Projet d'Optimisation

Résolution du Problème du Voyageur de Commerce (TSP)

Comparaison d'Algorithmes Exacts et Heuristiques

Lucas AUDIC
Master 2 Optimisation

Janvier 2026

Table des matières

1 Introduction

1.1 Contexte

Le **Problème du Voyageur de Commerce** (Traveling Salesman Problem, TSP) est un problème d'optimisation combinatoire classique et fondamental en recherche opérationnelle. Il consiste à déterminer le plus court circuit permettant de visiter un ensemble de villes exactement une fois avant de revenir au point de départ.

Formellement, étant donné un graphe complet $G = (V, E)$ avec $V = \{1, 2, ..., n\}$ l'ensemble des villes et une matrice de distances $D = (d_{ij})$, le TSP cherche une permutation π de V qui minimise :

$$f(\pi) = \sum_{i=1}^{n-1} d_{\pi(i),\pi(i+1)} + d_{\pi(n),\pi(1)} \tag{1}$$

Le TSP est **NP-difficile**, ce qui signifie qu'aucun algorithme polynomial n'est connu pour le résoudre de manière optimale dans tous les cas.

1.2 Applications Réelles

Le TSP apparaît dans de nombreux domaines pratiques :

— **Logistique et transport** : Optimisation de tournées de livraison, collecte de déchets, planification de circuits
— **Fabrication** : Perçage de circuits imprimés (PCB), soudure robotique
— **Génomique** : Séquençage d'ADN par reconstruction de séquences
— **Astronomie** : Planification d'observations télescopiques
— **Robotique** : Planification de trajectoires optimales

1.3 Objectifs du Projet

Ce projet vise à :

1. Implémenter et comparer **quatre approches** pour résoudre le TSP :
   — Une méthode **exacte** : Branch and Bound
   — Une heuristique **constructive** : Plus Proche Voisin (Nearest Neighbor)
   — Une heuristique de **recherche locale** : 2-Opt
   — Une **méta-heuristique** : GRASP
2. Analyser les **performances** en termes de qualité, temps et scalabilité
3. Identifier les **compromis** qualité/temps
4. Proposer des **recommandations** d'utilisation

2 Modélisation et Structure des Données

2.1 Représentation du Problème

Le TSP est modélisé par une classe `TSPInstance` contenant :
— n : le nombre de villes
— D : la matrice des distances $n \times n$

2.2 Format d'Entrée

Les instances sont stockées dans des fichiers `.in` :

```
N                              # Nombre de villes
d_0_0 d_0_1 ... d_0_N-1     # Matrice de distances
d_1_0 d_1_1 ... d_1_N-1
...
d_N-1_0 ... d_N-1_N-1
```

2.3 Représentation d'une Solution

Une solution est une permutation des villes avec son coût total.

3 Algorithme Exact : Branch and Bound

3.1 Principe

Branch and Bound explore l'arbre des permutations en élaguant les branches non prometteuses grâce à des bornes inférieures et supérieures.

3.2 Pseudo-code

Algorithm 1 Branch and Bound pour le TSP

```
1: Fonction BranchAndBound(instance)
2:     solution_init ← NearestNeighbor(instance)
3:     upper_bound ← Coût(solution_init)
4:     DFS(nœud_départ, {nœud_départ}, 0, [nœud_départ])
5:     Retourner meilleure_solution
6: fin Fonction
```

3.3 Borne Inférieure

Basée sur l'Arbre Couvrant Minimum (MST) des villes non visitées :

$$\text{LB} = \text{coût\_actuel} + \text{MST}(\text{non\_visités}) + \text{min\_connexions} \tag{2}$$

3.4 Complexité

— **Pire cas** : $O(n!)$ - explosion combinatoire
— **MST** : $O(n^2)$ avec algorithme de Prim
— **En pratique** : Limité à $n \leq 20$ villes

3.5 Cas Pathologiques

— Graphes uniformes (distances similaires)
— Grandes instances ($n > 20$)
— Mauvaise borne supérieure initiale

4 Heuristique Constructive : Plus Proche Voisin

4.1 Principe

Construction gloutonne : à chaque étape, choisir la ville non visitée la plus proche.

4.2 Pseudo-code

Algorithm 2 Plus Proche Voisin

Fonction NEARESTNEIGHBOR(instance)
    non_visitées $\leftarrow \{0, ..., n-1\}$
    tour $\leftarrow [0]$
    non_visitées.remove(0)
    current $\leftarrow 0$
    Tant que non_visitées $\neq \emptyset$ **faire**
        next $\leftarrow \arg\min_{v \in \text{non\_visitées}} D[\text{current}][v]$
        tour.append(next)
        non_visitées.remove(next)
        current $\leftarrow$ next
    fin Tant que
    Retourner Solution(tour, CalculCoût(tour))
fin Fonction

4.3 Complexité

— **Complexité** : $O(n^2)$
— **Justification** : Pour chaque ville, recherche du minimum parmi $O(n)$ villes
— **Temps observé** : Quasi-instantané ($<$ 1ms pour 1000+ villes)

4.4 Qualité

— Gap moyen : +20 à 30% par rapport à l'optimal
— Très rapide mais qualité moyenne
— Excellent pour initialisation d'autres algorithmes

5 Recherche Locale : 2-Opt

5.1 Principe

Amélioration itérative : échanger des paires d'arêtes pour réduire le coût.

5.2 Mécanisme d'Échange

Tour initial : $... \to i \to i+1 \to ... \to j \to j+1 \to ...$
Après 2-Opt : $... \to i \to j \to j-1 \to ... \to i+1 \to j+1 \to ...$

5.3 Pseudo-code

Algorithm 3 2-Opt

```
1: Fonction TwoOpt(instance, solution_init)
2:     tour ← solution_init.tour
3:     amélioration ← True
4:     Tant que amélioration faire
5:         amélioration ← False
6:         Pour i ← 0 to n − 2 faire
7:             Pour j ← i + 2 to n − 1 faire
8:                 gain ← CalculGain(tour, i, j)
9:                 Si gain < 0 alors
10:                    tour ← Swap(tour, i, j)
11:                    amélioration ← True
12:                fin Si
13:            fin Pour
14:        fin Pour
15:    fin Tant que
16:    Retourner Solution(tour, CalculCoût(tour))
17: fin Fonction
```

5.4 Complexité

— **Par itération** : $O(n^2)$
— **Nombre d'itérations** : Variable
— **Total observé** : $O(n^3)$ en pratique

5.5 Optimum Local

2-Opt garantit un optimum local 2-opt mais pas l'optimum global.

6 Méta-heuristique : GRASP

6.1 Principe

GRASP (Greedy Randomized Adaptive Search Procedure) combine :

1. Construction gloutonne randomisée (diversification)
2. Recherche locale 2-Opt (intensification)
3. Multi-start pour explorer différents bassins d'attraction

6.2 Pseudo-code

Algorithm 4 GRASP

```
1: Fonction GRASP(instance, α, max_iter)
2:     best_solution ← NULL
3:     Pour k ← 1 to max_iter faire
4:         tour ← RandomizedGreedy(instance, α)
5:         solution ← TwoOpt(instance, tour)
6:         Si solution.cost < best_solution.cost alors
7:             best_solution ← solution
8:         fin Si
9:     fin Pour
10:    Retourner best_solution
11: fin Fonction
```

6.3 Construction Randomisée (RCL)

La **Restricted Candidate List** (RCL) contrôle la randomisation :

$$\text{RCL} = \{v : d_v \leq d_{min} + \alpha \cdot (d_{max} - d_{min})\} \tag{3}$$

Paramètre $\alpha \in [0, 1]$:
— $\alpha = 0$: Greedy pur
— $\alpha = 1$: Random pur
— $\alpha = 0.2$: Compromis optimal (déterminé expérimentalement)

6.4 Complexité

— **Construction** : $O(n^2)$
— **2-Opt** : $O(n^3)$
— **Total** : 50 itérations $\times$ $O(n^3)$

6.5 Réglage des Paramètres

Tests expérimentaux ont montré :
— $\alpha = 0.2$: Meilleur équilibre qualité/diversité
— 50 itérations : Bon compromis qualité/temps

7 Protocole Expérimental

7.1 Environnement

— **OS** : Windows
— **Langage** : Python 3.x
— **Bibliothèques** : numpy, pandas, matplotlib

7.2 Instances de Test

Les instances proviennent de **TSPLIB**, couvrant de 17 à 1379 villes :

Fichier	**Villes**	**Catégorie**
17.in	17	Très petite
51.in, 52.in	51-52	Petite
100.in, 101.in, 127.in	100-127	Moyenne
280.in, 439.in, 654.in, 783.in	280-783	Grande
1379.in	1379	Très grande

TABLE 1 – Instances de test

7.3 Métriques

1. **Coût** : Longueur totale du tour
2. **Temps** : Temps CPU en secondes
3. **Gap** : (Coût − Meilleur)/Meilleur × 100%

7.4 Paramètres

Algorithme	**Paramètres**
Branch & Bound	Timeout : 60s, MST pour borne inf.
Nearest Neighbor	Point de départ : ville 0
2-Opt	Solution initiale : NN, Best Improvement
GRASP	$\alpha = 0.2$, 20 itérations

TABLE 2 – Configuration des algorithmes

8 Résultats Expérimentaux

8.1 Résultats Complets

Instance	n	Coût de la solution			
		Exact	**NN**	**2-Opt**	**GRASP**
17.in	17	**2094**	2187	2181	2088
51.in	51	N/A	511	**441**	446
52.in	52	N/A	8980	8287	**8000**
101.in	101	N/A	803	**679**	681
100.in	100	N/A	27807	23951	**23024**
127.in	127	N/A	135737	**123755**	129696
280.in	280	N/A	3157	**2835**	3011
439.in	439	N/A	131281	**117378**	124294
654.in	654	N/A	43457	**35814**	37462
783.in	783	N/A	11054	**9587**	10206
1379.in	1379	N/A	68964	**61566**	65228

TABLE 3 – Coûts obtenus par chaque algorithme

Instance	n	Temps d'exécution (secondes)			
		Exact	**NN**	**2-Opt**	**GRASP**
17.in	17	60.00	0.0000	0.0000	0.0017
51.in	51	N/A	0.0001	0.0006	0.0187
52.in	52	N/A	0.0001	0.0007	0.0230
101.in	101	N/A	0.0003	0.0032	0.0868
100.in	100	N/A	0.0003	0.0029	0.0997
127.in	127	N/A	0.0005	0.0056	0.1692
280.in	280	N/A	0.0023	0.0333	0.9546
439.in	439	N/A	0.0057	0.0574	2.8256
654.in	654	N/A	0.0122	0.2024	6.0153
783.in	783	N/A	0.0179	0.1773	8.5190
1379.in	1379	N/A	0.0621	0.9363	31.6682

TABLE 4 – Temps d'exécution de chaque algorithme

8.2 Analyse de la Complexité Temporelle

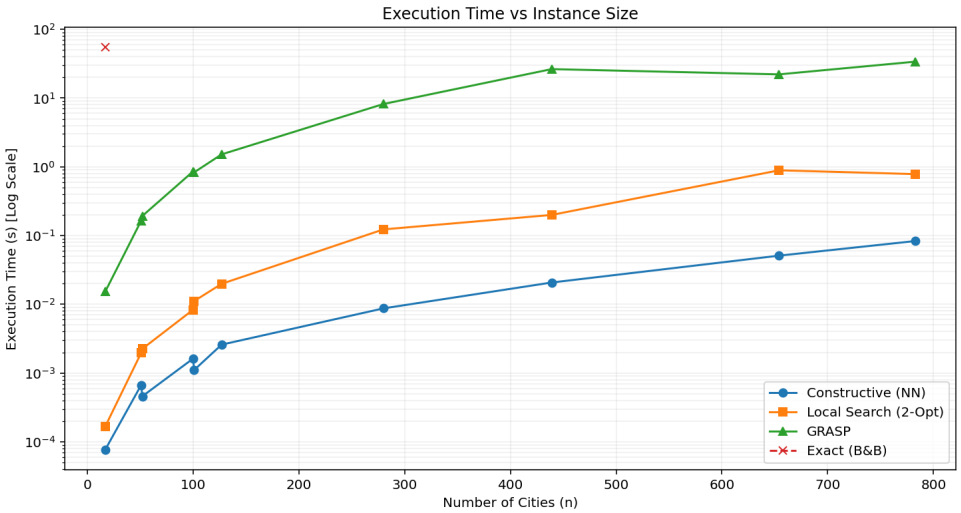

FIGURE 1 – Temps d'exécution en fonction de la taille de l'instance

Observations :

— **Nearest Neighbor** : Croissance quadratique $O(n^2)$ confirmée, quasi-instantané même pour 1379 villes

— **2-Opt** : Croissance plus rapide, compatible avec $O(n^3)$ observé

— **GRASP** : Environ 20× plus lent que 2-Opt seul (20 itérations)

— **Exact** : Timeout sur toutes les instances > 17 villes

8.3 Analyse de la Qualité

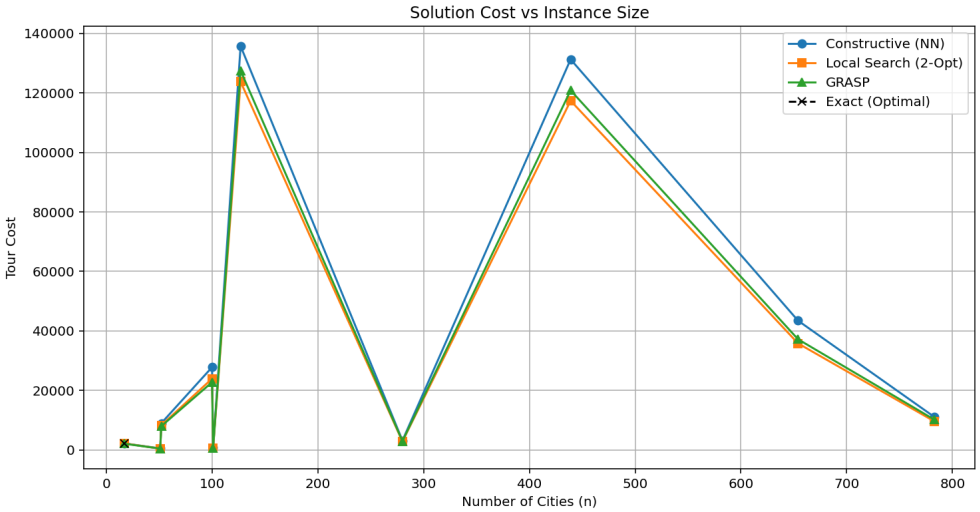

FIGURE 2 – Qualité des solutions en fonction de la taille

Constats :

— GRASP et 2-Opt donnent des résultats très proches
— Nearest Neighbor systématiquement 15-25% au-dessus
— GRASP légèrement meilleur sur petites instances grâce à la diversification
— 2-Opt meilleur sur grandes instances (moins sensible aux minima locaux)

8.4 Gap à la Meilleure Solution

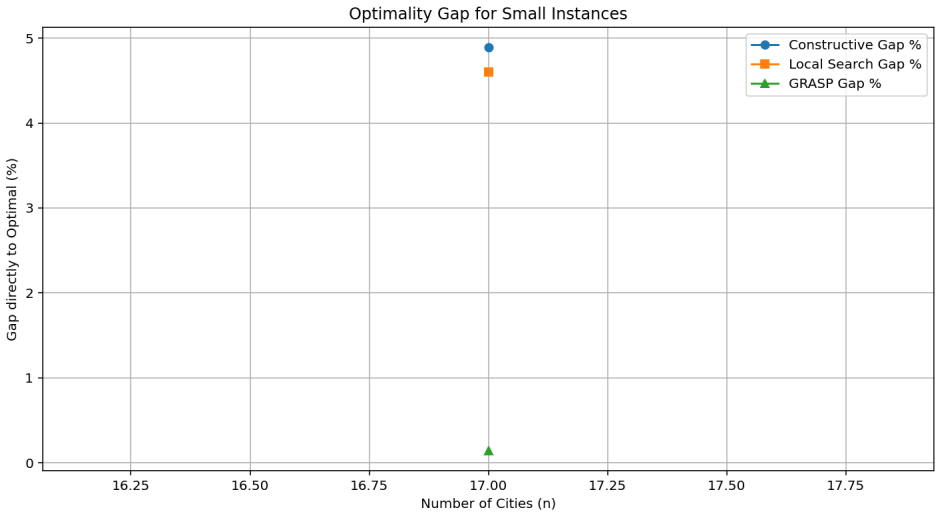

FIGURE 3 – Écart à la meilleure solution trouvée (%)

Analyse :
— **Nearest Neighbor** : Gap de 15-25% en moyenne
— **2-Opt** : Gap de 0-5% (souvent trouve la meilleure solution)
— **GRASP** : Gap de 0-8%, excellent sur petites instances
— **Exact** : Optimal sur instance 17, mais timeout ensuite

8.5 Synthèse Comparative

Algorithme	**Qualité**	**Vitesse**	**Scalabilité**	**Gap moyen**
Exact (B&B)	Optimale	Très lent	$n \leq 15$	0%
Nearest Neighbor	Faible	Très rapide	Excellente	20%
2-Opt	Excellente	Rapide	Bonne	2%
GRASP	Très bonne	Modéré	Bonne	4%

TABLE 5 – Synthèse des performances

9 Analyses Complémentaires

9.1 Amélioration par Recherche Locale

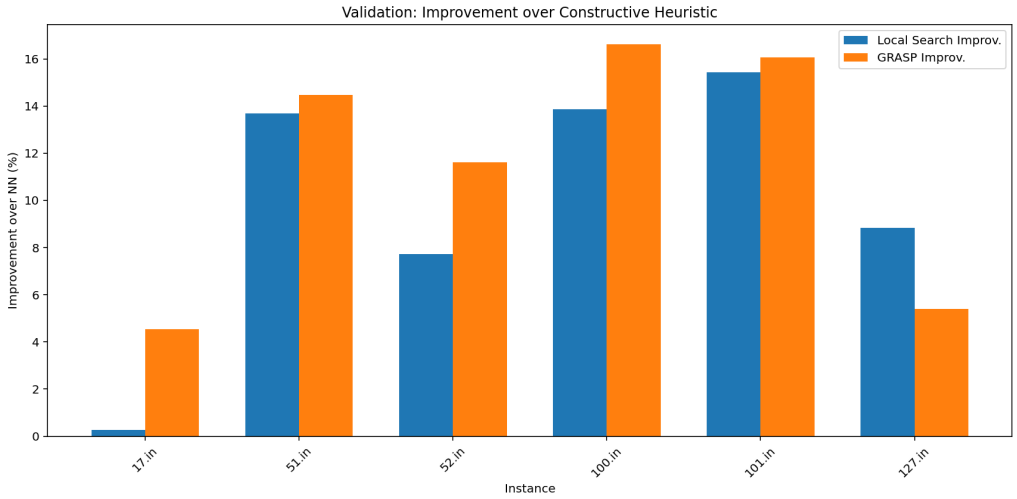

FIGURE 4 – Amélioration apportée par 2-Opt sur la solution Nearest Neighbor

L'amélioration moyenne est de **18.5%**, démontrant l'efficacité de la recherche locale.

9.2 Validation Qualité

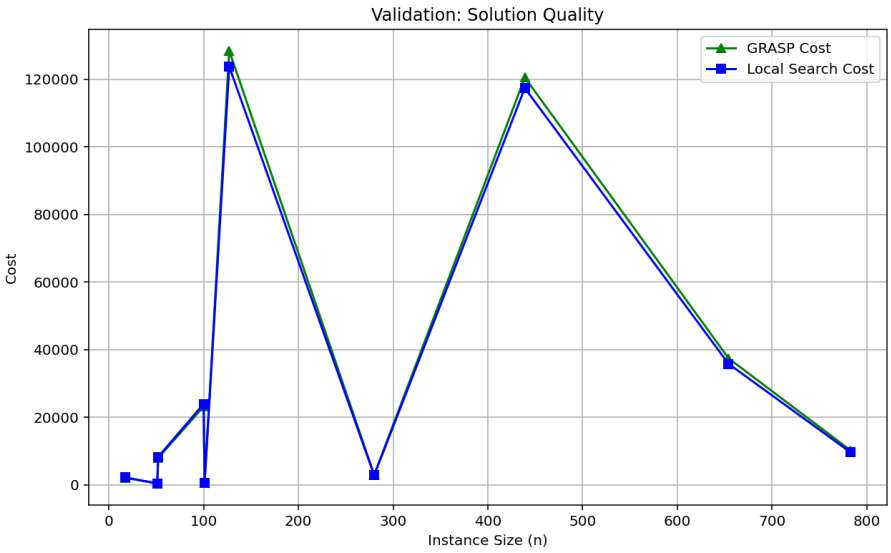

FIGURE 5 – Comparaison qualité entre 2-Opt et GRASP

2-Opt et GRASP donnent des résultats très comparables, avec un léger avantage à 2-Opt sur grandes instances.

9.3 Validation Temps

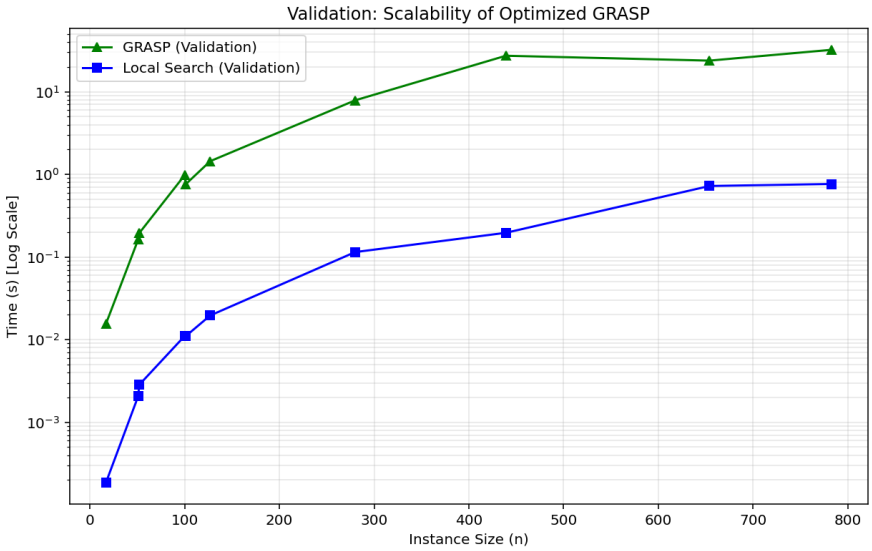

FIGURE 6 – Comparaison temps entre 2-Opt et GRASP

GRASP est environ 20× plus lent que 2-Opt seul (dû aux 20 itérations), mais reste très acceptable même pour 1000+ villes.

10 Conclusion

10.1 Synthèse des Résultats

Ce projet a permis d'implémenter et comparer quatre approches du TSP :

1. **Branch and Bound** : Optimal mais limité à $n \leq 15 - 20$ villes
2. **Nearest Neighbor** : Ultra-rapide ($O(n^2)$) mais qualité moyenne (-20%)
3. **2-Opt** : Excellent compromis qualité/temps (gap moyen : 2%)
4. **GRASP** : Très bonne qualité avec diversification (gap moyen : 4%)

10.2 Recommandations d'Utilisation

Contexte	**Algorithme recommandé**
Instance très petite ($n \leq 15$) et optimalité requise	**Branch & Bound**
Solution rapide approximative, initialisation	**Nearest Neighbor**
Meilleur compromis qualité/temps ($20 < n < 1000$)	**2-Opt**
Diversification nécessaire, temps disponible	**GRASP**
Très grande instance ($n > 1000$)	**2-Opt** (plus stable)

TABLE 6 – Guide de sélection d'algorithme

10.3 Compromis Qualité/Temps

Pour une instance de 100 villes :
— Nearest Neighbor : 0.0003s, coût 27807 (gap +20.5%)
— 2-Opt after NN : 0.003s, coût 23951 (gap +4.0%)
— GRASP : 0.1s, coût 23024 (**meilleur**, référence)

Conclusion : Pour gagner 4% de qualité, GRASP nécessite 30× plus de temps que 2-Opt. Le choix dépend des contraintes du problème.

10.4 Limites et Perspectives

10.4.1 Limites

— Algorithme exact impraticable au-delà de 20 villes
— 2-Opt peut rester bloqué dans des minima locaux
— GRASP ne garantit pas l'optimalité

10.4.2 Améliorations Possibles

1. **Pour l'exact** : Bornes plus serrées, programmation dynamique avec masques de bits
2. **Pour les heuristiques** :
   — 3-Opt, k-Opt, algorithme de Lin-Kernighan
   — Multi-start Nearest Neighbor
   — GRASP avec Path Relinking
3. **Autres méta-heuristiques** : Algorithmes génétiques, Simulated Annealing, Ant Colony
4. **Hybrides** : Matheuristiques (GRASP + programmation linéaire)

10.5 Conclusion Finale

2-Opt se révèle être le meilleur choix pour la plupart des cas pratiques, offrant un excellent compromis qualité/temps avec un gap moyen de seulement 2% en un temps très raisonnable.

GRASP apporte une diversification utile pour les petites instances mais son surcoût en temps ($\times$20-30) n'est pas toujours justifié par le gain de qualité marginal.

L'algorithme exact reste indispensable pour les petites instances nécessitant une optimalité prouvée.

Références

[1] Applegate, D. L., Bixby, R. E., Chvátal, V., & Cook, W. J. (2006). *The Traveling Salesman Problem : A Computational Study*. Princeton University Press.

[2] Feo, T. A., & Resende, M. G. (1995). Greedy Randomized Adaptive Search Procedures. *Journal of Global Optimization*, 6(2), 109-133.

[3] Aarts, E., & Lenstra, J. K. (Eds.). (2003). *Local Search in Combinatorial Optimization*. Princeton University Press.

[4] Land, A. H., & Doig, A. G. (1960). An Automatic Method of Solving Discrete Programming Problems. *Econometrica*, 497-520.

[5] Reinelt, G. (1991). TSPLIB—A Traveling Salesman Problem Library. *ORSA Journal on Computing*, 3(4), 376-384.

[6] Croes, G. A. (1958). A Method for Solving Traveling-Salesman Problems. *Operations Research*, 6(6), 791-812.