Assignment: Report

**MAKERERE** **UNIVERSITY**

****

**TRAVEL SALESMAN PROBLEM REPORT (TSP):**

**(Using Classical & SOM-Based Methods)**

Instructor (**Mr. Denish**): MAKERERE UNIVERSITY

By:

**GROUP\_S**

COLLEGE OF COMPUTING AND INFORMATION SCIENCES

SCHOOL OF COMPUTING AND INFORMATICS TECHNOLY

DEPARTMENT OF COMPUTER SCIENCE

BACHELOR OF SCIENCE IN COMPUTER SCIENCE (Year One)

(CSC: 1204). Data Structures & Algorithms Assignment of **Travel Salesman Problem (TSP) Using Classical & SOM-Based Methods.**

**semester II** Academic Year:

**2024:**

**Due date: 25TH March 2025**

# Assignment Team Members

|  |
| --- |
| Name REG.NO Role/Responsibility |

|  |  |  |
| --- | --- | --- |
| 1. Seanice Nabasirye   (0770862549) | 24/U/22871 | Representation & Data Structures |
| 1. Asiimwe Isaac   (0770563394) | 24/U/03765/PS | Assignment Task Manager. |
| 1. Kiyingi Solomon Jessy   (0767594340) | 24/U/25920/EVE | Analysis and Comparison |
| 1. Jonah Akandwanaho   (0756348528) | 24/U/03053/PS | SOM-Based Approach |
| 1. Keith Paul Kato   (0759021371) | 24/U/26593/EVE | Assignment Report/Documentation |
| 1. Isaac |  | Analysis and Comparison |

# **Task 1: Representation & Data Structures:**

The **(TSP)** can be effectively represented using an **Adjacent Matrix Approach,** a data structure most appropriate for storing the graph representation of the cities and the distances. It uses a 2D-Array matrix that can also be used for a dense graph where all nodes are connected as illustrated by the graph representation of the city distances **STRUCTURE OF ADJACENT MATRIX APPROACH.**

The graph representation of cities and distances has 7 cities therefore a (**7\*7)** matrix is used. In addition to that; if there’s no direct route between any city, so we set matrix **[i][j]** = **∞** which means “Infinite distance” or “no connection between the cities” where i and j are cities. Furthermore, since the graph is undirected (there can be a route to and from two cities i.e., from city 1 to city 2 & vice versa), the matrix is symmetric, implying (**matrix[i][j]** = **matrix[j][i]**). With diagonal elements set to **zero (0)**. This representation allows efficient look up. **REASONS FOR CHOOSING ADJACENT MATRIX APPROACH.**

1. The adjacent matrix is good for dense graphs, it efficiently stores all connections.
2. Checking if an edge exists between **2 nodes** is **O(1). O(1)** means that, the operation takes the same amount of time regardless of how many elements are in the data structure.
3. It is easy to implement using a **2D** array.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 1 | 0 | 12 | 10 | ∞ | ∞ | ∞ | 12 |
| 2 | 12 | 0 | 8 | 12 | ∞ | ∞ | ∞ |
| 3 | 10 | 8 | 0 | 11 | 3 | ∞ | 9 |
| 4 | ∞ | 12 | 11 | 0 | 11 | 10 | ∞ |
| 5 | ∞ | ∞ | 3 | 11 | 0 | 6 | 7 |
| 6 | ∞ | ∞ | ∞ | 10 | 6 | 0 | 9 |
| 7 | 12 | ∞ | 9 | ∞ | 7 | 9 | 0 |

**Objectives of the TSP.**

* The traveler must visit each and every city without leaving any unvisited.
* The traveler has to get back to the starting point after each and every visit of the cities.
* The traveler must make sure that the travelled distance is the least possible option that allows all cities to be visited.

**Assumptions of the TSP.**

* Each city is visited exactly only once without repeating.
* Only the starting point/city is the only city visited twice which is a return visit.
* The travel is a cycle, in that, the visits eventually return back to the starting point.
* Every city can be reached from any other city where a connection exists; thus, the connections are bi-directional.
* The traveler has no time constraints

# **Task 2: Classical TSP Solution:**

1. **Algorithm Selection:**

Ourchosen classical TPS method is dynamic programming (Held-Karp Algorithm)**.**

* **Justification:** Based on Accuracy, (where we must find the shortest route), efficiency (where the program should run in a reasonable time) and suitability (for practicability with small dataset.)
* **Optimal solution:** same as **BnB** but faster. Unlike greedy algorithms like **NN,** it finds the globally optimal solution.
* **Dramatically** reduces **redundant** calculations compared to brute force.
* **More scalable** than **BnB** for medium sized graphs.
* **Best choice** for exact results when **“n”** is small to moderate **(n ≤ 20) however** if “n” is **very large (>20),** **heuristics** like **NN** or genetic algorithms are better

1. **Implementation:**

Dynamic programminguses **memoisation** to optimize recursive subproblems, reducing complexity from **to**

* **State Representation:**
* Will use a **bit masking approach** to represent **subsets** of visited cities.
* We will define dp**[mask][j]** using previous states: dp**[mask][j]** = **min**(dp[**mask\{j}][i]** + cost**[i][j])** where **i** is any previouslyvisited city in the mask.
* **Base Case: dp[1][0] = 0 (starting at City 1).**
* **Time and space complexity** of TSP using Dynamic programming and Bit masking
* Time complexity analysis: The state of the DP is determined by,
* **Current City:** There are **“n”** cities. (7cities)
* **Visited mask:** There are **“”** possible subsets of cities. ()
* Since every function call makes a loop over all cities**,** the **time complexity is:**
* **Breakdown:**
* There are unique states in the DP table.
* And each state makes the most of **n** recursive calls.
* Since memoisation ensures state is computed only once, we avoid redundant precomputation.
* Thus, the final time complexity iswhich is exponential but significantly faster than brute force approach **O(n!)**
* **Space Complexity Analysis**
* The space used comes from the **DP memoisation Table:**
* The **@lru\_cache** stores result for atmost **states.**
* Each State holds an integer value **.**
* **Space used:**
* **Recursive Call Stack:**
* In the worst case, the recursion depth is **O(n)** space where each city is visited once before returning.
* This contributes an additional **O(n)** space. Thus, the **total space complexity is**

1. **Results: Calculations:**

**Example:**

**Formular:**

***S =*** *Set of vertices that the sales man**will visit exactly once and then come back to the start vertex*

* **We expand on these until get the overall minimum cost by taking the minimum cost between each route as we go back to the starting point. (1) between each root.**
* **For visual representation (a recursive tree can be demonstrated)**

**For the Code:**

* **Formular:**

* **Where:**
* represents the minimum cost to visit all cities in set S, ending at city **i.**
* The function recursively considers all cities in S and finds the minimum possible cost
* Base Case: When only the starting city is left, return the cost to return to it. i.e.,
* We start at city 1 **(i = 0).**
* The initial visited mask is **0000001 (Binary for only city 1 visited)**
* The function call is: tsp (0, 1)
* Exploring all next possible cities (**non-infinity paths**): Cities**, 2, 3, 7** with costs **12,** The function call is **tsp (0, 1)**
* If we move to city 2, the visited mask becomes **0000011** **(cities 1 & 2 visited): Recursive call: - tsp (1, 3)**
* We go on and on until we backtrack to sum up the minimum costs found along the way
* **tsp** (5, 63) = 9 + 12 = 21
* **tsp** (4, 31) = 7 + 21 = 28
* **tsp** (3, 15) = 11 + 28 = 39
* **tsp** (1, 3) = 8 + 42 = 50
* **tsp** (0, 1) = 12 + 50 = 62
* **tsp** (current city, visited set)