Année 2022-2023

RAPPORT DE PROJET

PROJET D'ALGORITHMIQUE DES GRAPHES

«Problème du voyageur de commerce»

Projet réalisé par :

Mounir AHMED DJIBABA

Badr ATIF

Walid BENCHABEKH

Imene BOUDJAOUI

Projet encadré par

Carla SELMI

SOMMAIRE

I. INTRODUCTION

Le problème du voyageur de commerce (TSP pour Traveling Salesman Problem) a été formulé pour la première fois au début des années 1800 par le mathématicien irlandais W.R. Hamilton. Il a été utilisé pour décrire la situation d'un commerçant qui souhaite visiter un certain nombre de villes pour vendre des produits et qui souhaite trouver le chemin le plus court pour cela.

Aucun algorithme déterministe polynomial n'a pas encore été découvert pour ce problème et il y a très peu de chances qu'il en existe un. Ce type de problème est dit NP-complet.

Une stratégie d'approche d'un problème NP-complet est de construire des algorithmes, dit d'approximation, qui fournissent de solutions presque optimales en temps polynomial.

Le travail proposé par ce projet est d'implémenter en Python quatre de ces algorithmes d'approximation et de faire une étude statistique des résultats obtenus.

II.BESOINS ET OBJECTIFS DU PROJET

1-CONTEXTE :

Le problème du voyageur de commerce (TSP) est un problème d'optimisation combinatoire qui consiste à trouver le chemin le plus court passant par une série de villes données une seule fois et retournant à la ville de départ. Il s'agit d'un des problèmes d'optimisation les plus connus et les plus étudiés dans la communauté scientifique.

.

2-MOTIVATION :

Le problème du voyageur de commerce (TSP) a été initialement formulé dans un contexte commercial, où il s'agissait de trouver le chemin le plus court pour un commerçant qui souhaitait visiter un certain nombre de villes pour vendre ses produits. Cependant, au fil des ans, il a été utilisé pour représenter de nombreux autres scénarios où il est nécessaire de planifier des itinéraires efficaces pour des activités de transport ou de livraison.

3-LES ENJEUX :

Il y a plusieurs enjeux liés à ce problème, notamment :

- Complexité : Le TSP est un problème NP-dur, ce qui signifie qu'il est très difficile de trouver une solution optimale pour des instances de grande taille en un temps raisonnable.

- Applications : Le TSP a de nombreuses applications pratiques, notamment la planification de tournées de livraison, la planification des itinéraires pour les véhicules de transport public, et la planification des tournées pour les ingénieurs de maintenance.

- Heuristiques : Les algorithmes exacts pour résoudre le TSP sont très coûteux en termes de temps et de ressources informatiques. Les heuristiques sont utilisées pour trouver des solutions approchées en un temps raisonnable.

- Optimisation : Le TSP est un problème d'optimisation combinatoire, c'est à dire qu'il s'agit de trouver la meilleure combinaison parmi un grand nombre de possibilités. Il est donc important de pouvoir évaluer efficacement les différentes combinaisons pour trouver la meilleure solution.

4-OBJECTIFS ET CONTRAINTES :

Les objectifs du TSP (Traveling Salesman Problem) sont de trouver un itinéraire qui :

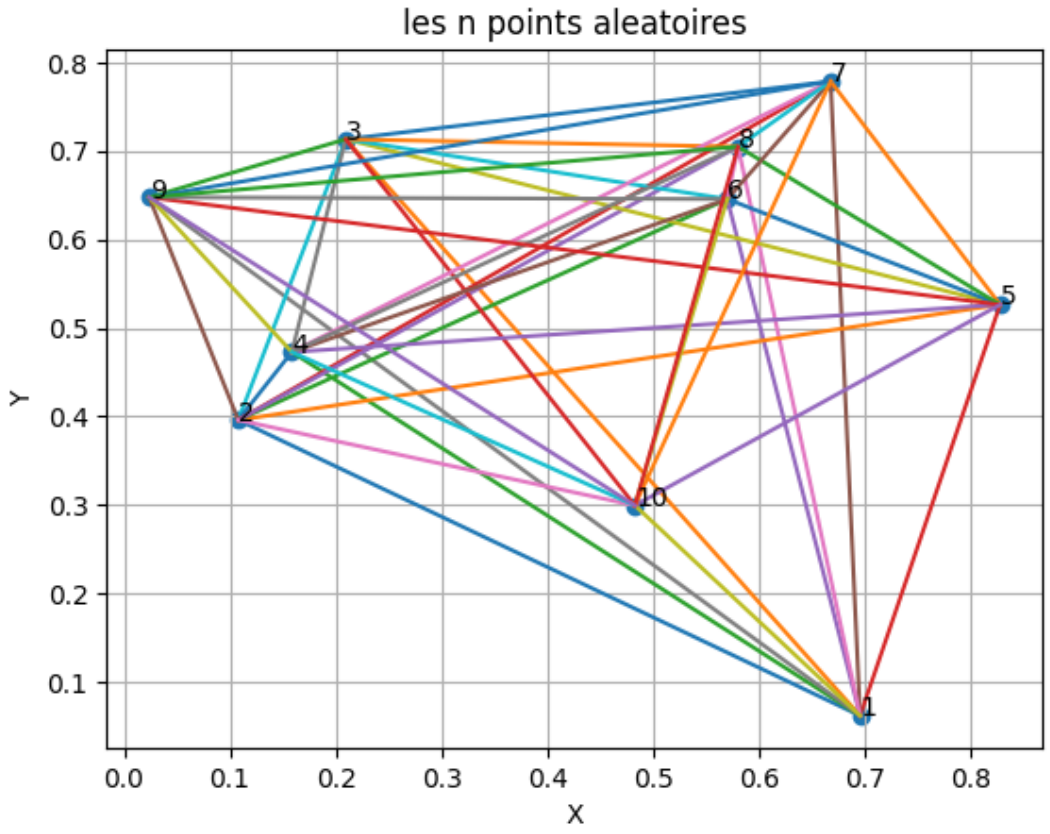
- passe par toutes les villes données exactement une fois,
- retourne à la ville de départ,
- minimise la distance totale parcourue.
-

Les contraintes du TSP sont :

- visiter toutes les villes données exactement une fois,
- retourner à la ville de départ,
- respecter les contraintes de temps, de capacité, de coûts, de sécurité, de respect de l'environnement, etc. qui sont spécifiques à chaque application.

III.ALGORITHMES D'APPROXIMATION

Pour un grahe initial de 10 sommets nous allons considérer 10 points 1, . . . , 10 du plan (qui représentent les sommets du graphe) dont les coordonnées sont tirées aléatoirement dans l'intervalle [0, 1] (bibliothèque numpy.random as rd). Les variables du problème sont notées : n pour le nombre des points, x, y pour les coordonnées des points et D pour la matrice des poids. La matrice D, de dimension n × n, est définie par D[i, j] contient la distance euclidienne entre pi et pj .



Sur ce dernier on appliquera :

1. L’ALGORITHME DU PLUS PROCHE VOISIN

a.Definition

L'algorithme du plus proche voisin (k-NN pour k plus proches voisins) est un algorithme d'apprentissage automatique qui permet de classer des données en fonction de leur similarité avec des exemples d'apprentissage. Il fonctionne en utilisant la métrique de distance pour déterminer la similarité entre les données d'entrée et les exemples d'apprentissage. L'algorithme détermine le k plus proches voisins des données d'entrée parmi les exemples d'apprentissage, et utilise la majorité des classes de ces k voisins pour classer les données d'entrée.

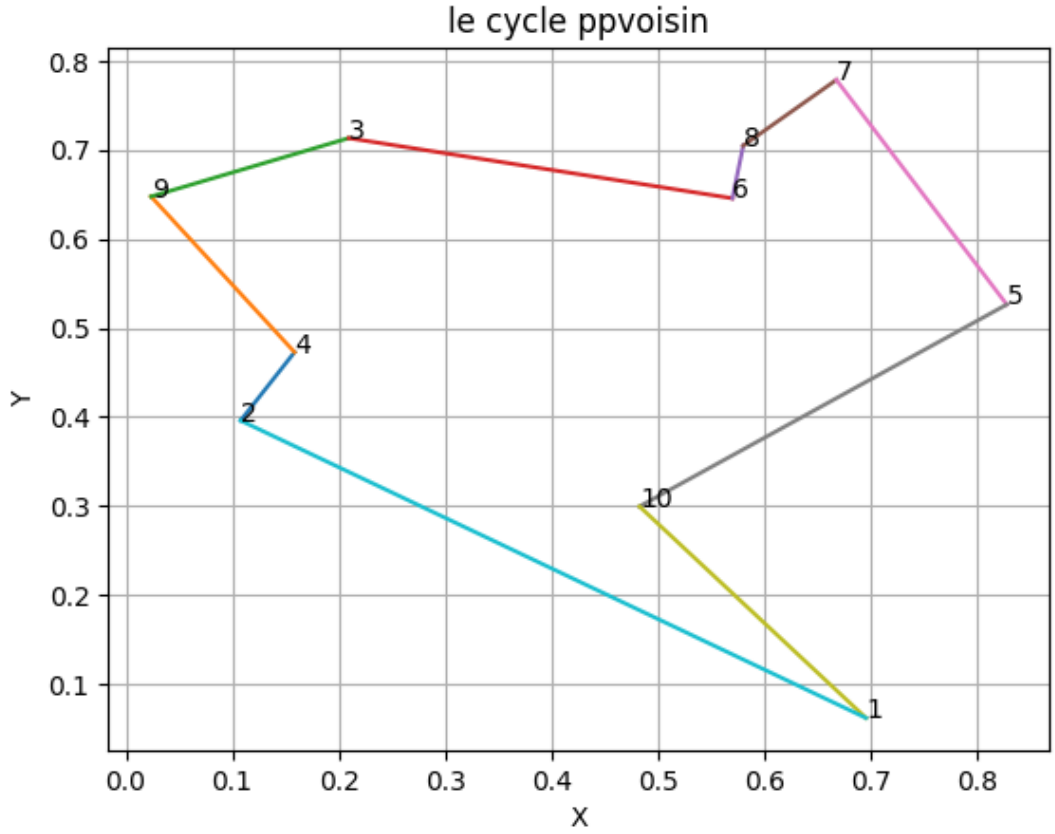
b.Fonctionnement

La fonction "Ppvoisin(s,d)" prend en entrée un sommet de départ "s" et une matrice de distance "d". Elle utilise l'algorithme de parcours de plus proche voisin pour trouver un cycle hamiltonien dans le graphe représenté par la matrice de distance. Elle initialise un ensemble de sommets non visités, un chemin vide et un sommet courant égal au sommet de départ. Elle utilise ensuite une boucle while pour visiter tous les sommets non visités en ordre de distance croissante par rapport au sommet courant. Elle ajoute chaque sommet visité au chemin et met à jour le sommet courant. La fonction retourne finalement le chemin trouvé qui contient tous les sommets visités, y compris le sommet de départ pour fermer le cycle.

c.Compléxité

La complexité de Ppvoisin dépend de la complexité de la fonction min(). Dans le pire des cas, la fonction min() parcourt tous les éléments de l'ensemble unvisited pour trouver le sommet non visité le plus proche, ce qui donne une complexité O(n^2) pour la boucle while. Dans l'ensemble, la fonction a une complexité O(n^2), car elle effectue n-1 itérations de la boucle while et chaque itération a une complexité O(n) pour trouver le sommet non visité le plus proche.

d.Affichage graphique



2. DEFAIRE LES CROISEMENTS

a. Definition

Cette partie du projet concerne le décroisement du cycle fourni par la fonction PPvoisin, une histoire d'optimisation au cas ou le poids du cycle est inferieur que le cycle fourni par la fonction précédente.

Pour vérifier si le décroisement est avantageux on compare le poids du sommet choisi avec tous les autres sommets pour enfin modifier éventuellement le
cycle et renvoyer le nouveau cycle en cas de modification.

b. Fonctionnement

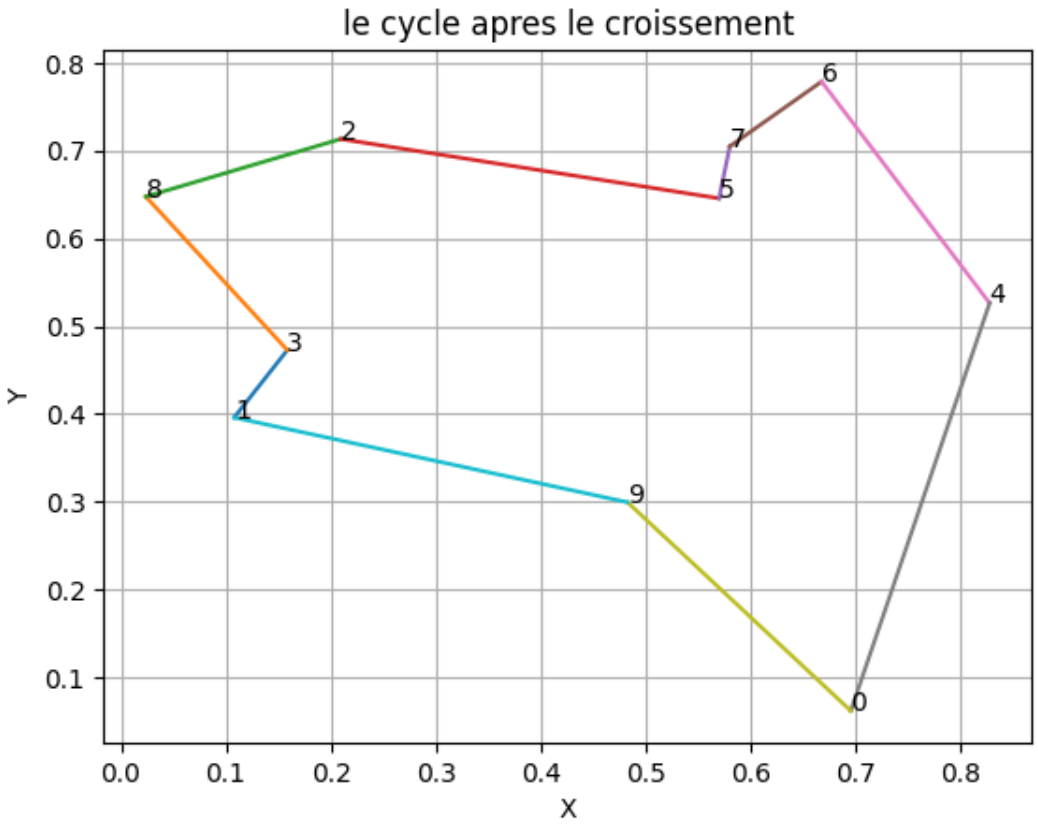
Dans la deuxième fonction OptimisePpvoisin, lorsque les arêtes (u, v) et (w, x) ont été trouvées dans le dictionnaire listeDesArcs, l'algorithme vérifie si le
décroisement entre ces deux arêtes est avantageux. Il calcule le poids des arêtes obtenues par le décroisement en utilisant la formule :
poids_arcs = listeDesArcs[(u,w)]['poids'] + listeDesArcs[(v,x)]['poids'] - (listeDesArcs[(u,v)]['poids'] + listeDesArcs[(w,x)]['poids'])
Cela signifie que le poids des arêtes obtenues par le décroisement est égal à la somme des poids des arêtes (u, w) et (v, x) moins la somme des poids des
arêtes (u, v) et (w, x) actuelles.
Si le poids des arêtes obtenues par le décroisement est inférieur à 0, cela signifie qu'il est avantageux de procéder au décroisement des arêtes (u, v) et (w, x) pour obtenir un graphe de poids inférieur. Il effectue alors le
décroisement en échangeant les éléments (u, v) et (w, x) dans la liste d'entrée. C'est ce qui permet de réduire le nombre de croisements entre les arêtes dans le graphe.

Il est important de noter que cet algorithme ne garantit pas de trouver la solution optimale, mais plutôt une solution améliorée par rapport à l'entrée initiale.

c.Compléxité

La complexité temporelle de ce programme est O(n^2) en raison des boucles for imbriquées dans la fonction OptimisePpvoisin. La boucle extérieure s'exécute n fois et pour chaque itération, la boucle intérieure s'exécute également n fois.
Cela donne un total de n*n itérations. La complexité temporelle de la première fonction listDesArcs est également O(n^2) pour les mêmes raisons (deux boucles imbriquées).
La complexité en espace de ce programme est O(n^2) car la taille du dictionnaire liste retourné par la première fonction est proportionnelle au carré du nombre de points dans les listes d'entrée.

d.Affichage graphique



3. L’ARETE DE POIDS MINIMUM

a.Definition

L'arête de poids minimum est l'arête dans un graphe ayant le poids le plus faible parmi toutes les arêtes du graphe. Elle relie deux sommets dans le graphe et a un poids associé qui peut représenter n'importe quoi, comme la distance entre les sommets ou le temps nécessaire pour traverser cette arête. L'arête de poids minimum est souvent utilisée dans les algorithmes de recherche pour trouver le chemin le plus court entre deux sommets.

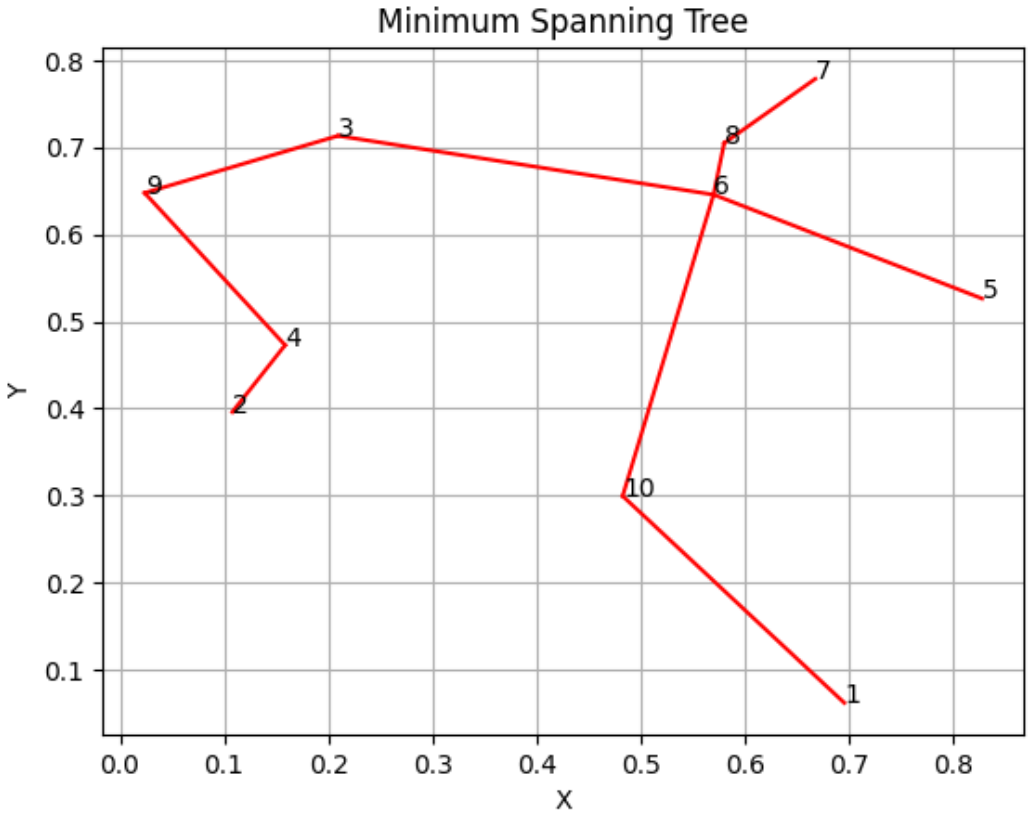
b.Fonctionnement

La fonction "Apminimum1" prend en entrée une liste d'arêtes et un entier "n" qui représente le nombre de sommets dans le graphe. Elle utilise l'algorithme de Kruskal pour trouver un arbre couvrant minimal dans le graphe. Elle trie d'abord les arêtes par ordre croissant de poids, puis utilise un objet de la classe EnsembleDisjoint pour maintenir l'ensemble des sommets qui sont connectés par les arêtes de l'arbre couvrant minimal en cours de construction. Elle utilise ensuite une boucle pour parcourir les arêtes triées en ordre croissant de poids, et pour chaque arête, elle vérifie si les sommets source et destination sont dans des ensembles disjoints en utilisant l'objet EnsembleDisjoint. Si c'est le cas, elle ajoute l'arête à l'arbre couvrant minimal en cours de construction et fusionne les ensembles des sommets source et destination. La fonction retourne finalement l'arbre couvrant minimal sous forme d'une liste de triplets.

c.Compléxité

La complexité pour trouver l'arête de poids minimum dans une matrice d'adjacence est de O(n^2), où n est le nombre de sommets dans le graphe. Cela est dû au fait qu'il faut parcourir toutes les arêtes dans la matrice d'adjacence pour trouver la plus petite. La complexité pour trouver le deuxième bord minimum est également de O(n^2) pour les mêmes raisons.

d.Affichage graphique



4. L'ARBRE COUVRANT DE POIDS MINIMUM

a.Definition

Un arbre couvrant de poids minimal (ACM) d'un graphe est un arbre couvrant (sous-ensemble qui est un arbre et qui connecte tous les sommets ensemble) dont la somme des poids des arêtes est minimale (c'est-à-dire de poids inférieur ou égal à celui de tous les autres arbres couvrants du graphe).

b.Fonctionnement

Dans cette partie on utilise un algorithme de calcule d'un arbre couvrant de poids minimal, qui est l'algorithme de Prim dans sa version la plus efficace, celle qui utilise un tas pour la gestion des sommets qui ne sont pas encore dans l'ACM courant afin de résoudre le problème du voyageur de commerce.

On commence par crée la classe **Tas** qui contient les attribues et les méthodes nécessaires pour le fonctionnement du tas. Parmi les méthodes les plus importantes dans la classe Tas on peut citer :

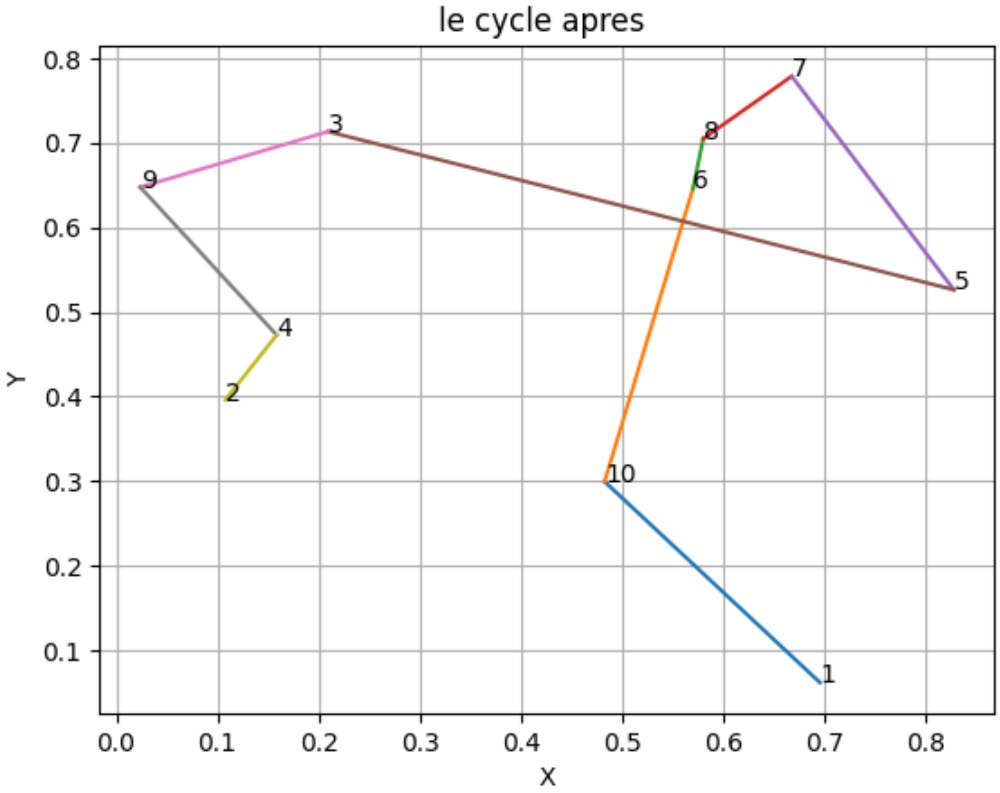
- **Inserer** : permet d'insérer dans le tas (l'insertion se fait toujours en dernière position).
- **VersLeHaut** : cette méthode est appelée dans le corps de la méthode Inserer, en effet, elle permet de réordonner le sommet qu'on vient d'insérer dans le tas. Parce que le tas est ordonné pour avoir en racine la clé minimale.
- **SupprimerMin** : permet de supprimer depuis le tas (la suppression se fait toujours en première position).
- **VersLeBas** : cette méthode est appelée dans le corps de la méthode SupprimerMin, en effet, elle permet de réordonner le tas après avoir supprimer sa racine. Parce que le tas est ordonné pour avoir en racine la clé minimale.

Après on crée la fonction **prim** qui prend en argument la liste d'arêtes de notre graphe et renvoie l'arbre couvrant de poids minimum. Et on donne ce résultat à la fonction **parcoursPref** qui fait le parcours préfixe de l'arbre obtenu, et qui donne une solution presque optimale au PVC.

c.Compléxité

La complexité de cet algorithme est O(m log n) avec m=|A| nombre d'arêtes et n= |S| nombre des sommets.

d.Affichage graphique



5. L’HEURISTIQUE DE LA DEMI SOMME

a.Definition

L'heuristique de la demi-somme est une méthode utilisée pour trouver une solution approximative à un problème de recherche de chemin. Il s'agit d'une heuristique qui est utilisée pour évaluer la distance entre deux nœuds dans un graphe de recherche. Elle est souvent utilisée pour la recherche de chemin à l'aide de l'algorithme A*.

L'heuristique de la demi-somme consiste à estimer la distance entre le nœud actuel et le nœud cible en prenant la somme des distances les plus courtes entre le nœud actuel et chaque nœud intermédiaire, puis en divisant cette somme par deux. Cette heuristique est basée sur l'hypothèse que le chemin le plus court entre deux nœuds passe généralement par un nœud intermédiaire. La demi-somme est souvent utilisée en combinaison avec d'autres heuristiques, comme la distance Euclidienne ou la distance Manhattan, pour améliorer la précision de l'estimation de la distance entre les nœuds. Il est important de noter que l'heuristique de la demi-somme ne garantit pas de trouver la solution optimale, mais elle permet d'éviter de parcourir des chemins inutiles et d'améliorer les performances de la recherche de chemin en réduisant le nombre de nœuds visités.

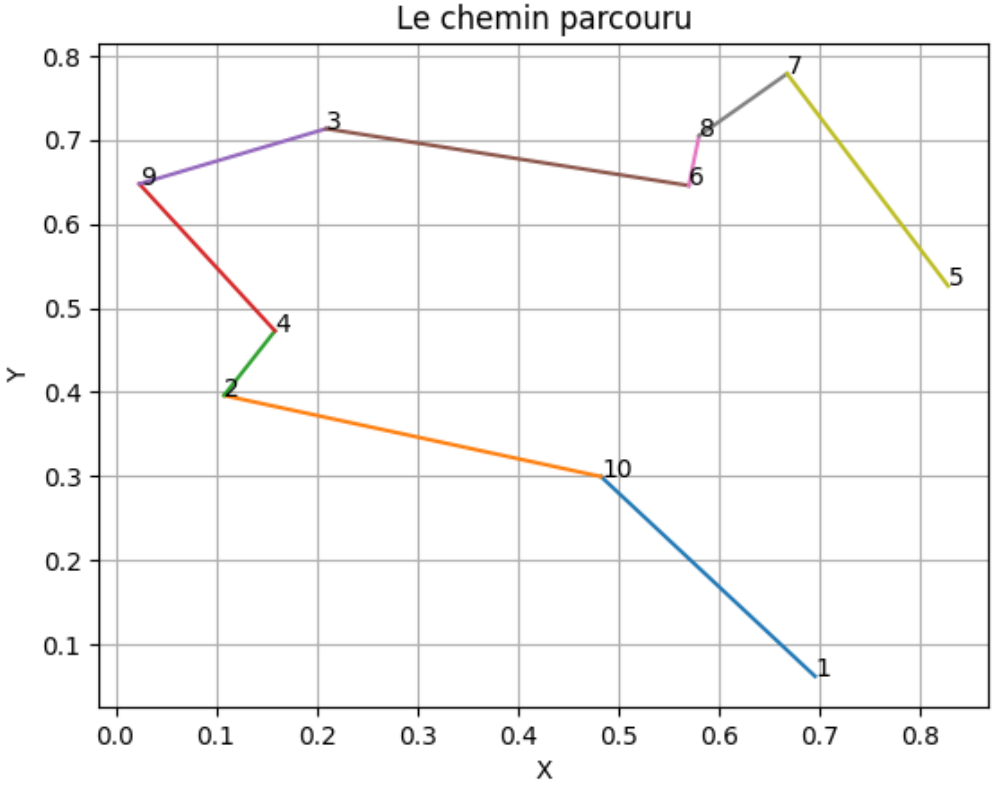
b.Fonctionnement

Ce code définit une fonction "TSPRec" qui résout le problème du voyageur de commerce (TSP) en utilisant un algorithme de branchement et de débranchement. La fonction prend plusieurs arguments, dont la matrice d'adjacence "adj" qui représente le graphe, la limite inférieure actuelle "curr_bound" de la solution, le poids actuel "curr_weight" de la solution, le niveau actuel "level" dans l'arbre de recherche, le chemin actuel "curr_path" qui est exploré, et un tableau booléen "visited" pour garder la trace des noeuds qui ont été visités dans le chemin actuel. La fonction utilise la variable globale "final_res" pour garder la trace de la meilleure solution trouvée jusqu'à présent et la variable globale "final_path" pour stocker la meilleure solution. Les fonctions d'aide "firstMin" et "secondMin" sont utilisées pour trouver les bords minimum et second minimum à partir d'un noeud donné, et la fonction "copyToFinal" est utilisée pour copier le chemin actuel vers la solution finale.

c.Compléxité

La complexité de cette heuristique est de O(n^2), où n est le nombre de sommets, car il faut parcourir tous les sommets pour trouver les coûts les plus faibles pour chaque sommet. Cependant, il convient de noter que cette heuristique ne garantit pas un résultat optimal, mais elle peut être utilisée pour éliminer rapidement des branches de l'arbre de recherche qui ne mènent pas à une solution optimale.

d.Affichage graphique



IV.MOYENNE DES CYCLES OBTENUS ET TEMPS D'EXECUTION

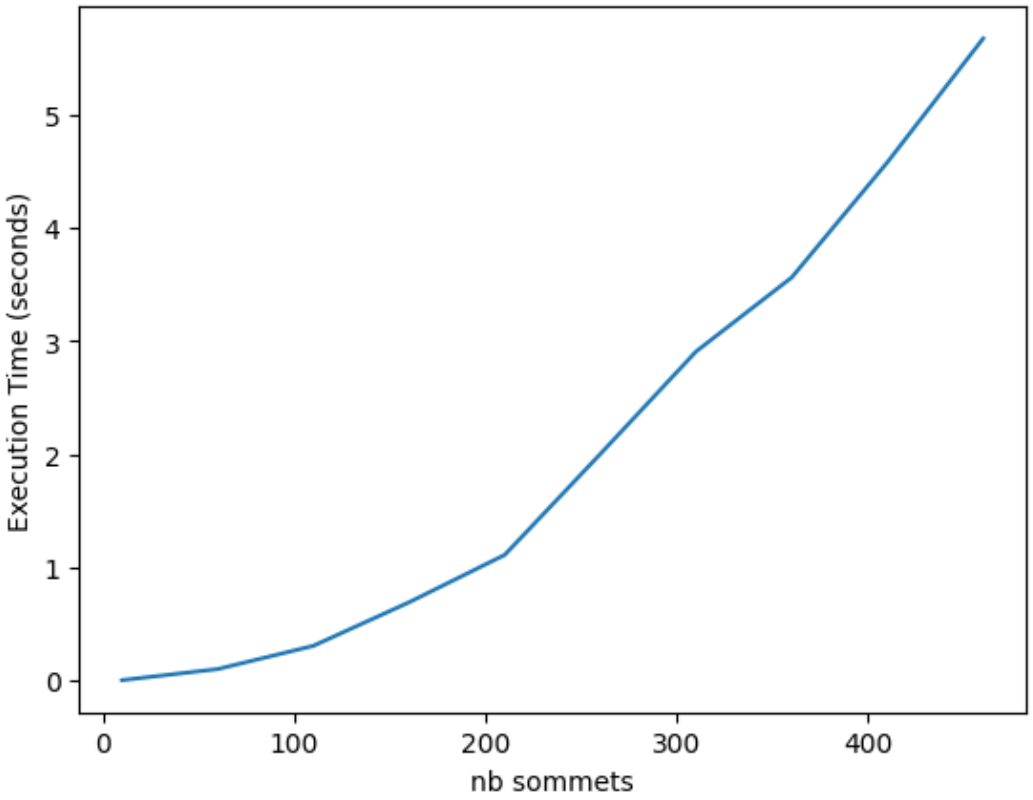
1. MOYENNE DES CYCLES OBTENUS

Pour notre exemple :

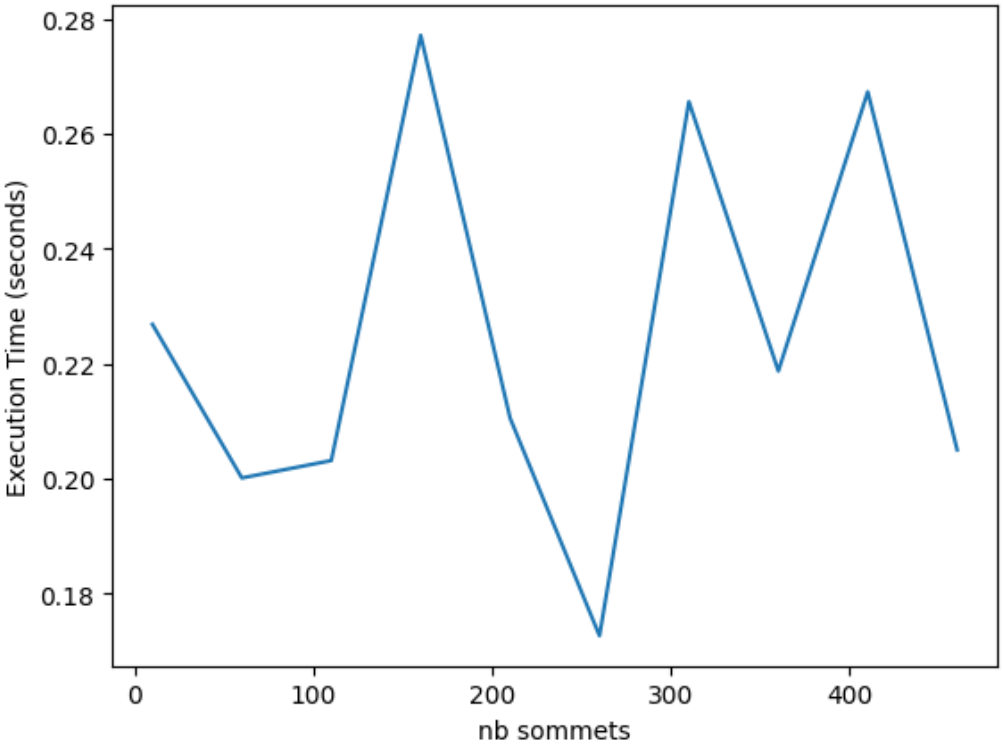
- sur 100 essais, la longueur moyenne des cycles obtenus par l'algorithme du plus proche voisin est 4.690944554538739
- sur 100 essais, la longueur moyenne des cycles obtenus par l'algorithme de défaire les croisements est 2.80764857072606
- sur 100 essais, la longueur moyenne des cycles obtenus par l'algorithme de l'arête de poids minimum est 2.097375339926495
- sur 100 essais, la longueur moyenne des cycles obtenus par l'algorithme de l'arbre couvrant de poids minimum est 2.101576613211608
- sur 100 essais, la longueur moyenne des cycles obtenus par l'algorithme de l'heuristique de la demi-somme est 2.0722869608368053

2. TEMPS D’EXECUTION

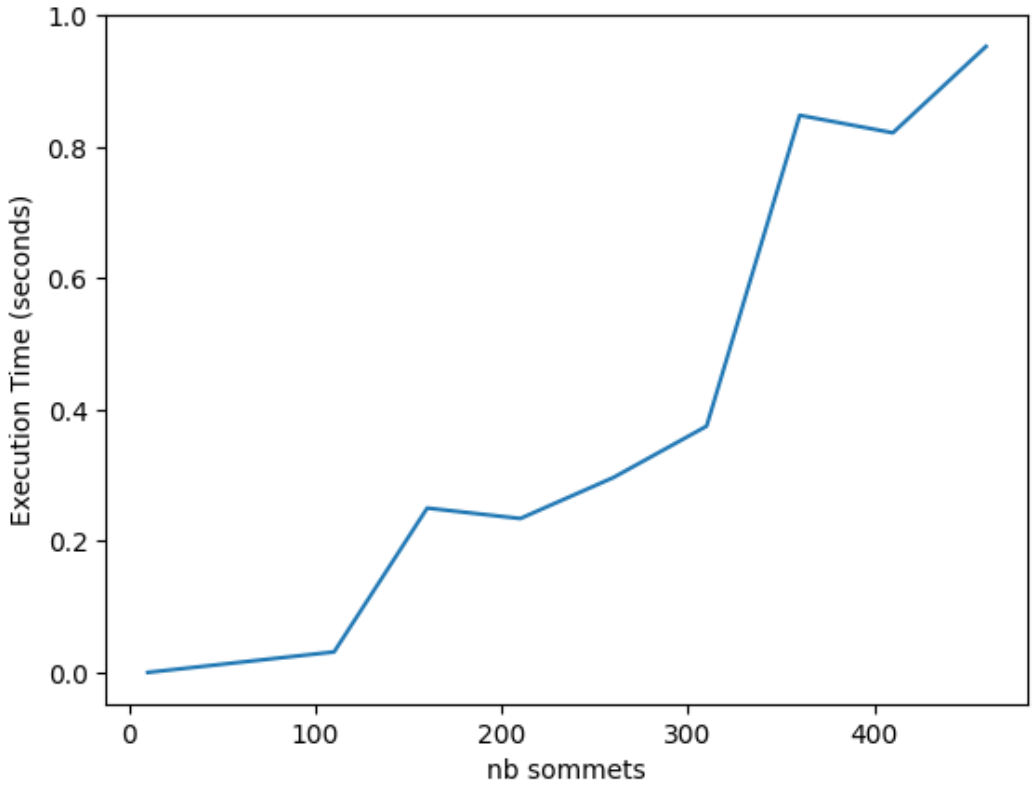
- Par l'algorithme du plus proche voisin



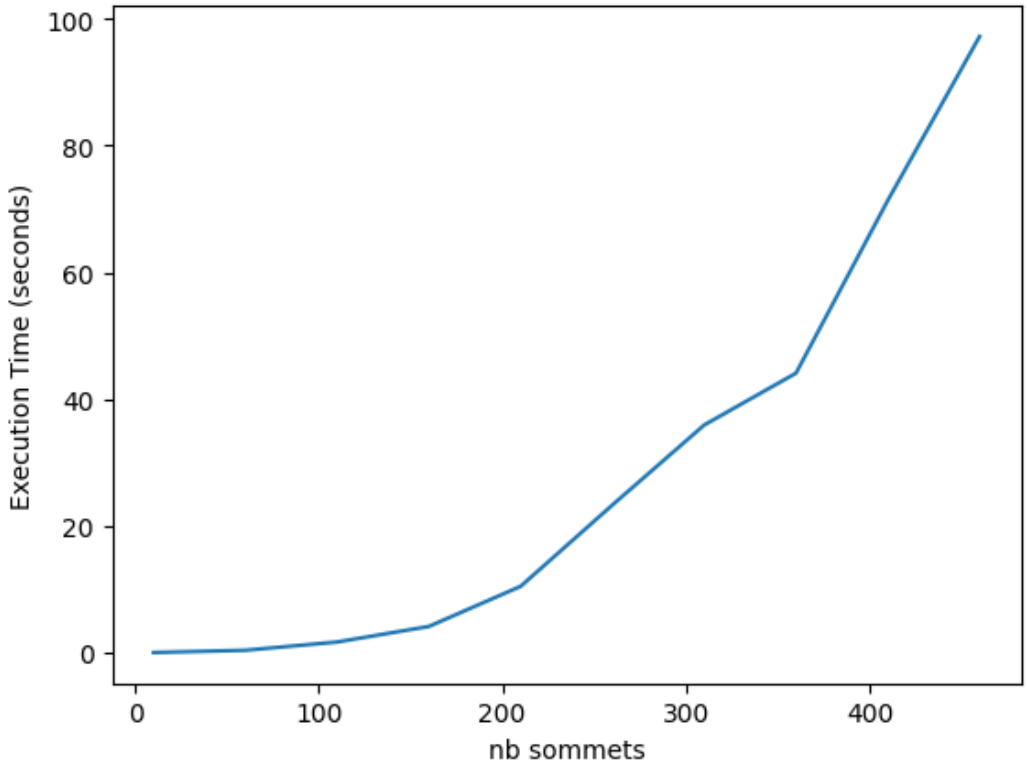
- Par l'algorithme de défaire les croisements



- L'algorithme de l’arête de poids minimum



- Par l'algorithme de l’arbre couvrant de poids minimum



- Par l'algorithme de l’heuristique de la demi-somme