Ministère de l'Enseignement Supérieur et de la Recherche Scientifique
Université Mohamed El-Bachir El-Ibrahimi Bordj Bou Arréridj
Faculté des mathématiques et de l'informatique
Département des Mathématiques

MÉMOIRE

Pour l'obtention du diplôme de

MASTER

En Recherche Opérationnelle

Présenté par

Boumerta Maram
Messadek Aicha

Théme

Étude empirique entre les algorithmes heuristiques et métaheuristiques pour le Problème du Voyageur de Commerce

Évalue devant le jury composé de:

Mr.Saha Adel	Université de BBA	Encadreur
Mr.Maache Salah	Université de BBA	Examinateur
Mr.Fillali Ferhat	Université de BBA	Président

Année Universitaire

2024/2025

Dédicace

Je dédie ce travail à mon encadrant, qui a su guider mes réflexions avec patience et expertise. Son accompagnement a été une source précieuse de rigueur et d'inspiration tout au long de ce projet.

À mes parents, dont le soutien inconditionnel et les sacrifices ont permis de construire mon parcours académique. Leur amour et leurs encouragements ont été la clé de ma persévérance et de ma réussite.

À ma famille et mes amis, qui ont toujours cru en moi et ont été présents dans les moments de doute comme dans les instants de succès. Leur confiance et leur bienveillance ont rendu cette aventure plus enrichissante.

Enfin, à toutes les personnes qui ont contribué, de près ou de loin, à ce travail. Que ce soit par un conseil, un échange d'idées ou un simple geste de soutien, leur présence a été précieuse dans cette démarche.

Remerciements

Ce travail est le fruit d'un engagement soutenu, d'un accompagnement bienveillant et de nombreux échanges enrichissants. Je tiens à exprimer ma profonde gratitude à toutes les personnes qui ont contribué, de près ou de loin, à sa réalisation.

À ceux qu'ont partagé leur expertise et leurs précieux conseils, permettant d'approfondir cette réflexion et d'enrichir chaque étape de ce projet.

À ceux dont le soutien infaillible, la patience et l'encouragement ont été une source de motivation essentielle tout au long de cette aventure.

Enfin, à tous ceux qui, par leurs discussions, leur aide technique ou simplement leur présence, ont participé à rendre ce parcours plus inspirant et plus constructif.

Résumé

Ce mémoire analyse les méthodes heuristiques et métaheuristiques pour résoudre le Problème du Voyageur de Commerce (TSP), comparant leur efficacité en termes de rapidité, qualité et complexité. Il met en évidence les forces et limites de chaque approche, montrant que les heuristiques comme Clarke et Wright et 2-opt offrent un bon compromis entre rapidité et précision. Enfin, l'étude souligne l'importance du TSP dans l'optimisation logistique et ouvre des perspectives pour des recherches futures.

Mots-clés : Problème du voyageur de commerce, méthodes heuristiques, métaheuristiques, optimisation, comparaison des algorithmes.

Abstract

This thesis analyzes heuristic and metaheuristic methods to solve the Traveling Salesman Problem (TSP), comparing their efficiency in terms of speed, solution quality, and complexity. It highlights the strengths and limitations of each approach, showing that heuristics like Clarke and Wright and 2-opt offer a good balance between speed and precision. Finally, the study emphasizes the importance of TSP in logistics optimization and opens perspectives for future research.

Keywords: Traveling Salesman Problem, heuristic methods, metaheuristics, optimization, algorithm comparison.

Contents

List of Figures

List of Tables

Liste d'abréviations:

- **RO** : Recherche opérationnelle.
- **TSP** : Traveling Salesman Problem.
- **PVC** : Problème du Voyageur de Commerce.
- **UPS** : United Parcel Service.
- **PO** : Problème d'Optimisation.
- **POC** : Problème d'Optimisation Combinatoire.
- **ACO** : Ant Colony Optimization.
- **PPV** : Plus Proche Voisin.
- **MST** : Minimum Spanning Tree.
- **CW** : Clarke and Wright.
- **VRP** : Vehicle Routing Problem.
- **NP** : Non-déterministe Polynomial.
- **AG** : Algorithmes Génétiques.
- **RS** : Recuit Simulé.

- **RT** : Recherche Tabou.

- **FF** : Formule Factorielle.

Introduction générale:

La Recherche Opérationnelle (RO) est une discipline scientifique qui vise à modéliser, analyser et résoudre des problèmes complexes de prise de décision, en s'appuyant sur les mathématiques, l'algorithmique et l'analyse de données. Elle est aujourd'hui largement utilisée dans des domaines variés comme la logistique, la gestion de production, les transports ou encore l'optimisation de réseaux.

L'optimisation combinatoire occupe une place centrale dans les domaines de la recherche opérationnelle, de l'informatique et des sciences appliquées.

Parmi les problèmes emblématiques qui y sont étudiés, le Problème du Voyageur de Commerce (PVC) s'illustre par sa complexité théorique et son utilité pratique dans des secteurs variés tels que la logistique, le transport ou encore la microélectronique.

Ce mémoire s'inscrit dans une démarche analytique et comparative des algorithmes heuristiques et métaheuristiques appliqués à la résolution du PVC. Il s'articule autour de deux grands objectifs : d'une part, comprendre les fondements théoriques de ces méthodes, leurs mécanismes et leurs domaines d'application; d'autre part, évaluer empiriquement leur performance sur différentes instances issues de la bibliothèque TSPLIB, en termes de qualité de la solution, temps de calcul et complexité algorithmique.

La première partie pose le cadre conceptuel, en introduisant les bases de l'optimisation combinatoire et les différentes classes de problèmes qui y sont associées. Le second chapitre est entièrement dédié au PVC : son historique, sa formulation mathématique, ses implications en complexité, et son intérêt tant académique qu'industriel.

La troisième partie explore les différentes approches de résolution, en distinguant méthodes exactes, heuristiques, et métaheuristiques. Enfin, le dernier chapitre propose une étude expérimentale approfondie qui confronte ces méthodes à travers des résultats concrets sur des cas tests.

Par cette analyse, ce travail ambitionne de dégager les points forts, les limites et les complémentarités des méthodes étudiées, en vue d'identifier les algorithmes les plus adaptés selon les contraintes spécifiques des instances du TSP. Il ouvre également des perspectives de recherche, notamment en direction de solutions hybrides combinant rigueur et efficacité.

Chapter I

Optimisation combinatoire

1.1 Introduction:

L'optimisation combinatoire est une discipline essentielle qui joue un rôle clé en recherche opérationnelle, en mathématiques discrètes et en informatique. Elle intervient dans de nombreux domaines, tels que la gestion, l'ingénierie, la production, les télécommunications, le transport, l'énergie et la médecine, où elle permet de résoudre des problèmes complexes[1].

Ces problèmes, bien que faciles à comprendre, sont souvent difficiles à résoudre, car la plupart appartiennent à la classe des problèmes NP-difficiles, rendant les solutions exactes impraticables lorsque la taille du problème augmente[1].

Pour surmonter ces défis, les chercheurs développent des approches innovantes, allant des méthodes heuristiques et métaheuristiques aux algorithmes exacts, afin d'optimiser les processus et améliorer l'efficacité des systèmes[1].

L'optimisation combinatoire est particulièrement précieuse dans des applications concrètes comme l'optimisation des itinéraires de transport ou la gestion des ressources en production, où elle permet d'améliorer les performances et de répondre à des défis industriels[1].

À la croisée des mathématiques, de l'informatique et de l'ingénierie, elle met en évidence le pouvoir des algorithmes et modèles pour résoudre des problèmes complexes du monde réel[1].

Dans ce chapitre,on pose les bases théoriques essentielles. On commence par introduire les concepts généraux d'optimisation et d'optimisation combinatoire, en expliquant leur importance dans divers domaines tels que les mathématiques et l'informatique.

On illustre également des exemples concrets de problèmes combinatoires, comme le problème d'affectation, le problème du sac à dos et le problème du voyageur de commerce (TSP), avant de conclure sur leur pertinence et leurs défis uniques.

1.2 Optimisation :

L'optimisation est une compétence essentielle qui permet de trouver la meilleure solution parmi un ensemble de possibilités. Cela implique de maximiser ou minimiser une fonction objective tout en respectant des contraintes spécifiques. Elle est un outil essentiel pour les entreprises et les organisations, car elle permet de résoudre des problèmes complexes et de prendre des décisions efficaces[2].

1.3 Problème d'optimisation:

1.3.1 Fonction objective:

La fonction objective est une expression mathématique qui définit le critère à optimiser dans un problème d'optimisation. Elle peut être une **minimisation**(réduction des coûts, du temps, des distances) ou une **maximisation**(augmentation du profit, de l'efficacité, de la performance), selon le contexte du problème étudié[3].

75Mathématiquement, elle est formulée comme :

- **Problème de minimisation** :

$$\min f(x)$$

- **Problème de maximisation** :

$$\max f(x)$$

où $f(x)$ est la fonction à optimiser, et x représente les variables de décision.

1.3.2 Définitions:

Nous avons retenu la définition de problème d'optimisation proposée par Collette et Siarry, qui est la suivante :

Définition 1:(Problème d'optimisation)

Un problème d'optimisation se définit comme la recherche du minimum ou du maximum d'une fonction donnée. Mathématiquement, dans le cas d'une minimisation, un problème d'optimisation se présentera sous la forme suivante[4] :

$$(\text{PO}) \begin{cases} \min f(x) \\ g_i(x) \leq 0, \quad i = 1, \ldots, n \quad \text{(contraintes d'inégalités)} \\ h_i(x) = 0, \quad i = 1, \ldots, m \quad \text{(contraintes d'égalités)} \end{cases} \tag{I.1}$$

En d'autres termes, résoudre un problème d'optimisation (PO) revient à déterminer une solution $s^* \in S$ minimisant ou maximisant la fonction f avec S l'ensemble des solutions ou l'espace de recherche, et $f : S \to Y$ une application ou une fonction d'évaluation qui, à chaque

configuration s, associe une valeur $f(s) \in Y$. Il est possible de passer d'un problème de maximisation à un problème de minimisation grâce à la propriété suivante[4] :

$$\max_{s \in S} f(s) = \min_{s \in S}(-f(s)). \tag{I.2}$$

Généralement, une solution $s \in S$ est un vecteur d'un espace à N dimensions.

Définition 2: (Problème d'optimisation continue)

Dans le cas de variables réelles, on a[4]:

$$S \subseteq \mathbb{R}^N \tag{I.3}$$

On parle alors de problème d'optimisation en variables continues. Un problème d'optimisation continue (PO) peut être formulé de la façon suivante [4]:

$$\text{(PO)} \quad \min_{s \in S} f(s) \tag{I.4}$$

Définition 3:(Problème d'optimisation combinatoire)

Un problème d'optimisation combinatoire est un problème d'optimisation dans lequel l'espace de recherche S est dénombrable. Un problème d'optimisation combinatoire (POC) peut être formulé ainsi [4]:

$$\text{(POC)} \quad \min_{s \in \mathbb{Z}^N} f(s) \tag{I.5}$$

Une solution S est un vecteur composé de N valeurs entières:

$$S \subseteq \mathbb{Z}^N \tag{I.6}$$

La principale différence entre les problèmes d'optimisation continue et combinatoire repose sur l'utilisation de variables discrètes. Dans les deux catégories de problèmes, une solution $s \in S$ est une instanciation des variables $x_i \in X$, où i est l'indice de la variable dans $[1, N]$, et X est le vecteur de dimensions N correspondant à la solution. $f(s)$ représente l'évaluation de cette solution.

Résoudre ces problèmes revient à trouver une solution optimale, appelée aussi *optimum global* [4].

1.4 Optimisation combinatoire:

L'optimisation combinatoire est une branche des mathématiques et de l'informatique qui traite des problèmes d'optimisation où l'ensemble des solutions possibles est discret ou fini. Ces problèmes consistent à sélectionner la meilleure solution parmi un ensemble donné, en maximisant ou minimisant une fonction objective tout en respectant des contraintes spécifiques[1].

1.5 Exemples de problème d'optimisation combinatoire:

1.5.1 Problème d'affectation:

Le problème d'affectation est un type spécifique de problème d'optimisation combinatoire où l'objectif est d'affecter un ensemble d'entités (par exemple des travailleurs, des tâches, ou des ressources) à un autre ensemble (par exemple des postes, des machines, ou des destinations) de manière optimale, tout en minimisant un coût total ou en maximisant une valeur globale[5].

1.5.1.1 Formulation Mathématique:

Le problème est généralement représenté par une matrice de coûts, où chaque ligne représente une entité à affecter et chaque colonne représente une destination ou une tâche. Le coût c_{ij} représente le coût d'affectation de l'entité i à la tâche j. L'objectif est de trouver une affectation optimale telle que [5]:

$$\min \sum_{i=1}^{n} \sum_{j=1}^{n} c_{ij} x_{ij} \tag{I.7}$$

Sous les contraintes suivantes :
Chaque entité est affectée à une seule tâche :

$$\sum_{j=1}^{n} x_{ij} = 1 \quad \forall i \tag{I.8}$$

Chaque tâche reçoit une seule entité :

$$\sum_{i=1}^{n} x_{ij} = 1 \quad \forall j \tag{I.9}$$

$x_{ij} \in \{0, 1\}, \quad$ où $x_{ij} = 1$ indique que l'entité i est affectée à la tâche j, et $x_{ij} = 0$ sinon.

1.5.2 Problème du sac à dos:

Le problème du sac à dos (Knapsack Problem) est un classique en optimisation combinatoire. Il consiste à déterminer quels objets, parmi un ensemble donné, doivent être sélectionnés pour maximiser leur valeur totale tout en respectant une contrainte de capacité ou de poids[6].

1.5.2.1 Formulation Mathématique:

On considérons un ensemble de n objets, où chaque objet i a [6]:

- Une valeur v_i,
- Un poids w_i.

L'objectif est de maximiser la valeur totale des objets sélectionnés, tout en respectant la limite de capacité du sac, notée W.

Fonction Objective :

$$\max \sum_{i=1}^{n} v_i x_i \tag{I.10}$$

Sous les contraintes :

$$\sum_{i=1}^{n} w_i x_i \leq W \tag{I.11}$$

$$x_i \in \{0, 1\}, \quad \forall i \tag{I.12}$$

où $x_i = 1$ si l'objet i est sélectionné, et $x_i = 0$ sinon.

1.5.3 Problème du voyageur de commerce:

Le Problème du Voyageur de Commerce (Traveling Salesman Problem-TSP) est un problème classique en optimisation combinatoire. Il consiste à trouver le trajet le plus court permettant à un voyageur de visiter un ensemble de villes, en revenant à son point de départ, tout en visitant chaque ville une et une seule fois[7].

1.5.3.1 Formulation Mathématique:

Le problème peut être représenté par un graphe complet $G = (V, E)$, où[7]. :

- V est l'ensemble des villes (les sommets du graphe),
- E est l'ensemble des trajets (les arêtes du graphe),
- Chaque arête (i, j) a un coût ou une distance c_{ij} associée.

L'objectif est de minimiser la distance totale parcourue :

$$\min \sum_{i=1}^{n} \sum_{j=1}^{n} c_{ij} x_{ij} \tag{I.13}$$

Sous les contraintes :

$$\sum_{j=1}^{n} x_{ij} = 1 \quad \forall i \quad \text{(Chaque ville est visitée exactement une fois)} \tag{I.14}$$

$$\sum_{i=1}^{n} x_{ij} = 1 \quad \forall j \quad \text{(Chaque ville est quittée exactement une fois)} \tag{I.15}$$

$$x_{ij} \in \{0, 1\} \quad \text{où } x_{ij} = 1 \text{ indique que le trajet entre } i \text{ et } j \text{ fait partie de la solution.} \tag{I.16}$$

1.6 Conclusion:

Dans ce chapitre, nous avons posé les bases de l'optimisation combinatoire, en définissant ses principes et son importance dans divers domaines.

Nous avons exploré les problèmes d'optimisation, distinguant les approches exactes et heuristiques.

Plusieurs exemples clés ont été abordés, comme le problème d'affectation, le problème du sac à dos et le problème du voyageur de commerce (PVC), mettant en lumière leurs formulations mathématiques et les techniques utilisées pour les résoudre.

Cette introduction permet de mieux comprendre les enjeux de l'optimisation et servira de fondement aux chapitres suivants, où nous approfondirons le problème du voyageur de commerce et ses méthodes de résolution.

Chapter II

Problème du voyageur de commerce

2.1 Introduction:

Le problème du voyageur de commerce (Traveling Salesman Problem-TSP) est un cas d'école en optimisation combinatoire et en informatique. Il s'agit de déterminer le trajet le plus efficace permettant de visiter un ensemble de villes une seule fois, tout en garantissant le retour à la ville de départ[8].

Ce problème, qui peut paraître simple, est classé comme NP-difficile, car le nombre de solutions possibles croît de manière exponentielle avec le nombre de villes[8].

En pratique, le problème du voyageur de commerce trouve des applications dans des domaines tels que la planification de tournées, l'optimisation des itinéraires pour les services de transport, la conception de circuits électroniques et bien d'autres. De nombreuses approches ont été développées pour résoudre ce problème[8].

Les méthodes exactes, comme l'algorithme branch-and-bound, garantissent une solution optimale, mais deviennent impraticables pour des problèmes de grande taille[8].

En revanche, les heuristiques et les métaheuristiques, telles que les algorithmes génétiques ou l'optimisation par colonies de fourmis, offrent des solutions approximatives en un temps raisonnable[8].

Le TSP représente un défi mathématique et algorithmique, à la fois terrain d'expérimentation pour tester de nouveaux algorithmes et outil pour résoudre des problèmes réels, alliant ainsi pertinence théorique et utilité pratique[8].

Dans ce chapitre, en concentre spécifiquement sur le TSP.en détaille l'historique, en retraçant son développement depuis les travaux de Hamilton jusqu'aux recherches modernes.

Ce chapitre propose une formulation mathématique du TSP et aborde sa complexité algorithmique, en mettant en lumière le défi posé par la croissance exponentielle des solutions possibles.

Enfin, en explore l'intérêt théorique et pratique de ce problème, en soulignant ses applications dans des secteurs tels que le transport et la logistique.

2.2 Problème du voyageur de commerce:

Un problème du voyageur de commerce correspond à un problème d'optimisation combinatoire classique,le voyageur de commerce doit effectuer un itinéraire en passant par chaque ville une seule fois. Il commence par une ville et termine sa tournée à son point de départ. Les distances entre les villes sont connues. Quel itinéraire doit-il choisir pour minimiser la distance parcourue [9] ?

2.3 Histoire:

- **Les années 1800**

  Sir William Rowan Hamilton, mathématicien irlandais, et Thomas Penyngton Kirkman, Britannique. Hamilton est le créateur du jeu *Icosian* en 1857. Il s'agissait d'un tableau de vingt trous, avec la contrainte qu'au moins un sommet ne soit visité qu'une seule fois, qu'aucune arête ne soit visitée plus d'une fois et que le point d'arrivée soit le même que le point de départ. Ce type de chemin a finalement été appelé **circuit hamiltonien**[10].

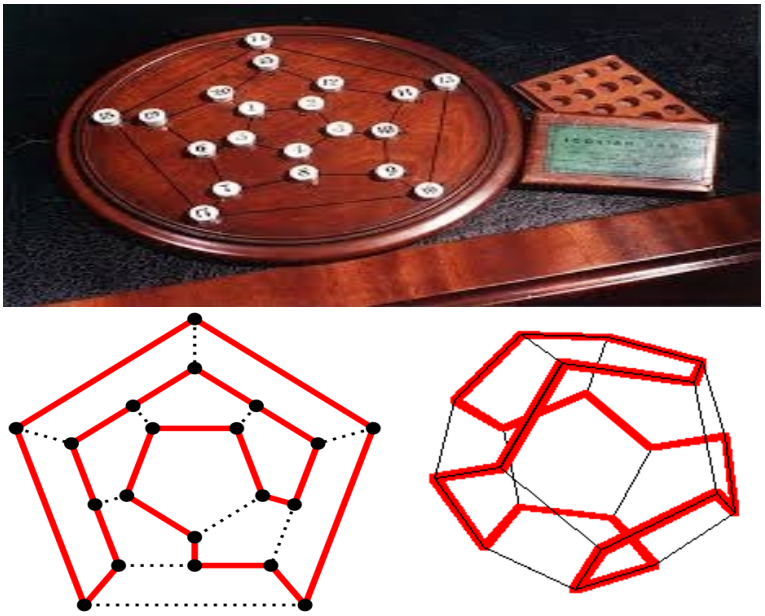

Figure 2.1: Le jeu Icosien d'Hamilton [9].

- **Vers 1929-1930**

  Karl Menger étudie la forme générale du TSP à Vienne et à Harvard[10].

- **Dans les années 1930**

Le mathématicien et économiste Karl Menger a étudié le TSP pour la première fois à Vienne et à Harvard. Ces travaux ont ensuite été étudiés par *Hassler Whitney* et *Merrill Flood* à Princeton[10].

Années	**Taille de l'instance**	**Équipe de recherche**
1954	49 villes	G.Dantzig,R.Fulkerson,S.Johnson
1971	64 villes	M.held,R.M.Karp
1975	67 villes	P.M.Camerini,L.Fratta,F.Maffioli
1977	120 villes	M.Grotschel
1980	318 villes	H.Crowder,M.W.Padberg
1987	532 villes	M.Padberg,G.Rinald
1987	2392 villes	M Padberg,G.Rinald
1994	7397 villes	David L.Applegate,Robert E.Bixby,Vasek Chvatal,William J.Cook
1998	13509 villes	David L.Applegate,Robert E.Bixby,Vasek Chvatal,William J.Cook
2001	15112 villes	David L.Applegate,Robert E.Bixby,Vasek Chvatal,William J.Cook
2004	24978 villes	David L.Applegate,Robert E.Bixby,Vasek Chvatal,William J.Cook

Table 2.1: Récapitulatif des étapes de le PVC[11].

2.4 La formule mathématique:

La formulation mathématique du Problème du Voyageur de Commerce (TSP) repose effectivement sur un graphe complet

$$G = (V, E)$$

, où chaque arête a un poids ou un coût associé, représentant la distance ou le temps entre deux villes[12].

En complément, les permutations jouent un rôle fondamental dans la recherche du trajet optimal. Deux types principaux de permutations peuvent être distingués et détaillés dans ce contexte [12]:

2.4.1 Une permutation quadratique:

Pour la matrice de coûts $C = (C_{ij})$, où C_{ij} représente la distance entre les villes i et j $(1 \leq i, j \leq n)$, on cherche[9] :

$$\pi = (\pi_1, \pi_2, \ldots, \pi_n) \tag{II.1}$$

- π_1**:** représente la première ville à visiter.

- π_2**:** la deuxième ville, et ainsi de suite jusqu'à π_n.

Cette permutation minimise la somme des distances parcourues[9] :

$$C_{\pi(1)\pi(2)} + C_{\pi(2)\pi(3)} + \cdots + C_{\pi(n-1)\pi(n)} + C_{\pi(n)\pi(1)}. \tag{II.2}$$

2.4.2 Une permutation linéaire:

On définit x_{ij} comme une variable binaire telle que[12] :

$$x_{ij} = \begin{cases} 1 & \text{si l'arc } (i,j) \text{ est utilisé,} \\ 0 & \text{sinon.} \end{cases} \tag{II.3}$$

L'objectif est de minimiser le coût total tout en respectant les conditions suivantes [12]:

- Chaque sommet du graphe doit être visité exactement une seule fois.
- Le chemin doit revenir au point de départ.

Le problème peut être formulé de manière mathématique comme suit [12] :

$$\min \sum_{i=1}^{n} \sum_{j=1}^{n} C_{ij} x_{ij} \tag{II.4}$$

- Avec:

$$\sum_{j=1}^{n} x_{ij} = 1, \quad \text{pour tout } i, \quad (2) \tag{II.5}$$

$$\sum_{i=1}^{n} x_{ij} = 1, \quad \text{pour tout } j, \quad (3) \tag{II.6}$$

$$\sum_{i \in S} \sum_{j \in S} x_{ij} \leq |S| - 1, \quad \forall S \subset V, 2 \leq |S| \leq n-1, \quad (4) \tag{II.7}$$

$$x_{ij} \in \{0,1\}, \quad \text{pour tout } i,j \in V. \tag{II.8}$$

• Les équations (2) et (3) garantissent que seul un arc est le point de départ et le point d'arrivée dans la boucle. L'équation (4) garantit que le modèle ne générera aucune solution de sous-boucle [12].

2.5 La complexité de certaines problématiques:

Dans le contexte du TSP, si n est le nombre de villes à visiter, le nombre d'ordres distincts (ou permutations) dans lesquels ces villes peuvent être visitées est $n!$[1].

Cependant, comme le TSP suppose que le voyageur commence et termine dans la même ville (un aller-retour), n'importe quel itinéraire peut être parcouru en sens inverse sans créer

une nouvelle solution. Cette symétrie implique que le nombre total d'itinéraires uniques est divisé par deux, réduisant ainsi le nombre de solutions à $(n-1)!$[1].

De plus, si la ville de départ est fixe (par exemple, le voyageur part toujours de la même ville), le nombre d'itinéraires uniques est réduit à:

$$\frac{(n-1)!}{2}$$

.

En effet, en fixant la ville de départ, on élimine les permutations redondantes impliquant différents points de départ[1].

Exemple :

pour un parcours avec 4 destinations, le nombre total de permutations est de $4! = 24$. En outre, si l'on considère que la direction de l'itinéraire est indifférente, le nombre d'itinéraires distincts s'élève à 12. Si la ville de départ est fixe, le nombre de permutations se réduit à :

$$\frac{(4-1)!}{2} = \frac{3!}{2} = \frac{6}{2} = 3$$

itinéraires uniques[1].

Il est important de noter que la fonction factorielle $n!$ connaît une croissance extrêmement rapide à mesure que n augmente[1].

Exemple :

Cette croissance exponentielle rend le problème du voyageur de commerce (TSP) irréalisable à résoudre par des méthodes de force brute lorsque n devient grand. Bien que les méthodes de calcul modernes soient capables de traiter des valeurs de n faibles à modérées, obtenir des solutions exactes pour des valeurs élevées de n reste un défi majeur dans le domaine de la recherche algorithmique[1].

Valeur de n	**Factorielle ($n!$)**	**Complexité ($O(n!)$)**	**Implication**
$n = 1$	1	$O(1)$	Pas d'itinéraires.
$n = 2$	2	$O(1)$	Solution évidente.
$n = 3$	6	$O(n!)$	Traitement direct.
$n = 4$	24	$O(n!)$	Calculs accessibles.
$n = 5$	120	$O(n!)$	Gérable par exhaustivité.
$n = 6$	720	$O(n!)$	Complexité notable.
$n = 7$	5 040	$O(n!)$	Approches avancées.
$n = 8$	40 320	$O(n!)$	Complexité accrue.
$n = 9$	362 880	$O(n!)$	Brute force impraticable.
$n = 10$	3 628 800	$O(n!)$	Heuristiques indispensables.
$n = 20$	2 432 902 008 176 640 000	$O(n!)$	Inaccessible classiquement.
$n = 50$	3.04110^{64}	$O(n!)$	Heuristiques essentielles.
$n = 100$	9.33210^{157}	$O(n!)$	Incalculable.

Table 2.2: Factorielles, complexités et implications pour le problème du voyageur de commerce[1].

2.6 L'intérêt:

Le PVC est un domaine d'expertise qui se situe à l'intersection des disciplines scientifiques des mathématiques, de l'informatique et des applications industrielles. Cette position stratégique confère à ce champ de recherche une pertinence et une valeur ajoutée scientifiques notables[9].

Il est un exemple d'étude d'un problème NP-complet dont les méthodes de résolution peuvent être appliquées à d'autres problèmes mathématiques discrets. En outre, ce matériau possède des applications directes, notamment dans les secteurs du transport et de la logistique[9] .

Le PVC est un problème qui concerne directement certains secteurs, notamment ceux du transport et de la logistique. Il optimise les itinéraires des bus scolaires en trouvant le chemin le plus court ou la distance que le bras mécanique d'une machine devra parcourir pour percer les trous d'une carte de circuit imprimé, les trous représentant des villes[9] .

2.7 Conclusion:

Dans ce chapitre, nous avons exploré en profondeur le problème du voyageur de commerce, un défi fondamental en optimisation combinatoire et en recherche opérationnelle.

Nous avons présenté son historique, depuis les premiers travaux de Hamilton et Menger, jusqu'aux avancées modernes. Nous avons également détaillé sa formulation mathématique, illustrant les différentes approches basées sur les permutations et les graphes pondérés.

L'analyse de sa complexité a montré que le TSP appartient à la classe des problèmes NP-difficiles, ce qui rend son résolution exacte impraticable pour des instances de grande taille.

Enfin, nous avons souligné l'importance du TSP à travers ses nombreuses applications dans des domaines tels que la logistique, le transport et l'optimisation industrielle.

Cette compréhension théorique du problème servira de base aux prochains chapitres, où nous examinerons les méthodes de résolution permettant d'obtenir des solutions optimales ou approchées.

Chapter III

Analyse des Méthodes de Résolution du Problème du Voyageur de Commerce

3.1 Introduction:

Le problème du Voyageur de Commerce (PVC) est un problème fondamental d'optimisation combinatoire. En raison de sa complexité algorithmique élevée, le problème est classé comme NP-difficile.

Pour résoudre ce problème, différentes méthodes existent, allant des approches exactes, telles que la programmation dynamique et la méthode de branch-and-bound , aux heuristiques comme plus proche voisin, la méthode d'insertion,le Clarke-Wright(CW),les améliorations locales comme le 2-opt et l'arbre de recouvrement minimal,et métaheuristiques comme les colonies de fourmis et les algorithmes génétiques [13].

Ces méthodes permettent d'obtenir des solutions proches de l'optimum tout en réduisant le temps de calcul, rendant le Problème voyageur de commerce(PVC) applicable dans divers domaines tels que la logistique, la planification des tournées et l'optimisation des réseaux[13].

Dans ce chapitre,on examine différentes approches pour résoudre le Problème voyageur de commerce. On présenté les méthodes exactes est les méthodes approchées (heuristiques et métaheuristiques).

3.2 Méthodes pour résoudre le problème du voyageur de commerce:

Afin de parvenir à une solution quant au problème rencontré par le voyageur de commerce,il est possible de mettre en œuvre deux méthodes distinctes:Les methodes exacts,Les methodes approchées.

3.2.1 Les methodes exactes:

3.2.1.1 Définition:

les méthodes exactes constituent des approches méthodiques visant à identifier la solution optimale à un problème donné[14].

Ces méthodes s'appuient sur un examen systématique de toutes les possibilités ou sur l'utilisation de techniques mathématiques pour réduire l'espace de recherche tout en assurant l'identification de la meilleure solution[14].

Dans le cadre de la problématique du voyageur de commerce (TSP), les méthodes exactes suivantes sont à considérer[14] :

3.2.1.2 Recherche exhaustive(Brute-Force):

La recherche exhaustive, également connue sous le nom de Brute-Force, est une méthode qui consiste à énumérer toutes les permutations possibles de l'ordre de visite des villes. Pour chaque permutation, la longueur totale du chemin correspondant est calculée[14].

L'objectif est de sélectionner le chemin le plus court parmi l'ensemble des chemins calculés. Cette approche présente l'avantage de garantir la découverte de la solution optimale.Cette approche est à la fois simple à comprendre et à implémenter[14].

Cependant, cette méthode présente des inconvénients notables, notamment une complexité factorielle ($O(n!)$), ce qui signifie que le temps de calcul augmente de façon exponentielle avec le nombre de villes[14].

Ainsi, même pour un nombre modéré de villes, le problème devient rapidement impraticable[14].

3.2.1.3 Algorithme de Branch and Bound :

Cette méthode explore l'espace de solutions de manière plus efficace que la recherche exhaustive. Elle repose sur la construction d'un arbre de recherche, où chaque nœud représente un sous-ensemble de solutions possibles[14].

Pour optimiser cette exploration, des techniques de « bounding » (calcul de bornes) sont employées pour estimer la longueur minimale des chemins dans chaque sous-ensemble. De plus, les branches de l'arbre qui ne peuvent pas conduire à une solution optimale sont éliminées (élagage)[14].

Les avantages de cette méthode résident dans sa capacité à surpasser la recherche exhaustive en termes d'efficacité, en offrant des solutions plus rapides et souvent plus précises. Elle peut

également traiter des problèmes légèrement plus grands[14].

Cependant, cette approche reste d'une complexité exponentielle, bien que généralement inférieure à celle de la recherche exhaustive. Il est important de noter que son efficacité dépend fortement de la qualité des bornes utilisées pour guider l'exploration[14].

3.2.1.4 La programmation dynamique, ou algorithme de Held-Karp:

Est une approche mathématique qui vise à résoudre des problèmes complexes de manière efficace[14].

Elle se caractérise par une décomposition du problème en sous-problèmes plus faciles à résoudre, et ce, de manière itérative. Elle stocke les solutions des sous-problèmes afin d'éviter leur recalcul. Cette approche, dite « bottom-up », permet de construire progressivement la solution optimale[14].

Cette approche présente plusieurs avantages, notamment par rapport à la recherche exhaustive et au Branch and Bound, pour certains problèmes[14].

Cependant, elle présente une complexité en temps de $O(n^2 2^n)$, ce qui constitue une amélioration significative par rapport à $O(n!)$[14].

Cependant, il est important de noter que cette approche présente une complexité exponentielle, ce qui limite inévitablement son applicabilité à des problèmes de taille réduite. De plus, elle demande beaucoup de mémoire pour stocker les solutions des sous-problèmes[14].

En résumé, les méthodes précises sont utiles si l'on cherche une solution optimale et que le problème est assez petit[14].

3.2.2 Les methodes approchées:

3.2.2.1 Définition:

La méthode approchées (approximation), est une approche stratégique employée pour la résolution de problèmes complexes lorsque les solutions exactes s'avèrent trop onéreuses en termes de calcul. Son objectif est de trouver une solution acceptable, proche de l'optimum, dans un délai raisonnable[15].

Dans le cadre du problème du voyageur de commerce (PVC), les méthodes d'approximation visent à minimiser la distance parcourue en passant par chaque ville une seule fois[15].

Bien que ces approches ne garantissent pas systématiquement l'optimalité, elles fournissent des résultats exploitables pour des problèmes de grande taille.on peut les subdiviser en deux classe :les heuristiques et les métaheuristiques[15].

3.2.2.2 Les Heuristiques :

Comme pour tous les problèmes NP-difficile, la recherche d'heuristiques performantes est l'une des principes préoccupation des spécialistes du domaine[16].

Le principe d'une méthode heuristique est de trouver, en temps raisonnable une solution de bonne qualité. Dans ce qui suit, nous décrirons les trois grandes familles d'heuristiques développées pour les problèmes de tournées : méthodes constructives, méthodes de recherche locales et méthodes en deux phases[16].

3.2.2.2.1 Les Méthodes constructives:

L'idée de base d'une heuristique constructive est de réduire la taille du problème à chaque étape pour limiter progressivement l'ensemble des solutions réalisables[16].

Les heuristiques constructives partent d'une solution vide et construisent étape par étape une solution finale s de S. Les choix effectués à chaque étape k sont faits selon certaines règles . Ces règles dépendent donc du problème considéré. Bien évidemment, à chaque étape k, l'ensemble des solutions réalisables S est réduit en un sous-ensemble S_k[16].

Ces heuristiques sont caractérisées par la facilité de mise en œuvre et la rapidité d'exécution. Par contre, la faible qualité des solutions trouvées est en général leur principal défaut [16]. Parmi ces méthodes, nous pouvons citer l' **Algorithme plus proche voisin** et la **Méthode de l'insertion la moins coût** et l'**Algorithme de Clarke et Wright**[16].

3.2.2.2.2 Les méthodes d'amélioration:

Ces méthodes sont un outil essentiel pour la création d'une solution initiale pour les méta-heuristiques. Elles utilisent des approches de recherche locale pour perfectionner progressivement une solution initiale déjà obtenue. À chaque itération d'une recherche locale, un voisinage de la solution actuelle est exploré pour identifier une meilleure solution[16].

Dans ce processus itératif, l'objectif est de trouver la meilleure solution possible, en définissant des critères d'arrêt spécifiques à chaque cas.Ces méthodologies ont été élaborées pour permettre à la recherche de s'affranchir des minima locaux. Parmi ces méthodes,nous pouvons citer **2-opt** [16].

3.2.2.2.3 Méthodes en deux phases:

Les méthodes en deux phases reposent sur une décomposition du problème en une partition en sous-groupes de l'ensemble des clients, suivie de la détermination de la tournée associée à chaque sous-groupe. Deux classes d'algorithmes peuvent être distinguées en fonction de l'ordre dans lequel ces deux phases sont réalisées [16]:

- **Route-first and cluster-second** : Dans cette méthode, une tournée initiale englobant tous les clients est d'abord déterminée. Ensuite, cette tournée est divisée en sous-groupes ou clusters de clients.

- **Cluster-first and route-second** : Ici, la première étape consiste à partitionner les clients en sous-groupes, puis à déterminer une tournée optimisée pour chaque sous-groupe.

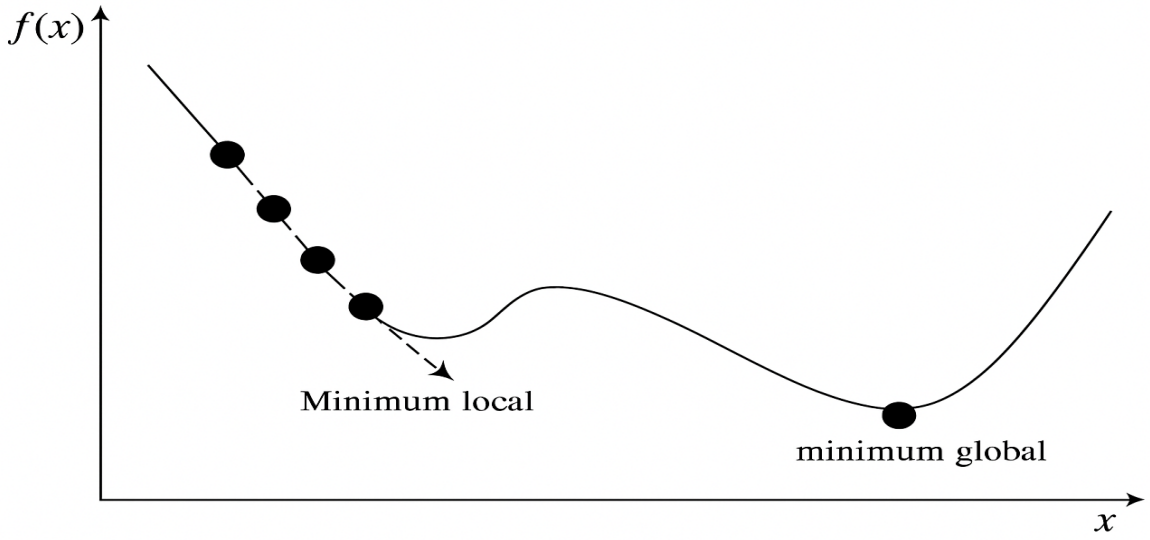

Figure 3.1: le principe des méthodes de voisinage[16].

3.2.2.2.4 l'heuristique du plus proche voisin:

Construit une solution étape par étape en choisissant à chaque itération l'élément le plus proche selon un critère donné, souvent utilisé dans les problèmes de tournée (comme le voyageur de commerce). [16].

3.2.2.2.5 L'heuristique d'insertion:

Consiste à construire progressivement une solution en insérant les éléments dans une position optimale au sein d'une solution partielle, ce qui permet de minimiser un coût ou d'améliorer la qualité de la solution à chaque étape[16].

3.2.2.2.6 les algorithmes gloutons:

Une approche où des choix locaux optimaux sont effectués à chaque étape avec l'espoir d'atteindre une solution globale satisfaisante. Bien que ces algorithmes ne garantissent pas toujours l'optimalité, ils offrent souvent un excellent compromis entre temps de calcul et qualité de solution[16].

3.2.2.2.7 Arborescence de poids minimal:

L'arborescence de poids minimal est une structure qui connecte tous les sommets d'un graphe avec un coût total minimal. Elle est utilisée dans les réseaux et l'optimisation des infrastructures[17].

Les principaux algorithmes de construction incluent Kruskal, qui ajoute les arêtes les moins coûteuses sans former de cycles, Prim, qui étend progressivement l'arbre en sélectionnant les arêtes de moindre poids, et Borůvka, qui favorise les calculs distribués[17].

Mathématiquement, le MST minimise la somme des poids des arêtes tout en couvrant tous les sommets. Il est essentiel dans la conception efficace des systèmes de transport et de communication[17].

3.2.2.3 Les Métaheuristiques :

Les métaheuristiques sont des méthodes générales de recherche dédiées aux problèmes d'optimisation difficiles. Ces méthodes sont généralement présentées sous la forme de concepts inspirés de la vie courante. Elles reprennent des idées que l'on retrouve parfois dans la vie quotidienne[16].

Elles s'inspirent notamment de l'éthologie (les colonies de fourmis), de la physique (le recuit simulé) et de la biologie (les algorithmes évolutionnaires)[16].

Les métaheuristiques permettent d'éviter de se limiter aux optimums locaux, qui sont des solutions réalisables mais peuvent être loin de l'optimum global. Dans ce qui suit, nous allons présenter les métaheuristiques les plus prisées par les chercheurs[16].

3.2.2.3.1 Recuit simulé:

Le recuit simulé combine une procédure de recherche locale avec un ensemble de règles et de mécanismes qui permettent de surmonter l'obstacle des extremums locaux, tout en évitant les problèmes de cycles[16].

Cette méthode a été appliquée avec succès pour résoudre de nombreux problèmes difficiles d'optimisation combinatoire, tels que les problèmes de routage de véhicules, d'ordonnancement et de coloration de graphes[16].

3.2.2.3.2 Recherche tabou:

La recherche tabou est une variante de l'algorithme de recherche par voisinage, au moins dans sa version de base[16].

Tant que l'on ne se trouve pas dans un optimum local, la recherche tabou se comporte comme toute méthode de voisinage et améliore la valeur objective à chaque étape. Lorsqu'un minimum local est toutefois atteint, il est possible de choisir au moins le voisinage le moins mauvais[16].

L'inconvénient de cette dernière démarche est qu'une itération peut parfois ne pas suffire pour s'extraire d'un minimum local, et qu'un déplacement peut provoquer un déplacement inverse à une étape ultérieure, créant ainsi un cycle autour du minimum local. C'est pour cette raison que la recherche tabou garde en mémoire les dernières solutions visitées pour éviter d'y revenir, ce qu'on appelle la *liste tabou*[16].

3.2.2.3.3 Les colonies de fourmis:

La méthode de la colonie de fourmis imite le comportement de ces insectes qui, lorsqu'ils rencontrent un obstacle sur leur trajet, trouvent toujours le chemin le plus court pour le contourner. Leur technique repose sur la pose de marqueurs chimiques, appelés *phéromones*, sur les trajets parcourus[16].

Cette méthode peut paraître surprenante au premier abord, mais un chemin plus court reçoit en réalité plus de phéromones qu'un chemin plus long[16].

Cette métaheuristique, introduite pour la première fois par Colorni, a été appliquée au problème du voyageur de commerce (TSP) et aux problèmes de routage de véhicules (*Vehicle Routing Problems*, VRP)[16].

3.2.2.3.4 Les algorithmes génétiques:

L'une des métaheuristiques les plus utilisées pour résoudre le problème de routage de véhicules est l'algorithme génétique. Il s'agit d'une méthode bioinspirée introduite par Holland en 1975 dans le cadre d'une analogie avec la sélection naturelle des espèces[16].

Elle a ensuite été formalisée par Goldberg en 1989 pour être appliquée à la résolution de problèmes d'optimisation[16].

Un algorithme génétique entretient une population de solutions au problème à optimiser. La qualité de ces solutions (appelées « individus ») est évaluée par une fonction d'évaluation (dite « fitness »)[16].

L'algorithme applique les opérateurs de croisement et de mutation à la population en vue d'en générer une autre. À partir de deux individus parents, l'opérateur de croisement génère un ou deux individus enfants qui héritent chacun une partie de la structure des parents[16].

L'opérateur de mutation agit quant à lui sur un seul individu en modifiant une partie de sa structure[16].

3.2.3 Différence entre les méthodes exactes et les méthodes approchées:

Critère	**Méthodes Exactes**	**Méthodes Approchées (Heuristiques et Méta-heuristiques)**
Définition	Trouvent la solution optimale avec une garantie de précision.	Fournissent des solutions approximatives, souvent proches de l'optimum, sans garantie.
Temps de calcul	Généralement élevé, surtout pour les problèmes de grande taille.	Plus rapide, adapté aux problèmes complexes ou de grande taille.
Complexité	Souvent exponentielle pour les problèmes NP-difficiles.	Moins complexe, souvent linéaire ou quadratique.
Exemples	Branch and Bound, Programmation Dynamique, Algorithmes de Coupage.	Algorithmes génétiques, Recuit simulé, Recherche Tabou, Colonies de Fourmis.
Utilisation	Adaptées aux petits problèmes ou lorsque la solution optimale est indispensable.	Utilisées pour les problèmes complexes où une solution rapide et acceptable est suffisante.
Exploration	Explore tout l'espace de recherche pour garantir l'optimalité.	Exploite des stratégies pour explorer efficacement une partie de l'espace de recherche.
Robustesse	Moins robuste face aux variations des données ou des contraintes.	Plus robustes et adaptables aux changements dans les données ou les contraintes.

Table 3.1: Comparaison entre les méthodes exactes et approchées[18].

3.3 Conclusion:

Ce chapitre a exploré les différentes approches permettant de résoudre le problème du voyageur de commerce (PVC), en mettant en lumière leurs forces et leurs limites.

Nous avons débuté par une présentation des méthodes exactes, qui garantissaient une solution optimale mais étaient souvent limitées par leur coût computationnel élevé. Parmi elles, nous avons examiné la recherche exhaustive (Brute Force), l'algorithme Branch-and-Bound, et la programmation dynamique (Held-Karp).

Ensuite, nous avons introduit les méthodes approchées, qui offraient des solutions rapides et de qualité acceptable sans explorer l'ensemble de l'espace de recherche.

Nous avons distingué ici les heuristiques comme l'algorithme du plus proche voisin et l'algorithme d'insertion au moindre coût, l'arborescence de poids minimal, les méthodes constructives, les méthodes d'amélioration et les méthodes en deux phases.

Les métaheuristiques, qui incluaient des stratégies plus sophistiquées telles que le recuit simulé, la recherche tabou, les colonies de fourmis et les algorithmes génétiques, ont également été étudiées.

Après cette analyse détaillée, nous avons comparé les méthodes exactes et approchées en fonction de plusieurs critères tels que la complexité algorithmique, la qualité des solutions produites et le temps de calcul. Cette étude nous a permis de mieux comprendre dans quels contextes chaque approche était la plus adaptée.

Chapter IV

Étude empirique

4.1 introduction:

Afin d'optimiser nos méthodes de résolution du problème « Voyageur Commerce », nous avons réalisé une étude comparative des différentes approches existantes.[8].

Il existe deux grandes approches pour le résoudre les méthodes heuristiques et métaheuristiques .

Les heuristiques sont des techniques souvent spécifiques qui exploitent des règles déterministes pour produire rapidement une solution acceptable, tandis que les métaheuristiques sont des cadres d'optimisation généraux qui permettent d'explorer l'espace des solutions de manière adaptative et souvent stochastique[19].

Cette étude propose une comparaison approfondie des performances des heuristiques classiques (*Cheapest Insertion, Clarke & Wright, 2-opt*) et des métaheuristiques avancées (*Ant Colony Optimization, Genetic Algorithm, Simulated Annealing*) appliquées au TSP.

L'objectif est d'évaluer leur **qualité de solution, temps de calcul et complexité algorithmique**, afin de dégager les forces et limites de chaque approche dans différents contextes pratiques.

4.2 Base de teste:

TSPLIB est une bibliothèque de référence contenant des instances standardisées du **problème du voyageur de commerce (TSP)** et de problèmes connexes[20].

Développée pour permettre la comparaison des performances des algorithmes d'optimisation, elle regroupe plusieurs types de problèmes, notamment le **TSP symétrique**, où chaque ville est connectée aux autres avec des distances identiques dans les deux directions, et le **TSP asymétrique (ATSP)**, où les distances peuvent différer selon la direction du trajet[20].

Elle inclut également le **problème du cycle hamiltonien (HCP)**, qui vérifie si un graphe contient un cycle hamiltonien, le **problème de commande séquentielle (SOP)**, qui impose des contraintes de précédence entre les villes, et le **problème de routage de véhicules capacitaires (CVRP)**, qui optimise les tournées de livraison en tenant compte des capacités des véhicules[20].

Les instances de la TSPLIB sont stockées sous des formats bien définis, comprenant les **coordonnées des villes** (EUC 2D, GEO), des **matrices de distances** (EXPLICIT), et des **listes d'arêtes** utilisées dans les problèmes de cycle hamiltonien[20].

Ces instances sont largement exploitées pour tester et comparer les performances des **algorithmes heuristiques et métaheuristiques**, permettant ainsi d'évaluer la qualité des solutions et le temps de calcul des différentes méthodes[20].

Parmi les instances populaires, on retrouve **berlin52.tsp** (52 villes en Allemagne), **ch150.tsp** (150 villes), et **lu634.tsp** (634 villes)[20].

TSPLIB joue un rôle clé dans l'optimisation combinatoire en servant de référence pour l'évaluation des méthodes de résolution[20].

Elle est utilisée notamment pour tester des algorithmes comme **Ant Colony Optimization**, **Cheapest Insertion**, et **Clarke & Wright**, contribuant ainsi à la recherche et à l'amélioration des techniques d'optimisation[20].

Cette bibliothèque est une ressource incontournable pour les chercheurs et praticiens en optimisation[20].

4.2.1 Instances de Problèmes du Voyageur de Commerce – Classification et Description:

Les instances de la **TSPLIB** jouent un rôle fondamental dans l'évaluation des algorithmes de résolution du **problème du voyageur de commerce (TSP)**. Elles offrent un ensemble standardisé de données permettant de comparer les performances des méthodes heuristiques et exactes sur des problèmes de différentes tailles et complexités[20].

Parmi ces instances, on retrouve des problèmes de petite taille, comme `att48.tsp` et `berlin52.tsp`, `kroD100.tsp` et `ch150.tsp`.D'autres les instances de taille intermédiaire, telles que `d198.tsp`, `a280.tsp` et `lin318.tsp`, . Enfin, des instances de grande envergure, comme `ali535.tsp`, `d657.tsp` et `lu634.tsp`.

Ce catalogue varié constitue une référence incontournable pour tester et comparer des approches comme les heuristiques, les métaheuristiques et les méthodes exactes appliquées au TSP[20].

L'analyse comparative des performances sur ces instances permet d'identifier les approches les mieux adaptées aux différents scénarios pratiques du TSP[20].

Instance	Description
att48.tsp	48 villes des États-Unis, avec distances pseudo-euclidiennes, souvent utilisée pour tester les algorithmes de TSP.
berlin52.tsp	52 villes situées à Berlin, souvent utilisées pour tester des algorithmes de TSP.
kroD100.tsp	Instance de 100 villes, dérivée des problèmes de KroA, KroB, KroC, KroD et KroE, qui sont des benchmarks classiques.
ch150.tsp	Contient 150 villes, souvent utilisées pour évaluer la performance des heuristiques.
d198.tsp	Instance de 198 villes, avec des coordonnées spécifiques.
a280.tsp	280 villes, une instance bien connue pour les tests d'optimisation.
lin318.tsp	318 villes, utilisée pour des comparaisons d'algorithmes.
ali535.tsp	Instance de 535 villes, souvent étudiée pour des problèmes de grande taille.
d657.tsp	657 villes, une instance plus complexe.
lu634.tsp	980 villes, utilisée pour des tests avancés.

Table 4.1: Instances du Problème du Voyageur de Commerce (TSP)[20].

- L'accès aux fichiers d'instances peut se faire via le site officiel : TSPLib.Net sur GitHub

4.2.2 Spécifications du système utilisé pour l'étude:

Cette étude établier sur un PC(DELL(Latitude 3380)) :

Nom du périphérique	DESKTOP-2844DTC
Processeur	Intel(R) Core(TM) i3-6006U CPU @ 2.00GHz, 2.00 GHz
RAM installée	4.00 GB
Carte graphique	Intel(R) HD Graphics 520 (128 MB)
Type de système	Système d'exploitation 64 bits, processeur x64

Table 4.2: Caractéristiques techniques du PC utilisé pour l'étude.

- Les résultats obtenus correspondent à la moyenne de 10 exécutions.

4.3 Approches Heuristiques et Métaheuristiques pour l'Optimisation du TSP:

Les approches heuristiques et métaheuristiques jouent un rôle clé dans l'optimisation du problème du voyageur de commerce(PVC)[21].

Les **heuristiques de construction**, telles que *Nearest Neighbor, Cheapest Insertion et Clarke & Wright*, permettent de générer rapidement une solution initiale en suivant des règles déterministes[21].

Ensuite, les **heuristiques d'amélioration**, comme *2-opt*, interviennent pour affiner et optimiser la solution obtenue, en explorant des permutations locales afin de réduire le coût du parcours[21].

En complément, les **métaheuristiques**, notamment *algorithme de la colonie de fourmis, algorithme génétique* exploitent des stratégies inspirées de phénomènes naturels ou de principes statistiques, permettant une exploration plus profonde de l'espace des solutions pour améliorer la qualité des résultats[21].

Cette étude applique ces méthodes à différentes **instances du TSP**, telles que *berlin52.tsp, ch150.tsp et lu634.tsp...ect*, afin d'évaluer leur efficacité et de comparer leurs performances en fonction du **temps de calcul, de la qualité des solutions produites et de leur complexité algorithmique**. L'analyse comparative vise ainsi à identifier les approches les plus adaptées aux différents scénarios d'application du TSP.

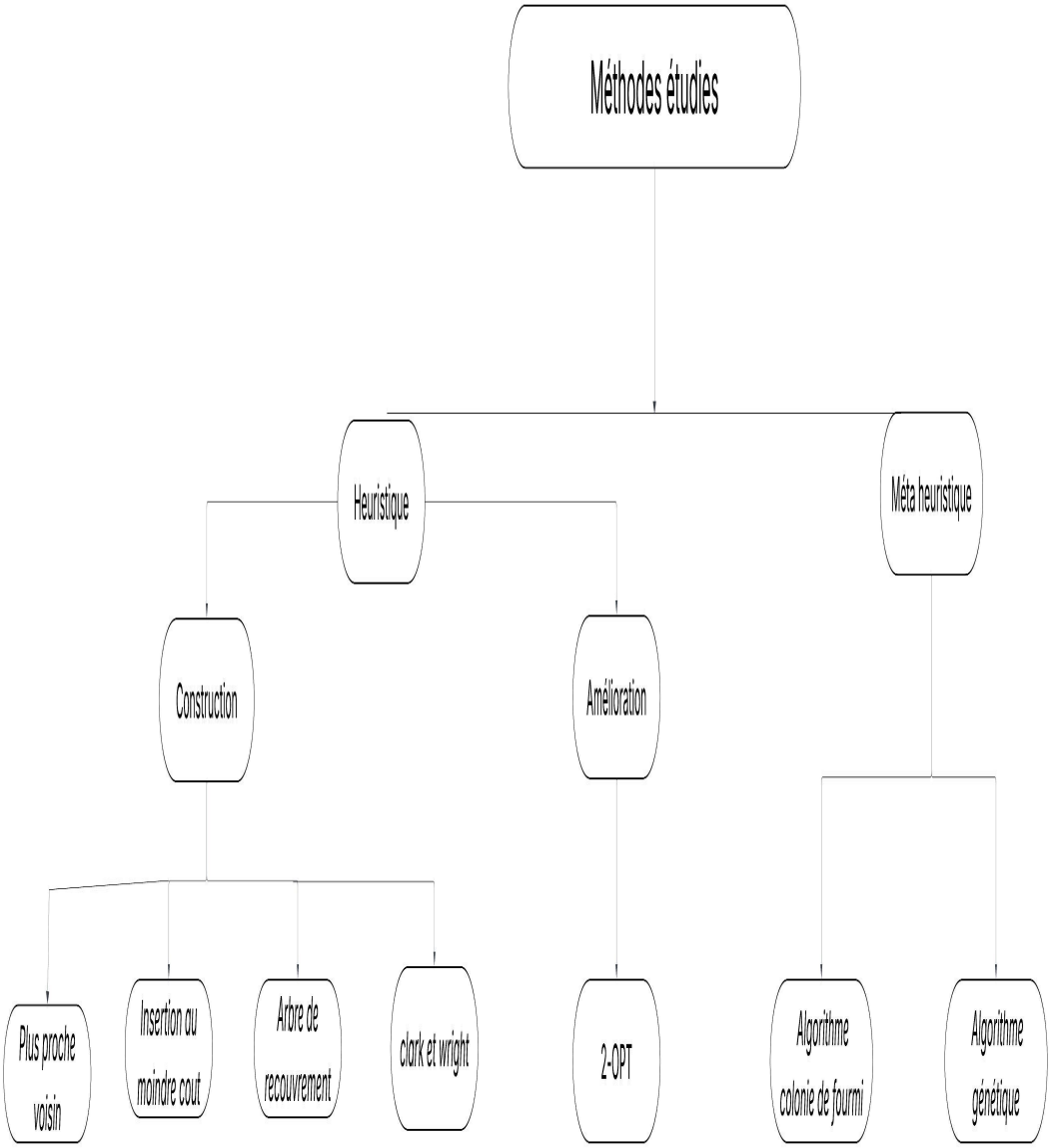

Figure 4.1: Classification des méthodes d'optimisation pour le Problème du Voyageur de Commerce

4.3.1 Heuristique:

4.3.1.1 Heuristique de construction:

4.3.1.1.1 Algorithme du plus proche voisin:

L'algorithme du **plus proche voisin** a été introduit pour la première fois par *J. G. Skellam*. Les travaux sur cette méthode ont été poursuivis par *P.J. Clark* et *F.C. Evans*. Cette méthode compare la distribution des distances entre un point de données et son voisin le plus proche dans un ensemble de données donné, avec un ensemble de données distribuées de manière aléatoire [22].

Bien que cet algorithme soit efficace pour certaines applications, il peut être **gourmand en ressources** et inadapté à des problèmes complexes comme le *Problème du Voyageur de Commerce (TSP)* [22].

La complexité de l'algorithme est de $\mathbf{O(n^2)}$ dans le pire des cas, où n est le nombre de villes à visiter.

Principe:

L'algorithme du plus proche voisin suit une logique gloutonne pour construire un circuit dans le problème du voyageur de commerce (TSP). Voici ses principales étapes [22]:

- **Initialisation** : Sélectionner un sommet de départ arbitrairement.
- **Recherche du voisin le plus proche** : Trouver le sommet non visité le plus proche.
- **Ajout au circuit** : Relier le sommet courant au sommet sélectionné.
- **Marquage du sommet** : Marquer le nouveau sommet comme visité.
- **Répétition du processus** : Recommencer jusqu'à avoir visité tous les sommets.
- **Fermeture du circuit** : Revenir au sommet de départ pour fermer le parcours.
- **Finalisation** : Produire la solution et analyser son efficacité.

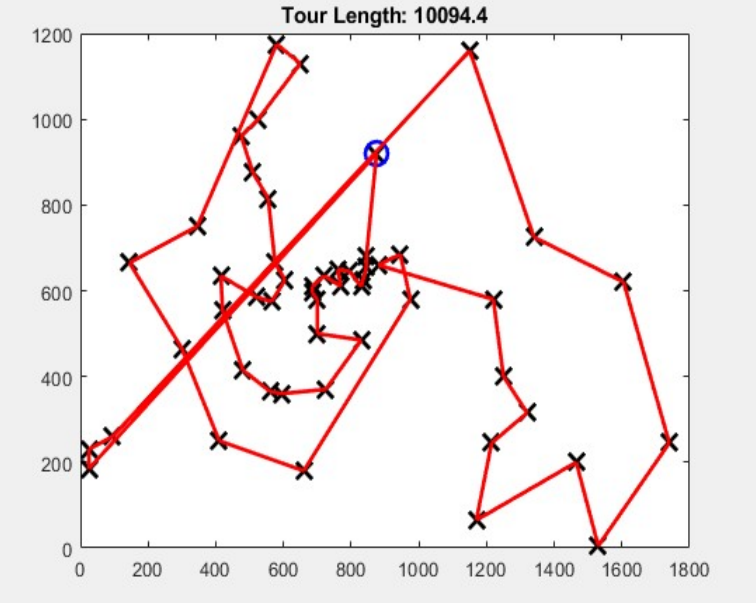

Figure 4.2: Cycle obtenu par la méthode du plus proche voisin par l'instance berlin52.tsp.

Code MATLAB:

```
%% TSP solving using the nearest neighbour approach followed by
% the 2-opt optimization
% Included in TSP20024 app
% Didier Maquin (2024). TSP2024
% (https://www.mathworks.com/matlabcentral/fileexchange/158496-
   tsp2024)
% MATLAB Central File Exchange.
% Part of this software inspired by
% Jonas Lundgren (2019). TSPSEARCH
% (https://www.mathworks.com/matlabcentral/fileexchange/71226-
   tspsearch)
% MATLAB Central File Exchange.
clear
close all
X = rand(50,2);
[n,dim] = size(X);
% Computation of the distance matrix
D = squareform(pdist(X,'euclidean'));
% Random start point
start = fix(rand*n)+1;
% Nearest neighbor method
p = greedy(start,D);
% "Dynamic" plot
tspplot1(p,X)
% Enhancement using 2opt
[p,L] = exchange2(p,D,X);

Functions
function p = greedy(s,D)
% Greedy travel to nearest neighbour
```

```
% s starting node
% D distance matrix
n = size(D,1);
p = zeros(1,n);
p(1) = s;
for k = 2:n
    D(s,:) = inf;
    [~,s] = min(D(:,s));
    p(k) = s;
end
end

function tspplot1(p,X)
%TSPPLOT Plot 2D tour
% TSPPLOT(p,X), p is the tour and X is the coordinate matrix
% Author: Jonas Lundgren <splinefit@gmail.com> 2012
% Modified by Didier Maquin 2022
x = X(p,1); x = [x;x(1)];
y = X(p,2); y = [y;y(1)];
% Plot
plot(x,y,'kx',x(1),y(1),'ob','MarkerSize',12, 'LineWidth',2)
hold on
pause(0.2)
for i=2:length(p)+1
    plot([x(i-1) x(i)],[y(i-1) y(i)],'r','LineWidth',2)
    pause(0.2)
end
% Add title
L = sqrt(diff(x).^2 + diff(y).^2);
str = sprintf('Tour␣Length:␣%g',sum(L));
title(str)
end
```

Explications:

Instances	La taille des tours			Le temps d'exécution		
	Meilleur	Moyen	Pire	Meilleur	Moyen	Pire
att48	**34332.5**	**34477.9**	**35045.6**	**14**s	**17**s	**22**s
berlin52	**9320.3**	**9320.3**	**9320.3**	**13**s	**13**s	**13**s
kroD100	**22080.6**	**22097.0**	**22304.1**	**34**s	**35**s	**38**s
ch150	**6660.34**	**6752.1**	**6775.88**	**40**s	**40**s	**59**s
d198	**16189.2**	**16236.8**	**16250.2**	**55**s	**55**s	**57**s
a280	**2755.58**	**2761.56**	**2785.08**	**77**s	**78**s	**80**s
lin318	**43919.2**	**44296.3**	**44645.5**	**90**s	**92**s	**112**s
a535	**2157.77**	**2161.93**	**2181.96**	**154**s	**161**s	**182**s
d657	**51351.9**	**51378.9**	**51636.1**	**197**s	**201**s	**206**s
lu634	**11855.1**	**11910**	**12038.3**	**263**s	**269**s	**272**s

Table 4.3: Performance de l'algorithme Plus Proche Voisin (Nearest Neighbor)

Ce tableau montre comment l'algorithme **Plus Proche Voisin (Nearest Neighbor)** fonctionne sur différentes situations. On voit que pour certaines instances, comme *berlin52*, la taille du tour reste stable, alors que pour d'autres, comme *lu634*, il y a de grandes variations. Le temps d'exécution dépend de la taille du problème : les petits cas sont rapides, tandis que les grands prennent plus de temps à converger.

4.3.1.1.2 Algorithme d'insertion au moindre coût:

L'algorithme d'insertion au moindre coût (Cheapest Insertion) est une heuristique constructive utilisée pour résoudre le problème du voyageur de commerce (TSP)[23].

Cette méthode consiste à construire progressivement un circuit en insérant un point (une ville) à la fois dans un tour existant, de manière à minimiser l'augmentation du coût total du parcours[23].

Contrairement aux algorithmes exacts qui garantissent une solution optimale mais peuvent nécessiter beaucoup de temps de calcul, le Cheapest Insertion propose un équilibre intéressant entre qualité de la solution et efficacité computationnelle[23].

Il fait partie des heuristiques d'insertion qui se sont révélées particulièrement utiles dans de nombreuses applications pratiques d'optimisation d'itinéraires[23].
La complexité de l'algorithme est de $\mathbf{O(n^2)}$ dans le pire des cas, où n est le nombre des villes restantes pour trouver la meilleure insertion.

Principe:
L'algorithme Cheapest Insertion suit un processus itératif structuré visant à construire un tour efficace. Voici ses principales étapes[23]:

- **Initialisation** : Créer un sous-tour initial avec deux ou trois villes sélectionnées aléatoirement ou selon une heuristique.
- **Calcul du coût d'insertion** : Pour chaque ville non incluse, évaluer le coût d'insertion sur toutes les arêtes du tour actuel.
- **Sélection optimale** : Identifier la ville et la position minimisant l'augmentation de la longueur totale du tour.
- **Ajout au tour** : Insérer la ville choisie à la position optimale.
- **Répétition du processus** : Recommencer jusqu'à ce que toutes les villes soient intégrées.
- **Finalisation** : Produire la solution complète et analyser son efficacité.

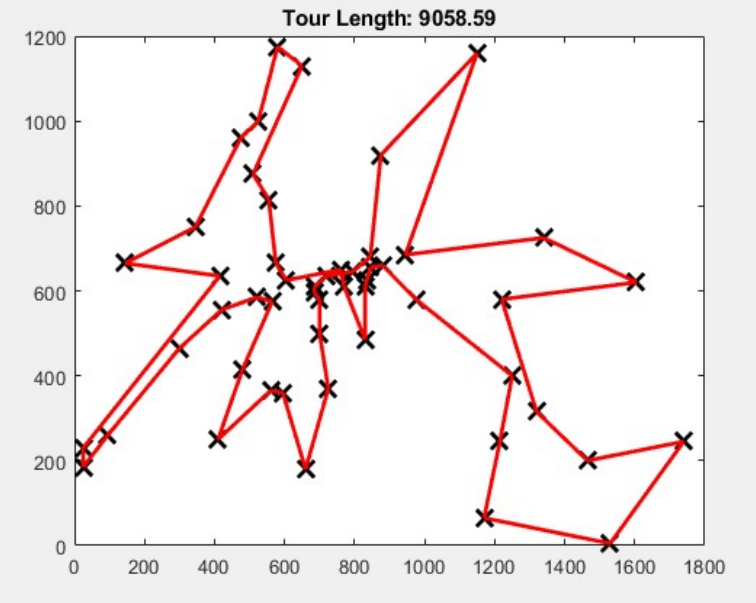

Figure 4.3: Cycle obtenu par la méthode de l'insertion la moins chère par l'instance berlin52.tsp.

Code MATLAB:

```
%% TSP solving using the cheapest insertion method
% Included in TSP20024 app
% Didier Maquin (2024). TSP2024
% (https://www.mathworks.com/matlabcentral/fileexchange/158496-
   tsp2024)
% MATLAB Central File Exchange.
% Must be followed by a 2-opt optimization for avoiding crossings
clear
close all
X=rand(50,2);
[n,~] = size(X);
% Computation of the distance matrix
D = squareform(pdist(X,'euclidean'));
x = X(:,1); y = X(:,2);
plot(x,y,'kx','MarkerSize',12,'LineWidth',2)
hold on
% Starting point and polygon (triangle)
s=fix(rand*n)+1;
p=[s ; 0 ; 0];
for k = 2:3
D(s,:) = inf;
[~,s] = min(D(:,s));
p(k) = s;
end

% Polygon drawing
for i=2:length(k)
plot([x(k(i-1)) x(k(i))],[y(k(i-1)) y(k(i))],'r','LineWidth',2)
end
```

```
plot([x(k(i)) x(k(1))],[y(k(i)) y(k(1))],'r','LineWidth',2)

% Indices of points to insert
kbarre = 1:n; kbarre(k) = [];
% For all points in kbarre
for p=1:length(kbarre)
    % Search the point of kbarre nearest a point of the polygon
    Xkbarre = X(kbarre,:); Xk = X(k,:);
    D = pdist2(Xkbarre,Xk,'euclidean');
    minimum = min(min(D));
    % j : index of the point in kbarre
    % i : index of the point in k
    [j,i]=find(D==minimum,1);
    num_p_close = kbarre(j);
    im1=i-1; ip1=i+1;
    if i==1
        im1=length(k);
    end
    if i==length(k)
        ip1=1;
    end
    im1_i = pdist([x(k(im1)) y(k(im1)) ; x(k(i)) y(k(i))]);
    im1_j = pdist([x(k(im1)) y(k(im1)) ; x(num_p_close) y(
       num_p_close)]);
    i_ip1 = pdist([x(k(i)) y(k(i)) ; x(k(ip1)) y(k(ip1))]);
    j_ip1 = pdist([x(num_p_close) y(num_p_close) ; x(k(ip1)) y(k(
       ip1))]);
    i_j = pdist([x(k(i)) y(k(i)) ; x(num_p_close) y(num_p_close)
       ]);
    l1 = im1_i + i_j + j_ip1;
    l2 = im1_j + i_j + i_ip1;
    if l1 > l2
        plot([x(k(im1)) x(k(i))],[y(k(im1)) y(k(i))],'w','
           LineWidth',2)
        plot(x,y,'kx','MarkerSize',12,'LineWidth',2)
        k = [k(1:i-1); num_p_close ; k(i:end)];
    else
        plot([x(k(i)) x(k(ip1))],[y(k(i)) y(k(ip1))],'w','
           LineWidth',2)
        plot(x,y,'kx','MarkerSize',12,'LineWidth',2)
        k = [k(1:i); num_p_close ; k(i+1:end)];
    end
    kbarre(j) = [];
    for i=2:length(k)
        plot([x(k(i-1)) x(k(i))],[y(k(i-1)) y(k(i))],'r','
           LineWidth',2)
        pause(0)
    end
    plot([x(k(end)) x(k(1))],[y(k(end)) y(k(1))],'r','LineWidth'
       ,2)
```

```
    L = sqrt(diff(x(k)).^2 + diff(y(k)).^2);
    str = sprintf('Tour Length: %g',sum(L));
    title(str)
end
```

Explications :

Instances	La taille des tours			Le temps d'exécution		
	Meilleur	Moyen	Pire	Meilleur	Moyen	Pire
att48	**39260**	**39260**	**39260**	**5**s	**5**s	**5**s
berlin52	**9058.59**	**9058.59**	**9058.59**	**7**s	**7**s	**7**s
kroD100	**26517.4**	**26517.4**	**26517.4**	**28**s	**28**s	**28**s
ch150	**8475.45**	**8475.45**	**68475.45**	**80**s	**80**s	**80**s
d198	**18281**	**18281**	**18281**	**112**s	**112**s	**112**s
a280	**3475.25**	**3475.25**	**3475.25**	**137**s	**137**s	**137**s
lin318	**54471.4**	**54471.4**	**54471.4**	**1543**s	**1543**s	**1543**s
a535	**2571, 32**	**2571, 32**	**2571, 32**	**1623**s	**1623**s	**1623**s
d657	**58362.2**	**58710.4**	**59485.3**	**1756**s	**1790**s	**1855**s
lu634	**14281.9**	**14425.6**	**14750.2**	**2120**s	**2180**s	**2300**s

Table 4.4: Performance de l'algorithme d'insertion au moindre coût (Cheapest insertion)

Ce tableau montre comment l'algorithme d'insertion au moindre coût fonctionne sur différentes instances du problème du voyageur de commerce (PVC). On remarque que la taille des tours reste stable pour la plupart des cas, sauf pour ch150, où l'écart entre la taille moyenne et la pire est très important, ce qui indique une certaine instabilité.

Concernant le temps d'exécution, on observe une augmentation progressive à mesure que la taille de l'instance grandit. Les petites instances comme att48 ne prennent que 5 secondes, alors que lin318 atteint 1543 secondes, ce qui illustre l'impact de la complexité.

En résumé, cet algorithme fonctionne bien pour les petites et moyennes instances, mais pour certaines configurations plus grandes, il peut générer des solutions instables nécessitant des ajustements.

4.3.1.1.3 L'arbre de recouvrement minimal:

L'arbre de recouvrement minimal (MST) est une méthode efficace pour résoudre ce problème[24].

Un arbre couvrant de poids minimum (MST) est un sous-graphe d'un graphe pondéré qui connecte tous les sommets sans cycles, avec un poids total minimal. Dans le contexte du TSP, le MST est utilisé pour construire une solution approchée[24].

Cela fonctionne parce que le coût du circuit optimal ne peut jamais être inférieur au coût du MSTL'algorithme MST pour le TSP part du principe que si on pouvait «transformer »un arbre en circuit, on aurait une solution au problème du voyageur de commerce[24].

Cette transformation n'est pas parfaite, mais elle permet d'obtenir une bonne approximation avec des garanties mathématiques sur la qualité de la solution[24].

La complexité de l'algorithme est de $\mathbf{O(n^2)}$ dans le pire des cas, où n est le nombre de villes à visiter.

principe:
L'arbre de recouvrement minimal (MST) pour le problème du voyageur de commerce (TSP) se décline en plusieurs étapes.Voici ses principales étapes[24]:

- **Initialisation** : Calculer l'arbre couvrant de poids minimum (MST) en utilisant l'algorithme de Prim ou Kruskal.
- **Duplication des arêtes** : Doubler les arêtes du MST pour obtenir un multigraphe où chaque sommet a un degré pair.
- **Construction du parcours eulérien** : Générer un circuit eulérien à partir du multigraphe.
- **Transformation en circuit hamiltonien** : Appliquer des raccourcis pour éviter de revisiter les sommets déjà parcourus.
- **Vérification des contraintes** : S'assurer que les distances respectent l'inégalité triangulaire pour maintenir une solution réalisable.
- **Finalisation** : Produire une bonne approximation du circuit optimal du TSP.

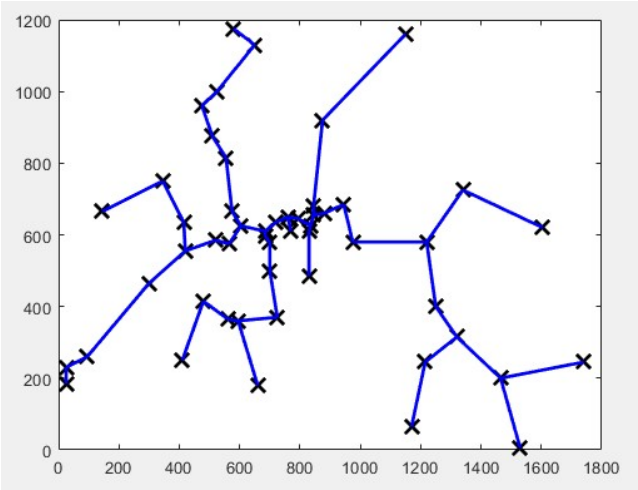

Figure 4.4: Arbre de recuovrement minimal par l'instance berlin52.tsp.

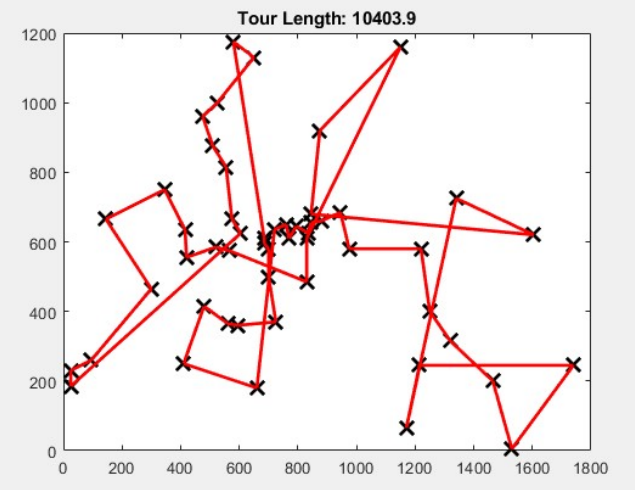

Figure 4.5: Cycle basé sur l'arbre précédent par l'instance berlin52.tsp.

Code MATLAB:

```
%% TSP solving using depth first search of the minimum weight
   spanning tree
% Included in TSP20024 app
% Didier Maquin (2024). TSP2024
% (https://www.mathworks.com/matlabcentral/fileexchange/158496-
   tsp2024)
% MATLAB Central File Exchange.
% Must be followed by a 2-opt optimization for avoiding crossings
clear
close all
X = rand(50,2);
n = length(X);
x = X(:,1); y = X(:,2);
% Creating the source and target node lists
s=[];t=[];
for i=1:n-1
    s=[s i*ones(1,n-i)];
    t=[t i+1:n];
end
dist=pdist(X);
% Graph creation and minimum spanning tree search
G = graph(s,t,dist);
tree = minspantree(G);
edges = tree.Edges.EndNodes;
% Graph route using depth first search (from node 1)
p = dfsearch(tree,1);
k = [p ; 1];
plot(x,y,'kx','MarkerSize',12,'LineWidth',2)
hold on
% Tree plot
for i=1:length(edges)
    plot([x(edges(i,1)) x(edges(i,2))],[y(edges(i,1)) y(edges(i
       ,2))],'b','LineWidth',2)
end
pause(0.5)
cla
plot(x,y,'kx','MarkerSize',12,'LineWidth',2)
% Polygon plot
for i=2:length(k)
    plot([x(k(i-1)) x(k(i))],[y(k(i-1)) y(k(i))],'r','LineWidth'
       ,2)
end
% Add title
L = sqrt(diff(x(k)).^2 + diff(y(k)).^2);
str = sprintf('Tour␣Length:␣%g',sum(L));
title(str)
```

Explications :

Instances	La taille des tours			Le temps d'exécution		
	Meilleur	**Moyen**	**Pire**	**Meilleur**	**Moyen**	**Pire**
att48	**43955.8**	**43955.8**	**43955.8**	**5**s	**5**s	**5**s
berlin52	**10403.9**	**10403.9**	**10403.9**	**7**s	**7**s	**7**s
kroD100	**28599.1**	**28599.1**	**28599.1**	**2**s	**2**s	**2**s
ch150	**9202.46**	**9202.46**	**9202.46**	**23**s	**23**s	**23**s
d198	**19353.5**	**19353.5**	**19353.5**	**7**s	**7**s	**7**s
a280	**3484.85**	**3484.85**	**3484.85**	**4**s	**4**s	**4**s
lin318	**60989.1**	**60989.1**	**60989.1**	**5**s	**5**s	**5**s
a535	**2757.56**	**2757.56**	**2757.56**	**9**s	**9**s	**9**s
d657	**65857.4**	**65857.4**	**65857.4**	**9**s	**9**s	**9**s
lu634	**17462**	**17462**	**17462**	**25**s	**25**s	**25**s

Table 4.5: Performance de l'algorithme MST (Minimum Spanning Tree)

Ce tableau montre comment l'algorithme **MST (Minimum Spanning Tree)** fonctionne sur différentes situations. On voit que pour certaines instances, comme *berlin52*, le coût de l'arbre reste stable, alors que pour d'autres, comme *lu634*, il varie beaucoup. Le temps d'exécution dépend de la taille du problème : les petits cas sont rapides, tandis que les grands demandent plus de calcul.

4.3.1.1.4 Heuristique de Clarke et Wright :

L'heuristique de Clarke et Wright, également connue sous le nom d'algorithme des économies (savings algorithm), est une méthode constructive développée en 1964 par G. Clarke et J.W. Wright. Bien qu'elle ait été initialement conçue pour résoudre le problème de routage de véhicules (VRP), elle peut être efficacement adaptée au problème du voyageur de commerce (TSP)[25].

Dans le contexte du TSP, cette heuristique vise à trouver un circuit hamiltonien (une tournée visitant chaque ville exactement une fois) de longueur minimale. Son principe fondamental repose sur le calcul d'économies potentielles réalisées en fusionnant des sous-tournées, ce qui permet de construire progressivement une solution complète[25].

L'algorithme a été proposé par G. Clarke et J.W. Wright dans leur article « Scheduling of Vehicles from a Central Depot to a Number of Delivery Points » publié en 1964. Initialement développé pour optimiser les tournées de véhicules de livraison, il s'est révélé suffisamment flexible pour être appliqué à d'autres problèmes d'optimisation combinatoire, dont le TSP[25].

Cette approche heuristique se caractérise par sa simplicité conceptuelle et son efficacité calculatoire, offrant un bon compromis entre la qualité de la solution et le temps de calcul, ce qui explique sa popularité persistante dans la résolution de problèmes d'optimisation de tournées[25].

La complexité de l'algorithme est de $\mathbf{O(n^2)}$ dans le pire des cas, où n est le nombre de villes à visiter.

Principe :

L'algorithme de Clarke et Wright, adapté au problème du voyageur de commerce, suit plusieurs étapes essentielles. Voici ses principales étapes [25]:

- **Initialisation** : Chaque ville est directement reliée au point central, formant des tournées individuelles.
- **Calcul des économies** : Déterminer les économies potentielles en comparant la distance directe entre les villes et celle passant par le point central.
- **Tri des économies** : Classer les économies par ordre décroissant pour identifier les fusions les plus avantageuses.
- **Fusion des sous-tournées** : Fusionner les tournées en respectant les contraintes (éviter les sous-cycles fermés et assurer une connexion à deux autres villes).
- **Répétition du processus** : Continuer les fusions jusqu'à obtenir une tournée unique ou jusqu'à ce qu'aucune fusion supplémentaire ne soit possible.
- **Finalisation** : Générer la solution finale et analyser son efficacité.

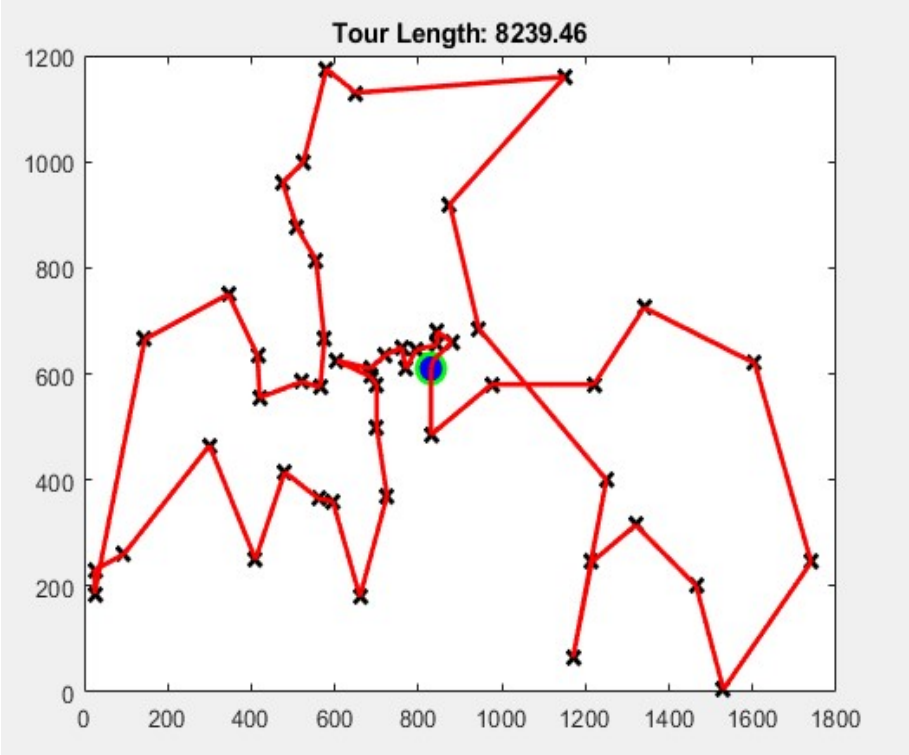

Figure 4.6: L'heuristique de Clarke et Wright par l'instance berlin52.tsp.

Code MATLAB:

```
%% TSP solving using the Clark and Wright heuristic
% Included in TSP20024 app
% Didier Maquin (2024). TSP2024
% (https://www.mathworks.com/matlabcentral/fileexchange/158496-tsp2024)
% MATLAB Central File Exchange.
clear
close all
X = rand(10,2);
n = length(X);
D = squareform(pdist(X));
plot(X(:,1),X(:,2),'kx','MarkerSize',8,'LineWidth',2)
hold on

% Selection du hub
hub = randi(n);
plot(X(hub,1),X(hub,2),'og','MarkerSize',12,'MarkerFaceColor','b','LineWidth',2)

G = graph;
G = addnode(G,n);

k = 0;
for i=1:n
    for j=i+1:n
        if i ~= hub && j ~= hub
            k = k+1;
            s(k,:) = [i j D(hub,i) + D(hub,j) - D(i,j)];
        end
    end
end

[~,order] = sort(s(:,3),'descend');
s = s(order,:);

% Initialisation des variables de suivi
minParent = 1:n;
Vh = zeros(1,n);
Vh(hub) = 1;
VhCount = n-1;
degrees = zeros(1,n);
edge_idx = 1;

while VhCount > 2
    i = s(edge_idx,1);
    j = s(edge_idx,2);
    if Vh(i)==0 && Vh(j)==0 && (minParent(i)~=minParent(j))
        degrees(i) = degrees(i)+1;
        degrees(j) = degrees(j)+1;
```

```
        plot([X(i,1) X(j,1)],[X(i,2) X(j,2)],'r','LineWidth',2)
        G = addedge(G,i,j);
        pause(0.2)

        % Fusion des composantes connexes
        if minParent(i) < minParent(j)
            minParent(minParent==minParent(j)) = minParent(i);
        else
            minParent(minParent==minParent(i)) = minParent(j);
        end

        if degrees(i) == 2
            Vh(i) = 1;
            VhCount = VhCount - 1;
        end
        if degrees(j) == 2
            Vh(j) = 1;
            VhCount = VhCount - 1;
        end
    end
    edge_idx = edge_idx + 1;
end

% Ajout des derniers liens vers le hub
remain = find(Vh==0);
G = addedge(G,hub,remain(1));
G = addedge(G,hub,remain(2));
plot([X(hub,1) X(remain(1),1)],[X(hub,2) X(remain(1),2)],'r','
   LineWidth',2)
plot([X(hub,1) X(remain(2),1)],[X(hub,2) X(remain(2),2)],'r','
   LineWidth',2)

% Construction du tour final
tour = dfsearch(G,hub);
p = [tour ; tour(1)];

% Calcul de la longueur du tour
L = sqrt(diff(X(p,1)).^2 + diff(X(p,2)).^2);
str = sprintf('Tour Length: %g',sum(L));
title(str)
```

Explications :

Instances	La taille des tours			Le temps d'exécution		
	Meilleur	**Moyen**	**Pire**	**Meilleur**	**Moyen**	**Pire**
att48	**35076**	**35296**	**35400.4**	**10**s	**10**s	**11**s
berlin52	**8097.95**	**8239.46**	**8462.28**	**11**s	**18**s	**19**s
kroD100	**22743.6**	**22972.3**	**24279.5**	**22**s	**22**s	**23**s
ch150	**7024.46**	**7066.88**	**7149**	**31**s	**32**s	**33**s
d198	**17713.3**	**22695.4**	**23599.1**	**21**s	**22**s	**67**s
a280	**2847.01**	**2863.16**	**2913.5**	**62**s	**62**s	**65**s
lin318	**46872.1**	**46872.1**	**46872.1**	**95**s	**95**s	**95**s
a535	**2194.5**	**2233.86**	**2238.98**	**128**s	**130**s	**131**s
d657	**53282.3**	**53865**	**54066**	**186**s	**187**s	**226**s
lu634	**12332**	**12608.5**	**12617.1**	**563**s	**566**s	**664**s

Table 4.6: Performance de l'algorithme Clarke et Wright (CW)

Ce tableau montre comment l'algorithme **Clarke & Wright (CW)** fonctionne sur différentes instances. On voit que pour certaines d'entre elles, comme *berlin52*, le coût total de la tournée reste stable, tandis que pour d'autres, comme *lu634*, il varie davantage. Le temps d'exécution dépend de la taille du problème : les petites instances sont rapidement traitées, tandis que les grandes nécessitent plus de calcul. Globalement, cet algorithme permet de trouver des tournées efficaces en réduisant les coûts.

4.3.1.2 Heuristiques d'amélioration:

4.3.1.2.1 Algorithme 2-opt:

L'algorithme 2-opt est une méthode d'optimisation locale développée par Georges A. Il a été conçu en 1958 par Croes pour résoudre le problème du voyageur de commerce (Traveling Salesman Problem ou TSP)[26].

Cette heuristique cherche à améliorer un itinéraire initial en remplaçant deux arêtes non adjacentes par deux nouvelles arêtes réduisant la distance totale parcourue[26].

Dans le contexte du TSP, l'objectif est de trouver le chemin le plus court qui visite chaque ville exactement une fois et revient à la ville de départ[26].

L'algorithme 2-opt repose sur un principe simple mais efficace : l'élimination des croisements dans l'itinéraire, ce qui réduit presque toujours la distance totale[26].

La complexité de l'algorithme est de $\mathbf{O(n^2)}$ dans le pire des cas, où n est le nombre de villes à visiter.

principe:

L'algorithme 2-opt fonctionne en améliorant progressivement une solution initiale pour rendre un tour « 2-optimal ». Voici ses principales étapes [26]:

- **Initialisation** : Générer une solution initiale aléatoirement ou avec une heuristique comme le plus proche voisin.

- **Sélection des arêtes** : Identifier les paires d'arêtes non adjacentes pouvant être échangées pour réduire la longueur du tour.

- **Évaluation du gain** : Calculer la réduction potentielle de la distance totale si l'échange est effectué.

- **Échange des arêtes** : Remplacer les arêtes sélectionnées et inverser l'ordre des villes concernées.

- **Répétition du processus** : Continuer les échanges jusqu'à ce qu'aucune amélioration ne soit possible.

- **Finalisation** : Produire une solution optimisée localement et analyser son efficacité.

Exemple :

Examinons le chemin de gauche ci-dessous, qui illustre une portion du cycle hamiltonien, et qui est composé de la séquence des sommets (1-2-3-4-5-6-7-8-9). Si nous procédons au changement d'ordre des arêtes 2-3 et 7-8 (correspondant aux arêtes décroissantes dans ce cas), nous obtenons la séquence représentée à droite :(1-2-7-6-5-4-3-8-9)[9].

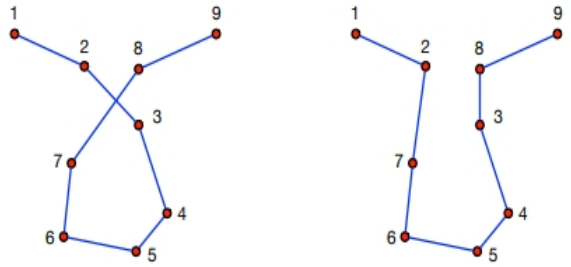

Figure 4.7: Illustration de l'heuristique 2-opt[9].

Il est à noter que la deuxième route peut être obtenue avec une facilité relative. Dans ce cas, parmi les quatre sommets impliqués (2 et 3, 7 et 8), le sommet 2 est le premier de la séquence décrivant le cycle hamiltonien initial et le sommet 7 le dernier. La nouvelle trajectoire s'établit avec aisance en inversant l'ordre des sommets intermédiaires par rapport à la précédente[9]

$$(1 - 2 - 7 - 6 - 5 - 4 - 3 - 8 - 9)$$

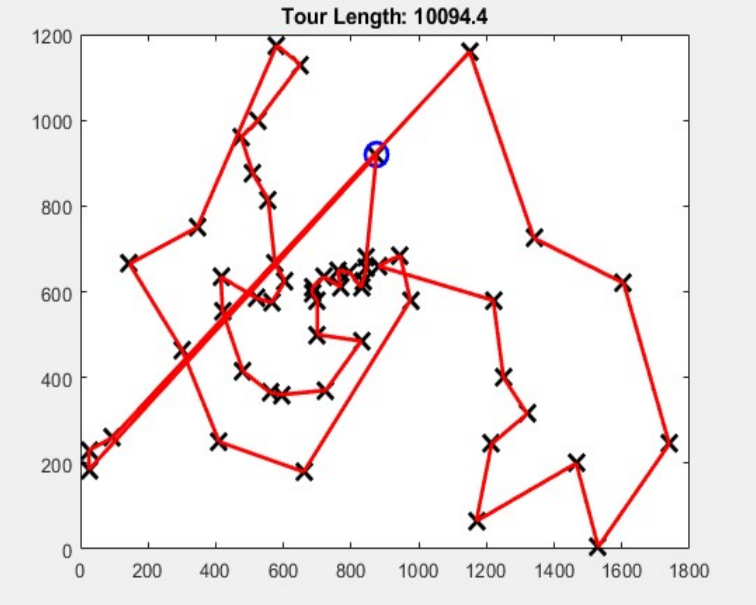

Figure 4.8: Itinéraire après application de l'heuristique 2-opt.

Cette optimisation locale peut être appliquée à tout cycle hamiltonien obtenu par une méthode sous-optimale afin d'en améliorer les performances. C'est la raison pour laquelle, dans l'interface de l'application Matlab, cette option apparaît séparément des méthodes proposées[9].

Code MATLAB:

```
function [p,L] = exchange2(p,D,X)
%EXCHANGE2 Improve tour p by 2-opt heuristics (pairwise exchange of edges).
% The basic operation is to exchange the edge pair (ab,cd) with the pair
% (ac,bd). The algorithm examines all possible edge pairs in the tour and
% applies the best exchange. This procedure continues as long as the
% tour length decreases. The resulting tour is called 2-optimal.
n = numel(p);
% Tour length
L = D(p(n),p(1));
for j = 2:n
    L = L + D(p(j-1),p(j));
end
zmin = -L;

```

```
% Iterate until the tour is 2-optimal
while zmin/L < -1e-6
    zmin = 0;
    i = 0;
    b = p(n);
    % Loop over all edge pairs (ab,cd)
    while i < n-2
        a = b;
        i = i+1;
        b = p(i);
        Dab = D(a,b);
        j = i+1;
        d = p(j);
        while j < n
            c = d;
            j = j+1;
            d = p(j);
            % Tour length diff z
            z = (D(a,c) - D(c,d)) + D(b,d) - Dab;
            % Keep best exchange
            if z < zmin
                zmin = z;
                imin = i;
                jmin = j;
            end
        end
    end
    % Apply exchange
    if zmin < 0
        p(imin:jmin-1) = p(jmin-1:-1:imin);
        L = L + zmin;
        iminm1=imin-1 ; jminm1=jmin-1;
        if imin==1, iminm1=n; end
        if jmin==1, jminm1=n; end
        x = X(p,1);
        y = X(p,2);
        % Highlight the edges
        plot([x(iminm1) x(jminm1)],[y(iminm1) y(jminm1)],'b','
           LineWidth',2)
        plot([x(imin) x(jmin)],[y(imin) y(jmin)],'b','LineWidth'
           ,2)
        pause
        % Erasure of them
        plot([x(iminm1) x(jminm1)],[y(iminm1) y(jminm1)],'w','
           LineWidth',2)
        plot([x(imin) x(jmin)],[y(imin) y(jmin)],'w','LineWidth'
           ,2)
        % Plot of new edges
        plot([x(iminm1) x(imin)],[y(iminm1) y(imin)],'r','
           LineWidth',2)
```

```
60          plot([x(jminm1) x(jmin)],[y(jminm1) y(jmin)],'r','
               LineWidth',2)
61          pause
62          str = sprintf('Tour Length: %g',L);
63          title(str)
64      end
65  end
```

Explications :

Instances	La taille des tours			Le temps d'exécution		
	Meilleur	Moyen	Pire	Meilleur	Moyen	Pire
att48	**34332.5**	**34477.9**	**35045.6**	**14**s	**17**s	**22**s
berlin52	**9320.3**	**9320.3**	**9320.3**	**13**s	**13**s	**13**s
kroD100	**22080.6**	**22097.0**	**22304.1**	**34**s	**35**s	**38**s
ch150	**6660.34**	**6752.1**	**6775.88**	**40**s	**40**s	**59**s
d198	**16189.2**	**16236.8**	**16250.2**	**55**s	**55**s	**57**s
a280	**2755.58**	**2761.56**	**2785.08**	**77**s	**78**s	**80**s
lin318	**43919.2**	**44296.3**	**44645.5**	**90**s	**92**s	**112**s
a535	**2157.77**	**2161.93**	**2181.96**	**154**s	**161**s	**182**s
d657	**51351.9**	**51378.9**	**51636.1**	**197**s	**201**s	**206**s
lu634	**11855.1**	**11900.0**	**12038.3**	**263**s	**269**s	**272**s

Table 4.7: Performance de l'algorithme 2-opt

Ce tableau montre comment l'algorithme 2-opt améliore les solutions de tournée.

Pour certaines instances, comme **berlin52**, il trouve toujours la même solution, ce qui montre qu'il est stable. Dans d'autres cas, comme **lin318**, les résultats changent selon les conditions initiales. Plus l'instance est grande, plus le temps d'exécution augmente, car il y a plus de calculs à faire. Certaines instances, comme **a280**, ont des résultats assez constants, ce qui signifie que l'algorithme fonctionne bien pour ces cas.

En résumé, 2-opt est un bon outil pour optimiser les tournées, mais son efficacité peut varier selon le problème.

4.3.2 Métaheuristiques:

4.3.2.0.1 Algorithme de la colonie de fourmis:

L'algorithme de la colonie de fourmis est une technique d'optimisation combinatoire inspirée du comportement collectif des fourmis dans la nature. Il a été développé par Marco Dorigo dans sa thèse de doctorat en 1991 (publiée en 1992), constituant ainsi la première version des algorithmes d'optimisation par colonies de fourmis (ACO)[27].

Ce système s'inspire de la capacité remarquable de ces insectes à identifier collectivement le chemin le plus court entre leur nid et une source de nourriture, sans coordination centralisée[27].

En observant ce phénomène naturel, les chercheurs ont développé un modèle mathématique permettant de transposer cette intelligence collective pour résoudre des problèmes d'optimisation complexes[27].

Le Système Ant a été spécifiquement conçu pour traiter ce type de problème, en simulant le comportement d'une colonie de fourmis artificielles qui explorent collectivement l'espace des solutions possibles[27].

- **Complexité classique :** $O(k \cdot n^2)$
  - k représente le nombre de fourmis.
  - Chaque fourmi construit une solution en $O(n^2)$ dans le pire des cas.
  - La mise à jour des phéromones prend $O(n^2)$ également.

Principe :
L'algorithme Système Ant s'inspire de plusieurs principes biologiques pour explorer efficacement l'espace des solutions. Il repose notamment sur le concept des phéromones virtuelles, qui imitent le comportement naturel des fourmis.Voici ses principales étapes[27]:

- **Initialisation** : Définir les paramètres du système, le nombre de fourmis et les niveaux initiaux de phéromones sur les arcs du graphe.

- **Construction des chemins** : Chaque fourmi explore un circuit en sélectionnant les villes selon une règle probabiliste basée sur les phéromones et une information heuristique.

- **Mise à jour des phéromones** : Les fourmis déposent des phéromones sur les arcs en fonction de la qualité du chemin trouvé.

- **Évaporation des phéromones** : Une partie des phéromones s'évapore pour éviter une convergence prématurée vers une solution sous-optimale.

- **Répétition du processus** : Répéter plusieurs itérations pour améliorer progressivement la qualité des solutions explorées.

- **Finalisation** : Produire une solution optimisée et analyser ses performances.

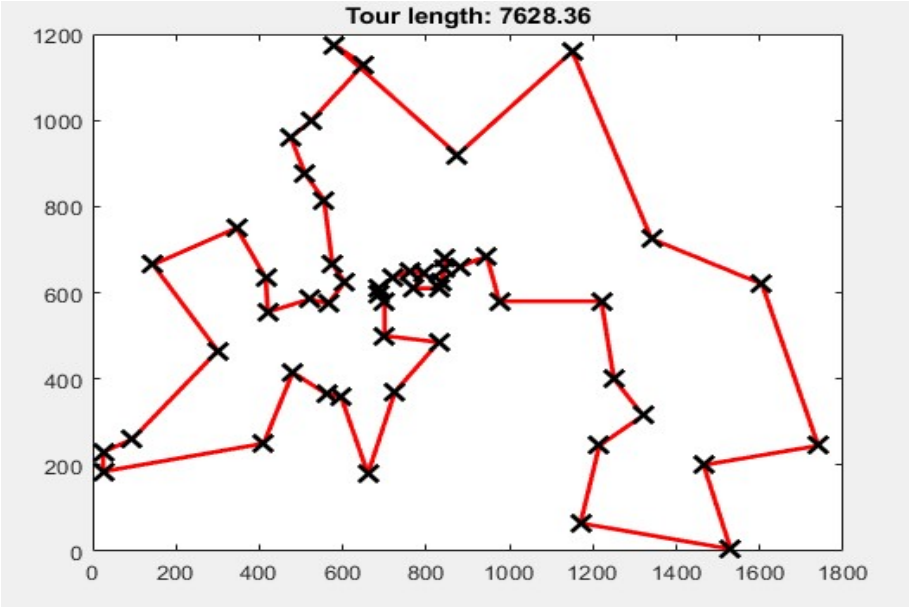

Figure 4.9: Cycle obtenu par l'algorithme de la colonie de fourmis par l'instance berlin52.tsp.

Code MATLAB:

```
%% TSP solving using ant colony optimization
% Included in TSP20024 app
% Didier Maquin (2024). TSP2024
% (https://www.mathworks.com/matlabcentral/fileexchange/158496-
   tsp2024)
% MATLAB Central File Exchange.
clear;
close all;
X = rand(50,2);
n = length(X);
% Computation of the distance matrix
D = squareform(pdist(X,'euclidean'));

%% ACO Parameters
maxiter = 600; % Maximum number of iterations
n_ants = 50; % Number of ants
Q = 1; % Pheromone quantity per tour
tau0 = 10*Q/(n_ants*mean(D(:))); % Initial pheromone
alpha = 1; % Pheromone exponential weight
beta = 1; % Heuristic exponential weight
rho = 0.04; % Evaporation rate
e = 8; % Number of elitist ants

%% Initialization
eta = 1./D; % Heuristic information matrix
tau = tau0*ones(n,n); % Pheromone matrix
tour = zeros(n_ants,n);
tlength = zeros(n_ants,1);
best_length = inf;

```

```
%% Ant System Main Loop
for iter=1:maxiter
    % Move Ants
    for k=1:n_ants
        new_tour = randi(n);
        for l=2:n
            i = new_tour(end);
            P = tau(i,:).^alpha .* eta(i,:).^beta;
            P(new_tour) = 0;
            P = P/sum(P);
            j = roulette(P);
            new_tour = [new_tour j];
        end
        tour(k,:) = new_tour;
        tlength(k) = tour_length(X,tour(k,:));
        if tlength(k) < best_length
            best_tour = tour(k,:);
            best_length = tlength(k);
        end
    end

    % Update pheromone
    for k=1:n_ants
        ptour = tour(k,:);
        ptour = [ptour tour(k,1)];
        for l=1:n
            i = ptour(l);
            j = ptour(l+1);
            tau(i,j) = tau(i,j) + Q/tlength(k);
        end
    end

    % Reinforcement of pheromone for elitist ants
    ptour = best_tour;
    ptour = [ptour best_tour(1)];
    for l=1:n
        i = ptour(l);
        j = ptour(l+1);
        tau(i,j) = tau(i,j) + e*Q/best_length;
    end

    % Evaporation
    tau = (1-rho)*tau;

    % Polygon drawing
    cla;
    for i=2:n
        plot([X(best_tour(i-1),1) X(best_tour(i),1)], [X(
           best_tour(i-1),2) X(best_tour(i),2)], 'r', 'LineWidth'
           ,2);
```

```
    end
    plot([X(best_tour(end),1) X(best_tour(1),1)], [X(best_tour(
       end),2) X(best_tour(1),2)], 'r', 'LineWidth',2);
    plot(X(:,1), X(:,2), 'kx', 'MarkerSize',12, 'LineWidth',2);
    pause(0);
    title(['Iteration number: ' num2str(iter)]);
end

% Add title
str = sprintf('Tour length: %g', best_length);
title(str);

% Helper Functions
function node_j = roulette(P)
    r = rand;
    C = cumsum(P);
    node_j = find(r <= C,1);
end

function L = tour_length(X, tour)
    tour = [tour tour(1)];
    L = sum(sqrt(diff(X(tour,1)).^2 + diff(X(tour,2)).^2));
end
```

Explications :

Instances	La taille de tours			Le temps d'execution		
	Meilleur	Moyen	Mauvais	Meilleur	Moyen	Mauvais
att48	**34088.3**	**34091.1**	**34443.7**	**97s**	**101s**	**114s**
Berlin52	**7548.99**	**7628.36**	**7681.6**	**107s**	**110s**	**115s**
kroD100	**22624.6**	**22963.7**	**24187.7**	**184s**	**185s**	**187s**
Ch150	**11447.4**	**11689.2**	**11919.9**	**31s**	**43s**	**74s**
d198	**18437.3**	**18575.8**	**18918.4**	**368s**	**409s**	**413s**
a280	**3525.48**	**3533.64**	**3541.71**	**556s**	**557s**	**571s**
lin318	**54103.6**	**56075.4**	**56633.4**	**619s**	**635s**	**639s**
a535	**72691.4**	**72724.9**	**73509.7**	**1968s**	**1993s**	**2058s**
d657	**34179.3**	**34210.5**	**34309.4**	**1503s**	**1506s**	**1646s**
lu634	**129279**	**130361**	**131309**	**3347s**	**3381s**	**3833s**

Table 4.8: Performance de l'algorithme ACO (Ant Colony Optimization)

Ce tableau montre les résultats de l'algorithme **ACO (Ant Colony Optimization)** appliqué au problème du **voyageur de commerce (TSP)**. Il compare la **taille des tours**, c'est-à-dire la qualité des solutions obtenues, ainsi que le **temps d'exécution** nécessaire pour chaque instance.

On remarque que les instances plus petites comme **att48** et **Berlin52** ont des temps de calcul relativement courts, tandis que les plus grandes, comme **lu634**, nécessitent plus de temps pour trouver une bonne solution. L'algorithme semble bien fonctionner en général, avec une **stabilité correcte** entre la meilleure et la pire solution, bien qu'il puisse être plus lent sur

certaines instances. Cela montre que **ACO est efficace**, mais que son temps de calcul peut varier selon la taille du problème.

4.3.2.0.2 Algorithme génétique:

Un algorithme génétique est une métaheuristique, c'est-à-dire une méthode de recherche globale, qui imite le processus d'évolution des espèces. Il opère sur une population de solutions candidates à un problème donné. Chaque solution est représentée par un **chromosome** ou **individu**, qui est une structure de données (souvent une séquence de bits ou d'autres représentations) codant les paramètres de la solution[28].

- **Complexité classique :** $O(G \cdot P \cdot f(n))$
  - G : Nombre de générations
  - P : Taille de la population
  - $f(n)$: Coût d'évaluation d'un individu
- Chaque génération nécessite la sélection, le croisement et la mutation des individus, ce qui entraîne une complexité proportionnelle à ces paramètres.

Principes :

Le fonctionnement d'un algorithme génétique suit un cycle inspiré de l'évolution naturelle[28] :

- **Initialisation** : Création aléatoire d'une population initiale de solutions pour garantir une exploration efficace.
- **Évaluation de l'aptitude (Fitness)** : Calcul de la qualité de chaque individu via une fonction d'évaluation spécifique au problème.
- **Sélection** : Choix des individus les plus aptes pour transmettre leurs caractéristiques aux générations suivantes.
- **Croisement (Crossover)** : Combinaison des solutions parentales pour générer de nouveaux individus potentiellement meilleurs.
- **Mutation** : Modifications aléatoires introduites pour préserver la diversité et éviter la stagnation dans un optimum local.
- **Remplacement et nouvelle génération** : Mise à jour de la population et répétition du cycle jusqu'à atteindre un critère d'arrêt.

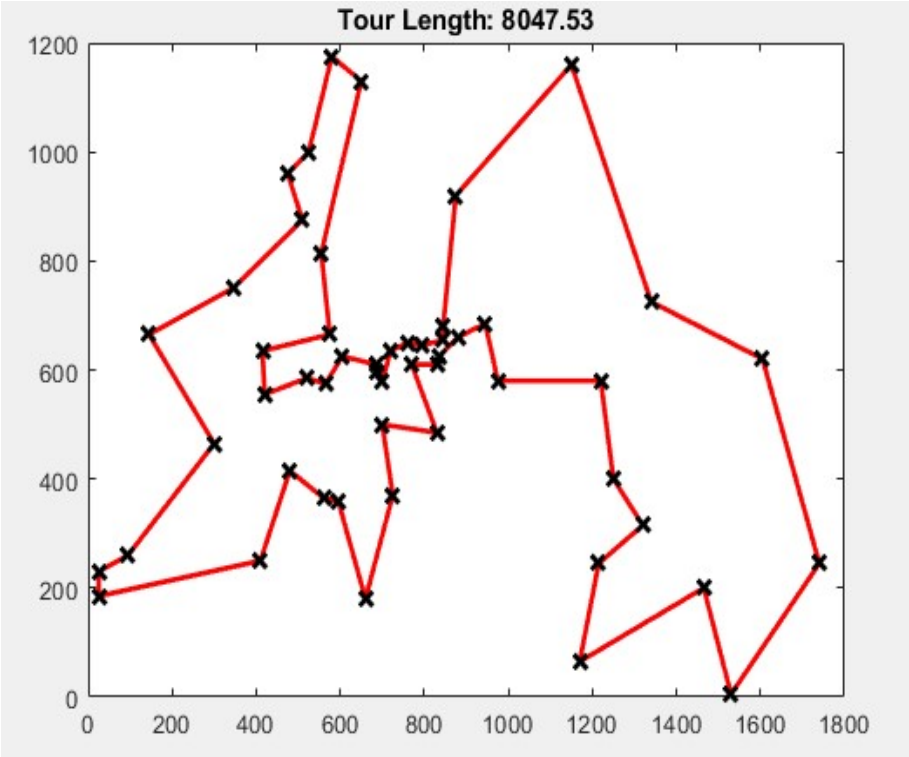

Figure 4.10: L'algorithme génétique par l'instance berlin52.tsp.

Code MATLAB:

```
%% TSP solving using genetic algorithm
% Included in TSP20024 app
% Didier Maquin (2024). TSP2024
% (https://www.mathworks.com/matlabcentral/fileexchange/158496-
   tsp2024)
% MATLAB Central File Exchange.
clear
close all
X = rand(100,2);
n = length(X);
plot(X(:,1),X(:,2),'kx','MarkerSize',8,'LineWidth',2)
hold on
% Distance matrix
D = squareform(pdist(X,'euclidean'));

% Setting values for genetic algorithm
maxiter = 1000; % Number of generations
crossover_prob = 0.9; % Crossover probability
mutation_prob = 0.06; % Mutation probability
popsize = 500; % Size of the population
best_length = Inf;

% Generate population with random cycles
for i=1:popsize
    tour(i,:) = randperm(n);
end
next_gen = zeros(popsize,n);
```

```
27
28 % Genetic algorithm itself.
29 for iter=1:maxiter
30     % Calculate the length of the Hamiltonian cycles
31     for i=1:popsize
32         tlength(i) = tour_length(X,tour(i,:));
33     end
34     [min_length,ind] = min(tlength);
35     best_tour = tour(ind,:);
36     tournament_size = 20;
37
38     for k=1:popsize
39         % Choosing parents for crossover using tournament
              selection
40         tournament_length = zeros(tournament_size,2);
41
42         % First parent selection
43         for i=1:tournament_size
44             random_tour = randi(popsize);
45             tournament_length(i,1) = tlength(random_tour);
46             tournament_length(i,2) = random_tour;
47         end
48         [~,ind] = min(tournament_length(:,1));
49         parent1 = tour(tournament_length(ind,2),:);
50
51         % Second parent selection
52         for i=1:tournament_size
53             random_tour = randi(popsize);
54             tournament_length(i,1) = tlength(random_tour);
55             tournament_length(i,2) = random_tour;
56         end
57         [~,ind] = min(tournament_length(:,1));
58         parent2 = tour(tournament_length(ind,2),:);
59
60         % Child generation
61         child = crossover(parent1, parent2, crossover_prob);
62         child = mutate(child, mutation_prob);
63
64         % New generation update
65         next_gen(k,:) = child;
66     end
67
68     % Assigning the created generation to the current population
69     tour = next_gen;
70     if min_length < best_length
71         best_length = min_length;
72     end
73
74     % Polygon plot
75     if rem(iter,20) == 0
```

```
        cla
        for i=2:n
            plot([X(best_tour(i-1),1) X(best_tour(i),1)], [X(
               best_tour(i-1),2) X(best_tour(i),2)], 'r', '
               LineWidth',2)
        end
        plot([X(best_tour(end),1) X(best_tour(1),1)], [X(
           best_tour(end),2) X(best_tour(1),2)], 'r', 'LineWidth'
           ,2)
        plot(X(:,1),X(:,2),'kx','MarkerSize',8,'LineWidth',2)
        pause(0)
        title(['Iteration number: ' num2str(iter)])
    end
end

% Add title
str = sprintf('Tour Length: %g', best_length);
title(str);

% Functions
function mutated_tour = mutate(tour,prob)
    mutated_tour = tour;
    if rand <= prob
        n = length(mutated_tour);
        index1 = randi(n);
        index2 = randi(n);
        if index1 < index2
            mutated_tour(index1:index2) = mutated_tour(index2:-1:
               index1);
        else
            mutated_tour(index1:-1:index2) = mutated_tour(index2:
               index1);
        end
    end
end

function cross_tour = crossover(tour1, tour2, prob)
    cross_tour = tour1;
    if rand <= prob
        cp = randi(length(tour1));
        for cpc=1:cp
            if tour2(cpc)~=tour1(cpc)
                ind = find(tour1 == tour2(cpc));
                tour1(ind) = tour1(cpc);
                tour1(cpc) = tour2(cpc);
            end
        end
        cross_tour = tour1;
    end
end
```

```
120
121 function L = tour_length(X, tour)
122     tour = [tour tour(1)];
123     L = sum(sqrt(diff(X(tour,1)).^2 + diff(X(tour,2)).^2));
124 end
```

Explications :

Instances	La taille des tours			Le temps d'exécution		
	Meilleur	Moyen	Pire	Meilleur	Moyen	Pire
att48	**34382.1**	**34539.0**	**36830.6**	**69**s	**69**s	**69**s
berlin52	**8024.24**	**8047.5**	**8276.4**	**72**s	**76**s	**80**s
kroD100	**22666.5**	**23305.8**	**23898.0**	**78**s	**78**s	**90**s
ch150	**7695.99**	**7998.34**	**8072.25**	**91**s	**92**s	**108**s
d198	**18484.2**	**18508.0**	**19168.9**	**95**s	**95**s	**102**s
a280	**4394.21**	**4531.18**	**4586.05**	**111**s	**113**s	**121**s
lin318	**79068.2**	**81219.1**	**82742.4**	**119**s	**121**s	**137**s
a535	**6009.54**	**6018.95**	**6142.0**	**176**s	**176**s	**190**s
d657	**169734.0**	**181001.0**	**182514.0**	**214**s	**215**s	**215**s
lu634	**79306.5**	**79576.3**	**80086.7**	**331**s	**333**s	**373**s

Table 4.9: Performance de l'algorithme génétique (AG)

Dans ce tableau, on remarque que certaines instances, comme **att48** et **berlin52**, ont des résultats relativement stables, ce qui montre que l'algorithme converge bien vers une solution optimale. Pour d'autres cas, comme **lu634**, les résultats varient davantage, ce qui suggère une sensibilité aux paramètres utilisés. Le temps d'exécution augmente avec la taille du problème, car **AG** doit explorer un plus grand nombre de solutions avant d'arriver à une réponse satisfaisante. En général, cet algorithme est efficace pour optimiser les tournées, mais ses performances dépendent du choix des paramètres, comme le taux de mutation et la taille de la population. Une comparaison avec d'autres méthodes comme **ACO** ou **2-opt** permettrait de mieux comprendre ses avantages et ses limites.

4.3.3 Solutions de tournées pour berlin52.tsp:

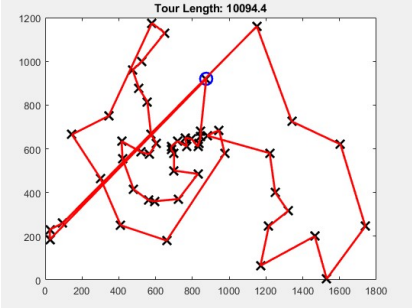

Figure 4.11: Solutions de tournées pour berlin52.tsp par l'algoritme de PPV

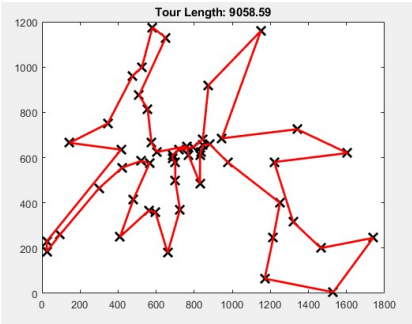

Figure 4.12: Solutions de tournées pour berlin52.tsp par l'algoritme de cheapest insertion

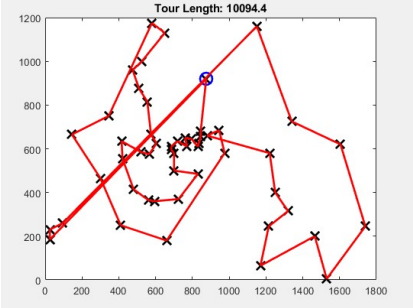

Figure 4.13: Solutions de tournées pour berlin52.tsp par l'algoritme 2-opt

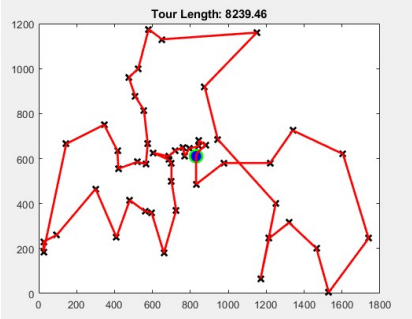

Figure 4.14: Solutions de tournées pour berlin52.tsp par l'algoritme de CW

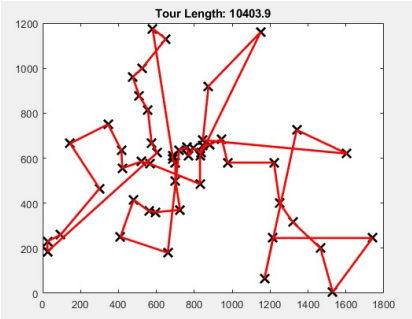

Figure 4.15: Solutions de tournées pour berlin52.tsp par l'algoritme de MST

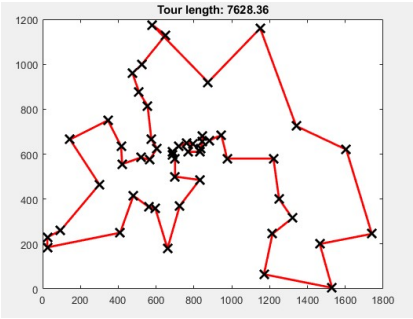

Figure 4.16: Solutions de tournées pour berlin52.tsp par l'algoritme d'ACO

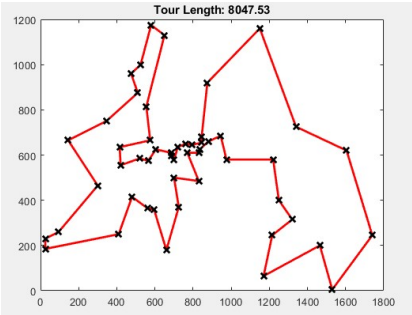

Figure 4.17: Solutions de tournées pour berlin52.tsp par l'algoritme d'AG

Figure 4.18: Les solutions cycliques générées par divers algorithmes pour l'instance berlin52.tsp.

4.4 Étude comparative des algorithmes:

4.4.1 Analyse comparative des performances des algorithmes:

4.4.1.1 L'instance att48:

Instances	La taille des tours			Le temps d'exécution		
	Meilleur	Moyen	Pire	Meilleur	Moyen	Pire
Plus Proche Voisin	**34332.5**	**34477, 9**	**35045, 6**	**46**	**17**	**22**
Cheapest Insertion	**39260**	**39260**	**39260**	**5s**	**5s**	**5s**
MST	**43955.8**	**43955.8**	**43955.8**	**5s**	**5s**	**5s**
Clarke et Wright	**35076**	**35296**	**35400.4**	**10s**	**10s**	**11s**
2-opt	**34332.5**	**34477.9**	**35045.6**	**14s**	**17s**	**22s**
ACO	**34088, 3**	**34091, 1**	**34443, 1**	**97s**	**101s**	**114s**
Génétique	**34382.1**	**34539.0**	**36830.6**	**69s**	**69s**	**69s**

Table 4.10: Performance des algorithmes pour l'instance att48

Le tableau présenté compare plusieurs **algorithmes d'optimisation** appliqués à l'instance **att48** du problème du voyageur de commerce .

Il est organisé en deux parties principales : d'une part, la qualité des solutions obtenues, mesurée par la taille des tours avec trois valeurs (meilleur, moyen et pire), et d'autre part, le temps d'exécution nécessaire pour chaque méthode, également divisé en ces trois catégories.

Les algorithmes testés incluent des heuristiques telles que **Plus Proche Voisin**, **Cheapest Insertion**, **MST**, **Clarke & Wright**, et **2-opt**, ainsi que des approches plus avancées comme **ACO** (Ant Colony Optimization) et **les Algorithmes Génétiques**.

On observe que **ACO** produit des solutions de haute qualité mais avec un temps de calcul élevé, tandis que des méthodes plus simples comme **Clarke et Wright** et **2-opt** offrent un équilibre entre performance et rapidité.

Cette analyse permet d'évaluer les forces et limites de chaque approche et d'orienter le choix de l'algorithme en fonction des exigences du problème à résoudre.

4.4.1.2 L'instance berlin52:

Instances	La taille des tours			Le temps d'exécution		
	Meilleur	Moyen	Pire	Meilleur	Moyen	Pire
Plus Proche Voisin	**9320.3**	**9320.3**	**9320.3**	**13s**	**13s**	**13s**
Cheapest Insertion	**9058.59**	**9058.59**	**9058.59**	**7s**	**7s**	**7s**
MST	**10403.9**	**10403.9**	**10403.9**	**7s**	**7s**	**7s**
Clarke et Wright	**8097.95**	**8239.46**	**8462.28**	**11s**	**18s**	**19s**
2-opt	**9320.3**	**9320.3**	**9320.3**	**13s**	**13s**	**13s**
ACO	**7548, 99**	**7628, 36**	**7681, 6**	**107s**	**110s**	**115s**
Génétique	**8024.24**	**8047.5**	**8276.4**	**72s**	**76s**	**80s**

Table 4.11: Performance des algorithmes pour l'instance berlin52

Le tableau présenté compare plusieurs **algorithmes d'optimisation** appliqués à l'instance **berlin52** du problème du voyageur de commerce .

Il présente deux critères essentiels : la qualité des solutions obtenues, mesurée par la taille des tours (*Meilleur, Moyen et Pire*), et le temps d'exécution nécessaire pour chaque algorithme, également réparti en ces trois catégories. Les méthodes comparées incluent des heuristiques classiques comme **Plus Proche Voisin**, **Cheapest Insertion**, **MST**, **Clarke & Wright** et **2-opt**, ainsi que des approches plus avancées telles que **ACO** (Ant Colony Optimization) et **les Algorithmes Génétiques**.

Les résultats indiquent que **ACO** génère les solutions les plus courtes mais à un coût computationnel élevé, tandis que des heuristiques comme **Cheapest Insertion** et **Clarke et Wright** offrent un bon compromis entre rapidité et qualité. En revanche, **MST** se révèle moins performant en termes de qualité de solution.

Cette analyse permet d'orienter le choix des algorithmes en fonction des exigences du problème étudié, que ce soit pour privilégier la précision ou optimiser le temps de calcul.

4.4.1.3 l'instance kroD100:

Instances	**La taille des tours**			**Le temps d'exécution**		
	Meilleur	**Moyen**	**Pire**	**Meilleur**	**Moyen**	**Pire**
Plus Proche Voisin	**22080.6**	**22097.0**	**22304.1**	**34s**	**35s**	**38s**
Cheapest Insertion	**26517.4**	**26517.4**	**26517.4**	**28s**	**28s**	**28s**
MST	**28599.1**	**28599.1**	**28599.1**	**2s**	**2s**	**2s**
Clarke et Wright	**22743.6**	**22972.3**	**24279.5**	**22s**	**22s**	**23s**
2-opt	**22080.6**	**22097.0**	**22304.1**	**34s**	**35s**	**38s**
ACO	**22624, 6**	**22963.7**	**24187, 7**	**184s**	**185s**	**187s**
Génétique	**22666.5**	**23305.8**	**23898.0**	**78s**	**78s**	**90s**

Table 4.12: Performance des algorithmes pour l'instance kroD100

Le tableau présenté compare plusieurs **algorithmes d'optimisation** appliqués à l'instance**kroD100** du problème du voyageur de commerce .

Il est structuré autour de deux critères principaux : la qualité des solutions obtenues, représentée par la **taille des tours** (Meilleur, Moyen et Pire), et le **temps d'exécution**, qui indique la rapidité de chaque méthode pour fournir une solution.

Les heuristiques classiques comme **Plus Proche Voisin**, **Cheapest Insertion**, **MST**, **Clarke & Wright** et **2-opt** sont comparées à des approches plus sophistiquées telles que **ACO** (Ant Colony Optimization) et **les Algorithmes Génétiques**.

On observe que **MST** produit des tours relativement longs mais avec un temps de calcul extrêmement rapide (**2 secondes**). À l'opposé, **ACO** offre de bonnes solutions en termes de taille des tours mais nécessite un temps de calcul significativement plus élevé. Les autres méthodes, comme **Clarke et Wright** et **les Algorithmes Génétiques**, montrent un compromis intéressant entre qualité et rapidité, avec des performances intermédiaires.

Cette analyse met en évidence les forces et limites de chaque approche, permettant d'orienter le choix de l'algorithme en fonction des contraintes de rapidité et de précision exigées pour résoudre le problème étudié.

4.4.1.4 L'instance ch150:

Instances	**La taille des tours**			**Le temps d'exécution**		
	Meilleur	**Moyen**	**Pire**	**Meilleur**	**Moyen**	**Pire**
Plus Proche Voisin	**6660.34**	**6752.1**	**6775.88**	**40s**	**40s**	**59s**
Cheapest Insertion	**8475.45**	**8475.45**	**68475.45**	**80s**	**80s**	**80s**
MST	**9202.46**	**9202.46**	**9202.46**	**23s**	**23s**	**23s**
Clarke et Wright	**7024.46**	**7066.88**	**7149**	**31s**	**32s**	**33s**
2-opt	**6660.34**	**6752.1**	**6775.88**	**40s**	**40s**	**59s**
ACO	**11447, 4**	**11689, 2**	**11919, 9**	**31s**	**43s**	**74s**
Génétique	**7695.99**	**7998.34**	**8072.25**	**91s**	**92s**	**108s**

Table 4.13: Performance des algorithmes pour l'instance ch150

Le tableau présenté compare plusieurs **algorithmes d'optimisation** appliqués à l'instance **ch150** du problème du voyageur de commerce.

Il est structuré en deux parties principales : d'une part, la **taille des tours**, qui mesure la qualité des solutions obtenues avec les valeurs *Meilleur, Moyen et Pire*, et d'autre part, le **temps d'exécution**, qui reflète la rapidité de calcul de chaque méthode selon les mêmes trois niveaux.

Les résultats montrent que **Plus Proche Voisin** et **2-opt** produisent des solutions similaires avec une taille de tour relativement courte, mais avec un temps de calcul potentiellement plus élevé en raison des améliorations successives.

MST, bien que rapide, offre des résultats moins performants en termes de longueur de tour. **Cheapest Insertion**, quant à lui, présente une anomalie dans la valeur *Pire*, ce qui peut suggérer une erreur de saisie ou une particularité algorithmique. **Clarke et Wright** et **les Algorithmes Génétiques** se situent dans une gamme intermédiaire, alliant **qualité des solutions** et **temps de calcul raisonnable**.

Enfin, **ACO**, bien qu'il tende à produire des tours plus longs, présente une variabilité plus importante en termes de durée de calcul.

Cette analyse met en évidence les forces et les faiblesses de chaque algorithme, permettant d'adapter le choix en fonction des exigences du problème : soit **optimiser la précision**, soit **minimiser le temps de calcul**.

4.4.1.5 L'instance d198:

Instances	La taille des tours			Le temps d'exécution		
	Meilleur	**Moyen**	**Pire**	**Meilleur**	**Moyen**	**Pire**
Plus Proche Voisin	**16189.2**	**16236.8**	**16250.2**	**55s**	**55s**	**57s**
Cheapest Insertion	**18281**	**18281**	**18281**	**112s**	**112s**	**112s**
MST	**19353.5**	**19353.5**	**19353.5**	**7s**	**7s**	**7s**
Clarke et Wright	**17713.3**	**22695.4**	**23599.1**	**21s**	**22s**	**67s**
2-opt	**16189.2**	**16236.8**	**16250.2**	**55s**	**55s**	**57s**
ACO	**18437, 3**	**18575, 8**	**18918, 4**	**368s**	**409s**	**413s**
Génétique	**18484.2**	**18508.0**	**19168.9**	**95s**	**95s**	**102s**

Table 4.14: Performance des algorithmes pour l'instance d198

Le tableau présenté compare plusieurs **algorithmes d'optimisation** appliqués à l'instance**d198** du problème du voyageur de commerce .

Il est structuré en deux sections principales : la **taille des tours**, qui évalue la qualité des solutions obtenues selon trois niveaux (*Meilleur, Moyen et Pire*), et le **temps d'exécution**, qui mesure la rapidité des calculs pour chaque méthode.

Les résultats montrent que **Plus Proche Voisin** et **2-opt** offrent des solutions relativement courtes avec des temps de calcul modérés. **Cheapest Insertion**, bien que produisant un tour plus long, nécessite un temps d'exécution nettement plus élevé (**112s**).

En revanche, **MST** se distingue par une très grande rapidité (**7s**) mais génère des solutions moins optimales. **Clarke et Wright** affiche une forte variabilité en termes de qualité de solution, allant d'un tour compétitif (**17,713.3**) à une valeur nettement plus élevée (**23,599.1**), avec un temps d'exécution également variable. **ACO**, tout en fournissant des solutions raisonnables, présente un temps de calcul extrêmement élevé, dépassant **400s** en moyenne.

Enfin, **les Algorithmes Génétiques** offrent une alternative intermédiaire, équilibrant qualité et temps d'exécution.

Cette analyse permet de mieux cerner les avantages et inconvénients de chaque approche, offrant des perspectives sur le choix optimal selon les exigences du problème étudié.

4.4.1.6 l'instance a280:

Instances	La taille des tours			Le temps d'exécution		
	Meilleur	**Moyen**	**Pire**	**Meilleur**	**Moyen**	**Pire**
Plus Proche Voisin	**2755.58**	**2761.56**	**2785.08**	**77s**	**78s**	**80s**
Cheapest Insertion	**3475.25**	**3475.25**	**3475.25**	**137s**	**137s**	**137s**
MST	**3484.85**	**3484.85**	**3484.85**	**4s**	**4s**	**4s**
Clarke et Wright	**2847.01**	**2863.16**	**2913.5**	**62s**	**62s**	**65s**
2-opt	**2755.58**	**2761.56**	**2785.08**	**77s**	**78s**	**80s**
ACO	**3525, 48**	**3533, 64**	**3541, 71**	**556s**	**557s**	**571s**
Génétique	**4394.21**	**4531.18**	**4586.05**	**111s**	**113s**	**121s**

Table 4.15: Performance des algorithmes pour l'instance a280

Le tableau présenté compare plusieurs **algorithmes d'optimisation** appliqués à l'instance **a280** du problème du voyageur de commerce.

Il est structuré en deux parties principales : la **taille des tours**, qui mesure la qualité des solutions obtenues selon trois niveaux (*Meilleur, Moyen et Pire*), et le **temps d'exécution**, qui indique la rapidité de calcul de chaque méthode.

Les résultats montrent que **Plus Proche Voisin** et **2-opt** produisent des solutions similaires avec une taille de tour relativement courte et un temps de calcul modéré. **Cheapest Insertion** génère un tour plus long avec un temps d'exécution significativement plus élevé (**137s**).

MST, bien que rapide (**4s**), propose des résultats moins optimaux en termes de longueur de tour. **Clarke et Wright** présente une variation modérée dans la qualité des solutions, maintenant un équilibre entre la taille du tour et le temps d'exécution. **ACO**, tout en offrant des solutions acceptables, requiert un temps de calcul extrêmement élevé (**556s à 571s**), ce qui peut limiter son applicabilité dans des contextes nécessitant une optimisation rapide.

Enfin, **les Algorithmes Génétiques** montrent une solution de qualité intermédiaire, mais avec un coût temporel plus élevé par rapport à certaines heuristiques.

Cette analyse met en évidence les forces et les limites de chaque approche, permettant d'orienter le choix de l'algorithme en fonction des contraintes du problème, que ce soit pour **minimiser le temps de calcul** ou **optimiser la qualité des solutions**

4.4.1.7 L'instance lin318:

Instances	**La taille des tours**			**Le temps d'exécution**		
	Meilleur	**Moyen**	**Pire**	**Meilleur**	**Moyen**	**Pire**
Plus Proche Voisin	**43919.2**	**44296.3**	**44645.5**	**90s**	**92s**	**112s**
Cheapest Insertion	**54471.4**	**54471.4**	**54471.4**	**1543s**	**1543s**	**1543s**
MST	**60989.1**	**60989.1**	**60989.1**	**5s**	**5s**	**5s**
Clarke et Wright	**46872.1**	**46872.1**	**46872.1**	**95s**	**95s**	**95s**
2-opt	**43919.2**	**44296.3**	**44645.5**	**90s**	**92s**	**112s**
ACO	**54103, 6**	**56075, 4**	**56633, 4**	**619s**	**635s**	**639s**
Génétique	**79068.2**	**81219.1**	**82742.4**	**119s**	**121s**	**137s**

Table 4.16: Performance des algorithmes pour l'instance lin318

Le tableau présenté compare plusieurs **algorithmes d'optimisation** appliqués à l'instance **lin318** du problème du voyageur de commerce.

Il est divisé en deux sections essentielles : la **taille des tours**, qui mesure la qualité des solutions obtenues selon trois niveaux (*Meilleur, Moyen et Pire*), et le **temps d'exécution**, qui indique la rapidité de calcul de chaque méthode.

Les résultats montrent que **Plus Proche Voisin** et **2-opt** génèrent des solutions de qualité similaire avec une longueur de tour relativement courte et un temps d'exécution modéré. **Cheapest Insertion**, bien que produisant une solution plus longue, nécessite un temps de calcul extrêmement élevé (**1543s**), ce qui le rend moins efficace pour des calculs rapides. **MST**,

en revanche, est très rapide (**5s**) mais produit une solution bien plus éloignée du meilleur résultat attendu. **Clarke et Wright**, tout en gardant une taille de tour stable, offre un équilibre intéressant entre performance et rapidité. **ACO** génère des solutions compétitives, bien que son temps d'exécution soit relativement long (**619s à 639s**).

Enfin, **les Algorithmes Génétiques** produisent des tours plus longs mais restent raisonnables en termes de temps de calcul comparé aux autres méthodes avancées.

Cette analyse permet de mieux cerner les avantages et inconvénients de chaque approche, offrant un éclairage sur le choix optimal à privilégier selon les contraintes du problème, que ce soit pour maximiser la qualité des solutions ou minimiser le temps de calcule.

4.4.1.8 L'instance ali535:

Instances	**La taille des tours**			**Le temps d'exécution**		
	Meilleur	**Moyen**	**Pire**	**Meilleur**	**Moyen**	**Pire**
Plus Proche Voisin	**2157.77**	**2161.93**	**2181.96**	**154s**	**161s**	**182s**
Cheapest Insertion	**2757.56**	**2757.56**	**2757.56**	**9s**	**9s**	**9s**
MST	**6009.54**	**6018.95**	**6142.0**	**176s**	**176s**	**190s**
Clarke et Wright	**2194.5**	**2233.86**	**2238.98**	**128s**	**130s**	**131s**
2-opt	**2157.77**	**2161.93**	**2181.96**	**154s**	**161s**	**182s**
ACO	**34179, 3**	**34210, 5**	**34509, 7**	**1503s**	**1506s**	**1646s**
Génétique	**6009.54**	**6018.95**	**6142.0**	**176s**	**176s**	**190s**

Table 4.17: Performance des algorithmes pour l'instance a535

Le tableau présenté compare plusieurs **algorithmes d'optimisation** appliqués à l'instance **a535** du problème du voyageur de commerce.

Il met en évidence deux aspects fondamentaux : la **taille des tours**, qui mesure la qualité des solutions obtenues selon trois niveaux (*Meilleur, Moyen et Pire*), et le **temps d'exécution**, qui reflète la rapidité de chaque méthode pour produire ces solutions.

Les résultats montrent que **Plus Proche Voisin** et **2-opt** offrent des performances similaires avec une taille de tour relativement courte, mais avec un temps de calcul **élevé** (jusqu'à **182s**). **Cheapest Insertion**, bien que produisant une solution plus longue, est extrêmement rapide (**9s**), ce qui le rend intéressant pour des situations où la rapidité est prioritaire. **MST**, malgré un temps de calcul modéré (**176s**), produit des solutions nettement moins performantes, tandis que **Clarke et Wright** maintient un bon équilibre entre qualité et vitesse. **ACO**, tout en offrant des solutions comparables aux autres méthodes avancées, présente un coût de calcul très élevé (**1503s à 1646s**), ce qui peut limiter son applicabilité dans un cadre où la rapidité est essentielle.

Enfin, **les Algorithmes Génétiques** fournissent des résultats similaires à **MST**, avec un temps de calcul équivalent.

Cette analyse met en lumière les forces et faiblesses de chaque approche, facilitant le choix de l'algorithme le plus adapté selon les exigences du problème, qu'il s'agisse de **minimiser le temps de calcul** ou **optimiser la qualité des solutions**.

4.4.1.9 L'instance d657:

Instances	La taille des tours			Le temps d'exécution		
	Meilleur	**Moyen**	**Pire**	**Meilleur**	**Moyen**	**Pire**
Plus Proche Voisin	**51351.9**	**51378.9**	**51636.1**	**197s**	**201s**	**206s**
Cheapest Insertion	**65857.4**	**65857.4**	**65857.4**	**9s**	**9s**	**9s**
MST	**169734.0**	**181001.0**	**182514.0**	**214s**	**215s**	**215s**
Clarke et Wright	**53282.3**	**53865.0**	**54066.0**	**186s**	**187s**	**226s**
2-opt	**51351.9**	**51378.9**	**51636.1**	**197s**	**201s**	**206s**
ACO	**72691, 4**	**72724, 9**	**73509, 7**	**1968s**	**1993s**	**2058s**
Génétique	**169734.0**	**181001.0**	**182514.0**	**214s**	**215s**	**215s**

Table 4.18: Performance des algorithmes pour l'instance d657

Le tableau présenté compare plusieurs **algorithmes d'optimisation** appliqués à l'instance **d657** du problème du voyageur de commerce.

Il est structuré autour de deux indicateurs clés : la **taille des tours**, qui mesure la qualité des solutions obtenues à travers trois niveaux (*Meilleur, Moyen et Pire*), et le **temps d'exécution**, qui reflète la rapidité de calcul de chaque méthode.

Les résultats montrent que **Plus Proche Voisin** et **2-opt** produisent des solutions relativement courtes avec un temps de calcul modéré. **Cheapest Insertion**, bien que générant un tour plus long, reste l'un des plus rapides (**9s**). **MST**, quant à lui, présente des solutions beaucoup plus longues, ce qui impacte directement la qualité de l'optimisation. **Clarke et Wright**, avec une taille de tour intermédiaire, offre un équilibre raisonnable entre performance et temps de calcul. **ACO**, tout en proposant des solutions compétitives, affiche un coût computationnel très élevé (**1968s à 2058s**), ce qui peut le rendre difficilement utilisable pour des calculs nécessitant une exécution rapide.

Enfin, **les Algorithmes Génétiques** affichent des performances similaires à **MST**, suggérant une complexité algorithmique qui impacte directement la qualité de la solution et le temps d'exécution.

Cette analyse met en lumière les forces et les limites de chaque approche, permettant d'orienter le choix de l'algorithme en fonction des contraintes du problème : **minimiser le temps de calcul** ou **optimiser la qualité de la solution**.

4.4.1.10 L'instance lu634:

Instances	La taille des tours			Le temps d'exécution		
	Meilleur	**Moyen**	**Pire**	**Meilleur**	**Moyen**	**Pire**
Plus Proche Voisin	**11855.1**	**11900.0**	**12038.3**	**263s**	**269s**	**272s**
Cheapest Insertion	**14281.9**	**14425.6**	**14750.2**	**2120**s	**2180**s	**2300**s
MST	**79306.5**	**79576.3**	**80086.7**	**331s**	**333s**	**373s**
Clarke et Wright	**12332.0**	**12608.5**	**12617.1**	**563s**	**566s**	**664s**
2-opt	**11855.1**	**11900.0**	**12038.3**	**263s**	**269s**	**272s**
ACO	**129279**	**130361**	**131309**	**3347s**	**3381s**	**3833s**
Génétique	**79306.5**	**79576.3**	**80086.7**	**331s**	**333s**	**373s**

Table 4.19: Performance des algorithmes pour l'instance lu634

Le tableau présenté compare plusieurs **algorithmes d'optimisation** appliqués à l'instance **lu634** du problème du voyageur de commerce.

Il met en évidence deux aspects clés : la **taille des tours**, qui mesure la qualité des solutions obtenues à travers trois niveaux (*Meilleur, Moyen et Pire*), et le **temps d'exécution**, qui évalue la rapidité de chaque méthode pour produire ces résultats.

Les résultats montrent que **Plus Proche Voisin** et **2-opt** fournissent des solutions similaires, avec des tailles de tour compétitives et des temps de calcul modérés (**263s à 272s**). **Cheapest Insertion**, bien qu'offrant une solution plus longue, reste extrêmement rapide (**25s**). **MST** et **les Algorithmes Génétiques**, tout en proposant des solutions de moindre qualité, affichent des temps d'exécution relativement modérés (**331s à 373s**). **Clarke et Wright**, en revanche, présente des performances intermédiaires, avec une certaine variabilité dans les solutions et un coût temporel plus élevé (**563s à 664s**). **ACO** offre des résultats compétitifs mais au prix d'un temps de calcul très élevé (**3347s à 3833s**), ce qui peut limiter son applicabilité dans des contextes nécessitant une optimisation rapide.

Cette analyse met en avant les forces et les limites de chaque méthode, permettant d'adapter le choix de l'algorithme en fonction des exigences du problème : **minimiser le temps de calcul** ou **optimiser la qualité des solutions**.

4.4.2 Évaluation des Algorithmes:Temps de Calcul, Qualité de la Solution et Complexité:

Dans le cadre de l'optimisation des tournées dans le Problème du Voyageur de Commerce (TSP), plusieurs algorithmes sont évalués selon trois critères fondamentaux : rapidité d'exécution, qualité de la solution obtenue, et complexité algorithmique. Certains algorithmes privilégient la rapidité, d'autres garantissent une solution plus proche de l'optimal, tandis que certains assurent une complexité minimale pour une meilleure efficacité computationnelle.

Critère	**Algorithme**	**Remarque**
Meilleur compromis	Clarke et Wright, 2-opt	Bon équilibre entre rapidité et qualité
Solution la plus rapide	MST	Très rapide, mais qualité plus faible
Solution la plus optimisée	Colonie de Fourmis	Excellente qualité, mais exécution lente
Meilleur en complexité	MST	Complexité minimale $O(n^2)$

Table 4.20: Synthèse comparative des algorithmes selon quatre critères

Ce tableau présente une synthèse comparative des performances des algorithmes en fonction de ces critères, permettant de visualiser les meilleurs choix selon les besoins spécifiques d'optimisation.

4.4.3 Meilleurs algorithme pour chaque instance:

Dans cette section, nous examinerons les meilleurs algorithmes selon plusieurs critères, notamment la qualité du tour obtenu, le temps d'exécution et la complexité algorithmique. L'objectif est d'identifier les méthodes offrant le meilleur compromis entre performance et faisabilité pour différentes instances du TSP.

4.4.3.1 Temps de calcul:

		Le temps d'exécution (s)		
Instance	**Meilleur algorithme**	**Meilleure**	**Moyenne**	**Pire**
berlin52	Insertion au Moindre Coût	7	7	7
kroD100	MST	2	2	2
ch150	Plus Proche Voisin / 2-opt(pas d'amélioration)	40	40,5	59
d198	MST	7	7	7
a280	MST	4	4	4
lin318	MST	5	5	5
ali535	MST	9	9	9
d657	MST	9	9	9
lu634	MST	25	25	25

Table 4.21: Comparaison des meilleurs algorithmes avec les tailles de tour et les temps d'exécution

Ce tableau montre quels algorithmes sont les plus rapides pour différentes instances du problème du voyageur de commerce (TSP). On remarque que **MST** est souvent le plus rapide, avec un temps d'exécution constant et très court. Cependant, sa rapidité peut parfois réduire la qualité des solutions trouvées.

Plus Proche Voisin / 2-opt prend un peu plus de temps, mais il permet d'obtenir de meilleures tournées(pas d'amelioration). **Insertion au Moindre Coût** est efficace dans l'instance **berlin52**, offrant une solution rapide en seulement 7 secondes.

Enfin, on observe que les instances plus grandes prennent plus de temps à être résolues. Par exemple, pour **lu634**, MST met 25 secondes, alors que pour **kroD100**, il ne prend que 2 secondes. Cela montre que plus il y a de villes, plus le calcul est long.

En résumé, **MST** est très rapide, mais d'autres algorithmes comme **2-opt** et **Insertion au Moindre Coût** peuvent améliorer la qualité des résultats.

4.4.3.2 Qualité de la solution:

		La taille des tours(Km)		
Instance	**Meilleur algorithme**	**Meilleure**	**Moyenne**	**Pire**
berlin52	Colonie de Fourmis	7548,99	7628,36	7681,6
kroD100	Clarke et Wright	22743,6	22972,3	24279,5
ch150	Plus Proche Voisin / 2-opt(pas d'amélioration)	6660,34	6752,1	6775,88
d198	Plus Proche Voisin / 2-opt(pas d'amélioration)	16189,2	16236,8	16250,2
a280	Plus Proche Voisin / 2-opt(pas d'amélioration)	2755,58	2761,56	2785,08
lin318	Plus Proche Voisin / 2-opt(pas d'amélioration)	43919,2	44296,3	44645,5
ali535	Plus Proche Voisin / 2-opt(pas d'amélioration)	2157,77	2161,93	2181,96
d657	Clarke et Wright	53282,3	53865,0	54066,0
lu634	Plus Proche Voisin / 2-opt(pas d'amélioration)	11855,1	11900,0	12038,3

Table 4.22: Meilleur algorithme pour chaque instance en fonction de la taille du tour.

Ce tableau compare les meilleurs algorithmes pour chaque instance du PVC, en tenant compte de la qualité des solutions et du temps d'exécution.

On remarque que Plus Proche Voisin / 2-opt est souvent une bonne option, car il offre des solutions efficaces sur plusieurs instances comme ch150, d198 et lu634.

L'algorithme de la Colonie de Fourmis est performant sur berlin52, trouvant une solution optimisée, mais avec un temps de calcul plus long. Clarke et Wright est bien adapté aux instances kroD100 et d657, où il fournit des tournées optimisées en peu de temps.

Enfin, les instances plus grandes, comme lin318 et ali535, demandent plus de temps de calcul, ce qui montre que le problème devient plus complexe à mesure que le nombre de villes augmente.

En résumé, chaque algorithme a ses forces : certains sont rapides, d'autres donnent des meilleures solutions, et il faut choisir celui qui correspond le mieux aux besoins du problème étudié.

4.4.3.3 Complexité:

L'algorithme **MST** s'est révélé être le plus performant en termes de complexité pour toutes les instances analysées, notamment **berlin52, kroD100, ch150, d198, a280, lin318, ali535, d657** et **lu634**.

Dans chaque cas, sa complexité reste $\mathbf{O(n^2)}$, confirmant sa stabilité et son efficacité dans la résolution de ces problèmes du voyageur de commerce. Cette régularité démontre que, malgré la variation des instances, **MST** offre une approche cohérente et optimisée dans ce contexte.

4.5 Différence entre les heuristiques est metaheuristiques:

Critère	**Heuristiques**	**Métaheuristiques**
Qualité des solutions	Moyenne à bonne, parfois sous-optimales	Bonne à excellente, souvent proches de l'optimal
Temps d'exécution	Rapide à modéré	Modéré à long (souvent plus lent)
Complexité	Généralement $O(n^2)$	Souvent $O(k \cdot n^2)$ ou plus selon les paramètres
Facilité de mise en œuvre	Simple, intuitive, peu de paramètres	Moins simple, nécessite un réglage précis et une bonne compréhension
Exemples	Plus Proche Voisin, Cheapest Insertion, Clarke & Wright, 2-opt, MST	Colonie de Fourmis (ACO), Algorithmes Génétiques, Recuit Simulé, Recherche Tabou

Table 4.23: Comparaison générale entre heuristiques et métaheuristiques

4.6 Conclusion:

Dans ce mémoire met en évidence **l'importance des méthodes heuristiques et métaheuristiques** dans la résolution du **Problème du Voyageur de Commerce (TSP)**.

Les **heuristiques**, telles que *Plus Proche Voisin, Insertion au Moindre Coût et Clarke & Wright*, permettent de construire rapidement une solution en suivant des règles simples. Elles offrent un bon compromis entre **rapidité et efficacité**, bien qu'elles puissent parfois produire des solutions sous-optimales. **Les heuristiques d'amélioration**, comme *2-opt*, optimisent ces solutions en **réduisant les croisements et en améliorant progressivement la qualité des tournées**.

Les **métaheuristiques**, comme *l'algorithme génétique et l'optimisation par colonie de fourmis*, explorent plus intelligemment l'espace de recherche en simulant des phénomènes biologiques ou statistiques. Elles permettent souvent d'obtenir des solutions de meilleure qualité, mais au prix d'un **temps de calcul plus long**.

L'étude comparative a montré que **le choix de la meilleure méthode dépend des contraintes du problème** : certaines approches sont privilégiées pour leur rapidité, tandis que d'autres assurent des solutions optimales au détriment du temps d'exécution. Ces résultats ouvrent **des perspectives pour des méthodes hybrides**, combinant plusieurs techniques afin d'obtenir des solutions encore plus performantes.

Conclusion générale

Ce mémoire est structuré en quatre chapitres, chacun abordant un aspect spécifique de l'étude.

Le premier chapitre introduit les concepts fondamentaux de l'optimisation combinatoire, expliquant les principales problématiques et les différentes méthodes utilisées en recherche opérationnelle.

Le deuxième chapitre est consacré au **Problème du Voyageur de Commerce (PVC)**, a été examiné avec une présentation de son historique, de sa formulation mathématique et des défis liés à sa complexité.

Ensuite, le troisième chapitre exploré les méthodes de résolution en comparant les approches exactes, heuristiques et métaheuristiques, mettant en évidence leurs performances et limitations.

Enfin, le dernier chapitre présenté une **étude empirique**, où différentes heuristiques et métaheuristiques ont été testées sur des instances du PVC afin de comparer leurs résultats en termes de temps d'exécution, qualité des solutions et complexité algorithmique.

Les résultats obtenus montrent les forces et faiblesses des différentes méthodes et ouvrent des perspectives pour des **futures recherches**, notamment en élargissant cette étude à d'autres heuristiques et métaheuristiques, et pourquoi pas à des méthodes exactes pour affiner l'optimisation des tournées.

Cette démarche pourrait apporter des avancées significatives dans l'optimisation logistique, la gestion des ressources et les systèmes de transport, rendant les solutions encore plus efficaces et adaptées aux besoins réels.

Bibliography

[1] Papadimitriou,C.H.,Steiglitz, K. (1998). *Combinatorial Optimization: Algorithms and Complexity*. Dover Publications.

[2] HAMDAOUI FAWZIA (2016). *Contribution à la résolution du problème du sac à dos multidimensionnel à choix multiple*. Université des Sciences et de la Technologie d'Oran Mohamed Boudiaf, Faculté des Mathématiques et Informatique, spécialité Informatique.

[3] M. S. Bazaraa, H. D. Sherali, and C. M. Shetty, *Nonlinear Programming: Theory and Algorithms*, John Wiley & Sons, 2013.

[4] KOUCHI Mohamed et Mili Kamel (2012–2013). *Le problème du voyageur de commerce bi-objectifs*. Disponible sur : https://wikimemoires.net/. Consulté le 18/04/2025 à 22:26.

[5] Kuhn, H. W. (1955). *The Hungarian method for the assignment problem*. Naval Research Logistics Quarterly, 2(1-2), 83-97.

[6] Kellerer, H., Pferschy, U., Pisinger, D. (2004). *Knapsack Problems*. Springer.

[7] Applegate, D. L., Bixby, R. E., Chvátal, V., Cook, W. J. (2007). *The Traveling Salesman Problem: A Computational Study*. Princeton University Press

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

[9] Didier Maquin (2024). *Traveling Salesman Problem*, Université de Lorraine. Disponible via Matlab Central File Exchange.

[10] Amanur Rahman Saiyed (2012). *The Traveling Salesman Problem*, Indiana State University, Terre Haute, IN, USA. asaiyed@sycamores.indstate.edu. Page: 2–3.

[11] Maredia A (2010). *History, Analysis and Implementation of Travelling Salesman Problem (TSP) and Related Problems*, Thesis, University of Houston-Downtown, pp. 1–40, Spring-2010.

[12] Kechmane, L., Nsiri, B., & Baalal, A. *Traveling Salesman Problem : Présentation et application*, Département de Mathématiques et Informatique, Laboratoire MACS, Université

Hassan II, Faculté des Sciences Casablanca, Km 8 route d'Eljadida, P.B 5366, Maarif 20100, Maroc.

[13] Kirkpatrick, S., Gelatt, C. D., & Vecchi, M. P. (1983). *Optimization by Simulated Annealing*. Science, 220(4598), 671–680.

[14] Artigues, C., Demassey, S., & Néron, E. (2008). *Proposition d'une méthodologie multicritère pour la résolution du problème d'ordonnancement d'un projet avec prise en compte des compétences et des ressources*, 123dok FR, pp. 31–35.

[15] Demange, M. (2008). *Algorithmes d'approximation : un petit tour en compagnie d'un voyageur de commerce*, Springer-Verlag France.

[16] Lust, T. (2020). *Les métaheuristiques pour traiter les problèmes NP-difficiles*, Bibliothèque Tangente, Hors Série (75), pp. 20–22.

[17] Blin, Laurent. *Algorithme distribué d'arbres couvrants de poids minimum*. Disponible en ligne : Lien vers le document.

[18] Douiri, S. M., Elbernoussi, S., & Lakhbab, H. (2025). *Cours des Méthodes de Résolution Exactes, Heuristiques et Métaheuristiques*, Université Mohammed V, Faculté des Sciences de Rabat.

[19] Souad Bernoussi, *Méthodes de Résolution Exactes, Heuristiques et Métaheuristiques*, Université Mohammed V, Faculté des Sciences de Rabat, 2025.

[20] Reinelt, G. (1991). *TSPLIB - A Traveling Salesman Problem Library*. Institut für Informatik, Universität Heidelberg. Disponible sur http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/.

[21] Marinakis, Y. (2024). *Heuristic and Metaheuristic Algorithms for the Traveling Salesman Problem*. Encyclopedia of Optimization, Springer. Disponible sur https://link.springer.com/rwe/10.1007/978-0-387-74759-0_262.

[22] Clark, P. J., & Evans, F. C. (1954). *Distance to nearest neighbor as a measure of spatial relationships in populations*, Ecology, 35(4), pp. 445–453.

[23] Reinelt, G. (1994). *The Traveling Salesman: Computational Solutions for TSP Applications*, Springer Science & Business Media.

[24] Christofides, N. (1976). *Worst-case analysis of a new heuristic for the traveling salesman problem*, Technical Report, Carnegie Mellon University.

[25] Clarke, G., & Wright, J. W. (1964). *Scheduling of Vehicles from a Central Depot to a Number of Delivery Points*, Operations Research, 12(4), pp. 568–581.

[26] Croes, G. A. (1958). *A method for solving traveling-salesman problems*, Operations Research, 6(6), pp. 791–812.

[27] Dorigo, M., Maniezzo, V., & Colorni, A. (1996). *Ant System: Optimization by a Colony of Cooperating Agents*. IEEE Transactions on Systems, Man, and Cybernetics—Part B, 26(1), 29-41.

[28] Slimane Belhia and Mohammed Bettaher, *Applications des algorithmes génétiques – Une étude détaillée sur leur fonctionnement et leurs applications*, Université Abdelhamid Ibn Badis - Mostaganem, 2019. Disponible en ligne : http://e-biblio.univ-mosta.dz/bitstream/handle/123456789/20337/MINF251.pdf?sequence=1