Half-TSP Problem

Contributors			
Student Number (ID)	Name	Surname	Job Distribution
150120076	Furkan	Gökgöz	coding/debugging
150120998	Abdelrahman	Zahran	coding/ debugging/ Report
150121043	Bora	Duman	coding/ Report
150121051	Arda	Öztürk	coding/ debugging

Sections Of the Report: -

Section (1): Problem Definition: -

The (Half-tsp) problem, also known as the **Half Traveling Salesman Problem**, is a variant of the classic Traveling Salesman Problem (TSP). The TSP is a well-known optimization problem in computer science that asks for the shortest possible route a salesman can take to visit a given set of cities, starting from one city, visiting each city exactly once, and returning to the starting city.

In the Half-tsp problem, the objective is to find a route that visits approximately half of the cities while still obeying the constraints of the TSP. The goal is to minimize the total distance traveled for this subset of cities.

Formally, given a set of cities and the distances between each pair of cities, the Half-tsp problem aims to find a subset of cities that includes roughly half of them, such that a closed tour can be formed by visiting these cities in a specific order. The challenge is to identify the subset of cities and the ordering that minimizes the total distance traveled.

The Half-tsp problem is computationally challenging because it is NP-hard, which means that there is no known efficient algorithm to solve it optimally in all cases. It requires exploring a large number of possible subsets and orderings, making it a complex optimization problem.

The Half-tsp problem has applications in various domains, such as transportation planning, logistics, and circuit design. It can be used to optimize routing problems where visiting all cities is not feasible or necessary, and finding a near-optimal solution for a subset of cities is sufficient.

There are several possible approaches to solving the Half-tsp problem:

- Greedy Algorithms: Greedy algorithms make locally optimal choices at each step to construct a solution. In the context of the Half-tsp problem, a greedy algorithm could start with an initial city and iteratively add the nearest unvisited city to the tour until approximately half of the cities have been visited. This approach is simple and efficient, but it may not guarantee the optimal solution.
- Genetic Algorithms: Genetic algorithms are inspired by the process of natural selection and evolution. They maintain a population of potential solutions and apply genetic operations such as mutation and crossover to generate new solutions. In the Half-tsp problem, genetic algorithms can be used to evolve a population of tours, improving them over successive generations. This approach can explore a wide search space and potentially find good-quality solutions.
- Ant Colony Optimization (ACO): ACO is inspired by the behavior of ants searching for food. In the Half-tsp problem, an ACO algorithm can simulate a population of artificial ants that deposit pheromone trails on edges as they construct tours. The pheromone trails guide the ants' choices, with higher pheromone concentrations indicating more attractive paths. Over time, the pheromone trails are reinforced or evaporated, leading to the discovery of good-quality routes.
- Local Search Algorithms: Local search algorithms iteratively improve an initial solution by exploring its neighborhood and moving to a better solution if one is found. In the context of the Half-tsp problem, a local search algorithm could start with a random or heuristic solution and iteratively swap cities in the tour to improve its quality. This approach can be effective in finding locally optimal solutions, but it may get trapped in suboptimal solutions.
- Integer Linear Programming (ILP): ILP formulations can be used to model the Half-tsp problem as an optimization problem with linear constraints. The objective is to minimize the total distance traveled subject to the constraints of visiting approximately half of the cities and forming a closed tour. Solving the ILP formulation requires specialized solvers that can handle integer variables, and it may be computationally expensive for large problem instances.

These approaches provide different strategies for solving the Half-tsp problem, each with its own advantages and trade-offs in terms of solution quality and computational efficiency. The choice of approach depends on the specific problem instance and the requirements of the application at hand.

Section (2): Algorithm Overview + Explanation.: -

Our approach (Modified NN + Modified 2-opt using local search) depends on using two main algorithms and several optimizations to solve the Traveling Salesman Problem (TSP): the Nearest Neighbor Algorithm (NN) for an initial solution, and the 2-opt heuristic with local search optimizing technique (shifting & swapping in an incremental manner) for optimizing the initial solution to generate a more optimal results in less time.

The Nearest Neighbor algorithm is a simple heuristic approach to solving the Half-tsp problem. It starts from an initial city and repeatedly selects the nearest unvisited city as the next city to visit until all cities have been visited. The algorithm constructs a path by connecting the cities in the order they are visited, resulting in a solution that may not be optimal but is usually quite good.

Here is an overview of the Nearest Neighbor algorithm for the Half-tsp problem:

Initialization: Select a starting city.

Main Loop: While there are unvisited cities: First, find the nearest unvisited city to the current city. Then add the nearest city to the path. After that, mark the nearest city as visited. Finally, Set the nearest city as the current city.

Termination: Return to the starting city, completing the path.
The Nearest Neighbor algorithm does not guarantee finding the optimal solution, but it is a quick and straightforward approach that often provides reasonably good solutions.
Now, let's discuss the mathematical formulas involved in the algorithm computation for the Half-tsp problem:
Given a set of cities C = {c1, c2, ..., cn}, where ci represents a city and n is the total number of cities, the mathematical formulas are as follows:

- Euclidean Distance: The Euclidean distance between two cities ci and cj can be computed using their coordinates (xi, yi) and (xj, yj) as:

D (ci, cj) = sqrt ((xi - xj)^2 + (yi - yj)^2)

- Nearest Neighbor Selection: To select the nearest unvisited city to the current city, you need to calculate the distance between the current city and each unvisited city and select the city with the minimum distance. The formula is:

NearestCity = argmin (D (currentCity, unvisitedCity))

- Total Distance Calculation: To calculate the total distance of the path, you need to sum up the distances between consecutive cities in the path, including the distance from the last city back to the starting city. The formula is: Total distance of the path: TD(P) = ∑ [i=1 to n-1] D (pi, pi+1) + D (pn, p1)

By applying these mathematical formulas in the Nearest Neighbor algorithm, you can find a suboptimal but reasonably good solution for the Half-tsp problem.
Local search is an optimization technique that aims to find an optimal solution within a given search space. It starts with an initial solution and explores the neighboring solutions by making incremental changes. The basic idea is to iteratively improve the solution by moving from one feasible solution to a better one until an optimal solution is reached or a stopping criterion is satisfied.
In the case of the Half-tsp problem, which involves finding the shortest path that visits a subset of cities and returns to the starting city, a local search algorithm can be applied to optimize the solution. Here is an overview of the algorithm's computation:
Initialization: Select an initial solution, which could be a random or a heuristically generated solution.
Main Loop: Evaluate the current solution by calculating the total distance of the path, then generate neighboring solutions by making incremental changes to the current solution. This can include swapping two cities, reversing a portion of the path, or applying other local modifications. After that, Select the best neighboring solution based on an evaluation function, which measures the quality of a solution. In the Half-tsp problem, the evaluation function would be the total distance of the path, If the best neighboring solution is an improvement over the current solution, update the current solution to the best neighboring solution. Finally, Repeat the above steps until a stopping criterion is met. This could be a maximum number of iterations, a time limit, or reaching a predefined optimal solution.
Termination: Return the best solution found during the search process.
It's important to note that local search algorithms, including those for the Half-tsp problem, do not guarantee finding the global optimum. They are efficient for finding good solutions within large search spaces but can get trapped in local optima.
The mathematical formulas for algorithm computation in the Half-tsp problem involve defining the evaluation function and the neighbor generation process. The evaluation function can be represented as follows:
Evaluation Function: Let D(i, j) represent the distance between city i and city j. Let P = (p1, p2, ..., pn) represent the path, where p1 is the starting city and pn is the ending city.
Total distance of the path: TD(P) = ∑ [i=1 to n-1] D(pi, pi+1) + D(pn, p1)
To generate neighboring solutions, different techniques can be used such as:

- Swap two cities: Given a path P = (p1, p2, ..., pn), select two random indices i and j such that $1 \leq i < j \leq n$. Generate a new path P' by swapping cities pi and pj in the path.
- Reverse a portion of the path: Given a path P = (p1, p2, ..., pn), select two random indices i and j such that $1 \leq i < j \leq n$. Generate a new path P' by reversing the order of cities between indices i and j in the path.

These formulas and techniques can be incorporated into a local search algorithm for the Half-tsp problem, allowing it to iteratively explore and improve solutions until an optimal or satisfactory solution is found.

2-opt Algorithm

The 2-opt algorithm is a local search algorithm commonly used to improve the quality of a solution in the Traveling Salesman Problem (TSP). In the standard TSP, the goal is to find the shortest possible route that visits each city exactly once and returns to the starting city.
However, in the half-TSP variant, the problem is slightly different. In this version, we only need to find a path that visits half of the cities and returns to the starting city. The objective is still to minimize the total distance traveled.
The 2-opt algorithm works by iteratively swapping pairs of edges in the current solution to see if the new arrangement produces a shorter path. The basic steps of the 2-opt algorithm for the half-TSP are as follows:

- Start with an initial solution that visits half of the cities and returns to the starting city. This solution can be generated randomly or using a different heuristic algorithm.
- Choose two edges in the current solution and consider reversing the order of the cities between them. This creates a new potential solution.
- Calculate the total distance of the new solution by summing the distances of the edges.
- If the new solution has a shorter distance than the current solution, accept the swap and update the current solution.
- Repeat steps 2-4 for all possible pairs of edges in the current solution.
- Continue iterating through all possible pairs of edges until no further improvements can be made.

The 2-opt algorithm iteratively explores different combinations of edge swaps, gradually improving the solution by eliminating inefficient loops and creating shorter paths. It is important to note that the algorithm may not always find the optimal solution, as it can get stuck in local optima.
To enhance the performance of the 2-opt algorithm, you can also incorporate other techniques such as simulated annealing, genetic algorithms, or tabu search. These methods can help to escape local optima and explore a wider search space.

Further Optimization - Shifting:

Our code will, as explained in the following Section (3), implement a technique to potentially escape from local minima, by cyclically shifting the cities in the tour and reapplying the 2-opt algorithm. The idea is that by changing the order of the cities, the 2-opt algorithm might find different beneficial swaps that it couldn't find before, leading to a shorter tour.

In conclusion, the combination of these algorithms will allow our program to generate a good solution to the TSP quickly, showcasing how heuristic methods can be used to tackle NP-hard problems like the TSP.

Section (3): Implementation (NN + 2-opt + shifting): -

In this section:

1. Data Structures and Utilities: The code uses a "City" structure to store the details of each city, such as its id, x and y coordinates, and its visited status. Two arrays, "cities" and "selected", are used to store the cities to visit and the cities in the order they are visited, respectively. The "dist" function is used to calculate the Euclidean distance between two cities.

City Structure: The code starts by defining a struct type called City, which is used as a data type throughout the program to represent the cities. This structure contains four fields:

id: The identifier for the city.

x: The x-coordinate of the city.

y: The y-coordinate of the city.

visited: A Boolean flag to indicate if the city has been visited in the tour.

Two arrays of City are created, cities and selected, to store all cities and the cities selected for the tour, respectively.

Arrays: Arrays are a fundamental data structure used extensively in this program. Arrays cities and selected store the city data. cities is a static array of size MAX_CITIES, where MAX_CITIES is a defined constant. The selected array stores the cities selected for the current tour and is of size MAX_CITIES / 2 + 1. The advantage of arrays is their efficient O (1) time complexity for accessing elements, which is extensively utilized in our algorithm's calculations.

Files: The program uses file I/O operations to read the input and write the output. The input file contains city data (ID, x-coordinate, y-coordinate), which is read and stored in the cities array. The output file is used to write the tour's total distance and the city IDs in the order they are visited.

Euclidean Distance Calculation:

The program uses the 'dist' function to calculate the Euclidean distance between two cities. This function takes in two City structures, calculates the difference in their x and y coordinates (respectively dx and dy), squares these differences, adds them, and takes the square root of the result. The output is rounded to the nearest integer. This function is fundamental to our implementation as it quantifies the distance between two cities.

2. Initial Solution Generation (Nearest Neighbor Algorithm): The main function of the code prompts the user for an input file containing the coordinates of the cities. The code reads the city data from the file into the "cities" array. The code then creates an initial tour using the Nearest Neighbor algorithm, which starts from the first city and repeatedly visits the closest yet unvisited city.

Nearest Neighbor Algorithm:

The Nearest Neighbor Algorithm is a straightforward approach to solving the TSP. The algorithm starts from a given city, and then iteratively selects the nearest unvisited city as the next city to visit until all cities have been visited. This heuristic provides a simple and efficient way to generate an initial feasible solution, although the resulting tour is usually far from optimal.

The main advantage of the NN algorithm is its simplicity and low computational complexity, which makes it an excellent choice for generating a starting solution for further optimization. However, the algorithm doesn't consider the global structure of the problem, so the resulting tour can be far from optimal.

3. Tour Optimization (2-opt Algorithm): The initial tour is then optimized using a 2-opt heuristic implemented in the "optimize_tour" function. The heuristic iteratively examines all pairs of edges in the tour and checks if reversing the segment between them would result in a shorter tour. If it would, the segment is reversed. This process continues until no beneficial swaps can be made, resulting in a local minimum for the tour distance.

2-opt Algorithm:

The 2-opt Algorithm is a simple, yet effective local search method used to improve an existing tour. Given a tour, the algorithm repeatedly selects a pair of edges and checks whether swapping the two edges (reversing the direction of the segment between them) will reduce the total distance of the tour. If it does, the swap is made. This process continues until there are no beneficial swaps left to make, at which point the algorithm has found a local minimum.

The 2-opt heuristic doesn't guarantee finding the shortest possible tour, but it's capable of significantly improving the initial solution from the NN algorithm. Its main advantage is that it's a simple and relatively fast method for improving a given tour. However, it only searches the immediate neighborhood of the current solution (hence being a local search method) and may thus get stuck in local minima. In optimization, 2-opt is a simple local search algorithm for solving the traveling salesman problem. The 2-opt algorithm was first proposed by Croes in 1958, although the basic move had already been suggested by Flood. The main idea behind it is to take a route that crosses over itself and reorder it so that it does not. A complete 2-opt local search will compare every possible valid combination of the swapping mechanism. This technique can be applied to the traveling salesman problem as well as many related problems. These include the vehicle routing problem (VRP) as well as the capacitated VRP, which require minor modification of the algorithm.

4. Further Tour Optimization (Shifting): The code further optimizes the tour by shifting all cities in the tour by one position, and then running the 2-opt algorithm again. If the new tour is shorter, it replaces the old tour. This process is repeated until the shifting operation doesn't provide any more improvement.

5. Result Display and Output: Finally, the total distance of the tour is printed. Depending on which distance is shorter, the initial tour or the optimized tour, the corresponding tour is written to an output file. The program also calculates and displays the execution time of the program.

Section (4): Performance Analysis (NN vs. NN + 2-opt + vs. NN + 2-opt + shifting): -

The performance of our TSP implementation can be analyzed from three different aspects - time complexity, space complexity, and practical runtime performance.

1. Time Complexity:

The time complexity is primarily governed by the Nearest Neighbor and 2-opt algorithms, as well as the additional shifting optimization.

Nearest Neighbor: The Nearest Neighbor algorithm runs in O(n^2) time where n is the number of cities. This is because, for each city, it examines all remaining unvisited cities to find the nearest one. In the worst-case scenario, this results in an order of n^2 comparisons.

2-opt: The 2-opt algorithm iteratively scans through all pairs of edges to find beneficial swaps. This results in a time complexity of O(n^2) for each iteration. However, as the algorithm runs until no improvements can be found, the worst-case time complexity can be much higher.
Shifting: The shifting operation and subsequent reapplication of the 2-opt heuristic may improve the solution but also adds additional time overhead. The time complexity for each shift operation is O(n), and it might be repeated multiple times depending on the degree of improvement.

2. Space Complexity:

The space complexity of the program is O(n), where n is the number of cities. The cities and selected arrays each hold n elements. Furthermore, the distances array holds n elements for the optimization phase. Therefore, the overall space complexity is linear.

3. Practical Runtime Performance:

In practice, the performance of the code will vary depending on several factors including the number of cities, the distribution of cities, the quality of the initial tour, and the machine specifications. For instance, for a large number of cities or in cases where the cities are distributed such that the initial tour is far from optimal, the program may take longer to run.
The program measures its own execution time using the clock_gettime function from the time.h library. This gives an accurate measure of real-world runtime, accounting for the time taken for I/O operations, memory allocation, and computation. It provides valuable insights about the code's performance on any system or problem instance.
In conclusion, while the code may have high theoretical time complexity due to the nature of the TSP and the use of the NN and 2-opt heuristics, it is an effective and practical solution for solving the TSP to a satisfactory degree. The usage of additional techniques like the shifting optimization may result in a better solution, albeit at the cost of additional computational resources.

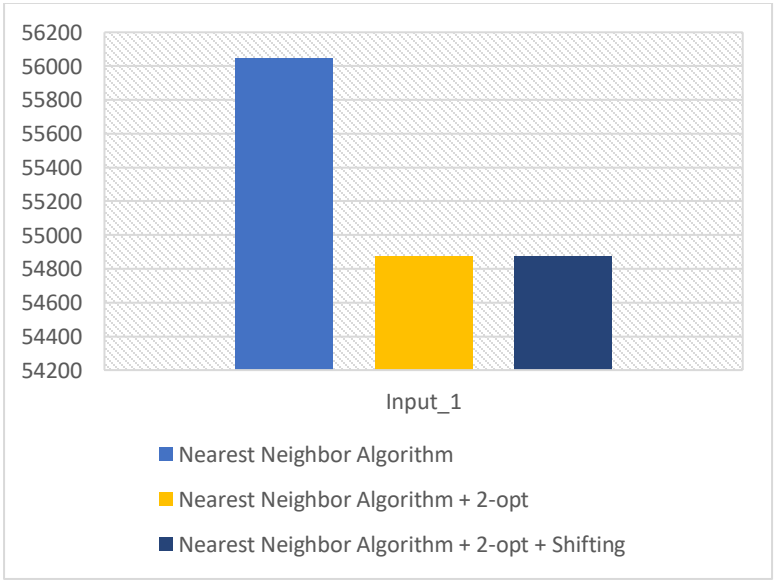
Input	Nearest Neighbor Algorithm	Nearest Neighbor Algorithm + 2-opt	Nearest Neighbor Algorithm + 2-opt + Shifting	Running Time (ms)
Input_1	56041	54872	54872	3.567600
Input_2	1618	1598	1594	20.317000
Input_3	824173	812041	811917	25022.909400
test-input-1	1618	1598	1594	9.483600
test-input-2	142727	138957	138843	94.349300
test-input-3	35223735	34860939	34860939	112041.281500
test-input-4	5867	5745	5744	631.996700

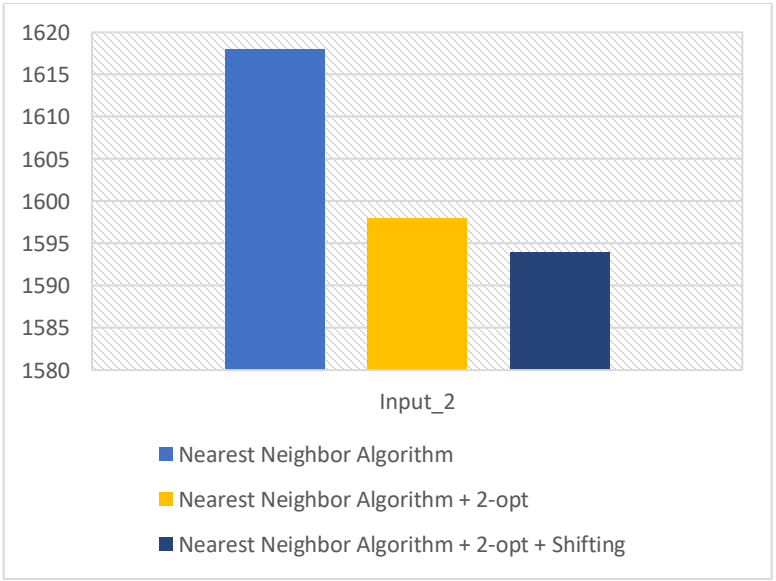
From graphs and the table of the tested inputs we realize that:

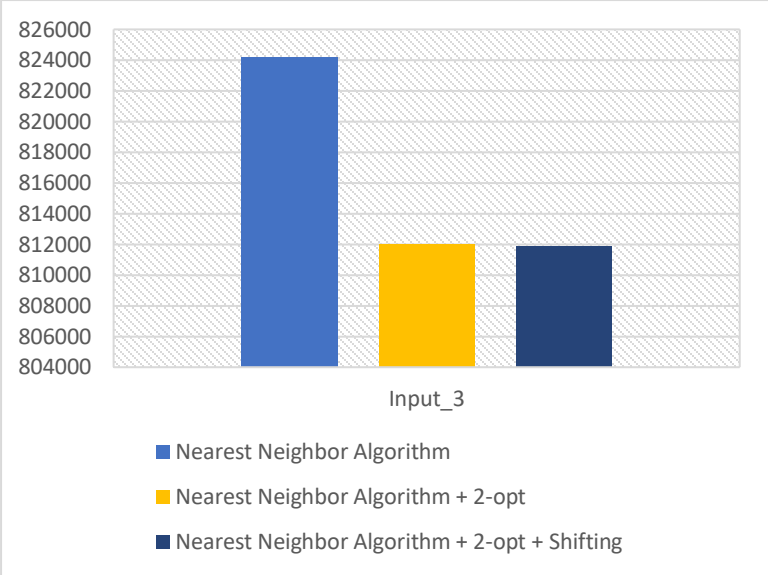
(NN + 2-opt + shifting optimization) is always better than NN algorithm and nearly better than (NN + 2-opt algorithm) with slight differences depending on input sizes (Number of cities, distances in between, coordinates, etc.).

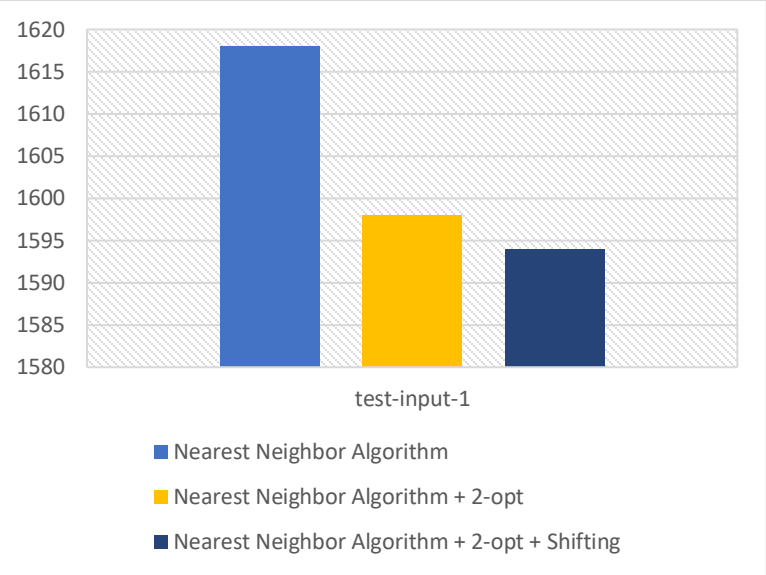
For most of the inputs of different sizes our approach (NN + 2-opt + Shifting) has roughly been estimated performing nearly 4 – 5 % in all cases for variant inputs sizes. While (NN + 2-opt) and (NN + 2-opt + Shifting) have nearly performed the same for most inputs, (NN + 2-opt + Shifting) have performed better for small inputs with small distances between the cities but for medium and big inputs they both performed with nearly same efficiency and optimality. The (NN + 2-opt + Shifting) has performed better than (NN + 2-opt) by 0.6 – 0.7 % in most cases and nearly in the same running times.

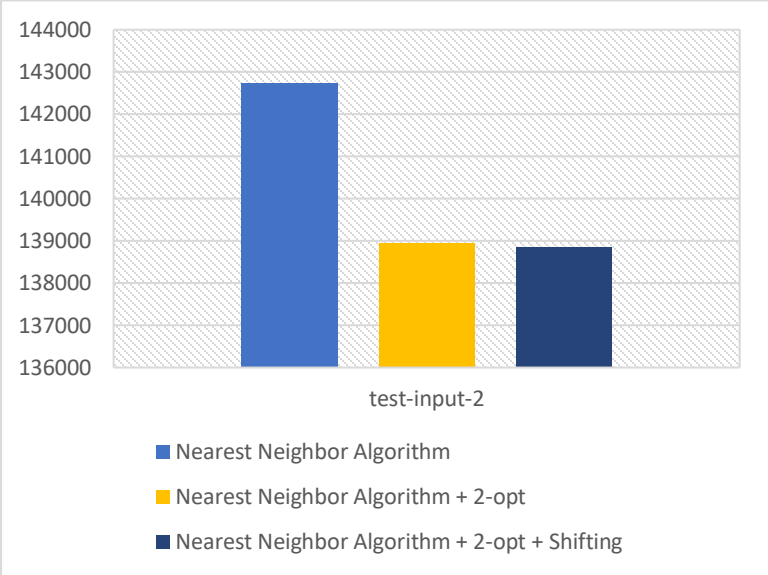
In summary, in case Half -TSP problem is used to solve for small inputs better to use (NN + 2-opt + Shifting), while for medium and big inputs there is no difference in using (NN + 2-opt) or (NN + 2-opt + Shifting). Both Algorithms will tend to provide similar solutions in reasonable time. Of course, (NN + 2-opt + Shifting) provides better solutions but difference still no big.

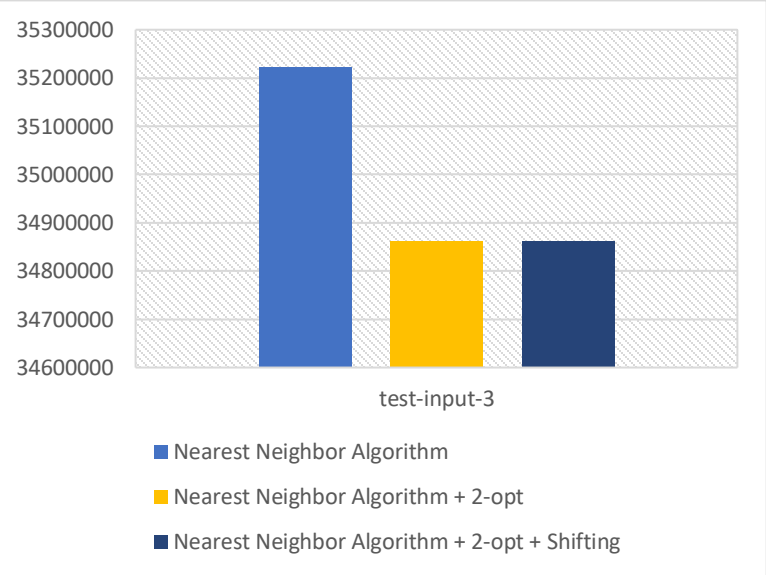


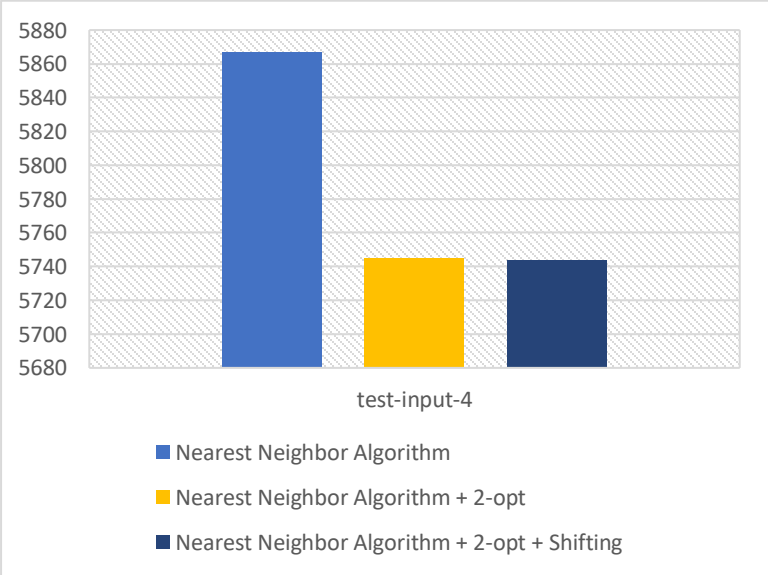












Section (5): References: -

Online Resources: -

- https://en.wikipedia.org/wiki/2-opt
- https://en.wikipedia.org/wiki/Local_search_(optimization)
- https://www.semanticscholar.org/paper/A-(slightly)-improved-approximation-algorithm-for-Karlin-Klein/4b31981bb53a068d9251f328f0bf3d7efbcc85d5
- https://arxiv.org/abs/1908.00227
- https://dl.acm.org/doi/pdf/10.1145/3406325.3451009