Projet d'Optimisation

Problème du Voyageur de Commerce (TSP)

Rapport Final

Algorithmes Exactes et Heuristiques

Master MIASHS — IMA-UCO (2025–2026)

Auteurs : Matthias Jourdren / Maxence Cornu Basset / Gaëtan Pezas

Date : 19 janvier 2026

Table des matières

Introduction

Le Problème du Voyageur de Commerce (**Traveling Salesperson Problem – TSP**) est un problème classique d'optimisation combinatoire. Il consiste à déterminer la tournée de coût minimal permettant de visiter chaque ville exactement une fois et de revenir à la ville de départ.

Ce problème est connu pour sa difficulté algorithmique, car il appartient à la classe des problèmes **NP-difficiles**. Dans ce projet, plusieurs approches ont été implémentées afin de comparer leurs performances et leur qualité de solution.

Méthodes Implémentées

Heuristique constructive : Plus proche voisin

Dans notre implémentation du TSP, nous avons adapté l'heuristique du plus proche voisin avec un graphe complet pondéré représentant les villes et leurs distances. La tournée est construite à partir d'une ville de départ fixée (la ville 0), en ajoutant à chaque étape une ville non encore visitée. Les distances sont directement lues dans la matrice d'adjacence fournie en entrée.

Les villes déjà visitées sont mémorisées afin de garantir que chaque sommet n'est visité qu'une seule fois. Une fois toutes les villes intégrées à la tournée, le coût du retour vers la ville de départ est ajouté afin d'obtenir un cycle hamiltonien valide. Cette méthode fournit rapidement une solution réalisable, qui sert également de solution initiale pour les méthodes plus avancées.

Algorithme 1 : Heuristique constructive du Plus Proche Voisin (avec boucle et indentation)

Données(*Matrice des distances D, nombre de villes n*)

Initialisation :

$v_{actuelle} \leftarrow 0$ `// Ville de départ`

$Tournee \leftarrow [v_{actuelle}]$

$VillesVisitees \leftarrow \{v_{actuelle}\}$

$CoutTotal \leftarrow 0$

Construction de la tournée :

tant que $|Tournee| < n$ **faire**

`// Sélectionner la ville non visitée la plus proche`

$v_{suivante} \leftarrow \text{argmin}_{v \notin VillesVisitees} D[v_{actuelle}, v]$

`// Mettre à jour la tournée et le coût`

Ajouter $v_{suivante}$ à $Tournee$

Ajouter $v_{suivante}$ à $VillesVisitees$

$CoutTotal \leftarrow CoutTotal + D[v_{actuelle}, v_{suivante}]$

`// Passer à la ville suivante`

$v_{actuelle} \leftarrow v_{suivante}$

fin

Clôture de la tournée :

$CoutTotal \leftarrow CoutTotal + D[v_{actuelle}, Tournee[0]]$ `// Retour à la ville de départ`

$Tournee$, $CoutTotal$

Complexité À chaque itération, toutes les villes non visitées sont examinées. La complexité temporelle est donc :

$$\mathcal{O}(n^2)$$

Algorithme exact : Branch and Bound

L'algorithme de Branch and Bound est adapté au TSP en explorant les permutations possibles des villes sous forme de chemins partiels construits récursivement à partir d'une ville de départ fixée. Chaque chemin partiel représente une tournée incomplète, dont le coût cumulé est calculé à partir des poids du graphe.

Une borne supérieure correspond au meilleur coût trouvé jusqu'à présent. Dès que le coût partiel d'un chemin dépasse cette borne, la branche correspondante est abandonnée, ce qui permet de réduire l'espace de recherche. Afin de rendre l'algorithme exploitable en pratique, une limite de temps est imposée, ce qui permet d'interrompre la recherche pour les instances trop grandes.

Algorithme 2 : Algorithme exact Branch and Bound pour le TSP

Données(*Matrice des distances D, nombre de villes n*)
$MeilleurCout \leftarrow +\infty$
$MeilleureTournee \leftarrow \emptyset$

Procédure Explorer(C_p, K_p, V_p)
// C_p : chemin partiel, K_p : coût partiel, V_p : villes visitées
si $K_p \geq MeilleurCout$ **alors**
  retourner // Élagage
fin
si $|C_p| = n$ **alors**
  $K_{final} \leftarrow K_p + D[dernier(C_p), premier(C_p)]$ **si** $K_{final} < MeilleurCout$ **alors**
    $MeilleurCout \leftarrow K_{final}$
    $MeilleureTournee \leftarrow C_p$
  fin
  retourner
fin
pour chaque *ville* $v \notin V_p$ **faire**
  Explorer($C_p \cup \{v\}$, $K_p + D[dernier(C_p), v]$, $V_p \cup \{v\}$)
fin
Explorer([0], 0, {0})
$MeilleureTournee$, $MeilleurCout$

Complexité Dans le pire cas, toutes les permutations sont explorées. La complexité est donc :

$$\mathcal{O}(n!)$$

Heuristique de recherche locale : 2-opt

La recherche locale 2-opt est appliquée au TSP en partant d'une tournée initiale obtenue par l'heuristique constructive. La tournée est représentée comme une liste ordonnée de villes, et son coût est calculé à partir de la matrice de distances.

L'adaptation repose sur l'évaluation systématique de modifications locales de la tournée, consistant à inverser des segments du parcours. Lorsqu'une modification permet de réduire le coût total, elle est acceptée et devient la nouvelle solution courante. Le processus est répété jusqu'à ce qu'aucune amélioration ne soit possible, ce qui correspond à un optimum local pour le problème du TSP.

Algorithme 3 : Recherche locale 2-opt pour le TSP

Données(*Tournée initiale T, matrice des distances D*) $amelioration \leftarrow Vrai$

tant que $amelioration$ **faire**
  $amelioration \leftarrow Faux$
  pour $i = 1$ **à** $n - 2$ **faire**
    pour $k = i + 1$ **à** $n - 1$ **faire**
      $T' \leftarrow$ inverser le segment $[i, k]$ de T
      si $Cout(T') < Cout(T)$ **alors**
        $T \leftarrow T'$
        $amelioration \leftarrow Vrai$
        Sortir des boucles
      fin
    fin
  fin
fin
T

Complexité Les échanges possibles sont en $\mathcal{O}(n^2)$. La complexité globale est donc :

$$\mathcal{O}(n^2)$$

Méta-heuristique : GRASP

La méta-heuristique GRASP est adaptée au TSP en combinant une phase constructive randomisée et une phase de recherche locale. La phase constructive repose sur une version modifiée de l'heuristique du plus proche voisin, dans laquelle le choix de la prochaine ville est effectué aléatoirement à partir d'une liste restreinte de candidats, définie à partir des distances dans le graphe.

Le paramètre α permet de contrôler le compromis entre choix glouton et diversification. Chaque solution construite est ensuite améliorée à l'aide de la recherche locale 2-opt. Cette procédure est répétée sur un nombre fixé d'itérations, et la meilleure tournée obtenue est conservée comme solution finale pour le TSP.

Algorithme 4 : Méta-heuristique GRASP pour le TSP

Données(*Matrice des distances D, paramètre α, nombre d'itérations $MaxIter$*)

$MeilleurCout \leftarrow +\infty$

$MeilleureSolution \leftarrow \emptyset$

pour $iter = 1$ **à** $MaxIter$ **faire**

// Phase 1 : Construction gloutonne randomisée

Choisir une ville de départ aléatoire (ville 0)

$T \leftarrow [v_{depart}]$

tant que $|T| < n$ **faire**

$Candidats \leftarrow$ villes non visitées

$C_{min}, C_{max} \leftarrow$ coûts min et max depuis la ville courante

$LRC \leftarrow \{v \in Candidats \mid D \leq C_{min} + \alpha(C_{max} - C_{min})\}$

$v_{choisie} \leftarrow$ choix aléatoire dans LRC

Ajouter $v_{choisie}$ à T

fin

// Phase 2 : Intensification

$T_{opt} \leftarrow$ RechercheLocale2Opt(T)

si $Cout(T_{opt}) < MeilleurCout$ **alors**

$MeilleurCout \leftarrow Cout(T_{opt})$

$MeilleureSolution \leftarrow T_{opt}$

fin

fin

$MeilleureSolution$

Complexité Chaque itération a un coût en $\mathcal{O}(n^2)$. La complexité totale est donc :

$$\mathcal{O}(I_{max} \times n^2)$$

Analyse des Résultats

Cette section présente les performances comparées des différentes méthodes implémentées pour le TSP sur plusieurs instances de test. Les résultats sont analysés en termes de coût de la tournée et de temps d'exécution.

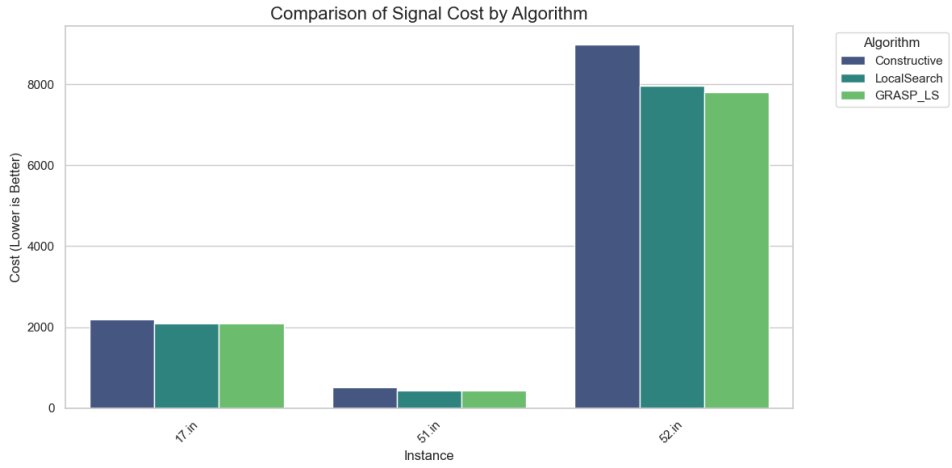

FIGURE 1 – Comparaison du coût de la tournée par algorithme

Le graphique ci-dessus (*figure 1*) montre que les méthodes heuristiques (Constructive, Local Search et GRASP_LS) parviennent à trouver des solutions très proches les unes des autres sur les petites instances. Cependant, sur les instances plus complexes, la supériorité des méthodes d'amélioration se dessine.

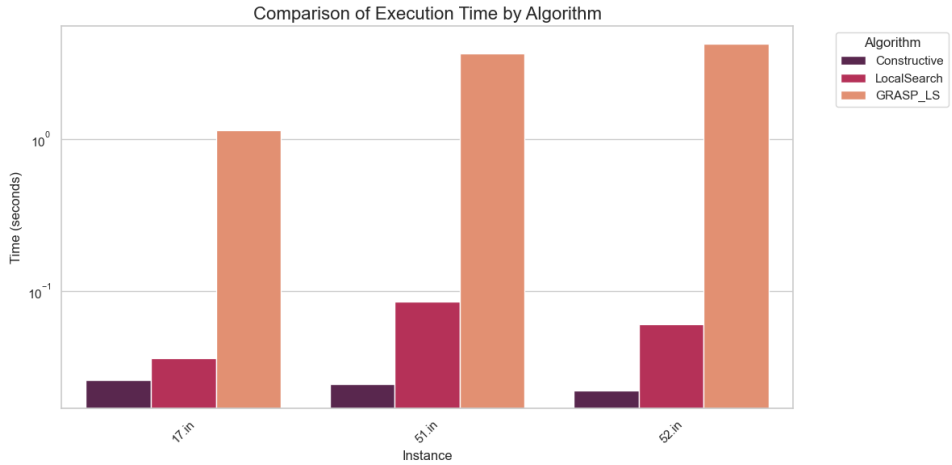

FIGURE 2 – Comparaison du temps d'exécution par algorithme

L'analyse des temps d'exécution (*figure 2*) souligne l'explosion combinatoire de l'algorithme exact **Branch and Bound**, qui atteint la limite de temps (300 secondes) dès que le nombre de villes augmente, sans pouvoir garantir l'optimalité. À l'opposé, l'heuristique constructive est quasi-instantanée, tandis que **GRASP_LS** présente un surcoût temporel maîtrisé par rapport à la recherche locale simple.

Conclusion

Ce projet a permis d'explorer différents paradigmes de résolution du Problème du Voyageur de Commerce (TSP). L'étude comparative des performances des algorithmes implémentés conduit aux constats suivants :

- **L'algorithme Branch and Bound** est indispensable pour garantir l'optimalité sur de très petites instances, mais sa complexité factorielle le rend rapidement inexploitable pour des problèmes de taille moyenne.
- **L'heuristique constructive** du plus proche voisin offre une excellente réactivité (environ 0,02 seconde), mais génère des solutions souvent éloignées de l'optimum.
- **La recherche locale (2-opt)** permet d'améliorer significativement les solutions initiales avec un surcoût computationnel très faible, ce qui en fait un outil essentiel dans une approche heuristique.

Élection de la meilleure méthode : Au regard de l'ensemble des instances testées, la méta-heuristique **GRASP_LS** s'impose comme la méthode la plus performante. En combinant efficacement la diversification (phase constructive randomisée) et l'intensification (recherche locale 2-opt), elle obtient systématiquement les meilleures solutions, au prix d'un temps de calcul légèrement supérieur mais toujours raisonnable.

Sur les instances testées (notamment *52.in*), **GRASP_LS** atteint le coût le plus faible (7800 contre 8980 pour la méthode constructive) tout en restant sous la barre des 5 secondes d'exécution. Elle constitue ainsi le meilleur compromis entre qualité de solution et temps de calcul, et est recommandée pour une utilisation généralisée dans le cadre du TSP.