**A**

**MINI PROJECT REPORT ON**

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

SUBMITTED TO THE SAVITRIBAI PHULE PUNE UNIVERSITY, PUNE
FOR
**Design and Analysis of Algorithm**

**FINAL YEAR OF ENGINEERING (COMPUTER ENGINEERING)**

**SUBMITTED BY**

1. Pravin Jain Roll No: 72
2. Onkar Kulkarni Roll No: 74

**DEPARTMENT OF COMPUTER ENGINEERING**

MARATHA VIDYA PRASARAK SAMAJ'S KARMAVEER ADV. BABURAO GANPATRAO THAKARE COLLEGE OF ENGINEERING, NASHIK-13

**SAVITRIBAI PHULE PUNE UNIVERSITY 2022-23**

**Contents**

**Introduction:**

The traveling salesman problem (TSP) is an algorithmic problem tasked with finding the shortest route between a set of points and locations that must be visited. In the problem statement, the points are the cities a salesperson might visit. The salesman‘s goal is to keep both the travel costs and the distance traveled as low as possible.

Focused on optimization, TSP is often used in computer science to find the most efficient route for data to travel between various nodes. Applications include identifying network or hardware optimization methods.  It was first described by Irish mathematician W.R. Hamilton and British mathematician Thomas Kirkman in the 1800s through the creation of a game that was solvable by finding a Hamilton cycle, which is a non-overlapping path between all nodes.

**Problem Definition:**

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

**Objectives:**

- For TSPS (Traveling Salesman Problem), relatively efficient
- for a small number of nodes, TSPS can be solved by exhaustive search
- for a large number of nodes, TSPs are very computationally difficult to solve exponential time to convergence
- Performs better against other global optimization techniques such as neural net, genetic algorithms, simulated annealing

- Can be used in dynamic applications (adapts to changes such as new distances,etc.

**Scope:**

- With optimization in mind, the “traveling salesman problem”, frequently denoted by the initials TSP 1 , is a fundamental subject related to traveling and transportation, with several generalizations and with insertion in more complex situations, and also akin to others apparently unrelated, resoluble by the techniques used for the typical case. The TSP is known for the striking contrast between the simplicity of its formulation and the difficulty of its resolution, some even saying that it still does not have a solution. It is a so-called NP-hard problem (its difficulty increasing more than polynomially with its size). Anyway, something substantial can be presented about the problem.

**Description :**

1. Traveling Salesperson Problem

- Traveling 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 the 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:

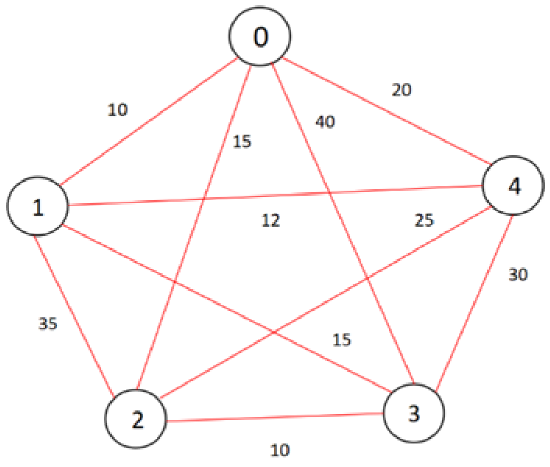
- Step - 1 - Constructing The Minimum Spanning Tree
- Creating a set mstSet that keeps track of vertices already included in MST.
- Assigning a key value to all vertices in the input graph. Initialize all key values as INFINITE. Assign the key value as 0 for the first vertex so that it is picked first.
- [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 the key value of all adjacent vertices of u. To update the key values, iterate through all adjacent vertices. For every adjacent vertex v, if the weight of edge u-v is less than the previous key value of v, update the key value as the weight of u-v.

Step - 2 - Getting the preorder walk/ Depth first search walk:

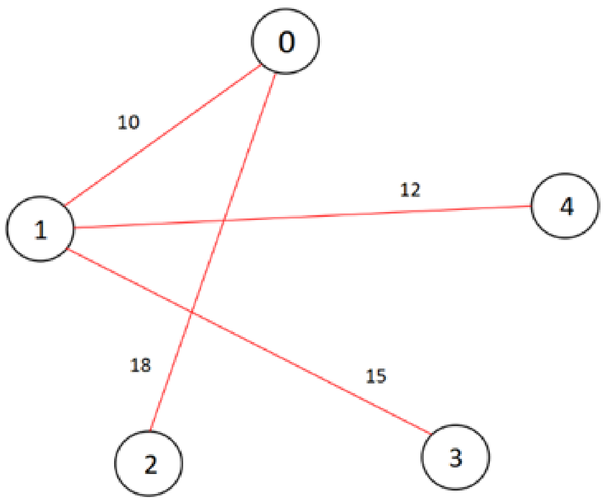
- Push the starting_vertex to the final_ans vector.
- Checking up on the visited node status for the same node.
- Iterating over the adjacency matrix (depth finding) and adding all the child nodes to the final_ans.
- Calling recursion to repeat the same.

3. Example:

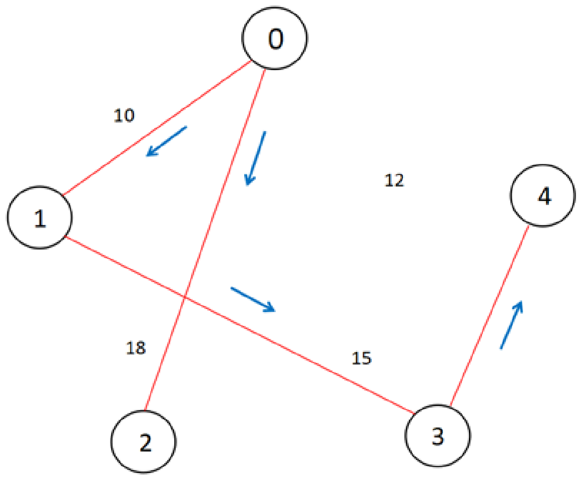
- Let's have a look at the graph(adjacency matrix) given as input
- After performing step-1, we will get a Minimum spanning tree as below
- Performing DFS, we can get something like this



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



- Performing DFS, we can get something like this



## Hardware and Software Requirements:

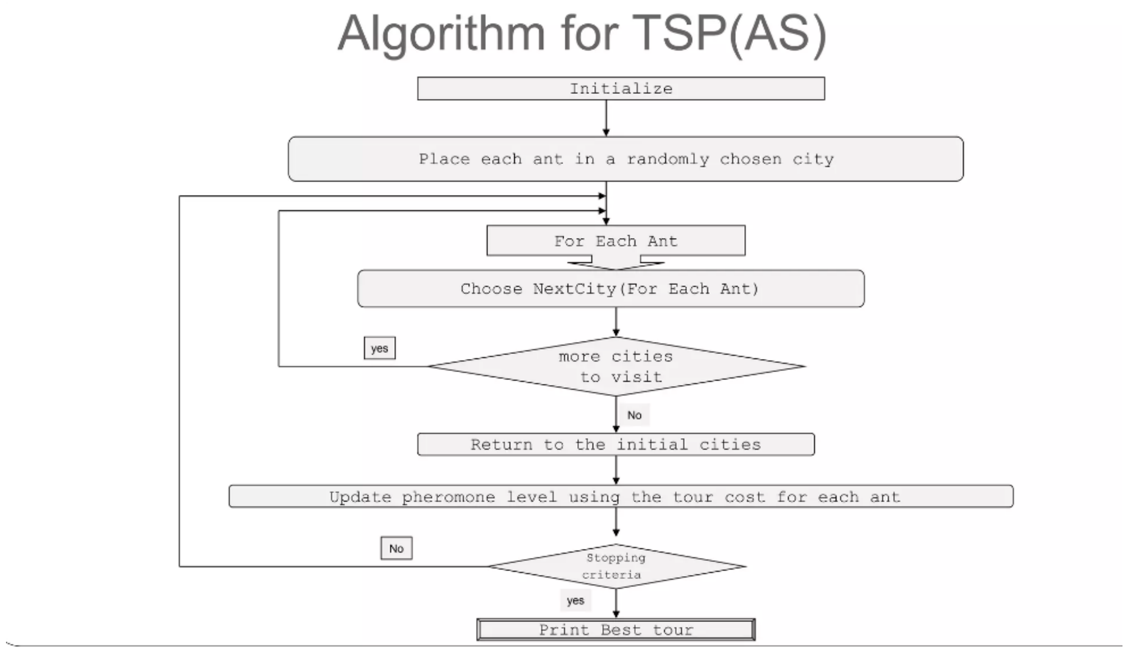
- **Hardware Requirements:**
    - GPU
    - 4GB RAM

- **Software Requirements:**
    - Windows 7+/ Ubuntu 16.04LTS
    - Visual Studio Code
    - C++

**Features:**

- Multi-Objective Evolutionary Algorithm
- Multi-Agent System
- Zero Suffix Method
- Optimization Algorithm

**System diagram:**


Algorithm for TSP(AS)
Initialize
Place each ant in a randomly chosen city
For Each Ant
Choose NextCity(For Each Ant)
yes
more cities to visit
No
Return to the initial cities
Update pheromone level using the tour cost for each ant
No
Stopping criteria
yes
Print Best tour

**Coding with Analyzed Output:**

```
 #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-
*********************************************************
```

## Conclusion:

- Efficient solutions found through the Traveling Salesman Problem help to optimize and minimize the cost of last-mile deliveries. These solutions help delivery businesses find a set of routes or paths to reduce delivery costs.

## References:

[1] Gerard Reinelt. The Traveling Salesman:
Computational Solutions for TSP Applications.
Springer-Verlag, 1994.

[2] D. B. Fogel, “An Evolutionary Approach to the
Traveling Salesman Problem”, Biol. Cybern. 60,139-
144 (1988)

[3] Kylie Bryant ,Arthur Benjamin, Advisor, “Genetic
Algorithms and the Traveling Salesman Problem”,
Department of Mathematics, December 2000