# Travelling Salesman Problem

Mini-project C++

Travelling salesman is a problem that can be tackled using Distance matrices. A distance matrix is two-dimensional array of equal number of rows and columns. It can be presented in the following way: -

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Place/Place | A | B | C | D |
| A | 0 | 10 | 20 | 30 |
| B | 10 | 0 | 35 | 25 |
| C | 20 | 35 | 0 | 50 |
| D | 30 | 25 | 50 | 0 |

This table above gives us the distances between a few places given. The TSP can be solved by putting this exact matrix in the code terminal.

Like we can see in the distance matrix above the distance between A-A; B-B and so on is 0 as it’s the same location. The distance between A-B is same as B-A since distance is a linear property.

We use the limits library to access numeric limits or setting maximum values. The provided C++ code solves the Traveling Salesman Problem (TSP), a classic optimization problem where a salesperson aims to find the shortest route to visit a set of cities exactly once and return to the starting city. The code uses the Nearest Neighbour Algorithm to find an approximate solution.

The program starts by reading the number of cities and a distance matrix from the user. The distance matrix represents the distances between cities, where **graph[i][j]** contains the distance from city **i** to city **j**. The code initializes various data structures to manage the cities, distances, and visited status.

The Nearest Neighbour Algorithm begins from an initial city chosen by the user. It iteratively selects the nearest unvisited city to the current city and moves to it. This process continues until all cities have been visited. The algorithm then returns to the initial city to complete the tour.

During the process, the code keeps track of the total distance travelled. The Nearest Neighbour Algorithm is not guaranteed to produce the optimal solution but often provides a reasonably good one in a short time.

For instance, if the salesperson wants to visit several cities and the code receives the distances between them, it calculates the order in which the cities should be visited to minimize the total distance. The resulting path is printed, and the total distance travelled is displayed as the output.

The code uses arrays and loops for efficient computation, as well as conditional statements to handle user input and ensure that the initial city is valid. It also maintains consistency by employing the **<limits>** library to represent infinity, making it a common practice when working with optimization problems where no connection between cities can be represented as an unbounded value. Overall, this code demonstrates a fundamental approach to solving the TSP and can serve as a starting point for more advanced TSP solvers.

A computer screen shot of a program code

Description automatically generatedA black screen with white text

Description automatically generated

