Travelling Salesman Problem

**Mudra Suthar - AU1741022**

**Rutvi Tilala - AU1741091**

Problem Statement:

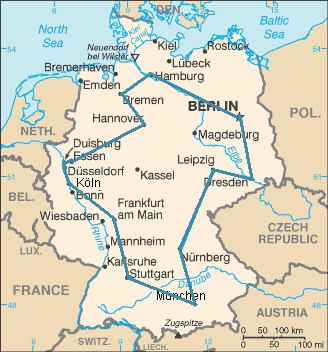
The travelling Salesman problem can be treated by 2 ways:

1.Given a set of cities and distance between every pair of cities, the problem is to find the shortest possible route ,asking the saleman the destination (the city which He/She wants to reach) and finding the minimum distance from source to that destination.(It  is not necessary to cover all the cities).

2.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 .

Application:

It is used in Google Map and real life scenarios.



List of data structures to be used :

1)2 D Array  as **adjacency matrix**

It is a square **matrix** used to represent a finite graph. The elements of the **matrix** indicate whether pairs of vertices are adjacent or not in the graph.

All the algorithms used in our codes uses adjacency matrix to store input taken from the user of the graph.

2)Stack

 Algorithm  **Dfs** uses **Stack** data structure. **DFS** is using backtracking. Backtracking can proceed only by **Stack**. The depth-first search uses a **Stack** to remember where it should go when it reaches a dead end.

3)Queue

Algorithm **BFS** uses **Queue** data structure,. Breadth first search does the search for nodes level-by-level, i.e. it searches the nodes with respect to their distance from the root. From this [simulation](https://www.cs.usfca.edu/~galles/visualization/BFS.html), you could see that BFS requires you to visit the child nodes in order their parents were discovered. Whenever we visit a node, we insert all the nodes into our data structure. If you use a queue data structure, it is guaranteed that, you pop the nodes in order their parents were discovered.

4)Priority Queue in Dijkstra algorithm - to perform 2 main operations : Extract min  and increase Key

Project brief

Travelling Salesman Problem can be solved by many algorithms depending on the problem statement given. Therefore, the purpose of this project is to explore which is the best shortest path algorithm by comparing the two types of algorithms. Such that it can be used to solve the problem path, search to analyze their efficiency in an environment based on two dimensional matrix which is best because of lookup time T.

Algorithms ,  pseudocodes , flowchart

PSEUDOCODES

1. Kruskal pseudocode

G –input graph in the form of adjacency matrix

KRUSKAL(G):

1 A = infinity

2 **for each** v ∈ G.V:

3 MAKE-SET(v)

4 **for each** (u, v) in G.E ordered by weight(u, v), increasing:

5 **if** FIND-SET(u) ≠ FIND-SET(v):

6 A = A ∪ {(u, v)}

7 UNION(u, v)

8 **return** A

A disjoint set is a data structure which keeps track of all elements that are separated by a number of disjoint (not connected) subsets. With the help of disjoints sets, you can keep a track of the existence of elements in a particular group.

**Find()** – It helps determine which subset a particular element belongs to.

It also helps determine if the element is in more than one subset.

**Union()** – It helps check whether a graph is cyclic or not. And helps connect or join two subsets

1. **DFS pseudocode**

DFS-iterative (G, s)://Where G is graph and s is source vertex

let S be stack

S.push( s )//Inserting s in stack

mark s as visited.

while( Sisnot empty):

//Pop a vertex from stack to visit next

v = S.top()

S.pop()

//Push all the neighbors of v in stack that are not visited

for all neighbors w of v inGraph G:

if w isnotvisited :

S.push( w )

mark w as visited

DFS-recursive(G, s):

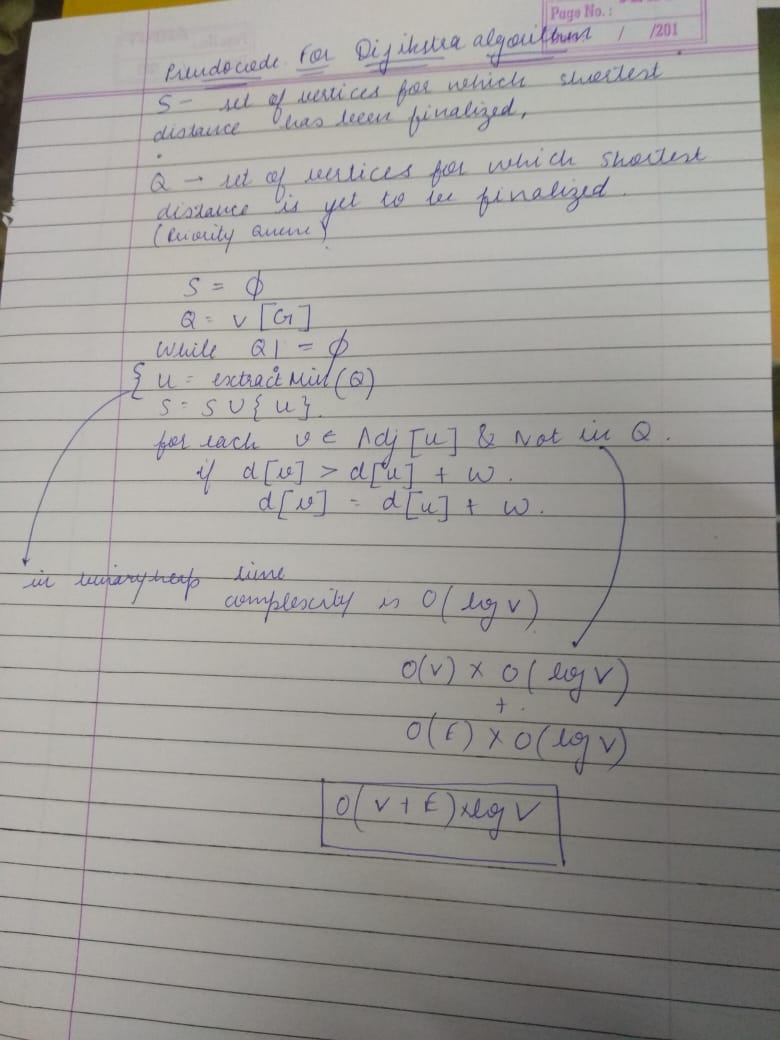
mark s as visited

for all neighbors w of s inGraph G:

if w isnot visited:

DFS-recursive(G, w)

1. **Dijkstra Pseudocode**



ALGORITHMS

**1) Dijkstra Algorithm**

The problem of finding shortest paths from a source vertex v to all other vertices in the graph. (In our Project We have taken our source vertex to be 1 ).

1. Create a set sptSet (shortest path tree set) that keeps track of vertices included in shortest path tree, i.e., whose minimum distance from source is calculated and finalized. Initially, this set is empty.  
   **2)** Assign a distance value to all vertices in the input graph. Initialize all distance values as INFINITE. Assign distance value as 0 for the source vertex so that it is picked first.  
   **3)** While sptSet doesn’t include all vertices  
   .
2. **a)** Pick a vertex u which is not there in sptSet and has minimum distance value.

.**b)** Include u to sptSet.

**c)** Update distance value of all adjacent vertices of u. To update the distance values, iterate through all adjacent vertices. For every adjacent vertex v, if sum of distance value of u (from source) and weight of edge u-v, is less than the distance value of v, then update the distance value of v.

Flowchart is as under: 

**2) Kruskal’s algorithm**

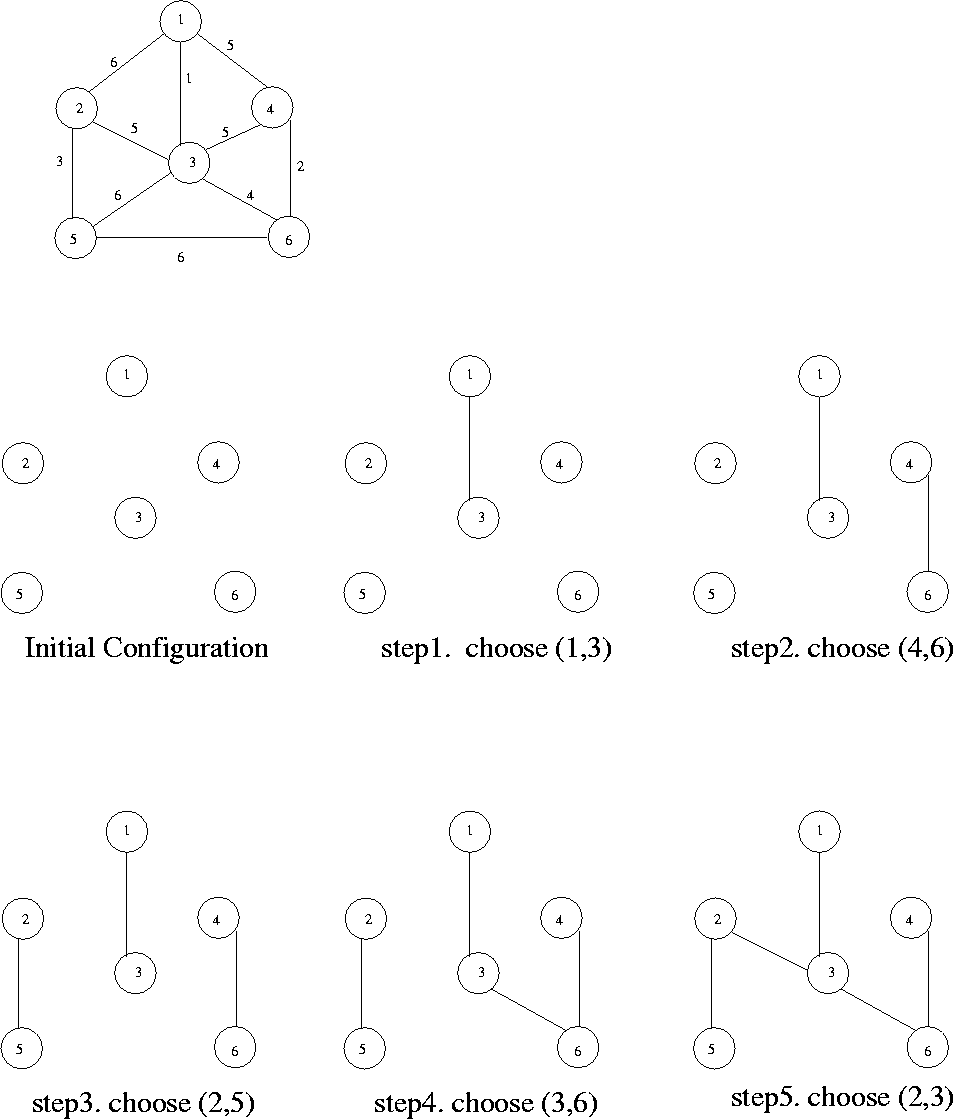
**1.Sort all the edges in non decreasing order of their weight.**

**2.Pick the smallest edge, Check if it forms a cycle with the spanning tree formed so far . If cycle is not formed , include this edge ,Else discard it.**

**3.Repeat step #2 until there are (V-1) edges in the spanning tree.**

Flowchart is as under:

|  |
| --- |
| **Figure 8.13:** An illustration of Kruskal's algorithm |
|  |



3) DFS

A standard DFS implementation puts each vertex of the graph into one of two categories:

1. Visited
2. Not Visited

The purpose of the algorithm is to mark each vertex as visited while avoiding cycles.

The DFS algorithm works as follows:

1. Start by putting any one of the graph's vertices on top of a stack.
2. Take the top item of the stack and add it to the visited list.
3. Create a list of that vertex's adjacent nodes. Add the ones which aren't in the visited list to the top of stack.
4. Keep repeating steps 2 and 3 until the stack is empty.

Flowchart is as under:

1. 

FileName of Each program:

There are in total 4 files in java-

1)ShortestPath 2)CreateGraph3)Krushkal 4)DFS

Source code for Shortest Path – codes are attached .

Input data: Source of this program is taken as 1 (by default.)

After that input is taken at runtime from the user :

User is asked to enter total no of nodes(cities ) which is within his/her range to travel.

After that user is asked to enter each origin node and destination node of the graph and length that is weight of this entered edge.

Output data:

If the user selects method 1 for implementation that is (Dijkstramethod ) then

This code finds shortest distances from source to all vertices. If we are interested only in shortest distance from the source to a single target, we can break the for the loop when the picked minimum distance vertex is equal to target (Step 3.a of the algorithm).

Time Complexity of the implementation is O(V^2).

* If the input [graph is represented using adjacency list](https://www.geeksforgeeks.org/graph-and-its-representations/), it can be reduced to O

(E log V) with the help of binary heap.This code is for undirected graph, same Dijkstra function can be used for directed graphs also.

* If user selects method 2 for implementation which is Krushkal method thenthe **output of Kruskal's algorithm** is a spanning tree. It gives output of all the edges andrespective minimum cost corresponding to each edge. .At the end it gives overall minimum cost.
* If the user selects method 3 (that is DFS )for implementation then DFS is a Graph traversal technique which takes the graph and a starting vertex(random) as input and gives a sequence of vertices as output. The sequence contains those vertices which are reachable from the starting vertex. i.e we are finding whether any vertex is reachable from any other vertex in a graph or not?

Algorithm for each logical process

* Insertion:

1. Inserting Node

(N+1)th node is added to N already existing nodes and corresponding that node adjacent matrix is updated.

1. Inserting Edge

New edge (weight) is added corresponding to newly added (N+1)th node.

* Deletion

1. Deleting Node

In this part of the process, we check whether existing number of nodes are zero or not , if nodes are zero we return output as “empty graph” else we delete Nth node from the last and so we are left with N-1 nodes.

1. Deleting Edge

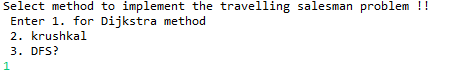
If edge exist then we update the edge to zero by taking location of required edge to be deleted.

* Display

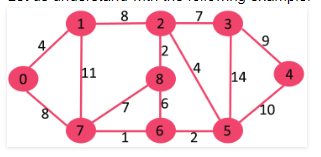
In the display part, input-graph is generated as adjacency matrix and user is will obtain outputfrom one of the methods among Dijkstra, Krushkal and DFS as per choice entered by user at the starting of the program.

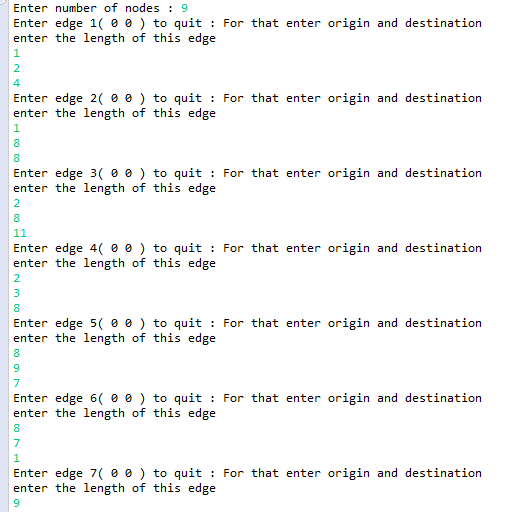
**Test result in form of Snapshots(PROCESSING)**

Selecting method

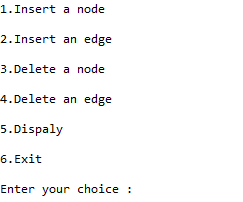


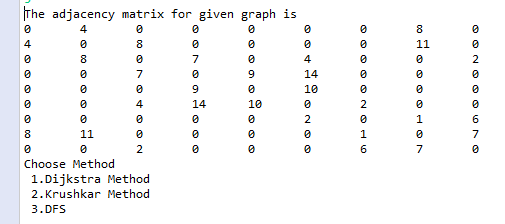
Taking input for Graph



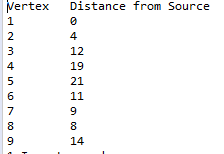


operations

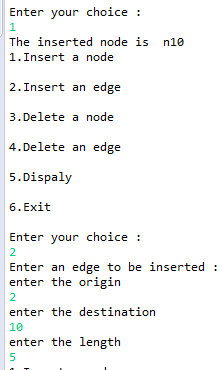


Displaying input in form of adjacent matrix

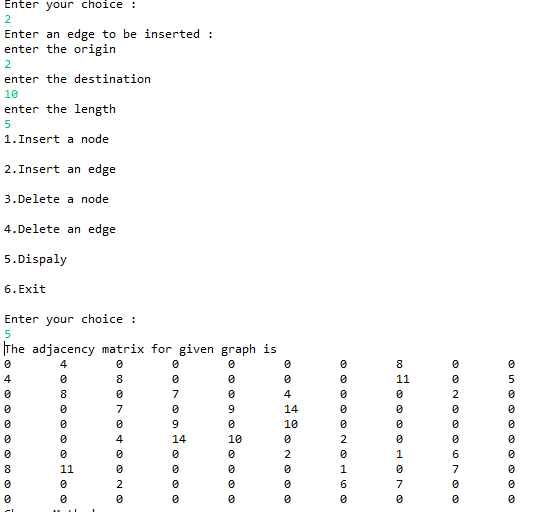
Dijkshtra implementation



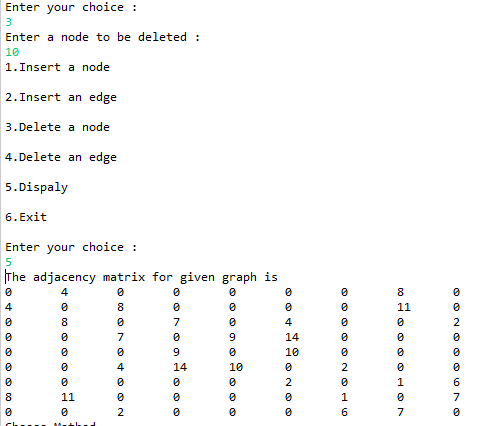
1.inserting a node



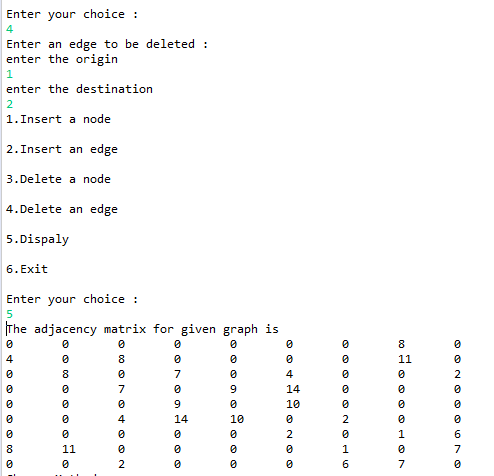
**2.Inserting a edge and displaying modified graph**



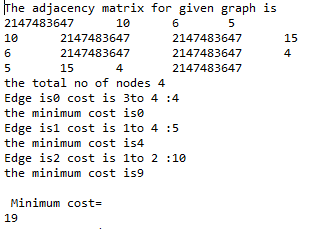
**3.Deleting node**



**4.Deleting Edge**



**Krushkal’s implementation**



Intermediate DS, Inter-linkages:

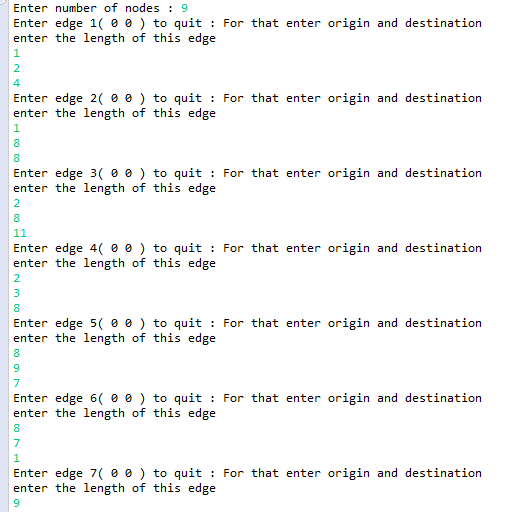
Stack : Algorithm  **Dfs** uses **Stack** data structure. **DFS** is using backtracking. Backtracking can proceed only by **Stack**. The depth-first search uses a **Stack** to remember where it should go when it reaches a dead end.

Priority Queue in Dijkstra algorithm - to perform 2 main operations : Extract min  and increase Key

List of Queries / Reports generated:

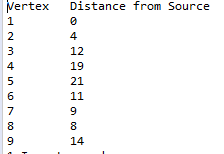
Problem statement(Input given)

Graph

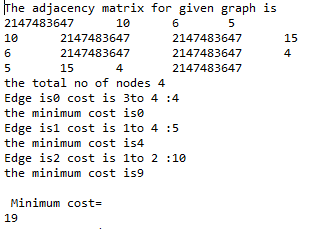


FINAL OUTPUT

Through dijiksta::



2)Through Krushkal



3)Through DFS

CONCLUSION

REFERENCES

https://www.geeksforgeeks.org/kruskals-algorithm-simple-implementation-for-adjacency-matrix/

<https://www.geeksforgeeks.org/dijkstras-shortest-path-algorithm-greedy-algo-7/>

https://www.programiz.com/dsa/graph-dfs