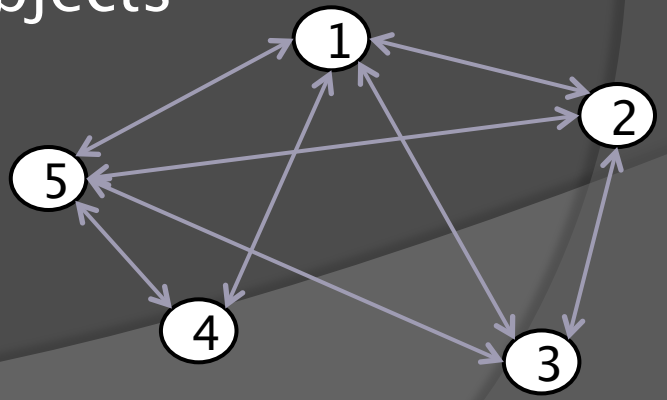
Advanced Computer Contest Preparation
Lecture 6

GRAPH TRAVERSAL (BFS/DFS)

Graphs

- Recall: What is a graph?
- A collection of:
  - vertices/nodes
  - edges (connects vertices)
- Shows relationships between objects
- “Circles connected by lines”



Graph Traversal

- A complete search of a connected portion of the graph
- A very important operation on graphs
- Examples:
  - Knight Hop (CCC 2010 J5)
    - http://wcipeg.com/problem/ccc10j5
  - Coin Game (CCC 2012 J5/S3)
    - http://wcipeg.com/problem/ccc12j5
  - Maze (CCC 2008 S3)
    - http://wcipeg.com/problem/ccc08s3
  - Spacetime Surfer (Mock CCC 2014 J5)
    - http://wcipeg.com/problem/mockccc14j5

Graph Traversal – General Idea

- Search can begin at any node
- All nodes connected to starting node will be visited
- Once visited, a node will be marked 'visited' so it will not be visited again
  - Use a `bool` array, *vis[nd]* means *nd* is visited
- Visiting order varies depending on method used

Graph Traversal – Neighbors

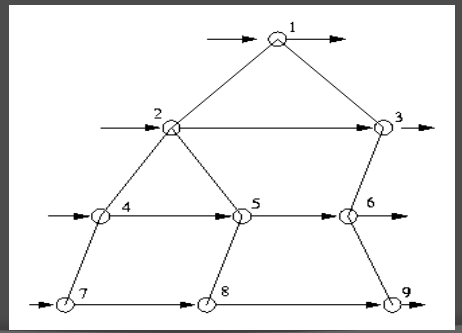
- During any graph search, we must iterate through all neighboring nodes of our current node
- Edge List: Go through all edges and check if the current edge contains the current node
- Adjacency Matrix: Iterate from $i = 1$ to N and check if $adj[current\_node][i]$ is true
- Adjacency List: Iterate through vector associated with current node

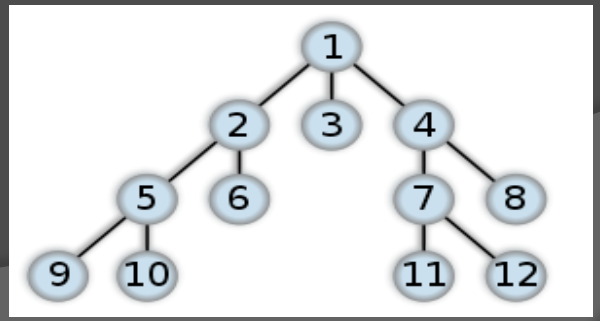
Graph Traversal – Ways

- There are 2 main ways to traverse a graph:
  - **Breadth-First Search (BFS)**
  - **Depth-First Search (DFS)**

BFS – Breadth-First Search

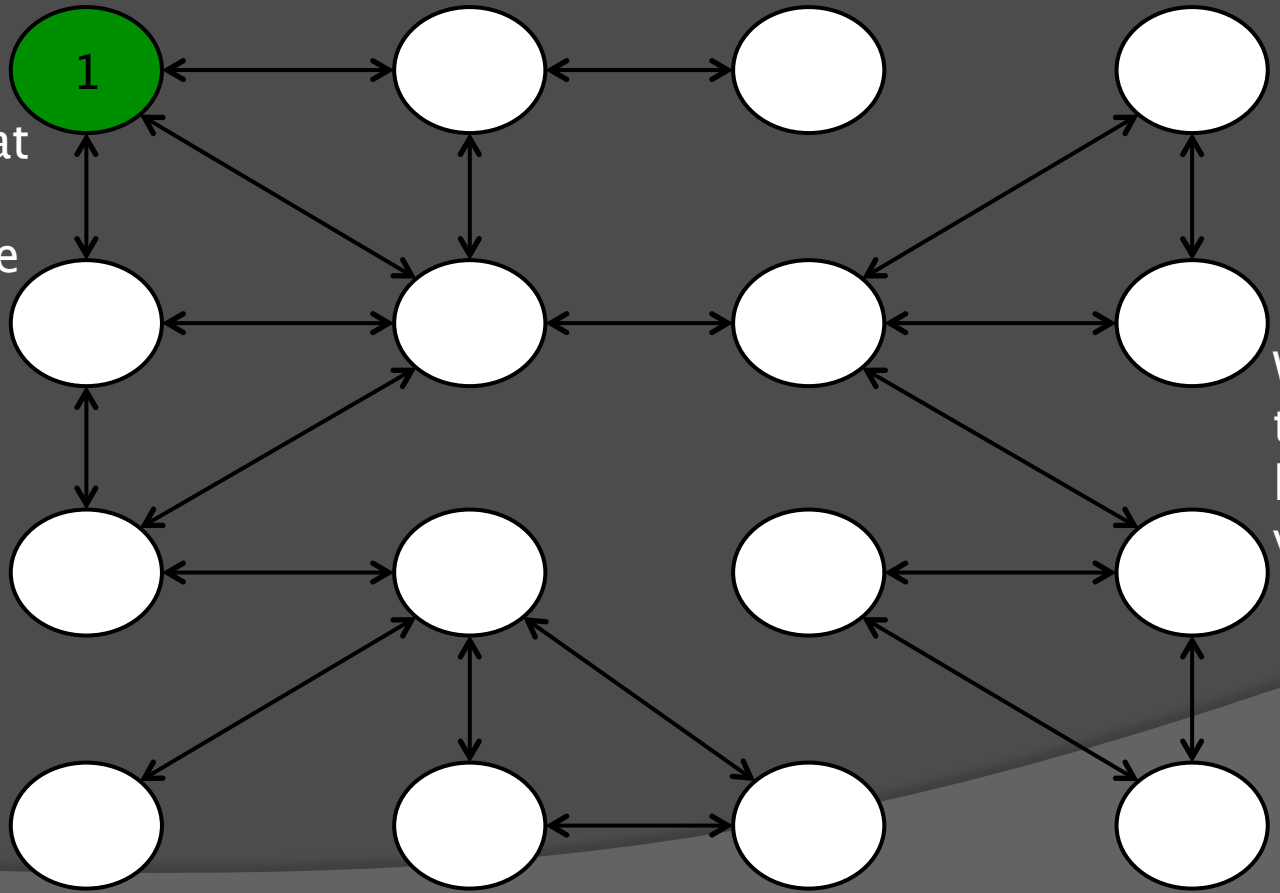
- Visits nodes depending on distance from root
- The closer the node, the sooner it is processed
- **Use a queue to store nodes to visit**
- Can be used to get distance (# of edges) to each node from starting node





BFS - Diagram

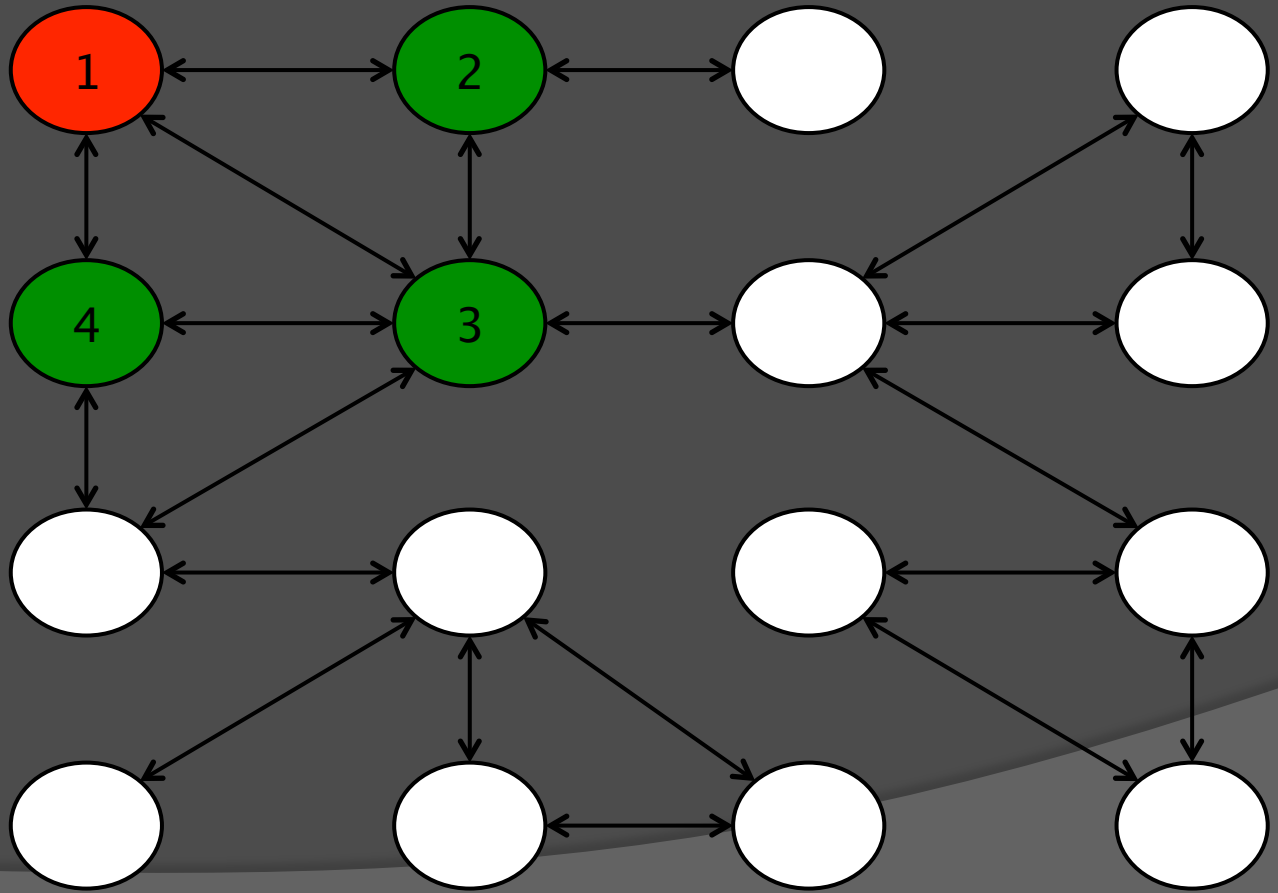
Green signifies that the node is in the queue



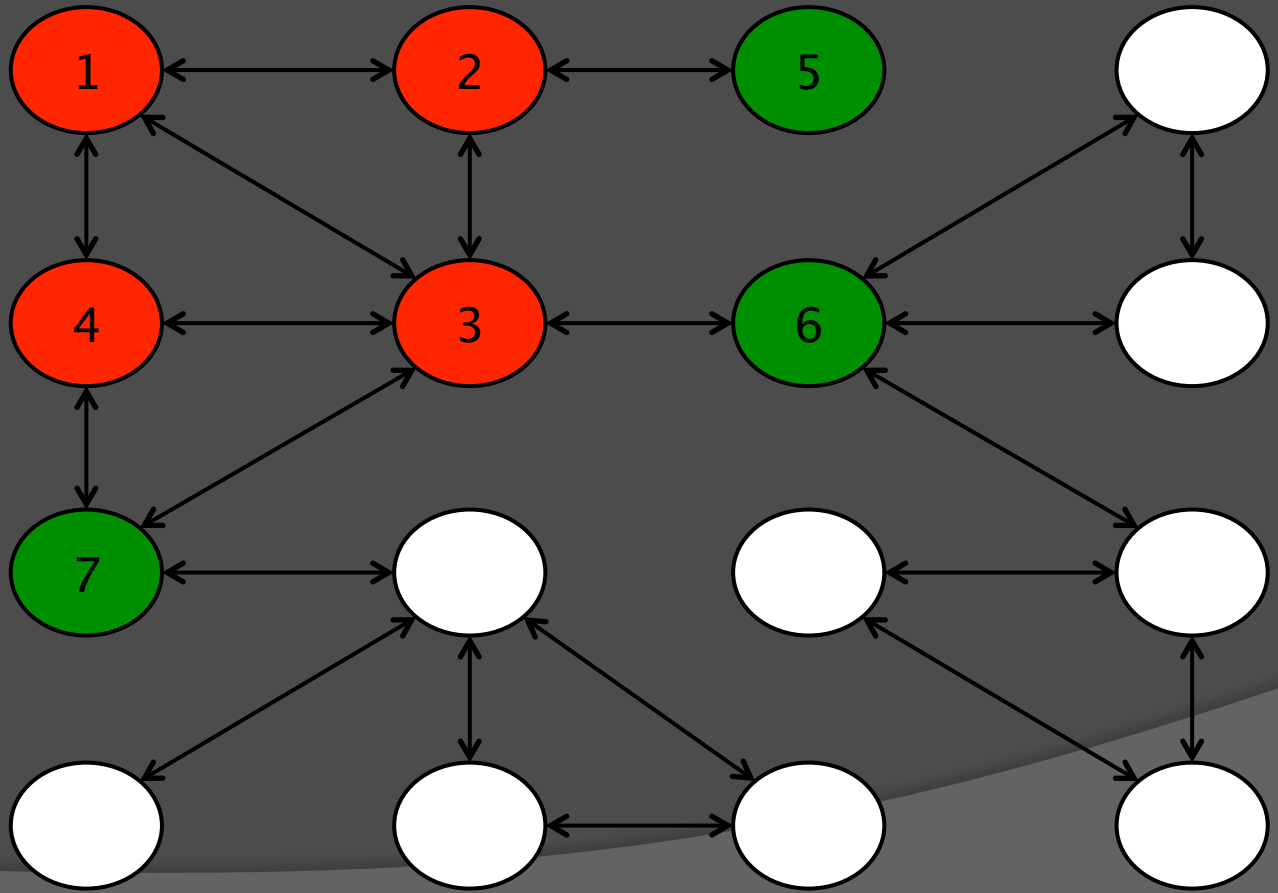
White means that the node has not been visited

BFS - Diagram

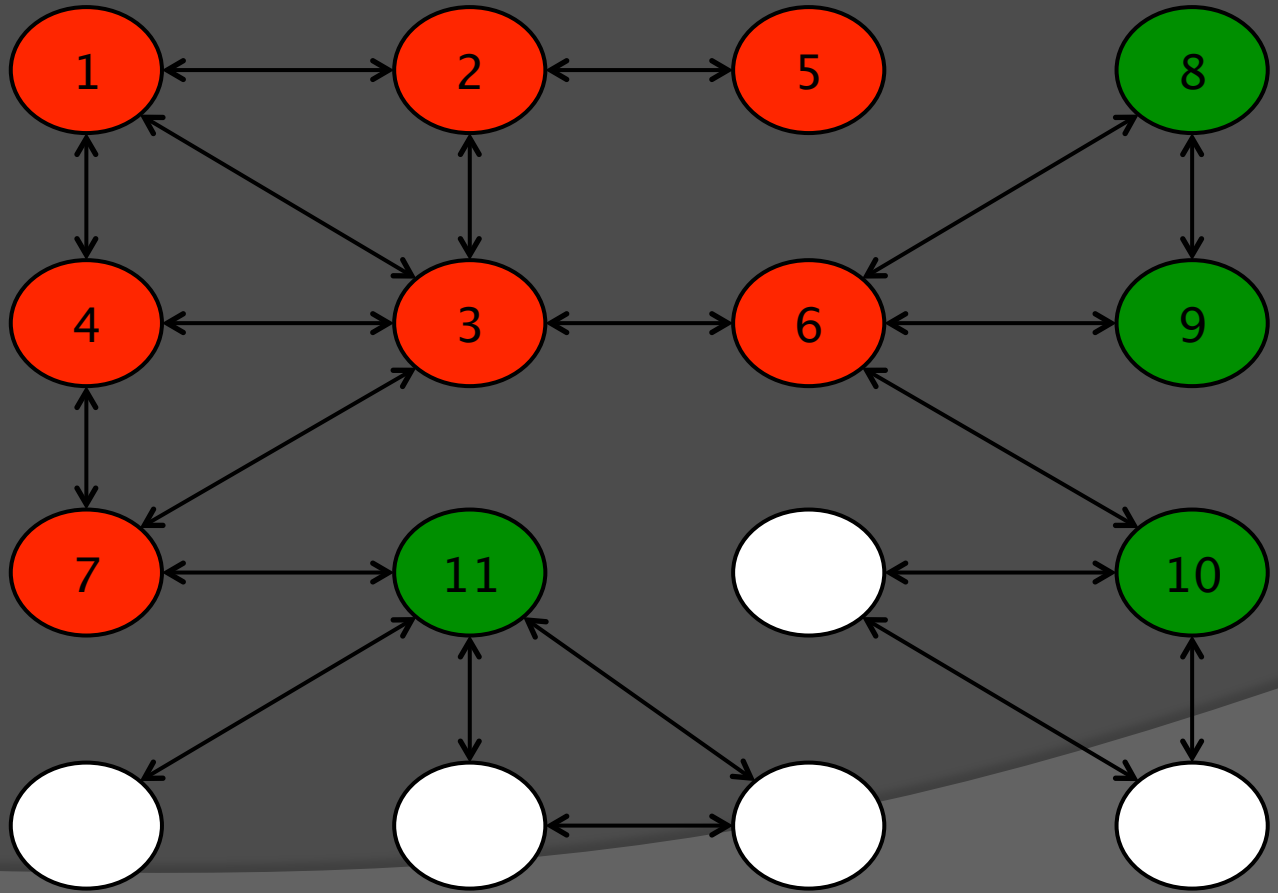
Red signifies that the node has been visited and processed



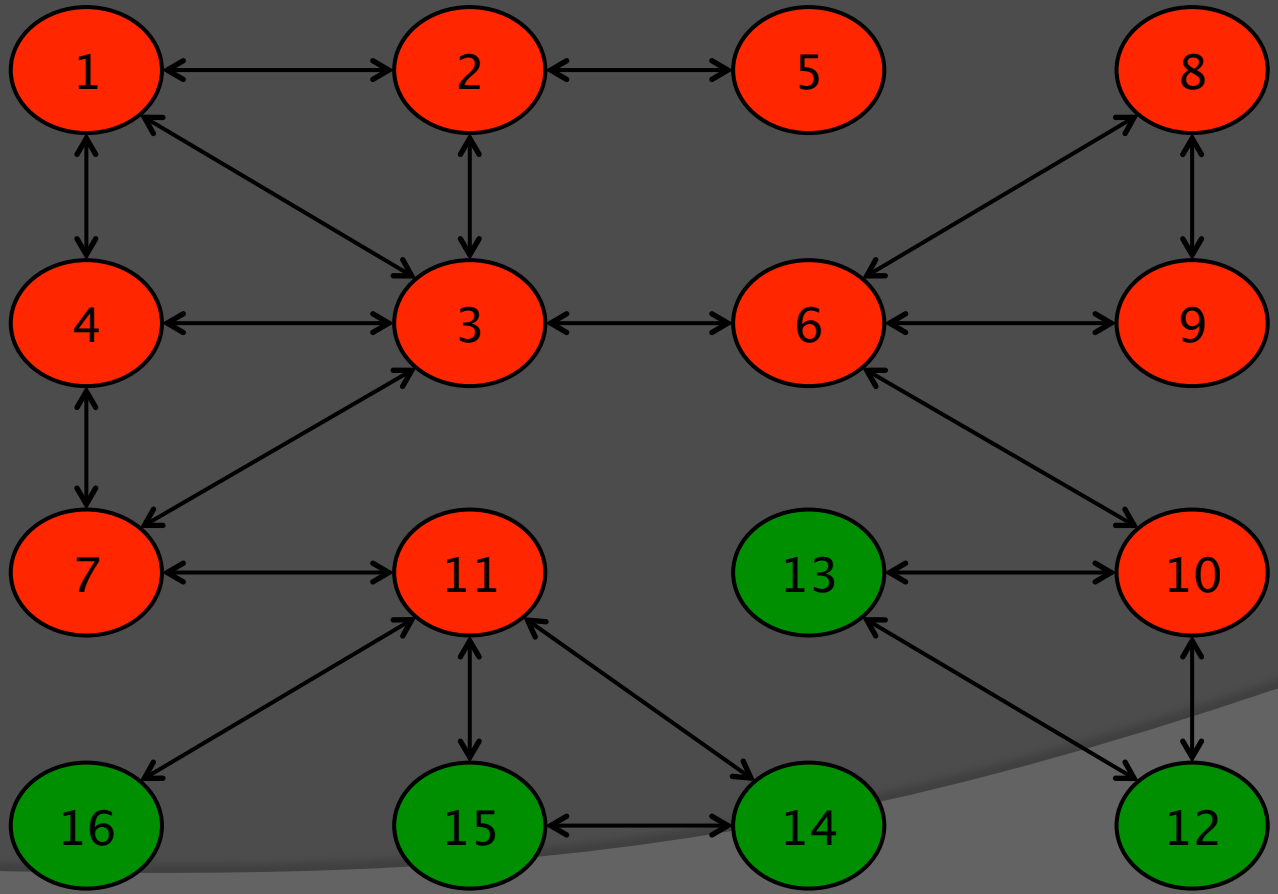
BFS - Diagram



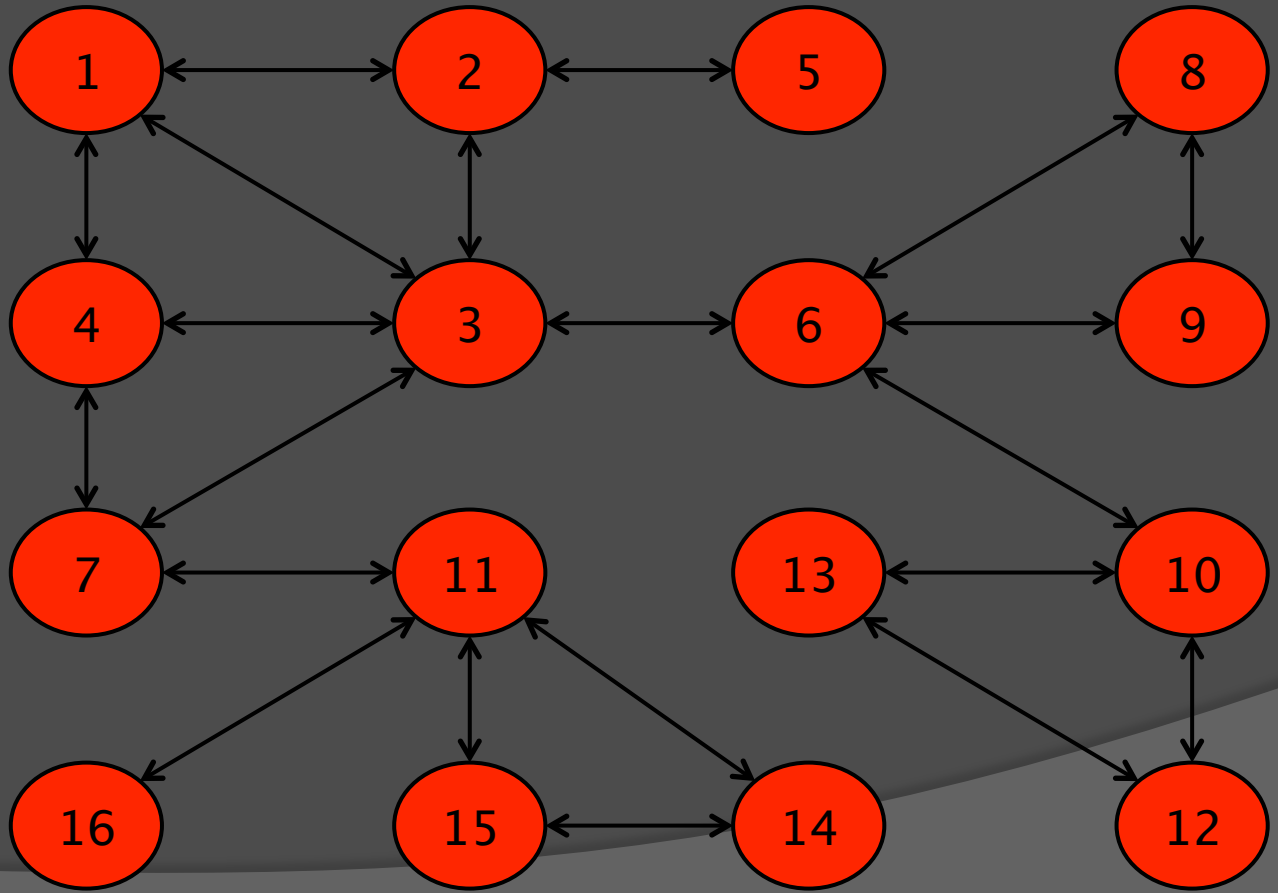
BFS - Diagram



BFS - Diagram



BFS - Diagram



BFS - Pseudocode

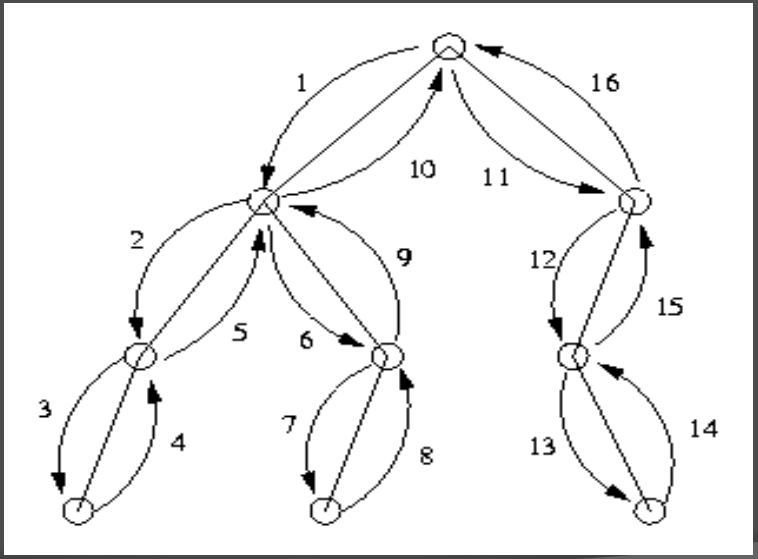
```
bfs(node start){
        queue<node> q;
        q.push(start);
        label start as visited;
        while (q is not empty){
                node cur = q.front(); q.pop();
                for (each node n adjacent to cur){
                        if (n is not visited){
                                label n as visited;
                                q.push(n);
                        }
                }
        }
}
```

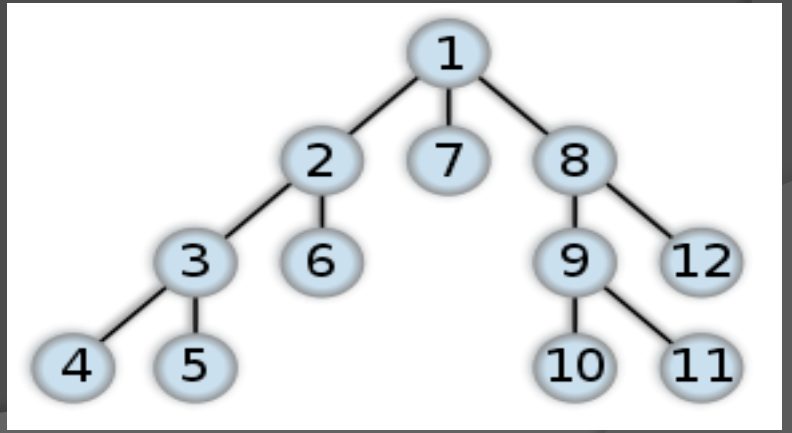
BFS – Pseudocode with `dist`

```
bfs(node start){
        queue<node> q;
        q.push(start);
        dist[start] = 0;
        label start as visited;
        while (q is not empty){
                node cur = q.front(); q.pop();
                for (each node n adjacent to cur){
                        if (n is not visited){
                                dist[n] = dist[cur]+1;
                                label n as visited;
                                q.push(n);
                        }
                }
        }
}
```

DFS – Depth-First Search

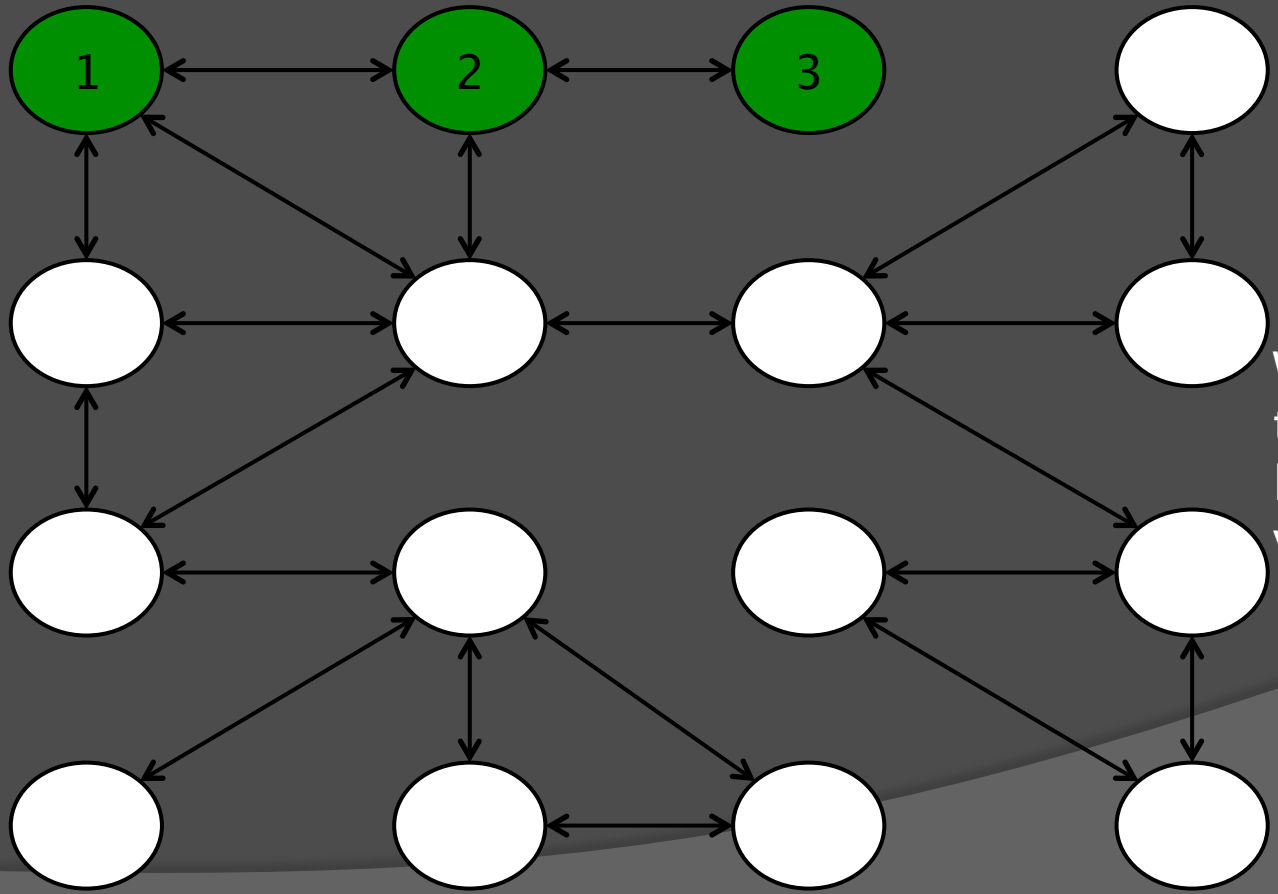
- Explores each branch to its furthest before backtracking
- **Use recursion/stack to implement DFS**





DFS - Diagram

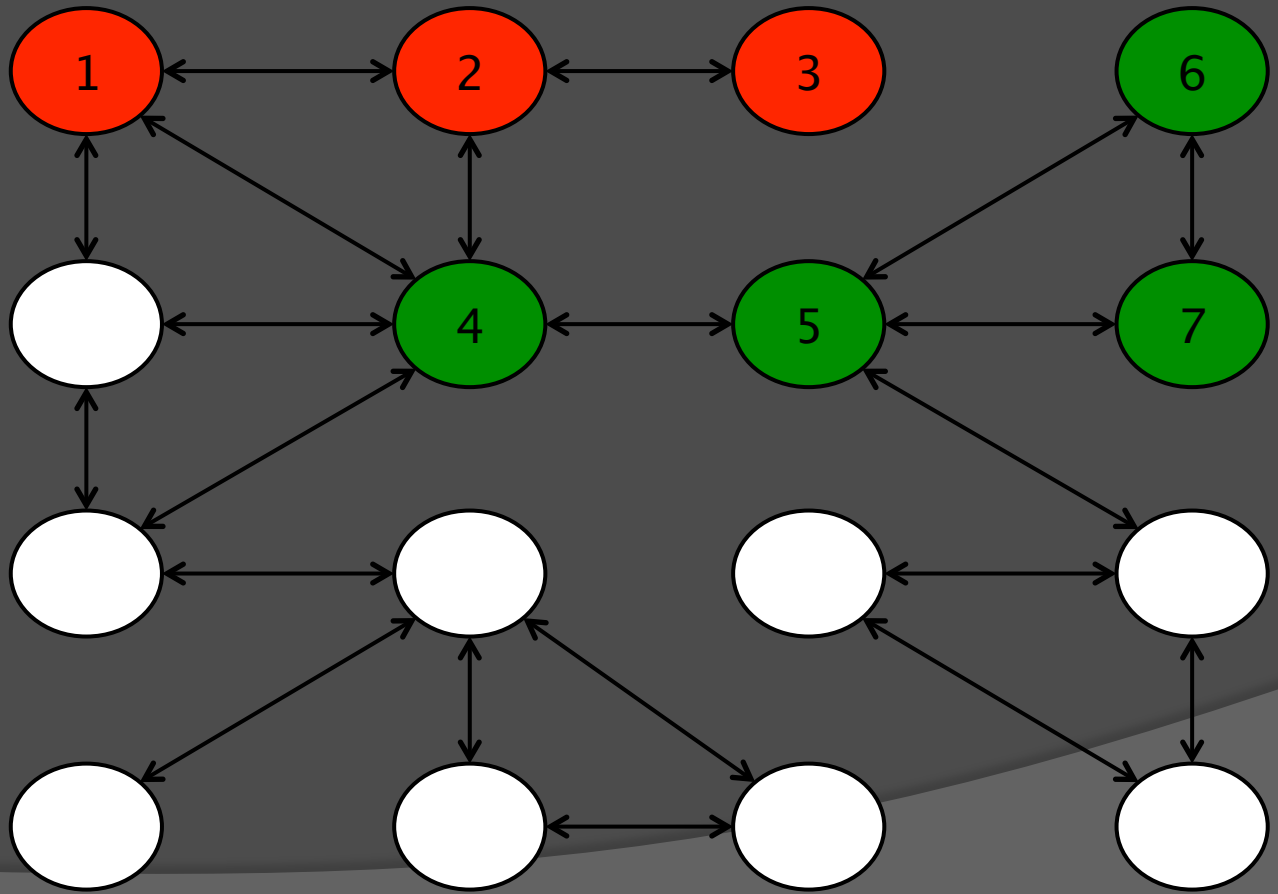
Green signifies travel path



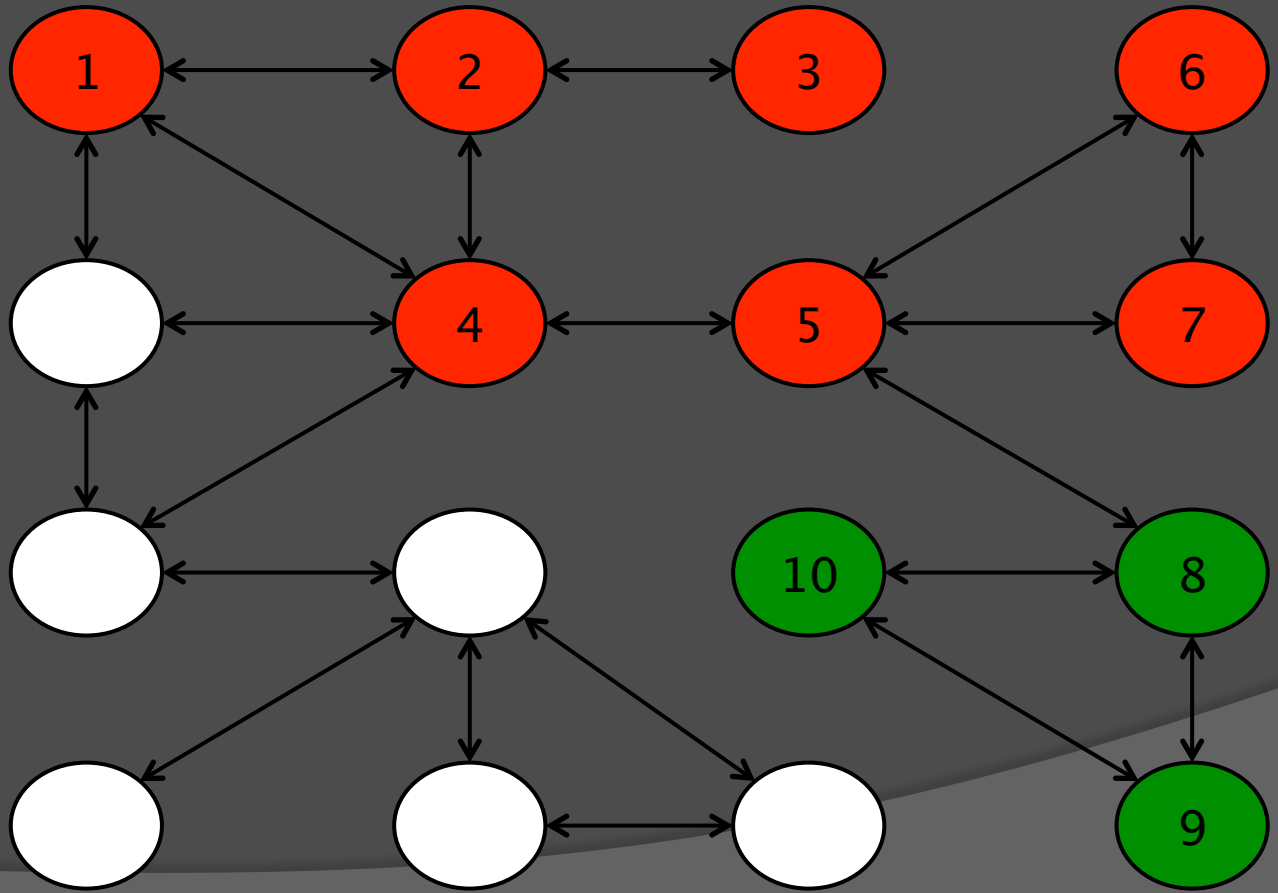
White means that the node has not been visited

DFS - Diagram

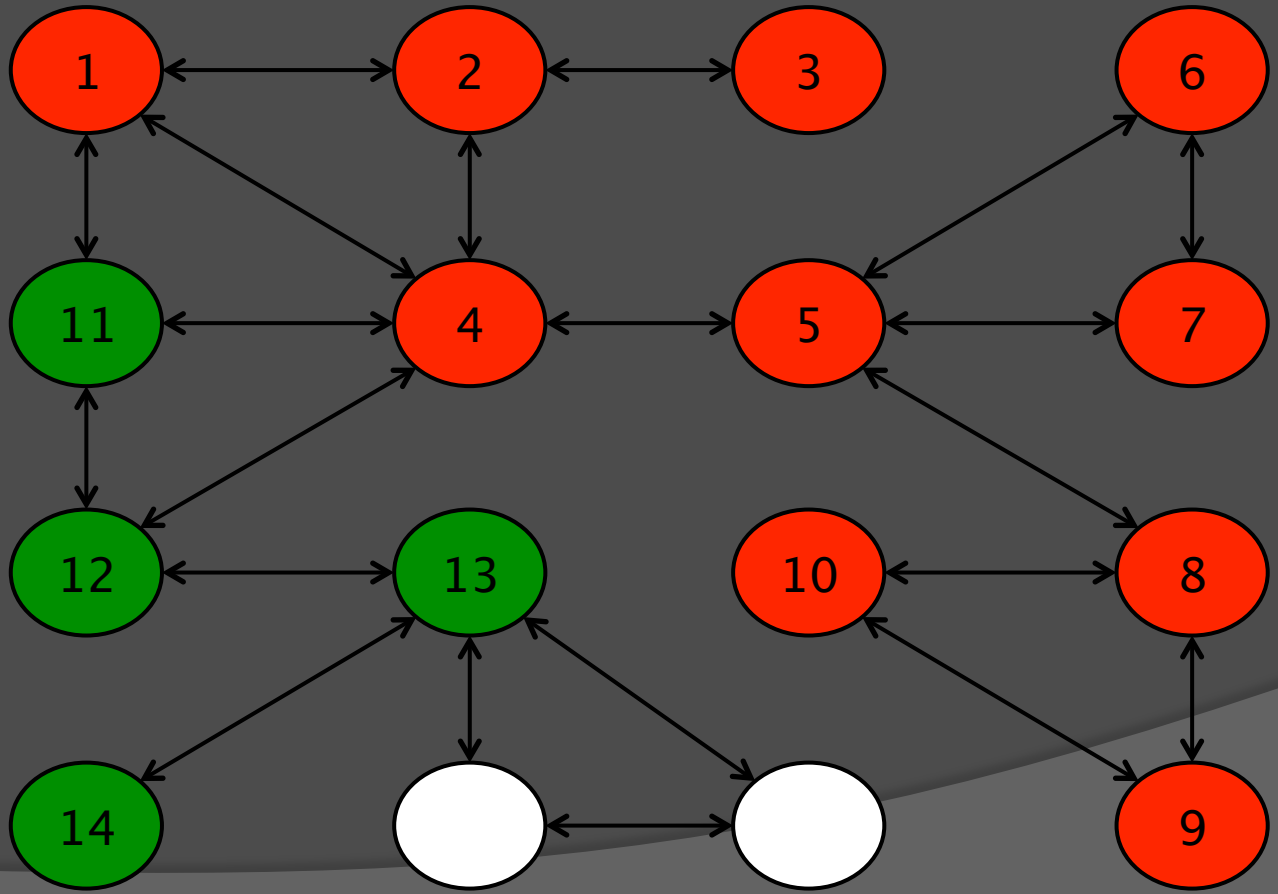
Red signifies that the node has been pushed into the stack



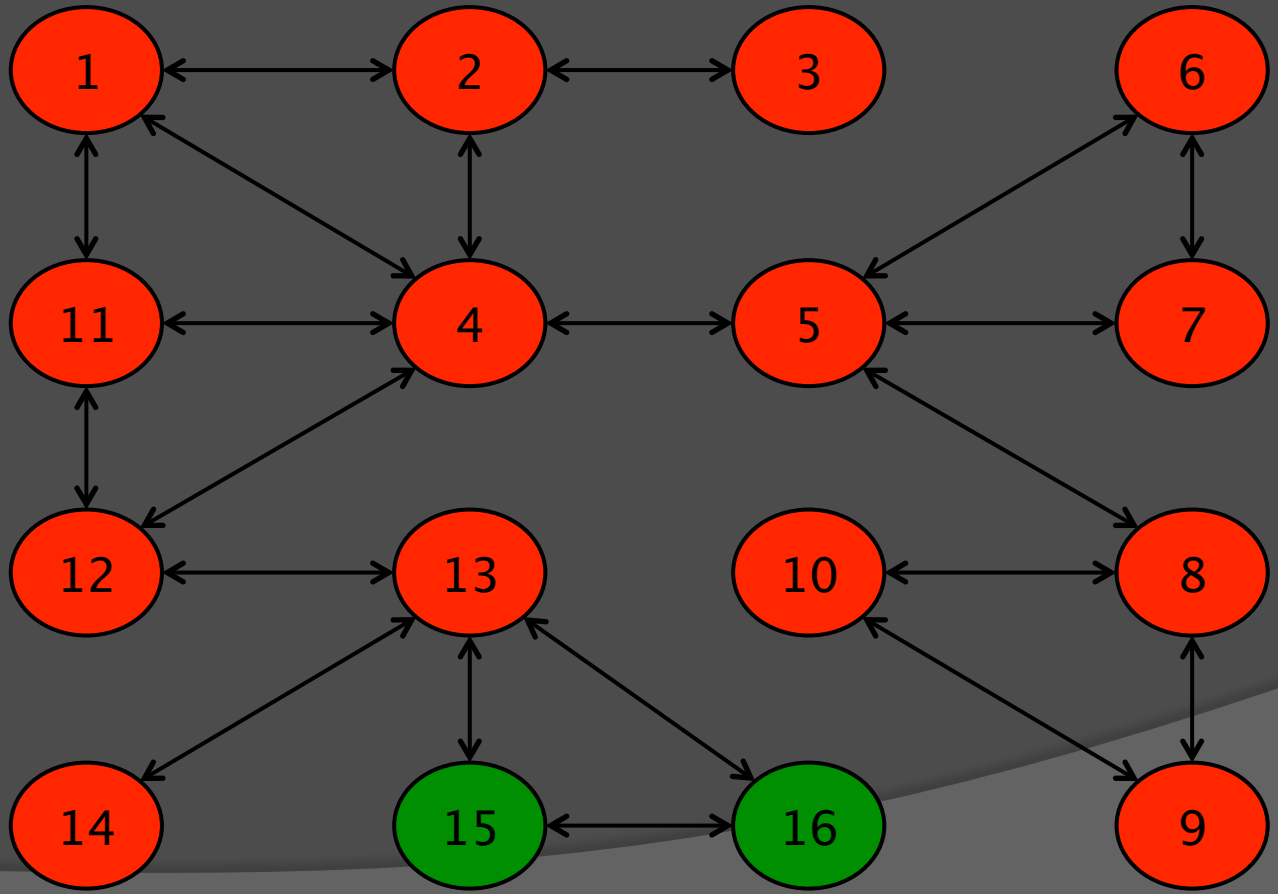
DFS - Diagram



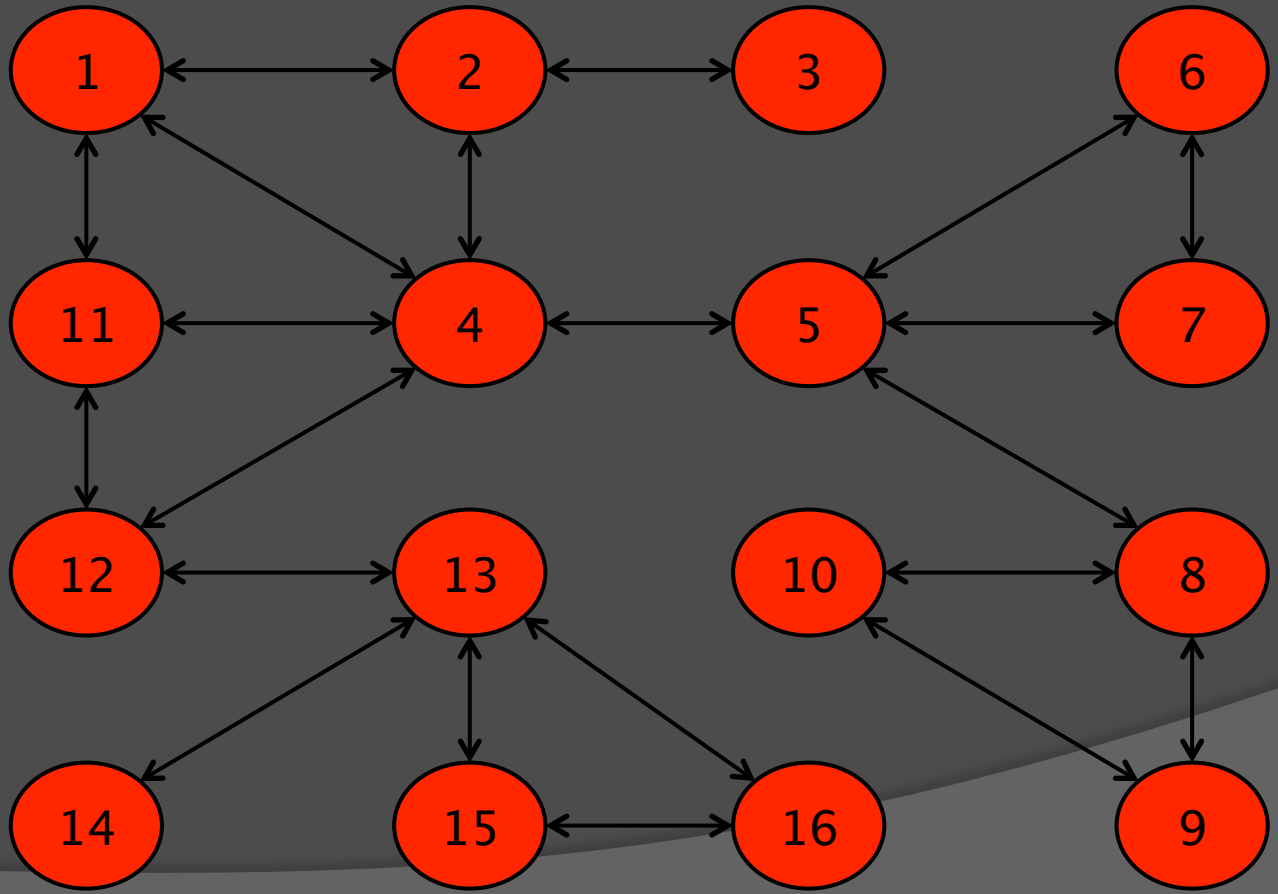
DFS - Diagram



DFS - Diagram



DFS - Diagram



DFS – Pseudocode (Stack)

```
dfs(node start){
        stack<node> s;
        s.push(start);
        label start as visited;
        while(s is not empty){
                node cur = s.top(); s.pop();
                for (each node n adjacent to cur){
                        if (n is not visited){
                                mark n as visited;
                                s.push(n);
                        }
                }
        }
}
```

DFS – Pseudocode (Recursive)

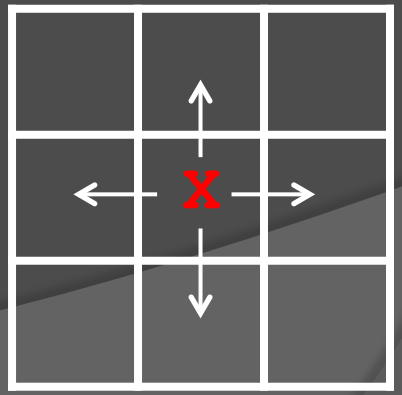
```
dfs(node n){
      if (n is visited) return;
      label n as visited;
      for (each node u adjacent to n){
            dfs(u);
      }
}
```

BFS/DFS on Not-so-Obvious Graphs

- Some problems require BFS/DFS even though there is not an obvious graph
- BFS/DFS can be done on **grids** (Knight Hop, Maze)
- BFS/DFS can be done on **states** (Coin Game)

Graph Search on Grids

- Each cell is a node
- Movement to adjacent cells are edges
- Keep an array of possible moves so that they can be easily iterated through
- Must check if cell is within the current range of the entire grid
  - For 0-indexed grids:
    - $0 <= row < R$ and $0 <= col < C$



Graph Search on States

- Each state is a node
- Each possible transition to another state is an edge
- Can use a string or an integer as a representation of a state
- Must use data structure such as a set to keep track of visited/not visited
- Must be able to convert state into integer/string
- Must convert from integer/string if queue/stack/ recursive method uses integer/string

BFS/DFS Comparison

- BFS
  - Visits nodes in order from edge distance from node
  - True distance only accurate if graph is unweighted
  - Longer code, slightly slower, more memory
- DFS
  - Explores branches as far as possible before backtracking
  - Does not get distance between start node and other nodes
  - Shorter code, faster runtime, less memory

Practice

- Given a connected, undirected, unweighted graph with N nodes (labeled 1 to N), print the distance from each node to node 1.
- Sample:

```
Edges:      Distances:
1 2         1: 0
1 3         2: 1
1 4         3: 1
2 3         4: 1
2 5         5: 2
3 7         6: 3
5 6         7: 2
6 7         8: 3
6 8
```

Practice

- Given a 2-D array of 1's and 0's, with 0 representing a floor and 1 representing a wall, print if it is possible to go from the top left corner to the bottom right corner.
- Samples:

```
0 0 0 0 0      0 0 0 0 0      0 0 1 1 1
1 1 1 1 0      0 1 1 1 0      1 0 0 1 1
0 0 0 0 0      0 1 0 0 0      1 1 0 0 1
0 1 1 1 1      0 1 0 1 1      1 1 1 0 1
0 0 0 0 0      0 0 0 0 0      1 1 1 1 0
Possible         Possible       Impossible
```

THANK YOU!