Title: Dijkstra's and Bellman-Ford algorithms

Sravani Tangeda

AP20110010174

sravani_tangeda@srmap.edu.in

Objective:

The main objective of this experiment is to implement Dijkstra's and Bellman Ford algorithms

Problem Statement:

We can determine the shortest route between any two graph vertices using Dijkstra's approach. Dijkstra algorithm is a single-source shortest path algorithm. The shortest path from the source to all nodes is calculated. It is different from the minimal spanning tree. Because not all of the vertices of the graph may be included in the shortest distance between two vertices.

In a weighted graph, This algorithm is used to find the shortest distance from the single vertex to all the other vertices of a weighted graph. It is similar to Dijkstra's but can work with negative weights.

Algorithm:

DIJKSTRA'S ALGORITHM

STEP 1: Mark the ending vertex with a distance of zero. Designate this vertex as current.

STEP 2: Find all vertices leading to the current vertex. Calculate their distances to the end. Since we already know the distance the current vertex is from the end, this will just require adding the most recent edge. Don't record this distance if it is longer than a previously recorded distance.

STEP 3: Mark the current vertex as visited. We will never look at this vertex again.

STEP 4: Mark the vertex with the smallest distance as current, and repeat from step 2.

BELLMEN-FORD ALGORITHM

STEP 1: Make a list of all the graph's edges. This is simple if an adjacency list represents the graph.

STEP 2: "V - 1" is used to calculate the number of iterations. Because the shortest distance to an edge can be adjusted V - 1 time at most, the number of iterations will increase the same number of vertices.

STEP 3: Begin with an arbitrary vertex and a minimum distance of zero. Because you are exaggerating the actual distances, all other nodes should be assigned infinity.

For each edge u-v, relax the path lengths for the vertices:

If distance[v] is greater than distance[u] + edge weight uv, then distance[v] = distance[u] + edge weight uv

STEP 4:If the new distance is less than the previous one, update the distance for each Edge in each iteration. The distance to each node is the total distance from the starting node to this specific node.

STEP 5: To ensure that all possible paths are considered, you must consider alliterations. You will end up with the shortest distance if you do this.

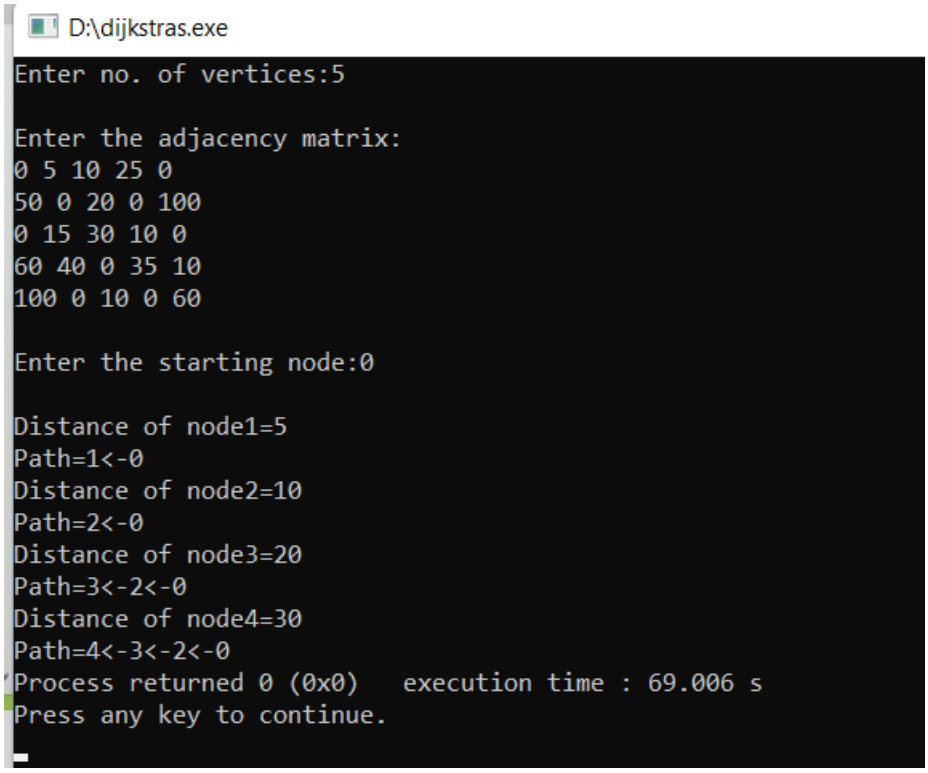
Dijkstra's Code:

```
#include<stdio.h>
#include<conio.h>
#define INFINITY 9999
#define MAX 10

void dijkstra(int G[MAX][MAX],int n,int startnode);

int main()
{
   int G[MAX][MAX],i,j,n,u;
   printf("Enter no. of vertices:");
   scanf("%d",&n);
   printf("\nEnter the adjacency matrix:\n");
   for(i=0;i<n;i++)
      for(j=0;j<n;j++)
      scanf("%d",&G[i][j]);
   printf("\nEnter the starting node:");
   scanf("%d",&u);
   dijkstra(G,n,u);
   return 0;
}
void dijkstra(int G[MAX][MAX],int n,int startnode)
{
```

```
int cost[MAX][MAX],distance[MAX],pred[MAX];
int visited[MAX],count,mindistance,nextnode,i,j;
//pred[] stores the predecessor of each node
//count gives the number of nodes seen so far
//create the cost matrix
for(i=0;i<n;i++)
   for(j=0;j<n;j++)
   if(G[i][j]==0)
   cost[i][j]=INFINITY;
else
   cost[i][j]=G[i][j];
//initialize pred[],distance[] and visited[]
for(i=0;i<n;i++)
{
   distance[i]=cost[startnode][i];
   pred[i]=startnode;
   visited[i]=0;
}
distance[startnode]=0;
visited[startnode]=1;
count=1;
while(count<n-1)
{
   mindistance=INFINITY;
//nextnode gives the node at minimum distance
for(i=0;i<n;i++)
   if(distance[i]<mindistance&&!visited[i])
   {
      mindistance=distance[i];
      nextnode=i;
```

```
    }
  //check if a better path exists through nextnode
  visited[nextnode]=1;
  for(i=0;i<n;i++)
    if(!visited[i])
    if(mindistance+cost[nextnode][i]<distance[i])
    {
      distance[i]=mindistance+cost[nextnode][i];
      pred[i]=nextnode;
    }
    count++;
  }
  //print the path and distance of each node
  for(i=0;i<n;i++)
    if(i!=startnode)
    {
      printf("\nDistance of node%d=%d",i,distance[i]);
      printf("\nPath=%d",i);
      j=i;
      do
        {
          j=pred[j];
          printf("<-%d",j);
        }while(j!=startnode);
    }
}
```

Output:



Bellman-Ford Code:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <limits.h>

struct Edge
{
    // This structure is equal to an edge. Edge contains two end points. These edges are directed edges so they
//contain source and destination and some weight. These 3 are elements in this structure
    int source, destination, weight;
};

// a structure to represent a connected, directed and weighted graph
struct Graph
{
```

```
    int V, E;
// V is number of vertices and E is number of edges

    struct Edge* edge;
// This structure contain another structure which we already created edge.
};

struct Graph* createGraph(int V, int E)
{
    struct Graph* graph = (struct Graph*) malloc( sizeof(struct Graph));
//Allocating space to structure graph

    graph->V = V;   //assigning values to structure elements that taken form user.

    graph->E = E;

    graph->edge = (struct Edge*) malloc( graph->E * sizeof( struct Edge ) );
//Creating "Edge" type structures inside "Graph" structure, the number of edge type structures are equal to number of edges

    return graph;
}

void FinalSolution(int dist[], int n)
{
// This function prints the final solution
    printf("\nVertex\tDistance from Source Vertex\n");
    int i;

    for (i = 0; i < n; ++i){
printf("%d \t\t %d\n", i, dist[i]);
```

```
}
}

void BellmanFord(struct Graph* graph, int source)
{
    int V = graph->V;

    int E = graph->E;

    int StoreDistance[V];

    int i,j;

    // This is initial step that we know , we initialize all distance to infinity except source.
// We assign source distance as 0(zero)

    for (i = 0; i < V; i++)
        StoreDistance[i] = INT_MAX;

    StoreDistance[source] = 0;

    //The shortest path of graph that contain V vertices, never contain "V-1" edges. So we do
here "V-1" relaxations
    for (i = 1; i <= V-1; i++)
    {
        for (j = 0; j < E; j++)
        {
            int u = graph->edge[j].source;

            int v = graph->edge[j].destination;
```

```
            int weight = graph->edge[j].weight;

            if (StoreDistance[u] + weight < StoreDistance[v])
                StoreDistance[v] = StoreDistance[u] + weight;
        }
    }

    // Actually upto now shortest path found. But BellmanFord checks for negative edge cycle. In this step we check for that
    // shortest distances if graph doesn't contain negative weight cycle.

    // If we get a shorter path, then there is a negative edge cycle.
    for (i = 0; i < E; i++)
    {
        int u = graph->edge[i].source;

        int v = graph->edge[i].destination;

        int weight = graph->edge[i].weight;

        if (StoreDistance[u] + weight < StoreDistance[v])
            printf("This graph contains negative edge cycle\n");
    }

    FinalSolution(StoreDistance, V);

    return;
}

int main()
{
```

```
    int V,E,S;  //V = no.of Vertices, E = no.of Edges, S is source vertex

printf("Enter number of vertices in graph\n");
    scanf("%d",&V);

printf("Enter number of edges in graph\n");
    scanf("%d",&E);

printf("Enter your source vertex number\n");
scanf("%d",&S);

    struct Graph* graph = createGraph(V, E);    //calling the function to allocate space to these many vertices and edges

    int i;
    for(i=0;i<E;i++){
        printf("\nEnter edge %d properties Source, destination, weight respectively\n",i+1);
        scanf("%d",&graph->edge[i].source);
        scanf("%d",&graph->edge[i].destination);
        scanf("%d",&graph->edge[i].weight);
    }

    BellmanFord(graph, S);
//passing created graph and source vertex to BellmanFord Algorithm function

    return 0;
}
```

Output:

D:\bellmen.exe

```
Enter number of vertices in graph
5
Enter number of edges in graph
10
Enter your source vertex number
0

Enter edge 1 properties Source, destination, weight respectively
0 1 6

Enter edge 2 properties Source, destination, weight respectively
0 2 7

Enter edge 3 properties Source, destination, weight respectively
1 2 8

Enter edge 4 properties Source, destination, weight respectively
1 4 -4

Enter edge 5 properties Source, destination, weight respectively
1 3 5

Enter edge 6 properties Source, destination, weight respectively
3 1 -2

Enter edge 7 properties Source, destination, weight respectively
2 3 -3

Enter edge 8 properties Source, destination, weight respectively
2 4 9

Enter edge 9 properties Source, destination, weight respectively
4 0 2

Enter edge 10 properties Source, destination, weight respectively
4 3 7

Vertex  Distance from Source Vertex
0               0
1               2
2               7
3               4
4               -2

Process returned 0 (0x0)   execution time : 147.335 s
Press any key to continue.
```

Problems Faced:

Initially, I found it difficult to implement Dijkstra’s and Bellman-Ford’s algorithms and know the difference and best approach among them.

Conclusion:

In this experiment, I have learned the implementation of Dijkstra's and Bellmen-Ford’s algorithms and learned that Dijkstra’s algorithm is better when it comes to reducing time complexity. However, when we have negative weights, we have to go with the Bellman-Ford algorithm.