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.

Structure d'une Instance :

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.

Définition d'une solution :

Une solution S est représentée par le couple (T, D) tel que :

— $T = [v_0, v_1, \ldots, v_n]$ est le vecteur représentant le *tour* (ordre de visite des villes).

— $D = [d_0, d_1, \ldots, d_n]$ est le vecteur des *décisions*, où chaque d_i est une liste binaire $d_i \in \{0,1\}^{k_i}$ (k_i étant le nombre d'objets dans la ville v_i).

Par exemple, si $d_i = [0, 1]$, cela signifie que pour la ville visitée à l'étape i, le premier objet n'est pas collecté tandis que le second l'est.

Données du problèmes :

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.

Visualisation :
Le fichier `parse.py` joue un rôle central dans la visualisation des solutions. Ses responsabilités incluent :
— **Lecture des instances :** Chargement et parsing des données du problème (villes, demandes, capacités).
— **Visualisation graphique :** Génération de graphiques illustrant les tournées optimisées pour une analyse visuelle immédiate.
— **Affichage console :** Sortie textuelle formatée des indicateurs de performance (profit, charge).

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

Stratégie de décision heuristique

Une fois toutes les villes qui ont des livraisons à livrer par ordre décroissant de poids. La sélection de la prochaine ville et des objets à charger repose sur deux mécanismes :

1. **Sélection MinMax de la ville :** Pour chaque ville i non visitée, on définit $w_{max,i}$ comme le poids de l'objet le plus lourd de cette ville. La ville choisie v^* est celle qui minimise ce poids :

$$v^* = \arg\min_i(w_{max,i})$$

2. **Condition de rentabilité :** Un objet de poids w et de profit p n'est chargé que si :

$$p > w \times N_{restant}$$

où $N_{restant}$ est le nombre de villes qu'il reste à visiter dans le tour. On essaye d'estimer le coût potentiel de charger cet objet dans le camion par rapport à son profit.

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* > *Estim 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**
  $Best\_Score \leftarrow -\infty$;
  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**
      $Cout\_Futur\_K \leftarrow poids_k \times D_{moy} \times \alpha$;
      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 :** $Poids\_Livr \times D_{moy} \times \alpha$;
    6. $Score\_Total \leftarrow Profit - Cot\_Transport + Gain\_Allgement$;
  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 : 2-opt

Après avoir généré une solution avec les méthodes constructives (Greedy, Nearest Neighbor ou solution aléatoire), nous cherchons à l'améliorer via une méthode itérative. L'idée n'est plus de construire le tour, mais de le modifier localement pour diminuer la distance totale parcourue (TSP).

Pour définir le voisinage, on utilise l'échange 2-opt : on sélectionne deux arêtes $(i-1, i)$ et $(j, j+1)$ qui forment un croisement sur le graphe. On les remplace par les arêtes $(i-1, j)$ et $(i, j+1)$ en inversant le sens du segment entre i et j. On adopte cette modification si la somme des distances des nouveaux arcs est inférieure à celle des anciens arcs : $dist(i-1, j) + dist(i, j+1) < dist(i-1, i) + dist(j, j+1)$.

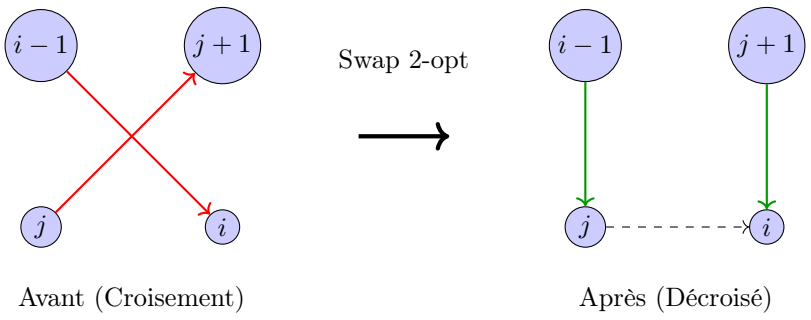


FIGURE 1 – Illustration du mouvement 2-opt : les arcs rouges sont remplacés par les arcs verts.

5.1 Stratégie du First Improvement

Pour parcourir ce voisinage, nous avons choisi la stratégie du **First Improvement**. Dès qu'un échange améliore la fonction objectif, on l'adopte immédiatement et on recommence la recherche à partir de ce nouveau tour. Afin de limiter le temps de calcul, la recherche est restreinte aux $k = 20$ plus proches voisins de chaque ville (selon la distance euclidienne). L'algorithme s'arrête lorsqu'aucun échange dans le voisinage ne permet plus d'amélioration.

5.2 Décision de chargement optimisée

Une fois le tour final fixé, nous appliquons une stratégie de remplissage optimisée. Contrairement aux méthodes constructives, le tour est ici connu à l'avance, ce qui nous permet de calculer la **distance exacte restante** à parcourir depuis chaque ville jusqu'au dépôt final.

Pour chaque ville, les objets sont triés par ratio profit/poids. Un objet est chargé si :

1. La capacité résiduelle du véhicule le permet.
2. Son profit est supérieur à son coût de transport réel : $Profit > Poids \times Distance\_Restante \times \alpha$.

Avantages et inconvénients :

— **Efficacité temporelle :** Moins coûteux que le *Best Improvement* car il n'explore pas tout le voisinage inutilement.

— **Optimum local :** Risque de rester bloqué sur une solution sous-optimale si aucun voisin immédiat n'est meilleur.

— **Décision séquentielle :** Le fait de séparer l'optimisation du trajet (TSP) et la décision de chargement (PD) limite l'exploration globale de l'espace des solutions.

Algorithme 3 : Hill Climbing 2-Opt

```
// Partie 1 : Optimisation Numba (Trajet)
T ← Tour initiale;
amélioration trouvée pour i ← 1 ă n − 2 faire
    pour chaque voisin j proche de i faire
        si dist(i − 1, j) + dist(i, j + 1) < dist(i − 1, i) + dist(j, j + 1) alors
            Inverser le segment T[i . . . j];
            amlioration ← Vrai;
            break;

// Partie 2 : Calcul des distances restantes
D_reste[n] ← 0;
pour i ← n − 1 ă 0 faire
    D_reste[i] ← D_reste[i + 1] + dist(T[i], T[i + 1]);

// Partie 3 : (Décisions)
pour chaque étape i du tour faire
    W ← W − PoidsLivraisons;
    Trier objets par ratio profit/poids;
    pour chaque objet faire
        CoutFutur ← D_reste[i] × f(poids, α, β);
        si W + poids ≤ Capacit et Profit > CoutFutur alors
            Charger l'objet;
            W ← W + poids;
```

6 Programmation Mathématique (PLNE)

L'objectif est de maximiser le profit net issu des ramassages, tout en respectant les contraintes de capacité et en minimisant les coûts de transport liés au poids. Nous ne prendrons pas en compte la distance, car celle-ci rendrait le PL beaucoup plus complexe et augmenterait considérablement les temps de calcul.

1. Ensembles et Paramètres

— $V = \{1, \ldots, n\}$: Ensemble des villes (où 1 est le dépôt de départ et n le dépôt d'arrivée).
— $T = \{0, \ldots, n-1\}$: Ensemble des étapes temporelles de la tournée.
— K_i^P : Ensemble des objets à **ramasser** dans la ville i.
— K_i^D : Ensemble des objets à **livrer** dans la ville i.
— $P_{i,k}$: Profit de l'objet de ramassage $k \in K_i^P$.
— $w_{i,k}^P$: Poids de l'objet de ramassage $k \in K_i^P$.
— $w_{i,k}^D$: Poids de l'objet de livraison $k \in K_i^D$.
— W_0 : Charge initiale du camion au départ du dépôt.
— Q : Capacité maximale du camion (`capacity`).
— α, β : Coefficients de coût de transport.

2. Variables de décision

— $x_{i,t} \in \{0, 1\}$: Vaut 1 si la ville i est visitée à l'étape t.
— $w_t \geq 0$: Charge du camion à l'étape t.
— $y_{t,i,k} \in \{0, 1\}$: Vaut 1 si l'objet de ramassage k de la ville i est collecté à l'étape t.
— $z_{t,i,k} \in \{0, 1\}$: Vaut 1 si l'objet de livraison k de la ville i est délivré à l'étape t.

3. Fonction Objectif

Maximiser le profit global Z :

$$\text{Max } Z = \sum_{t \in T} \sum_{i \in V} \sum_{k \in K_i^P} (P_{i,k} \cdot y_{t,i,k}) - \text{CoûtTransport} \tag{1}$$

Où le coût de transport est défini selon le mode :

— **Mode Linéaire :** $\sum_{t \in T} \alpha \cdot w_t$

— **Mode Quadratique :** $\sum_{t \in T} (\alpha \cdot w_t + \beta \cdot w_t^2)$
Remarque 1. Ici on a plus un programme linéaire à cause du w_t^2 , mais comme gurobi est toujours capable de résoudre ce MIQP on a quand même garder cette formulation pour les expérimentations (à garder en tête).

4. Contraintes

Tournée et Visites

Chaque étape est associée à une ville et chaque ville est visitée une seule fois :

$$\sum_{i \in V} x_{i,t} = 1 \qquad \forall t \in T \tag{2}$$

$$\sum_{t \in T} x_{i,t} = 1 \qquad \forall i \in V \tag{3}$$

Fixation du départ et de l'arrivée (Dépôts) :

$$x_{1,0} = 1 \tag{4}$$

$$x_{n,n-1} = 1 \tag{5}$$

Logique de collecte et livraison

On ne peut agir dans une ville que si le camion y est présent :

$$y_{t,i,k} \leq x_{i,t} \qquad \forall t \in T, \forall i \in V, \forall k \in K_i^P \tag{6}$$

$$z_{t,i,k} \leq x_{i,t} \qquad \forall t \in T, \forall i \in V, \forall k \in K_i^D \tag{7}$$

Gestion de la charge et Capacité

Initialisation et conservation du flux de poids :

$$w_0 = W_0 \tag{8}$$

$$w_t = w_{t-1} - \sum_{i \in V} \sum_{k \in K_i^D} (w_{i,k}^D \cdot z_{t,i,k}) + \sum_{i \in V} \sum_{k \in K_i^P} (w_{i,k}^P \cdot y_{t,i,k}) \qquad \forall t \in \{1, \ldots, n-1\} \tag{9}$$

$$w_t \leq Q \qquad \forall t \in T \tag{10}$$

Obligation de service

Toutes les livraisons prévues doivent être effectuées :

$$\sum_{t \in T} z_{t,i,k} = 1 \quad \forall i \in V, \forall k \in K_i^D \tag{11}$$

7 Expérimentation et Résultat

Dans cette section, nous étudions l'influence des différents hyperparamètres sur la qualité des solutions obtenues, ainsi que la scalabilité de nos algorithmes. L'analyse porte spécifiquement sur :

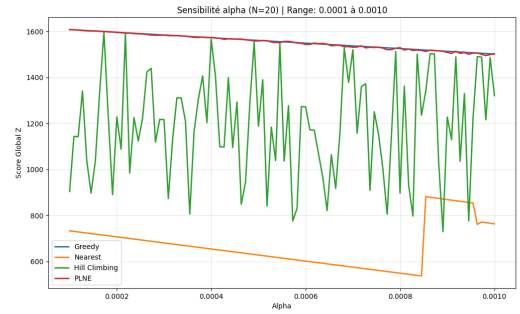
- **La sensibilité aux paramètres α et β :** nous observons comment la pondération du coût de transport (linéaire et quadratique) affecte le profit net.
- **Le temps de calcul :** nous mesurons l'efficacité temporelle de chaque méthode face à l'augmentation de la complexité.
- **La taille des instances :** nous évaluons la capacité des algorithmes à maintenir des performances robustes lorsque le nombre de villes (N) augmente.

7.1 Sensibilité au paramètre α

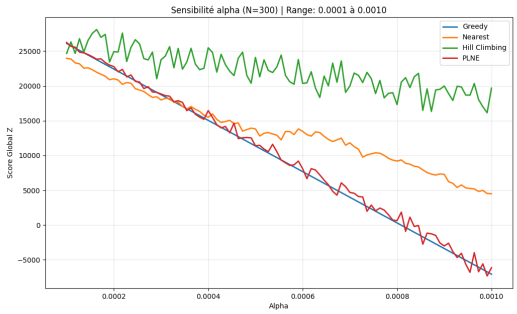
Le paramètre α représente le coût de transport linéaire par unité de poids. Nous avons testé quatre plages de valeurs pour observer le basculement entre un profit positif et un score négatif, moment où le coût lié à la charge surpasse le profit des collectes.

Dans l'ensemble des graphiques suivants, nous respectons le code couleur suivant :

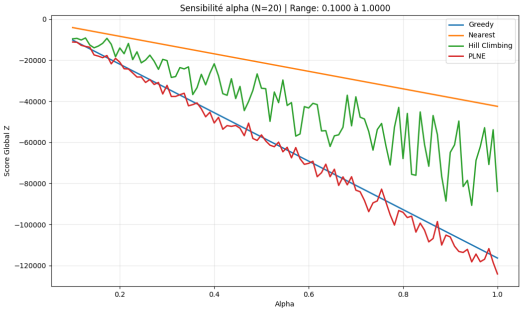
- Vert : Hill Climbing (Iterative)
- Rouge : PLNE
- Bleu : Greedy
- Jaune/Orange : Nearest Neighbor



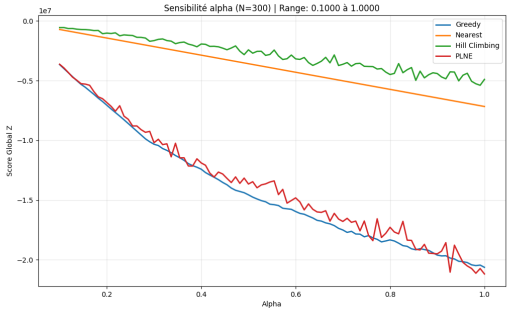
(a) $N = 20$, $\alpha \in [0.0001, 0.0010]$



(b) $N = 300$, $\alpha \in [0.0001, 0.0010]$



(c) $N = 20$, $\alpha \in [0.1, 1.0]$



(d) $N = 300$, $\alpha \in [0.1, 1.0]$

FIGURE 2 – Évolution du score global Z en fonction de α pour différentes échelles de poids et tailles d'instances.

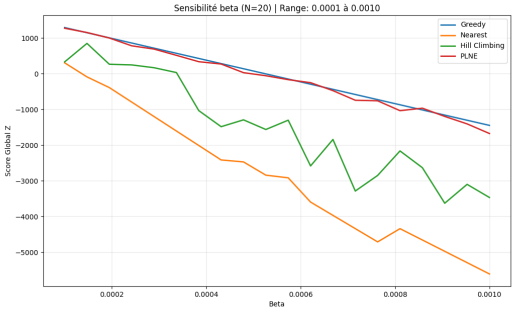
Analyse : Sur les grandes instances ($N = 300$), on observe que l'algorithme *Hill Climbing* (en vert) maintient les meilleurs scores. À l'inverse, le PLNE et le *Greedy* s'effondrent de manière quasi identique à mesure que α augmente. Ce comportement est attendu : lorsque α croît, l'impact du poids dans la fonction objectif devient prépondérant. Or, ces deux méthodes ne prennent pas en compte les distances de déplacement, ce qui dégrade fortement la qualité de leur solution globale sur de longs trajets chargés.

Cependant, sur de petites instances ($N = 20$) avec des valeurs d'α faibles, le *Greedy* et le PLNE surpassent nettement les méthodes *Nearest* et *Hill Climbing*. Dans ce scénario, le coût du transport impacte beaucoup moins la fonction objectif que le profit immédiat des objets. Le PLNE et le *Greedy* parviennent alors à maximiser les gains de collecte sans être pénalisés par l'absence d'optimisation géographique de leur tournée.

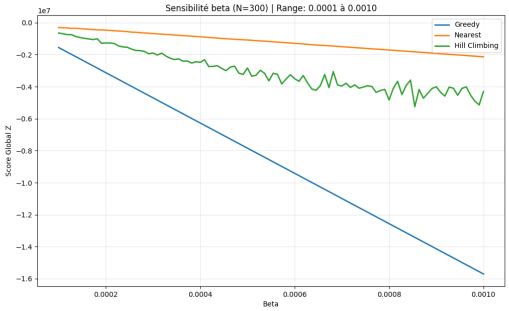
7.2 Sensibilité au paramètre β

Le paramètre β introduit un coût de transport quadratique proportionnel au carré de la charge du camion (w_t^2), simulant une usure mécanique ou une consommation énergétique s'accélérant avec le poids.

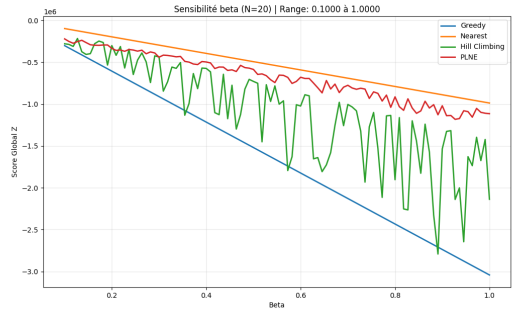
Comme précédemment, nous utilisons le même code couleur pour l'analyse :



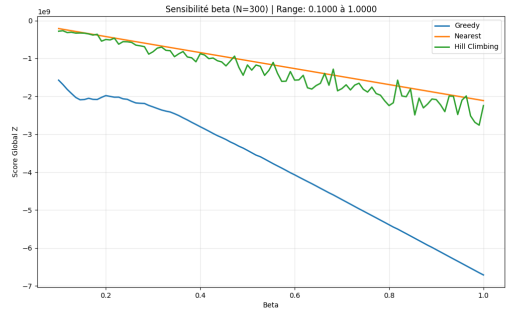
(a) $N = 20$, $\beta \in [0.0001, 0.0010]$



(b) $N = 300$, $\beta \in [0.0001, 0.0010]$



(c) $N = 20$, $\beta \in [0.1, 1.0]$



(d) $N = 300$, $\beta \in [0.1, 1.0]$

FIGURE 3 – Sensibilité du score global Z face au coût quadratique β pour différentes tailles d'instances.

Analyse des résultats :

— **Passage à l'échelle et PLNE :** Sur les instances $N = 20$, le modèle PLNE est présent et performant, particulièrement pour les faibles valeurs de β. En revanche, pour $N = 300$, le PLNE a dû être écarté car la complexité du problème quadratique rend les temps de résolution prohibitifs pour le solveur exact.

— **Domination du coût :** Pour les valeurs élevées ($\beta > 0.1$), le score global chute drastiquement vers des valeurs négatives. L'impact du carré du poids devient la composante principale de la fonction objectif, rendant le ramassage d'objets lourds économiquement pénalisant.

— **Robustesse des heuristiques :** L'algorithme *Hill Climbing* (en vert) et *Nearest Neighbor* (en ocre) s'avèrent les plus résilients. Ils parviennent à optimiser la structure de la tournée pour limiter la charge accumulée, contrairement à l'approche *Greedy* (en bleu) qui subit plus fortement la pénalité quadratique sur les grandes instances.

7.3 Temps d'exécution et Scalabilité

L'efficacité d'un algorithme ne se mesure pas seulement à la qualité de sa solution, mais aussi à son temps de réponse face à la croissance de la taille du problème (N).

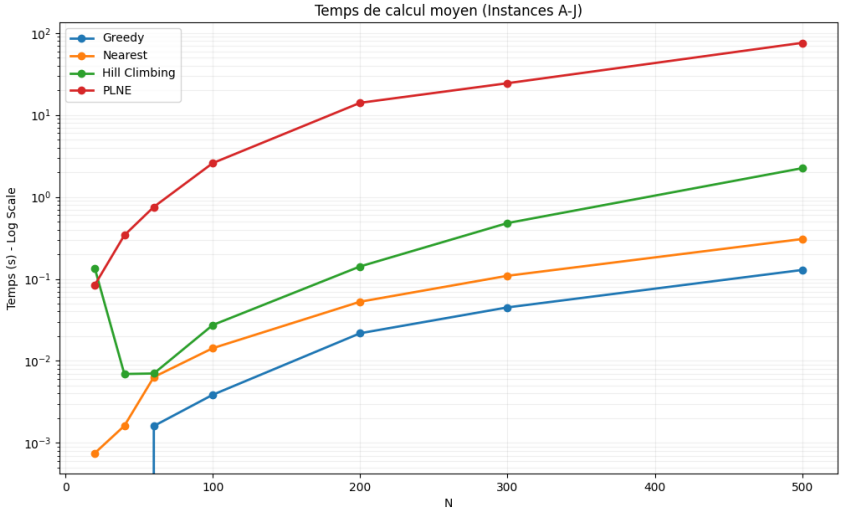


FIGURE 4 – Comparaison des temps de calcul moyens (échelle logarithmique).

Analyse :

— Le **PLNE** présente une croissance exponentielle, dépassant les 10 secondes dès $N = 200$, ce qui confirme la nécessité d'heuristiques pour les problèmes réels.

— **Hill Climbing** offre un excellent compromis : bien que plus lent que le *Greedy*, il reste sous la barre des 2 secondes pour $N = 500$ tout en fournissant des solutions de bien meilleure qualité (voir Fig. 2).

— Les méthodes **Greedy** et **Nearest** sont quasi instantanées mais offrent des performances instables sur les grandes instances.

7.4 conclusion