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

Phase 3: Complete formulation

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: 2024/5/7 -1445/10/28

Table of contents

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. [1] [2]

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:

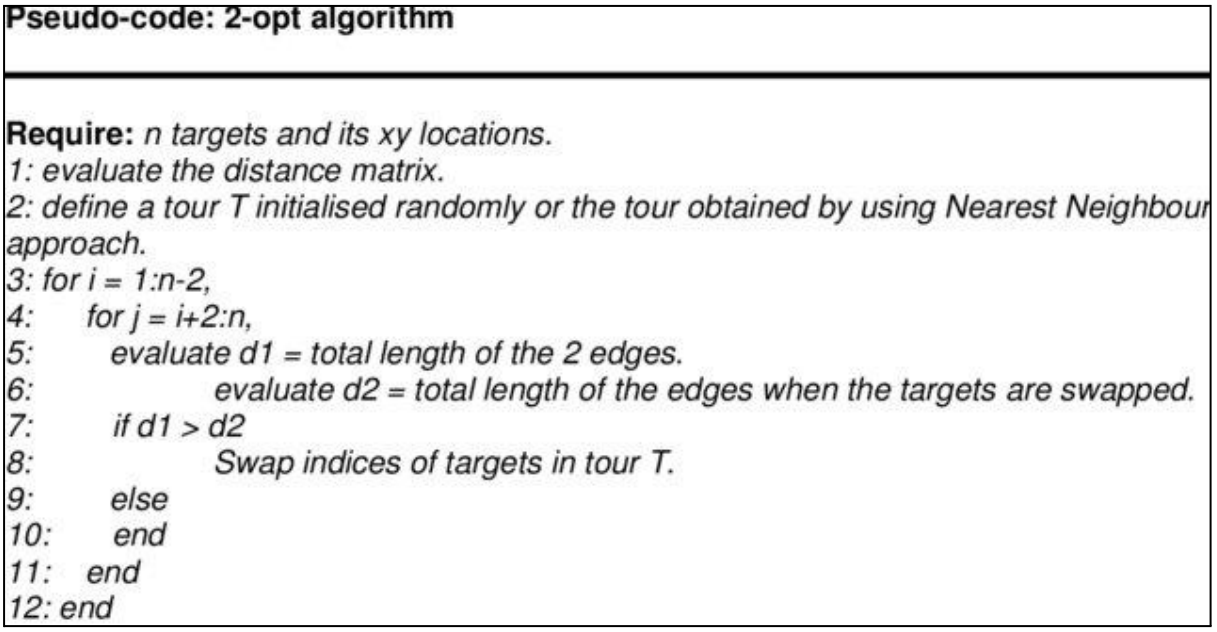

Figure [1] : Pseudo-code for 2-opt algorithm. (n.d.). ResearchGate. [3]

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[4]. 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:

```
# 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[2]: 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:

```
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[3]: 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:

```
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[4]: 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:

```
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[5]: 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:

```
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[6]: 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[1]: 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	0.0003390312	2-4-3-1
2-opt	66	0.0001165866	3-1-4-2

Table[2]: 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.

References

[1] “2-opt,” Wikipedia, https://en.wikipedia.org/wiki/2-opt (accessed May 7, 2024).

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

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

[4] 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).