Data Structures and Algorithms

Mini Project 2

Submitted To:

Dr. Omer Ahmed

Submitted By:

Manaal Waseem

FA18-BCE-074

Mini Project 1: Finding Shortest Path using Dijkstra's Algorithm

Project Description:

In this project you are required to implement the Dijkstra's Algorithm which is a Breadth First Search (BFS) algorithm for finding shortest path from a starting vertex (src) to every other vertex in the graph. You are provided with skeleton code that implements a directed weighted graph represented by an Adjacency Matrix.

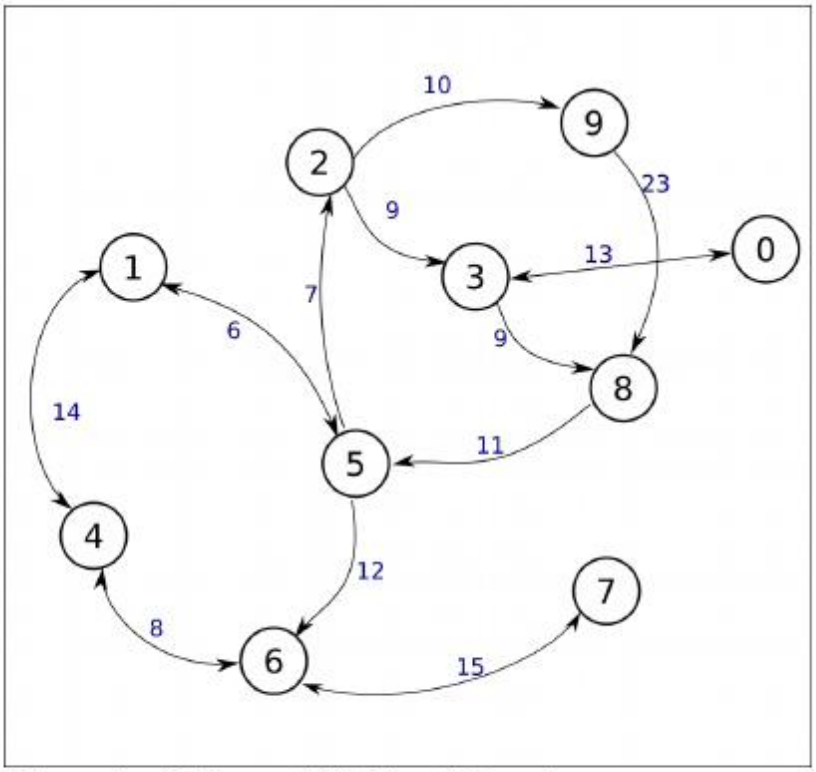


Figure 1: A Directed Weighted Graph

Restrictions:

- You will implement the Dijkstra's Algorithm using C language.
- This is not a group activity and each student must work independently.
- You are allowed to consult books or Internet resources but should mention the resources in your reports.
- You are not allowed to use classes and objects or other purely object oriented programming constructs.

Task:

Your task is to write a function

'void find_shortest_paths(int * graph, int size, int src, int * dist_array)'

that will compute the shortest path from src to all the other vertices in the graph using Dijkstra's Algorithm.

Project Design:

The project contains a file named **main.c** which includes the function to be completed.

```
void find_shortest_paths( int * graph, int size, int src, int * dist_array)

{
    /*** Complete this function ***/
    int i, count, min, u, v, visit_flag[size];

    for(i=0; i<size; i++)
    {
       visit_flag[i]=0; ///mark each node unvisited
       *(dist_array+i) = *((graph+src*size)+i); ///assign distances from source node in distance array
    }

    visit_flag[src] = 1;     ///mark the source node as visited
    count=2;

    while(count<size)
    {

        min=9999;

        for(v=0; v<size; v++)
        {
            if((dist_array[v] != -1) && (dist_array[v] < min) && !visit_flag[v])
            {
                min = dist_array[v];
                u = v;
            }
        }

        visit_flag[u]=1;  ///mark the current node as visited


        for(v=0; v<size; v++)
        {
            if((*((graph+u*size)+v) != -1) && !visit_flag[v])    ///check 1. wether a path exist between u and v or not
                                                                  ///       2. v has not been visited
            {
                ///If the distance to v is not already calculated or the distance through u is
                ///minimun then assign the calculated distance through u to dist_array at position v
                if ((dist_array[v] == -1) || ((dist_array[u]+*((graph+u*size)+v)) < dist_array[v]))
                        dist_array[v]=dist_array[u]+*((graph+u*size)+v);
            }
        }
        count++;
    }
}
```

Function: find_shortest_paths

Implementation of Function

Following is the description of implementation for function to be completed:

In this program, the function ***"find_shortest_paths"*** implements Dijkstra's algorithm and takes:

- Pointer to **'graph'** i.e. *adjacency matrix* (a 2-D array)
- **'size'** i.e. *total number of vertices* (number of rows as well as columns)
- **'src'**, the *initial node*
- Pointer to **'dist_array'** i.e. *one dimensional array to store shortest distance of each node from initial node* as its argument.

Variables **'i'**, **'count'**, **'min'**, **'u'**, and **'v'** have been declared. Moreover, an array **'visit_flag[size]'** has been declared to keep track of visited nodes. Now a **for loop** iterates from **'0'** upto **'size'**; in which firstly, array **'visit_flag[size]'** is populated by **zeros** i.e. *to mark each node unvisited initially* and secondly, distances from **'src'** are stored in **'dist_array'** by traversing through the row (row number = **'src'**) of **'graph'**. After exiting **for loop**, initial node (source node) **'src'** is marked visited i.e. **1** and **'count'** is initialized as **2**.

Now a **while loop** iterates until **'count'** is less than **'size'**; **'min'** is initialized as **9999**(a very large integer).

Again, a **for loop** iterates from **'0'** upto **'size'** that selects the unvisited node that has the smallest distance and set it as the new current node **'u'** which is implemented as; an **if-statement** checks following three conditions:

- Whether node **v** is the neighbor of current node.
- Distance (weight) of the node is less then **'min'**.
- It has not already been visited yet.

If these conditions are fulfilled, distance (weight) of node **v** is assigned to **'min'**. And value of **'v'** is assigned to **'u'** making it the current node. Once the **for loop** is exited, current node is marked visited.

Yet again, a **for loop** iterates from **'0'** upto **'size'**; an **if-statement** checks following two conditions:

- Whether node **v** is the neighbor of current node **'u'**.
- It has not already been visited yet.

If these conditions are fulfilled, the next **if block** the distance of the unvisited neighbor of **'u'** i.e. **'v'** is calculated through **'u'** and if this distance is either less than the already present distance in the **'dist_array'** for node **'v'** or there is no calculated distance present in the **'dist_array'** for node i.e. **dist_array[v] = -1** then it is assigned to **dist_array[v]**. **For loop** is exited and **'count'** is incremented by 1. Lastly, **while loop** is exited.

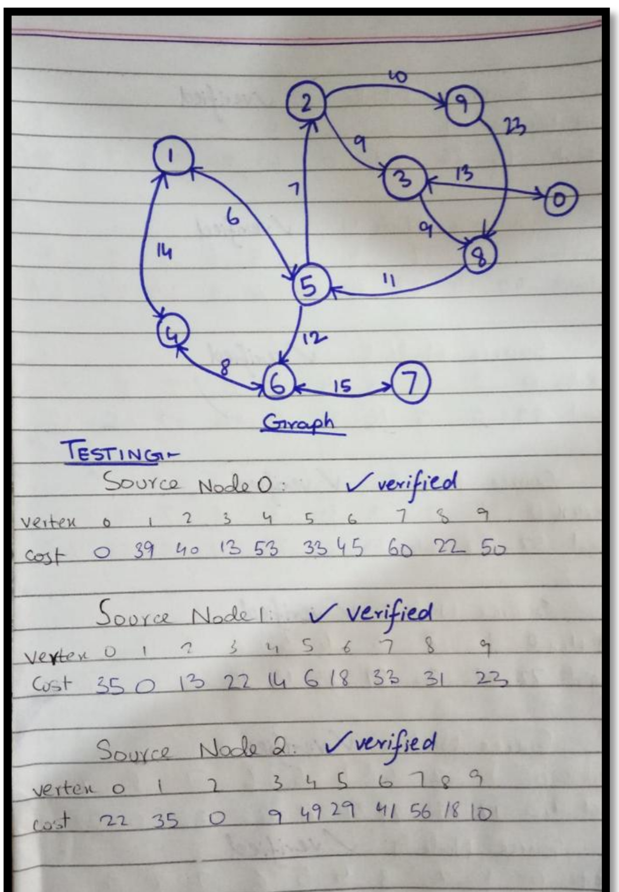
Output:

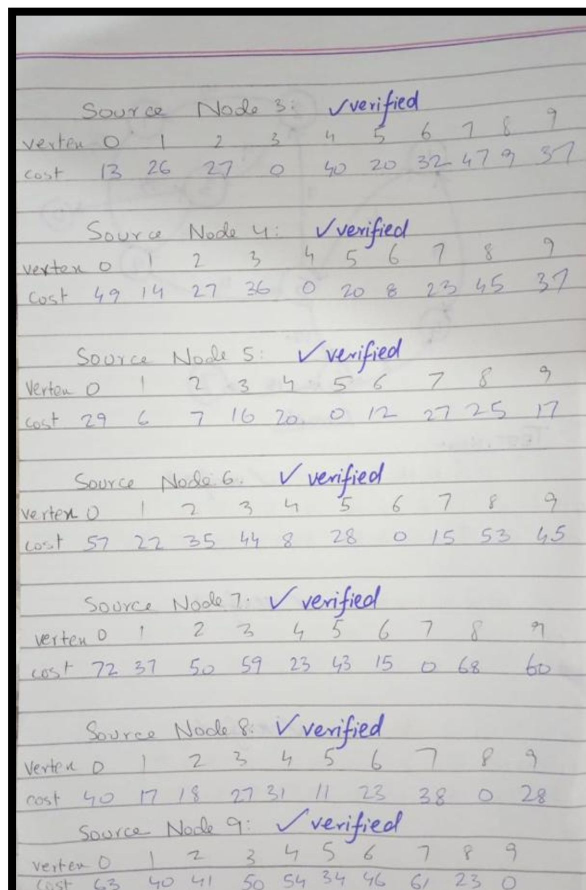
```
        0        1        2        3        4        5        6        7        8
9
0       0       -1       -1       13       -1       -1       -1       -1       -1
-1
1      -1        0       -1       -1       14        6       -1       -1       -1
-1
2      -1       -1        0        9       -1       -1       -1       -1       -1
10
3      13       -1       -1        0       -1       -1       -1       -1        9
-1
4      -1       14       -1       -1        0       -1        8       -1       -1
-1
5      -1        6        7       -1       -1        0       12       -1       -1
-1
6      -1       -1       -1       -1        8       -1        0       15       -1
-1
7      -1       -1       -1       -1       -1       -1       15        0       -1
-1
8      -1       -1       -1       -1       -1       11       -1       -1        0
-1
9      -1       -1       -1       -1       -1       -1       -1       -1       23
0

 Cost to vertices:          29        6        7       16       20        0       12
27      25       17
```

Testing:

Following are calculated costs to all vertices when each vertex is considered initial node (source node) respectively:





Resources:

Following resources have been consulted to better understand Dijkstra's algorithm:

- Class lecture.
- Data Structure using C by Reema Theraja, 2nd Edition.

THE END