A MINI PROJECT REPORT 18CSC204J -

Design and Analysis of Algorithms Laboratory

Submitted by

SUJAL CHORDIYA(RA2111003010938)

AND

PRANJALI SHARMA(RA2111003010939)

Under the guidance of

Dr.M.L.Sworna Kokila

Assistant Professor, Department of Networking and Communication

In Partial Fulfillment of the Requirements for the Degree of

BACHELOR OF TECHNOLOGY in COMPUTER SCIENCE ENGINEERING of Faculty of engineering and technology

DEPARTMENT OFNETWORKING AND COMMUNICATIONS COLLEGE OF ENGINEERING AND TECHNOLOGY SRM INSTITUTE OF SCIENCE AND TECHNOLOGY KATTANKULATHUR- 603 203

May 2023

SRM INSTITUTE OF SCIENCE AND TECHNOLOGY
KATTANKULATHUR – 603203

BONAFIDE CERTIFICATE

Certified that this B. Tech mini project report titled "**TRAVELLING SALESMAN PROBLEM**" is the bonafide work of Sujal Chordiya(RA2111003010938) and Pranjali Sharma(RA2111003010939) who carried out the project work under my supervision for **18CSC204J-Design and Analysis of Algorithms Laboratory**. Certified further, that to the best of my knowledge the work reported herein does not form part of any other thesis or dissertation on the basis of which a degree or award was conferred on an earlier occasion for this or any other candidate.

Dr.M.L.Sworna Kokila	**Dr.M.Pushpalatha**
ASSISTANT PROFESSOR	**Professor and Head**
Department of Networking and Communications	Department of Networking and Communications

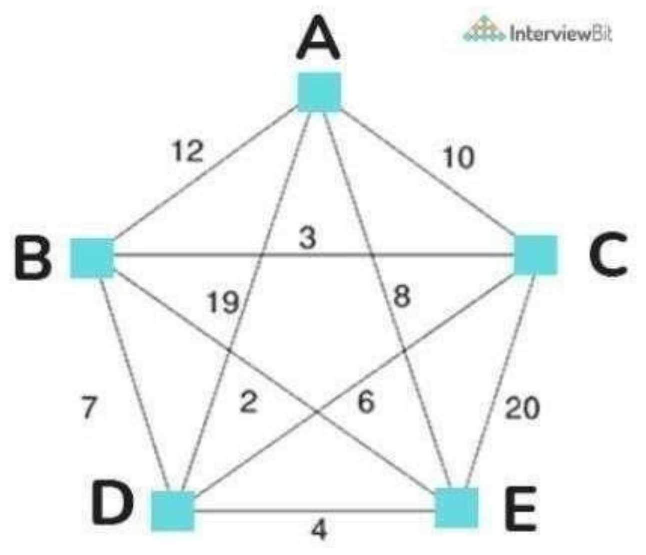
TABLE OF CONTENTS

Contribution Table:

Person	**Contribution**
Sujal Chordiya	50%
Pranjali Sharma	50%

Problem Statement:

Given a set of cities and the distance between every pair of cities as an adjacency matrix, the problem is to find the shortest possible route that visits every city exactly once and returns to thestarting point.



The above Problem can be solved using Travelling Salesman Approach using Dynamic Programming, Greedy Algorithm and Simple Approach

Problem Explanation:

The Travelling Salesman Problem (TSP) is a classic optimization problem in computer science and operations research. The problem can be stated as follows: given a list of cities and the distances between each pair of cities, find the shortest possible route that visits each city exactly once and returns to the starting city.

The TSP is an NP-hard problem, which means that the time required to solve the problem exactly increases exponentially with the number of cities. Therefore, for large instances of the problem, it is often necessary to use heuristic or approximate algorithms that provide good solutions in a reasonable amount of time.

One common heuristic algorithm for the TSP is the nearest neighbor algorithm, which starts at a given city and repeatedly visits the nearest unvisited city until all cities have been visited. Another heuristic algorithm is the 2-opt algorithm, which improves upon an initial tour by iteratively swapping pairs of edges that result in a shorter tour.

Despite the difficulty of the TSP, the problem has important practical applications in fields such as logistics, transportation, and manufacturing, where finding an optimal route can significantly reduce costs and improve efficiency.

Travelling Salesman Problem Using Dynamic Programming

In the travelling salesman problem algorithm, we take a subset N of the required cities that need to be visited, the distance among the cities dist, and starting city s as inputs. Each city is identified by a unique city id which we say like 1,2,3,4,5………n

Here we use a dynamic approach to calculate the cost function Cost(). Using recursive calls, we calculate the cost function for each subset of the original problem.

To calculate the cost(i) using Dynamic Programming, we need to have some recursive relation in terms of sub-problems.

We start with all subsets of size 2 and calculate C(S, i) for all subsets where S is the subset, then we calculate C(S, i) for all subsets S of size 3 and so on.

Algorithm:

1.Define the number of vertices V in the graph and the starting vertex s.

2.Create a vector to hold all vertices except the starting vertex.

3. Initialize the minimum path min_path to the maximum integer value.

4. Generate all permutations of the remaining vertices using the next_permutation function from the <algorithm> library.

5. For each permutation, calculate the path weight by starting at the starting vertex s, and adding the edge weights for each consecutive vertex in the permutation.

6. Add the weight of the edge from the last vertex to the starting vertex.

7. Update the minimum path min_path with the current path weight.

8. Return the minimum path min_path.

CODE IMPLEMENTATION USING C++:

```
#include <bits/stdc++.h> using namespace

std;

#define V 4 int travellingSalesmanProblem(int graph[][V],

int s)

{ vector<int> vertex; for


(int i = 0; i < V; i++)         if (i

!= s)

vertex.push_back(i);


    int min_path = INT_MAX; do
    {         int

current_pathweight = 0;

        int k = s;         for (int i = 0; i < vertex.size();

i++) {             current_pathweight +=
graph[k][vertex[i]];k = vertex[i];

        }

        current_pathweight += graph[k][s];
        // update minimum          min_path =

min(min_path, current_pathweight);


    } while (         next_permutation(vertex.begin(),

vertex.end()));



    return min_path;

}

int main()

{     int graph[][V] = { { 0, 10, 15, 20

},


                       { 10, 0, 35, 25 }, {

                       15, 35, 0, 30 },


{ 20, 25, 30, 0 } };     int s = 0;


    cout << travellingSalesmanProblem(graph, s) << endl;      return 0;


}
```

OUTPUT:

```
10
```

TIME COMPLEXITY:

To analyze the time complexity of the traveling salesman problem (TSP) using a dynamic programming approach, we can use the following recurrence relation:

Let T(S, i) be the minimum cost of a path that starts at city 1, visits all cities in set S exactly once, and ends at city i.

Then, we can define the recurrence relation as follows:

T(S, i) = min { C(j, i) + T(S-{i}, j) } for j in S and j not equal to i,

where C(j, i) is the cost of traveling from city j to city i. The base

case of the recurrence relation is:

T({1}, 1) = 0, and T(S, i) = infinity if |S| = 1 and i is not equal to 1.

Using this recurrence relation, we can compute the minimum cost of a path that starts at city 1, visits all cities in the set S exactly once, and ends at city i, for all possible subsets S of {2, 3, ..., n} and for all possible ending cities i in S.

The time complexity of solving the TSP using dynamic programming is O(n^2 * 2^n), where n is the number of cities. This is because there are 2^n possible subsets of cities, and for each subset, we need to consider n possible ending cities, and for each ending city, we need to compute the minimum cost of a path, which takes O(n) time.

However, this approach is only practical for small instances of the problem, since the time complexity grows exponentially with the number of cities. For larger instances, heuristic and approximation algorithms are used to find good solutions in a reasonable amount of time.

Travelling Salesman Problem Using Greedy Approach

This problem can be related to the Hamiltonian Cycle Problem, in a way that here we know a Hamiltonian cycle exists in the graph, but our job is to find the cycle with minimum cost. Also, in a particular TSP graph, there can be many hamiltonian cycles but we need to output only one that satisfies our required aim of the problem.

Approach: This problem can be solved using Greedy Approach. The steps are given below:

1. Create two primary data holders:
   - A list that holds the indices of the cities in terms of the input matrix of distances between cities.
   - Result array which will have all cities that can be displayed out to the console in any manner.
2. Perform traversal on the given adjacency matrix tsp[][] for all the city and if the cost of reaching any city from the current city is less than current cost the update the cost.
3. Generate the minimum path cycle using the above step and return their minimum cost.

CODE IMPLEMENTATION USING C++:

```
#include <bits/stdc++.h> using namespace

std;



// Function to find the minimum // cost path for

all the paths void
```

```
    findMinRoute(vector<vector<int> > tsp)

    {      int sum =

    0;     int

    counter = 0;

    int j = 0, i = 0;

       int min = INT_MAX;

    map<int, int> visitedRouteList;


      // Starting from the 0th indexed city i.e., the first city

 visitedRouteList[0] = 1;

    int route[tsp.size()];


      // Traverse the adjacency matrix tsp[][] while

    (i < tsp.size() && j < tsp[i].size()) {



            // Corner of the Matrix

    if (counter >= tsp[i].size() - 1)

            {
break;

            }



       // If this path is unvisited then and if the cost is less then update the cost        if
(j != i && (visitedRouteList[j] == 0))
          {               if (tsp[i][j]

< min)
                  {

                              min      =

                         tsp[i][j]; route[counter]
                         = j + 1;

                  }

          }

j++;



          // Check all paths from the ith indexed city

if (j == tsp[i].size()) {

sum += min;          min = INT_MAX;
                   visitedRouteList[route[counter] - 1]

= 1;                    j = 0;                  i =

route[counter] - 1;                   counter++;


          }

    }


    // Update the ending city in array from city which was last visited

i = route[counter - 1] - 1;
```

```
    for (j = 0; j < tsp.size(); j++)

    {


       if ((i != j) && tsp[i][j] < min)
           {                   min

= tsp[i][j]; route[counter]

= j + 1;

          }


    }

    sum += min;


    // Started from the node where we finished as well. cout

<< ("Minimum Cost is : "); cout << (sum);

}
// Driver Code int main()

{


    // Input Matrix     vector<vector<int> > tsp = {

{ -1, 10, 15, 20 },

                                                              { 10, -1, 35, 25 },

                                                              { 15, 35, -1, 30 },

                                                             { 20, 25, 30, -1 } };
    findMinRoute(tsp);
  }
```

OUTPUT:

```
80
```

TIME COMPLEXITY:

Let's consider a greedy algorithm that starts at a given city and iteratively selects the closest unvisited city until all cities have been visited. The recurrence relation for this algorithm can be expressed as follows: T(n) = T(n-1) + O(n)

where T(n) represents the time complexity of the algorithm for a set of n cities, and O(n) represents the time required to find the closest unvisited city at each step.

Solving this recurrence relation using the substitution method, we can obtain:

T(n) = T(n-2) + O(n-1) + O(n) =

T(n-3) + O(n-2) + O(n-1) + O(n)

= ...

= T(1) + O(2) + ... + O(n-1) + O(n)

Simplifying this expression, we get:

T(n) = O(1 + 2 + ... + n)

= O(n^2)

Therefore, the time complexity of the greedy approach for the TSP is O(n^2), which is polynomial but not optimal. Although the greedy approach is simple and efficient for small instances of the TSP, it can lead to suboptimal solutions for larger instances.

Conclusion:

BEST APPROACH AMONG TWO:-

If the Travelling Salesman Problem instance is small or moderate in size, and finding the optimal solution is a priority, the dynamic approach may be the best choice. However, if computational efficiency is a higher priority and finding an approximate solution quickly is acceptable, “the greedy approach may be suitable”.

It's also worth mentioning that there are many other advanced techniques and algorithms specifically designed for solving TSP, such as genetic algorithms, ant colony optimization, simulated annealing, and others, which can often provide good solutions with a good trade-off between solution quality and computational time. The best approach for solving TSP ultimately depends on the specific problem requirements and constraints, and it may require experimentation and comparison to determine the most suitable approach for a particular use case.

REFERENCES:

- https://www.wikipedia.org/
- https://practice.geeksforgeeks.org/home
- https://www.tutorialspoint.com/index.htm
- https://www.javatpoint.com/