Devoir OCO, M1 IWOCS

Date de rendu

Dimanche 15 juin 2025

Modalités

Devoir à rendre en binôme. Pas plus de deux étudiants par devoir. Inclure le rapport contenant la description, le modèle, les méthodes et les résultats ainsi que les fichiers source Python dans une archive (de préférence zip) à vos noms. Les résultats et les illustrations devront figurer dans le rapport. Vous utiliserez python pour le développement. L'utilisation de chatGPT ou tout autre outil d'aide est autorisé, à condition de le mentionner et de fournir les scripts utilisés.

Comme indiqué en TP, l'idée du projet est de reprendre le TSP pour étudier une variante, sur un ensemble de points obtenus à partir d'une base de données géographique (OSM de préférence). Un code fonctionnel minimal en Python pour le TSP est fourni sur Eurêka.

Sujet : étude d'une variante du TSP

Le problème du Voyageur de Commerce (Traveling Salesman Problem – TSP) est un des problèmes fondamentaux de l'Optimisation Combinatoire. Il consiste, dans un graphe $G = (S, A)$ où chaque arc $(i, j) \in A$ possède une longueur $c_{ij} \geq 0$, à rechercher un cycle hamiltonien (passant exactement une fois par chaque sommet) de longueur totale minimale. Le graphe n'est pas nécessairement symétrique. Il n'est pas forcément complet non plus, mais on peut le rendre complet en calculant le plus court chemin entre toute paire de sommets. Le TSP est dans la classe NP-complet

Formulation mathématique

Il existe plusieurs formulations en Programmation Linéaire en Nombres Entiers. La plus simple utilise une variable de décision $x_{ij} \in \{0,1\}$ pour chaque arc $(i, j) \in A$ qui indique si l'arc est utilisé ou pas.
Le modèle est alors

$$
\begin{cases}
min & \displaystyle\sum_{(i,j)\in A} c_{ij}x_{ij} & & (1) \\
s.t. & & & \\
& \displaystyle\sum_{a:(a,i)\in A} x_{ai} = 1 & \forall i \in S & (2) \\
& \displaystyle\sum_{b:(i,b)\in A} x_{ib} = 1 & \forall i \in S & (3) \\
& \displaystyle\sum_{(i,j)\in S'} x_{ij} \leq |S| - 1 & \forall S' \subset S, |S'| \geq 2 & (4) \\
& x_{ij} \in \{0,1\} & \forall (i,j) \in A & (5)
\end{cases}
$$

La fonction objectif (1) minimise la somme des longueurs des arcs sélectionnés. Les contraintes (2) imposent que chaque sommet possède exactement un arc sélectionné en entrée. Les contraintes (3) imposent que chaque sommet ait exactement un arc sélectionné en sortie. Les contraintes (4) interdisent la sélection de k arcs ou plus dans un sous-ensemble de k sommets. Elles interdisent donc

les sous-cycles puisque S' contient entre 2 et $n-1$ sommets. Enfin les contraintes (5) définissent les variables.

Heuristiques et métaheuristiques pour le TSP

Comme le TSP est NP-complet, calculer la solution optimale pour des instances de grande taille ne peut se faire en temps raisonnable. Dans ce cas, on préfère appliquer des méthodes rapides fournissant des solutions approchées : les heuristiques et les métaheuristiques. Voici une liste succincte de quelques-unes d'entre elles :

- Heuristiques constructives
  - Heuristique basique : on visite les sommets dans l'ordre de leur numéro
  - Heuristique aléatoire : on visite les sommets dans un ordre aléatoire
  - Heuristique du plus proche voisin : partant d'un sommet initial, on va sur le plus proche sommet voisin non encore visité. On recommence jusqu'à les avoir tous visités
- Recherches locales
  - 2Opt : à partir de la solution courante, on évalue toutes les solutions voisines qu'on peut obtenir en remplaçant 2 arcs par 2 autres. On prend la première solution voisine qui améliore et on recommence jusqu'à ne plus trouver de voisin améliorant
  - OrOpt : même chose mais en supprimant un sommet et en le réinsérant ailleurs
  - Swap : même chose mais en échangeant la position de 2 sommets quelconques
- Métaheuristiques
  - Multistart : on génère k solutions aléatoirement et on améliore chacune avec une recherche locale. On prend la meilleure solution obtenue à la fin des k descentes
  - Recuit simulé : on modifie une recherche locale. Au lieu de prendre le premier voisin améliorant, on génère un voisin au hasard. Il devient la nouvelle solution s'il est meilleur ou sinon selon une probabilité dépendant de la détérioration et d'une « température » qui décroit au fil des itérations.
  - Recherche tabou, GRASP, ILS, ELS, LNS, algorithme génétique…

Travail à réaliser

A partir du dernier TP sur le TSP, vous commencerez par choisir une extension de TSP et un type de point d'intérêt (Point Of Interest - POI) dans Le Havre à visiter dans le TSP. Voici quelques extensions du TSP :

- TSP avec prime : on n'est pas obligé de visiter tous les sommets, mais chaque sommet a une prime et on cherche à maximiser la somme des primes collectées moins la somme des coûts des arcs utilisés.
- TSP avec capacité : chaque client i a une demande D_i en colis et le véhicule a une capacité Q telle qu'il ne peut traiter tous les clients. Certains clients ne seront donc pas visités. On veut maximiser la somme des demandes livrées moins la somme des distances parcourues, tout en respectant la capacité du véhicule.
- TSP avec livraison et collecte : on doit livrer certains clients et en collecter d'autres. Chaque client est soit en livraison soit en collecte. Il faut passer sur les clients en livraison avant les clients en collecte.
- TSP avec transfert de produit : on considère un unique type de produit. Certains clients consomment une quantité fixe et connue de ce produit alors que d'autres fabriquent une quantité fixe et connue de ce produit. Les quantités récupérées chez un client peuvent donc servir pour les suivants dans la route. On ajoute une pénalité dans la fonction objectif. C'est la quantité chargée initialement dans le véhicule pour traiter les premiers clients (si nécessaire).

- TSP avec précédence : on a un ensemble de paires de précédence $\{i, j\} \in S \times S$ qui indiquent que le sommet i doit être visité avant le client j dans la route. Evidemment, les précédences ne doivent pas contenir de cycle. On cherche un cycle hamiltonien qui respecte ces précédences.
- TSP rouge/noir : certains clients sont rouges alors que d'autres sont noirs. La séquence de visites doit alterner clients rouges et noirs.
- TSP avec hub : chaque client i possède une liste $V_i \subset S$ de sommets voisins pour lesquels il peut servir de hub. Ainsi, visiter i dispense de devoir visiter les sommets dans V_i, ce qui permet de réduire le nombre de sommets visités.
- TSP avec cluster : les sommets sont partitionnés en sous-ensembles disjoints (exemple un quartier). Quand on visite le sommet d'un cluster, on doit visiter tous les autres sommets de son cluster avant de quitter le cluster.
- TSP avec fenêtre de temps : le coût des arcs est une durée. Chaque client i a une fenêtre de temps $[A_i, B_i]$ durant laquelle il peut être servi. On peut arriver avant A_i, mais on doit attendre la date A_i pour pouvoir traiter le client. On ne peut pas arriver au client i après B_i. On cherche un cycle hamiltonien de durée minimale qui respecte les fenêtres de temps.
- TSP asymétrique : le coût des arcs (i, j) et (j, i) n'est pas le même, par exemple à cause des sens interdits.

Toutes ces variantes reposent sur le TSP. La plupart nécessitent des données additionnelles. Une manière simple de générer ces données est de résoudre le TSP sur les données de base (graphe et matrice des coûts) puis de générer les données additionnelles de manière à ce que la solution obtenue pour le TSP de base ne respecte pas les contraintes additionnelles.

En ce qui concerne le type de points d'intérêt (POI), vous commencez par générer **aléatoirement** quelques instances pour développer, débugger et tester vos méthodes. Une fois que votre code est fonctionnel, vous utilisez les packages python osmnx pour récupérer des POI de la ville du Havre et calculer la matrice des distances – qui fera office de matrice des coûts. L'intérêt de cette approche est qu'on obtient en même temps les coordonnées GPS des POIs, qui qui permet de visualiser la solution. Voici quelques exemples de POIs :

- Les écoles
- Les restaurants
- Les pharmacies
- Les banques
- Les cafés
- Les fleuristes

La quantité de POIs n'est pas importante, à partir du moment où il y en a **au moins une vingtaine**. N'oubliez pas de stocker les données de chaque instance dans un fichier séparé afin de pouvoir les lire sans avoir besoin de les générer à nouveau. La démarche est donc la suivante :

1. On récupère les données de la ville en utilisant le module osmnx.graph
2. On récupère les POI en utilisant le module osmnx.features
3. On projette les POIs sur le point du graphe le plus proche en utilisant le module osmnx.distance
4. On remplit la matrice des distances entre ces points en utilisant aussi osmnx.distance

Rendu du projet

Le rapport devra contenir vos choix (variante du TSP et type de POI). Les variantes sont à peu près de la même difficulté et **je serai plus clément si vous traitez une variante que les autres groupes n'ont**

pas traité. Vous indiquerez en détail l'impact de la variante sur le modèle mathématique. Vous présenterez les méthodes de résolution et donnerez les résultats sur les fichiers d'instance que vous avez utilisé. Ça inclut la comparaison des valeurs et des temps de calcul, pour chaque méthode sur une même instance. Vous pouvez aussi insérer l'image des solutions obtenues.

Vous inclurez aussi les codes python et les scripts utilisés pour chatGPT ou autre outil d'aide, si vous les avez utilisés.