TMJM THE MINDANAWAN JOURNAL OF MATHEMATICS
OFFICIAL JOURNAL OF THE MINDANAO STATE UNIVERSITY - ILIGAN INSTITUTE OF TECHNOLOGY

Vol. 6 (2024), no. 2, pp. 79–111 ISSN: 2094-7380 (Print) | 2783-0136 (Online)
https://doi.org/10.62071/tmjm.v6i2.716.

Performance Analysis of Classical Algorithms for the Traveling Salesman Problem

Rene Luna-Garcia[1], Thricia Mae C. Candano[2,*], and Randy L. Caga-anan[3]

[1]Centro de Investigacion en Computacion
Instituto Politecnico Nacional, Mexico City, Mexico
lunar@cic.ipn.mx

[2,3]Department of Mathematics and Statistics
MSU-Iligan Institute of Technology, 9200 Iligan City, Philippines
thriciamae.candano@g.msuiit.edu.ph, randy.caga-anan@g.msuiit.edu.ph

Received: 20th December 2024 Revised: 3rd January 2025

Abstract

The Traveling Salesman Problem (TSP) is a fundamental optimization problem with wide-ranging applications in logistics, routing, and network design. This paper presents a comprehensive performance analysis of classical algorithms applied to solve the TSP, including exact methods like Brute Force and Dynamic Programming, and heuristic approaches such as Particle Swarm Optimization (PSO), Simulated Annealing (SA), Genetic Algorithms (GA), and k-Nearest Neighbors (KNN). The study evaluates these algorithms across multiple problem instances, varying in scale and complexity, to compare their solution quality, computational efficiency, and scalability.

1 Introduction

The Traveling Salesman Problem (TSP) is one of the most studied optimization problems in mathematics and computer science [9][32][16]. It involves finding the shortest possible route that visits a set of cities exactly once and returns to the starting point. Despite its seemingly simple formulation, the TSP is classified as an NP-hard problem, meaning that the computational resources required to solve it grow exponentially with the number of cities[8][22][30]. This complexity makes the TSP a benchmark for evaluating the performance of optimization algorithms and has inspired extensive research in both theoretical and practical domains.

Classical algorithms for solving the TSP can be broadly categorized into exact methods and heuristic approaches[1][4]. Exact methods, such as Brute Force[27][33] guarantee optimal solutions by exhaustively searching [20]. However, these methods become computationally impractical as the number of cities increases due to their exponential time complexity. In contrast, heuristic and metaheuristic algorithms, including Particle Swarm Optimization (PSO)[19][31], Simulated Annealing (SA)[28], Genetic Algorithms (GA)[7], and k-Nearest Neighbors (KNN)[18], offer a trade-off between solution quality and computational efficiency. These approaches do not

*Corresponding author
2020 Mathematics Subject Classification: 05C85, 90C27, 68Q25
Keywords and Phrases: Algorithm, Optimization, Time Complexity, Traveling Salesman Problem
This research is supported by the DOST-ASTHRDP Student Research Support Fund

guarantee an optimal solution but can provide the best possible solution within a reasonable timeframe, making them suitable for large-scale and real-world applications[4].

This article focuses on analyzing the performance of classical algorithms applied to the TSP. By examining their computational efficiency, solution quality, and scalability across different problem instances, this study aims to provide a comprehensive comparison of these methods. Additionally, the impact of algorithm tuning on heuristic approaches is explored to highlight the role of parameter optimization in improving performance. The findings of this study contribute to a deeper understanding of classical TSP-solving techniques and serve as a foundation for developing more efficient algorithms for real-world optimization challenges.

2 Classical Algorithms

This section details the key outcomes of applying various optimization algorithms to solve the Traveling Salesman Problem (TSP), modeled through graph theory. By analyzing the performance of classical algorithms such as Particle Swarm Optimization Algorithm (PSO), Simulated Annealing (SA),Genetic Algorithm (GA), Greedy Algorithm, Divide and Conquer Algorithm, K-Nearest Neighbor Algorithm (KNN), Dynamic Programming, and Brute Force Algorithm, the results reveal distinct differences in solution quality, computational efficiency, and scalability. The findings highlight which algorithms are more suited for specific problem sizes or conditions, offering valuable insights into the trade-offs between accuracy and resource consumption in optimizing the TSP.

2.1 K-Nearest Neighbors Algorithm (KNN)

The k-nearest neighbors (KNN) algorithm [18] is a simple, non-parametric, supervised learning method used for classification and regression tasks in statistical analysis. It classifies or predicts based on proximity in a multidimensional feature space. During training, the algorithm stores feature vectors and their associated class labels. In the classification phase, a test point is assigned the label most frequent among its k nearest neighbors, with k being a user-defined constant. This makes KNN an intuitive and widely used machine learning algorithm. Its adaptability to high-dimensional feature spaces and its reliance on distance metrics to determine "closeness" have made it a fundamental tool in machine learning, despite challenges such as computational inefficiency with large datasets and sensitivity to irrelevant features[3].
The **K-Nearest Neighbors (KNN) Approach to the Traveling Salesman Problem (TSP)**[17] described in the algorithm **??** aims to provide a heuristic solution for the TSP by leveraging proximity in a distance metric, specifically the Euclidean distance. The algorithm begins with a given set of cities (nodes) and their pairwise distances. A city is randomly selected as the starting point to initiate the tour. For each unvisited city, the algorithm computes the Euclidean distances from the current city to all unvisited cities and identifies the k-nearest neighbors, where k is a predefined constant. The Euclidean distance d between two cities located at coordinates (x_1, y_1) and (x_2, y_2) is calculated as:

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}.$$

Among the k-nearest neighbors, one city is chosen at random as the next destination. This city is added to the tour, and the process repeats until all cities are visited. Finally, the algorithm returns to the starting city to complete the tour, ensuring each city is visited exactly once. The output is a complete tour that approximates an efficient solution to the TSP.

Algorithm 2.1 K-Nearest Neighbors Approach to TSP

Input: Set of cities (nodes) and their pairwise distances
Choose a Starting City: Randomly select a city as the initial point in the route.
for each unvisited city **do**
  From the current city, find the closest $k-$ nearest neighbor based on its Euclidean distance
  Travel to one of the $k-$nearest city and add it to the tour.
end for
After visiting all cities, return to the starting city to complete the loop (tour).
Output: Complete tour visiting all cities once and returning to the starting city

Theorem 2.1. *Let n be the number of cities. The time complexity of the k-nearest neighbors approach for TSP is $O(n^2)$ [29].*

Proof. Let $T(n)$ denote the time complexity of solving the Traveling Salesman Problem (TSP) for n cities. Initially, choosing a starting city from the set of n cities is an $O(1)$ operation, as it only requires a constant-time selection. The main computational cost lies in finding the k-nearest neighbors for each city. For each of the $n-1$ unvisited cities, finding the nearest k cities requires scanning up to $n-1$ cities, which takes $O(n)$ time per city. Repeating this for $n-1$ iterations results in a total time complexity of $O(n) \times O(n-1) = O(n^2)$. Next, the operation of traveling to one of the k-nearest cities and marking it as visited is performed in $O(1)$ time per step. Repeating this for $n-1$ cities contributes $O(n)$ in total. Finally, completing the tour by returning to the starting city is an $O(1)$ operation. Summarizing these results, the overall time complexity is $T(n) = O(1) + O(n^2) + O(n) + O(1)$. The dominant term here is $O(n^2)$, so the total time complexity simplifies to $T(n) = O(n^2)$. 2.1 □

2.2 Dynamic Programming

Dynamic programming (DP) works by solving complex problems through a strategy of breaking them down into smaller, overlapping subproblems. Instead of solving the same subproblem multiple times, DP identifies these subproblems and solves each one independently. Once a subproblem is solved, its solution is stored in a table or array, allowing it to be reused later without recalculating. This storage of solutions is key to how DP avoids redundancy and reduces the overall computation time. As the process continues, DP builds up solutions to progressively larger subproblems using the stored solutions of the smaller ones. This method ensures that each subproblem is solved only once, optimizing efficiency and making it particularly effective for problems with overlapping subproblems and optimal substructure[12]. The **Dynamic Programming Approach to the Traveling Salesman Problem (TSP)** [13] in algorithm 2.2 is an optimization method that utilizes dynamic programming principles to minimize computational redundancy and determine the best possible tour. The algorithm begins by taking as input a set of cities and their pairwise distances, often represented as a distance matrix D, where $D[i][j]$ denotes the distance between city i and city j. The first step involves precomputing this distance matrix, a 2D array containing the distances between all pairs of cities, ensuring efficient distance lookups throughout the algorithm. Next, an initial solution is established by starting the tour at a designated city, such as city 0, and considering the initial cost of visiting one other city from this starting point. This forms the base case for the dynamic programming recurrence. The algorithm then iterates over subsets of cities (excluding the starting city). For each subset S, it computes the minimum cost to visit each city $j \in S$ from another city $i \in S$. The results of previously computed smaller subsets are used to calculate the costs for larger subsets, thereby avoiding redundant calculations. As the algorithm progresses, the results for subsets of increasing size are stored in a table or array. This dynamic programming table allows solutions for larger subsets to build upon those for smaller subsets. Once all subsets have been processed, the algorithm computes the minimum cost of returning to the starting city from any other city in the final subset, completing the tour and determining the total minimum cost. Finally, the algorithm outputs the best possible tour, which is the sequence of cities that minimizes the total travel cost.

Algorithm 2.2 Dynamic Programming Approach to TSP

Input: Set of cities and their pairwise distances
Compute the distance matrix between all pairs of cities.
Start with initial solutions where the tour begins at city 0 and visits one other city.
for each subset of cities **do**
    Calculate the minimum cost to reach each city in the subset from another city in the same subset.
end for
Update the results for each subset size, building up solutions from smaller subsets to larger ones.
Compute the minimum cost to return to the starting city from any city and complete the tour.
Output: Return the optimal tour (sequence of cities) that gives the minimum cost.

Theorem 2.2. *Let n be the number of cities. The time complexity of the dynamic programming approach for TSP is* $O(n^2 2^n)$ *[13].*

Proof. Let $T(n)$ denote the time complexity of solving the Traveling Salesman Problem (TSP) for n cities . This Dynamic programming (DP) algorithm uses a table $dp[S][i]$, where S is a subset of cities that includes the starting city, and i represents the last city visited within this subset. There are 2^n possible subsets of n cities, since each city can either be included or excluded. For each subset S, the algorithm calculates the minimum cost of ending at each of the n cities, resulting in $O(2^n \times n)$ entries in the DP table.
To fill each entry $dp[S][i]$, the algorithm must determine the minimum cost to reach city i by checking each possible preceding city j in the subset S, requiring $O(n)$ operations per entry. Thus, the total complexity for filling the DP table is $O(n)$ operations for each of the $O(2^n \times n)$ entries, yielding an overall time complexity of $O(2^n \times n^2)$.
After the DP table is filled, the algorithm computes the minimum cost to return to the starting city from any city i, which requires $O(n)$ time. However, this final step does not affect the overall complexity significantly, as it is dominated by the complexity of filling the DP table. Consequently, the overall time complexity of the DP approach to TSP is $T(n) = O(2^n \times n^2)$. 2.2 □

2.3 Brute Force Algorithm

A brute force algorithm [27][33] is a general problem-solving technique that involves trying all possible solutions until the correct one is found. It doesn't incorporate any shortcuts or optimizations, making it a very straightforward but inefficient approach for solving computational problems, particularly when the search space is large.
The **Brute Force Approach to the Traveling Salesman Problem (TSP)** [23] is a straightforward algorithm described in algorithm 2.3 that systematically explores all possible solutions to determine the optimal tour. The algorithm takes as input a set of n cities and their pairwise

distances, typically represented in a distance matrix D, where $D[i][j]$ indicates the distance between city i and city j. To simplify the problem, one city is selected as the starting point (e.g., city 1), reducing the problem to finding the optimal order for the remaining $n-1$ cities. Since the starting city is fixed, the remaining $n-1$ cities can be arranged in all possible permutations, resulting in $(n-1)!$ possible tours. Each permutation corresponds to a unique sequence of visiting the remaining cities. For each permutation P_j, the algorithm computes the total travel cost by summing the distances between consecutive cities in the permutation and adding the distance from the last city back to the starting city. Mathematically, the travel cost for a permutation $P_j = [1, i_1, i_2, \ldots, i_{n-1}]$ is calculated as:

$$\text{Cost}(P_j) = D[1][i_1] + D[i_1][i_2] + \cdots + D[i_{n-1}][1].$$

After evaluating all $(n-1)!$ permutations, the algorithm selects the tour with the minimum total cost as the optimal solution. Finally, it outputs the optimal tour, which is the sequence of cities that minimizes the total travel distance, along with the corresponding minimum travel cost. While this approach is simple and guarantees optimality, its factorial growth in computational complexity makes it impractical for large-scale problems.

Algorithm 2.3 Brute Force Approach to TSP

1: **Input:** Set of n cities and their pairwise distances
2: List all n cities and select the initial city.
3: Generate all possible tours for $n-1$ cities, yielding $(n-1)!$ possible permutations since the initial city is fixed and the rest can be permuted.
4: **for** each permutation P_j **do**
5: Compute the total travel cost for P_j.
6: **end for**
7: Select the permutation with the minimum total cost.
8: **Output:** The tour (sequence of cities) with the minimum total cost.

Theorem 2.3. *Let n be the number of cities. The time complexity of the brute force algorithm approach for TSP is $O(n!)$ [23].*

Proof. The total time complexity of the brute force approach is the sum of three main components: generating all permutations, computing the cost for each permutation, and comparing the costs to find the minimum. Let $T(n)$ denote the time complexity of solving the Traveling Salesman Problem (TSP) for n cities. First, generating all permutations involves $(n-1)!$ steps, as there are $(n-1)!$ ways to arrange the remaining cities after choosing the starting city. Next, for each permutation, the algorithm performs n additions to compute the total travel cost, resulting in

$$n \times (n-1)! = n!$$

operations. Finally, comparing the costs of all permutations to determine the minimum requires $(n-1)!$ comparisons. In asymptotic terms, the dominant factor in the time complexity is $n!$, as

$$n \times (n-1)! = n!.$$

Therefore, the overall time complexity of the brute force algorithm for TSP is

$$T(n) = O(n!).$$

2.3 □

2.4 Particle Swarm Optimization

In computational science, particle swarm optimization (PSO)[19][31] is a computational method that optimizes a problem by iteratively trying to improve a candidate solution with regard to a given measure of quality. It solves a problem by having a population of candidate solutions, here dubbed particles, and moving these particles around in the search-space according to simple mathematical formulae over the particle's position and velocity. Each particle's movement is influenced by its local best known position, but is also guided toward the best known positions in the search-space, which are updated as better positions are found by other particles. This is expected to move the swarm toward the best solutions. The **Particle Swarm Optimization (PSO) Approach to the Traveling Salesman Problem (TSP)**[34] in algorithm 2.4 is a heuristic optimization technique inspired by the social behavior of swarms. It adapts the principles of PSO to iteratively improve a population of candidate solutions, called particles, until an optimal or near-optimal tour is found. The algorithm begins by taking as input a set of cities and their pairwise distances, typically represented in a distance matrix. Each particle represents a candidate solution encoded as a sequence of cities, such as $[1, 3, 2, 4]$, which denotes the order in which the cities are visited. Initially, the total distance of the tour represented by each particle is calculated, serving as the fitness measure, with smaller distances indicating better solutions. Each particle's position (tour) is updated based on its *personal best position*, the best tour (lowest cost) it has discovered so far, and the *global best position*, the best tour found by any particle in the swarm. These updates involve local adjustments, such as modifying the current tour by swapping or reordering cities, and guidance toward the personal and global best tours to direct the particle toward promising regions of the search space. The algorithm iterates through a series of updates, where each particle adjusts its position and recalculates its cost. The personal best of each particle is updated if the current position improves upon it, and the global best is updated if any particle's current position outperforms the known global best. This process continues for a predefined number of iterations or until the swarm converges, meaning the particles are no longer significantly improving their solutions. The algorithm stops when one of the criteria, such as reaching the maximum number of iterations or achieving convergence, is met. At this point, the global best particle represents the shortest tour found by the swarm, which is returned as the algorithm's output.

Algorithm 2.4 PSO Approach to TSP

Input: Cities and pairwise distances
Each particle is a tour (city sequence).
Compute the total tour distance for each particle. Minimize this distance.
Update:
for each particle **do**
    Adjust the tour by swapping or reordering cities.
    Move towards personal and global best positions.
end for
Repeat for a set number of iterations or until convergence.
Stop when criteria are met. The global best particle gives the shortest tour.
Output: Shortest tour found.

Theorem 2.4. *Let n be the number of cities. The time complexity of the particle swarm optimization algorithm approach for TSP is $O(n^3)$[35].*

Proof. Let $T(n)$ denote the time complexity needed to solve the Traveling Salesman Problem (TSP) for n cities. First, initializing P particles involves assigning random tours, where each

tour consists of n cities. This requires $P \times n$ operations. Next, for each particle in each iteration, the algorithm computes the fitness (total distance of the tour), which involves n additions. For P particles, this results in $P \times n$ operations per iteration. Additionally, adjusting each particle's tour by swapping or reordering cities and moving it towards the global and personal bests also requires $O(n)$ operations per particle, contributing another $P \times n$ operations per iteration. Combining these steps, the time complexity for one iteration is $O(P \times n)$.
Finally, repeating the process for T iterations results in a total complexity of $T \times P \times n$. In asymptotic terms, assuming $P = O(n)$ (number of particles scales linearly with n) and $T = O(n)$ (number of iterations scales linearly with n), the dominant factor in the time complexity is:

$$T \times P \times n = n \times n \times n = n^3.$$

Therefore, the overall time complexity of the PSO algorithm for TSP is:

$$T(n) = O(n^3).$$

2.4 □

2.5 Simulated Annealing

Simulated Annealing (SA)[28] is a probabilistic optimization technique inspired by the annealing process in physics, where materials are heated and slowly cooled to reach a stable, low-energy state. It is commonly used to find approximate solutions to complex optimization problems. The algorithm works by exploring the solution space and accepting both improvements and occasional worse solutions to avoid getting trapped in local optima. As the "temperature" decreases over time, the algorithm becomes more selective, eventually converging to a near-optimal solution. SA is well-suited for large, non-linear, or combinatorial optimization tasks like the Traveling Salesman Problem (TSP). The **Simulated Annealing (SA) Approach for the Traveling Salesman Problem (TSP)**[10] is a probabilistic optimization method inspired by the physical process of annealing in materials science described in algorithm 2.5. This technique mimics the gradual cooling of materials to reach a stable, low-energy state and applies the same principle to optimization problems, enabling it to escape local optima and find near-optimal solutions. The algorithm begins by taking as input a set of cities and their pairwise distances, typically represented as a distance matrix. It starts with a random tour, which is an arbitrary sequence of cities, and calculates the total distance (cost) of this tour, serving as the initial solution. A high initial temperature T is set, governing the probability of accepting worse solutions. This allows the algorithm to explore the solution space freely in the early stages. A new solution is generated by making a small change to the current tour, such as swapping the positions of two cities, reversing the order of a segment, or other small perturbations. The total distance of this new tour is then calculated and compared with the current solution's cost. If the new solution is better (i.e., has a shorter tour), it is accepted as the current solution. If the new solution is worse, it may still be accepted with a probability given by $P = e^{-\Delta/T}$ [2] [28], where Δ is the difference in cost between the new and current solutions ($\Delta = \text{new cost} - \text{current cost}$). This probabilistic acceptance helps the algorithm escape local optima by occasionally moving to worse solutions, especially at high temperatures. The temperature is gradually reduced according to a cooling schedule, commonly $T = \alpha T$, where α (e.g., 0.9) is a constant between 0 and 1. The algorithm terminates when the temperature becomes sufficiently low, no significant improvement is observed over several iterations, or a maximum number of iterations is reached. Finally, the shortest tour found during the execution is returned as the output.

Algorithm 2.5 Simulated Annealing Approach for TSP

Input: Cities and pairwise distances
Start with a random tour (a sequence of cities). Calculate the total distance of this tour.
Set an initial high temperature T.
Generate a new solution by making a small change to the current tour (e.g., swapping the positions of two cities).
Calculate the total distance of the new tour.
Acceptance Criteria:
if the new tour is shorter (better) **then**
    Accept it as the current solution.
else
    Accept it with a probability: $P = e^{-\Delta/T}$, where Δ is the difference in cost between the new and current solutions.
end if
Reduce the temperature according to a cooling schedule (e.g., $T = \alpha T$, where $0 < \alpha < 1$).
Stop when the temperature is sufficiently low or no significant improvement is observed.
Output: Shortest tour found.

Theorem 2.5. *Let n be the number of cities. The time complexity of the simulated annealing algorithm approach for TSP is $O(n^3)$[5].*

Proof. Let $T(n)$ denote the time complexity of solving the Traveling Salesman Problem (TSP) for n cities. First, the algorithm begins with a random tour consisting of n cities, and calculating the total distance of this tour requires $O(n)$ operations. Generating a new solution involves making a small modification to the current tour, such as swapping two cities, which can be performed in $O(1)$. Calculating the total distance of the new tour again requires $O(n)$, as it involves summing up the distances between consecutive cities in the modified sequence.
The acceptance criteria involve comparing the new tour to the current one, which is an $O(1)$ operation. If the new solution is worse, the algorithm computes the acceptance probability $P = e^{-\Delta/T}$, which is also an $O(1)$ operation. Afterward, the temperature is reduced according to a cooling schedule, which is a constant-time operation ($O(1)$).
The process of generating, evaluating, and accepting/rejecting new solutions is repeated for a set number of iterations I, which depends on the cooling schedule and the convergence criteria. Thus, the time complexity per iteration is $O(n)$ for evaluating the tour, and for I iterations, the total time complexity becomes $O(I \times n)$.
In asymptotic terms, the total time complexity is dominated by $I \times n$. Assuming $I = O(n^2)$, a common practical choice for ensuring sufficient exploration of the solution space, the overall time complexity of the Simulated Annealing algorithm for TSP is:

$$T(n) = O(n^3).$$

2.5 □

2.6 Genetic Algorithm

A Genetic Algorithm (GA)[7][24] is a heuristic optimization technique inspired by the process of natural selection in biological evolution. It is particularly useful for solving complex problems where traditional optimization methods struggle, especially in problems with large search spaces. GA belongs to the class of evolutionary algorithms (EAs) and works by evolving a population of candidate solutions toward an optimal or near-optimal solution through processes analogous to selection, crossover (recombination), and mutation. The **Genetic Algorithm (GA) Approach for the Traveling Salesman Problem (TSP)**[14] is a heuristic optimization technique inspired by the principles of natural selection and biological evolution. It belongs to the class of evolutionary algorithms and is particularly effective for solving optimization problems with large and complex search spaces, such as the TSP described in algorithm2.6. The algorithm begins by taking as input a set of cities and their pairwise distances, typically represented in a distance matrix. Each individual in the population, also known as a chromosome, represents a candidate solution to the TSP, encoded as a sequence of cities (e.g., $[1, 3, 2, 4]$ represents a tour visiting city 1, then city 3, and so on). The fitness of each chromosome is determined by the total distance of the tour it represents, with shorter tours having higher fitness, as the goal is to minimize the total travel distance. Parents for reproduction are selected based on their fitness. Selected parents are paired to produce offspring through crossover (recombination), which combines parts of the parent chromosomes to create new solutions. To maintain genetic diversity and prevent premature convergence to suboptimal solutions, mutations are introduced in the offspring. Common mutation techniques include swap mutation, where the positions of two cities in the tour are randomly swapped, and reverse mutation, which reverses the order of cities in a randomly selected segment of the tour. The algorithm iterates through generations, repeating the selection, crossover, and mutation steps. It terminates when a fixed number of generations is reached or when a satisfactory solution, such as a tour below a specific distance, is found. Finally, the algorithm outputs the best tour discovered during its execution, representing the shortest travel route identified by the population.

Algorithm 2.6 Genetic Algorithm Approach for TSP

1: **Input:** Cities and pairwise distances
2: Each individual (chromosome) represents a tour (sequence of cities).
3: The fitness of each chromosome is based on the total distance of the tour it represents.
4: Select parents for reproduction based on their fitness.
5: Apply crossover to selected parents to produce offspring.
6: Introduce genetic diversity to prevent premature convergence. Common mutations include:

- Randomly swap two cities in the sequence.
- Reverse a subsection of the route.

7: Stop after a fixed number of generations or when a satisfactory solution has been found.
8: **Output:** Best tour found.

In [14], the time complexity of the genetic algorithm for the Traveling Salesman Problem (TSP) is reported as $O(n^2)$. However, in our analysis, we derive a complexity of $O(n^3)$ when accounting for the number of iterations over G generations. Typically, G is chosen to be proportional to n, which justifies the higher complexity in our model.

To elaborate, for example, in the genetic algorithm approach, the commonly cited complexity of $O(n^2)$ assumes a fixed starting city and does not simulate across all n cities. In contrast, our work includes simulations over all n cities and considers the iterative processes across generations, thereby providing a more comprehensive analysis of the algorithm's behavior.

Theorem 2.6. *Let n be the number of cities. The time complexity of the genetic algorithm approach for TSP is $O(n^3)$.*

Proof. Let $T(n)$ denote the time complexity of solving the Traveling Salesman Problem (TSP) for n cities. First, the algorithm starts by initializing a population of P individuals (chromo-

https://doi.org/10.62071/tmjm.v6i2.716

somes), where each chromosome represents a tour consisting of n cities. Generating P random tours requires $O(P \times n)$ operations. Next, the fitness of each individual is evaluated by calculating the total distance of its tour, which involves summing the pairwise distances of n cities. This requires $O(n)$ operations per chromosome, leading to $O(P \times n)$ for the entire population. In the selection step, parents are chosen based on their fitness, typically using methods like roulette wheel selection or tournament selection. This step involves sorting or ranking the population based on fitness, which has a complexity of $O(P \log P)$. Selection itself for P individuals generally requires $O(P)$ operations.
Crossover is applied to the selected parents to generate offspring. Depending on the crossover operator (e.g., partially matched crossover or order crossover), the operation typically takes $O(n)$ for each pair of parents. For P individuals, this results in $O(P \times n)$ operations for crossover.
Mutations, such as swapping two cities or reversing a subsection of the route, are applied to introduce diversity. Each mutation operates on a chromosome and takes $O(n)$ in the worst case. For P individuals, mutation contributes another $O(P \times n)$ operations per generation.
These steps are repeated for G generations. The total time complexity per generation is dominated by the fitness evaluation, crossover, and mutation steps, each contributing $O(P \times n)$. Over G generations, the total time complexity becomes $O(G \times P \times n)$.
In practical applications, P (population size) and G (number of generations) are often chosen proportional to n, i.e., $P = O(n)$ and $G = O(n)$. Substituting these values, the total time complexity can be expressed as:

$$T(n) = O(n \times n \times n) = O(n^3).$$

2.6 □

2.7 Greedy Algorithm

A greedy algorithm[25] is a problem-solving approach that makes the locally optimal choice at each step with the hope of finding a global optimum solution. This method does not consider the long-term consequences of each choice but focuses on the best immediate option available. The key idea is to select the best possible choice at each step, leading to a solution that may not always be the most optimal but is often good enough for many problems.
The **Greedy Algorithm Approach for the Traveling Salesman Problem (TSP)**[15] is a heuristic method that constructs a solution incrementally by making a series of locally optimal decisions. The algorithm 2.7 begins by taking as input a set of cities and their pairwise distances, typically represented as a distance matrix. It starts at an arbitrary city, chosen as the starting point for the tour. While there are still unvisited cities, the algorithm identifies the nearest unvisited city based on the given distances, travels to it, and marks it as visited. This iterative selection ensures that each move minimizes the immediate travel distance from the current city. Once all cities have been visited, the algorithm returns to the starting city to complete the tour, forming a closed loop. The final output of the algorithm is the complete tour, which is the sequence of cities visited, along with the total distance of the tour.

Algorithm 2.7 Greedy Algorithm Approach for TSP

1: **Input:** Cities and pairwise distances
2: Begin at an arbitrary city.
3: **while** not all cities have been visited **do**
4: Find the nearest unvisited city (based on the distance) and travel to it.
5: Mark the city as visited.
6: **end while**
7: Return to the starting city to complete the tour.
8: **Output:** Complete tour with the total distance.

Theorem 2.7. *Let n be the number of cities. The time complexity of the greedy algorithm approach for TSP is $O(n^2)$ [26].*

Proof. Let $T(n)$ denote the time complexity of solving the Traveling Salesman Problem (TSP) for n cities. The algorithm begins at an arbitrary city, which is selected in $O(1)$ time as this is a constant-time operation. At each step, the algorithm searches for the nearest unvisited city. Finding the nearest city involves checking the distances from the current city to all unvisited cities, which requires $O(n)$ comparisons in the worst case. Since this step is repeated for $n-1$ cities as the algorithm progressively visits all cities, the total complexity of this step is $O((n-1) \times n) = O(n^2)$.
Additionally, marking a city as visited is a constant-time operation ($O(1)$) and is performed $n-1$ times, contributing $O(n)$ in total. Finally, returning to the starting city to complete the tour takes $O(1)$ time. These lower-order operations ($O(1)$ and $O(n)$) do not affect the overall time complexity, which is dominated by the repeated search for the nearest unvisited city.
Therefore, the total time complexity of the Greedy Algorithm for TSP is $T(n) = O(n^2)$. 2.7 □

2.8 Divide and Conquer Algorithm

The Divide and Conquer algorithm [11] is a problem solving strategy that involves breaking down a complex problem into smaller, more manageable parts, solving each part individually, and then combining the solutions to solve the original problem. The **Divide and Conquer Algorithm for the Traveling Salesman Problem (TSP)**[6] is a heuristic approach that applies the divide-and-conquer strategy to break down the problem into smaller, more manageable subproblems, solve each subproblem independently, and then combine the results to form a solution for the original problem. The algorithm 2.8 takes as input a set of cities, typically provided with their coordinates and pairwise distances. To introduce variation and improve the chances of finding a better solution, the cities are shuffled randomly. The set of cities is then recursively divided into two subsets based on their x-coordinates or y-coordinates, chosen randomly to avoid bias. This step reduces the complexity by focusing on smaller subsets of the original problem. If a subset contains only one city, it is considered solved since no connections are required. For subsets with two cities, the solution is simply the edge (direct connection) between the two cities. For subsets with more than two cities, the algorithm recursively applies itself to solve the TSP for each subset independently. Once the solutions for the subsets are obtained, they are merged by finding the shortest connection (bridge) between the two subsets, minimizing the total travel distance. The merged solution is then adjusted to ensure there are no repeated cities or unnecessary connections. The final output is a merged solution representing a complete tour of all cities, along with the total tour distance.

Algorithm 2.8 Divide and Conquer Approach for TSP

```
Input: A set of cities with coordinates and pairwise distances
Randomly shuffle the cities to introduce variation.
Recursively split the cities into two subsets based on their x or y coordinates (chosen randomly).
Base Case:
if the number of cities in a subset is 1 then
    Return the city as the solution for this subset.
else if the number of cities in a subset is 2 then
    Return the edge connecting the two cities as the solution for this subset.
end if
Recursively solve the TSP for each subset.
Merge the solutions from the two subsets by finding the shortest connection (bridge) between the subsets to minimize total distance.
Adjust the merged solution to ensure no cities are repeated.
Output: The merged solution with the total tour distance.
```

Theorem 2.8. *Let n be the number of cities. The time complexity of the divide and conquer approach for TSP is $O(n^2)$[21].*

Proof. Let $T(n)$ denote the time complexity of solving the Traveling Salesman Problem (TSP) for n cities using the Divide and Conquer approach. The algorithm begins by dividing the n cities into two subsets of approximately equal size. Solving each subset independently results in the recurrence relation:

$$T(n) = 2T\left(\frac{n}{2}\right) + M(n),$$

where $2T\left(\frac{n}{2}\right)$ represents the cost of solving the two subsets recursively, and $M(n)$ is the cost of merging the solutions from the subsets. The merging step involves finding the shortest connections (bridges) between the two subsets. If each subset contains $n/2$ cities, the algorithm evaluates all pairwise connections between cities in the subsets. This results in:

$$M(n) = O\left(\frac{n}{2} \cdot \frac{n}{2}\right) = O\left(\frac{n^2}{4}\right) = O(n^2).$$

At the base case of the recursion, when the number of cities is 1 or 2, the problem is solved in constant time, $O(1)$. The recurrence $T(n) = 2T\left(\frac{n}{2}\right) + O(n^2)$ unfolds as follows:

i. At the first level of recursion, the merging cost is $O(n^2)$.

ii. At the second level, there are two subproblems of size $n/2$, each with a merging cost of $O\left(\left(\frac{n}{2}\right)^2\right) = O\left(\frac{n^2}{4}\right)$. The total merging cost for this level is:

$$2 \cdot O\left(\frac{n^2}{4}\right) = O\left(\frac{n^2}{2}\right).$$

iii. At the third level, there are four subproblems of size $n/4$, with a merging cost of:

$$4 \cdot O\left(\left(\frac{n}{4}\right)^2\right) = 4 \cdot O\left(\frac{n^2}{16}\right) = O\left(\frac{n^2}{4}\right).$$

At each level of recursion, the total merging cost is halved relative to the previous level. The merging costs form a geometric series:

$$O(n^2) + O\left(\frac{n^2}{2}\right) + O\left(\frac{n^2}{4}\right) + \dots$$

The sum of this series converges to $O(n^2)$, as the total cost of merging diminishes exponentially across levels of recursion.
Thus, the overall time complexity of the algorithm is dominated by the merging cost, which is $O(n^2)$. The splitting step, contributing $O(n)$ at each level, sums to $O(n \log n)$ over all levels, but it is asymptotically smaller than the merging cost. Therefore, the time complexity of the Divide and Conquer TSP algorithm is:

$$T(n) = O(n^2).$$

2.8 $\square$

Table 1: Time Complexity of the Different Algorithms Used

Algorithm	**Time Complexity**
PSO	$O(n^3)$
SA	$O(n^3)$
GA	$O(n^3)$
Greedy	$O(n^2)$
Divide & Conquer	$O(n^2)$
Dynamic Programming	$O(n^2 2^n)$
Brute Force	$O(n!)$
KNN	$O(n^2)$

https://doi.org/10.62071/tmjm.v6i2.716

Table 2: Experimental Results of Various Algorithms Across Different City Instances each with 10 Simulations

Cities	Algorithm	Best	Worst	Average cost	Variance	Std Dev.	Average time
cities_8	PSO	2379.6953	2379.6953	2379.6953	$2.068e^{-25}$	$4.547e^{-13}$	7.798 s
	SA	2379.6953	2379.6953	2379.6953	$2.068e^{-25}$	$4.547e^{-13}$	0.012 s
	GA	2379.6953	2379.653	2379.653	$2.068e^{-25}$	$4.547e^{-13}$	156.188 s
	Greedy	2379.670	3047.029	2732.819	47880.085	218.815	0.760 s
	Divide & Conquer	2379.695	3140.935	2688.423	110818.167	332.894	0.472 s
	NN	2379.695	3047.029	2762.610	34218.674	184.982	2.60 s
	KNN	2379.695	2776.642	2640.423	21031.975	145.023	2.85 s
	Dynamic Programming	2238.426	2892.123	2685.128	53516.457	231.336	0.003 s
	Brute Force	1974.371	3443.446	2557.793	172725.111	415.602	9.757 s
cities_16	PSO	74.001	75.477	74.592	0.240	0.490	12.060 s
	SA	74.378	77.546	75.653	0.824	0.908	0.014 s
	GA	73.988	74.152	74.105	0.006	0.074	148.432 s
	Greedy	77.127	97.200	91.094	41.825	6.467	1.457 s
	Divide & Conquer	76.869	105.284	87.948	63.340	8.327	0.995 s
	NN	84.428	104.735	90.154	59.677	7.725	5.19 s
	KNN	77.127	104.077	89.071	58.467	7.646	6.25 s
	Dynamic Programming	2815.412	3811.006	3290.850	91403.593	302.33	4.585 s
	Brute Force	—	—	—	—	—	—
cities_32	PSO	890.733	1146.593	992.360	5966.231	77.241	15.734 s
	SA	872.671	1047.484	955.515	2868.764	53.561	0.023 s
	GA	907.453	936.108	910.318	73.890	8.596	138.081 s
	Greedy	917.178	1194.209	1038.795	9970.301	99.851	2.662 s
	Divide & Conquer	1101.076	1390.162	1204.035	15327.999	123.806	1.966 s
	NN	917.178	1219.846	1105.907	8894.260	94.309	10.27 s
	KNN	917.178	1382.142	1104.951	14650.903	121.041	12.51 s
	Dynamic Programming	—	—	—	—	—	—
	Brute Force	—	—	—	—	—	—
cities_64	PSO	7533.121	8743.234	8025.122	163909.512	404.857	25.141 s
	SA	7082.393	8203.696	7742.635	103340.357	321.466	0.052 s
	GA	6743.790	7381.666	7126.282	38262.253	195.607	161.697 s
	Greedy	7909.445	9415.389	8613.979	180868.547	425.286	4.661 s
	Divide & Conquer	7663.135	11361.523	9381.196	1029031.23	1014.412	3.891 s
	NN	8113.459	8901.467	8483.541	76479.733	276.550	19.97 s
	KNN	7915.697	9176.967	8450.199	149480.429	386.627	20.15 s
	Dynamic Programming	—	—	—	—	—	—
	Brute Force	—	—	—	—	—	—
cities_128	PSO	1045.092	1145.378	1086.321	945.659	30.752	53.570 s
	SA	1073.724	1163.183	1118.767	989.023	31.449	0.097 s
	GA	969.470	1011.357	997.047	190.693	13.809	180.485 s
	Greedy	1026.946	1136.787	1076.326	974.630	31.219	9.165 s
	Divide & Conquer	1148.457	1600.845	1340.903	13999.907	118.321	8.229 s
	NN	1048.324	1115.780	1089.073	463.128	21.520	41.12 s
	KNN	1027.139	1132.151	1089.436	712.059	26.684	41.94 s
	Dynamic Programming	—	—	—	—	—	—
	Brute Force	—	—	—	—	—	—
cities_256	PSO	1424.557	1575.723	1505.023	1856.563	43.088	102.818 s
	SA	1396.215	1728.325	1546.455	9052.550	95.145	0.175 s
	GA	1342.939	1466.473	1435.518	1345.239	36.677	208.324 s
	Greedy	1390.980	1562.452	1475.609	2153.972	46.411	21.591 s
	Divide & Conquer	1736.906	2252.505	1982.109	27081.974	164.566	20.320 s
	NN	1483.161	1570.500	1514.891	833.613	28.872	82.72 s
	KNN	1400.648	1567.968	1474.610	3485.294	59.036	83.06 s
	Dynamic Programming	—	—	—	—	—	—
	Brute Force	—	—	—	—	—	—

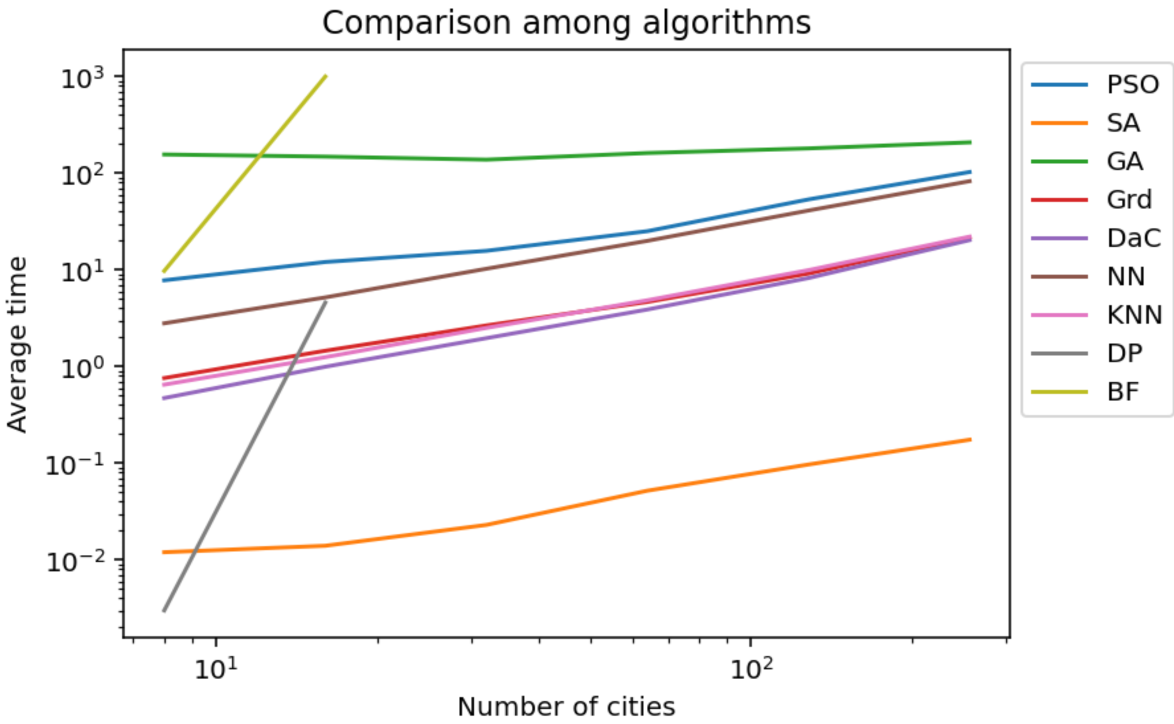

Figure 1: Performance Comparison of all algorithms

3 Discussion of the Experimental Results and Comparison among Algorithms

The graph in Figure 1 illustrates a performance comparison of various classical algorithms applied to the Traveling Salesman Problem (TSP). The algorithms evaluated include Particle Swarm Optimization (PSO), Simulated Annealing (SA), Genetic Algorithm (GA), Greedy Algorithm (Grd), Divide and Conquer (DaC), Nearest Neighbor (NN), k-Nearest Neighbors (KNN), Dynamic Programming (DP), and Brute Force (BF). The performance is assessed using the average computation time (y-axis) plotted against the number of cities (x-axis), both presented on a logarithmic scale.

From the results, the Brute Force algorithm clearly emerges as the slowest, consistent with its factorial time complexity ($O(n!)$). Dynamic Programming performs better than Brute Force but demonstrates an exponential increase in computation time as the number of cities grows, reflecting its $O(n^2 \cdot 2^n)$ complexity. Heuristic methods like Nearest Neighbor (NN) and k-Nearest Neighbors (KNN) show better scalability but do not consistently outperform optimization techniques such as Simulated Annealing or Genetic Algorithm.

Simulated Annealing strikes a balance between performance and scalability, showing consistent efficiency across varying problem sizes. Its probabilistic acceptance criterion enables it to escape local optima, making it a robust choice for solving TSP. Similarly, Genetic Algorithm and Particle Swarm Optimization are competitive, with PSO demonstrating particularly stable performance for larger problem sizes, suggesting its adaptability to complex solution spaces.

Deterministic approaches such as Divide and Conquer and the Greedy Algorithm perform well for smaller instances of the TSP but lose efficiency and solution quality as the problem size increases. This highlights their limitation in solving larger-scale, complex optimization problems. The comparison underscores the impracticality of exact methods like Brute Force and

https://doi.org/10.62071/tmjm.v6i2.716

Dynamic Programming for anything beyond small problem instances due to their computational demands.
Overall, the results emphasize the suitability of metaheuristic algorithms such as Simulated Annealing, Genetic Algorithm, and Particle Swarm Optimization for balancing computational efficiency and solution quality, especially for large-scale TSP instances. These approaches outperform exact methods and simpler heuristics, demonstrating their value in practical applications of TSP.

3.1 Algorithm Performance Variations

The results indicate notable variations in the performance of the algorithms applied to the Traveling Salesman Problem (TSP). The Brute Force approach produces the exact solution since it evaluates all possible routes, but it quickly becomes computationally infeasible as the number of cities increases due to its exponential time complexity. Dynamic Programming (DP) also delivers near-optimal solutions but exhibits increased variance as the problem size grows, reflecting its sensitivity to scaling. On the other hand, metaheuristic algorithms like Particle Swarm Optimization (PSO), Simulated Annealing (SA), and Genetic Algorithms (GA) present diverse performance profiles. PSO often yields competitive results with shorter computation times, particularly for smaller problem instances, while SA demonstrates stability but requires careful tuning to enhance convergence for larger datasets. GA, though robust and effective, appears to be slightly slower compared to PSO in terms of execution time.

3.2 Scalability

Scalability is a critical factor when addressing larger TSP instances, and the results clearly illustrate the limitations of exact algorithms like Brute Force and Dynamic Programming. Both methods exhibit exponential growth in computation time, rendering them impractical for problems with a high number of cities. In contrast, metaheuristic algorithms such as PSO and GA showcase significant scalability, as indicated by their consistent average computation times even as the number of cities increases. This makes them more suitable for real-world applications where problem sizes tend to be larger and computational efficiency is critical.

3.3 Variance in Solution Quality

Another important observation is the variance in solution quality across different algorithms. Metaheuristic approaches, being probabilistic in nature, show higher variance compared to deterministic methods like Brute Force and DP. While exact algorithms guarantee optimal solutions, metaheuristics rely on heuristics and randomness, which occasionally lead to suboptimal solutions. This variability underscores the importance of algorithm tuning and parameter optimization. Interestingly, the K-Nearest Neighbor (KNN) algorithm, likely utilized as a classifier or heuristic for TSP implementation, demonstrates a good balance between accuracy and computational efficiency, offering a promising alternative in specific scenarios.

3.4 Standard Deviation and Average Time

Standard Deviation and Average Times The comparison of standard deviations and average computation times reveals further insights into the behavior of these algorithms. Metaheuristic algorithms like PSO, SA, and GA clearly outperform exact methods in terms of average execution time, particularly for larger instances. This highlights their efficiency and adaptability in tackling large-scale problems. However, the higher standard deviations observed in PSO and GA suggest inconsistencies in their performance. Fine-tuning their parameters, such as adjusting the cooling schedule in SA or mutation rates in GA, could lead to more consistent and reliable solutions.

Acknowledgements

The authors are grateful to the anonymous referees for their helpful comments and suggestions.

References

[1] A. Baltz et al., *Exact and Heuristic Algorithms for the Travelling Salesman Problem with Multiple Time Windows and Hotel Selection*, Journal of the Operational Research Society, vol. 65, no. 9, pp. 1388–1402, 2014, doi:10.1057/jors.2014.17.

[2] Algorithm Afternoon, *Simulated Annealing: Chapter 3*, Algorithm Afternoon, Accessed January 2, 2025, https://algorithmafternoon.com/books/simulated_annealing/chapter03/.

[3] Baeldung, *K-Nearest Neighbors (KNN)*, https://www.baeldung.com/cs/k-nearest-neighbors.

[4] Baeldung, *TSP: Exact Solutions vs Heuristic vs Approximation Algorithms*, Baeldung on Computer Science, 2022, https://www.baeldung.com/cs/tsp-exact-solutions-vs-heuristic-vs-approximation-algorithms.

[5] B. Manthey and J. van Rhijn, *Towards a Lower Bound for the Average Case Runtime of Simulated Annealing on TSP*, arXiv preprint arXiv:2208.11444, 2022, https://arxiv.org/abs/2208.11444.

[6] CodingDrills, *Divide and Conquer Algorithms: Divide and Conquer on Graphs - Traveling Salesman Problem*, https://www.codingdrills.com/tutorial/introduction-to-divide-and-conquer-algorithms/travelling-salesman-prob.

[7] D. E. Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning*, Addison-Wesley, 1989, https://archive.org/details/geneticalgorithm0000gold.

[8] D. L. Applegate et al., *The Traveling Salesman Problem: A Computational Study*, Princeton University Press, 2007, https://press.princeton.edu/books/hardcover/9780691129938/the-traveling-salesman-problem.

[9] E. L. Lawler et al., *The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization*, Wiley, 1985, https://archive.org/details/travelingsalesma00lawl,

[10] Fourmilab, *Simulated Annealing: The Travelling Salesman Problem*, https://www.fourmilab.ch/documents/travelling/anneal/.

[11] GeeksforGeeks, *Divide and Conquer Algorithm: Explanation and Examples*, https://www.geeksforgeeks.org/divide-and-conquer/.

[12] GeeksforGeeks, *Dynamic Programming*, https://www.geeksforgeeks.org/dynamic-programming/.

[13] GeeksforGeeks, *Travelling Salesman Problem using Dynamic Programming*, https://www.geeksforgeeks.org/travelling-salesman-problem-using-dynamic-programming/.

[14] GeeksforGeeks, *Traveling Salesman Problem using Genetic Algorithm*, https://www.geeksforgeeks.org/traveling-salesman-problem-using-genetic-algorithm/.

[15] GeeksforGeeks, *Travelling Salesman Problem — Greedy Approach*, https://www.geeksforgeeks.org/travelling-salesman-problem-greedy-approach/.

[16] G. Gutin and A. P. Punnen (Eds.), *The Traveling Salesman Problem and Its Variations*, Springer, 2006, https://link.springer.com/book/10.1007/b101971.

[17] Hampden-Sydney College, *The Nearest-Neighbor Algorithm for the Traveling Salesman Problem*, https://people.hsc.edu/faculty-staff/robbk/Math111/Lectures/Fall\%202016/Lecture\%2033\%20-\%20The\%20Nearest-Neighbor\%20Algorithm.pdf.

[18] IBM, *What is the k-nearest neighbors algorithm?*, https://www.ibm.com/topics/knn.

[19] MDPI, *Particle Swarm Optimization: A Survey of Historical and Recent Developments*, https://www.mdpi.com/2504-4990/1/1/10.

[20] MIT Open Course Ware, *Optimization Methods in Management Science*, Lecture 17, Massachusetts Institute of Technology, 2013, https://ocw.mit.edu/courses/15-053-optimization-methods-in-management-science-spring-2013/ad6650acdf97013e60e559903d8d25fa_MIT15_053S13_lec17.pdf.

[21] M. Moreno Maza, *The Complexity of Divide-and-Conquer Algorithms*, University of Western Ontario, https://www.csd.uwo.ca/~mmorenom/CS874/Lectures/Introduction.html/node16.html.

[22] M. Weiss, *Proof that Traveling Salesman Problem is NP-hard,* Geeks for Geeks, 2023, https://www.geeksforgeeks.org/proof-that-traveling-salesman-problem-is-np-hard/.

[23] OpenGenus IQ, *Travelling Salesman Problem (Basics + Brute force approach)*, https://iq.opengenus.org/travelling-salesman-problem-brute-force/.

[24] P. Larrañaga et al., *Genetic Algorithms for the Travelling Salesman Problem: A Review of Representations and Operators*, Artificial Intelligence Review, vol. 13, no. 2, pp. 129–170, 1999, doi:10.1023/A:1006529012972.

[25] Programiz, *Greedy Algorithm: Explanation and Examples*, https://www.programiz.com/dsa/greedy-algorithm.

[26] P. Toth and D. Vigo, *The Vehicle Routing Problem*, SIAM Monographs on Discrete Mathematics and Applications, 2002, Chapter 3. https://epubs.siam.org/doi/book/10.1137/1.9780898718515.

[27] R. T. Koether, *The Traveling Salesman Problem – Brute Force Method*, Hampden-Sydney College, 2018, https://people.hsc.edu/faculty-staff/robbk/Math111/Lectures/Spring

[28] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, *Optimization by Simulated Annealing*, Science, vol. 220, no. 4598, pp. 671–680, 1983, doi:10.1126/science.220.4598.671.

[29] Sindhu Kumari, *Traveling Salesman Problem*, Medium, 2020, https://sindhukumari.medium.com/traveling-salesman-problem-9deb6853ac6.

[30] Tutorialspoint, *Proof that Travelling Salesman Problem is NP Hard*, https://www.tutorialspoint.com/proof-that-travelling-salesman-problem-is-np-hard.

[31] Wikipedia, *Particle Swarm Optimization*, https://en.wikipedia.org/wiki/Particle_swarm_optimization.

[32] W. J. Cook, *In Pursuit of the Traveling Salesman: Mathematics at the Limits of Computation*, Princeton University Press, 2012, https://archive.org/details/inpursuitoftrave0000cook.

[33] WsCube Tech, *Traveling Salesman Problem*, WsCube Tech Resources, https://www.wscubetech.com/resources/dsa/travelling-salesman-problem.

[34] X. Lian, Y. Wang, and Y. Shao, *Particle Swarm Optimization for Traveling Salesman Problem*, in *Proceedings of the 2004 International Conference on Machine Learning and Cybernetics*, vol. 7, pp. 3972–3975, 2004, doi:10.1109/ICMLC.2004.1382228.

[35] Y. Shi, H. Teng, and Z. Li, *An Improved Particle Swarm Optimization Algorithm for Traveling Salesman Problem*, Proceedings of the 2010 International Conference on Machine Learning and Cybernetics, IEEE, 2010, https://ieeexplore.ieee.org/document/5583011.

4 Appendices

This section presents a collection of Python programming implementations for various classical algorithms. These codes are foundational to solving optimization problems and demonstrate essential computational approaches. Each example highlights the logic and structure of the algorithm, serving as a practical reference for understanding their functionality. From basic implementations to more intricate procedures, this section aims to bridge the gap between theoretical concepts and practical application in Python.

4.1 KNN Algorithm Python Code

K-Nearest Neighbors Algorithm for TSP

```python
import random
import math

class City:
    def __init__(self, x, y, name=None):
        self.x = x
        self.y = y
        self.name = name

    def distance(self, other):
        """Calculate Euclidean distance to another city."""
        return math.sqrt((self.x - other.x)**2 + (self.y - other.y)**2)

    def __repr__(self):
        return f"City({self.name}, {self.x}, {self.y})" if self.name else f"City({self.x}, {self.y})"

def path_cost(route):
    """Calculate the total cost of a route."""
    cost = 0
    for i in range(len(route) - 1):
        cost += route[i].distance(route[i + 1])
    return cost

def knn_tsp(cities, k=1):
    """Solve TSP using the K-Nearest Neighbors (KNN) approach."""
    start_city = random.choice(cities)
    route = [start_city]
    unvisited = [city for city in cities if city != start_city]

    while unvisited:
        current_city = route[-1]
        nearest_neighbors = sorted(
            unvisited, key=lambda city: current_city.distance(city)
        )[:k]
        next_city = min(nearest_neighbors, key=lambda city: current_city.distance(city))
        route.append(next_city)
        unvisited.remove(next_city)

    route.append(start_city)
    total_cost = path_cost(route)
```

```python

    return route, total_cost
```

4.2 Dynamic Programming Python Code

Dynamic Programming for TSP

```python
import math
import itertools

# Function to calculate Euclidean distance
def calculate_distance(city1, city2):
    return math.sqrt((city1[0] - city2[0])**2 + (city1[1] -
        city2[1])**2)

# Function to solve TSP using Held-Karp Algorithm
def held_karp_tsp(cities):
    """
    Solve the Traveling Salesman Problem using the Held-Karp algorithm.

    Parameters:
        cities (list): List of tuples representing city coordinates
            [(x1, y1), (x2, y2), ...]

    Returns:
        tuple: (minimum cost, optimal path)
    """
    n = len(cities)
    distances = [[calculate_distance(cities[i], cities[j]) for j in
        range(n)] for i in range(n)]

    # Initialize DP table
    dp = {}
    for i in range(1, n):
        dp[(1 << i, i)] = distances[0][i]  # Cost to reach city i
            directly from city 0

    # Iterate through subsets of increasing size
    for subset_size in range(2, n):
        for subset in itertools.combinations(range(1, n), subset_size):
            bits = sum(1 << i for i in subset)
            for current in subset:
                prev_bits = bits & ~(1 << current)
                dp[(bits, current)] = min(
                    dp[(prev_bits, k)] + distances[k][current]
                    for k in subset if k != current
                )

    # Find the minimum cost to complete the tour
    bits = (1 << n) - 1  # All cities visited
    optimal_cost = min(
        dp[(bits & ~(1 << 0), i)] + distances[i][0] for i in range(1,
            n)
    )

```


https://doi.org/10.62071/tmjm.v6i2.716

```
    # Reconstruct the optimal path
    path = [0]
    last = 0
    visited = bits
    for _ in range(n - 1):
        next_city = min(
            range(n),
            key=lambda i: dp.get((visited & ~(1 << i), i), float('inf')) + distances[i][last]
            if visited & (1 << i) else float('inf')
        )
        path.append(next_city)
        visited &= ~(1 << next_city)
        last = next_city

    path.append(0)  # Return to starting city
    return optimal_cost, path


# Example Usage
if __name__ == "__main__":
    # Example cities as (x, y) coordinates
    cities = [
        (0, 0), (2, 3), (5, 2), (7, 7), (3, 8), (9, 1), (4, 6)
    ]

    cost, path = held_karp_tsp(cities)
    print("Optimal Path:", path)
    print("Minimum Cost:", cost)
```

4.3 Brute Force Algorithm Python Code

Brute Force Algorithm for TSP

```
from itertools import permutations
import math

# Function to calculate the Euclidean distance between two cities
def calculate_distance(city1, city2):
    return math.sqrt((city1[0] - city2[0])**2 + (city1[1] - city2[1])**2)

# Brute Force TSP Solver
def brute_force_tsp(cities):
    """
    Solve the Traveling Salesman Problem using the Brute Force approach.

    Parameters:
        cities (list): List of tuples representing city coordinates [(x1, y1), (x2, y2), ...]

    Returns:
        tuple: (minimum cost, optimal path)
    """
```

```python
    n = len(cities)
    all_permutations = permutations(range(n))  # Generate all
        permutations of city indices
    min_cost = float('inf')  # Initialize the minimum cost to infinity
    best_path = None  # Initialize the best path

    # Evaluate each permutation
    for perm in all_permutations:
        current_cost = 0
        # Calculate the total travel cost for this permutation
        for i in range(n - 1):
            current_cost += calculate_distance(cities[perm[i]],
                cities[perm[i + 1]])
        # Add the cost of returning to the starting city
        current_cost += calculate_distance(cities[perm[-1]],
            cities[perm[0]])

        # Update the minimum cost and best path
        if current_cost < min_cost:
            min_cost = current_cost
            best_path = perm

    # Reconstruct the optimal path
    optimal_path = [cities[i] for i in best_path]
    return min_cost, optimal_path


# Example Usage
if __name__ == "__main__":
    # Example cities as (x, y) coordinates
    cities = [
        (0, 0), (2, 3), (5, 2), (7, 7), (3, 8)
    ]

    # Solve TSP using Brute Force
    cost, path = brute_force_tsp(cities)
    print("Optimal Path (Coordinates):", path)
    print("Minimum Cost:", cost)
```

4.4 PSO Algorithm Python Code

Particle Swarm Optimization (PSO) Algorithm for TSP

```python
import random
import math

# Function to calculate the Euclidean distance between two cities
def calculate_distance(city1, city2):
    return math.sqrt((city1[0] - city2[0])**2 + (city1[1] -
        city2[1])**2)

# Function to calculate the total tour distance for a given sequence
    of cities
def calculate_tour_distance(cities, sequence):
    distance = 0
```


https://doi.org/10.62071/tmjm.v6i2.716

```
    for i in range(len(sequence) - 1):
        distance += calculate_distance(cities[sequence[i]],
            cities[sequence[i + 1]])
    distance += calculate_distance(cities[sequence[-1]],
        cities[sequence[0]])  # Return to start
    return distance

# Particle class for PSO
class Particle:
    def __init__(self, num_cities):
        self.position = random.sample(range(num_cities), num_cities)
            # Random city sequence
        self.velocity = []  # Velocity as a series of swaps
        self.best_position = list(self.position)  # Personal best
        self.best_cost = float('inf')  # Best cost for this particle
        self.current_cost = float('inf')  # Current cost

# PSO Algorithm for TSP
def pso_tsp(cities, num_particles=30, max_iterations=100, w=0.5,
    c1=1.5, c2=1.5):
    """
    Solve the Traveling Salesman Problem using Particle Swarm
        Optimization.

    Parameters:
        cities (list): List of tuples representing city coordinates
            [(x1, y1), (x2, y2), ...].
        num_particles (int): Number of particles in the swarm.
        max_iterations (int): Maximum number of iterations.
        w (float): Inertia weight.
        c1 (float): Cognitive weight.
        c2 (float): Social weight.

    Returns:
        tuple: (best cost, best tour)
    """
    num_cities = len(cities)

    # Initialize particles
    particles = [Particle(num_cities) for _ in range(num_particles)]
    global_best_position = None
    global_best_cost = float('inf')

    # Main PSO loop
    for iteration in range(max_iterations):
        for particle in particles:
            # Calculate the cost for the current position
            particle.current_cost = calculate_tour_distance(cities,
                particle.position)

            # Update personal best
            if particle.current_cost < particle.best_cost:
                particle.best_cost = particle.current_cost
                particle.best_position = list(particle.position)

            # Update global best
```

```python
            if particle.current_cost < global_best_cost:
                global_best_cost = particle.current_cost
                global_best_position = list(particle.position)

        # Update particle velocities and positions
        for particle in particles:
            new_velocity = []

            # Apply random swaps for velocity update
            for _ in range(random.randint(1, num_cities // 2)):
                if random.random() < w:
                    idx1, idx2 = random.sample(range(num_cities), 2)
                    new_velocity.append((idx1, idx2))

            # Apply cognitive and social components
            for i in range(num_cities):
                if random.random() < c1:
                    if particle.position[i] != particle.best_position[i]:
                        idx1, idx2 = particle.position.index(particle.best_position[i]), i
                        new_velocity.append((idx1, idx2))
                if random.random() < c2:
                    if particle.position[i] != global_best_position[i]:
                        idx1, idx2 = particle.position.index(global_best_position[i]), i
                        new_velocity.append((idx1, idx2))

            # Apply the velocity (swaps) to the particle's position
            for idx1, idx2 in new_velocity:
                particle.position[idx1], particle.position[idx2] = (
                    particle.position[idx2],
                    particle.position[idx1],
                )

            # Save the updated velocity
            particle.velocity = new_velocity

    return global_best_cost, [cities[i] for i in global_best_position]

# Example Usage
if __name__ == "__main__":
    # Example cities as (x, y) coordinates
    cities = [
        (0, 0), (2, 3), (5, 2), (7, 7), (3, 8), (6, 1), (4, 5)
    ]

    # Solve TSP using PSO
    cost, tour = pso_tsp(cities, num_particles=50, max_iterations=200)
    print("Optimal Tour:", tour)
    print("Minimum Cost:", cost)
```


https://doi.org/10.62071/tmjm.v6i2.716

4.5 SA Algorithm Python Code

Simulated Annealing Algorithm for TSP

```
import random
import math

# Function to calculate the Euclidean distance between two cities
def calculate_distance(city1, city2):
    return math.sqrt((city1[0] - city2[0])**2 + (city1[1] -
        city2[1])**2)

# Function to calculate the total distance of a tour
def calculate_tour_distance(cities, tour):
    distance = 0
    for i in range(len(tour) - 1):
        distance += calculate_distance(cities[tour[i]], cities[tour[i
            + 1]])
    distance += calculate_distance(cities[tour[-1]], cities[tour[0]])
        # Return to the start
    return distance

# Function to perform Simulated Annealing
def simulated_annealing_tsp(cities, initial_temperature=1000,
    cooling_rate=0.995, stop_temperature=1e-8, max_iterations=1000):
    """
    Solve the Traveling Salesman Problem using Simulated Annealing.

    Parameters:
        cities (list): List of tuples representing city coordinates
            [(x1, y1), (x2, y2), ...].
        initial_temperature (float): Starting temperature.
        cooling_rate (float): Rate at which the temperature decreases.
        stop_temperature (float): Minimum temperature to stop the
            algorithm.
        max_iterations (int): Maximum iterations at each temperature
            level.

    Returns:
        tuple: (best distance, best tour)
    """
    num_cities = len(cities)

    # Generate an initial random tour
    current_tour = list(range(num_cities))
    random.shuffle(current_tour)
    current_distance = calculate_tour_distance(cities, current_tour)

    # Initialize best solution
    best_tour = list(current_tour)
    best_distance = current_distance

    # Initialize temperature
    temperature = initial_temperature

    while temperature > stop_temperature:
```

```
        for _ in range(max_iterations):
            # Create a new tour by swapping two cities
            new_tour = list(current_tour)
            i, j = random.sample(range(num_cities), 2)
            new_tour[i], new_tour[j] = new_tour[j], new_tour[i]

            # Calculate the distance of the new tour
            new_distance = calculate_tour_distance(cities, new_tour)

            # Acceptance criteria
            if new_distance < current_distance or random.random() <
                math.exp((current_distance - new_distance) /
                temperature):
                current_tour = new_tour
                current_distance = new_distance

                # Update the best solution found so far
                if current_distance < best_distance:
                    best_tour = list(current_tour)
                    best_distance = current_distance

        # Reduce the temperature according to the cooling schedule
        temperature *= cooling_rate

    return best_distance, [cities[i] for i in best_tour]


# Example Usage
if __name__ == "__main__":
    # Example cities as (x, y) coordinates
    cities = [
        (0, 0), (2, 3), (5, 2), (7, 7), (3, 8), (6, 1), (4, 5)
    ]

    # Solve TSP using Simulated Annealing
    best_distance, best_tour = simulated_annealing_tsp(cities)
    print("Best Tour:", best_tour)
    print("Best Distance:", best_distance)
```

4.6 Genetic Algorithm Python Code

Genetic Algorithm for TSP

```
import random
import math

# Function to calculate the Euclidean distance between two cities
def calculate_distance(city1, city2):
    return math.sqrt((city1[0] - city2[0])**2 + (city1[1] -
        city2[1])**2)

# Function to calculate the total distance of a tour
def calculate_tour_distance(cities, tour):
    distance = 0
    for i in range(len(tour) - 1):
```


https://doi.org/10.62071/tmjm.v6i2.716

```python
        distance += calculate_distance(cities[tour[i]], cities[tour[i + 1]])
    distance += calculate_distance(cities[tour[-1]], cities[tour[0]])  # Return to start
    return distance

# Function to create an initial population
def create_population(cities, population_size):
    num_cities = len(cities)
    return [random.sample(range(num_cities), num_cities) for _ in range(population_size)]

# Function to select parents for crossover
def select_parents(population, fitness_scores):
    total_fitness = sum(fitness_scores)
    probabilities = [fitness / total_fitness for fitness in fitness_scores]
    parent1 = random.choices(population, weights=probabilities, k=1)[0]
    parent2 = random.choices(population, weights=probabilities, k=1)[0]
    return parent1, parent2

# Function to perform crossover
def crossover(parent1, parent2):
    size = len(parent1)
    start, end = sorted(random.sample(range(size), 2))
    child = [-1] * size
    child[start:end] = parent1[start:end]
    pointer = 0

    for gene in parent2:
        if gene not in child:
            while child[pointer] != -1:
                pointer += 1
            child[pointer] = gene

    return child

# Function to perform mutation
def mutate(tour, mutation_rate=0.1):
    for i in range(len(tour)):
        if random.random() < mutation_rate:
            j = random.randint(0, len(tour) - 1)
            tour[i], tour[j] = tour[j], tour[i]

# Genetic Algorithm for TSP
def genetic_algorithm_tsp(cities, population_size=100, generations=500, mutation_rate=0.1):
    """
    Solve the Traveling Salesman Problem using a Genetic Algorithm.

    Parameters:
        cities (list): List of tuples representing city coordinates [(x1, y1), (x2, y2), ...].
        population_size (int): Size of the population.
        generations (int): Number of generations.
        mutation_rate (float): Probability of mutation.
```

```python

    Returns:
        tuple: (best distance, best tour)
    """
    # Create the initial population
    population = create_population(cities, population_size)

    # Iterate over generations
    for generation in range(generations):
        # Calculate fitness scores (inverse of distance)
        fitness_scores = [1 / calculate_tour_distance(cities, tour) for tour in population]

        # Create the next generation
        next_generation = []
        for _ in range(population_size // 2):
            # Select parents and produce offspring
            parent1, parent2 = select_parents(population, fitness_scores)
            child1 = crossover(parent1, parent2)
            child2 = crossover(parent2, parent1)

            # Mutate the offspring
            mutate(child1, mutation_rate)
            mutate(child2, mutation_rate)

            # Add offspring to the next generation
            next_generation.extend([child1, child2])

        # Replace the old population with the new generation
        population = next_generation

    # Find the best solution in the final population
    best_tour = min(population, key=lambda tour: calculate_tour_distance(cities, tour))
    best_distance = calculate_tour_distance(cities, best_tour)

    return best_distance, [cities[i] for i in best_tour]


# Example Usage
if __name__ == "__main__":
    # Example cities as (x, y) coordinates
    cities = [
        (0, 0), (2, 3), (5, 2), (7, 7), (3, 8), (6, 1), (4, 5)
    ]

    # Solve TSP using Genetic Algorithm
    best_distance, best_tour = genetic_algorithm_tsp(cities)
    print("Best Tour:", best_tour)
    print("Best Distance:", best_distance)
```

4.7 Greedy Algorithm Python Code

https://doi.org/10.62071/tmjm.v6i2.716

Greedy Algorithm for TSP

```
import math

# Function to calculate the Euclidean distance between two cities
def calculate_distance(city1, city2):
    return math.sqrt((city1[0] - city2[0])**2 + (city1[1] - city2[1])**2)

# Function to calculate the total distance of a tour
def calculate_tour_distance(cities, tour):
    distance = 0
    for i in range(len(tour) - 1):
        distance += calculate_distance(cities[tour[i]], cities[tour[i + 1]])
    distance += calculate_distance(cities[tour[-1]], cities[tour[0]])  # Return to start
    return distance

# Greedy Algorithm for TSP
def greedy_tsp(cities):
    """
    Solve the Traveling Salesman Problem using the Greedy Algorithm.

    Parameters:
        cities (list): List of tuples representing city coordinates [(x1, y1), (x2, y2), ...]

    Returns:
        tuple: (total distance, tour as a list of city indices)
    """
    num_cities = len(cities)
    visited = [False] * num_cities  # Track visited cities
    tour = [0]  # Start from the first city
    visited[0] = True

    # Visit all cities
    for _ in range(num_cities - 1):
        last_visited = tour[-1]
        nearest_city = None
        nearest_distance = float('inf')

        # Find the nearest unvisited city
        for i in range(num_cities):
            if not visited[i]:
                distance = calculate_distance(cities[last_visited], cities[i])
                if distance < nearest_distance:
                    nearest_distance = distance
                    nearest_city = i

        # Mark the nearest city as visited and add it to the tour
        visited[nearest_city] = True
        tour.append(nearest_city)

    # Complete the tour by returning to the starting city
```

```
    total_distance = calculate_tour_distance(cities, tour)
    return total_distance, tour


# Example Usage
if __name__ == "__main__":
    # Example cities as (x, y) coordinates
    cities = [
        (0, 0), (2, 3), (5, 2), (7, 7), (3, 8), (6, 1), (4, 5)
    ]

    # Solve TSP using Greedy Algorithm
    total_distance, tour = greedy_tsp(cities)
    tour_coordinates = [cities[i] for i in tour]

    print("Tour (indices):", tour)
    print("Tour (coordinates):", tour_coordinates)
    print("Total Distance:", total_distance)
```

4.8 Divide and Conquer Algorithm for TSP

Divide and Conquer Algorithm for TSP

```
import math
import random

# Function to calculate the Euclidean distance between two cities
def calculate_distance(city1, city2):
    return math.sqrt((city1[0] - city2[0])**2 + (city1[1] -
       city2[1])**2)

# Function to divide cities into two subsets
def divide_cities(cities):
    """
    Divide cities into two subsets based on their x or y coordinates
       (randomly chosen).
    """
    if random.choice([True, False]):  # Split by x-coordinate
        cities.sort(key=lambda city: city[0])
    else:  # Split by y-coordinate
        cities.sort(key=lambda city: city[1])
    mid = len(cities) // 2
    return cities[:mid], cities[mid:]

# Function to find the shortest bridge between two subsets
def find_shortest_bridge(subset1, subset2):
    """
    Find the shortest edge connecting two subsets.
    """
    min_distance = float('inf')
    bridge = None

    for city1 in subset1:
        for city2 in subset2:
            distance = calculate_distance(city1, city2)
```

```
            if distance < min_distance:
                min_distance = distance
                bridge = (city1, city2)

    return bridge

# Recursive Divide and Conquer TSP Solver
def divide_and_conquer_tsp(cities):
    """
    Solve TSP using Divide and Conquer.

    Parameters:
        cities (list): List of tuples representing city coordinates [(x1, y1), (x2, y2), ...]

    Returns:
        tuple: (tour as a list of city coordinates, total distance)
    """
    # Base Case: If there is only one city, return it as the solution
    if len(cities) == 1:
        return cities, 0

    # Base Case: If there are two cities, connect them and return
    if len(cities) == 2:
        distance = calculate_distance(cities[0], cities[1])
        return cities + [cities[0]], 2 * distance

    # Recursive Case
    # Step 1: Divide cities into two subsets
    subset1, subset2 = divide_cities(cities)

    # Step 2: Recursively solve for each subset
    tour1, distance1 = divide_and_conquer_tsp(subset1)
    tour2, distance2 = divide_and_conquer_tsp(subset2)

    # Step 3: Find the shortest bridge between the two subsets
    bridge = find_shortest_bridge(subset1, subset2)

    # Step 4: Merge the solutions
    # Insert the bridge to connect the two subsets
    tour1 = tour1[:-1]  # Remove the last city to avoid duplicate in merging
    merged_tour = tour1 + [bridge[0], bridge[1]] + tour2

    # Calculate the total distance
    merged_distance = distance1 + distance2 + calculate_distance(bridge[0], bridge[1])

    return merged_tour, merged_distance


# Example Usage
if __name__ == "__main__":
    # Example cities as (x, y) coordinates
    cities = [
        (0, 0), (2, 3), (5, 2), (7, 7), (3, 8), (6, 1), (4, 5)
```

```
    ]

    # Solve TSP using Divide and Conquer
    tour, total_distance = divide_and_conquer_tsp(cities)
    print("Tour (coordinates):", tour)
    print("Total Distance:", total_distance)
```


https://doi.org/10.62071/tmjm.v6i2.716