Pune Institute of Computer Technology

Department of Computer Engineering
(2022- 2023)

“Different exact and approximation algorithms for Travelling-Sales-Person Problem”

Submitted to the

Savitribai Phule Pune University

In partial fulfilment for the award of the Degree of

Bachelor of Engineering

In

Computer Engineering

By

1) **Anuj Mahendra Mutha** **41443**

2) **Nidhi Patil** **41447**

3) **Amit Purohit** **41450**

Problem Statement:

Different exact and approximation algorithms for Travelling-Sales-Person Problem

Description:

1. **Travelling Sales Person Problem**

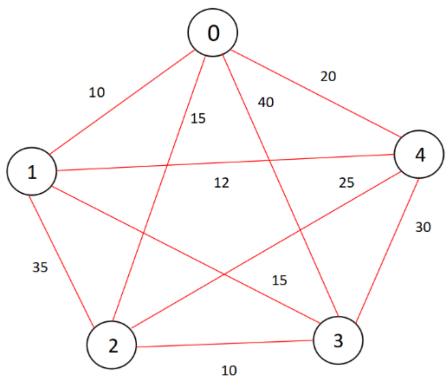
    - Travelling Salesman Problem is based on a real life scenario, where a salesman from a company has to start from his own city and visit all the assigned cities exactly once and return to his home till the end of the day.
    - The exact problem statement goes like this, **"Given a set of cities and distance between every pair of cities, the problem is to find the shortest possible route that visits every city exactly once and returns to the starting point."**
    - There are two important things to be cleared about in this problem statement,
        - **Visit every city exactly once**
        - **Cover the shortest path**

2. **Designing the code:**

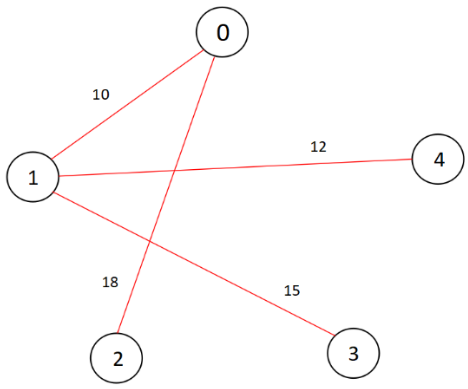
    - Step - 1 - Constructing The Minimum Spanning Tree
        - Creating a set mstSet that keeps track of vertices already included in MST.
        - Assigning a key value to all vertices in the input graph. Initialize all key values as INFINITE. Assign key value as 0 for the first vertex so that it is picked first.
        - [The Loop] While mstSet doesn't include all vertices
        - Pick a vertex u which is not there in mstSet and has minimum key value.(minimum_key())
        - Include u to mstSet.
        - Update key value of all adjacent vertices of u. To update the key values, iterate through all adjacent vertices. For every adjacent vertex v, if weight of edge u-v is less than the previous key value of v, update the key value as weight of u-v.
    - Step - 2 - Getting the preorder walk/ Defth first search walk:
        - Push the starting_vertex to the final_ans vector.
        - Checking up the visited node status for the same node.
        - Iterating over the adjacency matrix (depth finding) and adding all the child nodes to the final_ans.
        - Calling recursion to repeat the same.

3. **Example:**

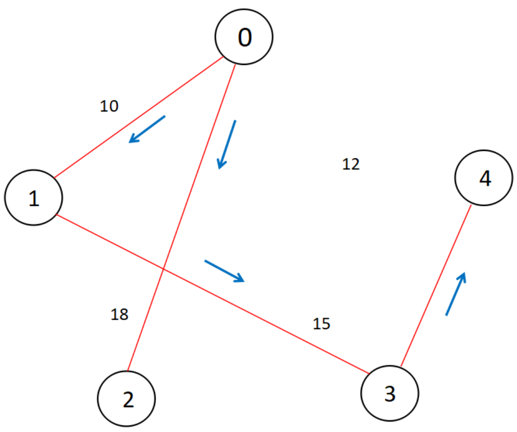
    - Let's have a look at the graph(adjacency matrix) given as input



- After performing step-1, we will get a Minimum spanning tree as below



- Performing DFS, we can get something like this



Code:

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

// Number of vertices in the graph
#define V 5
```

```
// Dynamic array to store the final answer
vector<int> final_ans;

int minimum_key(int key[], bool mstSet[])
{
    int min = INT_MAX, min_index;

    for (int v = 0; v < V; v++)
        if (mstSet[v] == false && key[v] < min)
            min = key[v], min_index = v;

    return min_index;
}

vector<vector<int>> MST(int parent[], int graph[V][V])
{
    vector<vector<int>> v;
    for (int i = 1; i < V; i++)
    {
        vector<int> p;
        p.push_back(parent[i]);
        p.push_back(i);
        v.push_back(p);
        p.clear();
    }
    return v;
}

// getting the Minimum Spanning Tree from the given graph
// using Prim's Algorithm
vector<vector<int>> primMST(int graph[V][V])
{
    int parent[V];
    int key[V];

    // to keep track of vertices already in MST
    bool mstSet[V];

    // initializing key value to INFINITE & false for all mstSet
    for (int i = 0; i < V; i++)
        key[i] = INT_MAX, mstSet[i] = false;

    // picking up the first vertex and assigning it to 0
    key[0] = 0;
    parent[0] = -1;

    // The Loop
```

```
    for (int count = 0; count < V - 1; count++)
    {
        // checking and updating values wrt minimum key
        int u = minimum_key(key, mstSet);
        mstSet[u] = true;
        for (int v = 0; v < V; v++)
            if (graph[u][v] && mstSet[v] == false && graph[u][v] < key[v])
                parent[v] = u, key[v] = graph[u][v];
    }
    vector<vector<int>> v;
    v = MST(parent, graph);
    return v;
}

// getting the preorder walk of the MST using DFS
void DFS(int** edges_list,int num_nodes,int starting_vertex,bool* visited_nodes)
{
    // adding the node to final answer
    final_ans.push_back(starting_vertex);

    // checking the visited status
    visited_nodes[starting_vertex] = true;

    // using a recursive call
    for(int i=0;i<num_nodes;i++)
    {
        if(i==starting_vertex)
        {
            continue;
        }
        if(edges_list[starting_vertex][i]==1)
        {
            if(visited_nodes[i])
            {
                continue;
            }
            DFS(edges_list,num_nodes,i,visited_nodes);
        }
    }
}
int main()
{
    // initial graph
    int graph[V][V] = { { 0, 10, 18, 40, 20 },
                        { 10, 0, 35, 15, 12 },
                        { 18, 35, 0, 25, 25 },
                        { 40, 15, 25, 0, 30 },
                        { 20, 13, 25, 30, 0 } };
```

```
    vector<vector<int>> v;

    // getting the output as MST
    v = primMST(graph);

    // creating a dynamic matrix
    int** edges_list = new int*[V];
    for(int i=0;i<V;i++)
    {
        edges_list[i] = new int[V];
        for(int j=0;j<V;j++)
        {
            edges_list[i][j] = 0;
        }
    }

    // setting up MST as adjacency matrix
    for(int i=0;i<v.size();i++)
    {
        int first_node = v[i][0];
        int second_node = v[i][1];
        edges_list[first_node][second_node] = 1;
        edges_list[second_node][first_node] = 1;
    }

    // a checker function for the DFS
    bool* visited_nodes = new bool[V];
    for(int i=0;i<V;i++)
    {
        bool visited_node;
        visited_nodes[i] = false;
    }

    //performing DFS
    DFS(edges_list,V,0,visited_nodes);

    // adding the source node to the path
    final_ans.push_back(final_ans[0]);

    // printing the path
    cout<<"Optmial Path to travel: ";
    for(int i=0;i<final_ans.size();i++)
    {
        cout << final_ans[i] << "-";
    }
    return 0;
}
```

OUTPUT***********************
Optimal Path to travel: 0-1-3-4-2-0-

Time Complexity:

- The time complexity for obtaining MST from the given graph is O(V^2) where V is the number of nodes.
- The time complexity for obtaining the DFS of the given graph is O(V+E) where V is the number of nodes and E is the number of edges.
- Hence the overall time complexity is O(V^2).

Space Complexity:

- The worst case space complexity for the same is O(V^2), as we are constructing a vector<vector<int>> data structure to store the final MST.
- The space complexity for the DFS is O(V).
- Hence space complexity of this algorithm is O(V^2).

Conclusion:

Hence studied travelling salesman problem and proposed a solution for it.