ANALYSIS OF ALGORITHM

**# Project Report**

**Algorithm:**

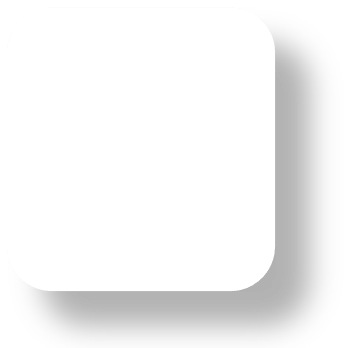
Dijkstra’s Algorithm for finding shortest path between source to all other vertexes in graph(destination).

**Group Members:**

Muhammad Zubair Khan NIM-BSCS-2021-36

Alina Khan NIM-BSCS-2021-42

Namal University Mianwali.



**Introduction:**

* Dijkstra’s Algorithm was developed by Dutch computer scientist Edsger W. Dijkstra’s is a well-known algorithm used to find shortest path between nodes in a weighted graph. It is efficient in finding single shortest path from one vertex (source) to all other ertexes in the graph(destination).
* This algorithm follows a greedy approach makes suitable(shortest) choice at every point and at each step it updates the distances of its adjacent(neighboring) vertexes.
* Dijkstra’s Algorithm uses priority Queue or min-heap as continually updates shortest path for each node which gives optimal path between all other nodes in a graph. It is proved to be very crucial and efficient in route planning, network optimization and resources allocation.
* Overall, it provides best and efficient solution to reach all vertexes in the graph and find shortest distance from other nodes in weighted graph.

**Aim of Algorithm:**

The primary goal of Dijkstra algorithm is to find shortest distance from one vertex to all other vertexes in graph. As in our daily life we have to deal with google map for reaching some specific location and for checking distances from that specific location. We decided that why should we not have some algorithm whose implementation can give us shortest path among one city (source) to all other cities of Pakistan. We took 28 major cities of Pakistan and linked these cities from different paths and picked up their actual distance (one city to other). Then we implemented Dijkstra’s Algorithm to give us easiness in finding that shortest path. Our primary goal was to provide passengers a more reliable and efficient method to finding and choosing shortest routes to reach some specific city from different cities, enable them to make informal decisions when it comes to travelling to any city because they can decide best routes which help in minimizing travelling time, fuel consumption and reduces overall travelling costs. By minimizing these especially travelling time they can serve their important time for some other better purposes.

**Why Dijkstra:**

* While selecting our algorithm it was very tough for us to decide which algorithm I should choose and which I shouldn’t. then there emerged a question in our mind why should I choose one algorithm and why should I don’t choose others. These questions created a lot of thinking for both of us as it was not an easy decision to favor one algorithm and other doesn’t. First it was our a little bit plan that we choose any sorting algorithm which we have not done till know. But later it was decided between us than go into other algorithm as most people knows about sorting algorithm, we wanted to choose some separate one. Then we reached at the conclusion to go to one the algorithm from TICK-TAC-TOE or DIJKSTRA ALGORITHM. We decide to perform implementation of both algorithm and present presentation of one. We search about DIJKSTRA ALGORITHM and came to know that it is famous algorithm for finding shortest path using greedy approach. We discussed with each other and one of our senior that which type of data we choose to finding shortest path using DIJKSTRA ALGORITHM.
* We reached at the conclusion to go for the implementation of DIJKSTRA ALGORITHM. As it is the shortest path Algorithm so we decided to apply it to one of the real world example so that others can took some advantage from its implementation. As in our life we often need to check the map to check distances between cities of every country if we wanted to travel. It was our wish to apply this algorithm for finding;
* The shortest path: Between different cities of Pakistan after taking actual distance of different cities and find shortest route so that not only for the passengers but other peoples can also take advantage from it.
* That’s why we choose that algorithm to calculate shortest distance if any one plans to go from one city(source) to another cities(destination) of Pakistan because this shortest path provide flexibility to passengers to choose which path, definitely they will prefer shortest path to save their time and minimum fuel costs and other expenditures while travelling.
* Then here comes a question how we succeed in doing this. We took almost 28 cities of Pakistan and asked from user to select from where he wants to travel (source) to reach(destination). First, we supposed maximum distance of all the cities which is unable to reach and where we are (source) as zero. Then we apply DIJKSTRA ALGORITHM and choose minimum distance at every point by following greedy approach or you can say by maintain priority Queue and compare their distance. After this process is finished, we succeed in finding a shortest distance between different cities of Pakistan from one city to all other cities.

**Formal Definition of Algorithm:**

* Dijkstra Algorithm is the graph algorithm which is used for finding the path between one node to all other nodes in weighted graphs non-negative weights. It is a single shortest path algorithm which iteratively explores the graph and finds shortest or minimum distance at each node.
* Formally, it can be expressed as
  + Graph is given directed or undirected having G= (V, E) where V is vertex and E is edges between the vertex with non-negative weights and having sources node which computes shortest distance by following Dijkstra algorithm.
* The algorithm can maintain empty array or dictionary, by maintaining dictionary we can see nodes and their distance too. Initially all the distance of vertexes and supposed to be infinity expect source vertex whose distance is supposed 0 at initial stage. An array is also used to maintain the visited vertexes. And a queue was used which initially stores all the vertexes in a graph.
* At each iteration algorithm finds nodes having shortest distance and using relaxation method of Dijkstra algorithm it updates the distance of its adjacent vertex, this process continuous till we visit all vertexes in graph and Queue becomes empty, after that it will return nodes with shortest distances from source node.

**Graph theory Basics:**

Fundamental graph theory concepts are used in graph making

List of graph theory Fundamental concepts

|  |  |
| --- | --- |
| **Graph theory component** | **Description/purpose** |
| vertex | It is used for representation of vertexes in graph. |
| Edges | Edges are used to form connection between two vertexes in a graph. |
| Directed/Undirected | A concept whether graph is directed or not undirected is used which shows whether the connections are bi-directional or unidirectional between two vertexes. |
| Weighted/Unweighted | A concept of Weighted/Unweighted is used which represents weights or costs between two vertexes, in this algorithm for making graph it was taken as positive because Dijkstra works only for it. |

*Table 1 for graph theory purposes*

**Experimental Set up:**

* While implementing we have to deal with different functions and libraries/Packages but we have not used any class in our implementation. Here is a purpose of functions and libraries /Packages that are used in implementation.

**Functions used:**

Functions that are used during implementation and their purposes is mention in following.

|  |  |
| --- | --- |
| **Function Name** | **Description/Purpose** |
| Declare Distance | A function of name Declare distance was used while implementing algorithm. The purpose of this graph was to declare distance to source as zero (0) and all other vertex as infinity (∞) at initial stage. |
| Dijkstra Algorithm | A function named Dijkstra algorithm was used which first calculates smallest path(routes) and then it will follow Dijkstra approach of relaxation, so it does relaxation after finding shortest path for each vertex in a graph.  This function returns shortest distances of all other nodes from source node. |
| Visualize Given Graph | A function name as Visualize Given Graph is used in the program. The main aim of using this function is to show the complete visualization of graph using graphical user interface. So this function shows graphical picture of given graph. |
| Plot shortest path graph | A function of name Plot shortest path graph is used in the program which we used to plot the shortest distance with his respective vertex. This visualizes the shortest path with vertexes graphical user interface. |
| Obtain graph | A function name as Obtain graph is used in the program which has its own functionalities. This is basically used to get the entry of source node and then it calls Dijkstra algorithm to check whether source is present in the graph or not. If source is present Then it will show visited order of vertexes with label name “Visited Order” and at last it will plot shortest distance graph with each vertex and its respective distance. |

*Table 1.1 for functions used/their purposes*

**Packages used:**

Libraries/Packages that are used during implementation and their purposes is mention in following.

|  |  |
| --- | --- |
| **Libraries/Packages** | **Purpose/significance/description** |
| Networkx | This is a famous library in python which is basically used for plotting the different connections/networks using edges and nodes.it is a powerful library in python for creating, manipulating, studying different structures or complex structure making it ideals to visualize networks and their relationships. |
| matplotlib. pyplot | It visualizes different informative networks. It has extensive functionalities such as for making graphs, histogram, scatter plot enhancing the data visualization capabilities of network. |
| Tkinter | A popular library in python which is basically used in giving graphical user interface to code. With the help of this library, we can give GUI to all the console base programs. |

*Table 1.2 libraries and their purposes*

**Pseudocode:**

There are different functions used but I will present Pseudocode of two maintain functions that are used in implementation.

**Input:** A Graph and a Source is given as input to function.

**Output:** Function will calculate shortest distance and return it.

Visited-vertex-in-order Empty

Distance Empty

Declare-distance (graph, source)

For each vertex in graph do

If vertex is source do

Distance[vertex] 0

Else all other vertex in graph do

Distance[vertex] ∞

return distance (This distance if declared infinity distance)

Dijkstra-Algorithm (graph, source)

Queue all vertexes in graph

While Queue is not None do

min-distance ∞

u None

for each vertex in Queue do

if distance[vertex]< min-distance do

min-distancedistance[vertex]

uvertex

for all adjacent and weight in graph[u] do

alter-distance distance[u]+ weight

if alter-distance <distance[adjacent] do

distance[adjacent] alter\_distance

Add u into visited vertex in order

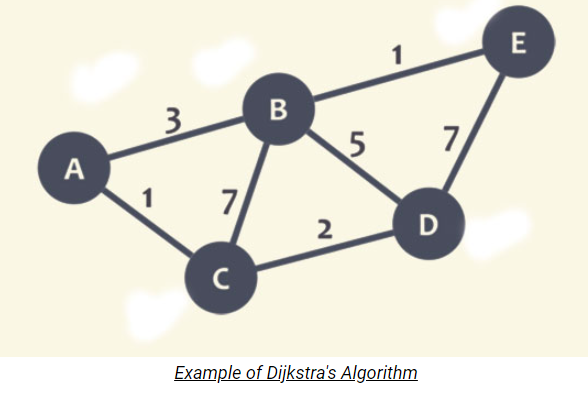
Remove u from Queue

return distance (Actual shortest distance)

**Algorithm Design:**

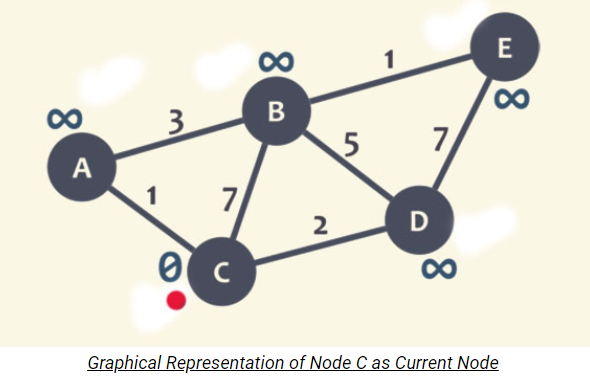
As greedy algorithms are the algorithms which works step by step at every event, they work on a step which is shortest from them. If we see Dijkstra algorithm it follows greedy approach and then calculates shortest path, so it is also a greedy algorithm. In Dijkstra algorithm we first create a graph with vertexes and then their adjacent vertexes respectively. There is some weight is also written while making graph from vertexes to their adjacent. The whole graph is made on the basis of vertexes and their adjacent node with weight. But in Dijkstra algorithm while writing weights the most caring thing is that it doesn’t work on negative weights so it is necessary to write weights of edges positive. This whole process is for overall graph. After graph is made, we move towards to algorithm.

* Firstly, we create empty dictionary for distance as at initial stage we don’t know any distance but we are working here with dictionary because at last we will keep record of each vertex (country) with corresponding distances from sources vertex, so this was the main purpose of this dictionary creation. And also, a list of visited vertex in their order is also maintained as empty because at initial stage we don’t have any vertex which is visited so it is maintained as empty initially. We will keep vertexes in visited and calculated shortest distance in these twos respectively at every step.

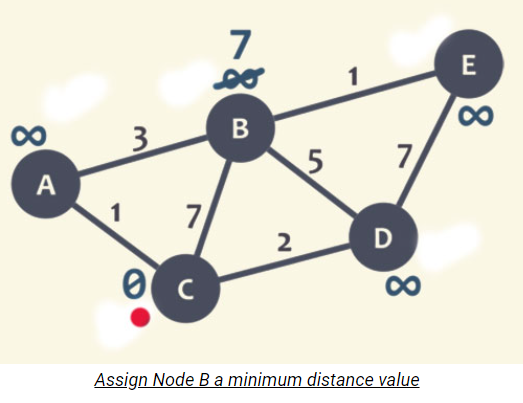


*Figure 0.1 simple graph*

* Next a function with name declare distance is defined, we used this function for a very important purpose. The main purpose of this function is to assign some temporal distances to all the vertexes even we don’t have any distance but this function declares distance. It checks all the vertexes in the graph and see which is source and destinations. It first assigns zero (0) distance to source vertex if it is preset in the graph. And then it checks all other vertexes in the graph and declare distances to these vertexes. It declares all the remaining vertexes distances except source vertexes as infinity (∞). So, at this stage we have 0 distance of source vertex and infinity (∞) distances of all other vertexes.

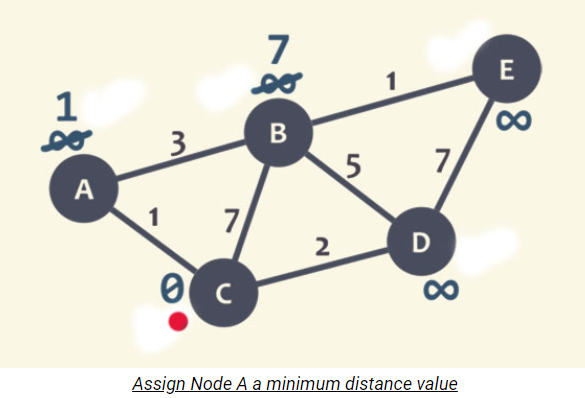
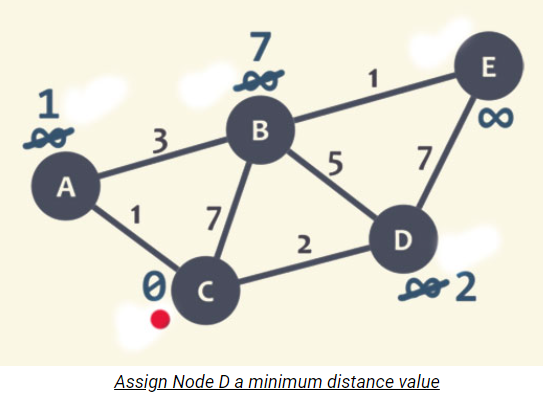


*Figure 0.2 graph with infinite distance*

* After doing this we have a function named as “Dijkstra Algorithm” which is mainly used for algorithm implementation after declaring distances to all vertexes. In this function first we keep all the vertexes that are the part of graph in the Queue so our all the vertexes are in the Queue at this stage. Next it iterates through each vertex in the graph. There is variable named as min-distance which is initially supposed as infinity, this is done to compare distance of all other vertexes in the graph and a variable u is placed which is initially taken as None but it basically keeps the exactly one visited node at once which is used for checking adjacent later on. Next, we iterate through each vertex in the Queue and compare their distances with min-distance which was infinity (∞) at initial stage and it updates min-distance if vertex distance is less than min-distance so it gets there, minimum distance vertex and then it keeps that minimum distance vertex in the u. This minimum distance vertex which was stored in the u than used to proceed process.
  + - * 

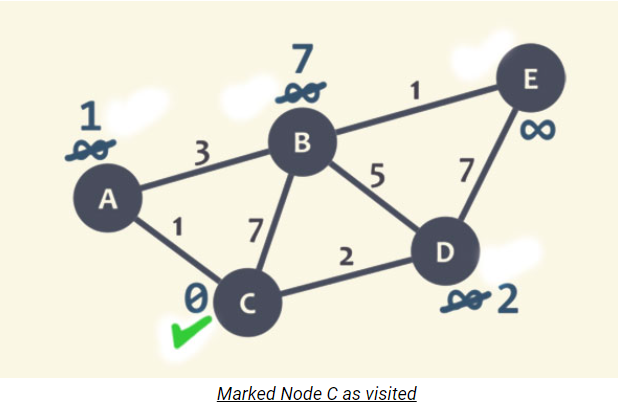
*Figure 0.3 graph for Adjacent update distances*

* In next step we check all the adjacent vertexes of minimum-distance vertex(u). we see adjacent vertexes and their weights and we add the distance of minimum-distance vertex with weights. After doing step their comes a relaxation in which we compare the distance of each adjacent vertex with the sum of its root vertex and weights. Here we check if distance of sum of root vertex and weights is less than each its adjacent vertex, if it satisfy we updates it. This is called relaxationin which every adjacent vertex is compared with the sum of its root vertex and weights. We keep that minimum distance vertex in an array to keep record visited order vertex and we also removes this minimum distance vertex from the Queue do that as it is visited so we don’t need to care of it in Queue. After that there comes second vertex from Queue and this process continues at every iteration element next to the Queue comes. This process is same for all the vertex in the graph, finds minimum distance vertex and updates it distance, keep record of visited nodes and remove it from Queue. This whole process continues till the Queue becomes empty, at this stage almost every vertex got visited. At the end, when Queue becomes None or empty it returns one shortest distance from one source vertex to all other vertexes in the graph (destination). This is the main step by step of execution of Dijkstra algorithm.

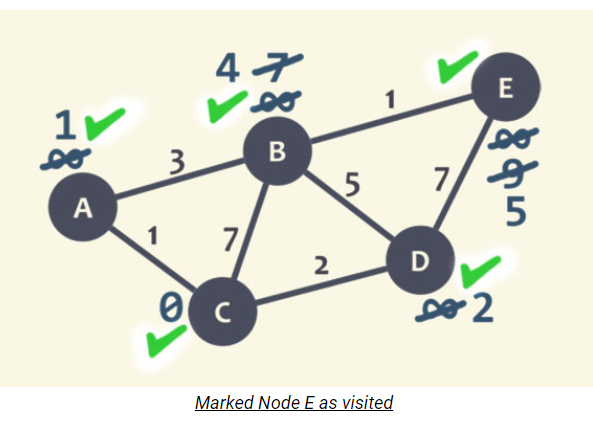
*Figure 0.4 graph for Adjacent update distances Figure 0.5 graph for Adjacent update distances*

Visited nodes are



*Figure 0.6 visited vertex*

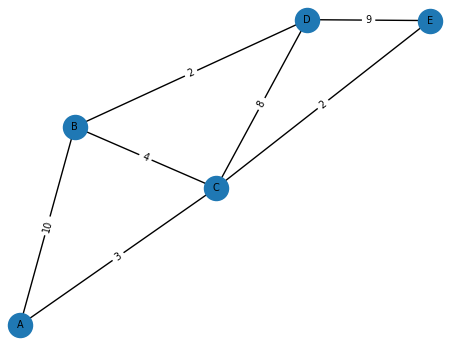
At the end we receive following graph after following all above steps



*Figure 0.7 all visited vertexes*

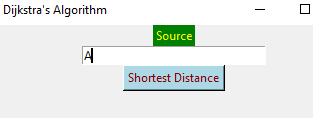
* As I added GUI also in it which further extends its design. A function visualize graph has been used which basically plots the graph with graphical user interface. It basically first adds all the nodes and then using nested loops, on graph (outer) and neighbors (inner) it adds edge of one node with its corresponding neighbors with weighted. This process continues for all the vertexes and it maintain edges and weight between every two nodes and then it plots final graph and plt.show it shows its graph.

For example for simple graph



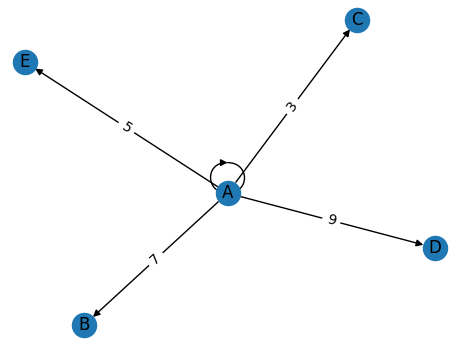
*Figure 0.8 simple graph*

* Obtain graph function is also used, it basically accepts source node entry and check whether it is in the graph or not and then shows the final results and plots final graph.



*Figure 0.9 source vertex*

* Function which is used in the code to plot graph of shortest distance from source vertex first declares source node as global to be used in any other function. It first forms the lists of all the vertexes and then add them as vertexes for graph. It iterates each item in that list and start adding edges between one source node to all other vertexes with distances. So, at the end all the nodes are represented in the graph in which all the vertexes are connected one source node, telling distance from source to other nodes in the graph.



*Figure 0.11 after calculating shortest distance*

**Algorithm analysis:**

Here is the what we analyze after implementing Dijkstra algorithm to our graph. Following are mentioned what is better and less better according to our analysis.

Time Complexity:

Step by step

* Time complexity of the piece declare-distance of code=O(V)
* Time complexity of the piece to find shortest vertex (greedy), of code=O(V(log(V))
* Time complexity of the piece for relaxation of code= O(E(log(V))
* Time complexity of the piece to maintain min-heap of code=O(V) **.:** This time complexity of for maintaining priority queue or min-heap**.**

Total time complexity:

Following is the time complexity of main parts of code

Time Complexity = declare-infinity-distance time +maintain min-heap time +trace-min-distance-vertex time +relaxation time

Time Complexity **=O(V)+O(V)**+**O (V (log(V))** + **O (E (log(V))**

As we consider biggest term as time complexity so here it is clear that biggest term is 0E(log(V)) instead of OV (log(V)) because every graph has more edges than vertexes as there could be multiple connections for single vertex.

So, Time Complexity of Dijkstra algorithm is;

|  |
| --- |
| Time Complexity =O(E(log(V)) |

Space Complexity:

* The space complexity of Dijkstra algorithm depends upon its implementation as it can be implemented in different ways. We implemented it with the most common method with priority queue or min-heap. Its space complexity with priority queue or min-heap is 0(V+E) where V is number of vertexes and E is a number of edges.
* While implementing Dijkstra algorithm we used priority queue or min-heap to store all the vertexes according to smallest distance order while used distance dictionary to show vertexes with their shortest path. As the number of vertexes increase, we have to increase the size of priority queue or min-heap to store all the vertexes which contribute to its space complexity about to O(V).
* Then off course we have to keep the track of all the path because each vertex has different connections so connection also increase so we have to maintain all the distances which leads to space complexity of O(E).

So overall space complexity of Dijkstra algorithm

|  |
| --- |
| Space Complexity= O(E+V) |

But is the worth notable thing the its space complexity also depends greatly on how Algorithm is implemented. Like in case of Adjacent Matrix its space complexity is about to O(V^2) due to store distance of all pairs of vertexes.

Dijkstra algorithm has great pros and cons. It has following advantages.

* **Correctness:** it provides and guarantees correct shortest path from one source vertex to all other vertexes in the graph of non-negative weights so it gives optimal solutions in term of graph path.
* **Versatility:** Dijkstra algorithm can be applied to all type of graphs including directed or undirected graph with or without cycles.it works with weighted graphs where each weights have positive weights.
* **Efficiency:** it is very efficient algorithm when it is implemented using priority queue or min-heap where it has time complexity of O (E (log V) where V is a number of vertexes and E is the number of edges.
* **Path Reconstruction flexibility:** it gives us flexibility to reconstruct the path after finding shortest path for one vertex which helps us to find shortest path for all the vertexes and leads to construct path again.

Although it has many advantages but it has some disadvantages too.

* **Memory usage:** As it maintains priority queue or min-heap heap for storing vertexes and their distances it also uses separate memory which may be good for small graphs but it could be result in use of high memory if we have to deal with largest graph having maximum number of vertexes and edges**.**
* **Negative weights:** We often hear that Dijkstra’s algorithm doesn’t work on negative edges but this is not a complete truth. Basically, it works for some negative weights too.

Let check it on graph 1

Before applying Dijkstra:

**0 10 ∞**

**5 -3**

**∞**

*Figure 1.1 graph with negative weights*

After applying Dijkstra:

**0 10 10**

**5 -3**

**5**

*Figure 1.2 resultant calculated shortest distances*

In this graph relaxation is totally fine and shortest distance found is exactly right so here Dijkstra is working.

Now let check Dijkstra on graph2

Before applying Dijkstra:

**0 10 ∞**

**5 -8**

**∞**

*Figure 2.1 graph with negative weights*

After applying Dijkstra:

**0 10 10**

**5 -8**

**5**

*Figure 2.2 resultant calculated shortest distances*

* Here in this graph, we can see when we visited vertex ‘A’ we apply relaxation to all its adjacent vertexes. Then we choose shortest distance vertex ‘C’ and apply relaxation but it has no adjacent vertexes so it doesn’t update distances of its adjacent. Then, when we visited ‘B’ and wanted to apply relaxation we see that 2<5 so in actual ‘C’ distance should be 2 but as it is already visited so, here we can’t update distance of vertex because according to Dijkstra relaxed vertex can not be relaxed again so it is giving here wrong distance of ‘C’ which is 5. So that’s why it is failed in that negative edge case.
* From graph1 and graph2 it is clear that we should not ignore graph completely if the graph is of negative weights but should check it because under certain negative weights it may give answer correct and in most cases of negative weights there is high possibility that it can give you wrong answer. So that’s is some reason Dijkstra algorithm is consider as not suitable for negative weights.

**Data Structures and Concepts:**

While implementation we have to deal with different data structures and concepts and each is used for a special purpose.

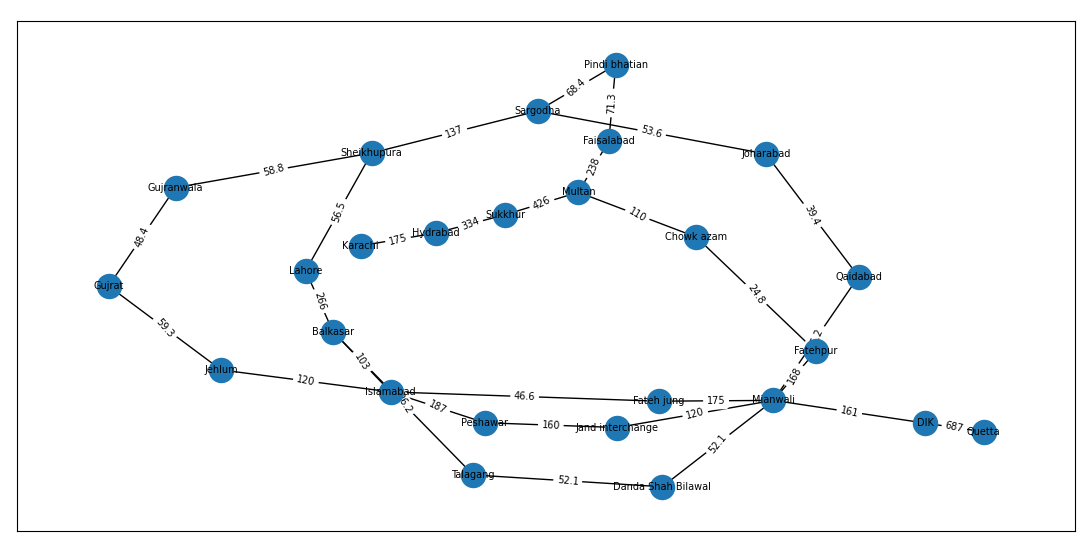
Here is a list of data structures used in implementation

|  |  |
| --- | --- |
| **Data Structure/Name** | **Description/Purpose** |
| Dictionary (graph) | A dictionary is used in implementation which has been used for storing vertexes and their corresponding weights between these vertexes. |
| Dictionary (distance) | A one more dictionary is used which is used to store shortest distances of each vertex that are part of graph. |
| Array (Queue) | An array is used which stores all the vertexes present in the graph |
| Array(distance) | An array is used which keeps all the shortest distance vertexes in array which latter used in plotting graph, all the vertexes pointed towards source vertex. |

*Table 1.3 for data structures and their purposes*

**Results:**

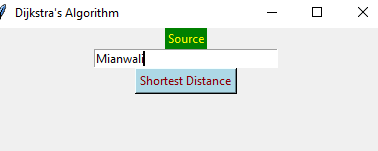
We have a graph of different cities 28 cities of Pakistan but we can extend cities as we want. We selected only 28 to visualize graph in good way. We showed in graph of the main one city of each province and calculated shortest distances from all other cities from its.



*Figure 2.3 graph of different cities*

Source:

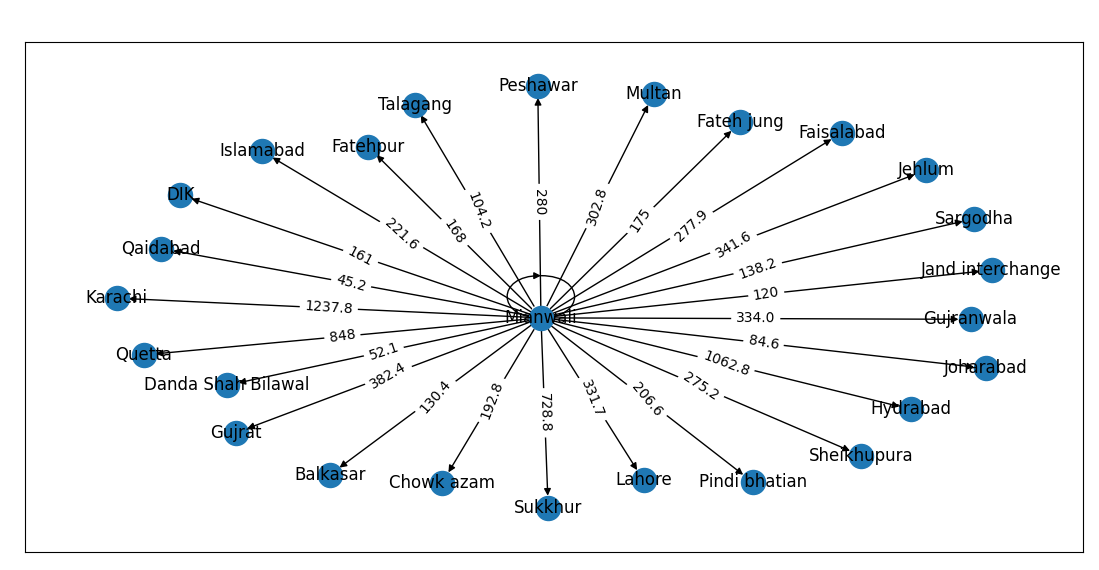
* When we have a source Mianwali.



*Figure 2.4 source vertex*

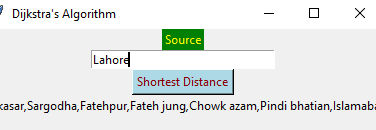
Shortest Distance from other cities:

The calculated shortest distance of all other cities from Mianwali is



*Figure 2.41 calculated shortest distance for all cities from source*

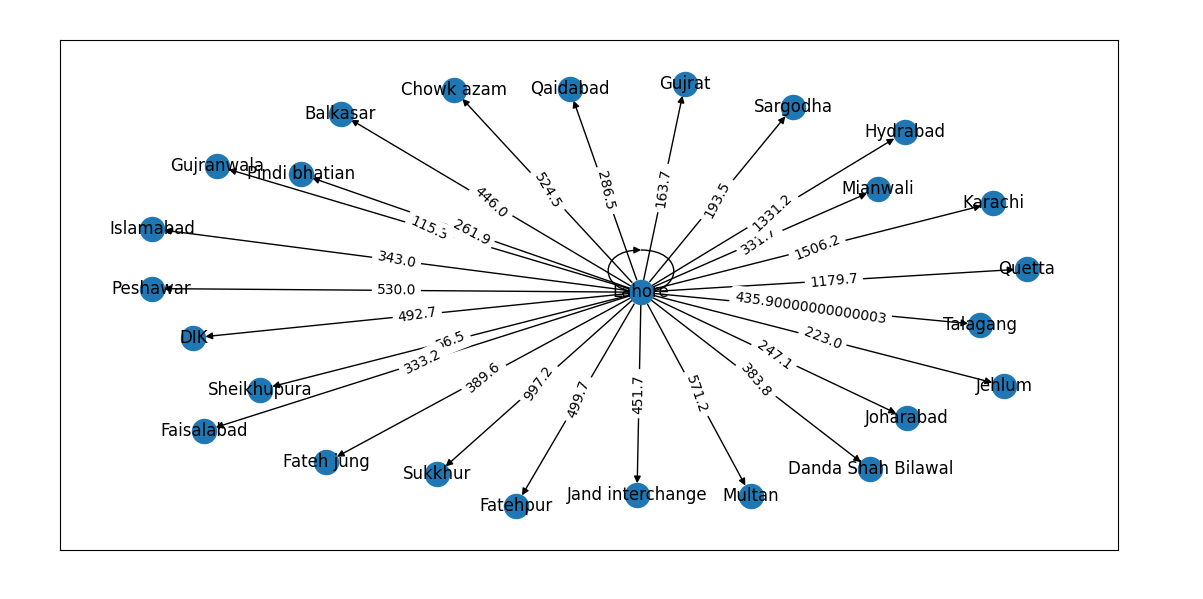
* When source is changed to Lahore



*Figure 2.5 source vertex*

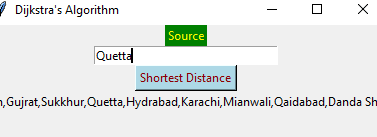
Shortest Distance from other cities:

The calculated shortest distance of all other cities from Lahore is



*Figure 2.51 calculated shortest distance for all cities from source*

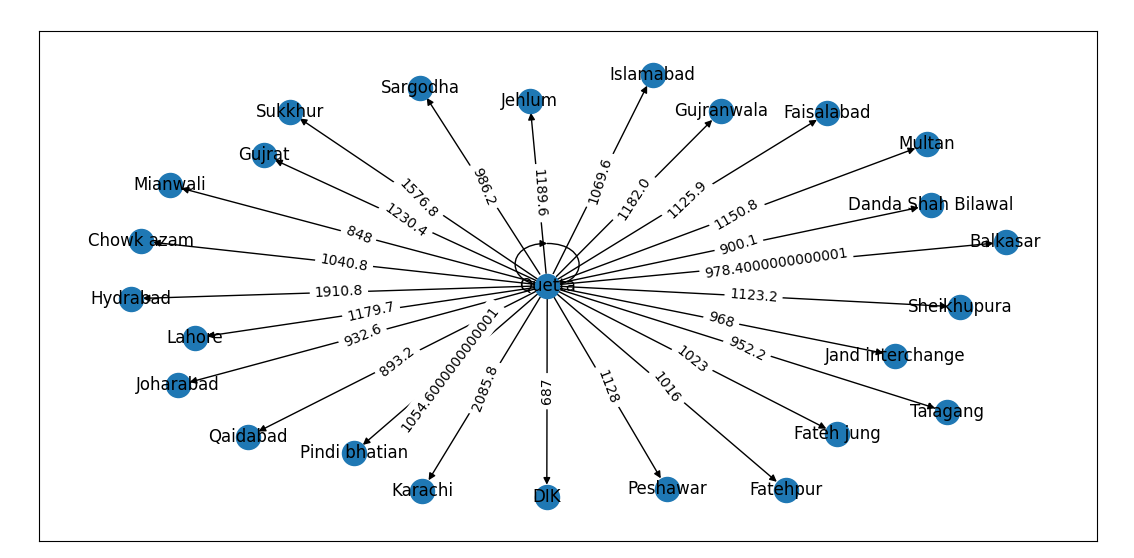
* When source is changed to Quetta



*Figure 2.6 source vertex*

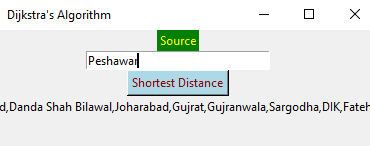
Shortest Distance from other cities:

The calculated shortest distance of all other cities from Quetta is



*Figure 2.61 calculated shortest distance for all cities from source*

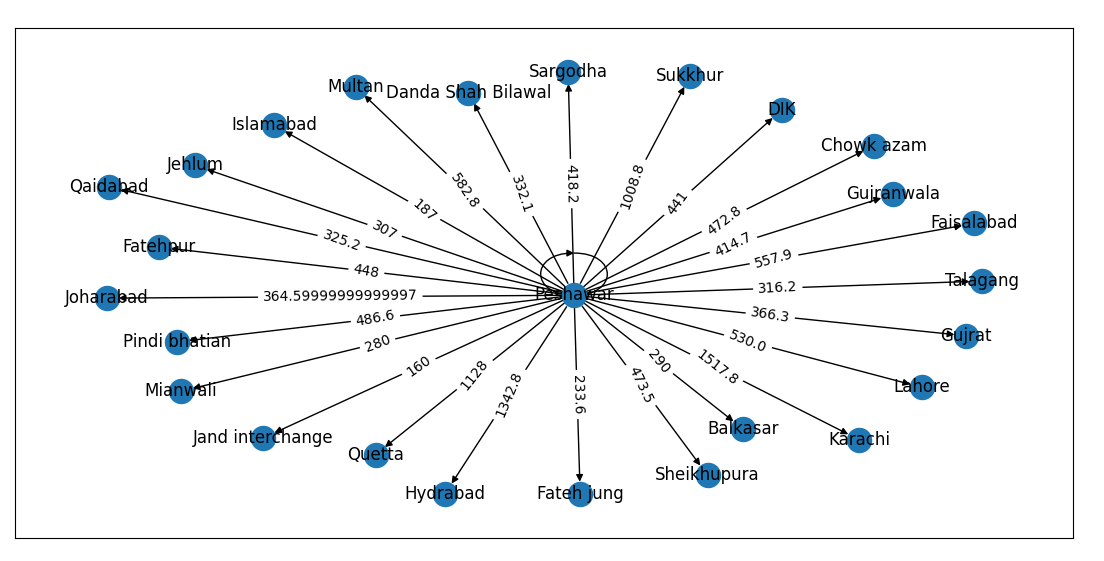
* When source is changed to Peshawar



*Figure 2.7 source vertex*

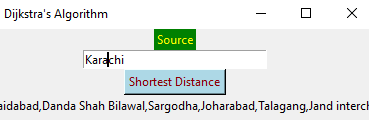
Shortest Distance from other cities:

The calculated shortest distance of all other cities from Peshawar is



*Figure 2.71 calculated shortest distance for all cities from source*

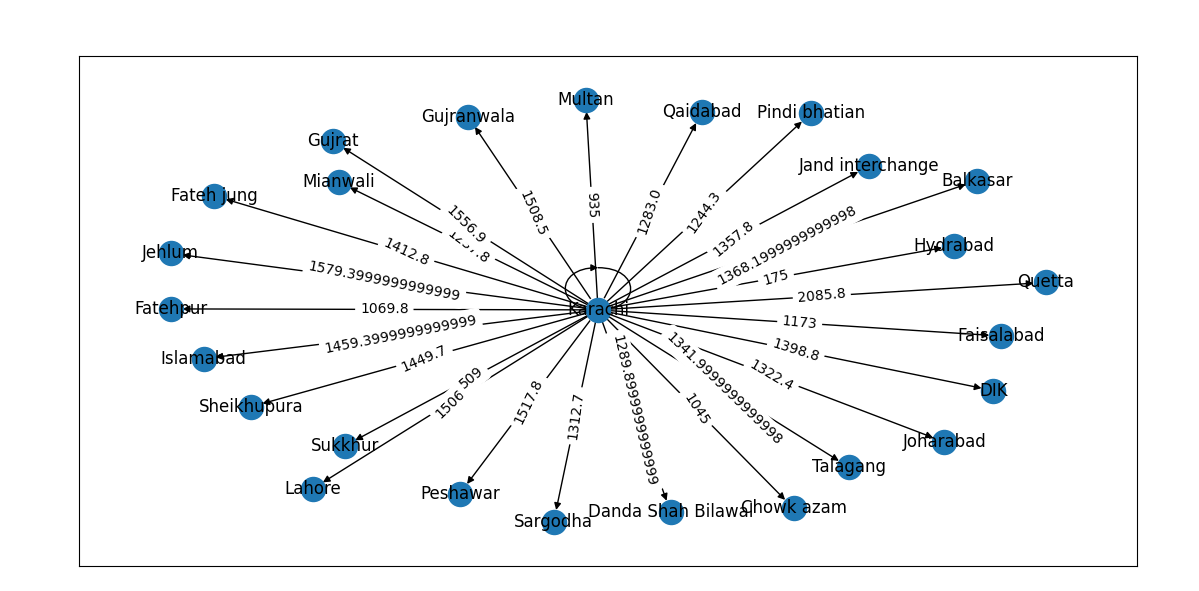
* When source is changed to Peshawar



*Figure 2.8 source vertex*

Shortest Distance from other cities:

The calculated shortest distance of all other cities from Karachi is



*Figure 2.81 calculated shortest distance for all cities from source*

**Code of Algorithm:**

Following is the complete code of algorithm used while implementation.

import networkx as nx

import matplotlib.pyplot as plt

import tkinter as tk

distance={}                                     # To store shortest calculated distances along with vertexes

visited\_vertex\_order=[]                         # store all the vertexes in which order they are visited

def declare\_distance(graph, source):            # A function to declare infinite distances to all the vertexes at initial stage which takes graph and source as inputs

    for vertex in graph:                        # A Loop to iterate all the vertexes in graph

        if vertex == source:                    # check if vertex is source

            distance[vertex] = 0                # if it is source declare its distance to zero

        else:                                   # check if vertexes are not source

            distance[vertex] = float('inf')        # declae all other vertexes distances as infinity

    return distance                         # return this declare distance

def dijkstra\_algorithm(graph, source):          #  function to implement dijkstra algorithm which takes graph and source as inputs

    print(declare\_distance(graph, source))             # call declare function so that this infinity distance is assigned to all vertexes except source

    Queue= list(graph.keys())                      # store all the vertees in graph and make a list of it

    while Queue:                                    # continue loop untill all the vertexes in visited and Queue becomes zero

        min\_distance = float('inf')                 # first suppose min-distance as infinity

        u = None                                     # initially is none but it will update according to minimum distance vertexes at each iteration

        for vertex in Queue:                            #iterate all the vertexes in Queue

            if distance[vertex] < min\_distance:         # if distance of current vertex is less than min-distance that is supposed,updated after each iteration

                min\_distance = distance[vertex]         #update min distance with current distance vertex

                u = vertex                              #store this minimum distance vertex in u

        for adjacent, weight in graph[u]:               #visit each adjacent vertexes of vertex that is in u and check their weights

            alter\_distance = distance[u] + weight       # store sum distance of current vertex and their edges weight into alter distance variable

            if alter\_distance < distance[adjacent]:             # check for relaxtion if alter-distance is less than adjacent vertex distance

                distance[adjacent] = alter\_distance                 # updateadjacent distance vertexes

        visited\_vertex\_order.append(u)                                  #save it in visited that vertex is selected according to their short distance store in u

        Queue.remove(u)                                         #removed that visited vertexes from Queue

    return distance                                             # return final shortest distance

def Visualize\_Given\_Graph(graph):                               #function to visualize graph

    Empty\_graph = nx.Graph()                                    #create an empty graph

    Empty\_graph.add\_nodes\_from(graph.keys())                    #add all the nodes from graph in empty graph

    for node, neighbors in graph.items():                       #iterate each node and their adjacent in graph

        print(neighbors)

        for neighbor, weight in neighbors:                       # iterate through neighbours and their weights

            Empty\_graph.add\_edge(node, neighbor, weight=weight)                 #add edges and their weights of node with their adjacents

        # g.nodes[node]['weight'] = sum(weight for \_, weight in neighbors)

    Layout = nx.spring\_layout(Empty\_graph)

    nx.draw\_networkx(Empty\_graph, pos=Layout, with\_labels=True, node\_size=300, font\_size=7)

    node\_labels = nx.get\_node\_attributes(Empty\_graph, 'weight')

    nx.draw\_networkx\_labels(Empty\_graph, pos=Layout, labels=node\_labels,font\_size='7')

    edge\_labels = nx.get\_edge\_attributes(Empty\_graph, 'weight')

    nx.draw\_networkx\_edge\_labels(Empty\_graph, pos=Layout, edge\_labels=edge\_labels,font\_size='7')

    plt.axis()

    plt.show()              #show graph

def Plot\_Shortest\_Path\_Graph(distancee):                        #to viisulaize shortest culculated vertexes along their distances

    global source\_node                                          #to use source node declare it as global variable

    G = nx.DiGraph()                                                #create an empty graph

    nodes = list(distance.keys())                               # store all the vertex from shortest distance claculated vertexes

    G.add\_nodes\_from(nodes)                                         # add the vertexes to make graph

    for i in range(0, len(nodes)):                                     #iterate through each vertexes

        current\_node = nodes[i]                                              #store each vertexes at each iteation in as current node

        G.add\_edge(source\_node,current\_node, weight=distance[current\_node])             #add edge between source and current node and also draw their calculated distance

    pos = nx.spring\_layout(G)

    nx.draw\_networkx\_nodes(G, pos)

    nx.draw\_networkx\_edges(G, pos)

    nx.draw\_networkx\_labels(G, pos)

    edge\_labels = nx.get\_edge\_attributes(G, 'weight')

    nx.draw\_networkx\_edge\_labels(G, pos, edge\_labels=edge\_labels)

    plt.axis()

    plt.show()                      #sgow graph

def Obtain\_Graph():                         #to obtain graph

    global source\_node                      #declare source node as global to use it

    source\_node = source\_entry.get()                #get source node

    if source\_node in graph:                        #check if source node is in graph

        shortest\_path\_distance = dijkstra\_algorithm(graph, source\_node)              #store shorted distance in shortest path

        print(visited\_vertex\_order,'visited order')

        print(shortest\_path\_distance,'smallest path')

        # print(distance)

        result\_label.config(text="Visited Order " + ','.join(visited\_vertex\_order))            # print visited orde,separted by commos

        Plot\_Shortest\_Path\_Graph(distance)                  #plot shortest distance graph

    else:

        result\_label.config(text="Invalid node!")           # if node is not in graph

graph={             #dict to create graph

    'Mianwali':[('Danda Shah Bilawal',52.1),('Fateh jung',175),('Qaidabad',45.2),('Jand interchange',120),('Fatehpur',168),('DIK',161)],

    'DIK':[('Mianwali',161),('Quetta',687)],

    'Quetta':[('DIK',687)],

    'Danda Shah Bilawal' :[('Mianwali',52.1),('Talagang',52.1)],

    'Talagang' :[('Danda Shah Bilawal',52.1),('Balkasar',26.2)],

    'Balkasar' :[('Islamabad',103),('Lahore',266),('Talagang',26.2)],

    'Fateh jung':[('Islamabad',46.6),('Mianwali',175)],

    'Islamabad' :[('Fateh jung',46.6),('Balkasar',103),('Jehlum',120),('Peshawar',187)],

    'Jand interchange' :[('Mianwali',120),('Peshawar',160)],

    'Peshawar':[('Jand interchange',160),('Islamabad',187)],

    'Qaidabad':[('Mianwali',45.2),('Joharabad',39.4)],

    'Joharabad':[('Qaidabad',39.4),('Sargodha',53.6)],

    'Sargodha' :[('Joharabad',53.6),('Sheikhupura',137),('Pindi bhatian',68.4)],

    'Sheikhupura':[('Sargodha',137),('Lahore',56.5),('Gujranwala',58.8)],

    'Lahore':[('Sheikhupura',56.5)],

    'Gujranwala':[('Gujrat',48.4),('Sheikhupura',58.8)],

    'Gujrat':[('Jehlum',59.3),('Gujranwala',48.4)],

    'Jehlum':[('Gujrat',59.3),('Islamabad',120)],

    'Pindi bhatian':[('Sargodha',68.4),('Faisalabad',71.3)],

    'Faisalabad':[('Pindi bhatian',71.3),('Multan',238)],

    'Multan':[('Faisalabad',238),('Chowk azam',110),('Sukkhur',426)],

    'Sukkhur':[('Hydrabad',334),('Multan',426)],

    'Hydrabad':[('Sukkhur',334),('Karachi',175)],

    'Karachi':[('Hydrabad',175)],

    'Chowk azam':[('Fatehpur',24.8),('Multan',110)],

    'Fatehpur':[('Chowk azam',24.8),('Mianwali',168)],

}

# to visalize initial graph

Visualize\_Given\_Graph(graph)

#create tk window

window = tk.Tk()

#set window geomerty

window.geometry("400x300")

#set title of window

window.title("Dijkstra's Algorithm")

# Create label

node\_label = tk.Label(window, text="Source",fg='yellow',bg='green')

node\_label.pack()

#source node as none initially

source\_node=None

# Create entry field

source\_entry = tk.Entry(window, width=30)

source\_entry.pack()

# Create button

traverse\_button = tk.Button(window, text="Shortest Distance", command=Obtain\_Graph,fg='darkred',bg='lightblue')

traverse\_button.pack()

# Create result label

result\_label = tk.Label(window, text="")

result\_label.pack()

window.mainloop()                   #show window

**Areas of improvement:**

Although Dijkstra algorithm is very efficient and reliable for calculating shortest path with positive edges but as it is said that nothing is perfect so there remains improvement in everything, similarly following are the necessary improvement needed in Dijkstra algorithm.

* Space Complexity:

Dijkstra algorithm implementation in min-heap or priority queue requires data structure to store vertexes and their shortest distance which is not of big cost for small graphs but as the graphs vertexes increases and number of links between vertexes (edges) increase where need a huge space to store all the vertexes and distance which is very costly there it should be implemented using techniques like fibnoocci series where it is implemented in relatively very low cost.

* Negative weights:

Dijkstra algorithm implementation usually preferred in implementing using positive weights, it gives some time correct answer with negative weights so it always considers that it works for positive weights and this the area where there needs an improvement so that it could work for all weights.

* Parallelization:

Dijkstra algorithm works in sequential ways for shortest distance vertexes. It selects sequentially one vertex whose distance is shorted while other vertexes remain in visit so parallel operation cannot be performed where there needs improvement it should perform parallel operation using some threads or processors.

* Time complexity of Dense Graph: if a graph is very dense mean many links from one vertex than time complexity could reach to O(V^2).

**References:**

For GUI:

<https://github.com/CGHoussem/dijkstra-gui>

For Checking data of Cities and their distances;

https://www.google.com/maps/@30.3593034,68.9966984,6z?entry=ttu