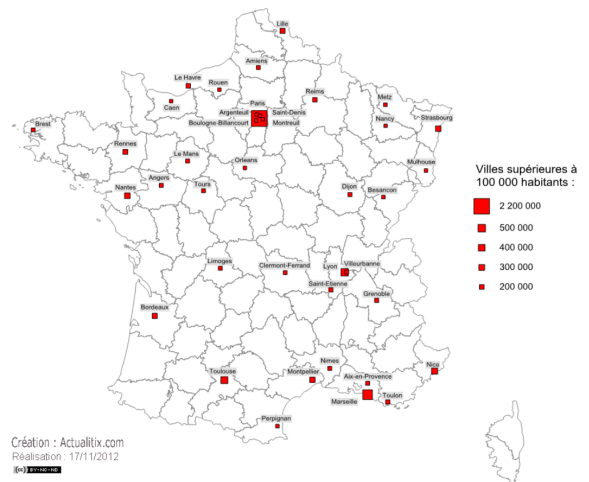
Nous allons donc nous demander, comment relier toutes les grandes villes de France en utilisant le moins de câble possible afin de créer le réseau le plus simple et le moins cher possible.



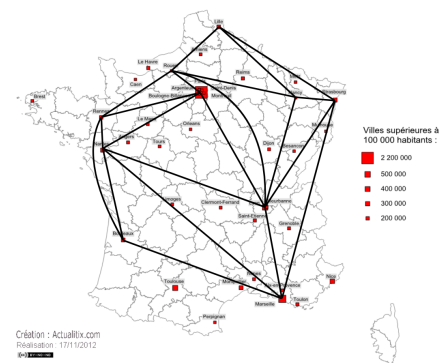
# Application manuelle

Ici nous avons la majorité des grandes villes de France. Nous ne les utiliseront pas tous pour simplifier la démarche.

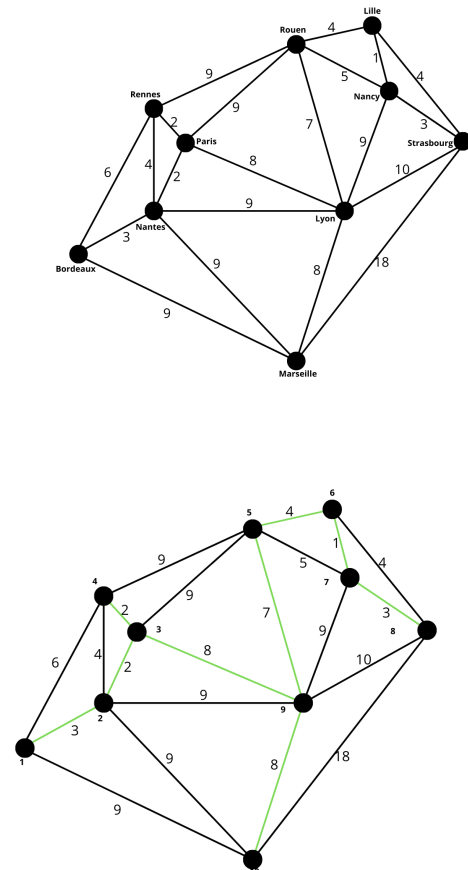
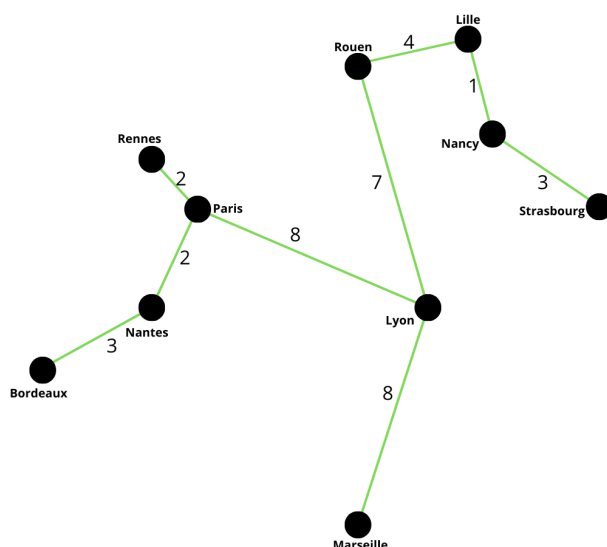
Tout d'abord, si relient toutes les villes entre elle nous obtenons un graphe compliqué à lire. Nous l'avons donc transformé en graphe plus lisible, c'est le même mais nous avons décalé les sommets par rapport aux autres.



Nous obtenons donc un graphe non orienté et pondéré ou les poids entre les arrêts représente la distance entre les ville. Nous observons que si nous relient toutes les villes au hasard nous obtenons un graphe avec beaucoup d'arêtes ce qui représente une longueur de câble très élevée et pas très optimisé. Par exemple pour aller de Marseille à Nantes il existe plus de 2 chemins différents. Hors puisque la fibre optique est extrêmement rapide, la longueur du chemin n'importe pas le temps de déplacement des informations. Donc avoir 2 chemins est inutile . Pour remédier à cela nous allons utiliser l'algorithme de kruskal afin d'obtenir l'arbre couvrant minimum . Tout d'abord nous allons chercher le poids minimum existant sur une arrête



L'idée principale derrière l'algorithme de Kruskal est de trier toutes les arêtes du graphe en fonction de leur poids. Cette étape permet d'identifier facilement les arêtes les moins coûteuses qui peuvent être ajoutées à l'arbre couvrant minimal sans créer de cycle.



De nombreuses implémentations utilisent un algorithme de tri rapide, tel que le tri par fusion ou le tri rapide, pour effectuer cette opération en un temps raisonnable.

Une fois les arêtes triées, l'algorithme de Kruskal ajoute progressivement les arêtes les moins coûteuses à l'arbre couvrant minimal, en s'assurant de ne pas créer de cycles. Pour ce faire, il maintient une structure de données appelée « ensemble disjoint » pour garder une trace des composantes connexes actuelles de l'arbre couvrant minimal. Lorsqu'une arête est ajoutée, elle relie deux sommets de deux composantes connexes distinctes. Si ces deux composantes étaient déjà connectées, cela signifierait que l'ajout de cette arête crée un cycle et doit donc être évité.

L'algorithme de Kruskal poursuit la sélection et l'ajout d'arêtes jusqu'à ce qu'un arbre couvrant minimal soit trouvé. Cela signifie généralement qu'il a ajouté  $(n-1)$  arêtes, où  $n$  est le nombre de sommets du graphe. À la fin de cette étape, l'arbre couvrant minimal est une sous-graphe acyclique connectée contenant tous les sommets du graphe initial et dont la somme des poids des arêtes est minimale.

## Application Python

Tout d'abord nous importons les différentes bibliothèques pour le calcul et la réalisation du graphe.

```
import networkx as nx # Importation de la bibliothèque NetworkX pour la manipulation de graphes
import matplotlib.pyplot as plt # Importation de la bibliothèque matplotlib pour le traçage de graphiques
import numpy as np # Importation de la bibliothèque numpy pour les calculs numériques
```

Ensuite, nous définissons une classe pour la création d'une arête, nous verrons par la suite l'utilisation de cette classe.

```
class Arete: # Définition de la classe Arete pour représenter une arête dans un graphe

    def __init__(self): # Définition du constructeur
        self.sommet_initial = 0 # Initialisation du sommet initial de l'arête
        self.sommet_final = 0 # Initialisation du sommet final de l'arête
        self.cout_arete = 0 # Initialisation du coût de l'arête

    def saisie_arete(self): # Méthode pour saisir les détails d'une arête
        self.sommet_initial = int(input("Entrez le sommet initial : ")) # Saisie du sommet initial
        self.sommet_final = int(input("Entrez le sommet final : ")) # Saisie du sommet final
        self.cout_arete = int(input("Entrez le coût de l'arête : ")) # Saisie du coût de l'arête
        return (self.sommet_initial, self.sommet_final, self.cout_arete) # Retourne les détails de l'arête sous forme de tuple
```

Nous définissons une classe **Graphe** pour la construction du graphe et le calcul de l'algorithme.

```
class Graphe: # Définition de la classe Graphe pour représenter un graphe

    def __init__(self, aretes): # Définition du constructeur
        self.nombre_sommet = 0 # Initialisation du nombre de sommets dans le graphe
        self.arbre_couvant = [] # Initialisation de la liste des arêtes de l'arbre couvrant minimal
        self.liste_aretes = aretes
```

Encore dans cette classe **Graphe**, nous définissons une fonction de tri par coût dans la liste renvoyée par la fonction `saisie_graphe()`.

```
def tri_par_cout(self): # Méthode pour trier les arêtes par coût
    liste_aretes = self.liste_aretes # Récupération de la liste des arêtes du graphe
    liste_aretes = sorted(liste_aretes, key=lambda x: x[2]) # Tri de la liste des arêtes par le troisième élément (coût)
    print(liste_aretes) # Affichage des arêtes triées par coût
    return liste_aretes # Retourne la liste des arêtes triées par coût
```

Dans la classe **Graphe**, nous définissons une fonction `ajout_arbre_couvant()`, elle a pour intérêt de calculer l'arbre couvrant minimal, donc sans les cycles potentiel

```
def ajout_arbre_couvant(self): # Méthode pour trouver l'arbre couvrant minimal (algorithme de Kruskal)
    listes_triees = self.tri_par_cout() # Appel de la méthode tri_par_cout pour trier les arêtes par coût

    for arete in listes_triees: # Parcours des arêtes triées par coût
        if len(self.arbre_couvant) == 0: # Si l'arbre couvrant est vide
            self.arbre_couvant.append(arete) # Ajout de l'arête à l'arbre couvrant
        else:
            cycle = False # Initialisation d'un indicateur de cycle à False
            for edge in self.arbre_couvant: # Parcours des arêtes de l'arbre couvrant
                if self.detect_cycle(arete, self.arbre_couvant):
                    cycle = True # S'il y a un cycle, on met l'indicateur à True
                    break
            if not cycle: # Si aucun cycle n'a été détecté
                self.arbre_couvant.append(arete) # Ajout de l'arête à l'arbre couvrant

    print("Arbre couvrant minimal (Kruskal) :", self.arbre_couvant) # Affichage de l'arbre couvrant minimal
```

Dans la classe **Graphe**, nous définissons la fonction pour détecter un éventuelle cycle dans l'arbre couvrant.

```
def detect_cycle(self, arete, arbre_couvant): # Méthode pour détecter les cycles dans l'arbre couvrant

    sommets_visites = set() # Initialisation d'un ensemble pour stocker les sommets visités
    for edge in arbre_couvant: # Parcours des arêtes de l'arbre couvrant
        sommets_visites.add(edge[0]) # Ajout du sommet initial à l'ensemble
        sommets_visites.add(edge[1]) # Ajout du sommet final à l'ensemble

    visited = set() # Initialisation d'un ensemble pour stocker les sommets visités
    queue = [arete[0]] # Initialisation d'une file avec le sommet initial de l'arête

    while queue: # Tant que la file n'est pas vide
        sommet = queue.pop(0) # Retrait du premier élément de la file
        if sommet == arete[1]: # Si le sommet est égal au sommet final de l'arête
            return True # Il y a un cycle, on retourne True
        visited.add(sommet) # Ajout du sommet à l'ensemble des sommets visités
        for edge in arbre_couvant: # Parcours des arêtes de l'arbre couvrant
            if edge[0] == sommet and edge[1] not in visited: # Si le sommet est le sommet initial de l'arête et que le sommet final n'a pas été visité
                queue.append(edge[1]) # Ajout du sommet final à la file
            elif edge[1] == sommet and edge[0] not in visited: # Si le sommet est le sommet final de l'arête et que le sommet initial n'a pas été visité
                queue.append(edge[0]) # Ajout du sommet initial à la file
```

Dans la classe **Graphe**, nous définissons une fonction pour créer la matrice d'adjacence de l'arbre couvrant minimal.

```
def adjacence(self): # Méthode pour générer la matrice d'adjacence de l'arbre couvrant

    liste_matrice = [] # Initialisation de la liste pour stocker la matrice d'adjacence
    nbr_arrete_arbre_couvrant = self.nbr_sommet_arbre_couvrant() # Récupération du nombre de sommets de l'arbre couvrant

    for i in range(nbr_arrete_arbre_couvrant): # Parcours des sommets de l'arbre couvrant
        initialisation_ligne = [] # Initialisation de la ligne de la matrice d'adjacence
        for j in range(nbr_arrete_arbre_couvrant): # Parcours des sommets de l'arbre couvrant
            initialisation_ligne.append(0) # Ajout de 0 à la ligne de la matrice d'adjacence
        liste_matrice.append(initialisation_ligne) # Ajout de la ligne à la liste de la matrice d'adjacence

    for i in range(nbr_arrete_arbre_couvrant - 1): # Parcours des arêtes de l'arbre couvrant
        liste_temp = [self.arbre_couvrant[i][0], self.arbre_couvrant[i][1]] # Récupération des sommets de l'arête
        if liste_matrice[list_temp[0] - 1][liste_temp[1] - 1] == 0 and liste_matrice[list_temp[1] - 1][liste_temp[0] - 1] == 0:
            liste_matrice[list_temp[0] - 1][liste_temp[1] - 1] = 1 # Ajout de 1 à la position correspondante dans la matrice
            liste_matrice[list_temp[1] - 1][liste_temp[0] - 1] = 1 # Ajout de 1 à la position correspondante dans la matrice

    return liste_matrice # Retourne la matrice d'adjacence de l'arbre couvrant
```

Dans la classe **Graphe**, nous définissons une fonction pour afficher la matrice d'adjacence, cette partie est facultative mais toujours plus agréable pour comprendre le graphe qui sort de l'algorithme.

```
def afficher_matrice(self): # Méthode pour afficher la matrice d'adjacence de l'arbre couvrant

    liste_matrice = self.adjacence() # Récupération de la matrice d'adjacence de l'arbre couvrant
    iteration = len(liste_matrice) # nombre d'itérations
    print("\nMatrice d'adjacence de l'arbre couvrant minimal :\n") # Affichage d'une ligne vide
    for i in range(iteration): # Parcours des lignes de la matrice
        for j in range(iteration): # Parcours des colonnes de la matrice
            print(liste_matrice[i][j], end=" ") # Affichage de chaque élément de la matrice
        print("\n") # Affichage d'une ligne vide entre chaque ligne de la matrice
```

Nous définissons une fonction en dehors de la classe Graphe. Cette fonction permet l'appelle de la fonction saisie\_arete vu au début du programme pour permettre la création d'un graphe entré par l'utilisateur, arête par arête avec le sommet initiale, le sommet finale et le cout de cette arête. Cette fonction retourne la liste de toutes les arêtes du graphe.

```
# Création d'un objet Arete
arete = Arete()

def saisie_graphe_depart():
    condition = input("Voulez vous saisir arete par arete ou utiliser le graphe natif ? [ oui / non ]\n-> ")
    if condition == "oui":
        nbr_sommet = int(input("Entrez le nombre de d'arrêtes de votre graphe : \n-> "))
        aretes = []
        for i in range(nbr_sommet):
            aretes.append(arete.saisie_arete())
        return aretes
    else:
        aretes = [
            # Initialisation d'une liste d'arêtes prédéfinies
            [1, 2, 3], [1, 4, 6], [1, 10, 9],
            [2, 3, 2], [2, 4, 4], [2, 10, 9], [2, 9, 9],
            [3, 4, 2], [3, 5, 9], [3, 9, 8],
            [4, 5, 9],
            [5, 6, 4], [5, 7, 5], [5, 9, 7],
            [6, 7, 1], [6, 8, 4],
            [7, 8, 3], [7, 9, 9],
            [8, 9, 10], [8, 10, 10],
            [9, 10, 8]
        ]
        return aretes

aretes = saisie_graphe_depart()
# Création d'un objet Graphe
graphe = Graphe(aretes)
# Appel de la méthode ajout_arbre_couvrant pour trouver l'arbre couvrant minimal
graphe.ajout_arbre_couvrant()
# Appel de la méthode afficher_matrice pour afficher la matrice d'adjacence de l'arbre couvrant
graphe.afficher_matrice()
```



Enfin, nous avons l'appelle de toute la classe graphe et des fonctions interne à cette classe.

```
aretes = saisie_graphe_depart()
# Création d'un objet Graphe
graphe = Graphe(aretes)
# Appel de la méthode ajout_arbre_couvant pour trouver l'arbre couvrant minimal
graphe.ajout_arbre_couvant()
# Appel de la méthode afficher_matrice pour afficher la matrice d'adjacence de l'arbre couvrant
graphe.afficher_matrice()

# Création d'un objet de graphe non dirigé
G = nx.Graph()

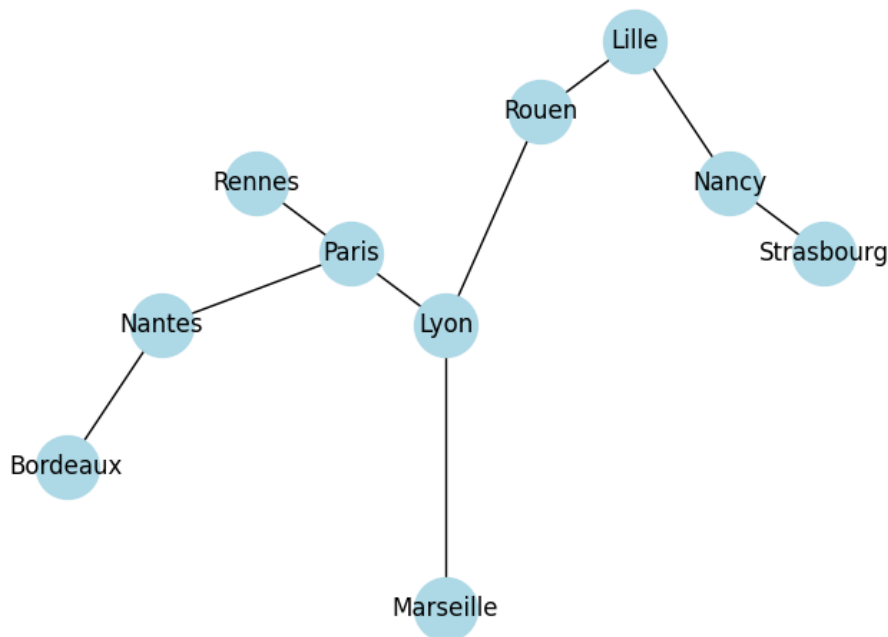
# Ajout des arêtes à partir de l'arbre couvrant minimal
for arete in graphe.arbre_couvant:
    G.add_edge(arete[1], arete[0])

# Dessin du graphe avec les noms des sommets
pos = nx.spring_layout(G) # Positionnement des nœuds
nx.draw(G, with_labels=True, node_color='lightblue', node_size=1000) # Dessin du graphe avec les noms des sommets
labels = nx.get_edge_attributes(G, 'weight') # Obtention des poids des arêtes
nx.draw_networkx_edge_labels(G, pos, edge_labels=labels) # Dessin des poids des arêtes

plt.title("Graphe de l'arbre couvrant minimal avec noms de sommets")
plt.show()

#Si vous voulez modifier le graphe de base avant que l'algorithme l'analyse et le traite vous pouvez modifier la liste ligne 24
```

## Résultat :



## Conclusion

Pour conclure, grâce au code python nous sont en capacité de recréer et d'utiliser le même algorithme. Cependant grâce à python cela est beaucoup plus rapide et beaucoup plus simple lorsque l'on utilise des graphes de plus en plus grands et les chances de se tromper sont donc réduites elles aussi.