King Saud University
College of Computer and Information Sciences
Software Engineering Department

[SWE485] Selected Topics in Software Engineering Project

Applying Search Techniques in a real-world problem
(Travelling Salesman Problem)

Group #13	
Name	**ID**
Hind Aljabri	441203846
Raghad Alghannami	442200381
Nouf AlMuammar	442201430
Shahad AlJandal	441201179
Dina Alromih	441202151
Rand Alobaid	442200463

Github Repository

Supervised by: Dr. Khaoula Hamdi
Submission Date: 2024/5/12 -1445/11/4

Table of contents

Phase 1	Problem Definition and Formulation

Transportation problem, Also called the The Travelling Salesman problem, is a problem where you have to visit *n* cities, each city should be visited only once, where you can start with any city, but you should finish at the same city. The objective is to optimize the total distance.

For this phase, we are defining the problem mathematically as a Constraint Satisfaction Problem by defining the following:

1. Variables and Domains

Assume that we are given a complete graph $G(N, E)$ where N is the set of vertices (or cities) {1,2,...,n} and E is the set of edges (or roads) {1,2,..,e}. For each pair of vertices $i, j \in N$, $i \neq j$ the edge $(i,\ j) \in E$ is associated with a weight (or distance) $d_{ij} \in R +$

To model the problem, we introduce a binary variable, x_{ij} that indicates if edge (i, j) is part of the tour or not.

$$x_{ij} \in \{0, 1\}, \qquad \forall i, j \in N$$

Where 0 indicates no route from city i to city j, and 1 indicates the presence of a route from city i to city j.

2. Constraints

1. Starting and ending should be in the same city.

2. Every city must be visited exactly once

Mathematically:

$$\sum_{j=1}^{n} x_{ij} = 1 \ \forall i$$ (meaning that from each city i we can go to only one city j)

$$\sum_{i=1}^{n} x_{ij} = 1 \ \forall j$$ (meaning that for each city j we came from only one city i)

These constraints ensure that there is one and only one incoming and one and only one outgoing edge at each vertex, which ensures that we have visited each city exactly once.

Additionally, to ensure that we have an only one entirely connected tour where the start city is the same as the end city, we should eliminate subtours by imposing the following constraint:

$$\sum_{i\in S,\, j\in S,\, i\neq j}^{n} x_{ij} \ \leq \ |S| - 1 \ , \forall S \subsetneq N \, , \ |S| \geq 2$$

So there is no subset of vertices that could form a subtour which is not connected to the rest of the vertices.[1]

3. Objective Function

The objective function for the transportation problem is to minimize the total distance traveled while visiting all cities exactly once. The total distance traveled is determined by summing the distances of all edges that are part of the tour.

Let's denote d_{ij} as the distance between city i and city j. The objective function can be defined as follows:

Objective Function:

$$\text{Min} \sum_{i=1}^{n} \sum_{j=1}^{n} d_{ij} \cdot x_{ij}$$

In this objective function:

- d_{ij} represents the distance between city i and city j.
- x_{ij} is a binary variable indicating whether there is a route from city i to city j, $x_{ij}=1$ if there is a route, $x_{ij}=0$ otherwise.
- The double summation iterates over all pairs of cities, considering their distances and whether there's a route between them.
- The product $d_{ij} \cdot x_{ij}$ computes the distance between cities i and j only if there is a route between them.
- Summing up all such products gives the total distance traveled while considering only the routes that are part of the tour.

Therefore, this objective function guides the optimization process to find the set of routes that minimize the total distance traveled while satisfying the given constraints.

Phase 2	Incremental formulation

In this phase, We will solve the Traveling Salesman Problem using the incremental formulation using a search algorithm. Our algorithm starts with an empty state and assigns a value to one variable at a time.

1.The Heuristic(s)

TSP is known to be a difficult problem because the number of possible routes grows exponentially with the number of cities. In addition, it falls into the NP-complete category, which means that it does not have a known efficient algorithm to solve it, and it may require an infeasible processing time to be solved by a brute-force approach, where all possible solutions are systematically checked to find the optimal one. Therefore less expensive heuristics that sacrifice optimality for efficiency are commonly used in order to obtain satisfactory solutions in a reasonable time[2].

Nearest Neighbor Heuristic:

One of the most popular heuristics for solving the TSP is the Nearest Neighbor heuristic, which is a greedy algorithm[3]. It has a O(n^2) computational complexity and It starts with a random city then gradually constructs a tour by repeatedly selecting the closest unvisited city[4]. It is a greedy approach, so it makes a locally optimal choice at each stage in the hope that these choices will approximate a global optimal [5].

2. The Data Structure

- `tsp` **(2D List)**: Represents the distances between cities. Each element `tsp[i][j]` holds the distance from city `i` to city `j`.
- `visited` **List**: The `visited` list keeps track of whether each city has been visited during the algorithm execution. It's initialized with `False` for each city and updated to `True` once the city is visited.
- `route` **List**: A list to record the sequence of visited cities, It starts with the specified `firstCity` and is updated iteratively as the algorithm progresses.

3. The pseudocode of the Algorithm

1. Initialize variables:
    - Initialize a variable to keep track of the total cost of the journey.
    - Prepare a list to record the route taken, starting with the initial city.
    - Create a dictionary to mark cities as visited.

2. Start from an initial city:
    - Set this city as the first city and mark it as visited, and add it to the route that records a list of visited cities.

3. While there are unvisited cities:

    a. Iterate over each city from the current city:

    - Find the least expensive unvisited city that can be traveled to from the current city.
    - If a city with a cost lower than the current minimum is found, update the minimum cost and the route to include this city.

    b. Once all cities are checked:

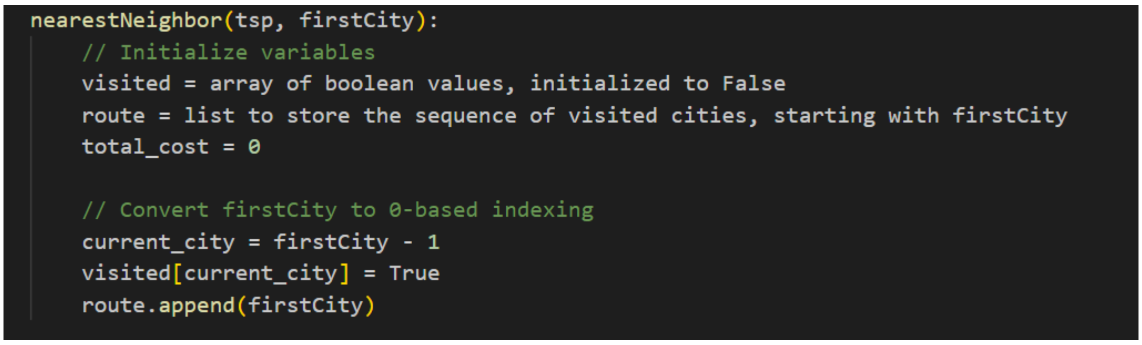
    - Add the minimum cost found to the total cost.
    - Mark the city as visited and set it as the current city for the next iteration.
    - Reset the search criteria to look for the next city.

4. After visiting all cities:
    - Find the cost of returning from the last visited city back to the initial city.
    - Add this cost to the total cost.

5. Output the results:
    - Print the total minimum cost to complete the journey.
    - Print the sequence of cities visited as the route taken.

The function:

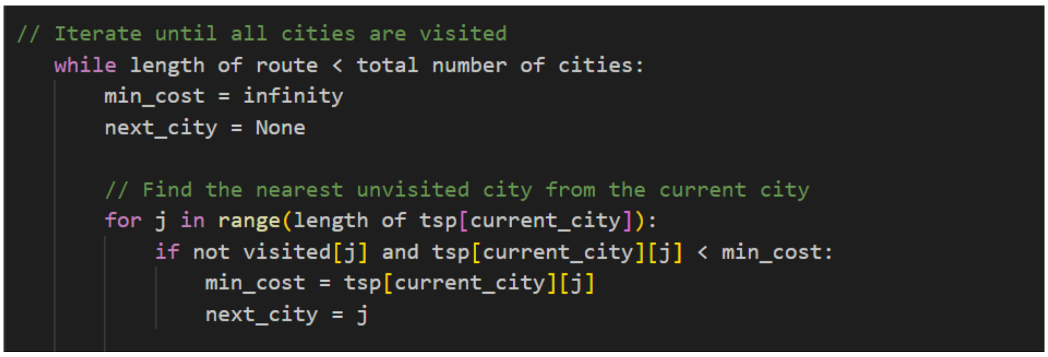
1. First start by setting up variables to track the total cost (`total_cost`), the visited cities (`visited`), which keeps track of which cities have been visited to prevent revisiting, and (`route`) to store the path taken, then mark the first city as visited.



```
nearestNeighbor(tsp, firstCity):
    // Initialize variables
    visited = array of boolean values, initialized to False
    route = list to store the sequence of visited cities, starting with firstCity
    total_cost = 0

    // Convert firstCity to 0-based indexing
    current_city = firstCity - 1
    visited[current_city] = True
    route.append(firstCity)
```

Figure[1]: a pseudocode segment showing the beginning of nearestNeighbor(tsp, firstCity) function

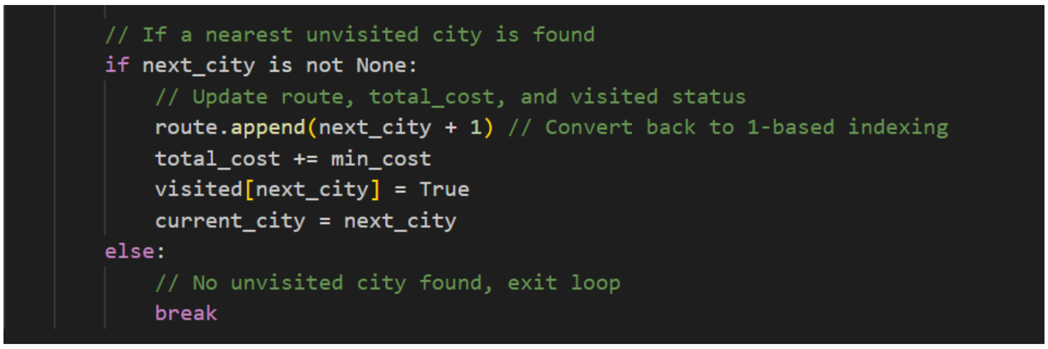
2. The main loop iterates through the `tsp` matrix, representing the costs between each pair of cities. For the current city, it looks for the least expensive, unvisited city `j` to travel next. This selection is based on the cost from the current city to the next city `j`, aiming to minimize this cost at each step.



```
// Iterate until all cities are visited
    while length of route < total number of cities:
        min_cost = infinity
        next_city = None

        // Find the nearest unvisited city from the current city
        for j in range(length of tsp[current_city]):
            if not visited[j] and tsp[current_city][j] < min_cost:
                min_cost = tsp[current_city][j]
                next_city = j
```

Figure[2]: a pseudocode segment showing the main loop that iterates tsp[][] searching for the minimum cost.

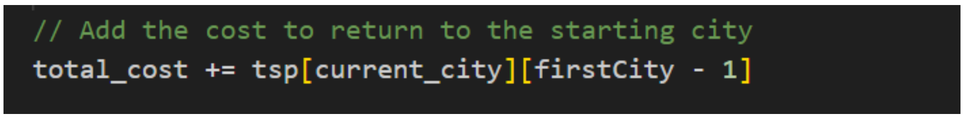
3. After finding the minimum cost `min_cost` for a part of the journey, this cost is added to `total_cost`. The `visited` and `route` are updated to reflect this choice, marking the chosen city as visited and preparing to move to the next city.



```
    // If a nearest unvisited city is found
    if next_city is not None:
        // Update route, total_cost, and visited status
        route.append(next_city + 1) // Convert back to 1-based indexing
        total_cost += min_cost
        visited[next_city] = True
        current_city = next_city
    else:
        // No unvisited city found, exit loop
        break
```

Figure[3]: a pseudocode segment showing the part of choosing the next city to visit.

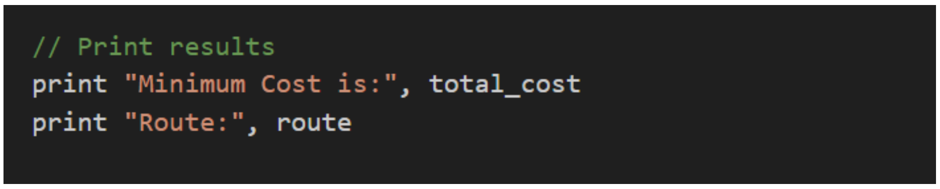
4. Once all cities have been visited, the algorithm finds the cost of returning to the first city from the last city visited. This final step's cost is added to `total_cost`.



```
// Add the cost to return to the starting city
total_cost += tsp[current_city][firstCity - 1]
```

Figure[4]: a pseudocode segment showing the part of calculating cost of returning to the firstCity

5. The function prints the total minimum cost to complete the tour and the sequence of cities visited as determined by the route taken.



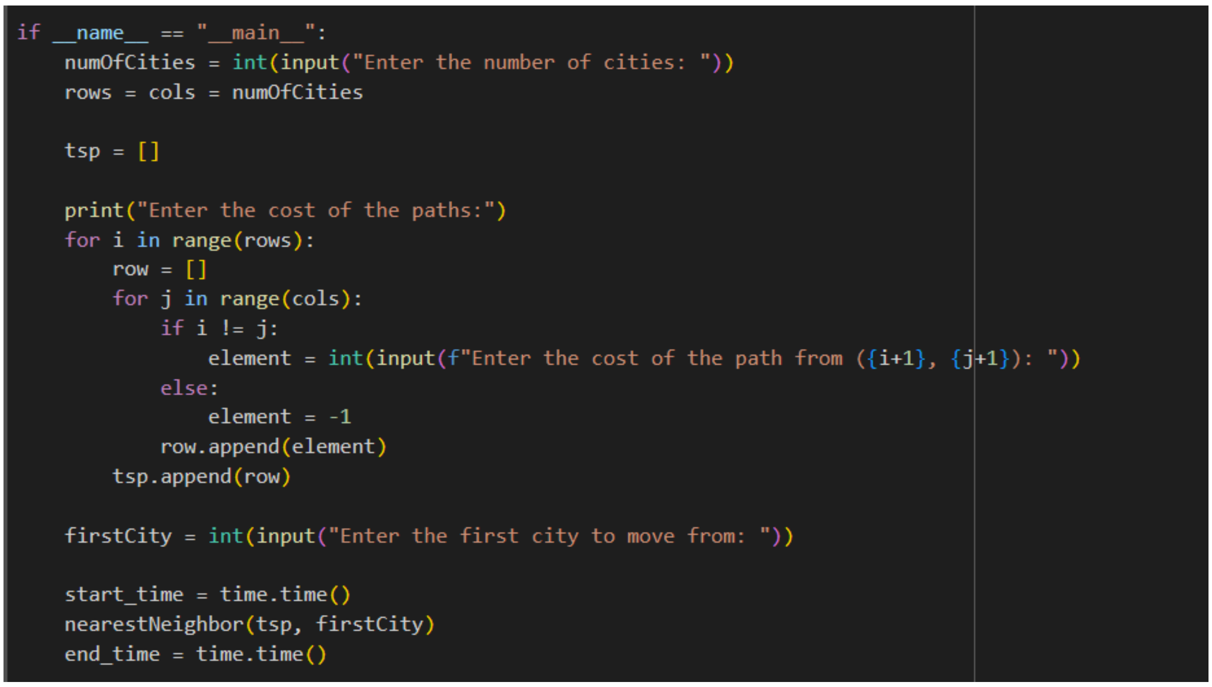
```
// Print results
print "Minimum Cost is:", total_cost
print "Route:", route
```

Figure[5]: a pseudocode segment showing the printing of the output

4. Screenshots of Relevant Parts of the Code with Comments

The algorithm was implemented using Python language[6]. Beneath are some code snippets along with comments explaining the code:

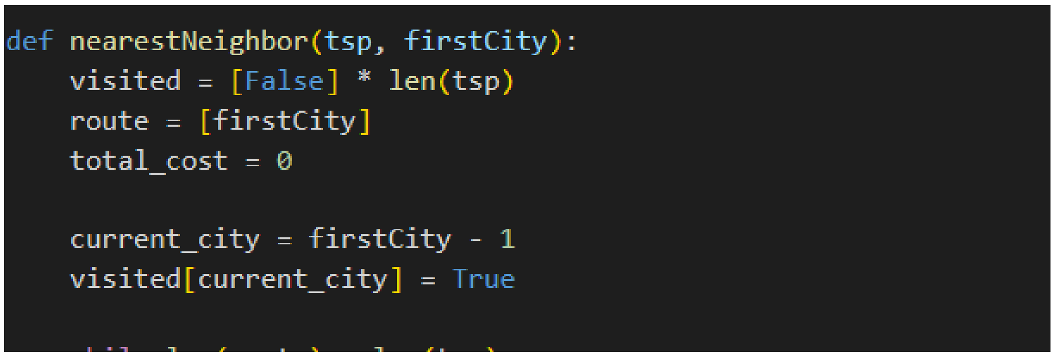
1. We started by prompting the user to enter the total number of cities, to construct a 2D list called `tsp` that stores the cost of each edge. We then asked for the first city to start traveling from, and lastly called the function `nearestNeighbor(tsp, firstCity)` to start applying our search algorithm:



```
if __name__ == "__main__":
    numOfCities = int(input("Enter the number of cities: "))
    rows = cols = numOfCities

    tsp = []

    print("Enter the cost of the paths:")
    for i in range(rows):
        row = []
        for j in range(cols):
            if i != j:
                element = int(input(f"Enter the cost of the path from ({i+1}, {j+1}): "))
            else:
                element = -1
            row.append(element)
        tsp.append(row)

    firstCity = int(input("Enter the first city to move from: "))

    start_time = time.time()
    nearestNeighbor(tsp, firstCity)
    end_time = time.time()
```

Figure[6]: a code segment showing the part that reads data from the user.

2. Inside `nearestNeighbor(tsp, firstCity)` , we first initialized the needed variables, then we created a `visited` list initialized with `False` and marked the first city as visited:



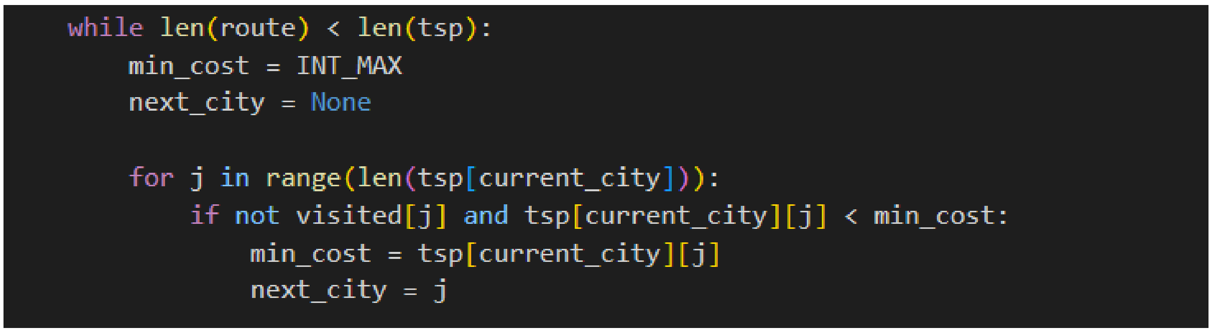
```python
def nearestNeighbor(tsp, firstCity):
    visited = [False] * len(tsp)
    route = [firstCity]
    total_cost = 0

    current_city = firstCity - 1
    visited[current_city] = True
```

Figure[7]: a code segment showing the beginning of nearestNeighbor(tsp, firstCity) function.

Variable	Used for
`total_cost`	keeps track of the minimum cost route
`current_city & j`	traverse the 2D array
`min_cost`	store the minimum cost, it was initialized to a very huge number `INT_MAX` to ensure that the first encountered cost value will be smaller than `min`, effectively updating `min` to that smaller value.
`visited`	List to track visited cities
`route`	List to store the route

Table[1]: a table showing the variables and their usage.

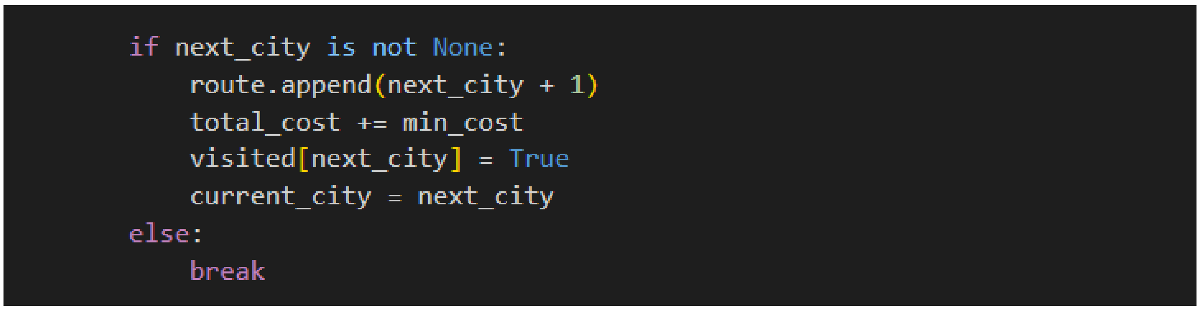
3. Next we performed traversal on the given adjacency matrix `tsp[][]` for all the cities, and if the cost of reaching any city from the current city is less than the current cost we have then updated the minimum cost:



```
while len(route) < len(tsp):
    min_cost = INT_MAX
    next_city = None

    for j in range(len(tsp[current_city])):
        if not visited[j] and tsp[current_city][j] < min_cost:
            min_cost = tsp[current_city][j]
            next_city = j
```

Figure[8]: a code segment showing the loop that traverses the tsp[][] array searching for the minimum cost.

5. When we have explored all the paths and nearest city is found, we added the minimal path cost to the `total_cost`, marked the city as visited, and prepared for visiting the next city in the route:



```
if next_city is not None:
    route.append(next_city + 1)
    total_cost += min_cost
    visited[next_city] = True
    current_city = next_city
else:
    break
```

Figure[9]: a code segment showing the part after checking all the paths and choosing the minimum one,then preparing to visit the next city.

6. The next snippet is for adding the cost of returning to the firstCity after visiting all the cities

```
# Add the cost to return to the starting city
total_cost += tsp[current_city][firstCity - 1]
```

Figure[10]: a code segment showing the part that adds the cost of returning to the first city.

7. Lastly we printed the minimum cost of the trip along with the `route`:

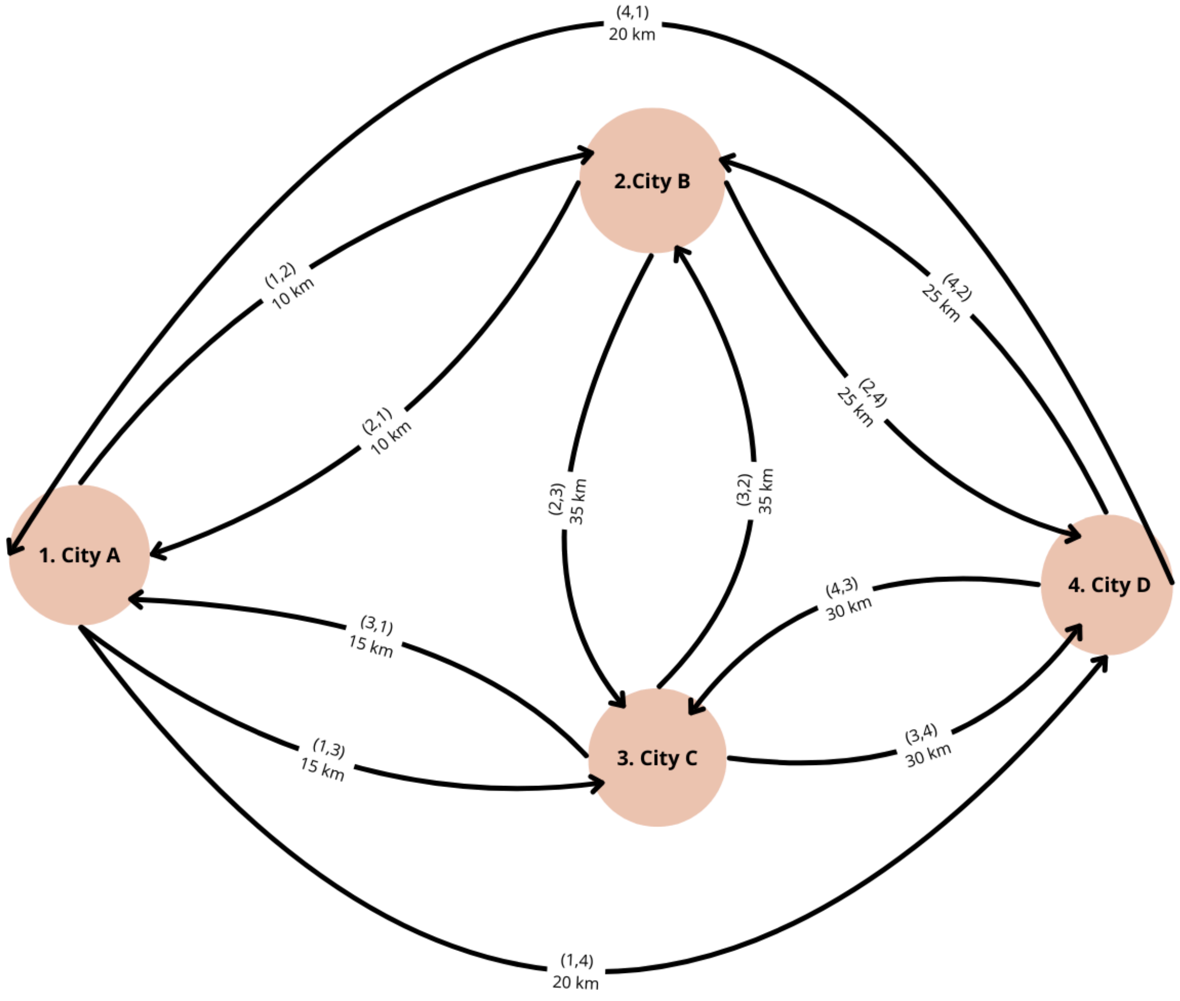
```
print("Minimum Cost is:", total_cost)
print("Route:", route)
```

Figure[11]: a code segment showing the print statement of total_cost and route.

- The complete code explained above could be found in the following link: SWE485-Project-Phase2.ipynb

5. Testing with Randomly generated/real world instances

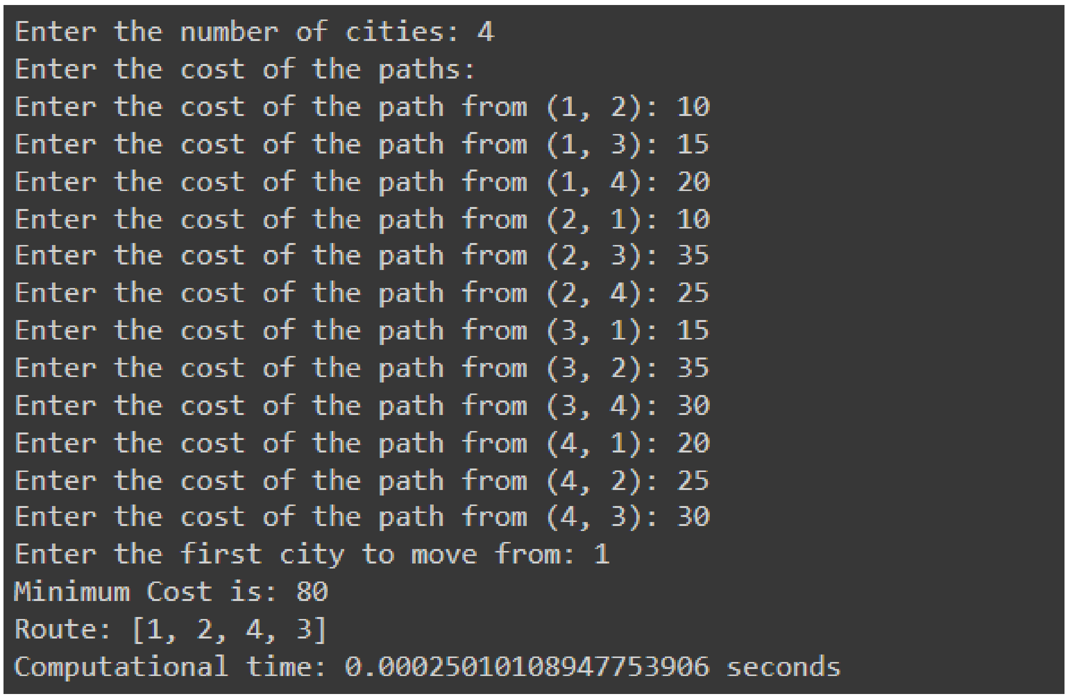
In the example, we are trying to find out the path/route with the minimum cost such that our aim is visiting city B, city C and city D once and return back to the source city A is achieved.



Figure[12]: a graph with nodes representing cities and edges representing paths.

6. Results and Computational Time of the Algorithm

The path through which we can achieve minimum cost, can be represented as **1**(City A) -> **2**(City B) -> **4**(City D) -> **3**(City C) -> **1**(City A). Here, we started from city 1(A) and ended in the same city after visiting all other cities once on our way. The cost of our path/route is calculated as follows:



```
Enter the number of cities: 4
Enter the cost of the paths:
Enter the cost of the path from (1, 2): 10
Enter the cost of the path from (1, 3): 15
Enter the cost of the path from (1, 4): 20
Enter the cost of the path from (2, 1): 10
Enter the cost of the path from (2, 3): 35
Enter the cost of the path from (2, 4): 25
Enter the cost of the path from (3, 1): 15
Enter the cost of the path from (3, 2): 35
Enter the cost of the path from (3, 4): 30
Enter the cost of the path from (4, 1): 20
Enter the cost of the path from (4, 2): 25
Enter the cost of the path from (4, 3): 30
Enter the first city to move from: 1
Minimum Cost is: 80
Route: [1, 2, 4, 3]
Computational time: 0.00025010108947753906 seconds
```

Figure[13]: a picture showing the code results after applying search on the given graph.

Phase 3	Complete formulation

In this phase, We will solve the Traveling Salesman Problem using a complete formulation with a local search algorithm using the 2-opt algorithm. The algorithm starts with a random solution that will be optimized through the process.

1. A description of the move/operator to transit from one state to another

In the 2-opt algorithm for solving the Traveling Salesman Problem (TSP) using local search, the move or operator involves swapping two edges in the tour to create a new tour.

Description of the 2-opt move:

1. Selecting Edges to Swap: Initially, two edges are selected from the current tour. These edges are not adjacent to each other in the tour.

2. Removing Selected Edges: Remove the selected edges along with the nodes they connect from the current tour. This creates two disconnected segments.

3. Reconnecting Segments: Reconnect the segments created by the removal of edges by adding the alternative edges. This can be done in two possible ways, depending on which endpoints are chosen to be connected.

4. Checking Feasibility: Ensure that the resulting tour is feasible, i.e., it visits all nodes exactly once and returns to the starting node.

5. Evaluating the New Tour: Calculate the total distance or cost of the new tour.

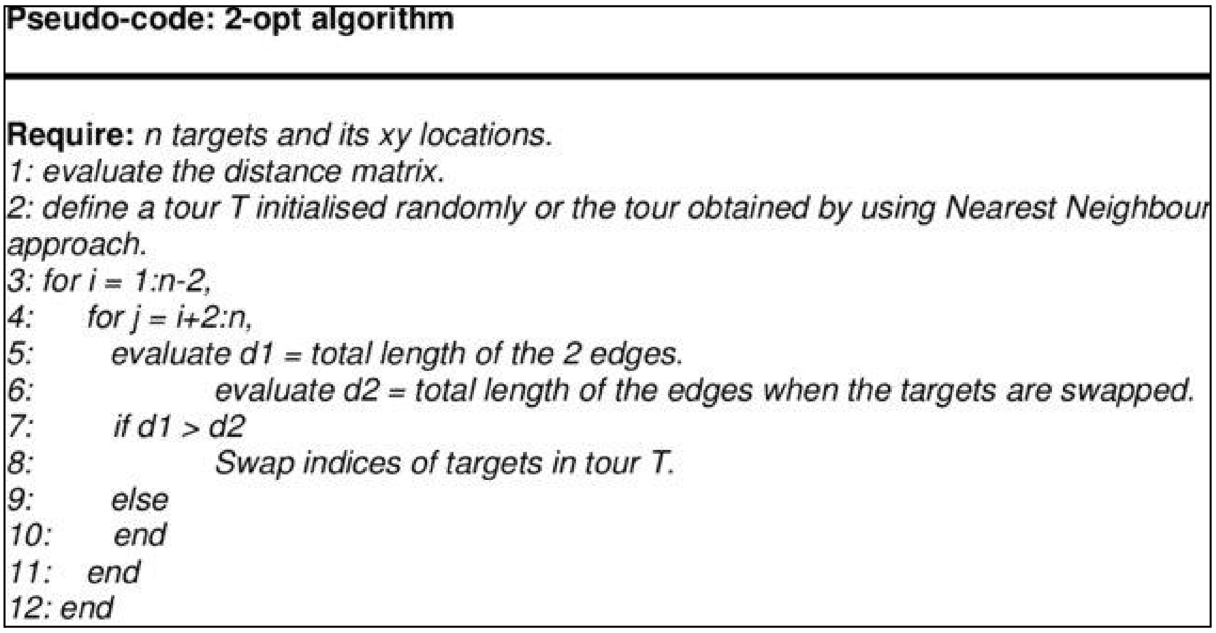
6. Accepting or Rejecting the Move: Compare the cost of the new tour with that of the previous tour. If the new tour is better (has a lower cost), accept the move and update the current tour. Otherwise, reject the move and keep the previous tour unchanged.

7. Termination Condition: Repeat steps 1-6 until no further improvement can be made or until a predefined stopping criterion is met.

The 2-opt algorithm iteratively applies these moves to improve the tour until it reaches a local minimum, where no further swaps can reduce the tour's cost. It's important to note that the 2-opt algorithm may not always find the optimal solution to the TSP but generally provides good-quality solutions efficiently. [7] [8]

2. The pseudocode of the Algorithm

Below is the pseudo code for the 2-opt algorithm, designed to optimize a given solution by swapping pairs of edges to reduce total cost:



```
Pseudo-code: 2-opt algorithm

Require: n targets and its xy locations.
1: evaluate the distance matrix.
2: define a tour T initialised randomly or the tour obtained by using Nearest Neighbour
approach.
3: for i = 1:n-2,
4:      for j = i+2:n,
5:        evaluate d1 = total length of the 2 edges.
6:                evaluate d2 = total length of the edges when the targets are swapped.
7:        if d1 > d2
8:                Swap indices of targets in tour T.
9:        else
10:       end
11:    end
12: end
```

Figure [14] : Pseudo-code for 2-opt algorithm from ResearchGate. [9]

1: best_path = GenerateRandomPath()

Create an initial random path.

2: best_path_distance = CalculatePathDistance(best_path)

Calculate the distance of the initial random path

3: improvement = True

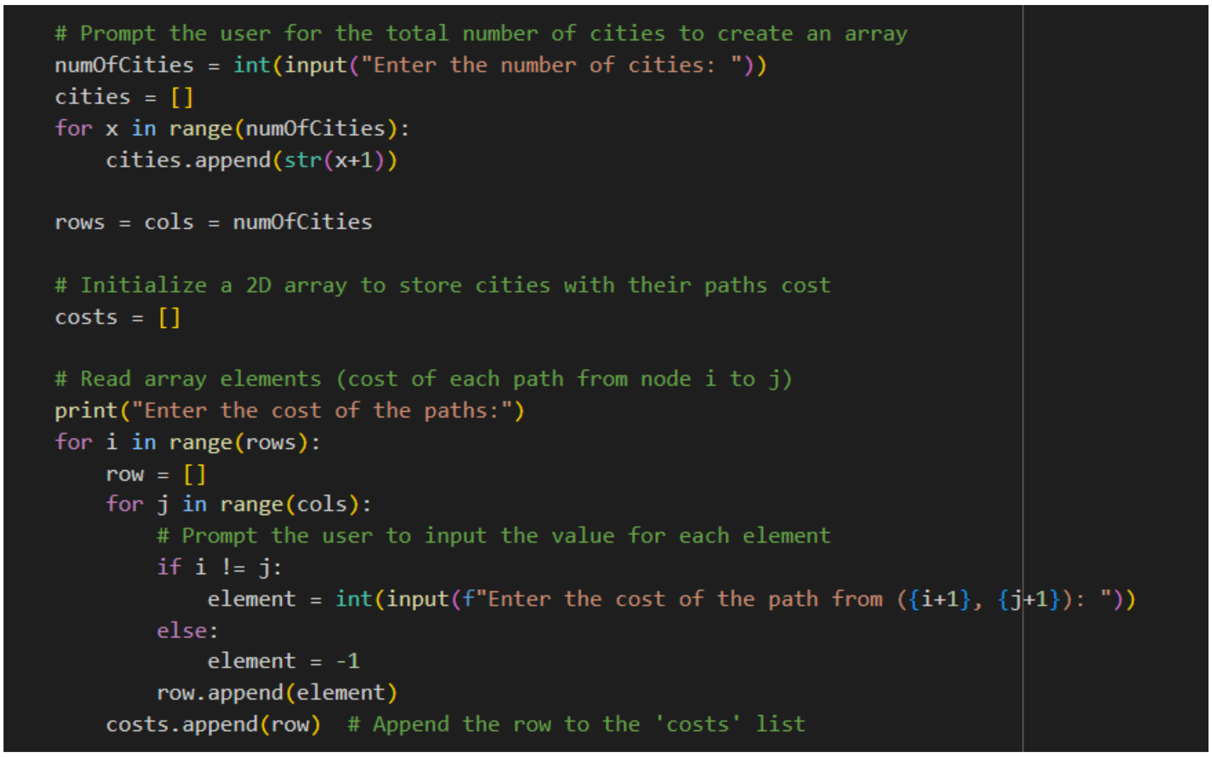
Set a flag to track whether any improvement has been made in the current iteration

```
4: while improvement:
5:     improvement = False
Assume no improvement has been made at the beginning of each iteration.
6:     for i in range(length(best_path) - 1):
Loop through each city in the path except the last one
7:         for j in range(i + 1, length(best_path)):
Loop through cities following the current city i
8:             new_path = OptSwap(best_path, i, j)
Generate a new path by swapping segments of the current path.
9:             new_path_distance = CalculatePathDistance(new_path)
Calculate the distance of the new path.
10:            if new_path_distance < best_path_distance:
Check if the new path is shorter than the current best path.
11:                best_path = new_path
Update the best path if the new path is better
12:                best_path_distance = new_path_distance :
Update the best path distance if the new path is better.
13:                improvement = True
 Set the improvement flag to true if a better path is found in this iteration.
14: return best_path, best_path_distance
```

3. Screenshots of Relevant Parts of the Code with Comments

The algorithm was implemented using Python language[10]. Beneath are some code snippets along with comments explaining the code:

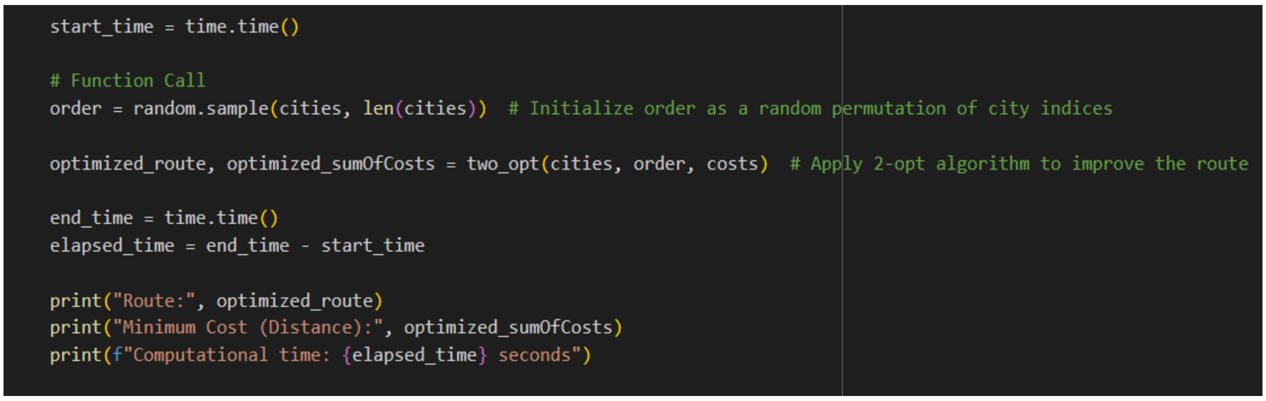
1. We started by prompting the user to enter the total number of cities to construct a 2D array called `costs` that stores the cost of each edge:



```python
# Prompt the user for the total number of cities to create an array
numOfCities = int(input("Enter the number of cities: "))
cities = []
for x in range(numOfCities):
    cities.append(str(x+1))

rows = cols = numOfCities

# Initialize a 2D array to store cities with their paths cost
costs = []

# Read array elements (cost of each path from node i to j)
print("Enter the cost of the paths:")
for i in range(rows):
    row = []
    for j in range(cols):
        # Prompt the user to input the value for each element
        if i != j:
            element = int(input(f"Enter the cost of the path from ({i+1}, {j+1}): "))
        else:
            element = -1
        row.append(element)
    costs.append(row)  # Append the row to the 'costs' list
```

Figure[15]: a code segment for reading from the user and constructing the 2D array part.

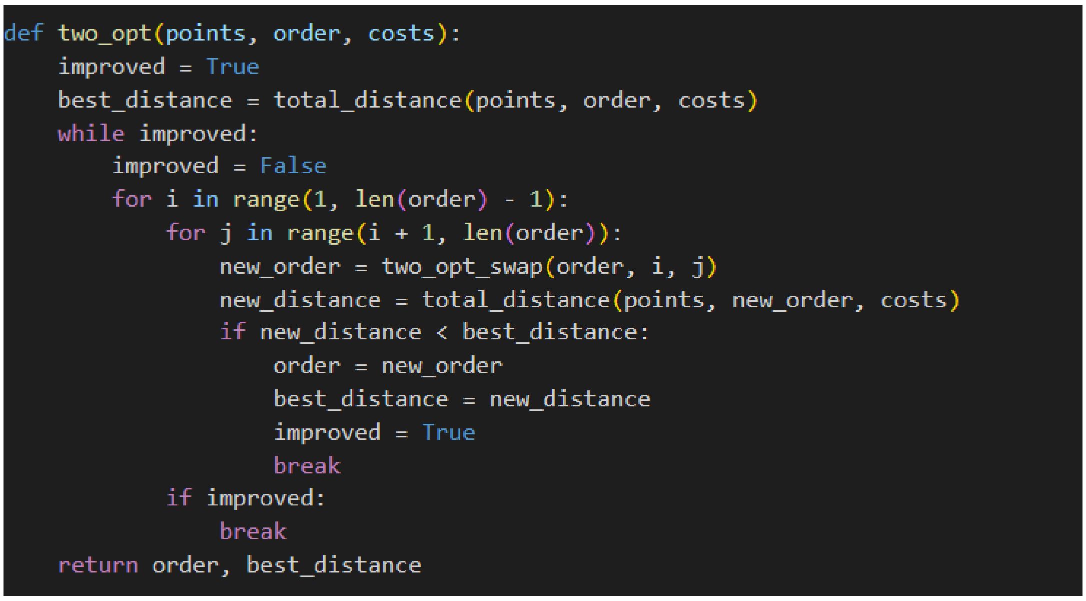
2. After that, we have started a timer to calculate the computational time, and executed the function `random.sample()`which is an inbuilt function in python to randomly order cities generating a random starting route, then performed the `two_opt()` function to improve the route:



```python
start_time = time.time()

# Function Call
order = random.sample(cities, len(cities))  # Initialize order as a random permutation of city indices

optimized_route, optimized_sumOfCosts = two_opt(cities, order, costs)  # Apply 2-opt algorithm to improve the route

end_time = time.time()
elapsed_time = end_time - start_time

print("Route:", optimized_route)
print("Minimum Cost (Distance):", optimized_sumOfCosts)
print(f"Computational time: {elapsed_time} seconds")
```

Figure[16]: a code segment showing the part that applies the algorithm using random.sample() and two_opt() functions.

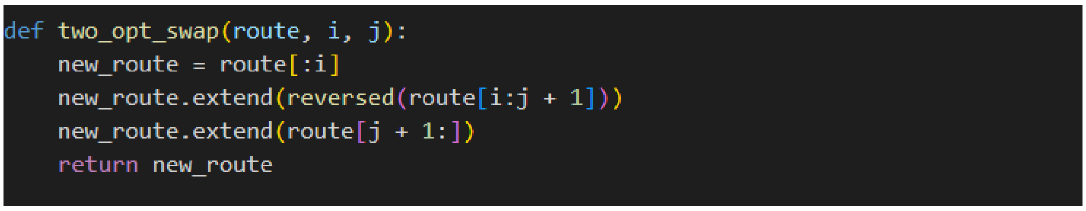
3. Having a closer look into the `two_opt()` function, it executes the 2-opt algorithm by continually searching for improved routes. If no better route is found, indicating a route with a total distance less than the initially constructed one, the function iterates through edges, performing swaps to generate a new route order. It calculates the distance of each new route until finding one with a lower distance than the previous one:



```python
def two_opt(points, order, costs):
    improved = True
    best_distance = total_distance(points, order, costs)
    while improved:
        improved = False
        for i in range(1, len(order) - 1):
            for j in range(i + 1, len(order)):
                new_order = two_opt_swap(order, i, j)
                new_distance = total_distance(points, new_order, costs)
                if new_distance < best_distance:
                    order = new_order
                    best_distance = new_distance
                    improved = True
                    break
            if improved:
                break
    return order, best_distance
```

Figure[17]: a code segment showing the two_opt() function body.

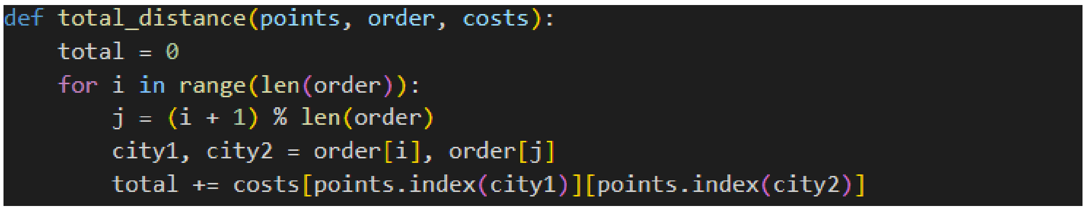
4. For the details of the swapping operation, it's performed through executing `two_opt_swap()`, which reverses the segment of cities between indices `i` and `j` in the given route, and then returns the modified route:



```python
def two_opt_swap(route, i, j):
    new_route = route[:i]
    new_route.extend(reversed(route[i:j + 1]))
    new_route.extend(route[j + 1:])
    return new_route
```

Figure[18]: a code segment showing the two_opt_swap() function body.

5. Lastly we have the `total_distance()` function, which computes the total distance traveled along a given route by summing up the costs associated with traveling between consecutive cities:



```python
def total_distance(points, order, costs):
    total = 0
    for i in range(len(order)):
        j = (i + 1) % len(order)
        city1, city2 = order[i], order[j]
        total += costs[points.index(city1)][points.index(city2)]
```

Figure[19]: a code segment showing the total_distance() function body.

- The complete code explained above could be found in the following link: SWE485-Project-Phase3.ipynb

4. Comparison of the results with the Incremental formulation implemented in Phase 2 (objective function and computational time)

To compare the objective function (minimum cost) and computational time between the greedy and 2-opt methods, we ran both algorithms for the same input data and then analyzed the results. In the following tables, we present the comparison of the objective function and computational time for the two methods:

Input Data:

- Number of cities: 4
- Cost of Paths:

	1	2	3	4
1	0	5	20	15
2		0	25	6
3			0	38
4				0

Table[2]: Cost of Paths Matrix.

In this matrix:

- The diagonal elements (0) represent that there is no cost associated with traveling from a city to itself.
- The other elements represent the cost of traveling from one city to another. For instance, traveling from city 1 to city 2 costs 5, and it's the same from city 2 to city1.

Results:

Algorithm	Objective Function: Minimum Cost (Distance)	Computational Time in sec	Route
Greedy	69	0.0026466846	3-1-2-4
2-opt	66	0.0001165866	3-1-4-2

Table[3]: Results of greedy and 2-opt algorithms..

Based on our observation of the provided dataset, we found that the 2-opt method consistently produced solutions with lower costs and marginally faster computational times compared to the greedy algorithm. However, it's important to note that the performance of these algorithms can vary depending on the characteristics of the problem instance. While the 2-opt method outperformed the greedy algorithm in this specific scenario, we cannot guarantee that it will always provide the optimal solution. The effectiveness of each algorithm depends on factors such as the structure of the cost matrix, the size of the dataset, and the randomness inherent in the greedy algorithm's initial solution. Therefore, when choosing between these algorithms, it's essential to consider the specific requirements of the problem and conduct thorough testing to determine the most suitable approach.

Additionally, it's worth noting that the greedy algorithm's solution can be influenced by the choice of the first city, while the 2-opt method uses a randomly chosen initial solution, adding another layer of variability to its performance, and its generally more effective at finding high-quality solutions compared to greedy algorithms.

References

[1] “Traveling salesperson problem: Dantzig-Fulkerson-Johnson formulation,” YouTube, https://youtu.be/5IbbUkmHidE?si=8tYmCUf2MpLe5vRK (accessed Mar. 5, 2024).

[2] X. Geng, Z. Chen, W. Yang, D. Shi, and K. Zhao, “Solving the traveling salesman problem based on an adaptive simulated annealing algorithm with greedy search,” *Applied Soft Computing*, vol. 11, no. 4, pp. 3680–3689, Jun. 2011. doi:10.1016/j.asoc.2011.01.039

[3] “Travelling salesman problem,” Wikipedia, https://en.wikipedia.org/wiki/Travelling_salesman_problem (accessed Mar. 31, 2024).

[4] B. Alsalibi, M. Babaeianjelodar, and I. Venkat, “(PDF) a comparative study between the nearest neighbor and genetic ...,” ResearchGate, https://www.researchgate.net/publication/258031702_A_Comparative_Study_between_the_Nearest_Neighbor_and_Genetic_Algorithms_A_revisit_to_the_Traveling_Salesman_Problem (accessed Mar. 31, 2024).

[5] S. Ejim, “(PDF) implementation of greedy algorithm in travel salesman problem,” ResearchGate, https://www.researchgate.net/publication/307856959_Implementation_of_Greedy_Algorithm_in_Travel_Salesman_Problem (accessed Mar. 31, 2024).

[6] GfG, “Travelling salesman problem: Greedy Approach,” GeeksforGeeks, https://www.geeksforgeeks.org/travelling-salesman-problem-greedy-approach/ (accessed Mar. 31, 2024).

[7] “2-opt,” Wikipedia, https://en.wikipedia.org/wiki/2-opt (accessed May 7, 2024).

[8] A. Davis, “Traveling salesman problem with the 2-opt algorithm,” Medium, https://slowandsteadybrain.medium.com/traveling-salesman-problem-ce78187cf1f3 (accessed May 7, 2024).

[9] A. Sathyan, *Pseudo-Code for 2-Opt Algorithm*. 2015.

[10] adavis-85, “ Traveling-Salesman-2-opt-with-Visualization,” Github, https://github.com/adavis-85/Traveling-Salesman-2-opt-with-Visualization/blob/main/TSP%20Functions.jl (accessed 2024).