Laetitia Berne
Gilles Deknache

Rapport de projet :

Optimisation sur graphe - Problème du voyageur de commerce

Semaine départementale - Septembre 2022

Plan :

I - Définition du problème
   a) Contexte
   b) Modélisation
   c) Hypothèses

II - Algorithmes de résolution

III - Résultats

I - Définition du problème

a) Contexte

Pour définir ce problème, rien de plus simple qu'une histoire.
Imaginons un voyageur qui souhaite visiter un grand nombre de villes en France. Cependant, il n'a pas un budget illimité et il souhaite réduire son temps de trajet au maximum. De plus, notre voyageur habite dans une ville ! Donc son trajet doit prendre en compte son retour à la maison. C'est pourquoi nous pouvons parler d'un tour. L'enjeu du problème du voyageur de commerce (travelling salesman problem ou TSP) est de trouver dans quelle ordre visiter les villes, le tout en minimisant sa distance de parcours.

b) Modélisation

Pour modéliser ce problème, nous allons utiliser un graphe orienté G(N,A), où N représente l'ensemble des villes à visiter (qui deviendront les sommets du graphe) et $A \subseteq N \times N$ l'ensemble des chemins entre ces villes (qui sont les arcs du graphe). Chacun des arcs possède une valeur non négative $d_{ij} \geq 0$ (avec $(i,j) \in A$), qui représente la distance (ou le temps de trajet) entre les différentes villes.
Pour résoudre ce problème, nous devons trouver un cycle, qui couvre tous les sommets du graphe.
Le TSP est un problème qui est facilement visualisable sur un graphe. On peut cependant le formaliser de manière rigoureuse.
On défini des variables binaires x_{ij} qui prennent la valeur 1 si l'arc (i, j) est utilisé dans la solution finale, 0 sinon.
On peut donc modéliser le problème comme suit (formulation de Dantzig-Fulkerson-Johnson) :

$$min \sum_{(i,j) \in A} d_{ij} x_{ij}$$

$$\sum_{j \in N \,/\, (j,i) \in A} x_{ij} = 1 \quad \forall i \in N \qquad \textit{arrivée au sommet i}$$

$$\sum_{j \in N \,/\, (i,j) \in A} x_{ij} = 1 \quad \forall i \in N \qquad \textit{départ du sommet i}$$

$$\sum_{(i,j) \in A \,/\, i,j \in S} x_{ij} \leq |S| - 1 \quad \forall S \subset N \,/\, |S| \geq 2$$

$$x_{ij} \in \{0\,;\,1\} \quad \forall (i,j) \in A$$

c) Hypothèses

Pour ne pas avoir recours à des algorithmes d'optimisation linéaire coûteux en temps, nous devons faire certaines hypothèses simplificatrices pour qu'un algorithme plus simple (c'est-à-dire avec une plus petite complexité) puisse le résoudre.

Les hypothèses faites sont les suivantes :
1- Le graphe doit être symétrique :
$\forall (i,j) \in A,\ d_{ij} = d_{ji}$

2- Le graphe doit être complet : tous les nœuds sont reliés entre eux par des arêtes autrement dit, pour tout nœuds distincts i et j, il existe une arête (i,j).

3- Le graphe doit être euclidien, c'est-à-dire qu'il doit respecter l'inégalité triangulaire, à savoir :
$\forall (i,j,k) \in A\ ,\ d_{ij} + d_{jk} \geq d_{ik}$

II- Algorithme heuristique de résolution

À l'aide de ces hypothèses on peut montrer que l'algorithme suivant est 2-approximé et heuristique ; c'est une méthode qui fournit rapidement une solution faisable - pas nécessairement optimale - pour un problème d'optimisation complexe. De plus, le résultat trouvé par l'algorithmique est au plus 2 fois supérieur à la solution optimale.

Maintenant, décrivons l'algorithme étape par étape !

Première étape :

On transforme le graphe orienté en un graphe non orienté.
Pour ce faire, il suffit de changer toutes les arêtes orientées i→j en arêtes non orientées.
On utilise pour cela la fonction python "to_undirected()"

Deuxième étape :

Ensuite, on résout le problème d'arbre de couverture à coût minimal (MST) sur le graphe non orienté.
Pour pouvoir résoudre ce problème, nous avons implémenté l'algorithme de Kruskal. Pour rappel, cet algorithme prend en entrée un graphe non orienté dont les arcs possèdent des poids et renvoie un arbre qui couvre tous les nœuds, dont la somme des poids des arcs conservés est minimale.
L'algorithme de Kruskal construit l'arbre pas à pas, en ajoutant un arc à la fois. Les arêtes sont ajoutées de la moins chère (celle qui a le plus petit poids) à la plus chère. Celles qui créent un cycle ne sont pas ajoutées. On s'arrête lorsqu'on a ajouté |N| - 1 arcs.

Pour l'analyse du code, on se basera sur le fichier Kruskal_algorithm.py que nous avons écrit.

Étapes a et b :

Nous créons un graphe initialement vide, ainsi qu'une liste qui contient les arcs du graphe original, sous forme de tuple. Les deux premiers éléments du tuple correspondent aux deux sommets de l'arête. Le dernier élément contient le poids de l'arête. Puis, la liste est triée selon le troisième élément du tuple, c'est-à-dire les poids. Arc_list contient donc tous les arcs du graphe, triés par poids croissant.

Nous créons de plus une liste vide linked_nodes, qui contiendra par la suite tous les nœuds déjà visités, donc déjà couverts par l'arbre.

Etape c :

Nous construisons maintenant l'arbre. On continue d'ajouter les arêtes tant que le nombre de sommets contenus dans notre arbre est inférieur à |N|-1. On parcourt la liste des arcs arc_list grâce à un compteur. Si l'arc que l'on considère possède ses sommets de départ et d'arrivée dans linked_nodes, on regarde s'il existe déjà un chemin entre ces deux nœuds. Si c'est le cas, on n'ajoute pas l'arc. S'il n'y a pas de chemin, on ajoute l'arc à l'arbre.
Si le sommet de départ ou d'arrivée de l'arc n'est pas dans linked_nodes, on l'y ajoute, ainsi que l'arc.

```
if arc_list[compteur][0] in linked_nodes :
    ajoute_noeud(T, arc_list, compteur, 1, linked_nodes)

elif arc_list[compteur][1] in linked_nodes :
    ajoute_noeud(T, arc_list, compteur, 0, linked_nodes)

else :
    ajoute_noeud(T, arc_list, compteur, 0, linked_nodes)
    ajoute_noeud(T, arc_list, compteur, 1, linked_nodes)

ajoute_arc(T, arc_list[compteur][0], arc_list[compteur][1], arc_list[compteur][2])
```

On teste ensuite l'algorithme précédent sur des exemples issus de tsplib95 et on trace l'arbre, en affichant son coût.

Troisième étape :

Sans perte d'information nous pouvons transformer le graphe retourné par l'algorithme de kruskal qui n'est pas orienté en algorithme orienté.
Il suffit de remplacer chaque arête (i,j) du graphe par deux arêtes i→j et j→i de même poids.
Nous pouvons faire cette opération puisque par hypothèse le graphe d'origine est symétrique.

Pour implémenter cela il suffit de faire appel à la fonction "to_directed()"

Quatrième étape :

Nous arrivons au cœur du programme. Nous allons utiliser la stratégie du raccourci (ou “shortcut strategy”) :
Répéter jusqu'à ce que tous les noeuds soient visités :

- choisir un noeud non visité
- ajouter ce noeud à la liste des noeuds visités
- ajouter à une liste tous ses voisins qui n’ont pas été visité
- créer un arc entre le noeud actuel et celui qui précède (avec le poids qui correspond)

Voici le code qui correspond à cette étape :

```
while len(O) != 0 : # Si O est vide alors tous les noeuds ont été visités

    node = O[-1]

    ## Construction de la solution sous forme de liste de noeuds
    O.remove(node)
    V.append(node)

    for neigbour in dir_mst_graph.neighbors(node) :
        if neigbour not in V and neigbour not in O:
            O += [neigbour]

    ## Construction au fur et à mesure de la solution en graphe et calcul du coût
    if compteur >= 1:

        i = V[compteur-1]
        j = V[compteur]
        weight = arc_list[str(i) + str(j)]
        edge = [(i,j,weight)]
        solution.add_weighted_edges_from(edge)
        cost += weight

    compteur += 1
```

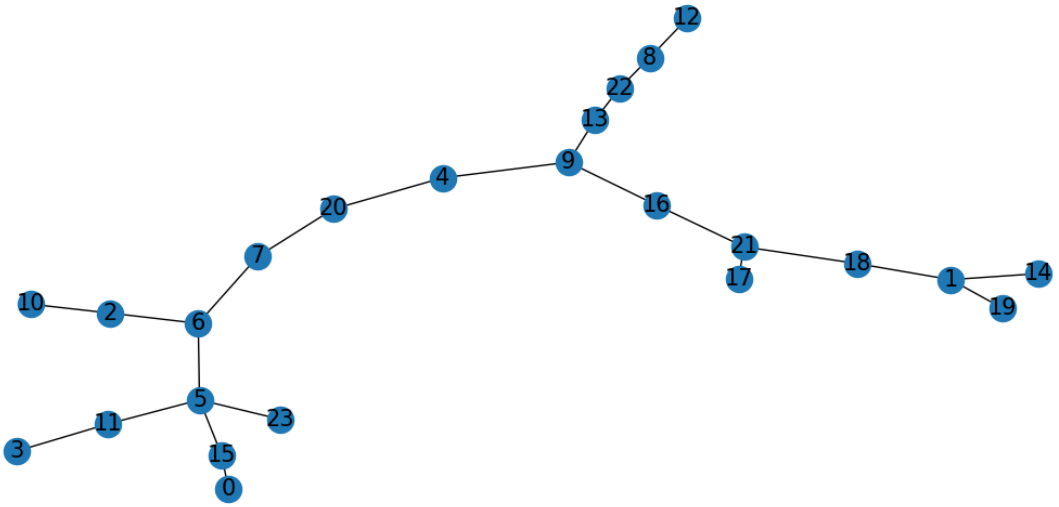
La fonction “add_weighted_edges_from” permet d’ajouter un arc avec un poids en une ligne.

Les poids du graphe d’origine ont été sauvegardé dans un dictionnaire pour pouvoir les retrouver rapidement de la manière suivante :

```
arc_list = dict()
for (i, j, weight) in graph.edges(data='weight'):
    arc_list[str(i) + str(j)] = weight
```

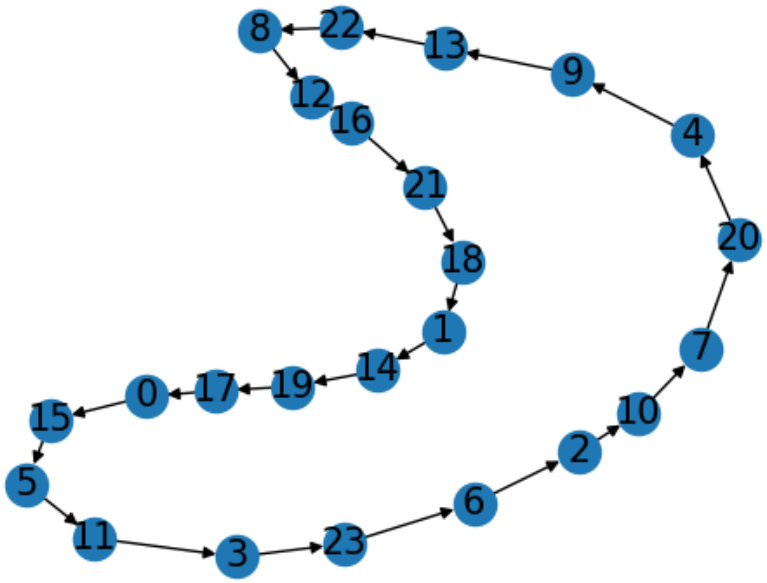
III - Résultats

Concernant tout d'abord l'algorithme de Kruskal, si on exécute le fichier Kruskal_algorithm.py avec "gr24.tsp", on obtient l'arbre suivant :



On remarque ici qu'il n'y a pas de cycle. Mais si on veut en être sûr, on peut appliquer la commande nx.find_cycle(T). Elle renverra une erreur si elle n'en trouve pas, ce qui empêchera de continuer l'exécution du programme mais permettra de vérifier que l'algorithme est correct.
On trouve un coût minimal de 1011.

Testons maintenant l'algorithme de TSP.
Avec le fichier "gr24.tsp", on obtient le graphe suivant :



Il parcourt tous les nœuds et forme un unique cycle. Le parcours des noeuds dans l'ordre (avec une origine quelconque) est [0, 15, 5, 11, 3, 23, 6, 2, 10, 7, 20, 4, 9, 13, 22, 8, 12, 16, 21, 18, 1, 14, 19, 17]
Le coût obtenu avec notre algorithme est de 2009.

Sachant que la solution optimale est de 1272, la solution obtenue est bien inférieure à deux fois la solution optimale. De plus, elle est supérieure au coût du problème de MST, mais inférieure à deux fois le coût du problème de MST.

La solution obtenue est donc satisfaisante en tout point de vue, et vérifie la 2-approximation.

La solution optimale de "eil51" est 426. Nous trouvons 704 avec notre algorithme, et 375 pour le problème de MST. Notre solution est donc correcte pour un algorithme qui fait de la 2-approximation (704 < 2*426). De plus, notre solution est comprise entre une et deux fois la solution du MST.

Essayons maintenant avec un graphe qui possède beaucoup plus de nœuds. Avec le fichier "bier127.tsp" qui possède 127 nœuds on obtient un coût de 183276 pour le TSP. Le coût du MST associé est 94706. On a donc bien coût(MST) < coût(TSP) < 2*coût(MST).

Effectuons un autre test sur un graphe de 130 nœuds. Si l'on considère le fichier "ch130.tsp", on sait que son optimum pour le TSP est de 6110 et celui pour le MST de 5166. Avec notre algorithme heuristique, nous trouvons 10328.
On remarque donc que la solution trouvée est bien inférieure à deux fois l'optimum, mais qu'elle est aussi supérieure à la solution du MST et inférieure à deux fois l'optimum du MST, qui vaut 10332.

En conclusion, nous pouvons dire que l'algorithme que nous avons écrit donne des résultats assez satisfaisants, même s'ils restent loin de l'optimum.