Tree using DFS (Assignment 5)

Group 7 : Anshul Agarwal *(IWM2017008)*, Parth Jain *(ICM2017006)*

***Abstract*—In this paper we will suggest an algorithm to derive a Spanning tree from the given adjacency and incidence matrix of a graph. Only depth first search (DFS) traversal can be used on the given graph.**

I. Keywords

Depth First Search, Spanning tree, Adjacency matrix, Incidence matrix, Traversal, Graph, Edges, Nodes, Weighted edges.

II. Introduction

A Graph is a mathematical structure to represent relationships between different sets of objects. The objects are symbolised by the vertices and the relationships by the edges. In graph theory , a graph G, is a pair **(V,E)**, where

- V, is a set elements called as vertices .
- $E \subset (x,y)|(x,y)\epsilon V^2$, called as edges.

A. Un-directed Graph

An undirected graph has edges with no direction(or bidirectional edges). If two vertices A and B are neighbors, then we can move from A to B and also from B to A. To highlight their usage in representing real life objects we can take the example of highways that allows bidirectional traffic. We can even consider the friendship relation on facebook. If person A is friend of person B then B is also friend of A.

B. Weighted Graph

If there is a weight or cost associated with every edge of a graph, we call that graph a weighted graph. Weighted graphs are extensively used to represent real world problems. For example, for a road connecting two cities, the weight of that road can represent the distance between the cities.

C. Adjacency matrix representation

An adjacency matrix for a finite graph is a square matrix that represents whether a pair of graph vertices are adjacent or not. By normal convention, it is a binary matrix and if two vertices are adjacent their corresponding cell will be 1 otherwise 0.

D. Incidence matrix representation

The incidence matrix of an undirected graph is an n m matrix I, where n and m are the numbers of vertices and edges respectively, such that I(i,j) = 1 if the vertex Vi and edge Ej are incident and 0 otherwise. The incidence matrix of a directed graph is an n m matrix I where n and m are the number of vertices and edges respectively, such that I(i,j) = 1 if the edge Ej leaves vertex Vi, I(i,j) = 1 if it enters vertex Vi and 0 otherwise.

E. Connected Components

Connected component of a graph is a subgraph in which any two vertices are connected with each other via some path. Also any vertex of this subgraph is not connected to a vertex that is part of the supergraph but not the subgraph.

F. Depth First Search Traversal

Depth First Search (DFS) algorithm traverses a graph in a depthward motion and uses a stack to remember to get the next vertex to start a search, when a dead end occurs in any iteration.

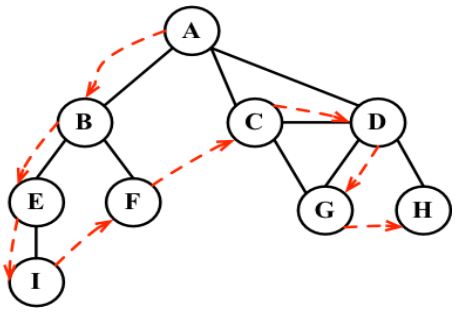


Fig. 1. Depth First Search Traversal of a graph

G. Spanning Tree

A Graph A is called a spanning tree of Graph B if A does not contain a cycle and A contains all vertices of B is a subset of vertex set and minimum possible number of edges. Only a connected graph can have a spanning tree.

H. Minimum Spanning Tree (MST)

The minimum spanning tree (MST) of a weighted graph is the spanning tree which has the minimum sum of weights of the edges. A graph can have multiple MSTs , however all of them should have the minimum sum.

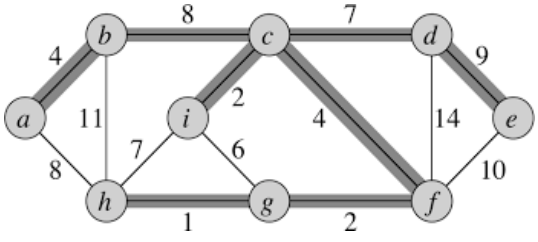


Fig. 2. the darkened subgraph represents a MST of the supergraph

III. ALGORITHM DESIGN AND EXPLANATION

Our approach to given problem for Adjacency Matrix input includes the following steps:

1. We are choosing an arbitrary vertex from the graph as a root of the spanning tree.

2. Then, we are successively adding vertices and edges in the path such that vertices should not be present in the path and the new edge is adjacent to the last vertex added in the path. We will keep adding the vertices and edges as long as possible.

3. A spanning tree must contain all the vertices, so once encountered all the vertices we stop.

4. If not, then we will backtrack the path and will search for the next to last vertex whether there is a possibility of a new path adjacent to it and is not included in the path yet.

5. If there is no possibility of new edge then we will move back to another vertex and repeat the same procedure again.

6. We will continue this procedure until all the edges will get covered.

The pseudo-code for this algorithm is:-

Algorithm 1 Input Adjacency Matrix

```
Nodes ← Input                                   ▷ t ← t+1
Edges ← Input                                   ▷ t ← t+1
while i < edges do                              ▷ t ← t+1
    InputVertex(u)                              ▷ t ← t+1
    InputVertex(v)                              ▷ t ← t+1
    adjacency[u][v] = 1                         ▷ t ← t+1
    adjacency[v][u] = 1                         ▷ t ← t+1
end while
```

Algorithm 2 Spanning tree from adjacency matrix

```
Vertex v ← Input Arbitrary Vertex               ▷ t ← t+1
Visited[Vertex v] ← True                        ▷ t ← t+1
Call FunctionOne(Vertex v)                      ▷ t ← t+1
```

Algorithm 3 FunctionOne

```
while u < numberofvertex do                     ▷ t ← t+1
    if vertex u adjacent to vertex v then       ▷ t ← t+1
        if Visited[u] == false then             ▷ t ← t+1
            Visited[Vertex u] ← True            ▷ t ← t+1
            Add Edge from v to u                ▷ t ← t+1
            CallFunctionOne(vertex u)           ▷ t ← t+1
        else                                    ▷ t ← t+1
            CallFunctionOne(vertex v)           ▷ t ← t+1
        end if
    end if
end while
```

IV. ALGORITHM DESIGN AND EXPLANATION

Our approach to given problem for Incidence Matrix input includes the following steps:

1. We are choosing an arbitrary vertex, say V_A from the graph as a root of the spanning tree.

2. Then, we are choose one of the edges that are incident on the vertex, and lets call the other vertex end of this edge V_B. So, V_B is not already present in the path, we add it and the corresponding edge to the path. We will keep adding the vertices and edges as long as possible.

3. A spanning tree must contain all the vertices, so once encountered all the vertices we stop.

4. If not, then we will backtrack the path and will search for the next to last edge considered whether there is a possibility of a new vertex adjacent to it and is not included in the path yet. If there is no possibility of new vertex then we will move back to another edge and repeat the same procedure again.

5. We will continue this procedure until all the vertices will get covered.

The pseudo-code for this algorithm is:-

Algorithm 4 Input Incidence Matrix

```
Nodes ← Input                                   ▷ t ← t+1
Edges ← Input                                   ▷ t ← t+1
while i < edges do                              ▷ t ← t+1
    InputVertex(u)                              ▷ t ← t+1
    InputVertex(v)                              ▷ t ← t+1
    incidence[u][i] = 1                         ▷ t ← t+1
    incidence[v][i] = 1                         ▷ t ← t+1
end while
```

V. ALGORITHM ANALYSIS

A. For Adjacency Matrix

In algorithm 2, an arbitrary vertex is chosen as a root of the spanning tree. It will be marked as visited in the boolean visited array. After that calling the FunctionOne funtion to look for other vertexes. The time complexity is O(1), where calling a another function and marking a vertex visited is constant. Then we look for all the vertexes adjacent to the root vertex and will check whether the new vertex is already visited or not. If it is not visited then we will add the edge in the spanning tree. To get this other vertex for an edge, we are using the function FunctionOne defined in algorithm 3. The time complexity of this algorithm is O(V*V).

B. For Incidence Matrix

In algorithm 5, we choose an arbitrary vertex to be the root of the spanning tree (ST). Then we consider all the edges incident on it, and check whether the other vertex for that

Algorithm 5 Spanning Tree from Incidence Matrix

```
Vertex v ← Input Arbitrary Vertex                    ▷ t ← t+1
Visited[v] ← True                                    ▷ t ← t+1
Stack < int > s ← Temporary stack                    ▷ t ← t+1
set < int > ST_edges ← For edges of ST               ▷ t ← t+1
s.push(v)                                            ▷ t ← t+1
while !s.empty() do                                  ▷ t ← t+1
    temp = s.top()                                   ▷ t ← t+1
    s.pop()                                          ▷ t ← t+1
    i = 0                                            ▷ t ← t+1
    while i < edges do                               ▷ t ← t+1
        if incidence[v][i] == 1 then                 ▷ t ← t+1
            vertex u = neighbor(v, i)                ▷ t ← t+1
            if Visited[u] == false then              ▷ t ← t+1
                visited[u] = true                    ▷ t ← t+1
                s.push(u)                            ▷ t ← t+1
                ST_edges.push(i)                     ▷ t ← t+1
            end if
        end if
        i = i + 1                                    ▷ t ← t+1
    end while
end while
return ST_edges
```

Algorithm 6 neighbor(vertex v, edge e) (for incidence matrix)

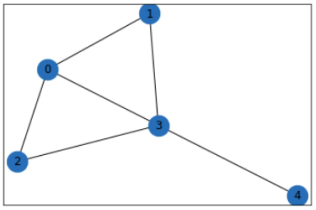
```
Vertex neighbor_vertex                               ▷ t ← t+1
i = 0                                                ▷ t ← t+1
while i < nodes do                                   ▷ t ← t+1
    if incidence[i][e] == 1 and i! = v then          ▷ t ← t+1
        neighbor_vertex = i                          ▷ t ← t+1
        break                                        ▷ t ← t+1
    end if
    i = i + 1                                        ▷ t ← t+1
end while
return neighbor_vertex
```

edge is previously visited or not. If it is not visited then this vertex is put in a stack and the corresponding edge is a part of the ST. To get this other vertex for an edge, we are using the function neighbor. neighbor is defined in algorithm 6. The time complexity of neighbor function is O(V), where V is the number of nodes or vertices in the graph. Since algorithm 5 is a DFS, and in the worst case, the function neighbor could be called O(V*E) times (where E is the number of edges), the time complexity of this algorithm is O(V*V*E).

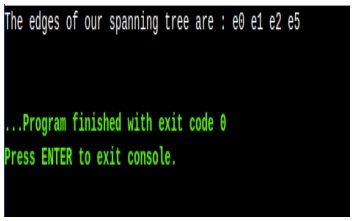
VI. Experimental Study

A. Experimental Result

Taken the input graph fig(1) and applying our algorithm on the graph and we got the spanning tree. fig(2) .
Input graph contains:
e0 = source: 0 destintion: 1
e1 = source: 0 destintion: 3
e2 = source: 0 destintion: 2
e3 = source: 2 destintion: 3
e4 = source: 1 destintion: 3
e5 = source: 3 destintion: 4



fig(1): Input Graph



fig(2): Output

B. Time Complexity

The time complexity analysis of an Algorithm is quite necessary to learn about it's efficiency and optimization.

The time complexity of an algorithm(1) for incidence matrix is-

$$T = O(E).$$

The time complexity of an algorithm(2) for incidence matrix is-

$$T = O(1).$$

The time complexity of an algorithm(3) for incidence matrix is-

$$T = O(V^2).$$

The time complexity of an algorithm(4) for incidence matrix is-

$$T = O(E).$$

The time complexity of an algorithm(5) for incidence matrix is-

$$T = O(V^2 * E).$$

The time complexity of an algorithm(6) for incidence matrix is-

$$T = O(V).$$

where V is the number of nodes or vertices in the graph and E is the number of edges.

C. Space Complexity

The Space Complexity for both adjacency and incidence matrix is-

$$O(V).$$

where V is number of nodes and E is number of Edges. This is because one boolean visited array of size V is used. The size of the stack at any moment could be the branching factor of the graph which is O(V). Also, a container is used to store the edges of the ST and the number of edges in a ST is V-1, i.e, O(V).

The Space Complexity for adjacency matrix is-

$$O(V).$$

where V is number of nodes and E is number of Edges. This is because one boolean visited array of size V is used.

VII. Conclusion

In this paper, we deduced an algorithm to find Spanning tree with the help of depth first search traversal of the given graph. The proposed algorithm uses Depth First Search, Recursion, Backtracking as the algorithm paradigms. The order of time complexity is $O(V^2 * E)$ for incidence matrix and $O(V^2)$ for adjacency matrix, where E is the number of edges in the graph and V is the number of vertexes. We hope to incorporate the use of Breadth First Search to find the spanning tree and also find the Minimum spanning tree (Spanning tree with minimum cost) using standard traversals.

References

[1] Online "https://en.wikipedia.org/wiki/Spanningtree"
[2] Online "https://www.javatpoint.com/spanning-tree"
[3] Online "https://www.programiz.com/dsa/graph-dfs"
[4] Online "https://www.tutorialspoint.com/graph_theory/graph_theory_fundamentals.html/"
[5] Online "https://en.wikipedia.org/wiki/Depth-first/search"
[6] Online "https://en.wikipedia.org/wiki/Adjacency_matrix"
[7] Online "https://www.electrical4u.com/what-is-incidence-matrix/"
[8] Online "https://www.overleaf.com"