

TRAVELING SALESMAN PROBLEM

A COURSE PROJECT REPORT

By

SAI PRAJWAL KACHAM (RA2111028010166)

DARSH RAWAT(RA2111028010174)

DUBBA MANJUNATH REDDY(RA2111028010140)

Under the guidance of

Dr.Kishore Anthuvan Sahayaraj K

In partial fulfillment for the Course

of

**18CSC204J - Design and Analysis of
Algorithms**

in NWC



FACULTY OF ENGINEERING AND TECHNOLOGY SRM

INSTITUTE OF SCIENCE AND TECHNOLOGY

Kattankulathur, Chengalpattu District

April, 2023



SRM INSTITUTE OF SCIENCE AND TECHNOLOGY
KATTANKULATHUR – 603 203
BONAFIDE CERTIFICATE

This is to certify that **18CSC204J – DESIGN AND ANALYSIS OF ALGORITHMS Mini Project report** is the bonafide work of **Sai Prajwal Kacham(RA2111028010166), Darsh Rawat(RA2111028010174), Dubba Manjunath Reddy(RA2111028010140)** who undertook the task of completing the project within the allotted time.

SIGNATURE

Dr.Kishore Anthuvan Sahayaraj K
Assistant Professor
Department of Computing Technologies
SRM INSTITUTE OF SCIENCE AND
TECHNOLOGY

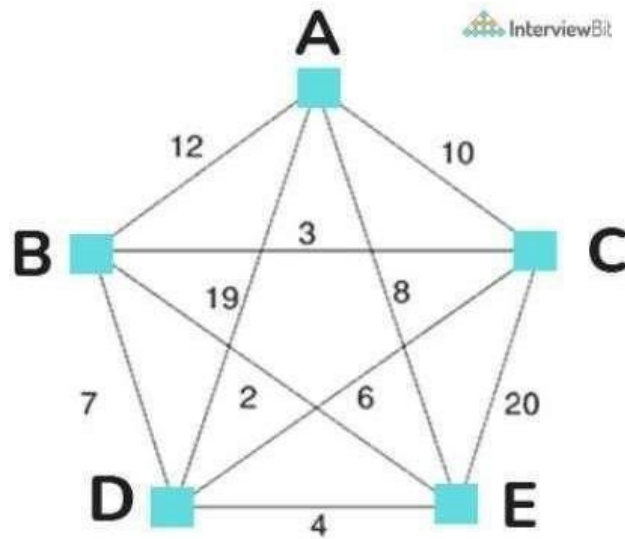
SIGNATURE

Dr.ANNAPURANI PANAIYAPPAN
Professor and Head
Department of Networking and Communications
SRM INSTITUTE OF SCIENCE AND
TECHNOLOGY

TABLE OF CONTENTS

1.	Cover Page	
2.	Certificate	
3.	Problem Statement	
4.	Table of Contents	
5.	Introduction	
6.	Dynamic Approach	
7.	Greedy Approach	
8.	Simple Approach	
11.	Justification	
12.	References	

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 the starting point.



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

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:

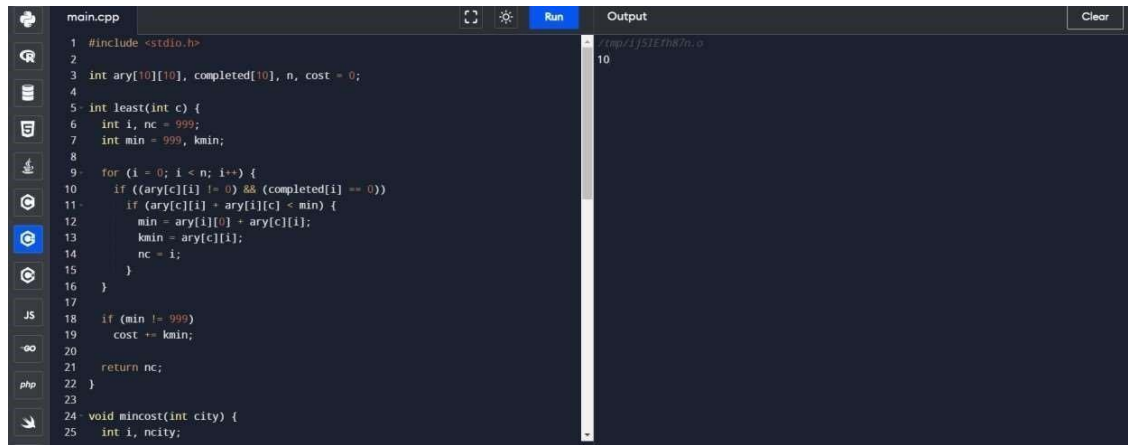
1. Define the number of vertices V in the graph and the starting vertex s .
2. Create a vector to hold all vertices except the starting vertex.
3. Initialize the minimum path min_path to the maximum integer value.
4. Generate all permutations of the remaining vertices using the `next_permutation` function 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];
        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:



```
main.cpp
1 #include <stdio.h>
2
3 int ary[10][10], completed[10], n, cost = 0;
4
5 int least(int c) {
6     int i, nc = 999;
7     int min = 999, kmin;
8
9     for (i = 0; i < n; i++) {
10         if ((ary[c][i] != 0) && (completed[i] == 0))
11             if (ary[c][i] + ary[i][c] < min) {
12                 min = ary[i][c] + ary[c][i];
13                 kmin = ary[c][i];
14                 nc = i;
15             }
16     }
17
18     if (min != 999)
19         cost += kmin;
20
21     return nc;
22 }
23
24 void mincost(int city) {
25     int i, ncity;
```

Output

10

TIME COMPLEXITY:

There are at most $O(2^n)$ subproblems, and each one takes linear time to solve. The total running time is, therefore, $O(n \cdot 2^n)$. The time complexity is much less than $O(n!)$ but still exponential.

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.

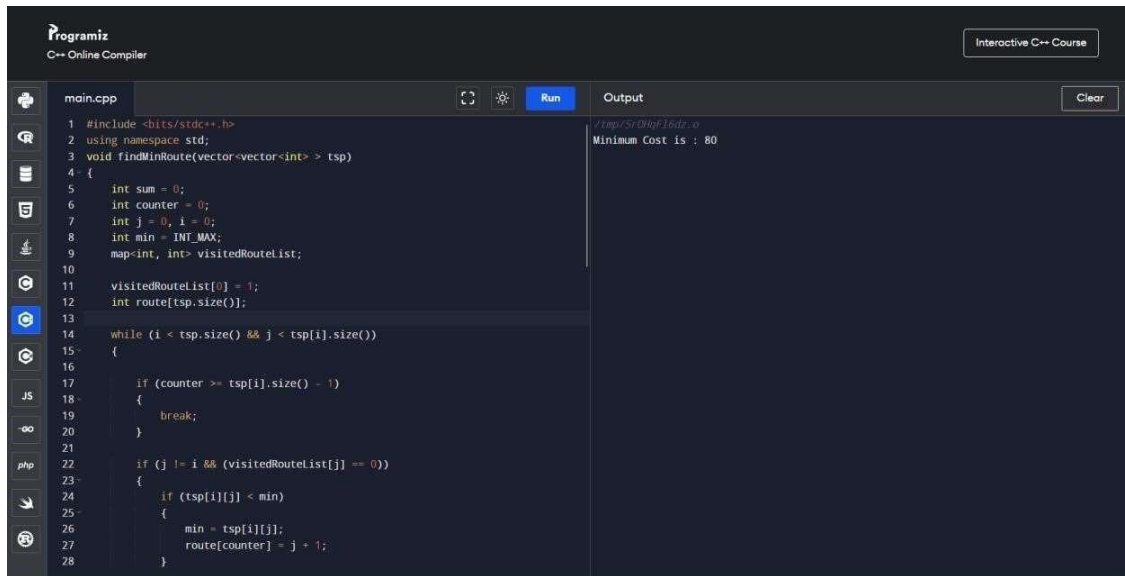
Approach: This problem can be solved using Greedy Approach. The steps are given below:

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;
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;
    visitedRouteList[0] = 1; int route[tsp.size()];
    tsp[][] while (i < tsp.size() && j < tsp[i].size())
    {
        if (counter >= tsp[i].size() - 1)
        {
            break;
        }
        if (j != i && (visitedRouteList[j] == 0))
        {
            if (tsp[i][j] < min)
            j++;
        };
        if (j == tsp[i].size())
        {
            sum += min;
            min = INT_MAX;
            visitedRouteList[route[counter] - 1]
            route[counter] - 1;
            counter++;
        }
    }
    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;
    cout << ("Minimum Cost is : ");
    cout << (sum);
}
int main()
{
    vector<vector<int>>> tsp = {
        { -1, 10, 15, 20 },
        { 10, -1, 35, 25 },
        { 15, 35, -1, 30 },
        { 20, 25, 30, -1 } };
    findMinRoute(tsp);
}
```

OUTPUT:



The screenshot shows the Programiz C++ Online Compiler interface. The code in `main.cpp` implements a Traveling Salesman Problem (TSP) solution. It includes headers, uses the `std` namespace, and defines a `findMinRoute` function that takes a vector of vectors representing the TSP graph. The function initializes variables for sum, counter, j, i, min, and a map for visited routes. It then iterates through the graph to find the minimum cost route. The output shows the minimum cost is 80.

```
1 #include <bits/stdc++.h>
2 using namespace std;
3 void findMinRoute(vector<vector<int> > tsp)
4 {
5     int sum = 0;
6     int counter = 0;
7     int j = 0, i = 0;
8     int min = INT_MAX;
9     map<int, int> visitedRouteList;
10
11     visitedRouteList[0] = 1;
12     int route[tsp.size()];
13
14     while (i < tsp.size() && j < tsp[i].size())
15     {
16
17         if (counter >= tsp[i].size() - 1)
18         {
19             break;
20         }
21
22         if (j != i && (visitedRouteList[j] == 0))
23         {
24             if (tsp[i][j] < min)
25             {
26                 min = tsp[i][j];
27                 route[counter] = j + 1;
28             }
29         }
30     }
```

Output: Minimum Cost is : 80

TIME COMPLEXITY:

The time complexity of the Travelling Salesman Approach is dependent on the specific algorithm and data structures used. In this code, the time complexity is $O(n^2 * \log n)$. It can be improved using data structures like priority queue and hash tables.

Travelling Salesman Problem Using Simple Approach

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 the starting point.

- > Consider city 1 as the starting and ending point. Since the route is cyclic, we can consider any point as a starting point.
- > Now, we will generate all possible permutations of cities which are $(n-1)!$.
- > Find the cost of each permutation and keep track of the minimum cost permutation.
- > Return the permutation with minimum cost.

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];

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 },

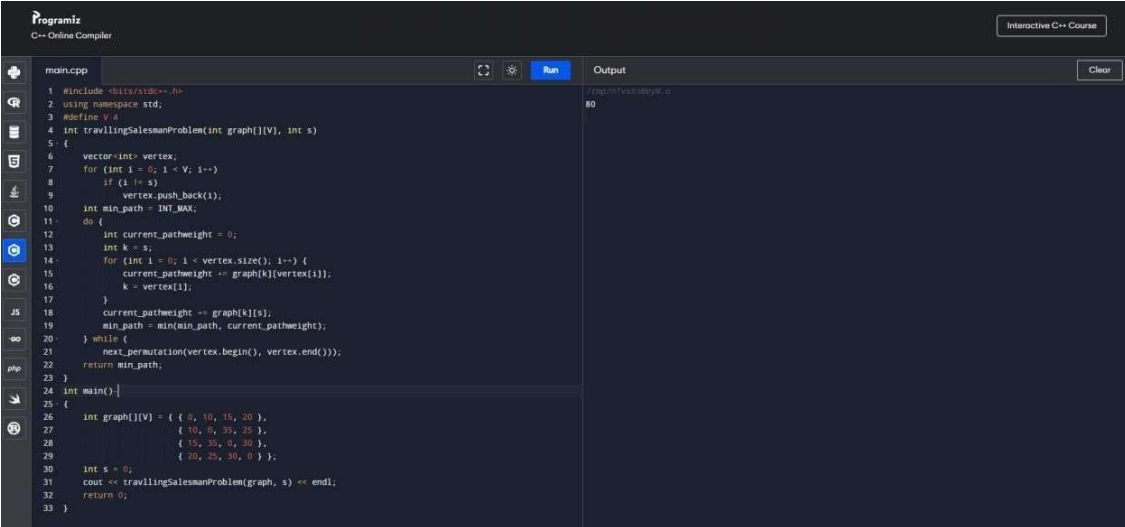
{ 20, 25, 30, 0 } };

int s = 0;

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

return 0;}
```

OUTPUT



The screenshot shows the Programiz C++ Online Compiler interface. The left sidebar contains icons for file management, search, and other IDE features. The main editor displays a C++ program for the Travelling Salesman Problem. The code defines a recursive function `travellingSalesmanProblem` that uses backtracking to find the minimum path weight. It includes a `main` function that initializes a graph and calls the solver. The right panel shows the output of the program, which is the number 80.

```
1 #include <bits/stdc++.h>
2 using namespace std;
3 #define V 4
4 int travellingSalesmanProblem(int graph[][V], int s)
5 {
6     vector<int> vertex;
7     for (int i = 0; i < V; i++)
8         if (i != s)
9             vertex.push_back(i);
10    int min_path = INT_MAX;
11    do {
12        int current_pathweight = 0;
13        int k = s;
14        for (int i = 0; i < vertex.size(); i++) {
15            current_pathweight += graph[k][vertex[i]];
16            k = vertex[i];
17        }
18        current_pathweight += graph[k][s];
19        min_path = min(min_path, current_pathweight);
20    } while (1);
21    next_permutation(vertex.begin(), vertex.end());
22    return min_path;
23 }
24 int main() {
25 {
26     int graph[][V] = { { 0, 10, 15, 20 },
27                         { 10, 0, 35, 25 },
28                         { 15, 35, 0, 30 },
29                         { 20, 25, 30, 0 } };
30     int s = 0;
31     cout << travellingSalesmanProblem(graph, s) << endl;
32     return 0;
33 }
```

Output: 80

Time complexity: $O(N!)$, Where N is the number of cities.

Space complexity: $O(1)$.

JUSTIFICATION :

BEST APPROACH AMONG THREE :-

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”. The simple approach is usually not practical for large-scale TSP instances due to its exponential computational complexity

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/>