TP3 - Voyageur de commerce

Janvier 2025

1 Le problème du voyageur de commerce

Le problème du voyageur de commerce (TSP pour *Traveling Salesman Problem*) est un problème d'optimisation combinatoire classique. Il consiste à trouver un chemin le plus court possible qui passe exactement une fois par chaque ville d'un ensemble donné et qui revient à la ville de départ. Ce problème est **NP-complet**.

Dans le cadre de ce TP, nous allons étudier différentes approches pour résoudre le problème du voyageur de commerce :

- Une solution naïve par force brute
- Une heuristique gloutonne
- Une approche métaheuristique (recuit simulé).

Vous utiliserez le fichier `worldcities.csv` qui contient les coordonnées géographiques de nombreuses villes du monde.

Vous pourrez également vous inspirer du fichier `info602-tp3-johndoe.py` qui contient des exemples de code pour lire le fichier CSV et calculer la distance entre deux villes.

1.1 Calcul de la distance entre deux villes

Pour résoudre le problème du voyageur de commerce, nous avons besoin de calculer la distance entre deux villes. Pour cela, vous pouvez utiliser la formule de Haversine pour les coordonnées géographiques (latitude et longitude).

2 Solution naïve par force brute

La première approche consiste à énumérer toutes les permutations possibles des villes et à calculer la longueur du chemin pour chacune d'entre elles. La solution optimale est celle qui minimise cette longueur.

Dans le fichier `worldcities.csv`, récupérez les coordonnées géographiques de 10 villes du France. Implémentez un programme en Python qui génère toutes les permutations possibles des villes et qui retourne le chemin optimal pour le problème du voyageur de commerce.

Vous incluerez Chambéry dans tous vos tests, qui, bien évidemment, sera la ville de départ et d'arrivée.

Augmentez progressivement le nombre de villes et mesurez le temps de calcul nécessaire pour obtenir la solution optimale.

1. A quelle taille votre algorithme devient-il impraticable ?
2. Quelle est la complexité de cet algorithme en fonction du nombre de villes ?

3 Solution heuristique gloutonne

Une **heuristique** est une méthode de résolution qui ne garantit pas de trouver la solution optimale, mais qui fournit une solution rapidement. L'algorithme du **plus proche voisin** est une heuristique classique pour le problème du voyageur de commerce.

Le principe est le suivant :

- pour chaque ville, choisir la ville la plus proche non encore visitée
- relier ces villes dans l'ordre choisi
- revenir à la ville de départ

Implémentez l'algorithme du plus proche voisin en Python et testez-le sur les mêmes instances que précédemment.

1. A quelle taille d'instance cette heuristique commence-t-elle à donner des résultats significativement différents de la solution optimale ?
2. A quelle taille votre algorithme devient-il impraticable ?
3. Quelle est la complexité de cet algorithme en fonction du nombre de villes ?

4 Solution métaheuristique : recuit simulé

Le problème de l'heuristique du plus proche voisin est qu'elle peut être piégée dans un minimum local et ne pas trouver la solution optimale. Pour éviter ce problème, nous allons implémenter une approche métaheuristique : le **recuit simulé**.

Une **métaheuristique** est une méthode générale pour résoudre des problèmes d'optimisation qui, elle non plus, ne garantit pas de trouver la solution optimale, mais, elle aussi fournit une solution rapidement. La différence est que les métaheuristiques une combinaison d'heuristique et de méthodes aléatoires, permettant notamment d'éviter les minima locaux.

L'algorithme du recuit simulé est inspiré du processus de refroidissement et de recuit des métaux. Il consiste à accepter des solutions moins bonnes que la solution courante avec une certaine probabilité, qui diminue au fur et à mesure du temps.

Le recuit simulé nécessite de définir plusieurs paramètres :

- une **température initiale** qui détermine la probabilité d'accepter des solutions moins bonnes au début
- un **taux de refroidissement** qui permet de diminuer la température au fur et à mesure
- un **critère d'arrêt** (nombre d'itérations, température minimale, etc.)

Le principe de l'algorithme est le suivant :

- Générer une solution initiale aléatoire (une permutation des villes)
- Calculer le coût de cette solution, ce sera la température initiale
- Tant que le critère d'arrêt n'est pas atteint :
  - Générer une solution voisine (en échangeant deux villes dans la tournée)
  - Calculer le coût de cette solution
  - Si la solution est meilleure, l'accepter
  - Sinon, l'accepter avec une probabilité qui dépend de la température : P(accepter) = exp((coût courant - coût voisin) / température)
  - Mettre à jour la température : température = température * (1 - taux de refroidissement)

Implémentez l'algorithme du recuit simulé en Python et testez-le sur les mêmes instances que précédemment.