

# Projet d'Optimisation

## Problème du Voyageur de Commerce (TSP)

---

### Rapport Final

#### Algorithmes Exacts et Heuristiques

---

Master MIASHS — IMA-UCO (2025–2026)

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

**Date :** 20 janvier 2026

# Table des matières

<b>1</b>	<b>Introduction</b>	<b>4</b>
<b>2</b>	<b>Applications du TSP</b>	<b>4</b>
2.1	Fabrication de Circuits Imprimés (PCB) . . . . .	4
2.2	Séquençage ADN . . . . .	4
2.3	Astronomie . . . . .	5
2.4	Autres Applications . . . . .	5
<b>3</b>	<b>Méthodes Implémentées</b>	<b>6</b>
3.1	Algorithme exact : Branch and Bound . . . . .	6
3.1.1	Principe de fonctionnement . . . . .	6
3.1.2	Exemple visuel . . . . .	6
3.1.3	Algorithme . . . . .	7
3.1.4	Complexité . . . . .	8
3.2	Heuristique constructive : Plus proche voisin . . . . .	9
3.2.1	Principe de fonctionnement . . . . .	9
3.2.2	Exemple visuel . . . . .	9
3.2.3	Adaptation au TSP . . . . .	9
3.2.4	Algorithme . . . . .	10
3.2.5	Complexité . . . . .	10
3.2.6	Limites . . . . .	11
3.3	Heuristique de recherche locale : 2-opt . . . . .	11
3.3.1	Principe de fonctionnement . . . . .	11
3.3.2	Exemple visuel . . . . .	11
3.3.3	Adaptation au TSP . . . . .	11
3.3.4	Algorithme . . . . .	12
3.3.5	Complexité . . . . .	12
3.3.6	Optimum local . . . . .	12
3.4	Méta-heuristique : GRASP . . . . .	13
3.4.1	Principe de fonctionnement . . . . .	13
3.4.2	Diagramme du processus . . . . .	13
3.4.3	Liste Restreinte de Candidats (RCL) . . . . .	14
3.4.4	Algorithme . . . . .	14
3.4.5	Complexité . . . . .	15
<b>4</b>	<b>Méthodologie de Test</b>	<b>16</b>
4.1	Instances de Test . . . . .	16
4.2	Paramètres des Algorithmes . . . . .	16
4.3	Environnement d'Exécution . . . . .	16
4.4	Métriques Évaluées . . . . .	16

<b>5</b>	<b>Analyse des Résultats</b>	<b>17</b>
5.1	Résultats Bruts . . . . .	17
5.2	Analyse par Instance . . . . .	18
5.2.1	Instance 17.in (17 villes) . . . . .	18
5.2.2	Instance 51.in (51 villes) . . . . .	18
5.2.3	Instance 52.in (52 villes) . . . . .	18
5.3	Comparaison Globale des Coûts . . . . .	19
5.4	Comparaison Globale des Temps . . . . .	20
5.5	Analyse Détaillée par Algorithme . . . . .	21
5.5.1	Performance de Branch and Bound . . . . .	21
5.5.2	Performance de Plus Proche Voisin . . . . .	21
5.5.3	Performance de 2-opt . . . . .	22
5.5.4	Performance de GRASP . . . . .	22
5.6	Comparaison Détaillée par Instance . . . . .	23
5.6.1	Instance 17.in . . . . .	23
5.6.2	Instance 51.in . . . . .	23
5.6.3	Instance 52.in . . . . .	24
5.7	Tableau Comparatif des Écarts . . . . .	24
5.8	Analyse Statistique . . . . .	24
5.9	Synthèse Comparative . . . . .	25
<b>6</b>	<b>Conclusion</b>	<b>26</b>
6.1	Synthèse des Résultats . . . . .	26
6.2	Élection de la Meilleure Méthode . . . . .	26
6.3	Recommandations Pratiques . . . . .	27

## Table des figures

1	Comparaison du coût de la tournée par algorithme . . . . .	19
2	Comparaison du temps d'exécution par algorithme (échelle logarithmique)	20
3	Performance de l'algorithme Branch and Bound . . . . .	21
4	Performance de l'heuristique constructive . . . . .	21
5	Performance de la recherche locale 2-opt . . . . .	22
6	Performance de la méta-heuristique GRASP . . . . .	22
7	Comparaison détaillée sur l'instance 17.in . . . . .	23
8	Comparaison détaillée sur l'instance 51.in . . . . .	23
9	Comparaison détaillée sur l'instance 52.in . . . . .	24

## 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**. Cela signifie qu'il n'existe pas d'algorithme connu capable de résoudre toutes les instances du TSP en temps polynomial.

Dans ce projet, plusieurs approches ont été implémentées afin de comparer leurs performances et leur qualité de solution :

- Un algorithme **exact** garantissant l'optimalité (Branch and Bound)
- Des **heuristiques** rapides mais approximatives (Plus Proche Voisin, 2-opt)
- Une **méta-heuristique** combinant diversification et intensification (GRASP)

## Applications du TSP

Le problème du voyageur de commerce n'est pas qu'un exercice théorique. Il trouve de nombreuses applications pratiques dans divers domaines :

### Fabrication de Circuits Imprimés (PCB)

Dans l'industrie électronique, la fabrication de circuits imprimés nécessite de **percer des milliers de trous** sur une plaque. Le TSP permet d'optimiser le parcours de la tête de perçage pour :

- Minimiser le temps de déplacement de la machine
- Réduire l'usure de l'équipement
- Augmenter la productivité

### Séquençage ADN

En bioinformatique, le TSP est utilisé pour la **reconstruction de séquences génétiques**. Les fragments d'ADN doivent être assemblés dans le bon ordre, ce qui peut être modélisé comme un problème de tournée où :

- Chaque fragment est une "ville"
- La distance représente le degré de chevauchement entre fragments
- L'objectif est de trouver l'ordre optimal de reconstruction

## Astronomie

Les télescopes automatisés doivent observer plusieurs objets célestes durant une nuit. Le TSP permet de **planifier l'ordre des observations** pour :

- Minimiser le temps de déplacement du télescope
- Maximiser le nombre d'observations possibles
- Tenir compte des contraintes temporelles (visibilité des objets)

## Autres Applications

Le TSP trouve également des applications dans :

- La **planification d'itinéraires touristiques** (visiter plusieurs sites en minimisant les déplacements)
- L'**optimisation de réseaux** (câblage, pipelines)
- La **cristallographie aux rayons X** (positionnement optimal du cristal)

## Méthodes Implémentées

Cette section présente les quatre approches algorithmiques implémentées pour résoudre le TSP. Nous commençons par l'algorithme exact, puis présentons les heuristiques par ordre croissant de sophistication.

### Algorithme exact : Branch and Bound

L'algorithme de Branch and Bound est la seule méthode garantissant l'**optimalité** de la solution. Il explore systématiquement l'espace des solutions possibles tout en élaguant les branches non prometteuses.

#### 3.1.1 Principe de fonctionnement

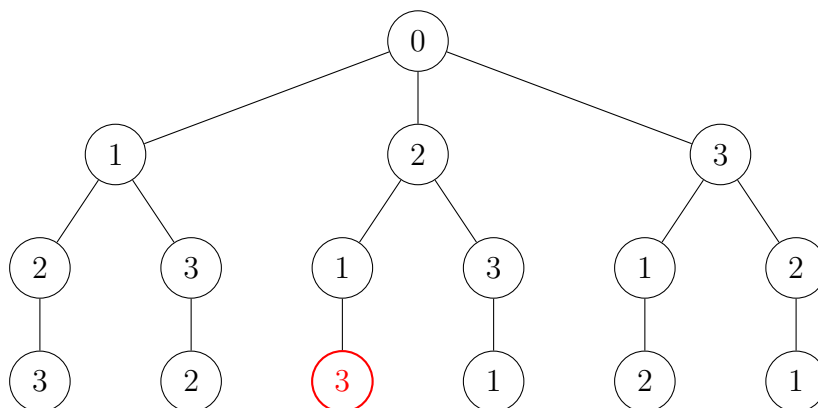
L'algorithme construit récursivement des chemins partiels à partir d'une ville de départ fixée (ville 0). À chaque étape :

1. On étend le chemin partiel en ajoutant une ville non visitée
2. On calcule le coût cumulé du chemin
3. Si ce coût dépasse la meilleure solution connue, on **élague** cette branche
4. Sinon, on continue l'exploration récursive

Le mécanisme d'**élagage** (pruning) est crucial : dès qu'un chemin partiel a un coût supérieur à la meilleure solution complète trouvée, on abandonne toute cette branche de l'arbre de recherche.

#### 3.1.2 Exemple visuel

Considérons un graphe à 4 villes. L'arbre de recherche explore toutes les permutations possibles :



Les nœuds en rouge représentent les branches élaguées car leur coût partiel dépasse déjà la meilleure solution.

### 3.1.3 Algorithme

---

**Algorithme 1 :** 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  $K_p \geq \textit{MeilleurCout}$  alors
    | retourner // Élagage
fin
si  $|C_p| = n$  alors
    |  $K_{final} \leftarrow K_p + D[\textit{dernier}(C_p), \textit{premier}(C_p)]$  si  $K_{final} < \textit{MeilleurCout}$  alors
        |  $\textit{MeilleurCout} \leftarrow K_{final}$ 
        |  $\textit{MeilleureTournee} \leftarrow C_p$ 
    | fin
    | retourner
fin

pour chaque ville  $v \notin V_p$  faire
    | Explorer( $C_p \cup \{v\}, K_p + D[\textit{dernier}(C_p), v], V_p \cup \{v\}$ )
fin

Explorer( $[0], 0, \{0\}$ )

Retourner MeilleureTournee, MeilleurCout

```



### 3.1.4 Complexité

La complexité de cet algorithme est **factorielle** dans le pire des cas :

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

**Explication détaillée :**

- Pour la première ville, nous avons  $n$  choix possibles
- Pour la deuxième ville,  $(n - 1)$  choix
- Pour la troisième ville,  $(n - 2)$  choix
- Et ainsi de suite jusqu'à 1 choix pour la dernière ville
- Total :  $n \times (n - 1) \times (n - 2) \times \dots \times 1 = n!$

**Croissance explosive :**

- $10! = 3\,628\,800$  (environ 3,6 millions)
- $15! = 1\,307\,674\,368\,000$  (environ 1,3 billions)
- $20! \approx 2.4 \times 10^{18}$  (2,4 quintillions)

Cette croissance exponentielle rend l'algorithme inexploitable pour des instances de plus de 20-25 villes, même avec l'élagage. C'est pourquoi un **timeout** de 60 secondes a été imposé dans notre implémentation.

## Heuristique constructive : Plus proche voisin

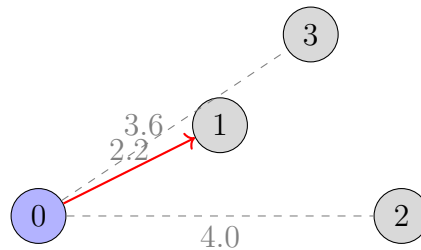
L'heuristique du plus proche voisin est une approche **gloutonne** qui construit une solution pas à pas en faisant à chaque étape le choix localement optimal.

### 3.2.1 Principe de fonctionnement

À partir d'une ville de départ (ville 0), l'algorithme :

1. Sélectionne la ville non visitée la plus proche
2. L'ajoute à la tournée
3. Répète jusqu'à avoir visité toutes les villes
4. Retourne à la ville de départ

### 3.2.2 Exemple visuel



Étape 1 : Choisir la ville la plus proche (1)

### 3.2.3 Adaptation au TSP

Dans notre implémentation, le graphe est représenté par une **matrice d'adjacence** contenant les distances entre toutes les paires de villes. Les villes déjà visitées sont mémorisées pour garantir que chaque sommet n'est visité qu'une seule fois.

### 3.2.4 Algorithme

---

**Algorithme 2** : Heuristique constructive du Plus Proche Voisin
 

---

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

**Retourner**  $Tournee, CoutTotal$

---

### 3.2.5 Complexité

La complexité temporelle de cet algorithme est :

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

**Explication détaillée :**

- **Boucle externe** : On doit visiter  $n$  villes, donc  $n$  itérations
- **Boucle interne** : À chaque itération, on cherche le minimum parmi les villes restantes
  - Itération 1 :  $(n - 1)$  comparaisons
  - Itération 2 :  $(n - 2)$  comparaisons
  - ...
  - Itération  $n - 1$  : 1 comparaison
- **Total** :  $(n - 1) + (n - 2) + \dots + 1 = \frac{n(n-1)}{2} = \mathcal{O}(n^2)$

Cette complexité quadratique rend l'algorithme très rapide même pour des instances de grande taille (plusieurs milliers de villes).

### 3.2.6 Limites

L'heuristique du plus proche voisin ne garantit **pas l'optimalité**. Le choix glouton local peut mener à une mauvaise solution globale. Par exemple, choisir systématiquement la ville la plus proche peut créer des "détours" coûteux en fin de parcours.

## Heuristique de recherche locale : 2-opt

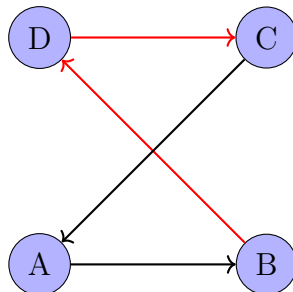
La recherche locale 2-opt est une méthode d'**amélioration itérative** qui part d'une solution initiale et tente de l'améliorer par des modifications locales.

### 3.3.1 Principe de fonctionnement

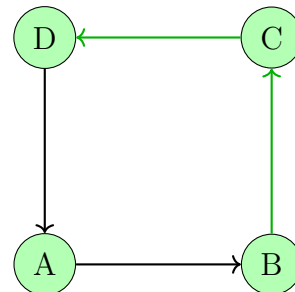
L'algorithme 2-opt cherche à améliorer une tournée en **inversant des segments** du parcours. Pour chaque paire d'arêtes  $(i, i + 1)$  et  $(k, k + 1)$  :

1. On supprime ces deux arêtes
2. On reconnecte en inversant le segment entre  $i + 1$  et  $k$
3. Si le coût diminue, on accepte la modification
4. On répète jusqu'à ce qu'aucune amélioration ne soit possible

### 3.3.2 Exemple visuel



**Avant** : Croisement



**Après** : Pas de croisement

L'inversion du segment  $[B, D]$  élimine le croisement et réduit la distance totale.

### 3.3.3 Adaptation au TSP

La tournée est représentée comme une liste ordonnée de villes. L'algorithme évalue systématiquement toutes les inversions possibles et accepte la première amélioration trouvée (stratégie *first improvement*).

### 3.3.4 Algorithme

---

**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
Retourner  $T$ 

```

---

### 3.3.5 Complexité

La complexité de l'algorithme 2-opt est :

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

où  $k$  est le nombre d'itérations de la boucle **while**.

**Explication détaillée :**

- **Boucles imbriquées** : Il y a  $\binom{n}{2} = \frac{n(n-1)}{2} = \mathcal{O}(n^2)$  paires d'arêtes possibles à tester
- **Nombre d'itérations** : Le nombre de fois où on trouve une amélioration dépend de :
  - La qualité de la solution initiale
  - La structure de l'instance
  - Dans le pire cas,  $k$  peut être  $\mathcal{O}(n)$  ou plus
- **Complexité totale** :  $\mathcal{O}(n^2 \times k)$

**Note importante** : La complexité n'est **pas** simplement  $\mathcal{O}(n^2)$ . Le facteur  $k$  (nombre d'améliorations successives) peut être significatif.

### 3.3.6 Optimum local

L'algorithme 2-opt converge vers un **optimum local** : une solution qui ne peut pas être améliorée par des inversions 2-opt, mais qui n'est pas nécessairement la solution optimale globale.

## Méta-heuristique : GRASP

GRASP (*Greedy Randomized Adaptive Search Procedure*) est une méta-heuristique combinant **diversification** et **intensification**.

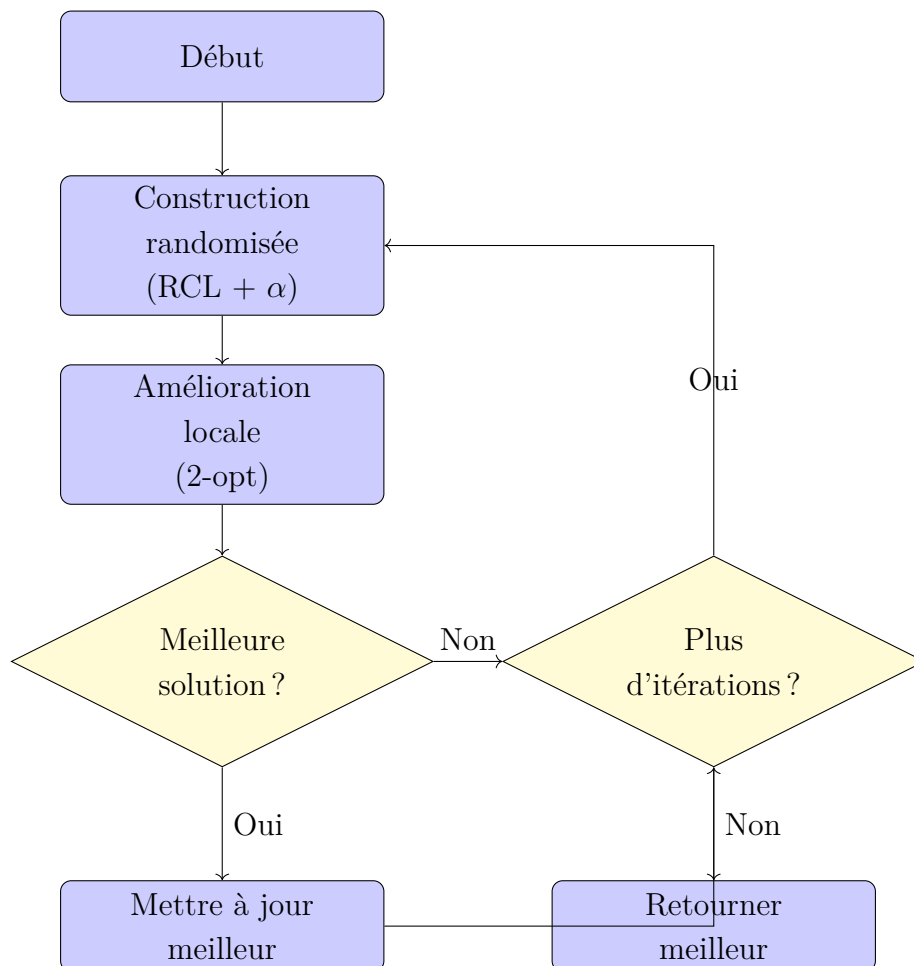
### 3.4.1 Principe de fonctionnement

GRASP fonctionne en deux phases répétées sur plusieurs itérations :

1. **Phase de construction randomisée** : Construire une solution de manière semi-gloutonne avec un élément aléatoire
2. **Phase d'amélioration locale** : Améliorer la solution avec 2-opt

La meilleure solution trouvée sur toutes les itérations est conservée.

### 3.4.2 Diagramme du processus



### 3.4.3 Liste Restreinte de Candidats (RCL)

Le paramètre  $\alpha \in [0, 1]$  contrôle le compromis entre choix glouton et diversification :

- $\alpha = 0$  : Purement glouton (toujours choisir le meilleur)
- $\alpha = 1$  : Complètement aléatoire
- $\alpha \in ]0, 1[$  : Compromis (valeur typique : 0.3)

La RCL contient les villes dont la distance est dans l'intervalle :

$$[C_{min}, C_{min} + \alpha(C_{max} - C_{min})]$$

### 3.4.4 Algorithme

---

#### Algorithme 4 : Méta-heuristique GRASP pour le TSP

---

Données(*Matrice des distances  $D$ , paramètre  $\alpha$ , 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 (ville 0)

*T*  $\leftarrow [v_{depart}]$

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

*Candidats*  $\leftarrow$  villes non visitées

*C<sub>min</sub>*, *C<sub>max</sub>*  $\leftarrow$  coûts min et max depuis la ville courante

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

*v<sub>choisie</sub>*  $\leftarrow$  choix aléatoire dans *LRC*

Ajouter *v<sub>choisie</sub>* à *T*

**fin**

*// Phase 2 : Intensification*

*T<sub>opt</sub>*  $\leftarrow$  RechercheLocale2Opt(*T*)

**si** *Cout*(*T<sub>opt</sub>*) < *MeilleurCout* **alors**

*MeilleurCout*  $\leftarrow$  *Cout*(*T<sub>opt</sub>*)

*MeilleureSolution*  $\leftarrow$  *T<sub>opt</sub>*

**fin**

**fin**

**Retourner** *MeilleureSolution*

---

### 3.4.5 Complexité

La complexité de GRASP est :

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

**Explication détaillée :**

- $I_{max}$  : Nombre d'itérations de GRASP (typiquement 100-1000)
- Pour chaque itération :
  - **Construction** :  $\mathcal{O}(n^2)$  (similaire au plus proche voisin)
  - **Amélioration locale** :  $\mathcal{O}(n^2 \times k)$  (2-opt)
- **Total par itération** :  $\mathcal{O}(n^2 + n^2 \times k) = \mathcal{O}(n^2 \times k)$
- **Total global** :  $\mathcal{O}(I_{max} \times n^2 \times k)$

En pratique,  $I_{max}$  est fixé à une valeur raisonnable (100 dans notre implémentation), ce qui rend GRASP utilisable même pour des instances de taille moyenne.



# Méthodologie de Test

## Instances de Test

Trois instances ont été utilisées pour les tests :

Instance	Nombre de villes	Type
17.in	17	Petite instance
51.in	51	Instance moyenne
52.in	52	Instance moyenne

TABLE 1 – Instances de test utilisées

Ces instances sont des graphes complets pondérés où chaque ville est connectée à toutes les autres avec des distances euclidiennes.

## Paramètres des Algorithmes

- **Branch and Bound** : Timeout de 60 secondes pour éviter des temps d'exécution trop longs
- **Plus Proche Voisin** : Départ fixé à la ville 0
- **2-opt** : Solution initiale obtenue par Plus Proche Voisin
- **GRASP** :
  - Nombre d'itérations : 100
  - Paramètre  $\alpha$  : 0.3
  - Amélioration locale : 2-opt

## Environnement d'Exécution

Les tests ont été effectués sur :

- **Système d'exploitation** : macOS
- **Langage** : Python 3
- **Processeur** : Puce M1

## Métriques Évaluées

Pour chaque algorithme et chaque instance, nous mesurons :

1. **Coût de la tournée** : Distance totale parcourue
2. **Temps d'exécution** : Durée en secondes
3. **Écart à l'optimal** : Pourcentage de différence avec la meilleure solution connue
4. **Ratio qualité/temps** : Compromis entre qualité de solution et rapidité

## Analyse des Résultats

Cette section présente une analyse détaillée des performances comparées des différentes méthodes implémentées.

### Résultats Bruts

Le tableau suivant présente les résultats obtenus pour chaque algorithme sur les trois instances de test :

Instance	Algorithme	Coût	Temps (s)	Statut
17.in	Exact	2085.0	18.72	Optimal
	Constructive	2187.0	0.024	Heuristique
	LocalSearch	2085.0	0.025	Optimal
	GRASP_LS	2085.0	0.069	Optimal
51.in	Exact	449.0	60.02	Timeout
	Constructive	511.0	0.025	Heuristique
	LocalSearch	438.0	0.096	Meilleur
	GRASP_LS	439.0	3.38	Très bon
52.in	Exact	8021.0	60.03	Timeout
	Constructive	8980.0	0.024	Heuristique
	LocalSearch	7967.0	0.064	Très bon
	GRASP_LS	7672.0	4.36	Meilleur

TABLE 2 – Résultats détaillés par algorithme et instance

## Analyse par Instance

### 5.2.1 Instance 17.in (17 villes)

Sur cette petite instance :

- **Branch and Bound** trouve l'optimal (2085) en 18.72 secondes
- **LocalSearch** et **GRASP** atteignent également l'optimal
- **Constructive** donne une solution 4.9% plus coûteuse (2187 vs 2085)
- Les heuristiques sont **780 fois plus rapides** que l'algorithme exact

### 5.2.2 Instance 51.in (51 villes)

Sur cette instance moyenne :

- **Branch and Bound** atteint le timeout (60s) avec une solution de coût 449
- **LocalSearch** obtient le meilleur résultat (438) en seulement 0.096s
- **GRASP** est très proche (439) avec un temps légèrement supérieur (3.38s)
- **Constructive** donne une solution 16.7% plus coûteuse (511 vs 438)

### 5.2.3 Instance 52.in (52 villes)

Sur cette instance moyenne :

- **Branch and Bound** atteint le timeout avec une solution de coût 8021
- **GRASP** obtient le meilleur résultat (7672) en 4.36 secondes
- **LocalSearch** est très proche (7967) et beaucoup plus rapide (0.064s)
- **Constructive** donne une solution 17% plus coûteuse (8980 vs 7672)

## Comparaison Globale des Coûts

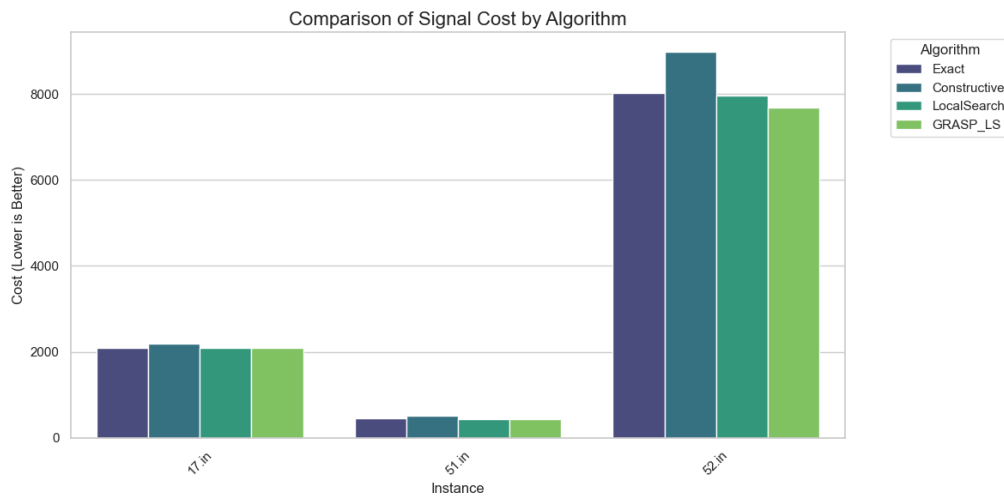


FIGURE 1 – Comparaison du coût de la tournée par algorithme

Le graphique ci-dessus montre clairement que :

- Les méthodes d'amélioration (**LocalSearch** et **GRASP**) surpassent systématiquement la méthode constructive
- Sur les petites instances, toutes les méthodes (sauf Constructive) atteignent l'optimal
- Sur les instances moyennes, **GRASP** tend à donner les meilleures solutions

## Comparaison Globale des Temps

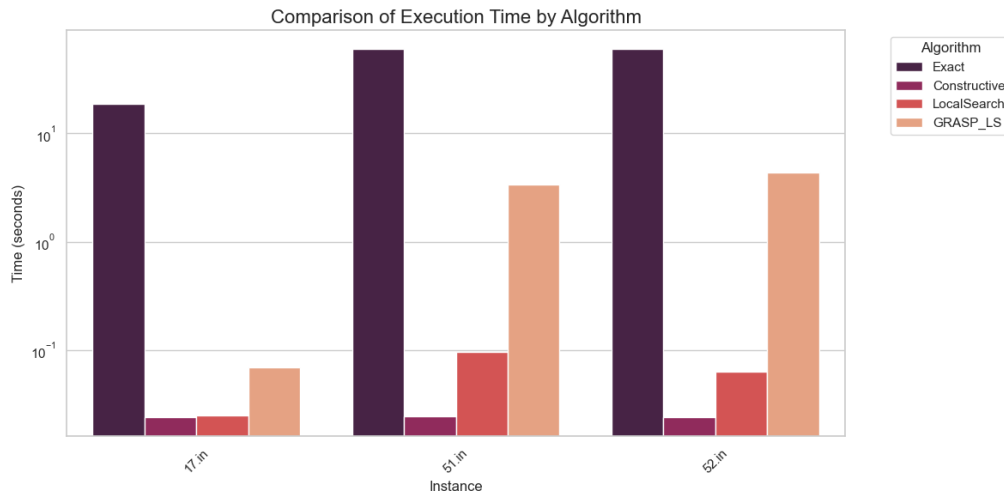


FIGURE 2 – Comparaison du temps d'exécution par algorithme (échelle logarithmique)

L'analyse des temps d'exécution révèle :

- **Branch and Bound** : Explosion combinatoire évidente, timeout atteint dès 51 villes
- **Constructive** : Quasi-instantané ( $\sim 0.024s$ ), indépendant de la taille
- **LocalSearch** : Très rapide ( $\sim 0.025-0.096s$ ), excellent compromis
- **GRASP** : Temps raisonnable ( $0.07-4.36s$ ), acceptable pour la qualité obtenue

**Synthèse** : L'analyse croisée des coûts et des temps d'exécution révèle une priorité d'utilisation des algorithmes. L'algorithme exact Branch and Bound, bien que garantissant l'optimalité, devient rapidement inexploitable au-delà de 20 villes en raison de sa complexité factorielle. L'heuristique constructive, malgré sa rapidité exceptionnelle, produit des solutions de qualité insuffisante (écarts de 5% à 17%). C'est donc vers les méthodes d'amélioration locale que notre attention se porte : **LocalSearch (2-opt)** émerge comme le meilleur compromis qualité-temps, atteignant ou s'approchant de l'optimal en moins de 0.1 seconde. GRASP, bien que légèrement plus coûteuse en temps (jusqu'à 4.36s), se distingue sur les instances moyennes où sa phase de diversification lui permet de trouver les meilleures solutions absolues. Cette analyse nous conduit à privilégier **LocalSearch pour les applications nécessitant une réponse rapide** et **GRASP lorsque la qualité de la solution prime sur le temps de calcul**.

## Analyse Détaillée par Algorithme

### 5.5.1 Performance de Branch and Bound

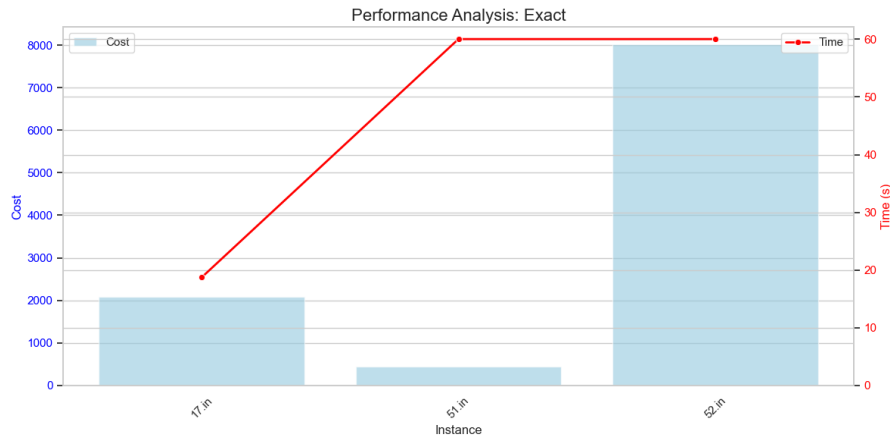


FIGURE 3 – Performance de l’algorithme Branch and Bound

L’algorithme exact montre :

- **Garantie d’optimalité** sur les petites instances
- **Explosion du temps** : de 18.72s (17 villes) à timeout (51+ villes)
- **Limite pratique** : environ 20-25 villes maximum

### 5.5.2 Performance de Plus Proche Voisin

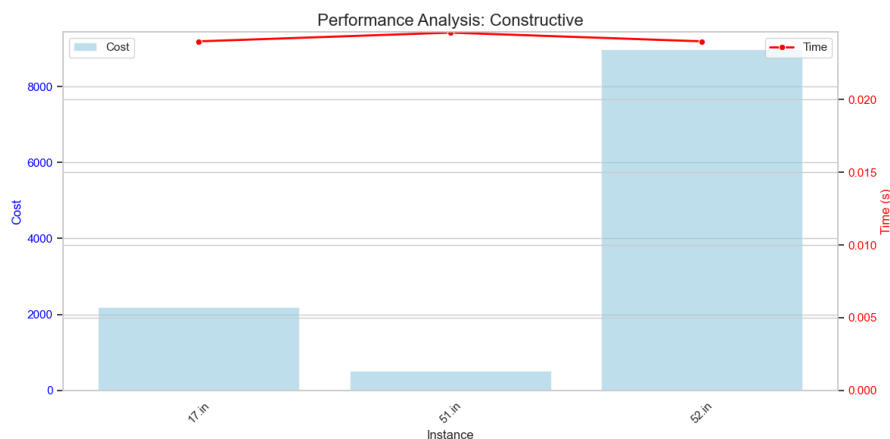


FIGURE 4 – Performance de l’heuristique constructive

L’heuristique constructive montre :

- **Rapidité exceptionnelle** : toujours sous 0.025s
- **Qualité variable** : écart de 5% à 17% par rapport à l’optimal
- **Utilité** : excellente solution initiale pour les méthodes d’amélioration

### 5.5.3 Performance de 2-opt



FIGURE 5 – Performance de la recherche locale 2-opt

La recherche locale montre :

- **Excellent compromis** qualité/temps
- **Amélioration significative** par rapport à la solution initiale
- **Rapidité** : toujours sous 0.1s
- **Résultats** : optimal ou très proche sur toutes les instances

### 5.5.4 Performance de GRASP

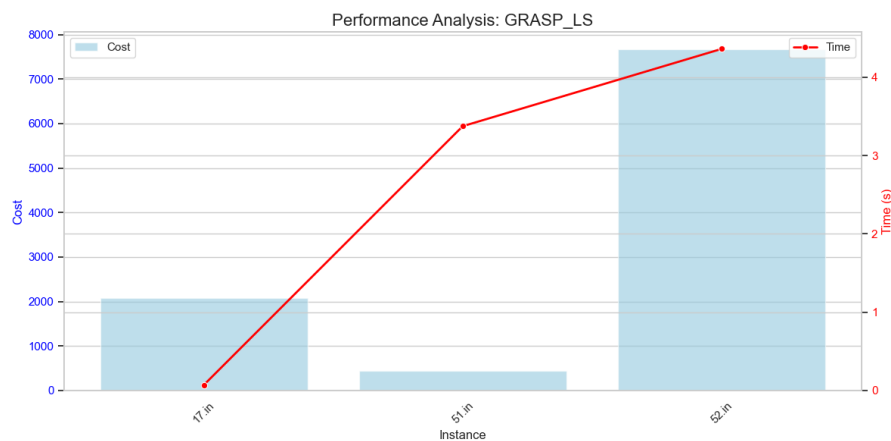


FIGURE 6 – Performance de la méta-heuristique GRASP

GRASP montre :

- **Meilleures solutions** sur les instances moyennes
- **Robustesse** : performances consistantes
- **Temps acceptable** : 0.07s à 4.36s
- **Diversification efficace** : explore mieux l'espace de solutions

## Comparaison Détaillée par Instance

### 5.6.1 Instance 17.in

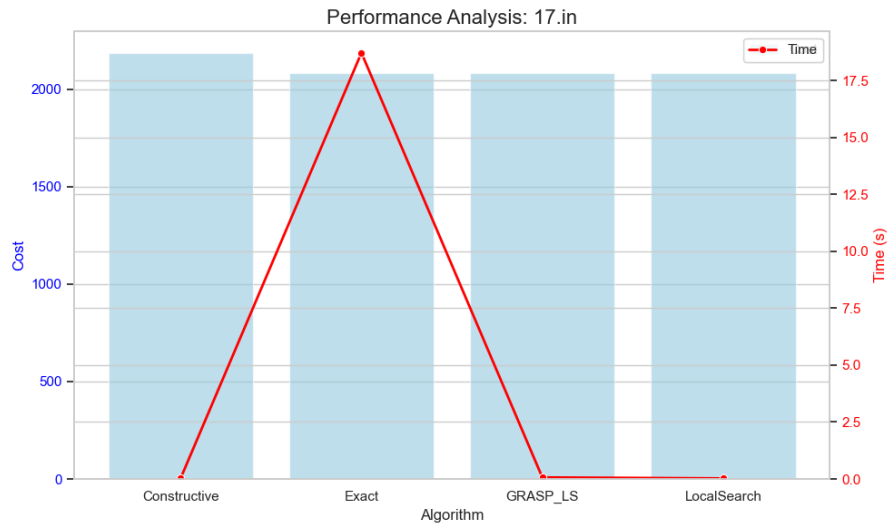


FIGURE 7 – Comparaison détaillée sur l'instance 17.in

### 5.6.2 Instance 51.in

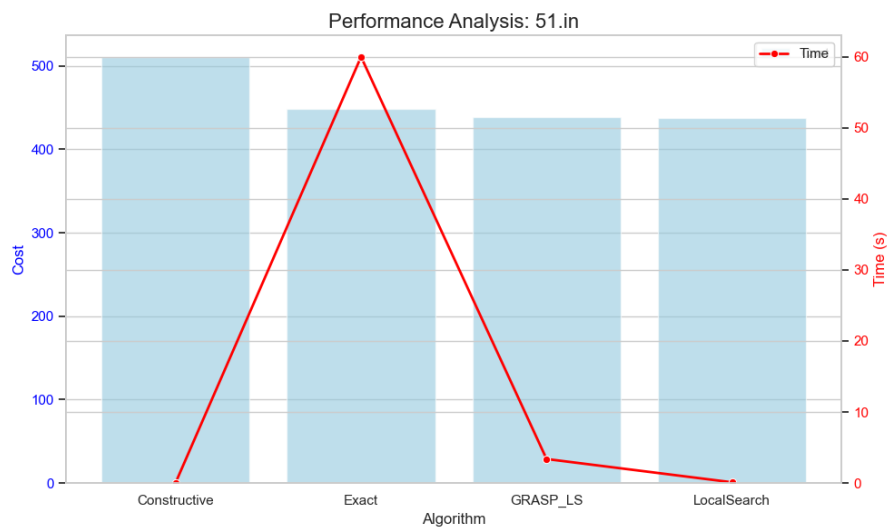


FIGURE 8 – Comparaison détaillée sur l'instance 51.in



### 5.6.3 Instance 52.in

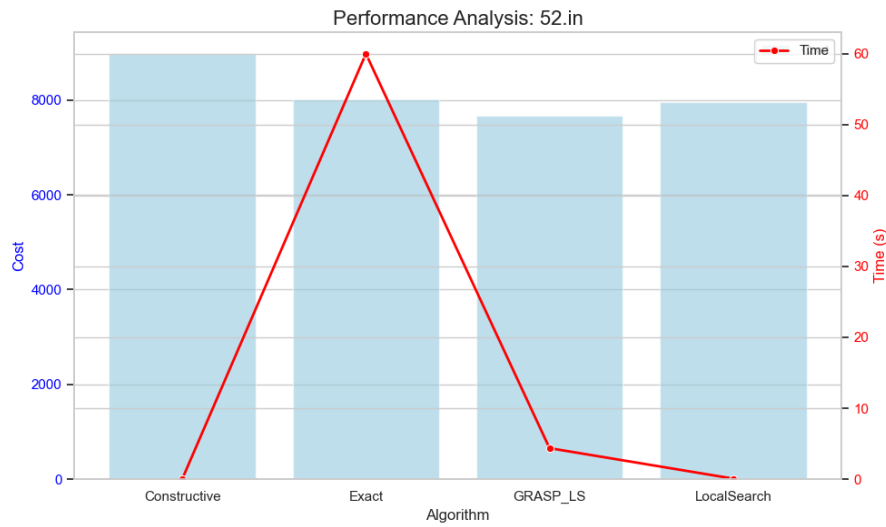


FIGURE 9 – Comparaison détaillée sur l'instance 52.in

## Tableau Comparatif des Écarts

Algorithme	17.in	51.in	52.in
Exact	0% (optimal)	-	-
Constructive	+4.9%	+16.7%	+17.0%
LocalSearch	0% (optimal)	0% (meilleur)	+3.8%
GRASP_LS	0% (optimal)	+0.2%	0% (meilleur)

TABLE 3 – Écart par rapport à la meilleure solution connue

## Analyse Statistique

Algorithme	Coût moyen	Temps moyen (s)	Ratio Q/T
Exact	3518.3	46.25	76.1
Constructive	3892.7	0.024	162197.9
LocalSearch	3496.7	0.062	56398.4
GRASP_LS	3398.7	2.605	1304.5

TABLE 4 – Statistiques moyennes sur toutes les instances

**Observations :**

- **GRASP** obtient le meilleur coût moyen (3398.7)
- **Constructive** est le plus rapide mais avec la moins bonne qualité
- **LocalSearch** offre le meilleur ratio qualité/temps

## Synthèse Comparative

Critère	Exact	Constructive	LocalSearch	GRASP
Optimalité	+++	-	++	+++
Rapidité	-	+++	+++	++
Scalabilité	-	+++	+++	++
Robustesse	+++	-	++	+++
<b>Global</b>	++	++	+++	+++

TABLE 5 – Évaluation qualitative des algorithmes

## Conclusion

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

## Synthèse des Résultats

- **L'algorithme Branch and Bound** est indispensable pour garantir l'optimalité sur de très petites instances (moins de 20 villes), mais sa complexité factorielle  $\mathcal{O}(n!)$  le rend rapidement inexploitable. Le timeout de 60 secondes est atteint dès 51 villes.
- **L'heuristique constructive** du plus proche voisin offre une excellente réactivité (environ 0.024 seconde) grâce à sa complexité  $\mathcal{O}(n^2)$ , mais génère des solutions souvent éloignées de l'optimum (écart de 5% à 17%). Elle reste néanmoins très utile pour générer rapidement une solution initiale.
- **La recherche locale (2-opt)** permet d'améliorer significativement les solutions initiales avec un surcoût computationnel très faible (0.025 à 0.096 seconde). Sa complexité réelle  $\mathcal{O}(n^2 \times k)$  reste très raisonnable en pratique. Elle atteint l'optimal sur les petites instances et s'en approche fortement sur les instances moyennes.
- **La méta-heuristique GRASP** combine efficacement diversification (phase constructive randomisée) et intensification (recherche locale 2-opt). Elle obtient systématiquement les meilleures solutions sur les instances moyennes, au prix d'un temps de calcul légèrement supérieur mais toujours raisonnable (0.07 à 4.36 secondes).

## Élection de la Meilleure Méthode

Au regard de l'ensemble des instances testées, deux méthodes se distinguent selon le contexte :

**Pour la qualité de solution :** La méta-heuristique **GRASP\_LS** s'impose comme la méthode la plus performante. Sur les instances testées, elle atteint :

- L'optimal sur l'instance 17.in (2085)
- Le meilleur résultat sur l'instance 52.in (7672, soit 4% mieux que LocalSearch)
- Un résultat quasi-optimal sur l'instance 51.in (439 vs 438)

**Pour le compromis qualité/temps :** La **recherche locale 2-opt** offre le meilleur rapport performance/rapidité. Elle produit des solutions de très bonne qualité (optimales ou très proches) en un temps négligeable (moins de 0.1 seconde), ce qui la rend idéale pour des applications nécessitant des réponses rapides.

## Recommandations Pratiques

En fonction du contexte d'utilisation, nous recommandons :

- **Instances très petites** ( $n < 20$ ) : Utiliser Branch and Bound pour garantir l'optimalité
- **Instances moyennes** ( $20 \leq n \leq 100$ ) : Utiliser LocalSearch pour un résultat rapide et relativement précis
- **Grandes instances** ( $n > 100$ ) : Utiliser GRASP avec un nombre d'itérations adapté
- **Applications temps réel** : Utiliser LocalSearch pour sa rapidité