Projet MAOA : PD-TSP

Lin-Jie WU et Paul CIBIER

Sorbonne Université 2025/2026

Table des matières

1 Présentation du projet

L'objectif principal de ce projet est de proposer et de comparer plusieurs méthodes pour donner une solution au problème du PD-TSP (Pick-and-Deliver Traveling Salesman Problem). Dans notre cas d'étude, on travaille avec une hypothèse particulière : tous les objets qu'on doit livrer aux clients sont déjà chargés dans le camion au moment où il quitte le dépôt.

Pour mener cette étude, on s'intéresse à deux variantes de la fonction de coût pour voir comment elles influencent nos décisions de trajet et de chargement :

— **La fonction de coût linéaire :** Le coût est proportionnel au poids. Si on transporte deux fois plus de poids, on paye deux fois plus cher.

— **La fonction de coût quadratique :** Ici, le coût augmente beaucoup plus vite avec le poids (au carré). Cela permet de simuler une pénalité en cas de surcharge ou une consommation de carburant qui s'envole quand le camion est plein.

On a aussi testé deux situations différentes : une où le coût de transport dépend uniquement du poids (la distance n'est pas prise en compte), et une autre plus réaliste où la distance parcourue multiplie le coût du poids. Les formules qu'on utilise sont les suivantes :

— **Mode linéaire :** $dist \times C(w) \times \alpha$

— **Mode quadratique :** $dist \times (C(w) \times \alpha + C(w)^2 \times \beta)$

Ici, α et β sont des réglages qu'on peut ajuster, et $C(w)$ représente le poids actuel dans le camion.

2 Organisation des données et des instances

Pour le code, on a choisi d'utiliser **Python** avec le solveur **Gurobi** pour la partie optimisation mathématique.

Toutes les informations d'une ville et de ses objets sont regroupées dans une classe `PDTSP_Instance` (dans le fichier `model.py`). On y trouve notamment :

— **Un dictionnaire villes :** Pour chaque ville, on stocke ses coordonnées, si c'est un dépôt ou non, et surtout deux listes : `pickups` pour les objets à ramasser et `deliveries` pour les objets à livrer.

— **Les objets :** Ce sont des dictionnaires avec un identifiant, un poids, et un profit. À noter que les objets à livrer ont un profit de zéro puisqu'ils sont obligatoires.

— **La capacité et le poids initial :** `capacity` est la limite de charge du camion et `w0` est le poids au départ.

— **La matrice des distances :** On calcule une matrice carrée qui contient les distances euclidiennes entre toutes les villes pour éviter de les recalculer à chaque fois.

Concernant les données de test, on utilise les instances de *H. Hernández-Pérez*. Comme les données de base sont un peu simples, on utilise une fonction `prepare_split_data` pour découper les demandes de ramassage en plusieurs petits objets et leur donner un profit aléatoire basé sur leur poids.

3 Algorithme Glouton (Greedy)

La première méthode qu'on a implémentée se trouve dans `greedy.py`. C'est une méthode qui ne regarde pas du tout la distance géographique. L'idée est de combiner deux idées simples : on vide le camion en priorité pour être léger, puis on va chercher les plus légers en priorité

Heuristique Greedy Delivery (Logique simplifiée)

Data : Instance P, paramètres α, β, mode (linéaire/quadratique)
Result : Tournée T et décisions de chargement D

while *il reste des clients à visiter* **do**
  1. Sélection de la ville :
  Parcourir l'ensemble des clients restants :
  - **Priorité 1 :** Choisir la ville présentant la livraison la plus lourde (stratégie d'allègement).
  - **Priorité 2 :** Sinon, choisir la ville qui minimise l'impact sur la charge (celle dont le poids de l'objet principal est le plus faible)

  2. Décision de chargement interne :
  Une fois la ville choisie, trier ses objets par rentabilité et pour chaque objet :
  if $Profit > Cot \times Clients\_restants$ **then**
    Charger l'objet (mettre à jour la charge du camion) ;
    Enregistrer la décision dans D ;
  end
  3. Mise à jour :
  Ajouter la ville à la tournée T ;
  Retirer la ville de la liste des clients à visiter ;
end

3.1 Avantages et limites de l'approche

Avantages et stratégie

- **Rapidité :** La méthode est très facile à implémenter et offre un temps de calcul quasi instantané, même sur des instances complexes.
- **Gestion du poids :** Livrer les objets lourds en priorité réduit immédiatement la charge du camion, ce qui fait baisser le coût cumulé sur tout le reste du trajet.
- **Rentabilité garantie :** Grâce à la règle *Profit > Coût transport*, chaque objet chargé contribue positivement à la fonction objectif sans jamais la dégrader.

Limites majeures

Le défaut principal est l'absence totale de prise en compte de la **distance géographique**. L'algorithme choisit sa cible uniquement par rapport au poids ou au profit.

- **Efficacité variable :** Si l'objectif ne dépend que du poids, les résultats sont excellents.
- **Incohérence spatiale :** Dès que la distance compte, l'algorithme perd en efficacité. Il peut générer des allers-retours entre des villes éloignées pour un simple objet lourd, ce qui fait exploser le coût total à cause des kilomètres parcourus.

4 Méthode du Plus Proche Voisin Pondéré (Nearest Neighbor)

Dans le fichier `nearest.py`, on propose une approche plus fine que le simple glouton. Bien que le Nearest Neighbor soit souvent limité à la distance géométrique, nous l'avons ici adapté en une Heuristique de Construction à Score, au lieu de se ruer sur la ville la plus proche géographiquement, l'algorithme calcule un **score global** pour chaque candidat. Ce score permet d'évaluer si le déplacement est "rentable" sur le long terme.

4.1 La logique du Score

Pour chaque ville candidate, on calcule un score selon la formule suivante :

$$Score = Profit\_total - Cot\_Transport + Gain\_Allgement$$

L'objectif est de choisir, à chaque étape, la ville qui offre le meilleur compromis. Voici le détail de chaque composante :

1. **Le Coût de Transport :** C'est ce que nous coûte le trajet pour aller de la ville actuelle à la ville candidate avec le poids que l'on a déjà dans le camion.
   — En mode linéaire, on fait : $dist \times poids\_actuel \times \alpha$.
   — En mode quadratique, le poids compte beaucoup plus : $dist \times (poids\_actuel \times \alpha + poids\_actuel^2 \times \beta)$.
2. **Le Profit total (Choix des objets) :** Avant de valider une ville, on regarde les objets qu'elle propose. On ne prend pas tout au hasard. Pour chaque objet, on estime son **coût futur** : c'est le poids de l'objet multiplié par la distance moyenne qu'il reste à parcourir jusqu'à la fin de la tournée. *Critère de décision :* On ne charge un objet que si son **Profit > Coût futur**. Si l'objet est trop lourd et qu'on est encore loin de l'arrivée, il va nous coûter plus cher en transport que ce qu'il nous rapporte. On ne le prend donc pas.
3. **Le Gain d'Allègement :** C'est l'un des points les plus importants. Si la ville candidate nous permet de livrer un objet, le camion va s'alléger. Cet allègement va réduire le coût de transport pour **tous** les trajets restants. On calcule ce gain en multipliant le poids livré par la distance moyenne restante. Plus on livre tôt un objet lourd, plus le bonus au score est grand.

4.2 Fonctionnement de l'algorithme

L'algorithme tourne en boucle tant qu'il reste des clients à visiter. À chaque itération, il simule le passage par chaque ville encore disponible pour calculer ces trois paramètres.

Algorithme 2 : Pseudo-code détaillé du Nearest Neighbor Pondéré

```
Data : Instance, paramètres α et β, mode (linéaire/quadratique)
while il reste des villes dans Customers do
    Best_Score ← −∞;
    for chaque ville c dans Customers do
        1. Calculer la distance dist entre la ville actuelle et c;
        2. Calculer le Cot_Transport pour faire ce trajet avec la charge actuelle;
        3. Estimer la distance moyenne restante D_moy entre c et les autres villes;
        4. Calcul du profit interne :
        for chaque objet k dans la ville c do
            Cout_Futur_K ← poids_k × D_moy × α;
            if Profit_k > Cout_Futur_K et capacité suffisante then
                Ajouter Profit_k au profit de la ville et marquer l'objet comme "à prendre";
            end
        end
        5. Calcul du gain d'allègement : Poids_Livr × D_moy × α;
        6. Score_Total ← Profit − Cot_Transport + Gain_Allgement;
    end
    Choisir la ville avec le Best_Score;
    Mettre à jour la position actuelle, le tour, les objets pris et le poids du camion;
end
```

Cette méthode est bien plus efficace que le glouton de base car elle évite de faire des détours inutiles pour des objets peu rentables, et elle cherche activement à livrer les objets lourds le plus vite possible pour "soulager" le score de transport.

5 Méthode itérative : Recherche Locale par Swap

Après avoir généré une solution avec les méthodes constructives (Greedy ou Nearest Neighbor ou même solution aléatoire), nous cherchons à l'améliorer via une méthode itérative. L'idée est de ne plus construire le tour, mais de le modifier localement pour augmenter la fonction objectif.

5.1 Le mécanisme du Voisinage (Swap)

Pour explorer de nouvelles solutions, nous utilisons une structure de voisinage classique du TSP : le **Swap**. Cela consiste à choisir deux villes dans le tour actuel et à échanger leurs positions.

Cependant, dans notre problème de PD-TSP, un simple échange de villes peut avoir des conséquences importantes :

- **La charge du camion :** Si on avance le ramassage d'un objet lourd dans le tour, le camion risque de dépasser sa capacité maximale (W_{max}) plus tôt.
- **La rentabilité :** Un objet qui était rentable à la position 10 ne l'est peut-être plus à la position 5 si le coût de transport cumulé devient trop élevé.

5.2 Réparation de la solution (Repair Decisions)

À chaque fois qu'un swap est effectué, la solution devient potentiellement "invalide" ou sous-optimale au niveau des objets. Nous avons donc implémenté une fonction de **réparation**. Pour un nouveau tour donné, cette fonction parcourt chaque ville et reprend la décision de charger ou non chaque objet en fonction de la nouvelle charge actuelle et de la distance restante. Cela garantit que chaque voisin testé est à la fois réalisable et optimisé au niveau de son gain.

5.3 Stratégie du First Improvement

Pour parcourir ce voisinage, nous avons choisi la stratégie du **First Improvement**. Dès qu'un swap est trouvé qui améliore la fonction objectif par rapport à la solution actuelle, on l'adopte immédiatement et on recommence la recherche à partir de ce nouveau point.

Avantages et inconvénients :

- **Efficacité temporelle :** Cette stratégie est beaucoup moins coûteuse que le *Best Improvement*, car elle évite de parcourir l'intégralité des voisins (qui sont très nombreux) quand une piste d'amélioration est déjà trouvée.
- **Risque d'optimum local :** Comme c'est une approche gloutonne, l'algorithme s'arrête dès qu'il ne trouve plus aucun voisin meilleur dans son environnement immédiat. On risque alors de rester bloqué sur un optimum local qui n'est pas le meilleur possible au niveau global.

6 Programmation Mathématique (PLNE)

7 Expérimentation et Résultat