Graph Traversal

A **traversal** is a way of exploring all of a graph's edges and vertices. A traversal is considered efficient if it runs in linear time (proportional to the number of vertices and edges).

Depth First Search (DFS) and **Breadth First Search (BFS)** are ways to traverse an undirected graph.

We will also look at ways to traverse directed graphs.

DFS for Undirected Graphs

DFS is good for:

- Finding a path from one vertex to another
- Determining if a graph is connected
- Finding a spanning tree of a graph (**Spanning Tree:** a subgraph of G that contains no cycles and all of G's vertices)

DFS Algorithm

```
DFS(v)
    mark v as being visited
    for each edge e incident to v
        if e has not been visited then
            w = e's other endpoint (i.e. not v)
            if w has not been visited then
                set e to be a "forward edge"
                DFS(w)
            else
                set e to be a "back edge"
```

N.B. e is considered “visited” if it is a forward or back edge.

Does this algorithm really visit all the vertices in a graph's component?

Yes! Suppose that there was a vertex w that wasn't visited using this DFS algorithm. Let w be the first unvisited vertex on the path from vertex v to some vertex x (where x can = w). Since w is the first unvisited vertex, there must be a vertex y that is a adjacent to w that has been visited. But by the above algorithm, when we visited y, we must have looked at the edge wy. So w must have been visited.

Does this algorithm form a spanning tree (by looking at the "forward edges")?

Yes! Since we only mark an edge as being a "forward edge" if its endpoint has not been visited, we can never form a cycle. (and we already said that it visits all of the vertices)

Run Time

DFS runs it O(n+m) time since DFS gets called n times and each edge is examined twice.

Since DFS can be used to determine:

- If G is connected
- G's spanning tree
- The number of connected components of G
- A path between two vertices

each of these can be implemented in O(n+m) time as well.

BFS for undirected graphs

Similar to DFS, but instead of traversing “deep into the graph”, it divides vertices up into “levels”.

Can be used to determine the shortest path between two vertices.

BFS Algorithm

```
BFS(v)
    i=0
    add v to vertexList[i]
    while vertexList[i] is not empty
        vertexList[i+1] = empty
        for each vertex v in vertexList[i]
            for each edge e incident to v
                if e has not been visited
                    w = endpoint of e not equal to v
                    if w has not been visited
                        set e to be a “forward edge”
                        vertexList[i+1] = vertexList[i+1] + w
                    else
                        set e to be a “cross edge”
        i=i+1
```

Things to Notice about BFS

If BFS is performed on Graph G, starting at vertex v then

- All vertices in v's component of G will be visited
- The "forward edges" result in a spanning tree (called a BFS tree)
- For each vertex x in vertexList[i], the path of the BFS tree between v and x has i edges. Any other path between v and x contains at least i edges.

Run Time

BFS runs it O(n+m) (like DFS).

Since BFS can be used to determine:

- If G is connected
- G's spanning tree
- The number of connected components of G
- The minimum number of edges (shortest path) between two vertices

each of these can be implemented in O(n+m) time as well.

Traversing Directed Graphs

A directed graph is called **strongly connected** if for any two vertices u and v in G, there exists a path from u to v and from v to u.

We can ask questions about directed graphs such as:

- Is there a path from u to v?
- Find all vertices that can be reached from vertex v
- Is G strongly connected?

A modified version of DFS can help us answer these questions.

DFS for Directed Graphs

DFS can be modified to traverse directed graphs:

```
DirectedDFS(v)
    mark v as visited
    for each outgoing edge e=(v,w)
        if w has not been visited
            DirectedDFS(w)
```

A Little More Terminology

A **Directed Acyclic Graph (DAG)** is a directed graph without cycles.

A **Topological Ordering** of G is an ordering $v_1, v_2, ..., v_n$ of all the vertices such that for each edge (v_i, v_j), i<j.

A directed graph can have a topological ordering if and only if it is acyclic.

Topological Sort Algorithm

```
TopologicalSort(G)
    Q = empty
    for each vertex u of G
        incounter(u) = in-degree(u)
        if in-degree(u) = 0
            enqueue u into Q;
        i = 1
        while Q is not empty
            u= dequeue(Q)
            u = v_i
            i = i+1
            for each outgoing edge of u
                incounter(u) = incounter(u) - 1
                if incounter(u) = 0
                    enqueue(Q, w)
```

If the topological sort algorithm doesn't number all the vertices, it means that there is no topological ordering.

This algorithm runs in O(n+m) time.