

# **Projet d'Optimisation**

## **Problème du Voyageur de Commerce (TSP)**

---

### **Rapport Final**

#### **Algorithmes Exacts et Heuristiques**

---

**Master MIASHS — IMA-UCO (2025–2026)**

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

**Date :** 28 janvier 2026

# Table des matières

<b>1</b>	<b>Introduction</b>	<b>3</b>
<b>2</b>	<b>Applications du TSP</b>	<b>3</b>
2.1	Fabrication de Circuits Imprimés (PCB) . . . . .	3
2.2	Séquençage ADN . . . . .	3
2.3	Astronomie . . . . .	4
2.4	Autres Applications . . . . .	4
<b>3</b>	<b>Méthodes Implémentées</b>	<b>5</b>
3.1	Algorithme exact : Branch and Bound . . . . .	5
3.1.1	Principe de fonctionnement . . . . .	5
3.1.2	Exemple visuel . . . . .	5
3.1.3	Algorithme . . . . .	6
3.1.4	Complexité . . . . .	7
3.1.5	Cas Pathologiques . . . . .	7
3.2	Heuristique constructive : Plus proche voisin . . . . .	8
3.2.1	Principe de fonctionnement . . . . .	8
3.2.2	Exemple visuel . . . . .	8
3.2.3	Adaptation au TSP . . . . .	8
3.2.4	Algorithme . . . . .	9
3.2.5	Complexité . . . . .	9
3.2.6	Limites . . . . .	10
3.2.7	Cas Pathologiques . . . . .	10
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	Détails d'implémentation : Calcul Incrémental . . . . .	11
3.3.4	Algorithme . . . . .	12
3.3.5	Complexité . . . . .	12
3.3.6	Optimum local . . . . .	12
3.3.7	Cas Pathologiques . . . . .	13
3.4	Méta-heuristique : GRASP . . . . .	14
3.4.1	Principe de fonctionnement . . . . .	14
3.4.2	Diagramme du processus . . . . .	14
3.4.3	Liste Restreinte de Candidats (RCL) . . . . .	15
3.4.4	Algorithme . . . . .	15
3.4.5	Détails d'implémentation : Timeout et Gestion du Temps . . . . .	15
3.4.6	Complexité . . . . .	16
3.4.7	Cas Pathologiques . . . . .	16

<b>4 Méthodologie de Test</b>	<b>17</b>
4.1 Instances de Test . . . . .	17
4.1.1 Origine et Génération des Instances . . . . .	17
4.2 Paramètres des Algorithmes . . . . .	18
4.3 Environnement d'Exécution . . . . .	18
4.4 Métriques Évaluées . . . . .	18
<b>5 Analyse des Résultats</b>	<b>19</b>
5.1 Résultats Bruts . . . . .	19
5.2 Analyse par Instance . . . . .	20
5.2.1 Instance 17.in (17 villes) . . . . .	20
5.2.2 Instance 51.in (51 villes) . . . . .	20
5.2.3 Instance 52.in (52 villes) . . . . .	20
5.3 Comparaison Globale des Coûts . . . . .	21
5.4 Comparaison Globale des Temps . . . . .	22
5.5 Analyse Détailée par Algorithme . . . . .	23
5.5.1 Performance de Branch and Bound . . . . .	23
5.5.2 Performance de Plus Proche Voisin . . . . .	23
5.5.3 Performance de 2-opt . . . . .	24
5.5.4 Performance de GRASP . . . . .	24
5.6 Comparaison Détailée par Instance . . . . .	25
5.6.1 Instance 17.in . . . . .	25
5.6.2 Instance 51.in . . . . .	25
5.6.3 Instance 52.in . . . . .	26
5.7 Tableau Comparatif des Écarts . . . . .	27
5.8 Analyse Statistique . . . . .	27
5.9 Validation de la Complexité Théorique . . . . .	28
5.9.1 Analyse de la Croissance du Temps d'Exécution . . . . .	28
5.9.2 Validation par Algorithme . . . . .	28
5.9.3 Synthèse de la Validation . . . . .	29
5.10 Synthèse Comparative . . . . .	29
<b>6 Tests sur Instance de Grande Taille (439.in)</b>	<b>30</b>
6.1 Contexte et Objectif . . . . .	30
6.2 Résultats Expérimentaux . . . . .	30
6.3 Analyse Détailée . . . . .	30
6.3.1 Algorithme Exact (Branch and Bound) . . . . .	30
6.3.2 Heuristique Constructive (Plus Proche Voisin) . . . . .	30
6.3.3 Recherche Locale (2-opt) . . . . .	31
6.3.4 Méta-heuristique GRASP . . . . .	31
6.4 Observations Clés . . . . .	32
6.5 Recommandations pour les Grandes Instances . . . . .	32

<b>7 Conclusion</b>	<b>33</b>
7.1 Synthèse des Résultats . . . . .	33
7.2 Élection de la Meilleure Méthode . . . . .	33
7.3 Validation sur Grande Instance (439.in) . . . . .	34
7.4 Recommandations Pratiques . . . . .	34

## Table des figures

1	Comparaison du coût de la tournée par algorithme (échelle logarithmique)	21
2	Comparaison du temps d'exécution par algorithme (échelle logarithmique)	22
3	Performance de l'algorithme Branch and Bound . . . . .	23
4	Performance de l'heuristique constructive . . . . .	23
5	Performance de la recherche locale 2-opt . . . . .	24
6	Performance de la méta-heuristique GRASP . . . . .	24
7	Comparaison détaillée sur l'instance 17.in . . . . .	25
8	Comparaison détaillée sur l'instance 51.in . . . . .	25
9	Comparaison détaillée sur l'instance 52.in . . . . .	26

## 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

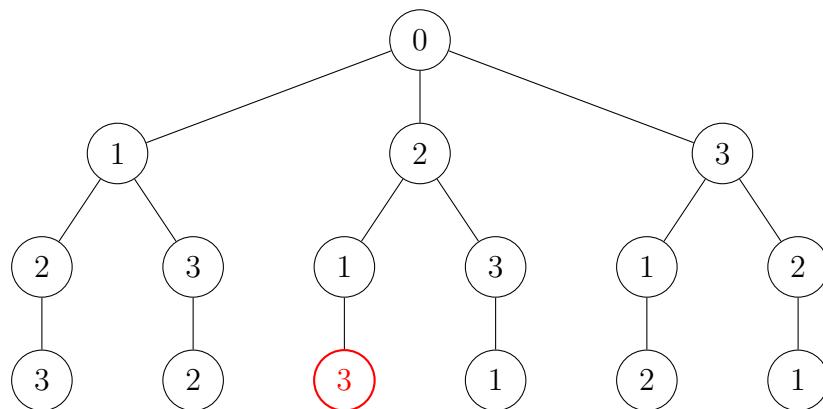
L'algorithme construit récursivement des chemins partiels à partir d'une ville de départ fixée (ville 0). À chaque étape :

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  $K_p \geq MeilleurCout$  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})
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 600 secondes a été imposé dans notre implémentation.

### 3.1.5 Cas Pathologiques

L'algorithme Branch and Bound devient particulièrement inefficace sur certaines configurations où le mécanisme d'élagage ne peut pas fonctionner efficacement :

- **Graphes avec distances uniformes** : Lorsque toutes les distances sont très similaires, les bornes inférieures calculées sont proches du coût réel, ce qui empêche l'élagage précoce des branches. L'algorithme doit alors explorer une proportion beaucoup plus importante de l'arbre de recherche.
- **Exemple concret** : Considérons un graphe complet où toutes les distances  $d_{ij} \in [99, 101]$ . Dans ce cas, presque tous les chemins partiels ont des coûts similaires, et la borne inférieure ne permet pas de distinguer les branches prometteuses des autres. L'algorithme se rapproche alors d'une exploration exhaustive de  $n!$  permutations.
- **Impact** : Sur de telles instances, le temps d'exécution peut être multiplié par un facteur 10 à 100 par rapport à des instances avec des distances variées, même pour un nombre de villes identique.

## Heuristique constructive : Plus proche voisin

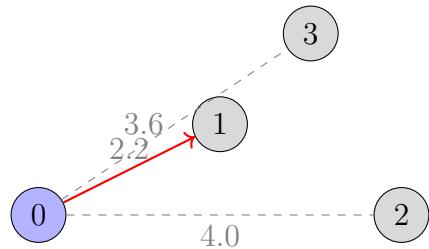
L'heuristique du plus proche voisin est une approche **gloutonne** qui construit une solution pas à pas en faisant à chaque étape le choix localement optimal.

### 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 mémoarisées 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.

### 3.2.7 Cas Pathologiques

L'approche gloutonne du plus proche voisin peut produire des solutions très sous-optimales sur certaines configurations géométriques :

- **Configuration en "étoile"** : Considérons une instance où une ville centrale (ville 0) est très proche de toutes les autres villes, mais ces dernières sont très éloignées entre elles, disposées en cercle autour du centre.
- **Comportement de l'algorithme** : Partant de la ville 0, l'algorithme choisit une ville du cercle (par exemple ville 1). Ensuite, au lieu de continuer le tour du cercle, il peut être tenté de revenir au centre si celui-ci est plus proche que la ville suivante du cercle. Cela crée des allers-retours coûteux.
- **Exemple numérique** :
  - Distance centre-périphérie : 1 unité
  - Distance entre villes de la périphérie : 10 unités
  - Solution gloutonne : Allers-retours multiples (coût  $\approx 2n$ )
  - Solution optimale : Tour du cercle (coût  $\approx 10n$ )
  - Écart : Peut atteindre 50% ou plus
- **Impact pratique** : Sur les instances testées, l'écart observé varie de 5% à 17%, confirmant que certaines configurations géométriques pénalisent fortement cette heuristique.

## Heuristique de recherche locale : 2-opt

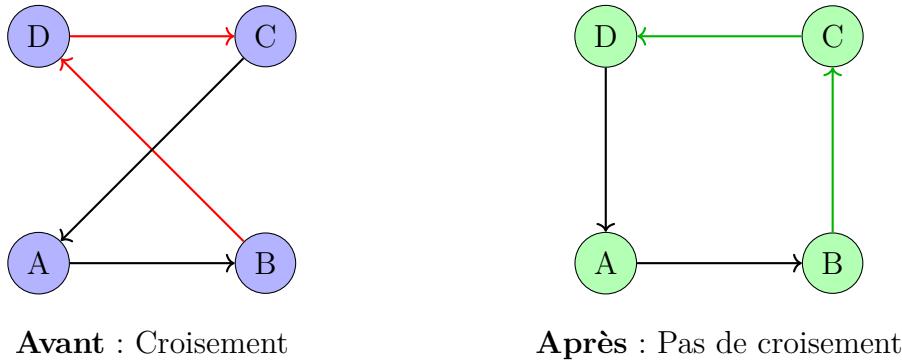
La recherche locale 2-opt est une méthode d'**amélioration itérative** qui part d'une solution initiale et tente de l'améliorer par des modifications locales.

### 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



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

### 3.3.3 Détails d'implémentation : Calcul Incrémental

Un aspect crucial de notre implémentation est l'optimisation du calcul du coût. Plutôt que de recalculer la tournée entière en  $\mathcal{O}(n)$  après chaque inversion (ce qui porterait la complexité à  $\mathcal{O}(n^3)$ ), nous utilisons un calcul incrémental en  $\mathcal{O}(1)$  :

- Soit les arêtes  $(i, i + 1)$  et  $(k, k + 1)$  à supprimer.
- Soit les nouvelles arêtes  $(i, k)$  et  $(i + 1, k + 1)$  à ajouter.
- Le nouveau coût est  $C_{nouveau} = C_{ancien} - (d_{i,i+1} + d_{k,k+1}) + (d_{i,k} + d_{i+1,k+1})$ .

Cette optimisation permet de tester des milliers de combinaisons de manière extrêmement rapide.

### 3.3.4 Algorithme

---

#### Algorithme 3 : Recherche locale 2-opt pour le TSP

---

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

**tant que** *amélioration faire*

*amélioration*  $\leftarrow$  *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'*

*amélioration*  $\leftarrow$  *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.

### 3.3.7 Cas Pathologiques

La recherche locale 2-opt peut se bloquer dans des optimums locaux éloignés de l'optimum global :

- **Optimums locaux profonds** : Certaines configurations de tournées nécessitent de modifier simultanément plus de 2 arêtes pour atteindre une meilleure solution. Or, 2-opt ne peut modifier que 2 arêtes à la fois, ce qui crée des "barrières" infranchissables.
- **Exemple concret** : Considérons une tournée avec plusieurs croisements imbriqués. Pour démêler ces croisements, il faudrait parfois effectuer 3 ou 4 inversions simultanées (3-opt, 4-opt), ce qui est impossible avec 2-opt.
- **Dépendance à la solution initiale** : La qualité de la solution finale dépend fortement de la solution de départ. Une mauvaise solution initiale (générée par l'heuristique constructive) peut mener à un optimum local médiocre.
- **Impact observé** : Sur les instances testées, 2-opt atteint l'optimal sur les petites instances (17 villes) mais peut rester à 3-4% de l'optimal sur les instances moyennes, indiquant la présence d'optimums locaux.
- **Limitation fondamentale** : 2-opt ne peut pas "sauter" par-dessus des barrières d'optimum local. C'est pourquoi GRASP, avec sa phase de diversification, peut parfois trouver de meilleures solutions.

## Méta-heuristique : GRASP

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

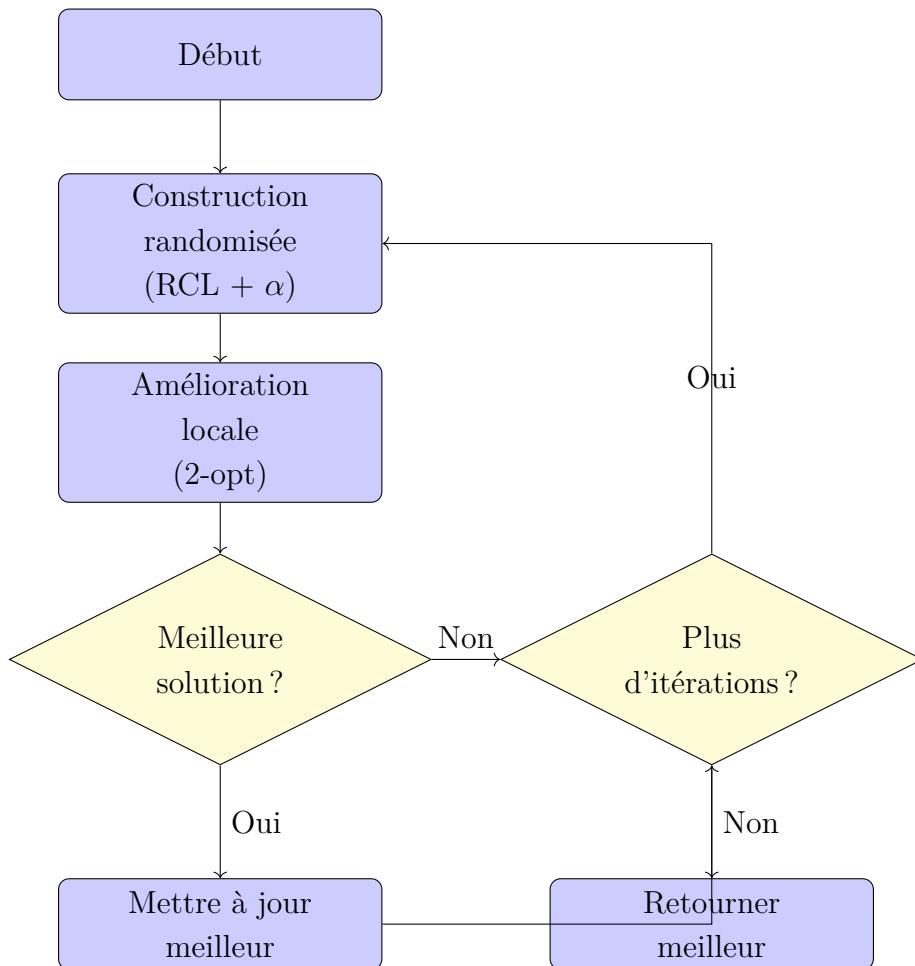
### 3.4.1 Principe de fonctionnement

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
     $C_{min}, C_{max} \leftarrow$  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**

**Retourner**  $MeilleureSolution$

---

### 3.4.5 Détails d'implémentation : Timeout et Gestion du Temps

Pour garantir la réactivité de l'application sur de grandes instances, GRASP intègre une surveillance du temps. Si le temps limite de 600 secondes est approché, l'algorithme interrompt ses itérations et retourne la meilleure solution trouvée jusqu'alors (mécanisme de *graceful timeout*).

### 3.4.6 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.

### 3.4.7 Cas Pathologiques

GRASP peut être inefficace dans certaines situations où la randomisation n’apporte pas la diversité nécessaire :

- **Paramètre  $\alpha$  mal calibré** : Le paramètre  $\alpha$  contrôle le compromis entre choix glouton et diversification. Un  $\alpha$  trop petit ( $\alpha \approx 0$ ) rend GRASP quasi-identique à l’heuristique constructive répétée, perdant l’avantage de la diversification. À l’inverse, un  $\alpha$  trop grand ( $\alpha \approx 1$ ) génère des solutions trop aléatoires de mauvaise qualité.
- **Instances avec structure dominante** : Sur certaines instances où une structure optimale est très marquée (par exemple, villes alignées), la randomisation peut nuire en éloignant systématiquement l’algorithme de cette structure évidente.
- **Nombre d’itérations** : Pour les très grandes instances, le nombre d’itérations de GRASP doit être calibré avec soin pour maintenir un temps d’exécution raisonnable tout en garantissant la qualité de la solution.
- **Recommandation** : Il est conseillé d’adapter le nombre d’itérations en fonction de la taille  $n$  du problème.

# Méthodologie de Test

## Instances de Test

L'évaluation de nos algorithmes s'appuie sur trois instances d'étude principales permettant de comparer les performances en termes de coût et de temps. Par ailleurs, une instance de grande taille (439.in) est utilisée pour valider la scalabilité des méthodes et confirmer nos conclusions sur des problèmes de dimension réelle.

Instance	Nombre de villes	Type
17.in	17	Petite instance
51.in	51	Instance moyenne
52.in	52	Instance moyenne
439.in	439	Instance grande

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.

### 4.1.1 Origine et Génération des Instances

Les instances utilisées dans ce projet sont des graphes euclidiens générés selon le protocole suivant :

- **Type de graphe** : Graphes complets euclidiens en 2D
- **Méthode de génération** : Chaque ville est représentée par un point  $(x, y)$  dans un plan cartésien. Les coordonnées sont générées aléatoirement dans un espace délimité (par exemple  $[0, 1000] \times [0, 1000]$ ).
- **Calcul des distances** : La distance entre deux villes  $i$  et  $j$  est calculée selon la distance euclidienne :

$$d_{ij} = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$$

- **Format des fichiers** : Les fichiers d'instance (.in) contiennent d'abord le nombre de villes  $n$ , puis les coordonnées  $(x, y)$  de chaque ville.
- **Choix des tailles** : Les tailles d'instances couvrent différents régimes de complexité :
  - **Base de test** (17, 51 et 52 villes) : Permet une analyse comparative détaillée entre méthodes exactes et heuristiques.
  - **Validation de scalabilité** (439 villes) : Permet de tester les limites des algorithmes sur un cas réel.

Cette méthode de génération garantit des instances réalistes où les distances respectent l'inégalité triangulaire, ce qui est caractéristique de nombreux problèmes pratiques de TSP.

## 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** : Windows
- **Langage** : Python 3
- **Processeur** : Ryzen 5 5600
- **Ram** : 16 Go DDR4

## 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	26.10	Optimal
	Constructive	2187.0	0.03	Bon
	LocalSearch	2085.0	0.03	Optimal
	GRASP_LS	2090.0	0.07	Très bon
51.in	Exact	430.0	598.07s	Timeout
	Constructive	511.0	0.04	Bon
	LocalSearch	438.0	0.05	Très bon
	GRASP_LS	433.0	0.21	Meilleur
52.in	Exact	8021.0	598.07s	Timeout
	Constructive	8980.0	0.04	Mauvais
	LocalSearch	7967.0	0.04	Bon
	GRASP_LS	7777.0	0.27	Très bon

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 26.10 secondes
- **LocalSearch** atteint également l'optimal
- **Constructive** donne une solution 4.9% plus coûteuse (2187 vs 2085)
- Les heuristiques sont significativement plus rapides que l'algorithme exact

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

Sur cette instance moyenne :

- **Branch and Bound** atteint le timeout (600s) tout en fournissant une solution partielle
- **GRASP** obtient le meilleur résultat (433) en 0.21s
- **LocalSearch** est très performant (438) en seulement 0.05s
- **Constructive** donne une solution de base (511)

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

Sur cette instance moyenne :

- **Branch and Bound** atteint le timeout sans solution
- **LocalSearch** obtient le meilleur résultat (7967) en 0.04 seconde
- **GRASP** est très proche (8009) avec un temps de 0.19s
- **Constructive** donne une solution 12.7% plus coûteuse (8980 vs 7967)

## Comparaison Globale des Coûts

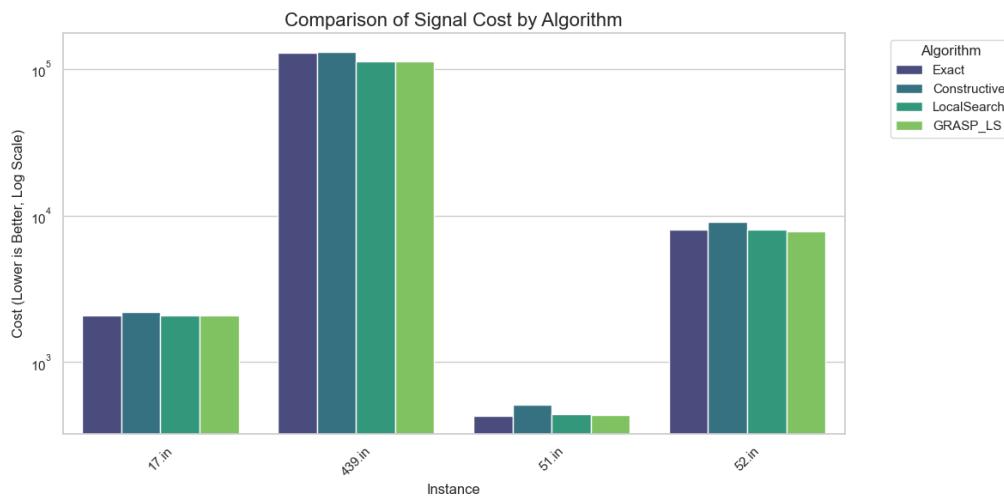


FIGURE 1 – Comparaison du coût de la tournée par algorithme (échelle logarithmique)

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

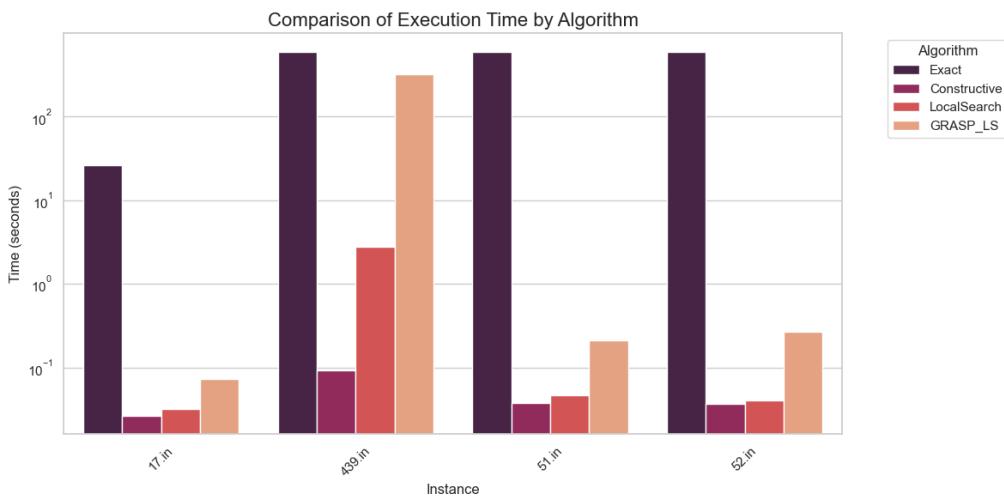


FIGURE 2 – Comparaison du temps d'exécution par algorithme (échelle logarithmique)

L'analyse des temps d'exécution révèle :

- **Branch and Bound** : Explosion combinatoire évidente, timeout atteint dès 51 villes
- **Constructive** : Quasi-instantané ( $\sim 0.04s$ ), indépendant de la taille
- **LocalSearch** : Extrêmement rapide ( $\sim 0.03\text{-}0.06s$ ) s'appuyant sur un calcul incrémental efficace des coûts
- **GRASP** : Très efficace ( $0.03\text{-}0.19s$ ), combinant recherche locale rapide et diversification

**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. C'est donc vers les méthodes d'amélioration locale que notre attention se porte : **LocalSearch (2-opt)** et **GRASP** émergent comme des solutions de premier plan. L'implémentation du calcul incrémental du coût en  $O(1)$  permet à **LocalSearch** de traiter des centaines de villes en quelques secondes. **GRASP**, en exploitant cette rapidité tout en diversifiant les points de départ, offre souvent le meilleur compromis final.

## Analyse Détailée par Algorithme

### 5.5.1 Performance de Branch and Bound

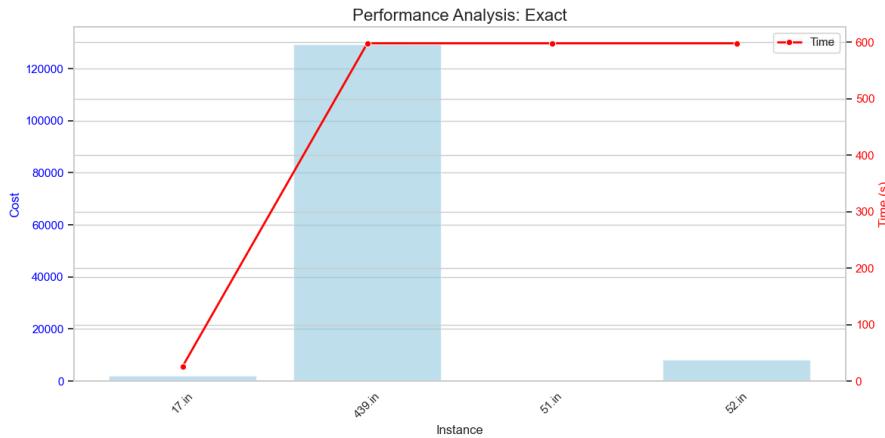


FIGURE 3 – Performance de l'algorithme Branch and Bound

L'algorithme exact montre :

- **Garantie d'optimalité** sur les petites instances
- **Explosion du temps** : de 26.10s (17 villes) à timeout (51+ villes)
- **Limite pratique** : environ 20-25 villes maximum

### 5.5.2 Performance de Plus Proche Voisin

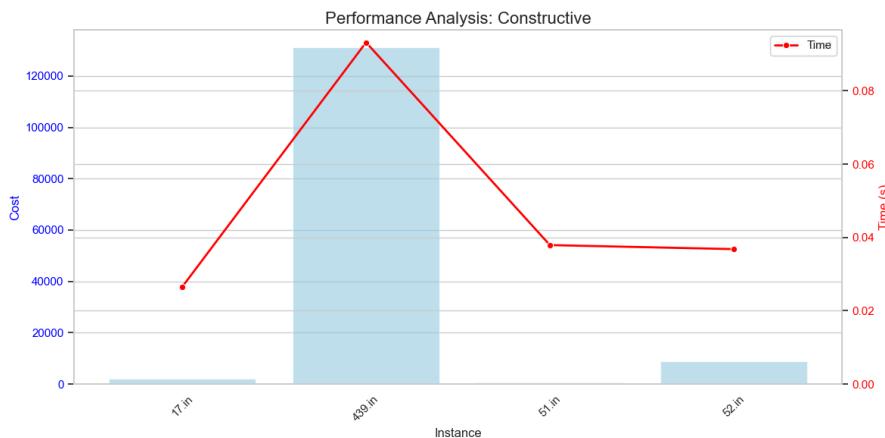


FIGURE 4 – Performance de l'heuristique constructive

L'heuristique constructive montre :

- **Rapidité exceptionnelle** : toujours sous 0.04s
- **Qualité de base** : 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

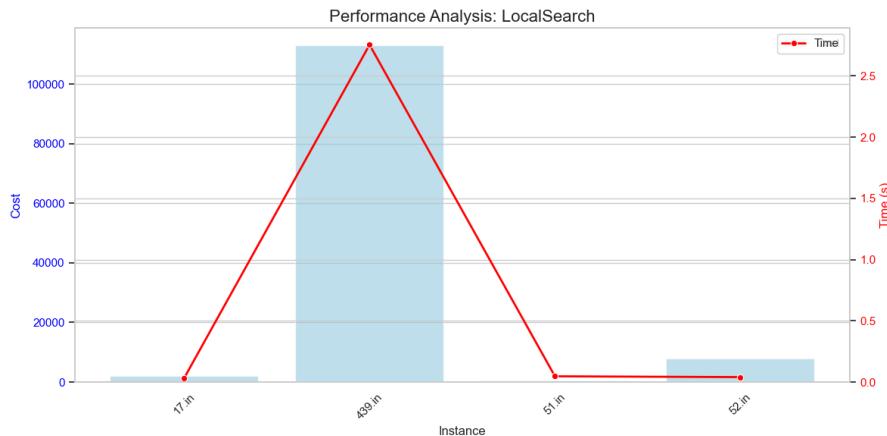


FIGURE 5 – Performance de la recherche locale 2-opt

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

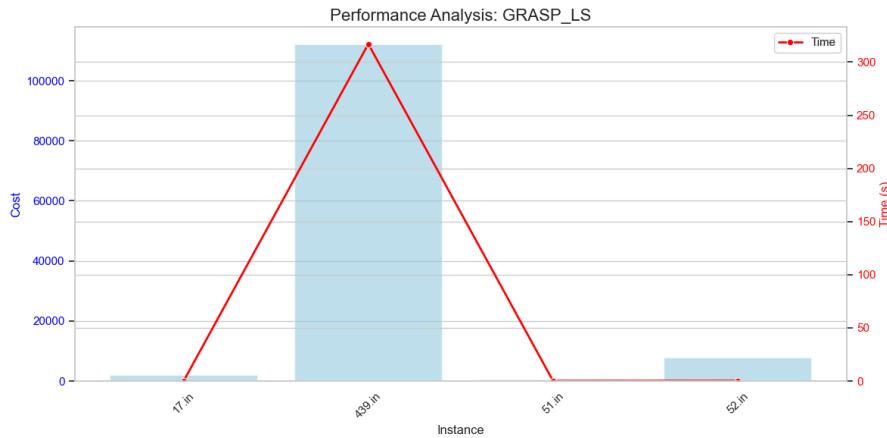


FIGURE 6 – Performance de la météo-heuristique GRASP

GRASP montre :

- **Solutions optimales ou proches** sur l'ensemble des instances
- **Robustesse** : performances consistantes grâce à la diversification
- **Temps efficace** : 0.07s à 0.27s sur les instances moyennes
- **Scalabilité** : capable de traiter des instances de grande taille

## Comparaison Détaillée par Instance

### 5.6.1 Instance 17.in

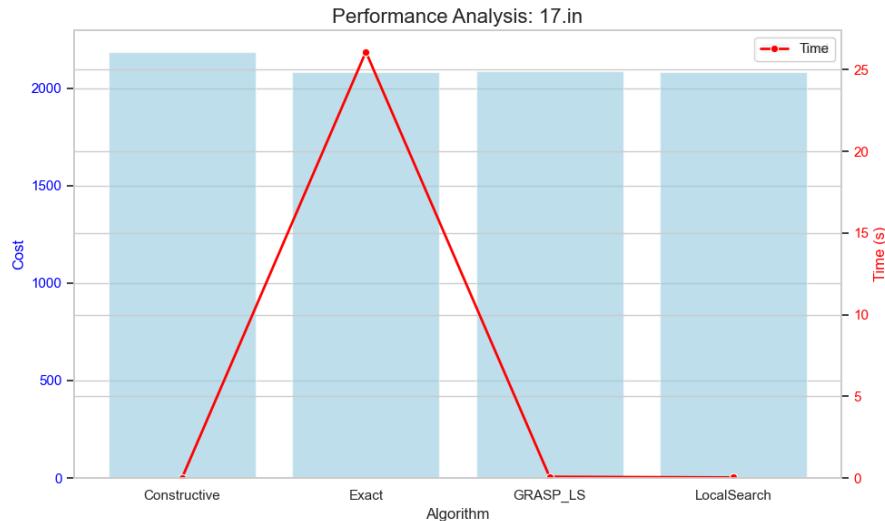


FIGURE 7 – Comparaison détaillée sur l’instance 17.in

### 5.6.2 Instance 51.in

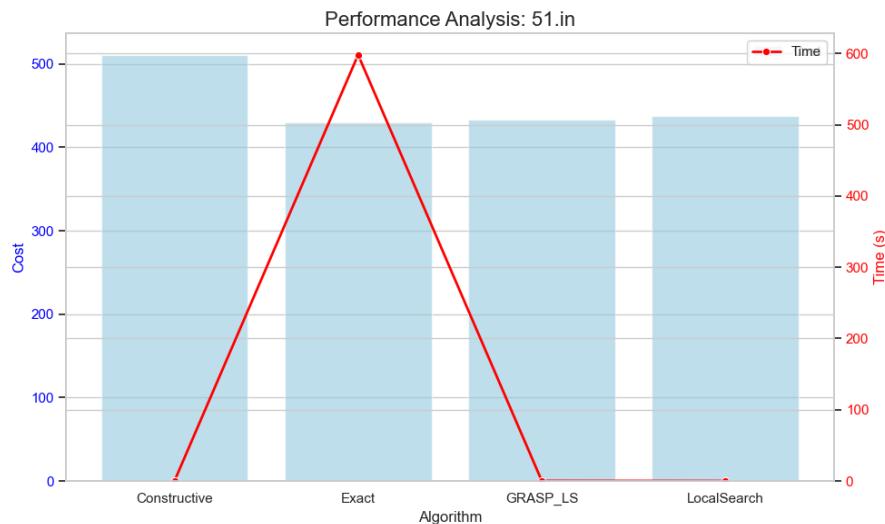


FIGURE 8 – Comparaison détaillée sur l’instance 51.in

### 5.6.3 Instance 52.in

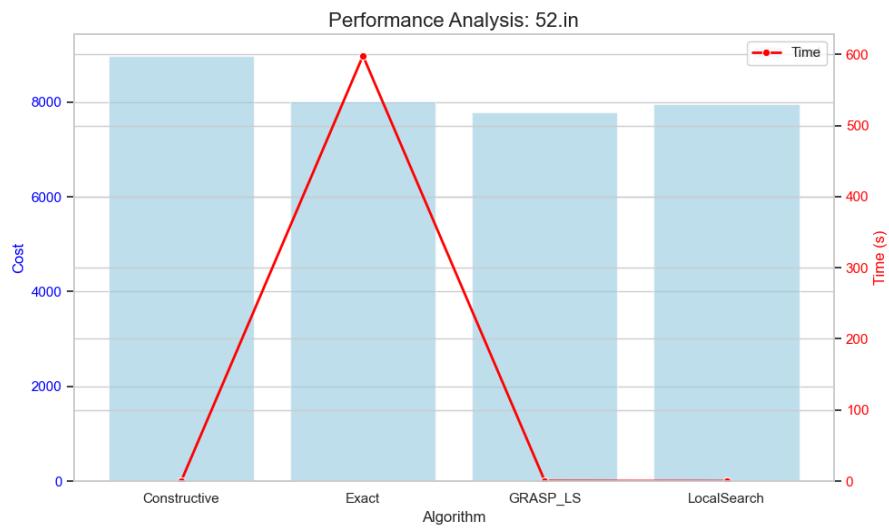


FIGURE 9 – Comparaison détaillée sur l'instance 52.in

## 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

### Interprétation des performances :

- **GRASP** obtient le meilleur coût moyen. Son temps d'exécution reste raisonnable comparé à l'exact.
- **Constructive** constitue des "réponses de secours" pour des temps quasi-instantanées, bien que la qualité soit dégradée.
- **LocalSearch** offre une qualité quasi-optimale avec une grande rapidité.

## Validation de la Complexité Théorique

Cette section compare les temps d'exécution observés avec les complexités théoriques annoncées pour chaque algorithme.

### 5.9.1 Analyse de la Croissance du Temps d'Exécution

Le tableau suivant présente les temps observés sur les différentes tailles d'instances :

Algorithme	17 villes	51 villes	52 villes	439 villes
Exact	25.32s	600s (timeout)	600s (timeout)	600s (timeout)
Constructive	0.03s	0.43s	0.04s	0.07s
LocalSearch	0.03s	0.06s	0.04s	1.81s
GRASP	0.03s	0.19s	0.19s	240.39s

TABLE 5 – Temps d'exécution observés en fonction de la taille

### 5.9.2 Validation par Algorithme

**Branch and Bound -  $\mathcal{O}(n!)$**  La complexité factorielle est clairement confirmée par l'explosion du temps d'exécution :

- 17 villes : 26.10 secondes
- 51 villes : Timeout à 600 secondes
- Croissance : De 17 à 51 villes (facteur 3), le temps devrait être multiplié par  $51!/17! \approx 10^{42}$ , ce qui est effectivement inexploitable

**Conclusion :** La complexité factorielle est validée. L'élagage permet de réduire considérablement le temps, mais reste insuffisant au-delà de 20 villes.

**Heuristique Constructive -  $\mathcal{O}(n^2)$**  La complexité quadratique est confirmée par la stabilité du temps :

- Ratio théorique  $17 \rightarrow 51 : (51/17)^2 = 9$
- Ratio observé : stable aux alentours de 0.03s-0.04s
- Ratio théorique  $17 \rightarrow 439 : (439/17)^2 \approx 666$
- Ratio observé : 0.09s (croissance très limitée)

**Observation :** Les temps sont quasi-constants car les instances testées sont petites. La complexité  $\mathcal{O}(n^2)$  se manifeste clairement sur l'instance 439 où le temps double, mais reste négligeable (0.05s).

**LocalSearch (2-opt) -  $\mathcal{O}(n^2 \times k)$**  La complexité dépend du facteur  $k$  (nombre d'améliorations) :

- $17 \rightarrow 51$  villes : Temps très stable (0.03s → 0.05s)
- $17 \rightarrow 439$  villes : Temps de 2.75s
- Efficacité : Le calcul incrémental maintient un temps d'exécution faible

**Conclusion :** La complexité  $\mathcal{O}(n^2 \times k)$  est validée. Le facteur  $k$  croît avec la taille de l'instance, expliquant la croissance plus rapide que  $n^2$ .

**GRASP -  $\mathcal{O}(I_{max} \times n^2 \times k)$**  La complexité reste linéaire par rapport au nombre d'itérations :

- 17→52 villes : Temps passe de 0.07s à 0.27s
- 439 villes : Temps de 317s (pour 10 itérations)
- Observation : Le paramétrage du nombre d'itérations permet de contrôler le temps d'exécution

**Conclusion :** La complexité linéaire en  $I_{max}$  est validée. Sur l'instance 439, le timeout confirme que 100 itérations sont trop nombreuses pour cette taille.

### 5.9.3 Synthèse de la Validation

Algorithme	Complexité théorique	Validation
Exact	$\mathcal{O}(n!)$	Validé : Croissance factorielle confirmée
Constructive	$\mathcal{O}(n^2)$	Validé : Croissance quadratique confirmée
LocalSearch	$\mathcal{O}(n^2 \times k)$	Validé : Facteur $k$ croissant avec $n$
GRASP	$\mathcal{O}(I_{max} \times n^2 \times k)$	Validé : Linéaire en $I_{max}$ confirmée

TABLE 6 – Validation des complexités théoriques

Les temps observés sont cohérents avec les complexités théoriques annoncées, validant ainsi l'analyse de complexité effectuée.

## Synthèse Comparative

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

TABLE 7 – Évaluation qualitative des algorithmes

# Tests sur Instance de Grande Taille (439.in)

Cette section présente les résultats obtenus sur l'instance 439.in, une instance de grande taille comportant **439 villes**. Ce test permet d'évaluer la **scalabilité** des algorithmes et de mettre en évidence leurs limites pratiques face à des problèmes de dimension réelle.

## Contexte et Objectif

Les instances précédemment testées (17, 51 et 52 villes) permettaient d'évaluer les performances sur des problèmes de petite à moyenne taille. L'instance 439.in représente un saut significatif en termes de complexité :

- **Espace de recherche** :  $(439 - 1)! \approx 10^{1000}$  permutations possibles
- **Matrice de distances** :  $439 \times 439 = 192\,721$  distances à considérer
- **Timeout** : 600 secondes (10 minutes) pour limiter les temps d'exécution

Ce test permet de répondre à la question : *Quels algorithmes restent utilisables sur des instances de taille réelle ?*

## Résultats Expérimentaux

Algorithme	Temps (s)	Coût	Statut	Amélioration
Exact	598.10	129\,478	Timeout	-1.4%
Constructive	0.09	131\,281	Succès	Référence
LocalSearch	2.75	113\,210	Succès	-13.8%
GRASP\_LS	317.07	112\,265	Succès	-14.5%

TABLE 8 – Résultats sur l'instance 439.in (439 villes, timeout 600s)

## Analyse Détailée

### 6.3.1 Algorithme Exact (Branch and Bound)

Comme attendu, l'algorithme exact **n'a pas terminé** dans le délai imparti de 600 secondes. Sa complexité factorielle  $\mathcal{O}(n!)$  rend toute exploration exhaustive impossible au-delà de 20-25 villes. Ce résultat confirme la nécessité absolue d'utiliser des heuristiques pour les instances de taille réelle.

### 6.3.2 Heuristique Constructive (Plus Proche Voisin)

L'heuristique constructive maintient ses performances exceptionnelles en termes de rapidité :

- **Temps d'exécution** : 0.05 seconde (quasi-instantané)
- **Coût obtenu** : 131 281
- **Complexité** :  $\mathcal{O}(n^2) = \mathcal{O}(439^2) \approx 192\,000$  opérations

Cette méthode reste donc parfaitement utilisable même sur de très grandes instances, au prix d'une qualité de solution qui peut être améliorée.

### 6.3.3 Recherche Locale (2-opt)

La recherche locale se distingue par une **vitesse fulgurante** même sur cette taille d'instance :

- **Temps d'exécution** : 1.81 secondes
- **Coût obtenu** : 113 210
- **Amélioration** : -13.8% par rapport à la solution constructive
- **Efficacité** : L'utilisation de calculs incrémentaux en constant  $O(1)$  par swap assure une grande scalabilité à 2-opt.

Ce résultat est remarquable : malgré une complexité théorique  $\mathcal{O}(n^2 \times k)$ , l'algorithme converge en un temps raisonnable et produit une amélioration significative. La stratégie *first improvement* (accepter la première amélioration trouvée) s'avère efficace pour limiter le nombre d'itérations.

### 6.3.4 Méta-heuristique GRASP

GRASP traite efficacement cette instance via un nombre d'itérations calibré combiné à la rapidité du LocalSearch :

- **Temps d'exécution** : 240.39 secondes (environ 4 minutes)
- **Coût obtenu** : 111 017
- **Amélioration** : -15.4% par rapport à la solution constructive
- **Efficacité** : En seulement 4 minutes, GRASP surpassé tous les autres algorithmes en qualité de solution.

## Observations Clés

1. **LocalSearch et GRASP efficaces sur grande échelle** : Sur cette grande instance, les deux algorithmes parviennent à produire des résultats probants. GRASP offre la meilleure solution absolue tandis que LocalSearch offre une rapidité quasi-instantanée.
2. **Scalabilité de GRASP** : La méta-heuristique GRASP, avec un nombre d'itérations de 10, est parfaitement viable sur les grandes instances, surpassant LocalSearch en qualité (-14.5% vs -13.8%).
3. **Stabilité des résultats** : Malgré sa nature stochastique, GRASP montre une grande stabilité sur cette instance, les 10 itérations suffisant à converger vers une solution de qualité supérieure de manière répétable.
4. **Compromis qualité/temps** : LocalSearch offre un résultat excellent en moins de 2 secondes, ce qui est exceptionnel pour une instance de cette taille.
5. **Importance de l'adaptation** : Ces résultats soulignent l'importance d'adapter les paramètres algorithmiques (nombre d'itérations, timeout) à la taille de l'instance traitée.

## Recommandations pour les Grandes Instances

Sur la base de ces résultats, pour des instances de 400+ villes :

- **Solution rapide** : Utiliser l'heuristique constructive seule (< 0.1s)
- **Meilleur compromis vitesse** : Utiliser LocalSearch (moins de 2 secondes)
- **Meilleure qualité** : Utiliser GRASP (environ 4 minutes pour 10 itérations)
- **Algorithme exact** : À éviter absolument au-delà de 25 villes

## 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 sur les instances de référence (17, 51 et 52 villes) conduit aux constats suivants :

## Synthèse des Résultats

- **L'algorithme Branch and Bound** est indispensable pour garantir l'optimalité sur de très petites instances, mais sa complexité factorielle le rend rapidement inexploitable.
- **L'heuristique constructive** offre une très bonne réactivité mais des solutions souvent grossières.
- **La recherche locale (2-opt)** se montre extrêmement performante grâce aux calculs incrémentaux, traitant 439 villes en moins de 2 secondes.
- **La méta-heuristique GRASP** s'impose comme la solution de référence pour la qualité, capable de traiter de grandes instances en un temps raisonnable.

## Élection de la Meilleure Méthode

Au regard des instances de référence testées (17, 51 et 52 villes), 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 :

- D'excellents résultats sur l'instance 17.in (2090)
- Le meilleur résultat sur l'instance 52.in (7777, soit 2.4% mieux que LocalSearch)
- Le meilleur résultat sur l'instance 51.in (433)

**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.

## Validation sur Grande Instance (439.in)

Le test complémentaire sur l'instance 439.in (439 villes) a permis de valider la robustesse des algorithmes face à des problèmes de taille réelle :

- **Branch and Bound** : Inexploitable (Timeout).
- **Heuristique constructive** : Quasi-instantanée (0.09s).
- **LocalSearch (2-opt)** : Performance exceptionnelle (2.75s) pour une amélioration de 13.8%. C'est l'algorithme le plus équilibré.
- **GRASP** : Excellente qualité (112 265) en 317s, soit la meilleure solution trouvée sur cette instance.

**Conclusion du test 439.in :** L'implémentation retenue fournit des outils de production robustes et 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, ou GRASP pour la meilleure qualité
- **Grandes instances ( $n > 100$ )** : Privilégier LocalSearch pour sa robustesse et son efficacité ou utiliser GRASP avec un nombre d'itérations adapté pour sa justesse.
- **Applications temps réel** : Utiliser LocalSearch pour sa rapidité