**Experiment No: 7**

**AIM:** Implementation Single Source Shortest path algo(Dijkstra’s algo) and obtaining its step count.

**THEORY:**

Dijkstra's Algorithm is the best-known algorithm for the single-source shortest-paths problem. This algorithm is applicable to undirected and directed graphs with nonnegative weights only.

The major difference between this algorithm and the Bellman Ford algorithm is that the former fails for graphs with negative weight edges.

**Algorithm writing:**

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

….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.

**ALGORITHM:**

**AlgorithmShortestPaths**(u,cost,dist,n)

// dist[j],1<j <n, is setto the lengthof the shortest

// path from vertex v to vertex j in a digraph G with n

// vertices. dist[v] is set to zero. G is represented by its

// cost adjacency matrix cost[1:n, 1:n].

{

**for** i :=1to n do

{// Initialize S.

S[i]:=false;

distance[i]:=cost[v,i];

}

S[v]:=true; dist[v] :=0.0;// *Put v in S*

**for** num :=2 to n-1 do

{

// Determine n-1 paths from v.

Choose u from among those vertices not

in S such that dist[u] is minimum;

S[u]:=true; // Put u in S.

for (each w adjacent to u with S[w]= false) do

// Updated distances.

**if** (dist[w] >dist[u]+cost[u, w]) then

dist[w] :=dist[u]+cost[u, w];

}

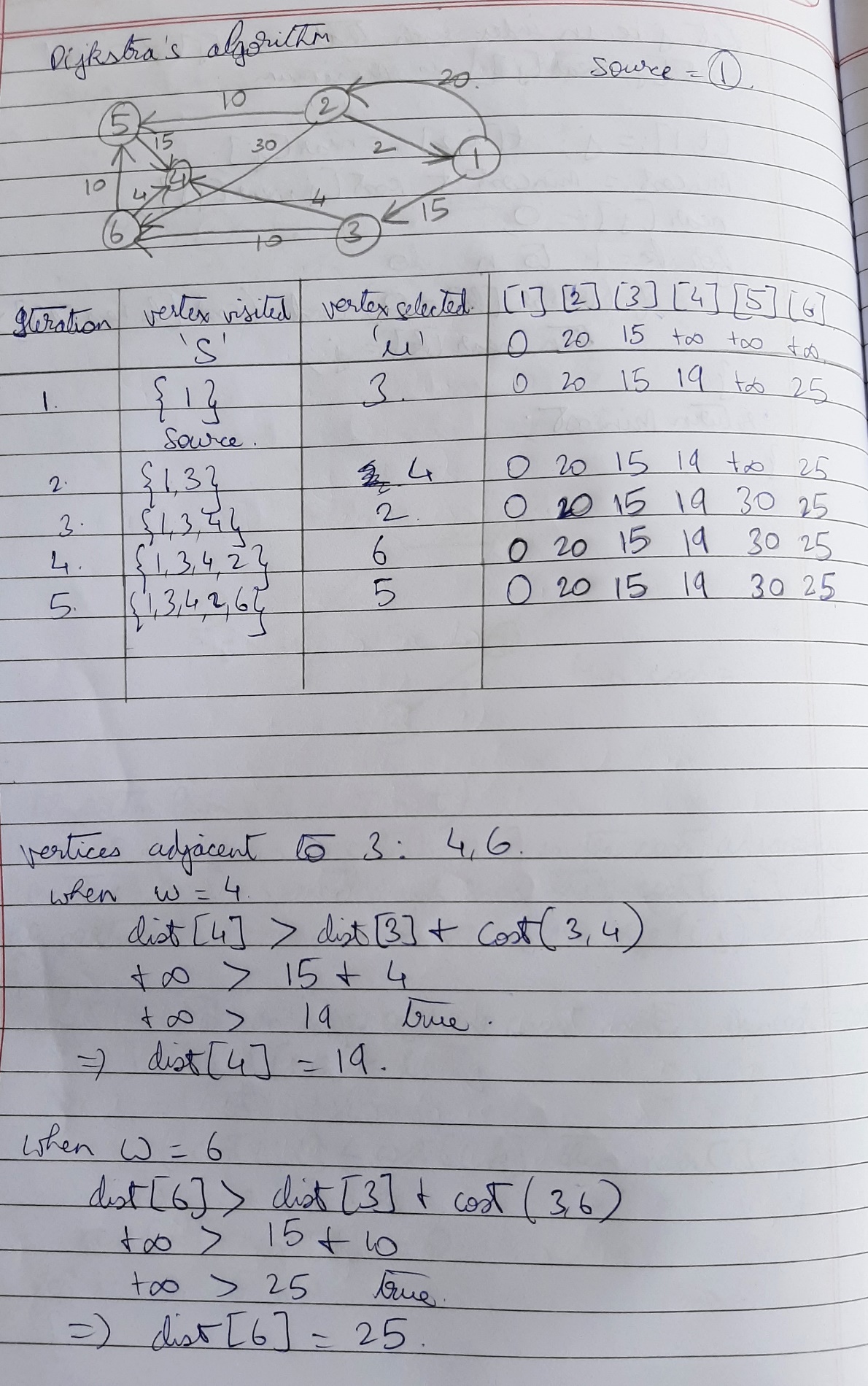
}

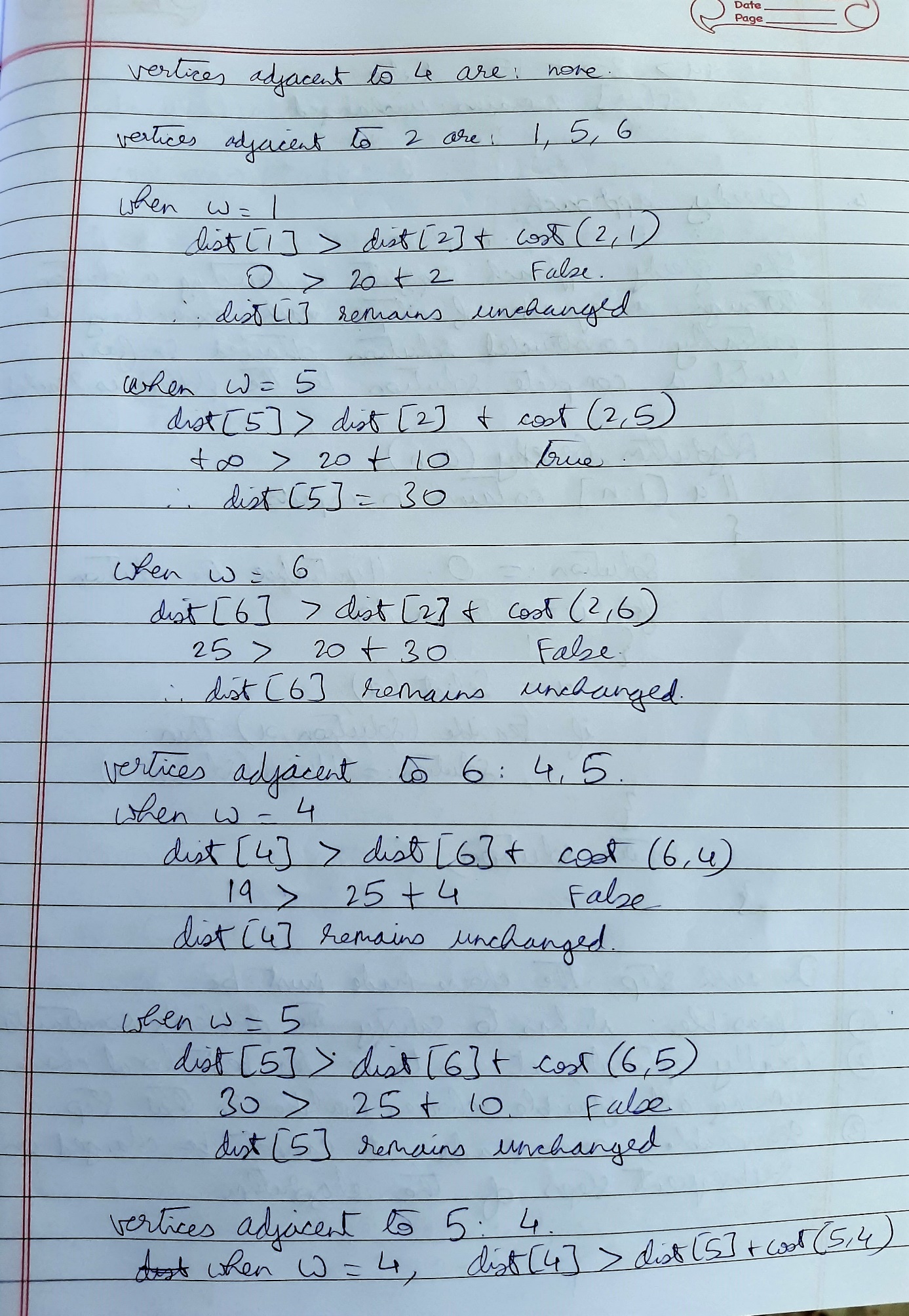
*Time Complexity*

• The time efficiency of Dijkstra’s algorithm depends on the data structures used for implementing the priority queue and for representing an input graph itself.

* Efficiency is Θ(||) for graphs represented by their weight matrix and the priority queue implemented as an unordered array.
* For graphs represented by their adjacency lists and the priority queue implemented as a minheap, it is in O (|E| log |V| )

*Problem Tracing*





PROGRAM IMPLEMENTATION:

#include<iostream>

using namespace std;

const int v=10;

int n, count=0;

void show\_paths(int \*dist)

{

cout<<"\nVertex Distance from source:\n";

for(int i=0;i<n;i++)

cout<<" "<<i<<" "<<dist[i]<<endl;

}

int findmin(bool \*visited, int \*dist)

{

int min=99, u;

count++;

for(int i=0;i<n;i++)

{

count+=2;

if(visited[i]==false && min>dist[i])

min=dist[i],u=i,count+=2;

}

count++;

return u;

}

void shortestpaths(int graph[v][v], int s)

{

int dist[n]; //collection of shortest paths from source

bool visited[n]; //Keeps track of visited vertices

for(int i=0;i<n;i++)

visited[i]=false,

dist[i]=99, count+=3;

count++; //end of for

dist[s]=0; //distance from source to itself is zero

count++;

for(int i=0;i<n;i++)

{

int u = findmin(visited,dist);

visited[u]=true;

count+=3; //outer for and assignments

for(int j=0;j<n;j++)

{

count+=2; //for loop and if statement

if(graph[u][j]!=0 && j!=s)

if(dist[j]> dist[u] + graph[u][j])

dist[j] = dist[u] + graph[u][j],count+=2;

}

}

show\_paths(dist);

}

int main()

{

cout<<"Enter number of vertices\n";

cin>>n;

int graph[v][v]

for(int i=0;i<n;i++)

for(int j=0;j<n;j++)

{

cout<<"\nEnter edge cost between "<<i<<" and "<<j<<":";

cin>>graph[i][j];

}

int source;

cout<<"Enter the source vertex\n";

cin>>source;

shortestpaths(graph,source);

cout<<"\nStepCount="<<count<<endl;

return 0;

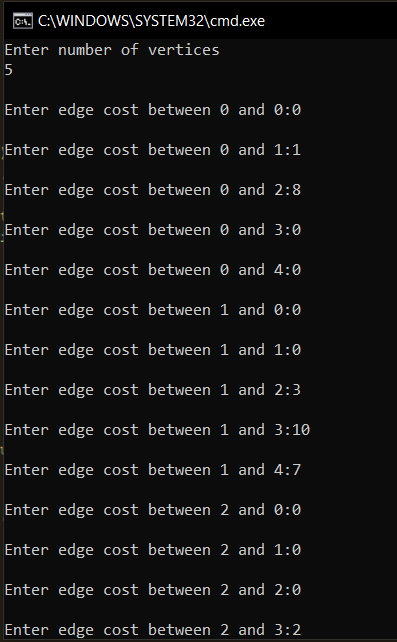
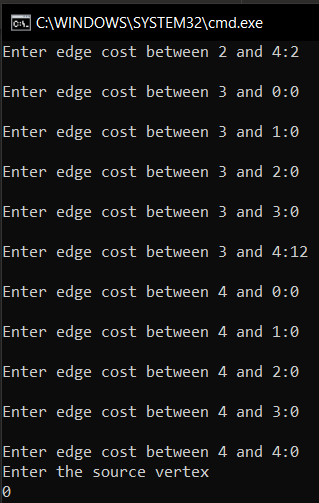
}

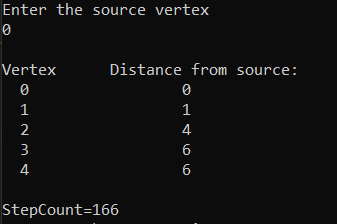
}

OUTPUTS:

1. When v=5

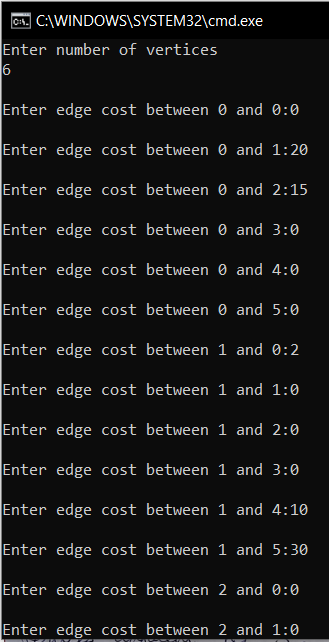
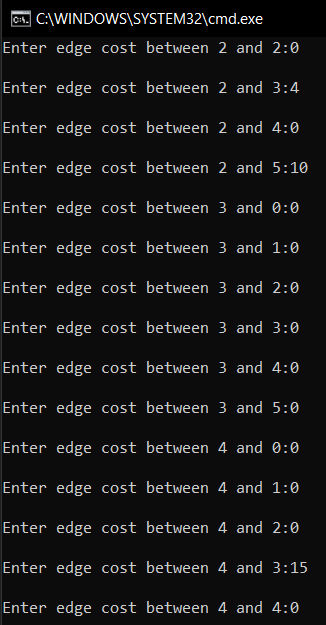
**Count=166**

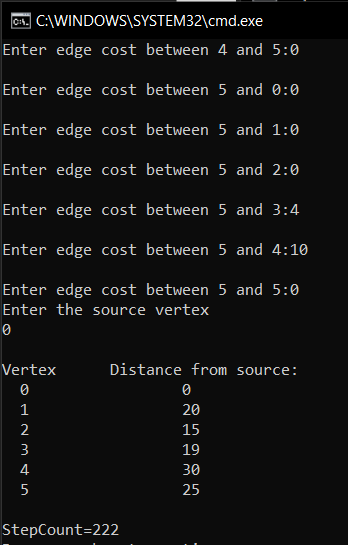
 



1. When v=6

**Count=222**

** **

****

**Conclusion**:

1. 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.
2. Dijkstra’s algorithm doesn’t work for graphs with negative weight cycles, i.e., it does not guarantee the correct results for a graph with negative edges.