MATHEMATIQUES DISCRETES
PYTHON

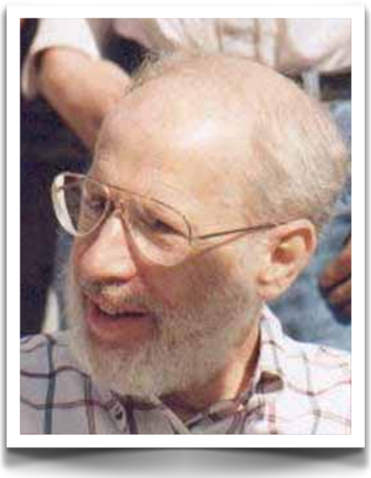
Algorithme de Kruskal

Enzo Flayac

Gus Farine

Sommaire :

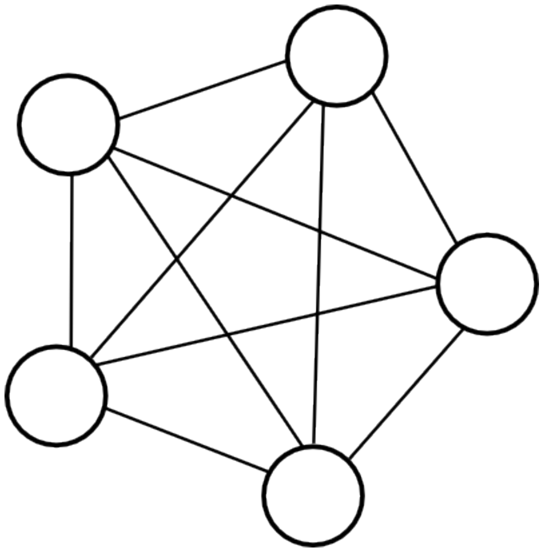
Histoire :



Joseph Kruskal est un mathématicien statisticien et informaticien américain du 20ème 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 encore utilisé dans de nombreux domaines de nos jours.

Théorie des graphes :

Les graphes



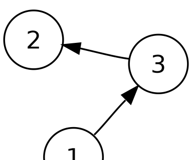
Un graphe est un modèle mathématique constitué de sommets et d'arêtes. Ils ont de multiples utilisations comme dans les transports, les réseaux routiers, le métro, les réseaux ferroviaires ainsi que les télécommunications.

Par exemple dans un réseau ferroviaire, les sommets peuvent représenter des gares, et les arêtes représentent les chemins de fer.

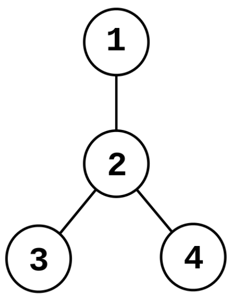
Graphes orientés et pondérés

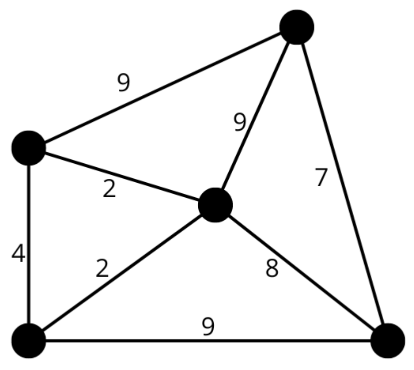
Un graphe est orienté si ses arêtes ne peuvent être parcourues que dans un sens. L'orientation des arêtes est indiquée par des flèches sur les arêtes.

Graphe orienté :



Graphe non orienté :





Un graphe pondéré est un graphe étiqueté dont toutes les étiquettes sont des nombres réels positifs ou non nuls. On appelle ces nombres les poids des arêtes.

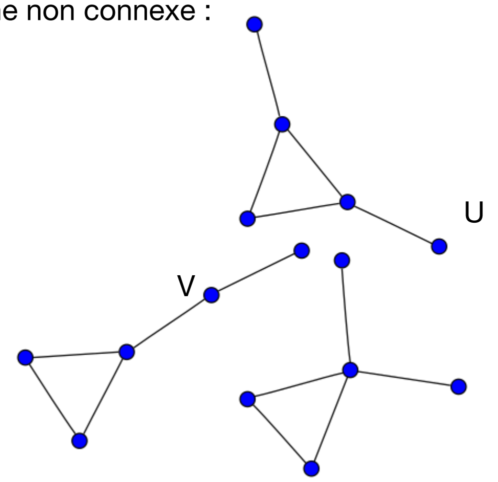
Le poids d'une chaîne est la somme des poids des arêtes qui constitue la chaîne.

Connexité des graphes

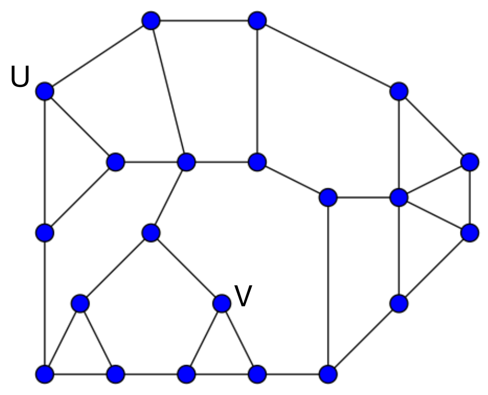
Un graphe non orienté est dit connexe si pour tout sommet U et V du graphe il existe une chaîne les reliants.

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

Graphe non connexe :



Graphe connexe :



Algorithme de kruskal

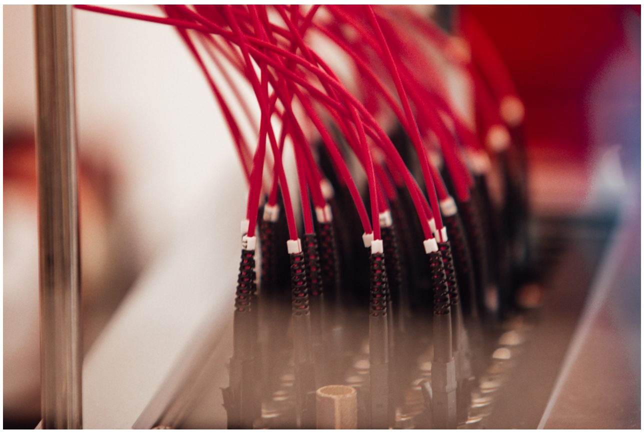
L'algorithme de Kruskal permet donc de trouver l'arbre couvrant minimum d'un graphe connexe non orienté et pondéré. Ce graphe comportera n-1 arêtes où n est le nombre de sommets.

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

Application de l’algorithme

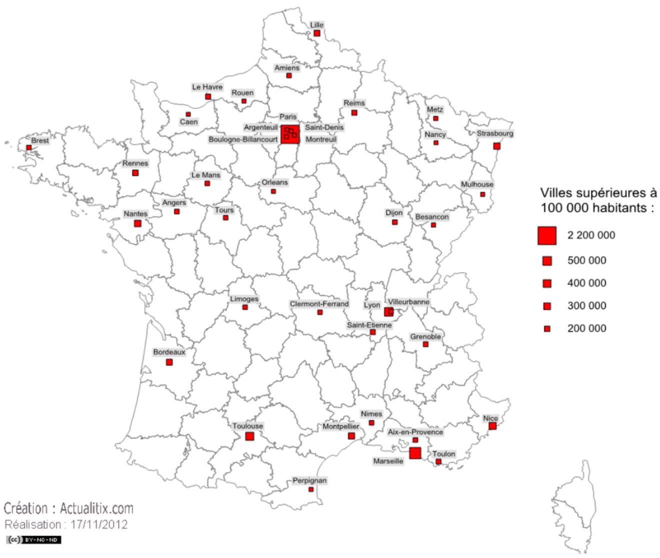
Problématique

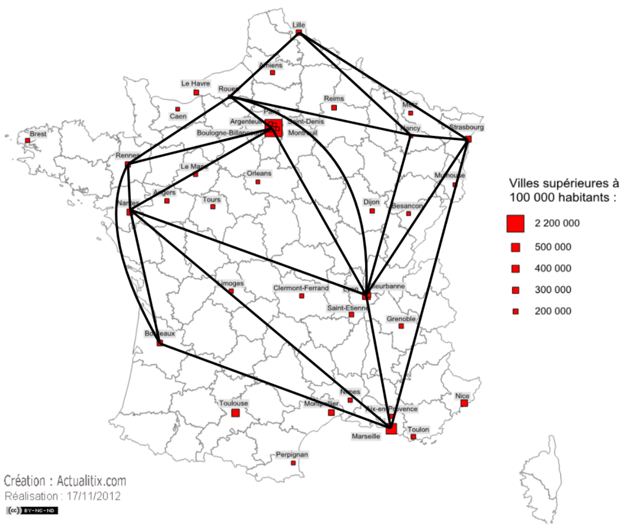
De nos jours l’accès à internet est devenu primordial. Or celui ci devient de plus en plus rapide notamment grâce aux nouvelles technologies telle que la fibre optique.
Celle ci se présente sous forme de câble optique et permet d’atteindre des vitesses de connexion à très haut débit, la longueur du chemin parcouru n’influe donc plus sur la vitesse. Le chemin le plus court n’est donc pas le plus optimal dans ce cas présent.
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

Sur cette carte, nous avons la majorité des grandes villes de France. Nous ne les utiliserons pas toutes pour simplifier la démarche.

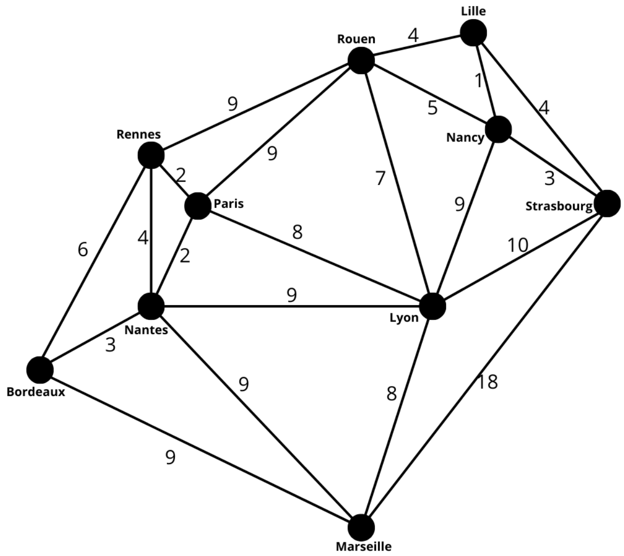


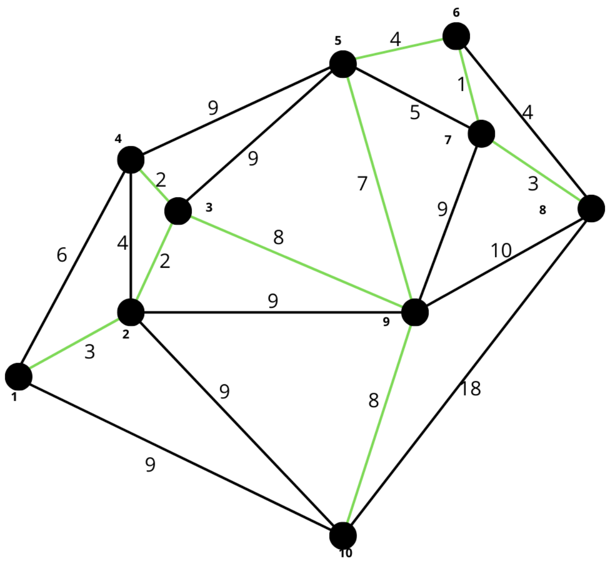


Nous observons que si nous relions toutes les villes entre elles nous obtenons un graphe compliqué à lire. Nous l’avons donc transformé en graphe plus lisible, c’est identique mais nous avons décalé les sommets par rapport aux autres afin qu’il soit plus agréable regarder.

Nous obtenons donc un graphe non orienté et pondéré où les poids entre les arêtes représentent la distance entre les villes.

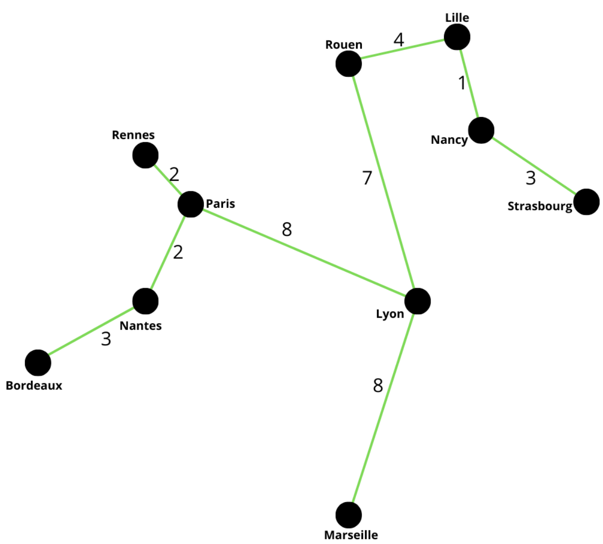
Si nous relions toutes les villes au hasard nous obtenons un graphe avec 21 arêtes ce qui représente une longueur de câble très élevée et pas très optimisée.
Par exemple pour aller de Marseille à Nantes il existe plus de 2 chemins différents.
Puisque la fibre optique est extrêmement rapide, la longueur du chemin n'importe pas le temps de déplacement des informations : le fait d'avoir 2 chemins est donc 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 arête et classer celles-ci.
Cela permet d'identifier facilement les arêtes les moins coûteuses qui peuvent être ajoutées à l'arbre couvrant minimal sans créer de cycle.

Une fois les arêtes triées, l'algorithme va ajouter progressivement les arêtes les moins coûteuses à l'arbre couvrant minimal, en s'assurant de ne pas créer de cycles.



L'algorithme de Kruskal poursuit la sélection et l'ajout d'arêtes jusqu'à ce qu'un arbre couvrant minimal soit trouvé. Nous obtenons normalement un graphe comportant n-1 arêtes, où n est le nombre de sommets. L'algorithme a donc déterminé l'arbre couvrant graphe connexe reliant tout les sommets entre eux avec le poids total des arêtes le plus faible.

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_couvrant(), elle a pour intérêt de calculer l’arbre couvrant minimal, donc sans les cycles potentiels.

```python
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 éventuel cycle dans l’arbre couvrant.

```python
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.

```python
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_couvant = self.nbr_sommet_arbre_couvant()  # Récupération du nombre de sommets de l'arbre couvrant

    for i in range(nbr_arrete_arbre_couvant):  # 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_couvant):  # 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_couvant - 1):  # Parcours des arêtes de l'arbre couvrant
        liste_temp = [self.arbre_couvant[i][0], self.arbre_couvant[i][1]]  # Récupération des sommets de l'arête
        if liste_matrice[liste_temp[0] - 1][liste_temp[1] - 1] == 0 and liste_matrice[liste_temp[1] - 1][liste_temp[0] - 1] == 0:
            liste_matrice[liste_temp[0] - 1][liste_temp[1] - 1] = 1  # Ajout de 1 à la position correspondante dans la matrice
            liste_matrice[liste_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 (variable) iteration: int  nombre d'itérations
        print("\nMatric                          vant 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("")  # Affichage d'une ligne vide entre chaque ligne de la matrice
```

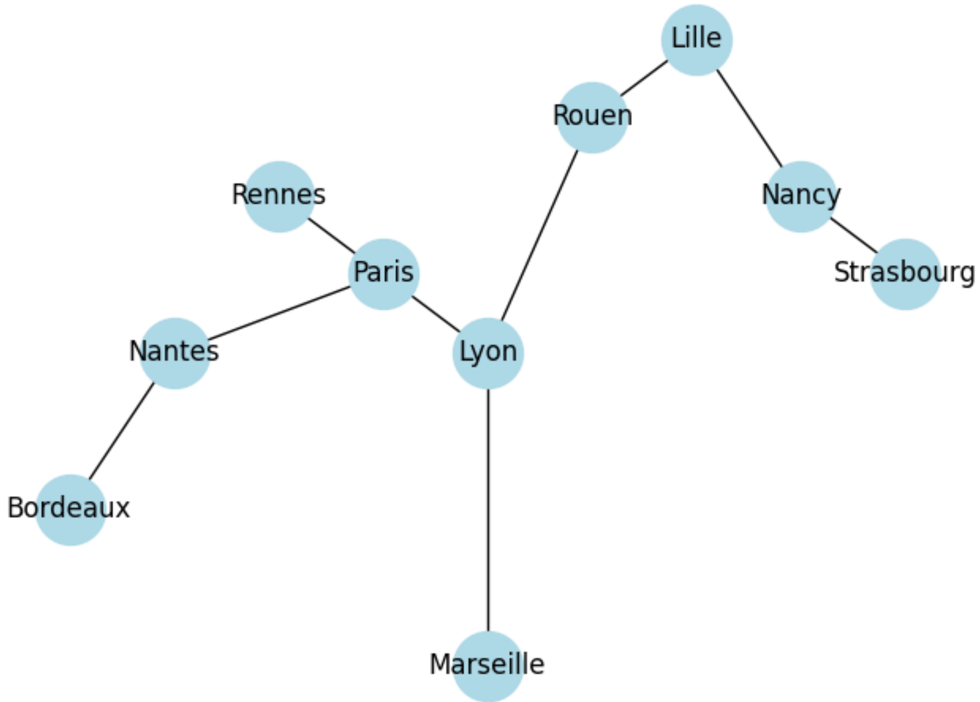
Nous définissons une fonction en dehors de la classe Graphe. Cette fonction permet l’appel de la fonction saisie_arete vue au début du programme pour permettre la création d’un graphe entré par l’utilisateur, arête par arête avec le sommet initial, le sommet final et le cout de cette arête. Cette fonction 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_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()
```

Enfin, nous avons l’appel de toute la classe graphe et des fonctions internes 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 sommes 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 ainsi, la probabilité d'erreur est réduite.
Un des problèmes majeur lors de nos recherches fut de trouver des informations sur l'origine de cet algorithme et de son créateur, car ce sujet n'est pas très référencé. Ce projet nous a cependant permis de découvrir et d'apprendre à utiliser les outils graphiques intégrés à Python afin de tracer les graphes. Cela nous a aussi permis de découvrir un algorithme très utile autant bien par son utilisation que pour notre culture générale.

Sources

https://www.pairform.fr/algorithme-kruskal.html

http://ressources.unit.eu/cours/EnsROtice/module_avance_thg_voo6/co/algoKruskal.html

https://datascientest.com/algorithme-de-kruskal-tout-savoir