

[SWE485] Selected Topics in Software Engineering Project

Applying Search Techniques in a real-world problem

Phase 2: Incremental 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: 2024/3/31 -1445/9/21

Table of contents

1.The Heuristic(s)	3
2. The Data Structure	3
3. The pseudocode of The Algorithm	4
4. Screenshots of Relevant Parts of the Code with Comments	7
5. Testing with Randomly generated/real world instances	11
6. Results and Computational Time of The Algorithm	12
References	13

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^[1].

Nearest Neighbor Heuristic:

One of the most popular heuristics for solving the TSP is the Nearest Neighbor heuristic, which is a greedy algorithm^[2]. 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^[3]. 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^[4].

2. The Data Structure

- **Matrix (2D List):** Represents the distances between cities. Each element `tsp[i][j]` holds the distance from city `i` to city `j`.
- **Visited List:** A dictionary that keeps track of whether a city has been visited. If `visitedRouteList[i]` is set to **1**, it means the city `i` has been visited.
- **Route List:** A list To record the sequence of visited cities.

3. The pseudocode of The Algorithm

1. Start from an initial city:

- Set this city as the first city and mark it as visited.

2. 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.

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 (`sumOfCosts`), how many cities have been visited (`visitedCitiesCounter`), and the minimum cost found in each iteration (`min`). The `visitedRouteList` keeps track of which cities have been visited to prevent revisiting, and route stores the path taken.

```
Function findMinRoute(TSP_Matrix, StartCity)
    Initialize sumOfCosts to 0
    Initialize visitedCitiesCounter to 0
    Set min to a very high value (e.g., INT_MAX)
    Initialize visitedRouteList as a dictionary with all values set to 0
    Initialize route list with the first city as the starting point

    Mark the first city as visited in visitedRouteList
```

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

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

```
Loop until all cities are visited
    If visitedCitiesCounter is equal to the number of cities - 1
        Exit loop

    For each city j in TSP_Matrix starting from city i
        If the path from city i to j is unvisited and the cost is less than min
            Update min to this new cost
            Update the current position in route to city j + 1

    Increment j by 1

    If all cities have been checked
        Add min to sumOfCosts
        Reset min to INT_MAX
        Mark the current city in the route as visited
        Reset j to 0
        Update i to the last city visited
        Increment visitedCitiesCounter by 1
```

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

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

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

```
Update i to the last city visited before returning

For each city j
    If the path from city i to j and back to the start city has a cost less than min
        Update min to this new cost
        Update the route to include city j + 1

Add this last min cost to sumOfCosts
```

Figure[3]: a pseudocode segment showing the part of calculating cost of returning to the firstCity

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

```
Print the total minimum cost and the route taken

End Function
```

Figure[4]: 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^[5]. 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 `tsp` that stores the cost of each edge. We then asked for the first city to start traveling from, and lastly called the function `findMinRoute(tsp, firstCity)` to start applying our search algorithm:

```
# Driver Code
if __name__ == "__main__":

    # Prompt the user for the total number of cities to create an array
    numOfCities = int(input("Enter the number of cities: "))
    rows = cols = numOfCities

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

    # 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)
        tsp.append(row)

    # Prompt the user to input the value of the firstCity to travel from
    firstCity = int(input("Enter the first city to travel from: "))

    start_time = time.time()
    # Function Call
    findMinRoute(tsp, firstCity)
```

Figure[5]: a code segment showing the part that reads data from the user.

2. Inside `findMinRoute(tsp, firstCity)` , we first initialized the needed variables:

```
INT_MAX = 2147483647

# Function to find the minimum
# cost path for all the paths
def findMinRoute(tsp, firstCity):
    sumOfCosts = 0 #keeps track of the minimum cost route
    visitedCitiesCounter = 0 #keeps track of the number of visited cities
    j = 0
    i = 0
    min = INT_MAX
    visitedRouteList = DefaultDict(int)
    route = [firstCity]
```

Figure[6]: a code segment showing the beginning of `findMinRoute(tsp, firstCity)` function.

Variable	Used for
<code>sumOfCosts</code>	keeps track of the minimum cost route
<code>visitedCitiesCounter</code>	keeps track of the number of visited cities
<code>i & j</code>	traverse the 2D array
<code>min</code>	store the minimum cost, it was initialized to a very huge number <code>INT_MAX</code> to ensure that the first encountered cost value will be smaller than <code>min</code> , effectively updating <code>min</code> to that smaller value.
<code>visitedRouteList</code>	Dictionary to track visited cities
<code>route</code>	List to store the route

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

3. We have then marked the first city as visited and created a `route` list initialized with 0s:

```
# Starting from the 0th indexed
# city i.e., the first city
visitedRouteList[0] = firstCity
route = [0] * len(tsp)
```

Figure[7]: a code segment showing the initialization of `visitedRouteList[]` and `route` list.

4. 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:

```
# Traverse the adjacency
# matrix tsp[][]
while i < len(tsp) and j < len(tsp[i]):

    # Corner of the Matrix
    if visitedCitiesCounter >= len(tsp[i]) - 1:
        break

    # If this path is unvisited then
    # and if the cost is less then
    # update the cost
    if j != i and (visitedRouteList[j] == 0):
        if tsp[i][j] < min:
            min = tsp[i][j]
            route[visitedCitiesCounter] = j + 1

    j += 1
```

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 from i, we added the minimal path cost to the sumOfCosts, marked the city as visited, and prepared for visiting the next city in the route:

```
# Check all paths from the
# ith indexed city
if j == len(tsp[i]):
    sumOfCosts += min
    min = INT_MAX
    visitedRouteList[route[visitedCitiesCounter] - 1] = 1
    j = 0
    i = route[visitedCitiesCounter] - 1
    visitedCitiesCounter += 1
```

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 finding the minimum cost from the last visited city to any unvisited city:

```
for j in range(len(tsp)):

    if (i != j) and tsp[i][j] < min:
        min = tsp[i][j]
        route[visitedCitiesCounter] = j + 1

sumOfCosts += min
```

Figure[10]: a code segment showing the part that finds the minimum cost of cities to visit.

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

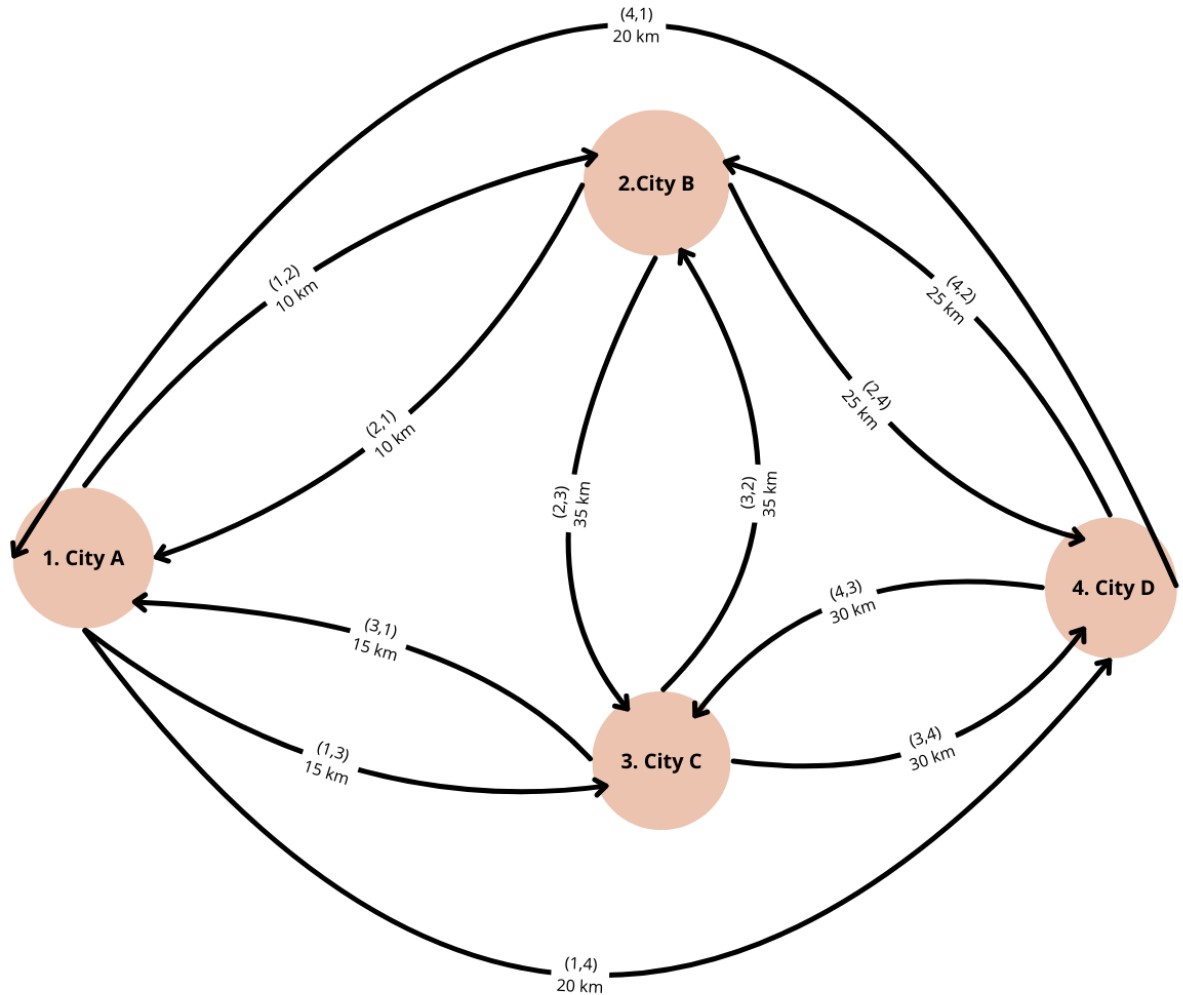
```
print("Minimum Cost is :", sumOfCosts)
print("Route:", route)
```

Figure[11]: a code segment showing the print statement of sumOfCosts 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

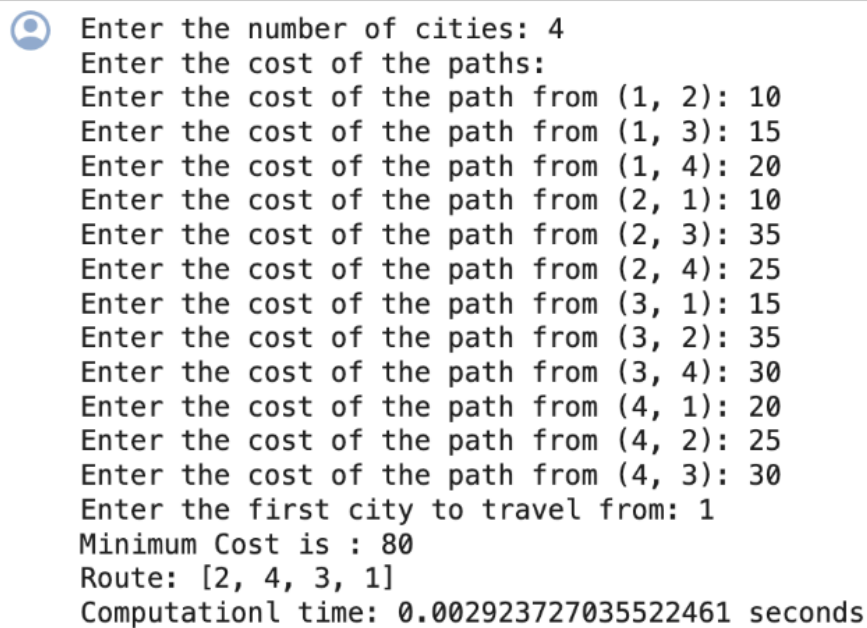
In the example, we are trying to find out the path/route with the minimum cost such that our aim is visiting city B, city C and city D once and return back to the source city A is achieved.



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 travel from: 1
Minimum Cost is : 80
Route: [2, 4, 3, 1]
Computationl time: 0.002923727035522461 seconds
```

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

References

- [1] 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
- [2] "Travelling salesman problem," Wikipedia,
https://en.wikipedia.org/wiki/Travelling_salesman_problem (accessed Mar. 31, 2024).
- [3] 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).
- [4] 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).
- [5] GfG, "Travelling salesman problem: Greedy Approach," GeeksforGeeks,
<https://www.geeksforgeeks.org/travelling-salesman-problem-greedy-approach/> (accessed Mar. 31, 2024).