

# Projet d'Optimisation

Le Problème du Voyageur de Commerce (TSP)

---

## Rapport Final

Implémentation et Analyse d'Algorithmes

---

Master MIASHS — IMA-UCO (2025–2026)

**Auteurs :** Matthias Jourden / Maxence Cornu Basset / Gaëtan Pezas

**Date :** 16 janvier 2026

# Table des matières

<b>1</b>	<b>Introduction et Énoncé du Problème</b>	<b>2</b>
<b>2</b>	<b>Méthodes Implémentées</b>	<b>2</b>
2.1	Algorithme Exact : Branch and Bound . . . . .	2
2.2	Heuristique Constructive : Plus Proche Voisin . . . . .	3
2.3	Recherche Locale : 2-opt . . . . .	4
2.4	Méta-heuristique : GRASP_LS . . . . .	6
<b>3</b>	<b>Expérimentations et Résultats</b>	<b>6</b>
3.1	Analyse . . . . .	6
<b>4</b>	<b>Conclusion</b>	<b>7</b>

# Introduction et Énoncé du Problème

Le Problème du Voyageur de Commerce (**Traveling Salesperson Problem - TSP**) est l'un des problèmes d'optimisation combinatoire les plus célèbres.

L'objectif est de trouver le chemin le plus court qui visite chaque ville exactement une fois et revient au point de départ. Mathématiquement, cela revient à chercher un **cycle hamiltonien de poids minimum** dans un graphe complet pondéré  $G = (V, E)$ . Le TSP est classé comme **NP-difficile**.

Nous avons implémenté et comparé quatre approches différentes :

- Une méthode exacte (Branch and Bound)
- Une heuristique constructive (Plus proche voisin)
- Une heuristique de recherche locale (2-opt)
- Une heuristique de recherche locale (2-opt)
- Une méta-heuristique (GRASP\_LS)

## Méthodes Implémentées

### Algorithme Exact : Branch and Bound

L'algorithme "Branch and Bound" est une méthode qui cherche la meilleure solution en explorant toutes les possibilités, comme dans un arbre. Pour gagner du temps, on utilise une astuce : si un début de chemin est déjà trop long par rapport au meilleur chemin complet que l'on connaît, on arrête tout de suite d'explorer cette direction. On appelle cela "l'élagage". Pour notre problème de voyageur, cela nous permet de trouver le chemin le plus court possible (optimal) sans avoir besoin de tester tous les chemins un par un, ce qui serait impossible car il y en a trop.

**Complexité :**  $O(n!)$ .

---

**Algorithme 1 : Pseudo-code Branch and Bound (Détailé)**

---

**Données :** chemin\_actuel (liste des villes déjà visitées), cout\_actuel (longueur du chemin)  
// Condition d'arrêt (Élagage) : si on dépasse déjà le meilleur score connu  
**Si**  $\text{cout\_actuel} \geq \text{meilleur\_cout\_trouve}$  **Retourner** (On arrête cette branche, inutile de continuer)  
// Si on a visité toutes les villes (feuille de l'arbre)  
**Si**  $\text{longueur}(\text{chemin\_actuel}) == \text{Nombre\_Total\_Villes}$  // On ajoute le retour au point de départ  
 $\text{cout\_total} \leftarrow \text{cout\_actuel} + \text{distance}(\text{derniere\_ville}, \text{ville\_depart})$   
// Est-ce que c'est un nouveau record ?  
**Si**  $\text{cout\_total} < \text{meilleur\_cout\_trouve}$   $\text{meilleur\_cout\_trouve} \leftarrow \text{cout\_total}$   
 $\text{meilleur\_tour} \leftarrow \text{copier}(\text{chemin\_actuel})$   
**Retourner**  
// Exploration des villes suivantes  
**Pour chaque** ville  $v$  qui n'est PAS encore dans chemin\_actuel Marquer  $v$  comme visitée  
// Appel récursif : on continue le chemin avec  $v$   
 $\text{BranchAndBound}(\text{chemin\_actuel} + v, \text{cout\_actuel} + \text{distance}(\text{derniere}, v))$   
Marquer  $v$  comme non visitée (pour permettre d'autres chemins)

---

## Heuristique Constructive : Plus Proche Voisin

L'algorithme du Plus Proche Voisin est très intuitif : on part d'une ville et on va toujours vers la ville la plus proche qui n'a pas encore été visitée. C'est simple et rapide. Pour notre problème, cette méthode permet d'obtenir très vite un premier trajet complet. Même si ce trajet n'est pas parfait (il peut être un peu long), il sert de base de départ.

**Complexité :**  $O(n^2)$ .

---

**Algorithme 2 : Pseudo-code Plus Proche Voisin (Détailé)**

---

```
// Initialisation
Choisir une ville de départ arbitraire (ex : ville 0)
ville_actuelle ← ville_depart
Marquer ville_actuelle comme "VISITÉE"
chemin ← [ville_actuelle]
cout_total ← 0
// Boucle principale : on visite tout le monde
Tant que il reste des villes NON visitées // Recherche du voisin le plus
    proche
    Trouver la ville  $v$  non visitée qui a la plus petite distance avec ville_actuelle
    // Mise à jour
    Ajouter  $v$  au chemin
    cout_total ← cout_total + distance(ville_actuelle,  $v$ )
    ville_actuelle ←  $v$  (on se déplace)
    Marquer  $v$  comme "VISITÉE"
    // Ne pas oublier de revenir au point de départ !
    cout_total ← cout_total + distance(ville_actuelle, ville_depart)
    Ajouter ville_depart à la fin du chemin
```

---

## Recherche Locale : 2-opt

Le 2-opt est une technique pour corriger les défauts d'un trajet existant. Dans notre cas (villes sur une carte 2D), si deux routes se croisent, le chemin est forcément plus long que si elles ne se croisent pas. Le 2-opt repère ces croisements et "démêle" le chemin en inversant une section du tour. En répétant cela jusqu'à ce qu'il n'y ait plus de croisements, on obtient un trajet beaucoup plus propre et court.

**Complexité :**  $O(n^2)$ .

---

**Algorithme 3 : Pseudo-code Recherche Locale 2-opt (Détailé)**

---

```
// On part d'un tour existant (même mauvais)
amelioration_trouvee ← Vrai
Tant que amelioration_trouvee est Vrai amelioration_trouvee ← Faux (on
    remet à faux pour ce tour)
// On teste toutes les paires d'arêtes possibles (i et k)
Pour i de 1 à N-2 Pour k de i+1 à N-1 // On simule l'inversion du
    segment entre i et k
nouveau_tour ← InverserSegment(tour_actuel, i, k)
// Est-ce que ça a raccourci le trajet ?
Si cout(nouveau_tour) < cout(tour_actuel) tour_actuel ← nouveau_tour (On
    valide le changement)
amelioration_trouvee ← Vrai
// On recommence la boucle principale pour revérifier
Sortir des boucles "Pour" et recommencer le "Tant que"
Retourner tour_actuel
```

---

## Méta-heuristique : GRASP\_LS

La méthode GRASP mélange hasard et optimisation pour trouver une solution excellente. Si on cherche toujours le "meilleur" choix immédiat (comme le Plus Proche Voisin), on risque de se coincer dans une solution moyenne. GRASP construit une solution en choisissant parfois une ville un peu moins bonne au hasard, ce qui permet d'explorer des chemins différents. Ensuite, on applique le 2-opt pour améliorer ce chemin à fond. En répétant cela plusieurs fois, on a plus de chances de tomber sur le meilleur trajet possible.

**Complexité :** Dépend du nombre de fois qu'on répète l'algorithme.

---

**Algorithme 4 :** Pseudo-code GRASP\_LS (Détailé)

---

```
meilleur_cout_global ← Infini
meilleure_solution_globale ← Vide
Pour iteration de 1 à Nombre_Max_Iterations // Étape 1 : Construction
    semi-aléatoire
// alpha = 0 (glouton pur) à 1 (totalement aléatoire)
Solution_Candidate ← ConstructionGloutonneAleatoire(alpha)
// Étape 2 : Optimisation locale
Solution_Optimisee ← RechercheLocale2Opt(Solution_Candidate)
// Mise à jour du record
Si cout(Solution_Optimisee) < meilleur_cout_global meilleur_cout_global ←
    cout(Solution_Optimisee)
meilleure_solution_globale ← Solution_Optimisee
Retourner meilleure_solution_globale
```

---

## Expérimentations et Résultats

Les tests ont été effectués sur des instances de taille 20 (données types `random` et `double_circle`).

Algorithme	Status	Temps (s)	Coût Relatif	Qualité
Exact (B&B)	Timeout	> 60.0	-	Optimale
Plus Proche Voisin	Succès	0.03	100%	Faible
Recherche Locale	Succès	0.03	95%	Moyenne
<b>GRASP_LS</b>	Succès	0.12	<b>90%</b>	<b>Excellente</b>

TABLE 1 – Comparaison des algorithmes sur des instances de 20 villes

## Analyse

1. **Exact (B&B) :** Souffre de l'explosion combinatoire. Au-delà de 15 villes, le temps d'exécution dépasse les limites raisonnables.

2. **Plus Proche Voisin** : Très rapide mais les choix locaux mènent souvent à des détours coûteux en fin de parcours.
3. **2-opt** : Indispensable pour raffiner les solutions et supprimer les intersections d'arêtes.
4. **GRASP** : C'est la méthode la plus performante. En explorant plusieurs zones de l'espace de recherche (randomisation) et en optimisant chaque point (2-opt), elle trouve des solutions proches de l'optimum très rapidement.

## Conclusion

L'heuristique retenue comme étant la plus efficace est **GRASP\_LS**. Elle offre le meilleur compromis entre temps de calcul et qualité de la solution, ce qui est essentiel pour les problèmes de grande taille. Sa robustesse provient de sa capacité à éviter les optima locaux tout en exploitant les forces de la recherche locale.