Illinois Institute of Technology

CAD Techniques VLSI techniques

Home Work 1

Ishan Loganathan Reddy

CWID : A20472779

Date: 16th September,2021
Professor : Kyuwon Choi

Acknowledgment: I acknowledge all of the work (including figures and codes) belongs to me and/or persons who are referenced.

Sign : Ishan Loganathan Reddy

INTRODUCTION:

Graph Theory Algorithm:

Definition:
A graph G = (V, E) is a set V of vertices and a set E of edges. Each edge $e \in E$ is associated with two vertices u and v from V , and we write e = (u, v). We say that u is adjacent to v, u is incident to v, and u is a neighbor of v. Graphs are a common abstraction to represent data.

In this assignment we implement 7 types of different algorithms in graph theory. They are namely:

- Breadth First Search
- Depth First Search
- Djikstra's Algorithm
- Kruskal's Algorithm
- Bellman-Ford Algorithm
- Prim's Algorithm
- Floyd-Warshall's Algorithm

Storing a graph:
How do we actually store a graph on a computer?
We use one of two methods: An adjacency matrix or an adjacency list. In the adjacency matrix, we store a matrix A of size |V |×|V | with entry Aij = 1 if edge (i, j) is in the graph, and Aij = 0 otherwise.
For this assignment we focus on the adjacency matrix method. The adjacency matrix of a graph G is a V ⇥ V matrix of 0s and 1s, normally represented by a two-dimensional array A[1 .. V, 1 .. V], where each entry indicates whether a particular edge is present in G.

Graph Exploration:
We can say that node u is accessible by node v if there is an existing path between the two nodes. A walk in an undirected graph G is a sequence of vertices, where each adjacent pair of vertices are adjacent in G; informally, we can also think of a walk as a sequence of edges. A walk is called a path if it visits each vertex at most once. For any two vertices u and v in a graph G, we say that v is reachable from u if G contains a walk (and therefore a path) between u and v.
Directed graphs require slightly different definitions. A directed walk is a sequence of vertices v0v1v2···v` such that vi1vi is a directed edge for every index i.

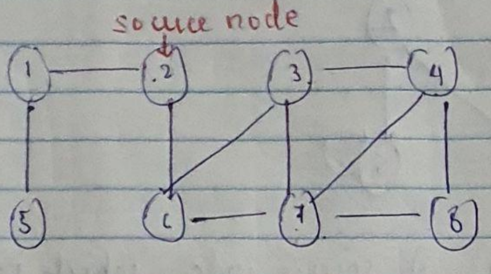
Breadth First Search Algorithm:
In breadth first search, we start with the neighbor of the source node, and once conditions are satisfied, we move on to the neighbors of the neighbors. We implement a queue function in this method, this is an object of the algorithm that supports two operations:

- enqueue() = Puts the data element x of the current node to the back of the queue.
- dequeue() = Returns the oldest value of the queue when this operation is executed.

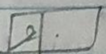
Basic Algorithm:

```
for v ∈ V do
    dist[v] ← ∞
end
dist[s] ← 0
Initialize queue Q
Q.enqueue(s)
while Q is not empty do
    u ← Q.dequeue()
    for (u, v) ∈ E do
        if dist[v] is ∞ then
        Q.enqueue(v)
        dist[v] ← dist[u] + 1
    end
end
End
```

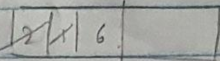
1. <u>Breadth - First - Search.</u> (BFS)



Step 0 : 2

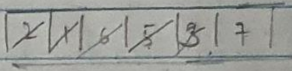


<u>Step 1</u>: neighbour of 2 : 1, 6 0



∴ order = 2, 1, 6

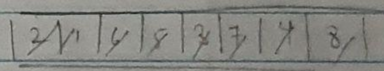
<u>Step 2</u>: neighbour of 1 = 5, 6 = 3, 7.



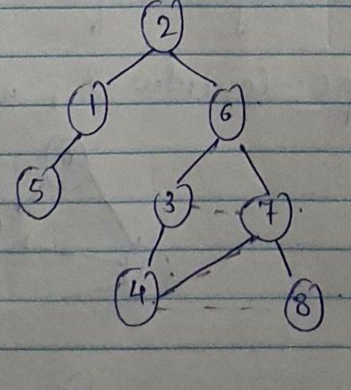
∴ order = 2, 1, 6, 5, 3, 7.

<u>Step 3</u>: neighbour of 3 = 4, 7. } 7 = 4, 8.

∴ order = 2, 1, 6, 5, 3, 7, 4, 8



The tree looks as follows: (BFS spanning tree)



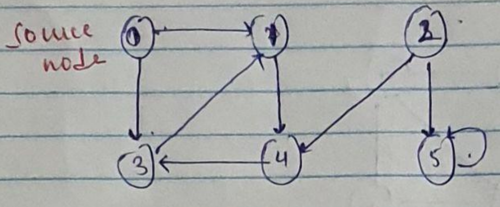
Depth First Search Algorithm:

In depth first search we explore a graph by starting at a vertex and then following a path of unexplored vertices away from this vertex as far as possible. We repeat this procedure until all the nodes are explored(visited). This algorithm uses a stack methodology, which supports insertions and deletions in every iteration or exploring cycle. The spanning tree formed by the parent-edges make the depth-first spanning tree. This algorithm begins at the root node. So the basic idea is to start from the root or any arbitrary node and mark the node and move to the adjacent unmarked node and continue this loop until there is no unmarked adjacent node. Then backtrack and check for other unmarked nodes and traverse them. Finally, print the nodes in the path.

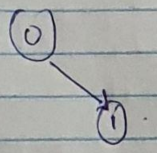
Basic Algorithm:

```
procedure DFS(G, v) is
    label v as discovered
    for all directed edges from v to w that are in G.adjacentEdges(v) do
        if vertex w is not labeled as discovered then
            recursively call DFS(G, w).
```

2. <u>Depth - first - Search</u> : <u>(DFS)</u>

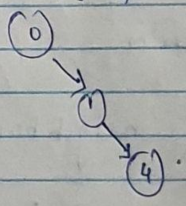


<u>step1:</u> consider neighbour of source code which is 1, & 3 we consider 1 here.



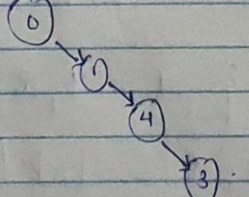
<u>step2:</u> Consider neighbour for 1 which is 4

∴



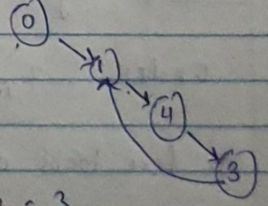
<u>Steps:</u> Consider neighbour for 4 = 3.

∴



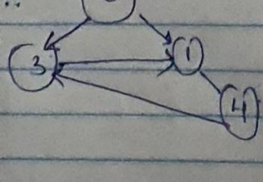
<u>step4:</u> consider neighbour for 3 which is 1.

∴

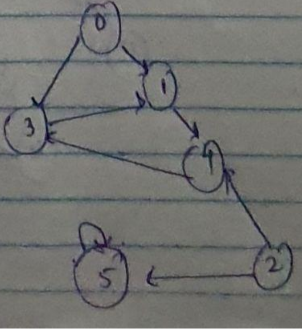


<u>Steps:</u> consider the other neighbour of 0 = 3.

∴



<u>Step6:</u> Consider 2, neigbours are 5.



Djikstra's Algorithm:
The graph can either be directed or undirected with the condition that the graph needs to embrace a non-negative value on every edge. An algorithm that is used for finding the shortest distance, or path, from starting node to target node in a weighted graph is known as Dijkstra's Algorithm.
This algorithm makes a tree of the shortest path from the starting node, the source, to all other nodes in the graph. It is different from the minimum spanning tree as the shortest distance among two vertices might not involve all the vertices of the graph.
Hence, if there are any weights present on the negative edge of the graph this algorithm would not run properly. Dijkstra's algorithm works on the principle of relaxation where an approximation of the accurate distance is steadily displaced by more suitable values until the shortest distance is achieved.
<u>Basic Algorithm:</u>

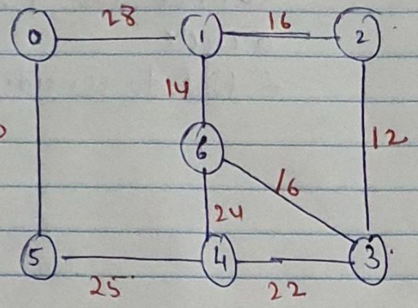
```
Algorithm
1) Create a shortest path tree set that keeps track of vertices included in the shortest-path tree, i.e., whose minimum distance from the source is calculated and finalized. Initially, this set is empty.
2) Assign a distance value to all vertices in the input graph. Initialize all distance values as INFINITE. Assign distance value as 0 for the source vertex so that it is picked first.
3) While the shortest path tree set doesn't include all vertices
a) Pick a vertex u which is not there in the shortest path tree set and has a minimum distance value.
b) Include u to the shortest path tree set.
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 the sum of distance value of u and weight of edge u-v, is less than the distance value of v, then update the distance value of v.
```

Kruskal's Algorithm:
This algorithm finds a minimum spanning tree of an undirected weighted graph. A MST of a connected graph is a subset of edges that form the tree that includes every vertex. In this, we must note that the sum of the weights of all the edges in the tree is minimized.

<u>Basic Algorithm:</u>

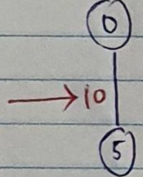
```
Kruskal(G) is
  F:= ∅
  for each v ∈ G.V do
    MAKE-SET(v)
  for each (u, v) in G.E ordered by weight(u, v), increasing do
    if FIND-SET(u) ≠ FIND-SET(v) then
      F:= F ∪ {(u, v)} ∪ {(v, u)}
      UNION(FIND-SET(u), FIND-SET(v))
  return F
```

5. **Krushkal's Algorithm.**

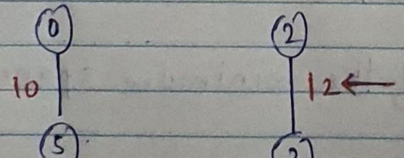


No. of ~~edges~~ vertices = 7.
∴ No. of edges for spanning tree = 7 − 1
= 6.

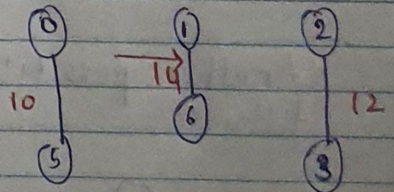
Step1: Consider the edge with the lowest weight. Here it is ~~(1,2)~~ (0,5) = 10. ∴ we add it to our tree.



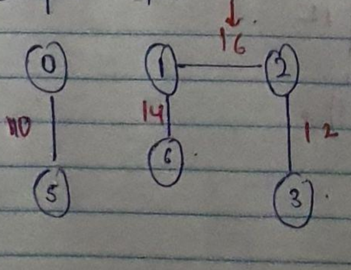
step2: (2,3) = 12.



step3: (1,6) = 14

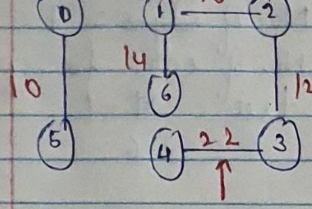


step4: (1,2) = 16



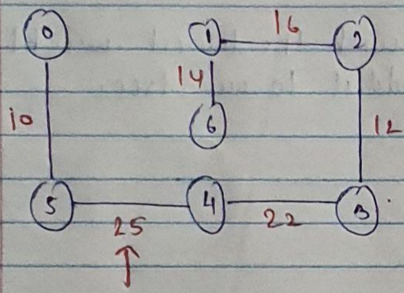
steps: (1,3) = 16, we cannot add this since it will form a a cycle.
∴ Do not consider edge (1,3)

Step5: (4, 3) = 22.



Step6: (6, 4) = 24
Cannot add this as it will form a cycle.
∴ Not to consider (6, 4)

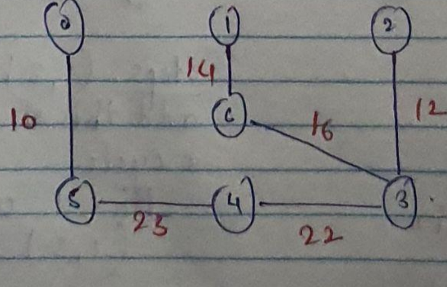
Step7: (5, 4) = 25.



~~This is~~

Above is the final minimum spanning tree.
Total cost = 10 + 12 + 14 + 16 + 22 + 25
= 99

Another possibility of the minimum spanning tree is as follows:



whose cost is also 99.

Bellman-Ford Algorithm:
The bellman-ford algorithm is an algorithm that computes the shortest paths from a single source vertex to all the other vertices in a weighted graph. It is slower than Dijkstra's algorithm for the same problem, but more versatile, as it is capable of handling graphs in which some of the edge weights are negative numbers.
Method - Bellman–Ford proceeds by relaxation, in which approximations to the correct distance are replaced by better ones until they eventually reach the solution.

<u>Basic Algorithm:</u>

```
function BellmanFord(list vertices, list edges, vertex source) is
    distance := list of size n
    predecessor := list of size n

    // Step 1: initialize graph
    for each vertex v in vertices do

        distance[v] := inf             // Initialize the distance to all vertices to infinity
        predecessor[v] := null         // And having a null predecessor

    distance[source] := 0              // The distance from the source to itself is, of course, zero

    // Step 2: relax edges repeatedly

    repeat |V|−1 times:
        for each edge (u, v) with weight w in edges do
            if distance[u] + w < distance[v] then
                distance[v] := distance[u] + w
                predecessor[v] := u

    // Step 3: check for negative-weight cycles
    for each edge (u, v) with weight w in edges do
        if distance[u] + w < distance[v] then
            error "Graph contains a negative-weight cycle"

    return distance, predecessor
```

Floyd-Warshall Algorithm:
This algorithm's single execution will find the lengths (summed weights) of shortest paths between all pairs of vertices. The Floyd–Warshall algorithm compares all possible paths through the graph between each pair of vertices. The Floyd-Warshall algorithm, also variously known as Floyd's algorithm, is an algorithm for efficiently and simultaneously finding the shortest paths between every pair of vertices in a weighted and potentially directed graph.

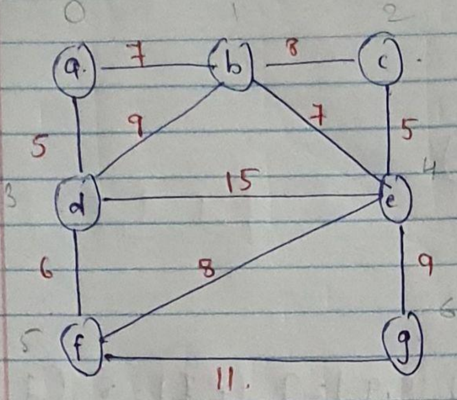
<u>Basic Algorithm:</u>

```
let dist be a |V| × |V| array of minimum distances initialized to ∞ (infinity)
for each edge (u, v) do
  dist[u][v] ← w(u, v)  // The weight of the edge (u, v)
for each vertex v do
  dist[v][v] ← 0
for k from 1 to |V|
  for i from 1 to |V|
    for j from 1 to |V|
      if dist[i][j] > dist[i][k] + dist[k][j]
        dist[i][j] ← dist[i][k] + dist[k][j]
      end if
```

Prim's Algorithm:
The algorithm operates by beginning this tree one vertex at a time.
It selects a random node at each step adding the cheapest possible connection from the tree to another vertex.
<u>Basic Algorithm:</u>

- Initialize a tree with a single vertex, chosen arbitrarily from the graph.
- Grow the tree by one edge: of the edges that connect the tree to vertices not yet in the tree, find the minimum-weight edge, and transfer it to the tree.
- Repeat step 2 (until all vertices are in the tree).

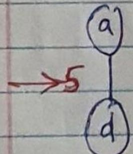
4. Prims Algorithm.



No of vertices = 7.
No of edges for minimum spanning tree = 7 - 1
= 6

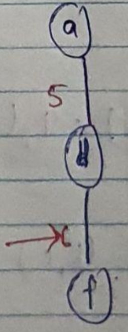
Consider any node, as the parent Node. Here we consider a as the root node.

Step1: (a,d) = 5.



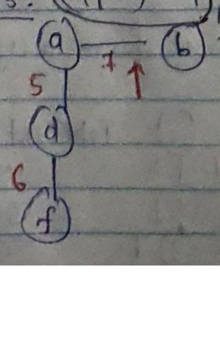
{a, d}

(d,e) = 15
Step2: (a,b) = 7, (d,b) = 9, $\boxed{(d,f) = 6}$
[Consider all the neighbours of a, b & choose one that is the least]



{a, d, f}

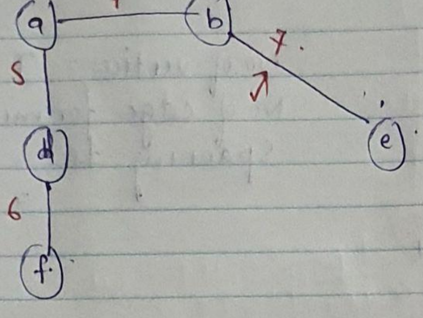
Step3: $\boxed{(a,b) = 7}$, (d,b) = 9), (d,e) = 15, (f,e) = 8, (f,g) = 11



{a, b, d, f}

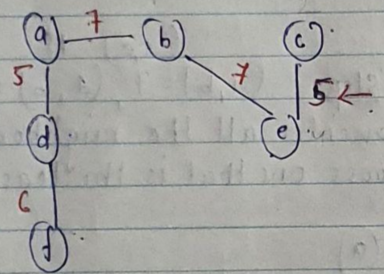
a, b, d, f

Step 4: (b, c) = 8, (b, e) = 7, (d, b) = 9, (d, e) = 15, (f, e) = 8, (f, g) = 11



{a, b, d, f, e}

Step 5: (b, c) = 8, (e, c) = 5, (e, g) = 9, (e, f) = 8, (d, b) = 9,
(f, g) = 11



{a, b, c, d, e, f}

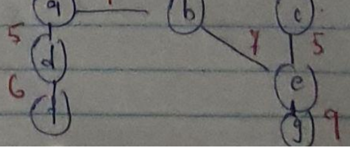
Step 6: ~~(b, c) = 8~~, ~~(b, d)~~ = 9, (e, d) = 15, (e, g) = 9, ~~(e, f) = 8~~,
(f, g) = 11

Step 7: cannot select (b, c) = 8 : forms a cycle.

Step 8: (e, f) = 8 ⇒ cannot select as it forms a cycle.

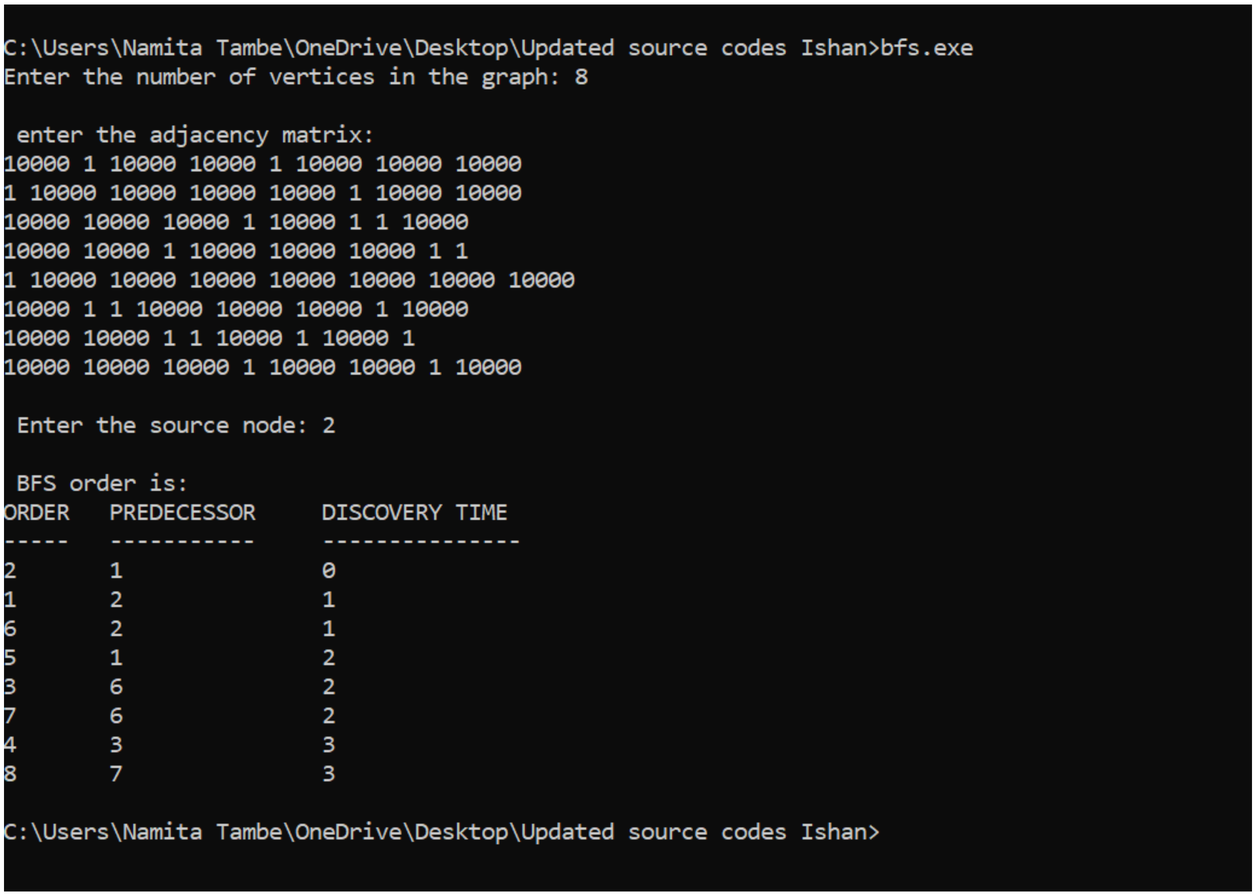
Step 9: (b, d) = 9 ⇒ cannot select as it forms a cycle.

Step 10: (e, g) = 9.



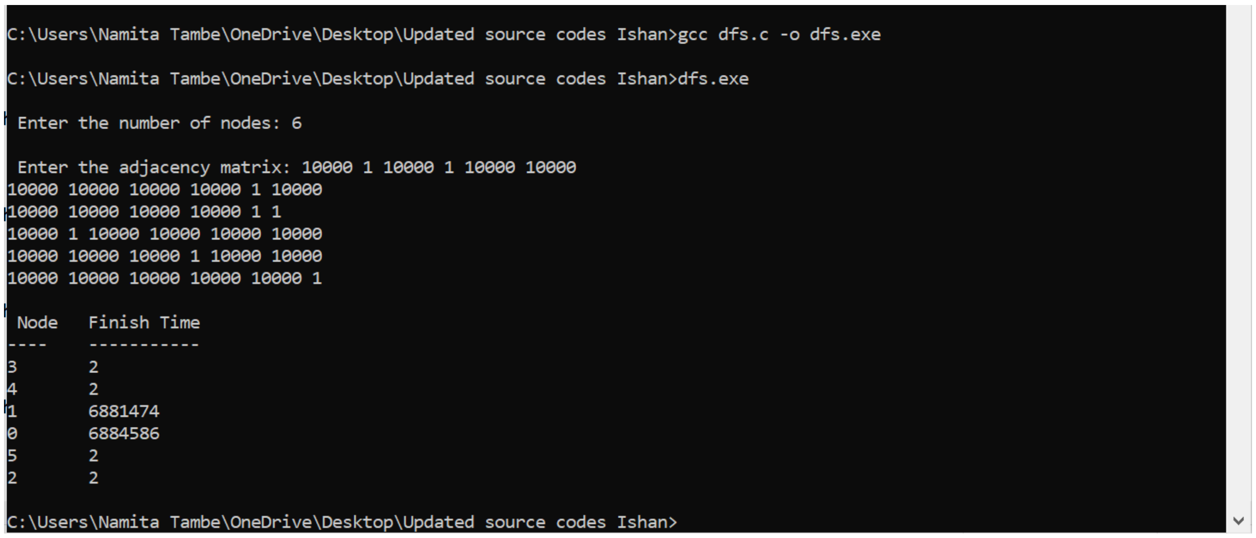
Total cost = 5 + 6 + 7 + 7
+ 5 + 9
= $\underline{39}$

OUTPUTS:



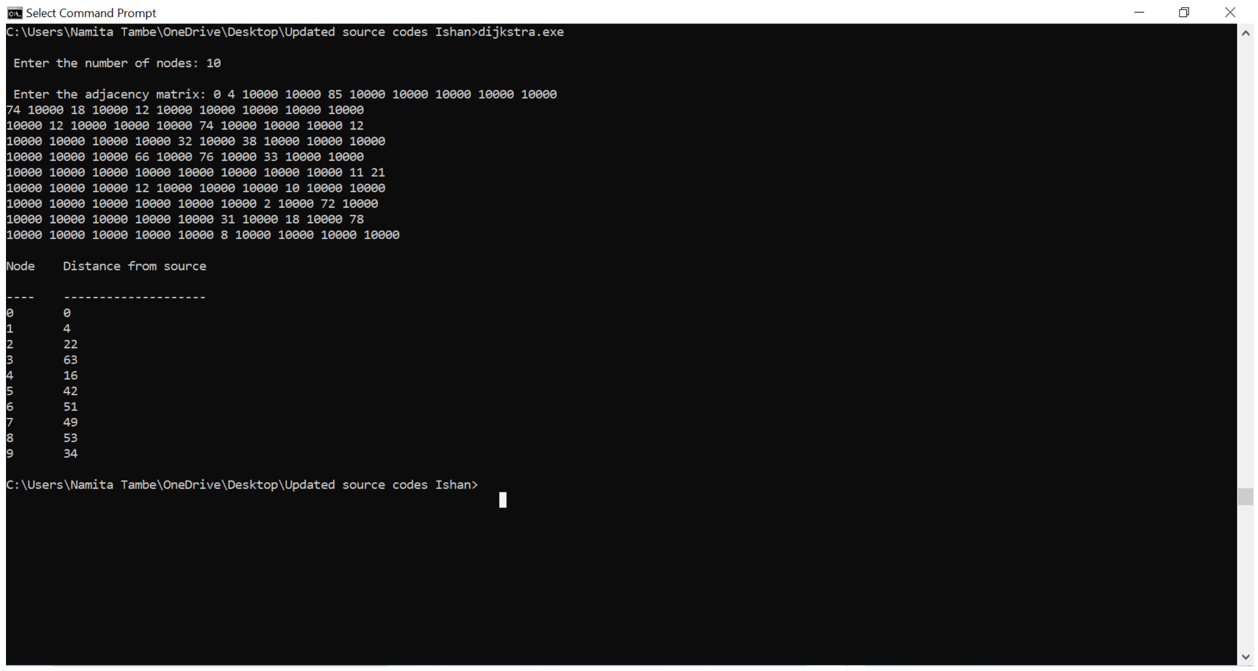
```
C:\Users\Namita Tambe\OneDrive\Desktop\Updated source codes Ishan>bfs.exe
Enter the number of vertices in the graph: 8

 enter the adjacency matrix:
10000 1 10000 10000 1 10000 10000 10000
1 10000 10000 10000 10000 1 10000 10000
10000 10000 10000 1 10000 1 1 10000
10000 10000 1 10000 10000 10000 1 1
1 10000 10000 10000 10000 10000 10000 10000
10000 1 1 10000 10000 10000 1 10000
10000 10000 1 1 10000 1 10000 1
10000 10000 10000 1 10000 10000 1 10000

 Enter the source node: 2

 BFS order is:
ORDER   PREDECESSOR     DISCOVERY TIME
-----   -----------     ---------------
2       1               0
1       2               1
6       2               1
5       1               2
3       6               2
7       6               2
4       3               3
8       7               3

C:\Users\Namita Tambe\OneDrive\Desktop\Updated source codes Ishan>
```

Figure - bfs.c output



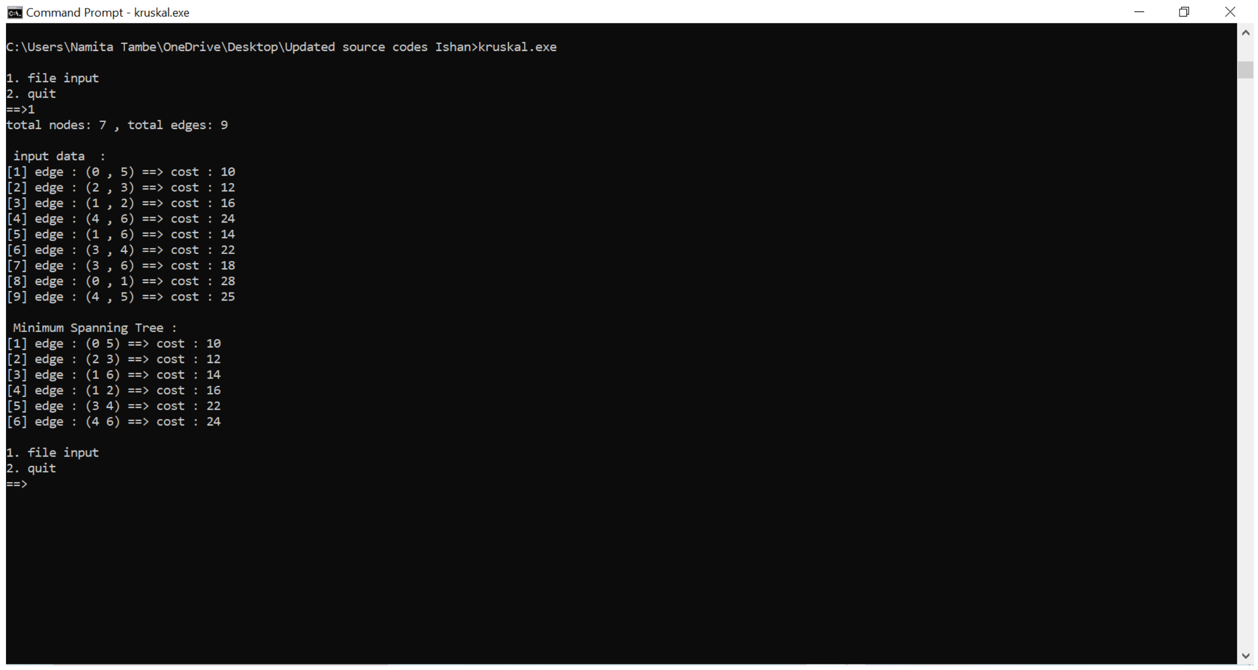
```
C:\Users\Namita Tambe\OneDrive\Desktop\Updated source codes Ishan>gcc dfs.c -o dfs.exe

C:\Users\Namita Tambe\OneDrive\Desktop\Updated source codes Ishan>dfs.exe

 Enter the number of nodes: 6

 Enter the adjacency matrix: 10000 1 10000 1 10000 10000
10000 10000 10000 10000 1 10000
10000 10000 10000 10000 1 1
10000 1 10000 10000 10000 10000
10000 10000 10000 1 10000 10000
10000 10000 10000 10000 10000 1

 Node   Finish Time
----    -----------
3       2
4       2
1       6881474
0       6884586
5       2
2       2

C:\Users\Namita Tambe\OneDrive\Desktop\Updated source codes Ishan>
```

Figure - dfs.c output



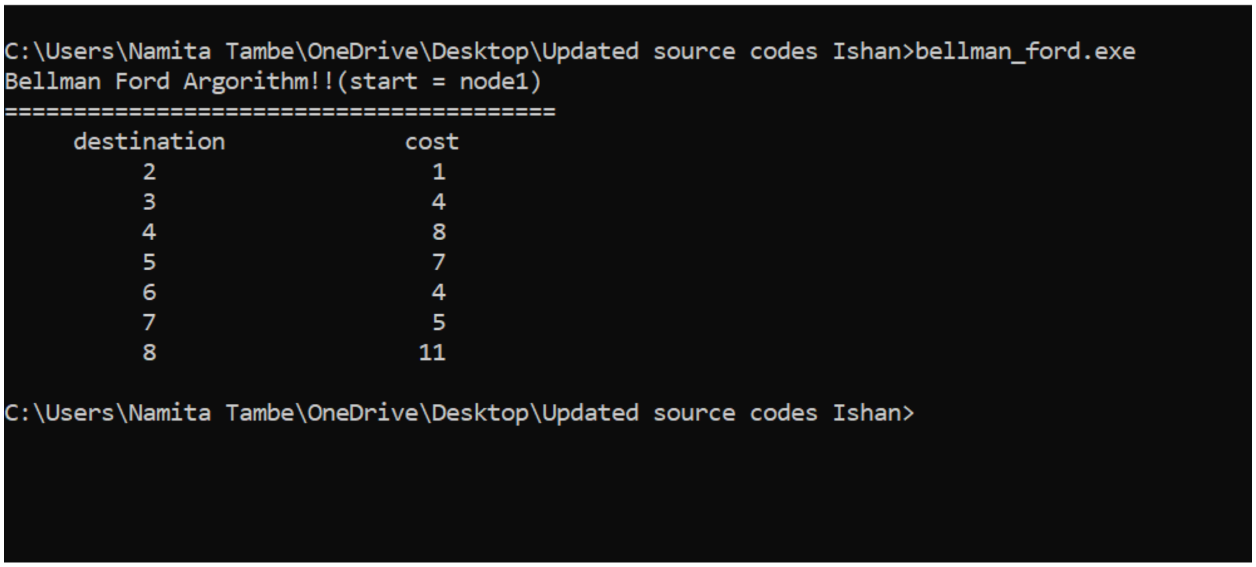
```
C:\Users\Namita Tambe\OneDrive\Desktop\Updated source codes Ishan>dijkstra.exe

 Enter the number of nodes: 10

 Enter the adjacency matrix: 0 4 10000 10000 85 10000 10000 10000 10000 10000
74 10000 18 10000 12 10000 10000 10000 10000 10000
10000 12 10000 10000 10000 74 10000 10000 10000 12
10000 10000 10000 10000 32 10000 38 10000 10000 10000
10000 10000 10000 66 10000 76 10000 33 10000 10000
10000 10000 10000 10000 10000 10000 10000 10000 11 21
10000 10000 10000 12 10000 10000 10000 10 10000 10000
10000 10000 10000 10000 10000 10000 2 10000 72 10000
10000 10000 10000 10000 10000 31 10000 18 10000 78
10000 10000 10000 10000 10000 8 10000 10000 10000 10000

Node    Distance from source

----    -------------------
0       0
1       4
2       22
3       63
4       16
5       42
6       51
7       49
8       53
9       34

C:\Users\Namita Tambe\OneDrive\Desktop\Updated source codes Ishan>
```

Figure - djikstra.c output



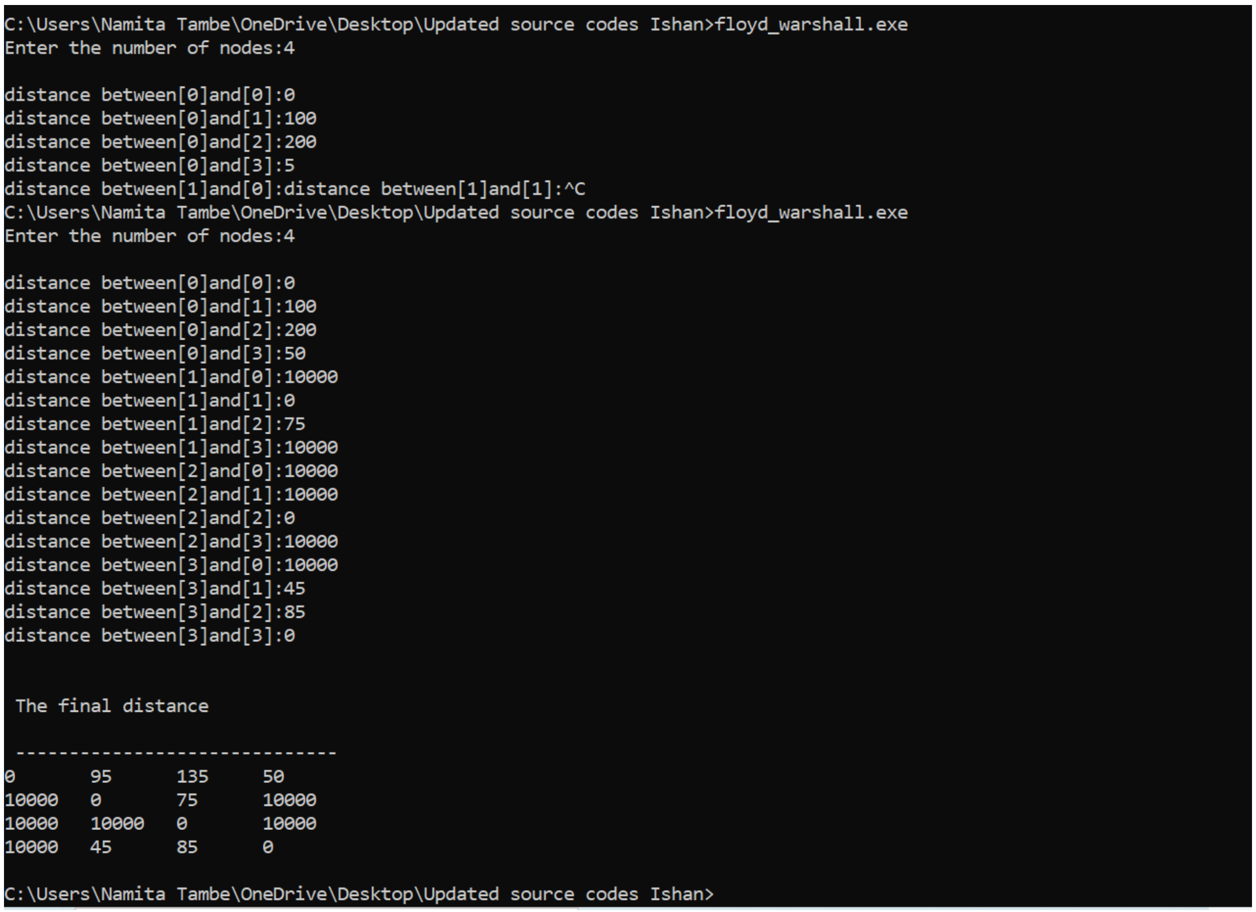
```
C:\Users\Namita Tambe\OneDrive\Desktop\Updated source codes Ishan>kruskal.exe

1. file input
2. quit
==>1
total nodes: 7 , total edges: 9

 input data  :
[1] edge : (0 , 5) ==> cost : 10
[2] edge : (2 , 3) ==> cost : 12
[3] edge : (1 , 2) ==> cost : 16
[4] edge : (4 , 6) ==> cost : 24
[5] edge : (1 , 6) ==> cost : 14
[6] edge : (3 , 4) ==> cost : 22
[7] edge : (3 , 6) ==> cost : 18
[8] edge : (0 , 1) ==> cost : 28
[9] edge : (4 , 5) ==> cost : 25

 Minimum Spanning Tree :
[1] edge : (0 5) ==> cost : 10
[2] edge : (2 3) ==> cost : 12
[3] edge : (1 6) ==> cost : 14
[4] edge : (1 2) ==> cost : 16
[5] edge : (3 4) ==> cost : 22
[6] edge : (4 6) ==> cost : 24

1. file input
2. quit
==>
```

Figure - kruskal.c output



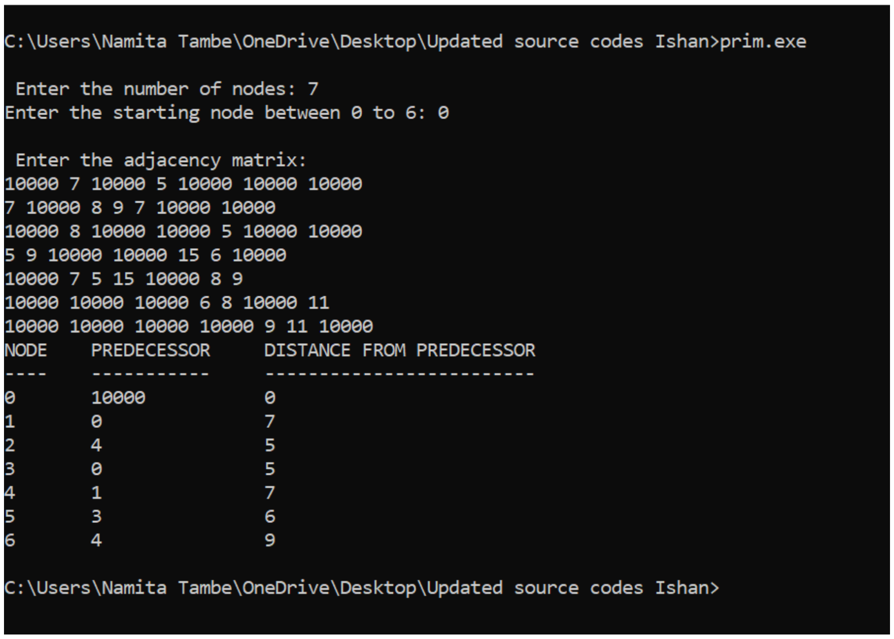
```
C:\Users\Namita Tambe\OneDrive\Desktop\Updated source codes Ishan>bellman_ford.exe
Bellman Ford Argorithm!!(start = node1)
========================================
     destination            cost
          2                   1
          3                   4
          4                   8
          5                   7
          6                   4
          7                   5
          8                  11

C:\Users\Namita Tambe\OneDrive\Desktop\Updated source codes Ishan>
```

Figure - bellman-ford.c output



```
C:\Users\Namita Tambe\OneDrive\Desktop\Updated source codes Ishan>floyd_warshall.exe
Enter the number of nodes:4

distance between[0]and[0]:0
distance between[0]and[1]:100
distance between[0]and[2]:200
distance between[0]and[3]:5
distance between[1]and[0]:distance between[1]and[1]:^C
C:\Users\Namita Tambe\OneDrive\Desktop\Updated source codes Ishan>floyd_warshall.exe
Enter the number of nodes:4

distance between[0]and[0]:0
distance between[0]and[1]:100
distance between[0]and[2]:200
distance between[0]and[3]:50
distance between[1]and[0]:10000
distance between[1]and[1]:0
distance between[1]and[2]:75
distance between[1]and[3]:10000
distance between[2]and[0]:10000
distance between[2]and[1]:10000
distance between[2]and[2]:0
distance between[2]and[3]:10000
distance between[3]and[0]:10000
distance between[3]and[1]:45
distance between[3]and[2]:85
distance between[3]and[3]:0


 The final distance

 ------------------------------
0       95      135     50
10000   0       75      10000
10000   10000   0       10000
10000   45      85      0

C:\Users\Namita Tambe\OneDrive\Desktop\Updated source codes Ishan>
```

Figure - Floyd-warshall.c output



```
C:\Users\Namita Tambe\OneDrive\Desktop\Updated source codes Ishan>prim.exe

 Enter the number of nodes: 7
Enter the starting node between 0 to 6: 0

 Enter the adjacency matrix:
10000 7 10000 5 10000 10000 10000
7 10000 8 9 7 10000 10000
10000 8 10000 10000 5 10000 10000
5 9 10000 10000 15 6 10000
10000 7 5 15 10000 8 9
10000 10000 10000 6 8 10000 11
10000 10000 10000 10000 9 11 10000
NODE    PREDECESSOR     DISTANCE FROM PREDECESSOR
----    -----------     -------------------------
0       10000           0
1       0               7
2       4               5
3       0               5
4       1               7
5       3               6
6       4               9

C:\Users\Namita Tambe\OneDrive\Desktop\Updated source codes Ishan>
```

Figure - Prim.c output