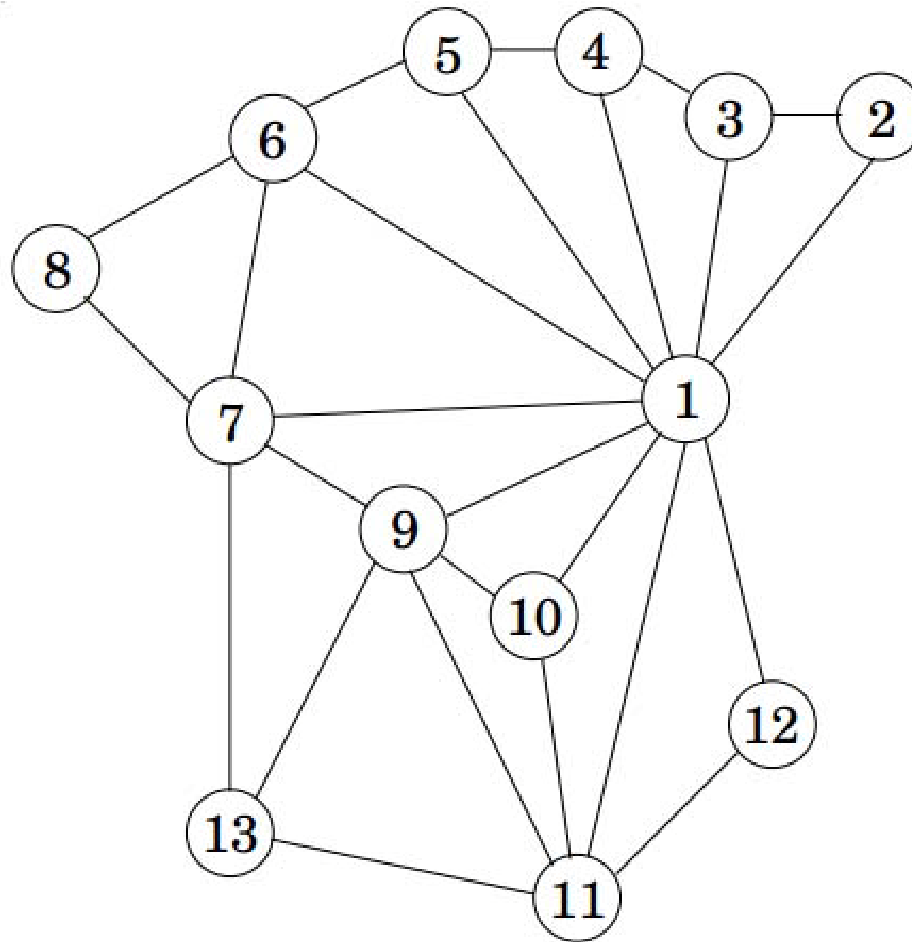
CS 161: Design and Analysis of Algorithms

Mark Zhandry

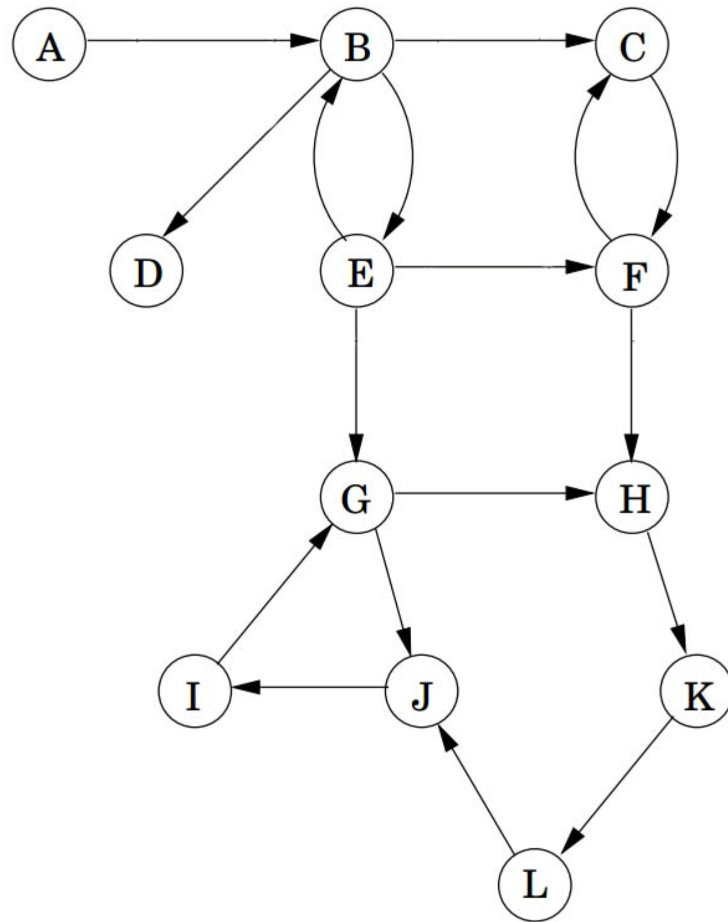
Graphs 1:
Basic Graphs/Undirected Connectivity

- Representation
- Graph Properties
- BFS
- DFS
- Connectivity

Undirected Graphs



Directed Graphs



Graph Examples

Example	Nodes	Edges
Social Network	People	Friendships
WWW	Websites	Links
Games	Board positions	Legal Moves
Road Maps	Intersections	Roads
Software	Functions	Function Calls
Chemicals	Atoms	Bonds
Scheduling	Tasks	Precedence Constraints
Electricity Grid	Power Stations	Power Lines

Graph Basics

- V = set of nodes/vertices
- E = set of edges
  - Denote an edge from u to v as (u,v)
  - Order matters in directed graph, but not in undirected graph
- Usually write a graph G as the pair (V,E)

Representing Graphs

- Adjacency Matrix:
  - Matrix A with |V| rows and columns
  - $A_{u,v}$ = 1 if and only if the edge (u,v) exists
  - Undirected graph: A is symmetric
- Adjacency List:
  - Array with |V| entries
  - The entry for u is the list of edges (u,v)

Graph Concepts

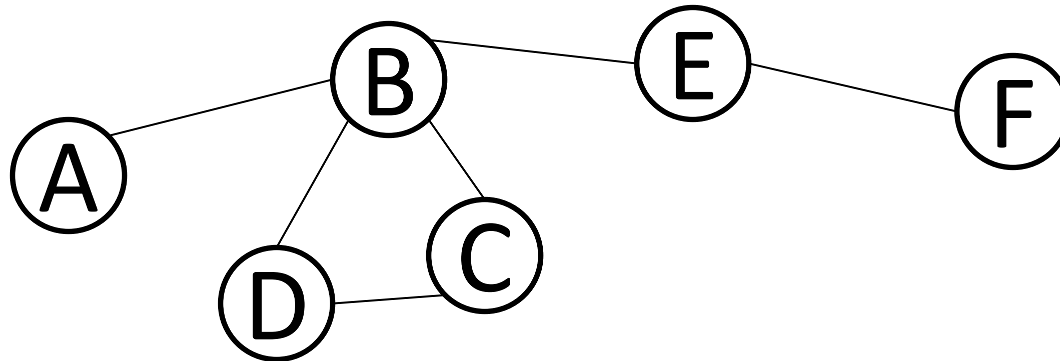
- Undirected graphs:
  - degree(v) = number of edges incident on v
- Directed graphs:
  - Indegree(v) = number of edges into v
  - Outdegree(v) = number of edges out from v

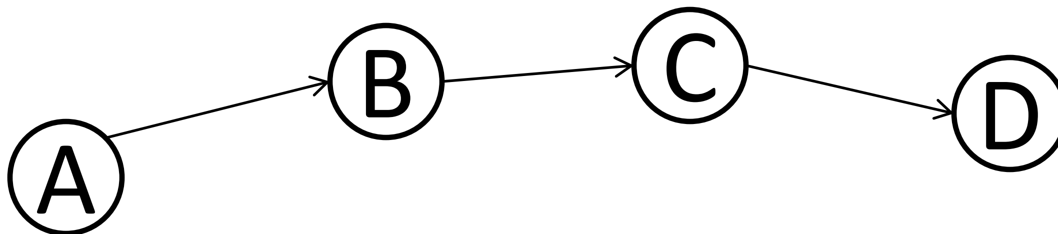
Representing Graphs

- Space requirements:
  - Adjacency Matrix: $O(|V|^2)$
  - Adjacency List: $O(|V|+|E|)$
- Time to see if (u,v) is in graph:
  - Adjacency Matrix: $O(1)$
  - Adjacency List: O(degree(v)) or O(outdegree(u))
- Time to get all (u,v) for specific u:
  - Adjacency Matrix: $O(|V|)$
  - Adjacency List: O(degree(v)) or O(outdegree(v))

Graph Concepts

- Path: sequence v_1, v_2, … v_k, where (v_i, v_{i+1}) is an edge in the graph

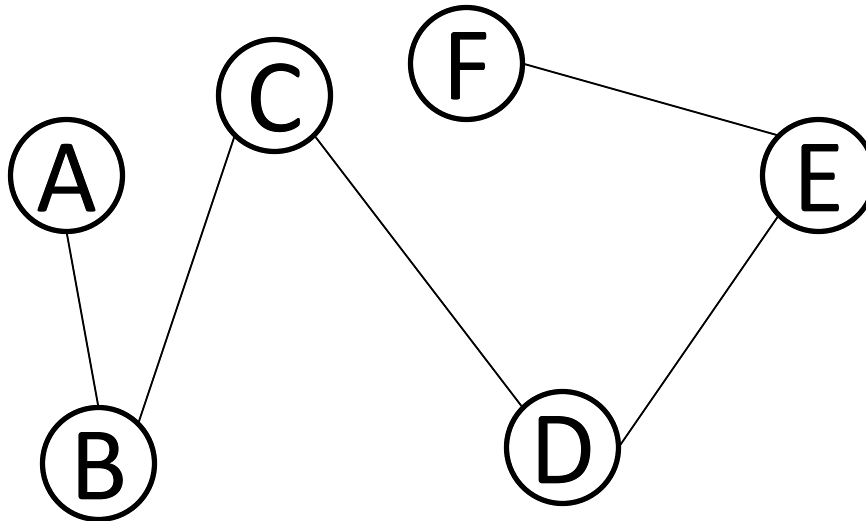




- Length of path = number of edges in path
  - Infinity if no path

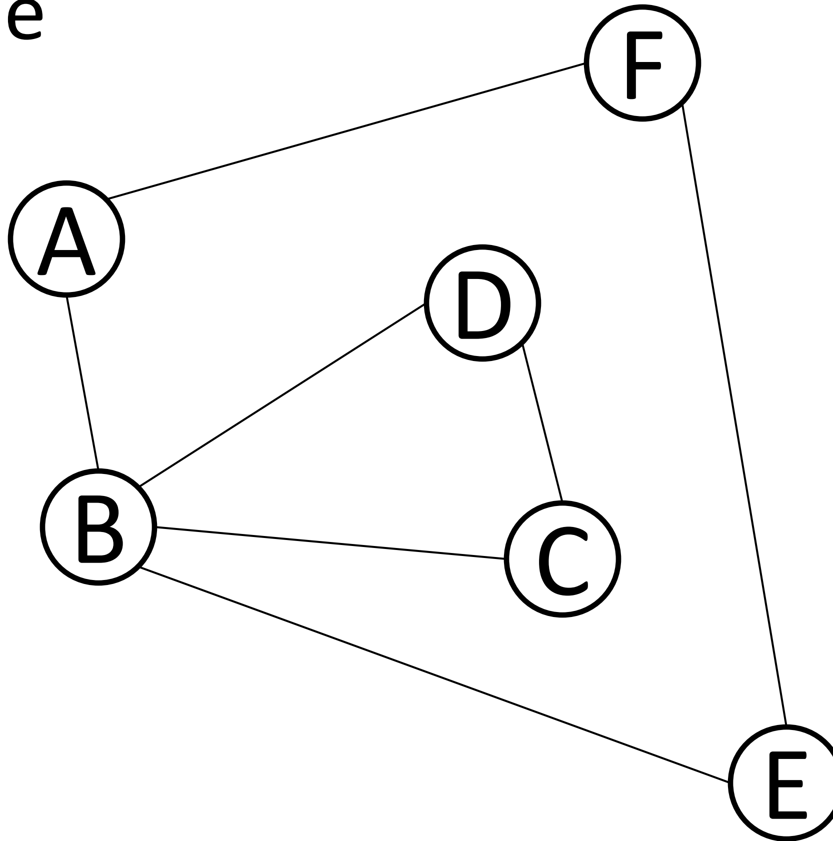
Graph Concepts

- Simple Path: path where all nodes are different



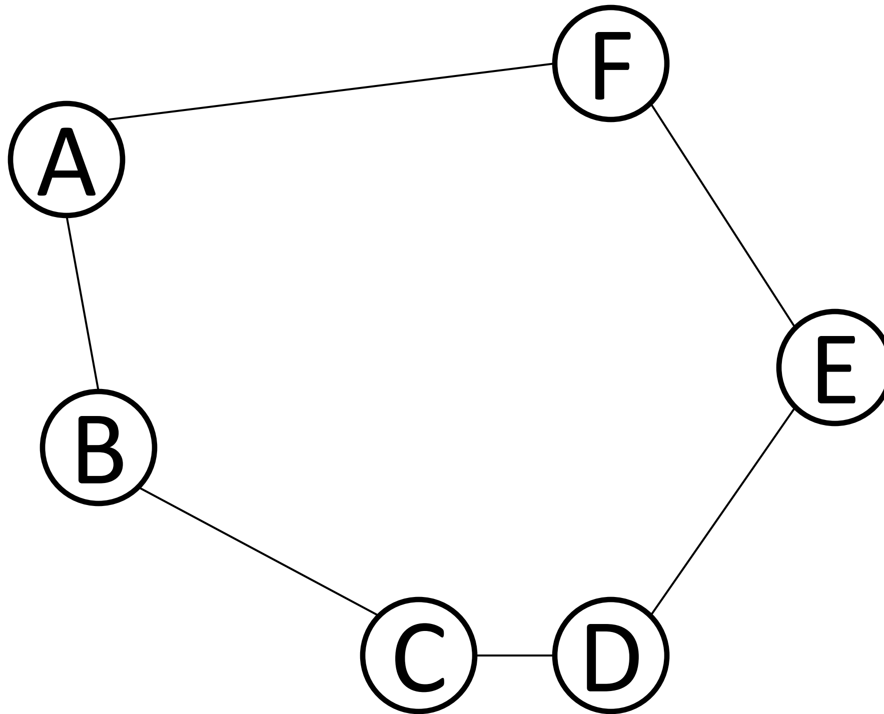
Graph Concepts

- Cycle: path where first and last node are the same



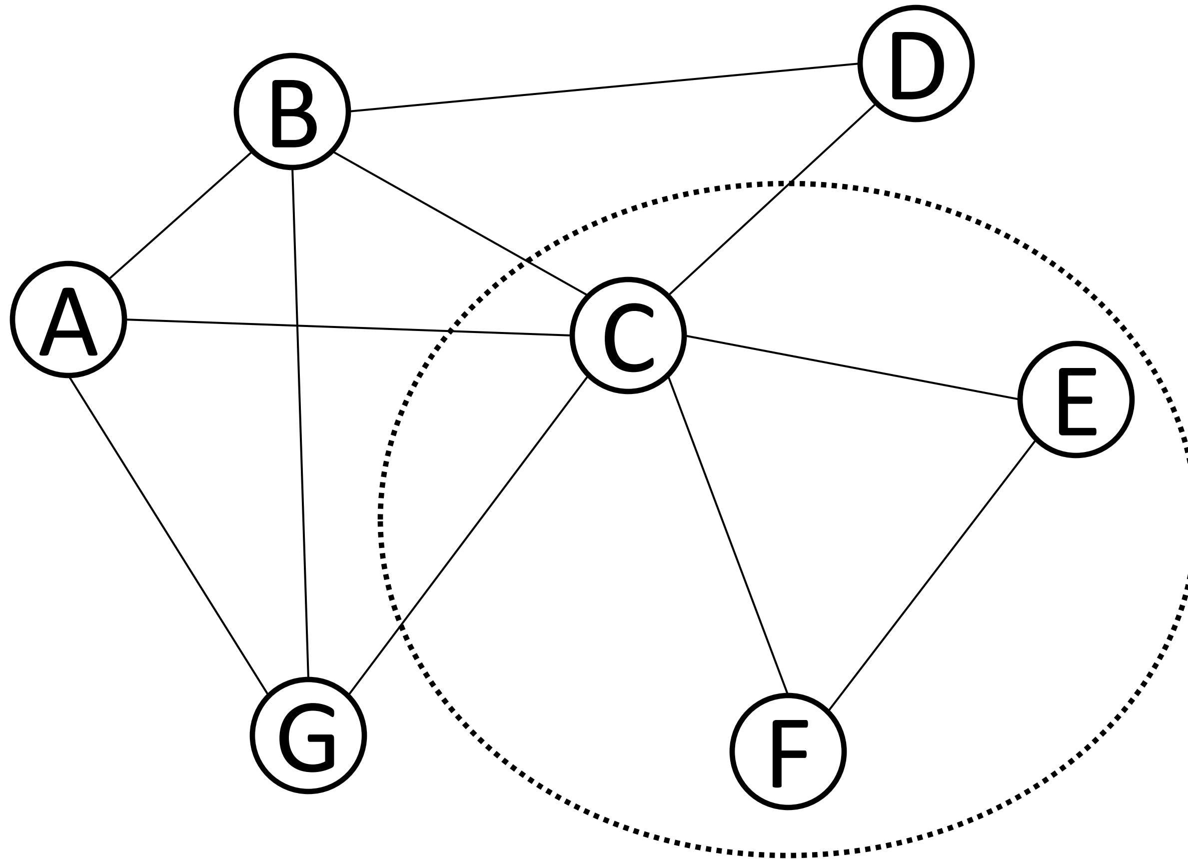
Graph Concepts

- Simple Cycle: cycle where all nodes are different (except the first and last)



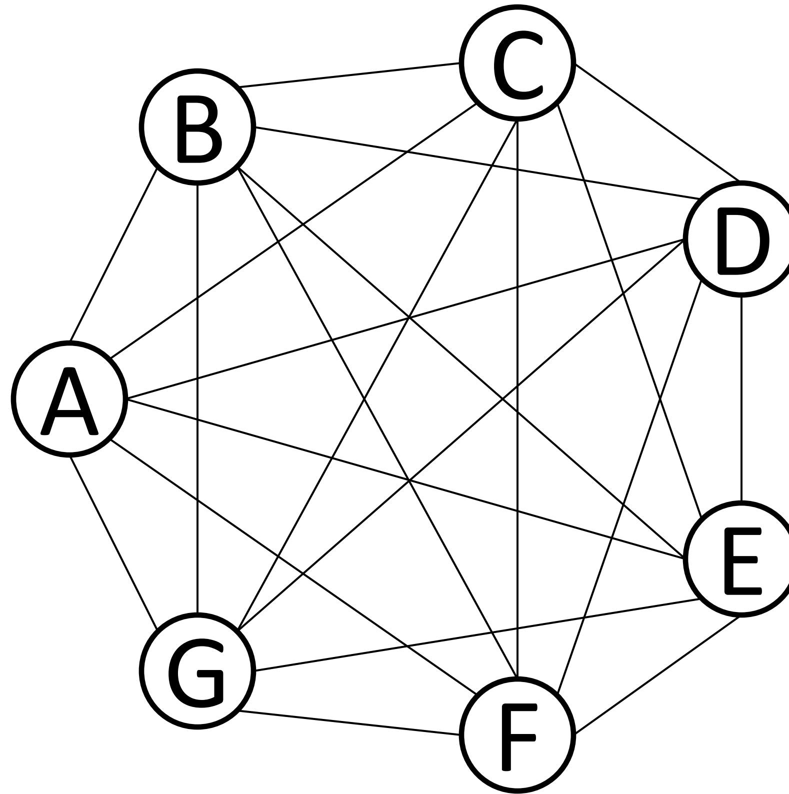
Graph Concepts

- Subgraph: A graph G'=(V',E') such that V' is a subset of V and E' is a subset of E



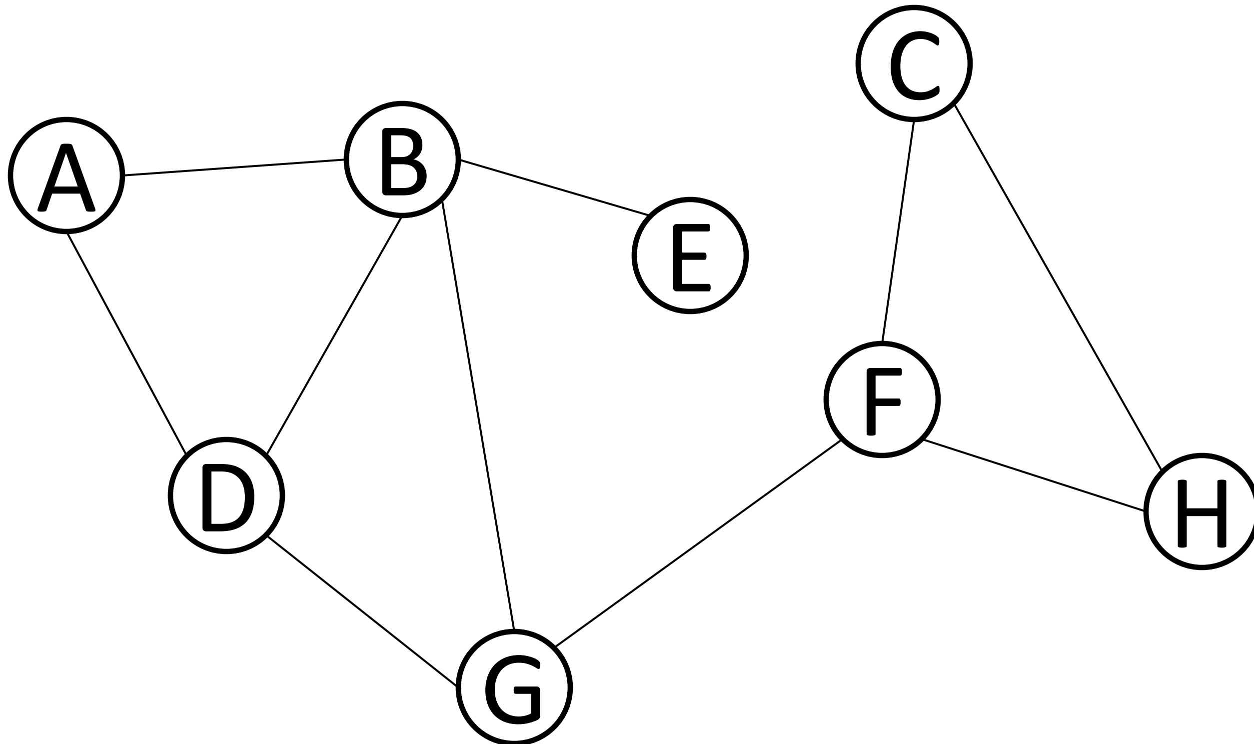
Types of Graphs

- Complete Graph: all possible edges



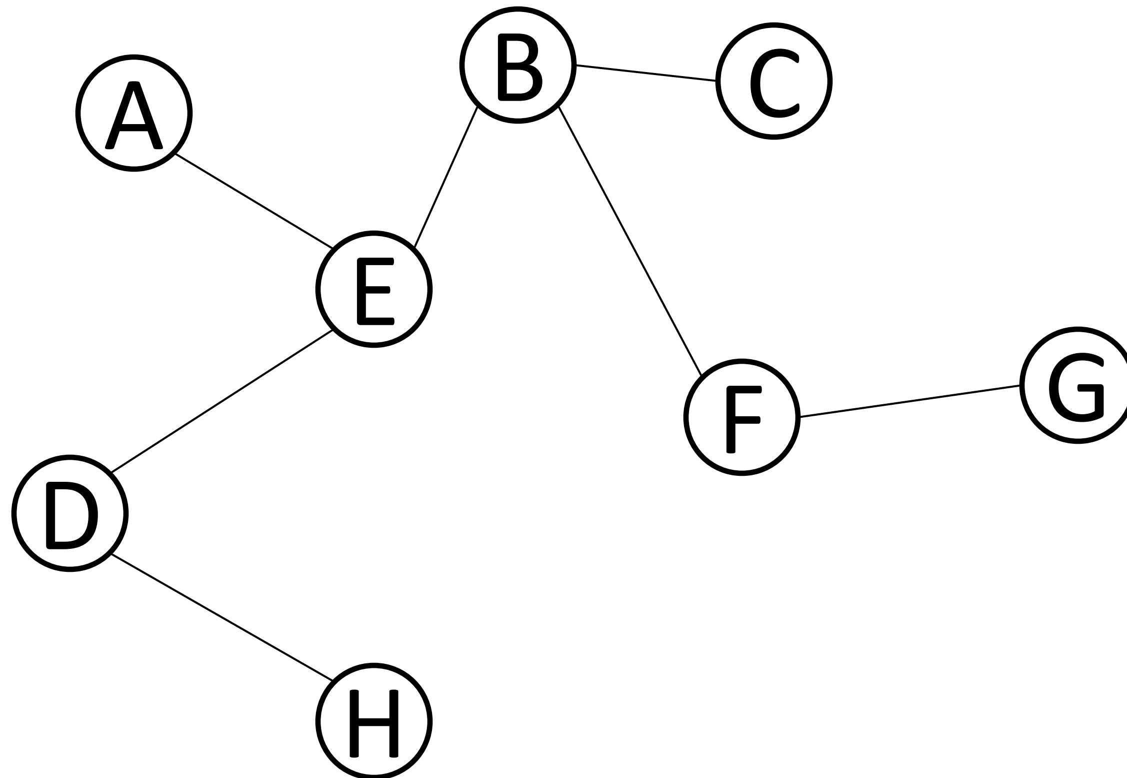
Types of Undirected Graphs

- Connected Graph: there is a path connecting any two nodes



Types of Undirected Graphs

- Tree: connected graph with no simple cycles



Tree Properties

- $|E|=|V|-1$
  - Proof: If $|V|=1$, trivial
  - Assume true for $|V|=n-1$, we prove true for $|V|=n$
  - Pick a leaf node, and remove it. $|V'|=|V|-1$ and $|E'|=|E|-1$ (is there always a leaf node?)
  - $|E|=|E'|+1=|V'|-1+1=|V'|=|V|-1$

Tree Properties

- Every two nodes u and v have exactly 1 path between them
  - Proof: Suppose they have 2 paths p_1 and p_2
  - Let w be the first node in p_1 that is also in p_2
  - Let p be the path obtained by following p_1 from u to w, and then p_2 from w back to u
  - p is a simple cycle!

Shortest Paths

- The distance between nodes u and v is the minimum length of all paths from u to v
  - Shortest path is simple (why?)
- How do we compute the distance between u and v?

Breadth-First Search

- Idea: Let w_i be the neighbors of v, and let p_i be the shortest path from u to w_i.
- Any path from u to v must have one of the w_i be the last node before v.
- Then the shortest path from u to v must be one of the p_i followed by v

Breadth-First Search

- What if we already knew the distances from u to all of the nodes with distance at most d
- We can find the distances to all nodes with distance at most d+1 as follows:
  - For every node v with distance d, look at every outgoing edge (v,w)
  - If w does not have distance at most v, its distance is d+1
- Base case: u is the only node with distance 0 from u

Breadth-First Search

Algorithm: Input graph G=(V,E), node u

- Set distance(u) = 0, distance(v)=∞ for v≠u
- q = new queue containing u
- While(q is not empty)
  - v = q.poll()
  - For every (v,w) in E with distance(w)=∞:
    - Set distance(w)= distance(v)+1
    - q.add(w)

Proof of Correctness

- Claim: For each d=0,1,…, there is some point at which
  - all nodes with distance at most d have their distances correctly set,
  - all other nodes have distance set to infinity,
  - and q contains exactly the nodes at distance d

Proof of Correctness

- True for d = 0 at beginning. Inductively assume true for d-1
  - At some point, q contains nodes of distance d-1, all nodes with distance at most d-1 have correct distance, and all other nodes have distance set to infinity

Proof of Correctness

- Process all elements currently in q.
  - If a node v has distance at most d-1, distance(v) not changed
  - If a node v has distance d, it has some neighbor with distance d-1 that we will process
    - Thus we will set distance(v) to d and add it to q
  - If a node v has distance > d, all of its neighbors have distance > d-1, so we do not change distance(v)

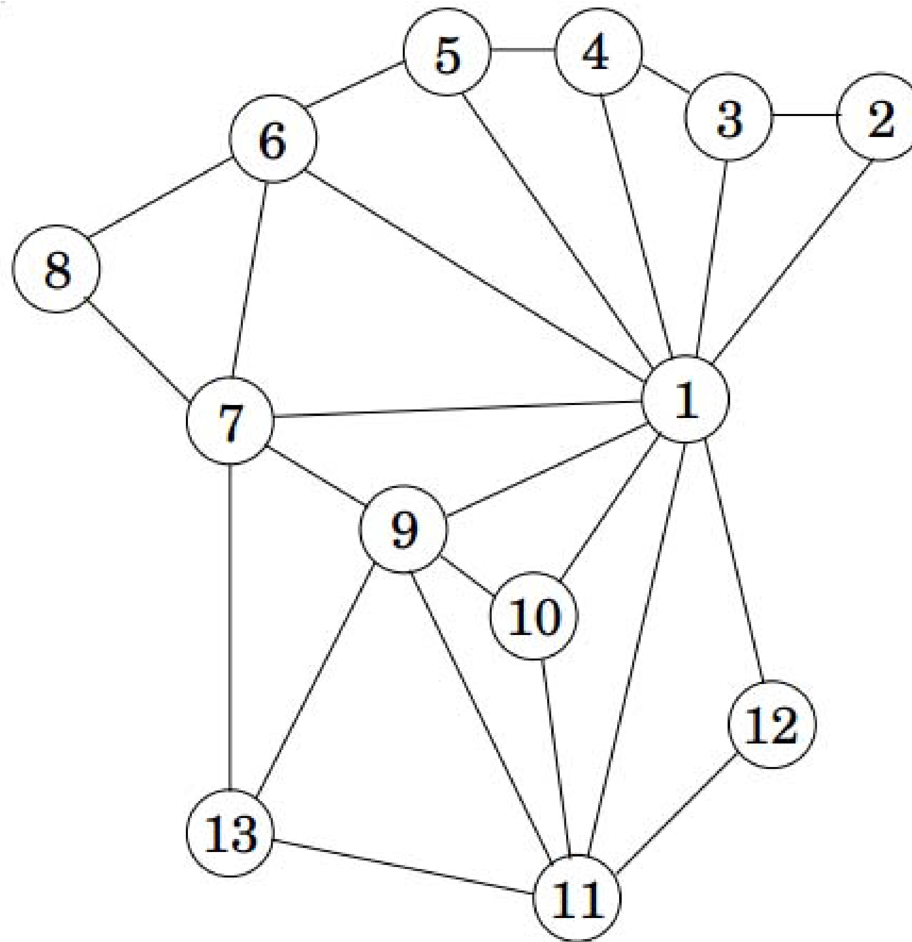
Proof of Correctness

- End result:
  - All nodes with distance d in queue
  - All nodes v with distance > d have distance(v)=∞
  - All nodes v with distance at most d have correct distance(v)
- Once d > distance of farthest node, all nodes have correct distance, queue is empty, and program stops

Running Time

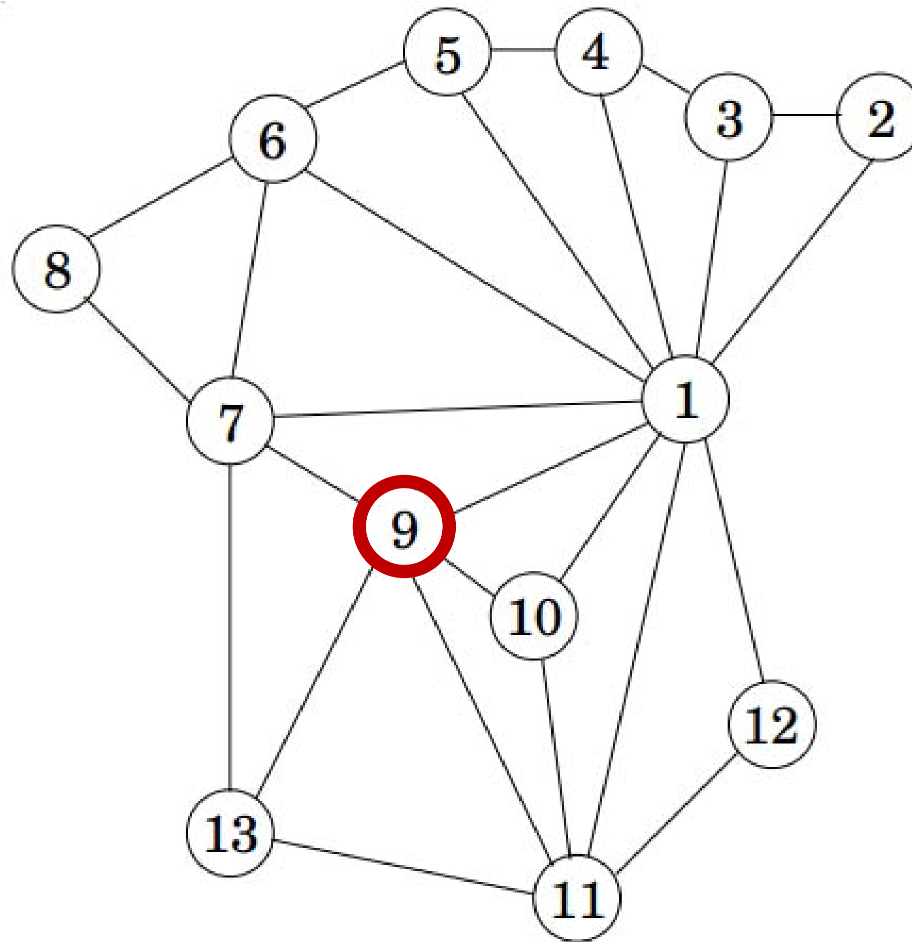
- Every node added to queue only when its distance is changed from infinity to something finite
  - Each node added only once
- Each edge (u,v) only examined when u is removed from queue
  - Also when v is removed for undirected graphs
  - Therefore, each edge examined at most twice
  - If we use adjacency list, getting the edges from a node takes constant time per edge
- Running time: $O(|V|+|E|)$

BFS Example



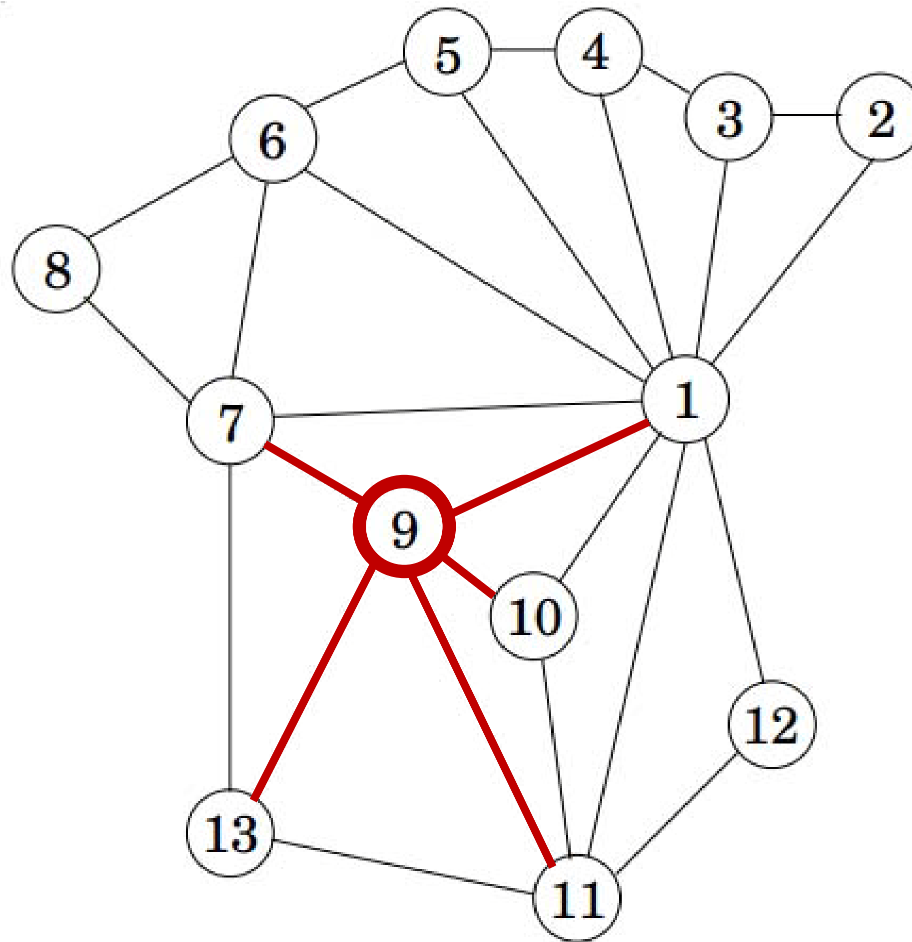
BFS Example

d = 0



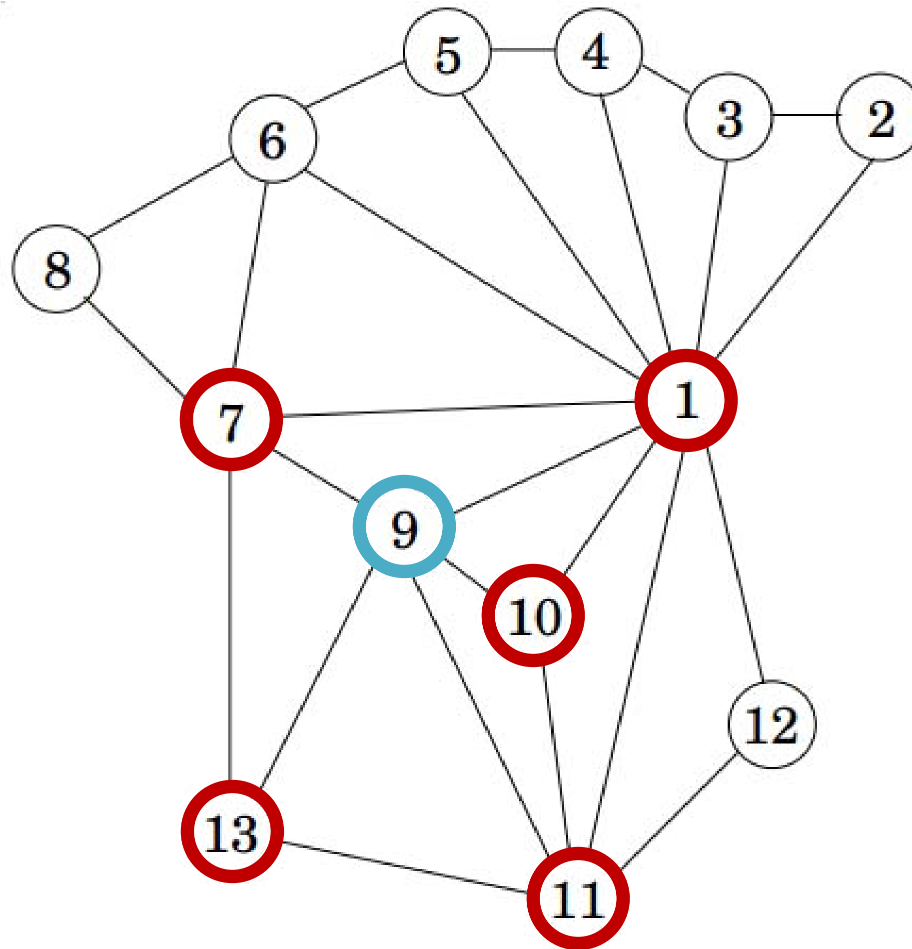
BFS Example

d = 0



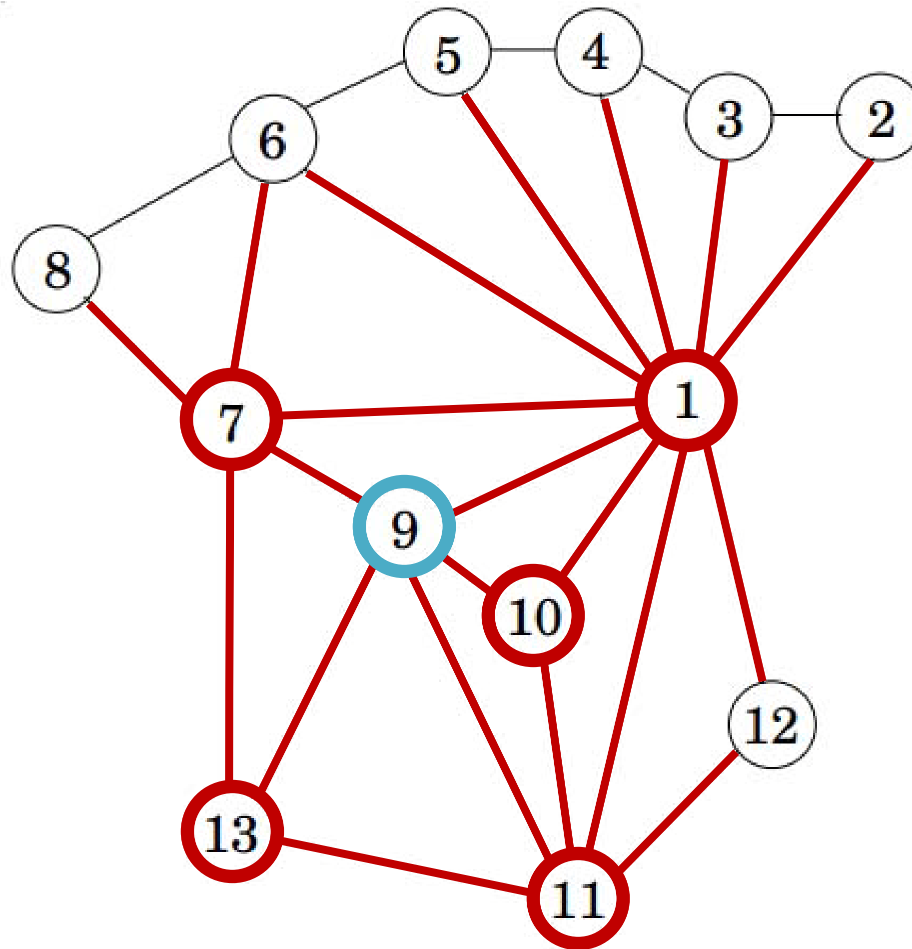
BFS Example

d = 1



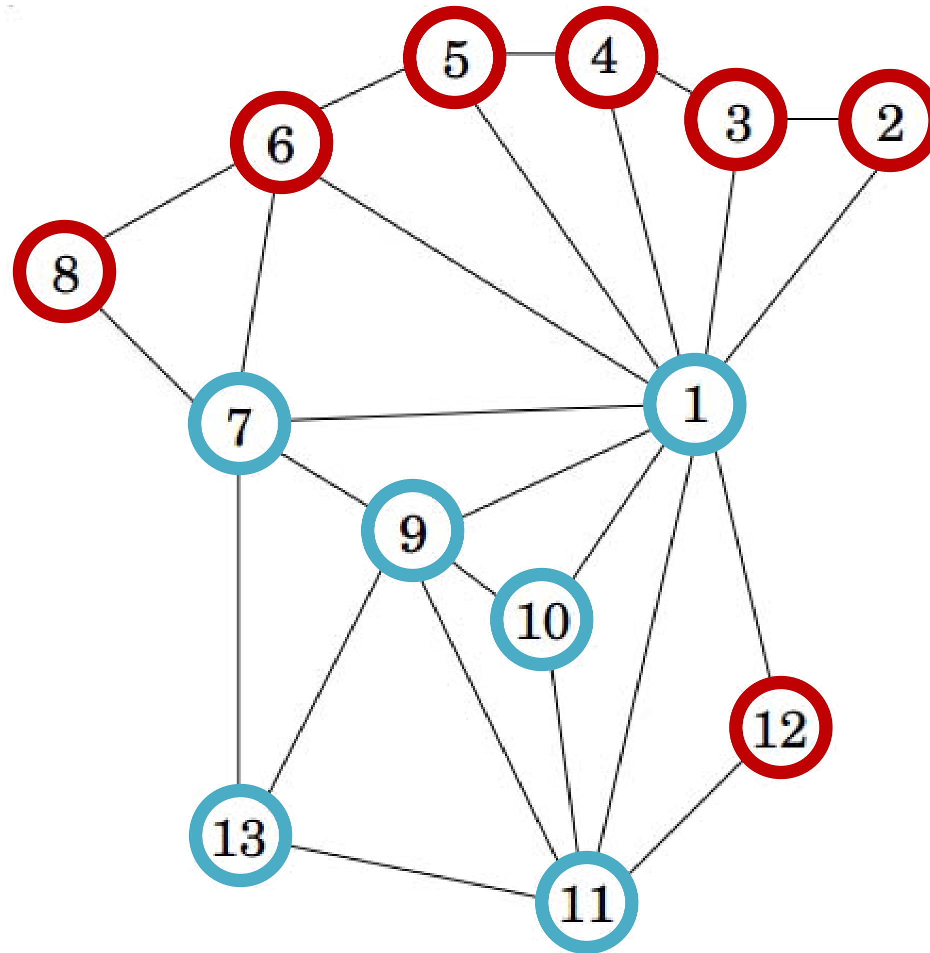
BFS Example

d = 1



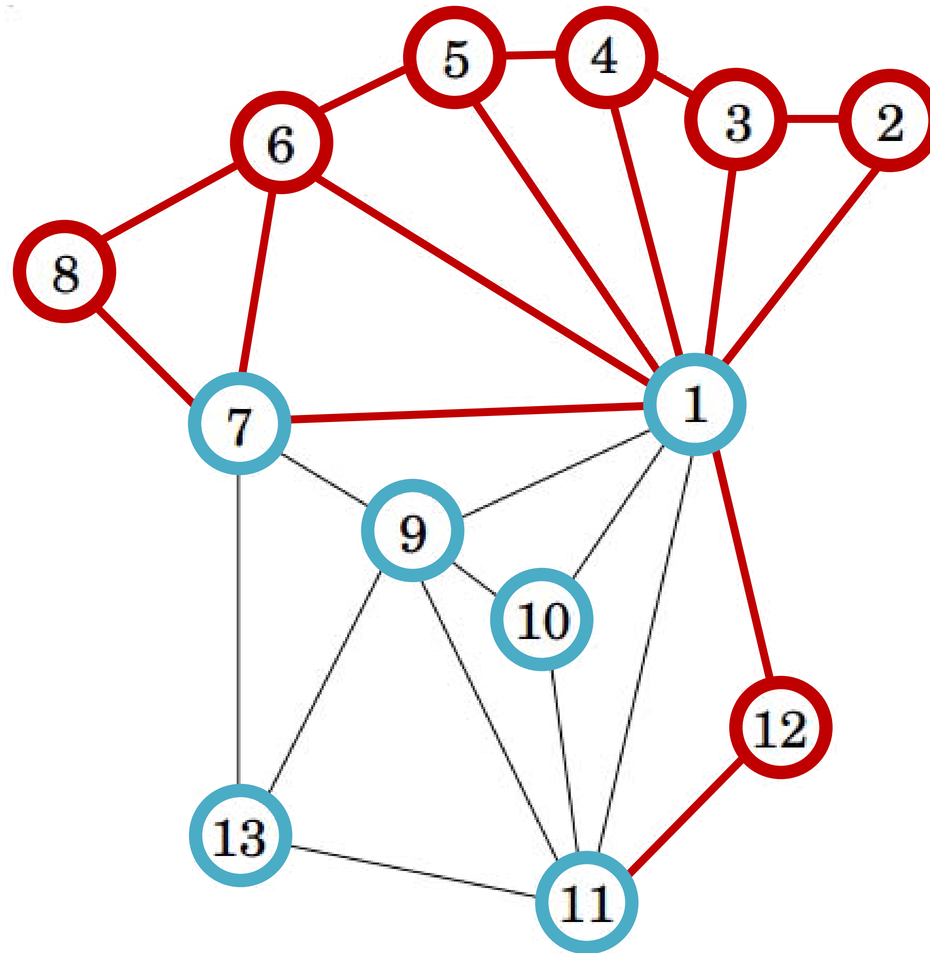
BFS Example

d = 2



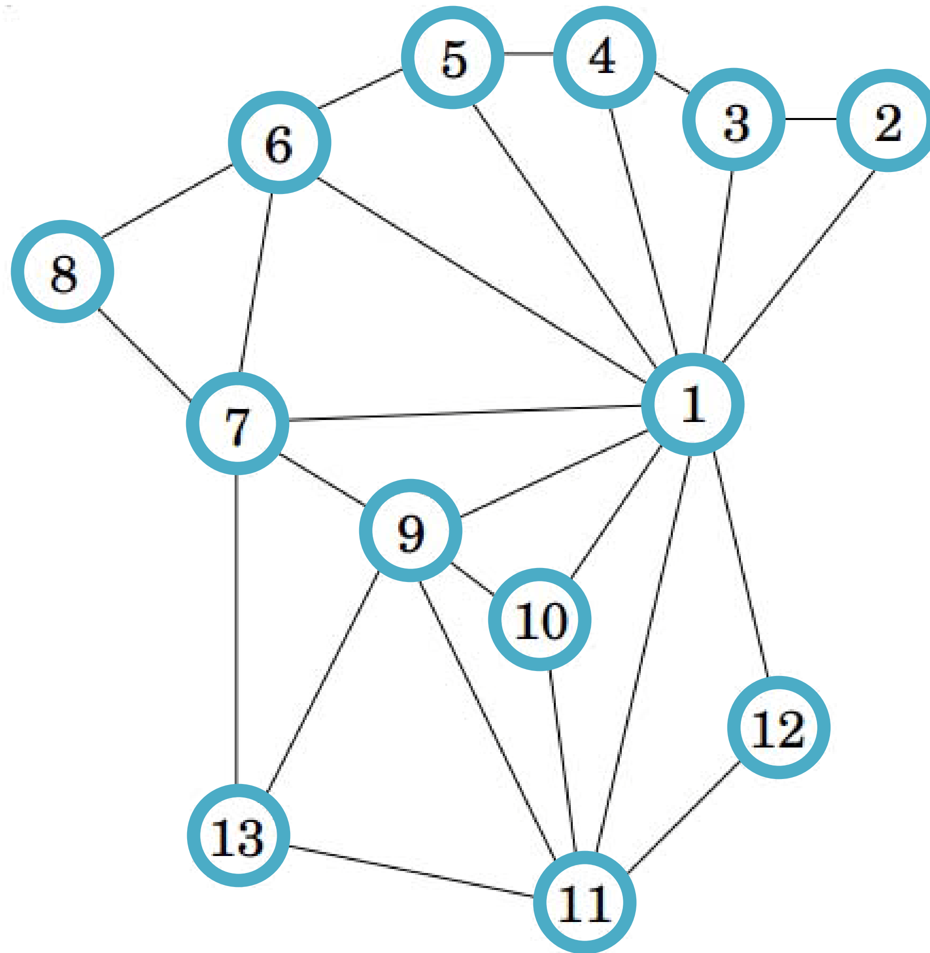
BFS Example

d = 2



BFS Example

d = 3



Abstract View of BFS

- BFS(G,u) =

```
visited(u) = true
q.add(u)
While q is not empty:
    v = q.peak()
    for each edge (v,w) where not visited(w)
        previsit(w)
        visited(w) = true
        q.add(w)
    postvisit(v)
    q.poll()
```

DFS: Exchange Queues for Stacks

- Explore(G,u) =

```
visited(u) = true
s.push(u)
While q is not empty:
    v = s.peak()
    for each edge (v,w) where not visited(w)
        previsit(w)
        visited(w) = true
        s.push(w)
    postvisit(v)
    s.pop()
```

Recursion: Implicit Stack

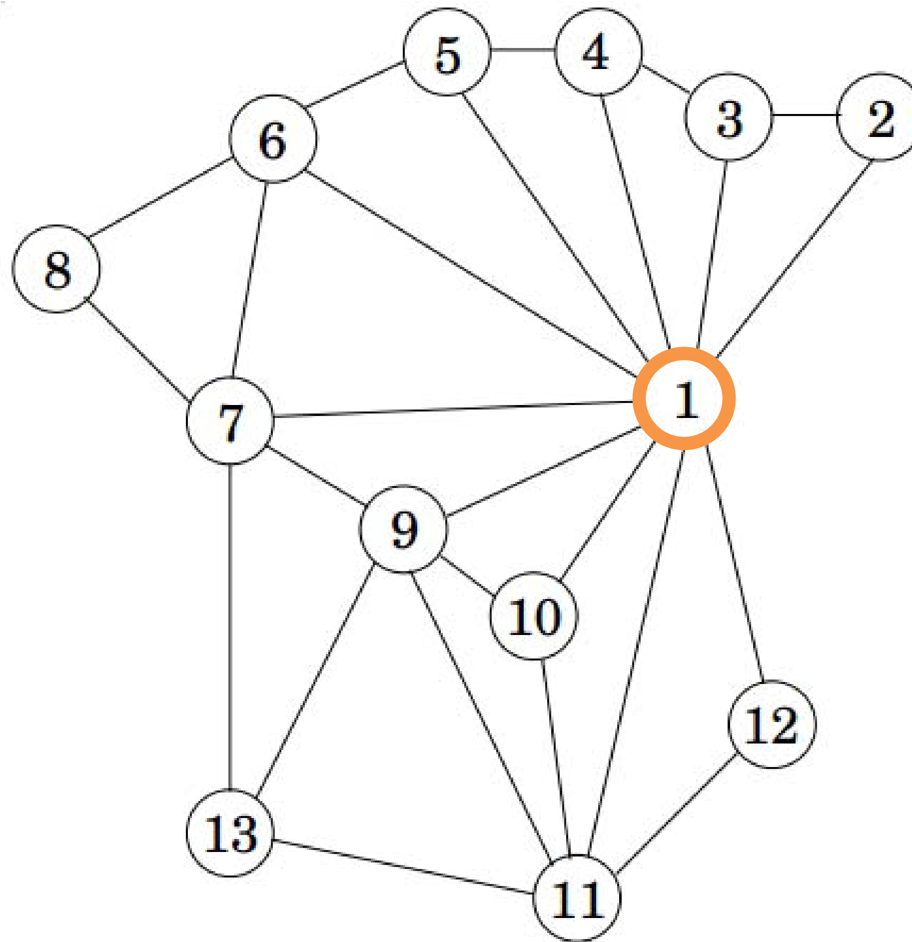
- Explore(G,u) =

```
visited(u) = true
previsit(u)
For each edge (u,v) where not visited(v):
   explore(v)
postvisit(u)
```

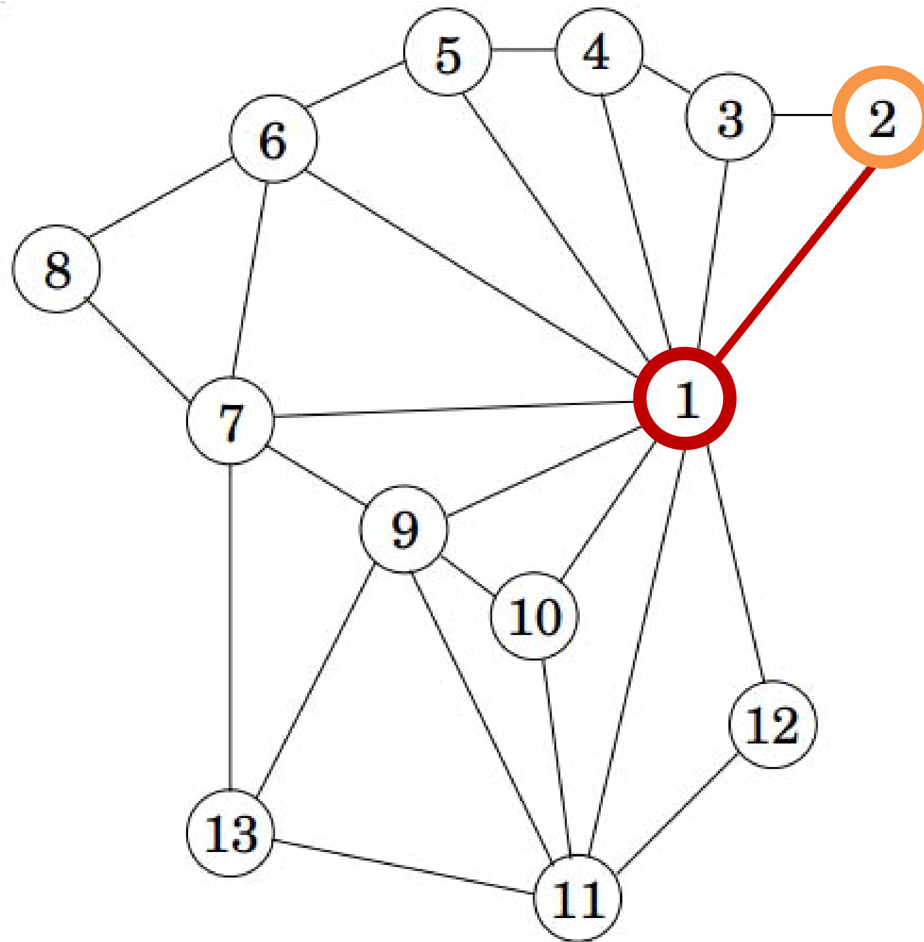
Running Time

- Same as BFS, so $O(|V|+|E|)$

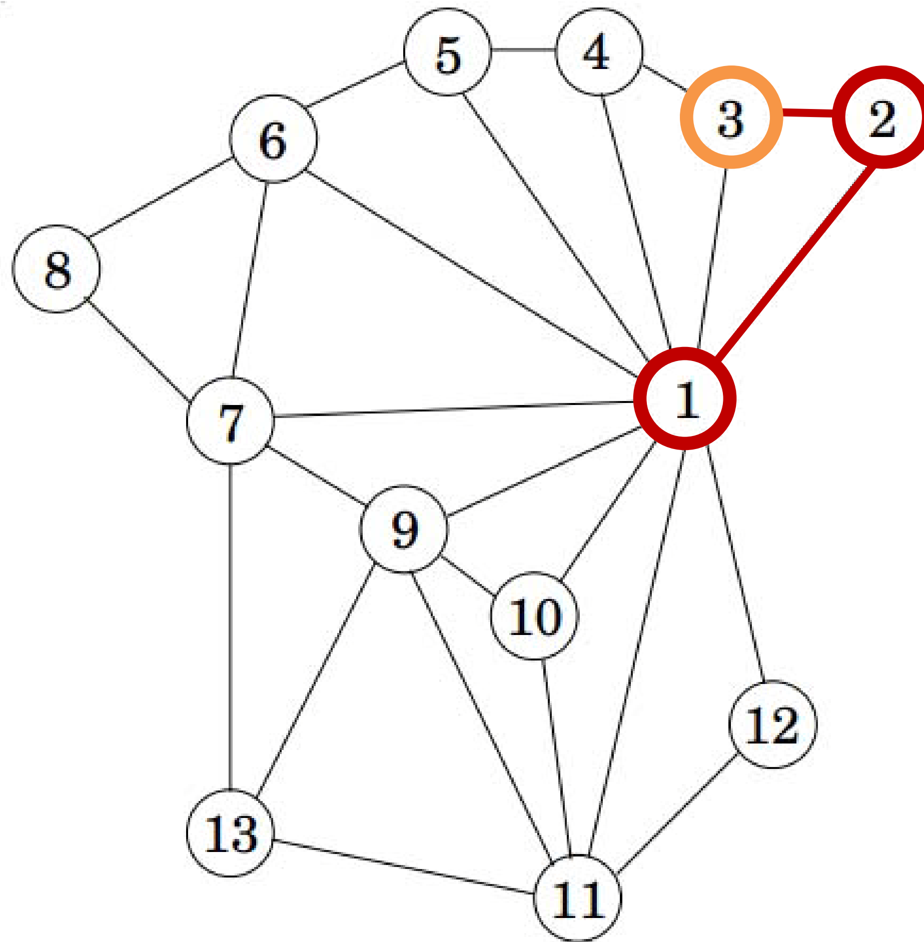
DFS Example



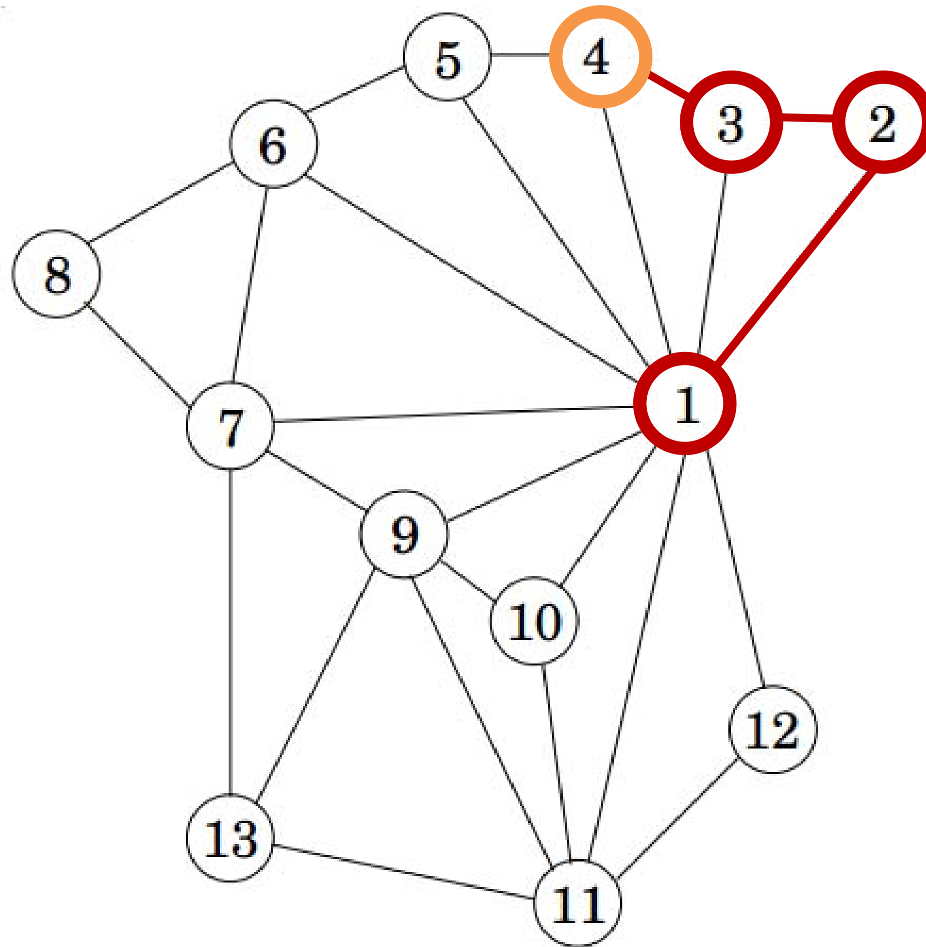
DFS Example



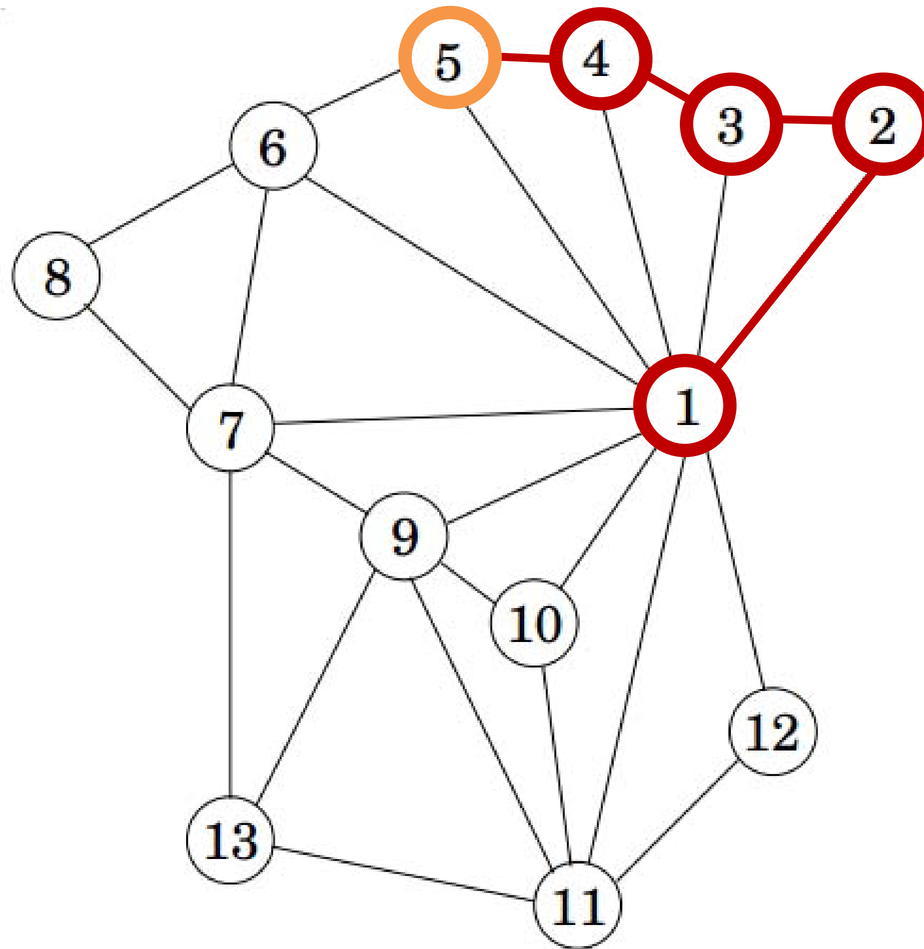
DFS Example



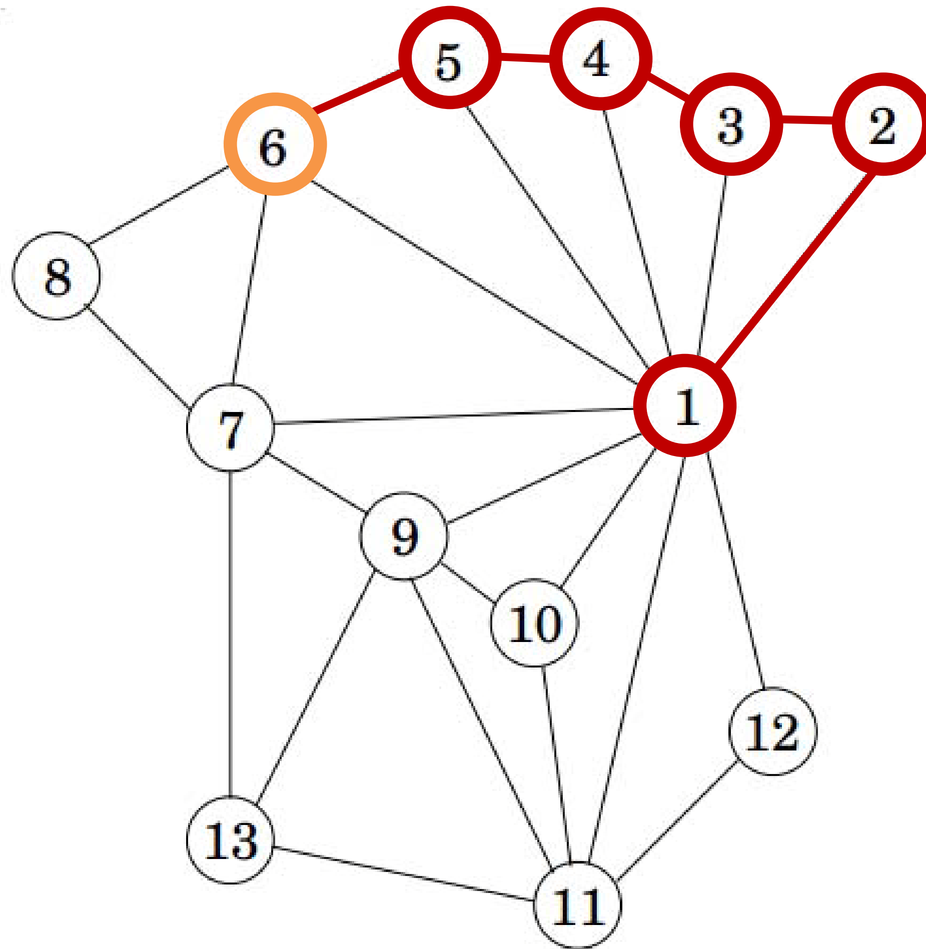
DFS Example



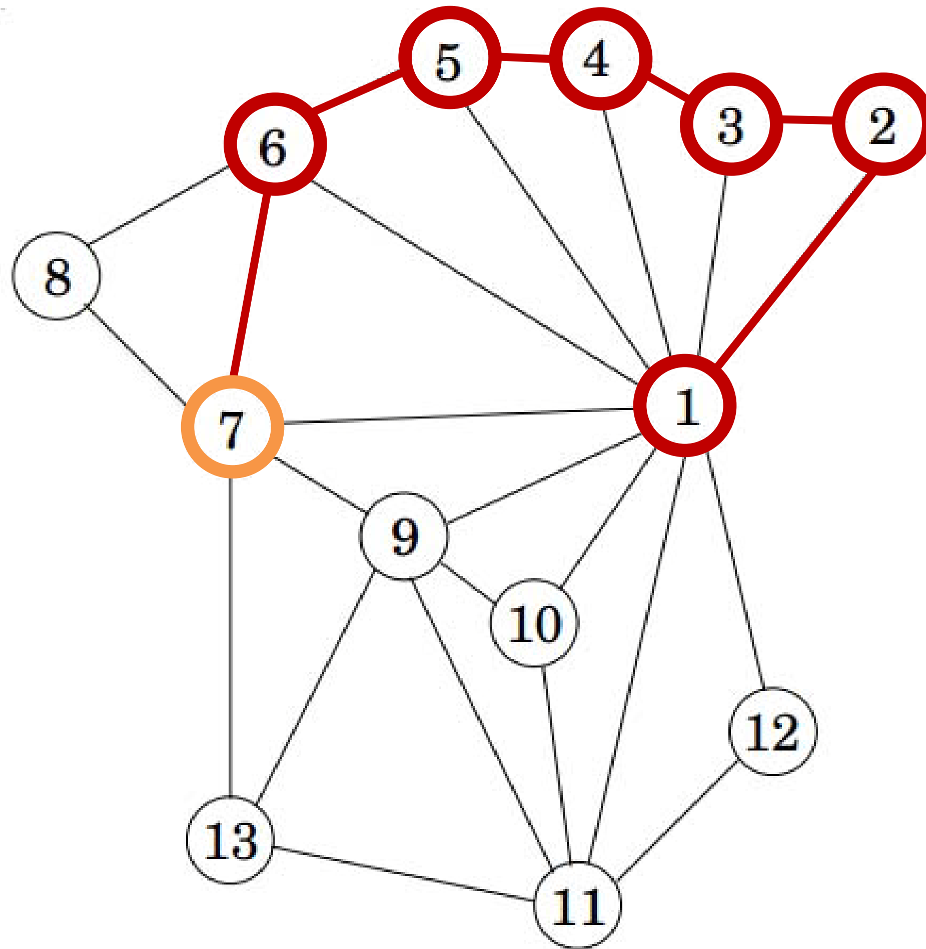
DFS Example



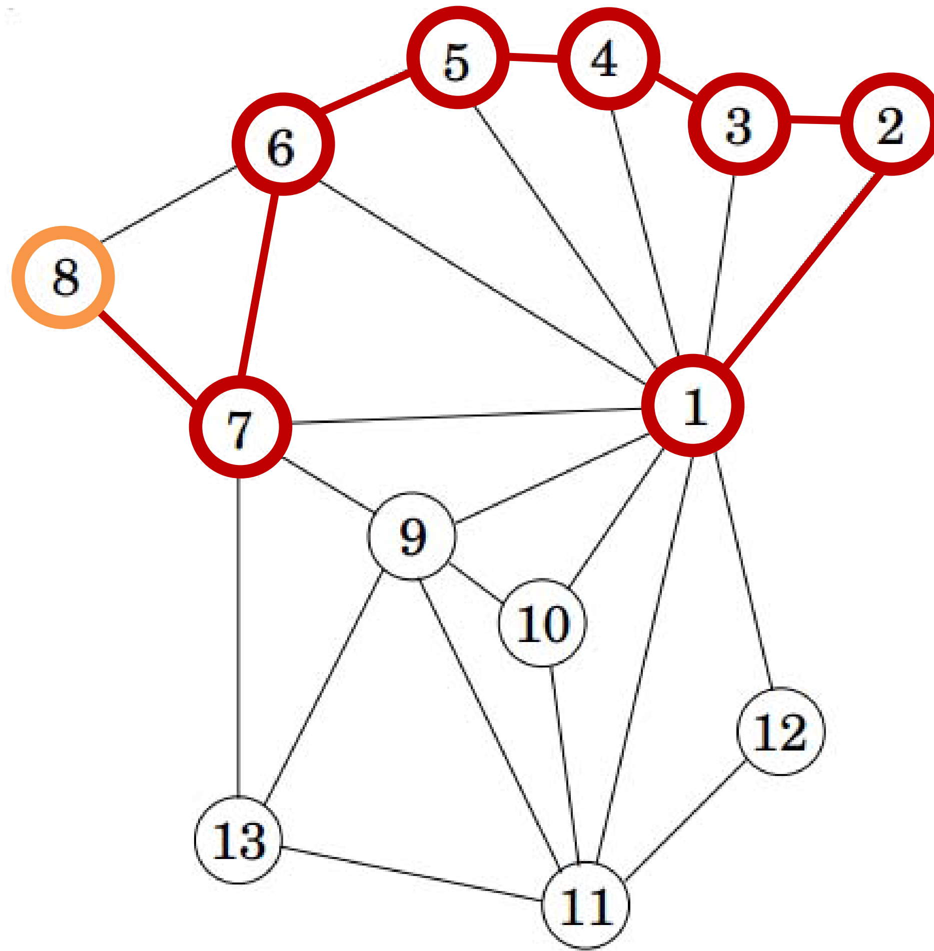
DFS Example



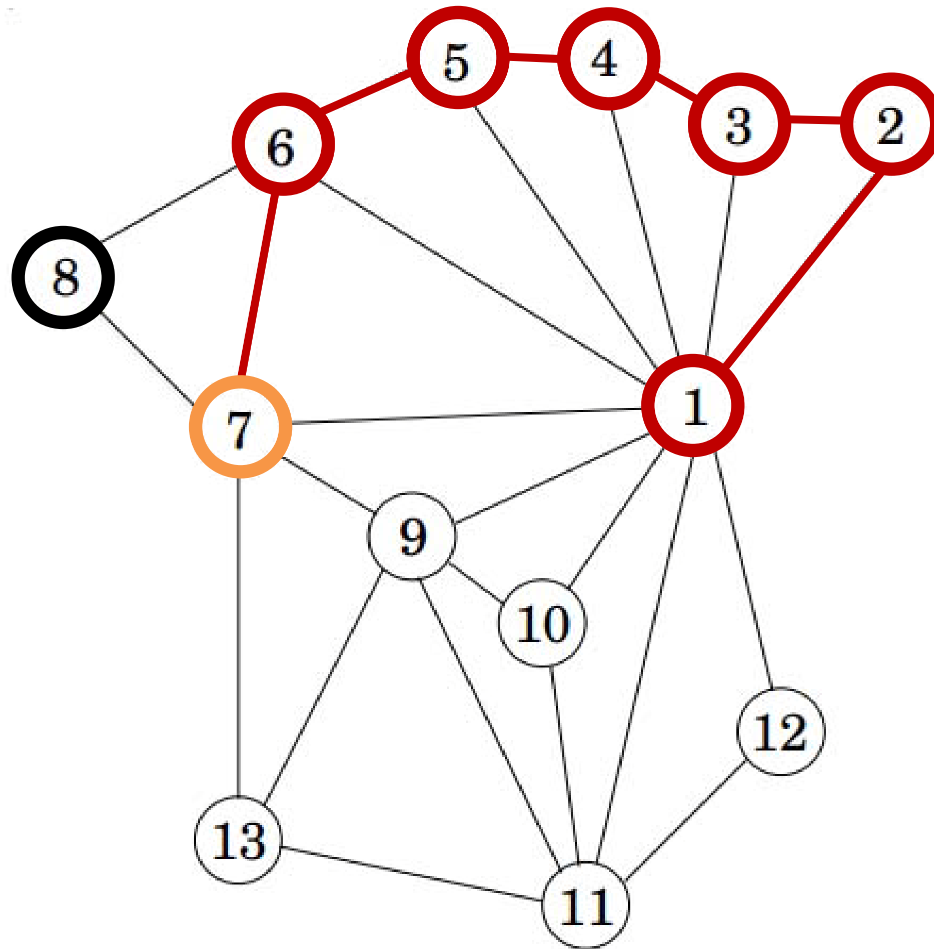
DFS Example



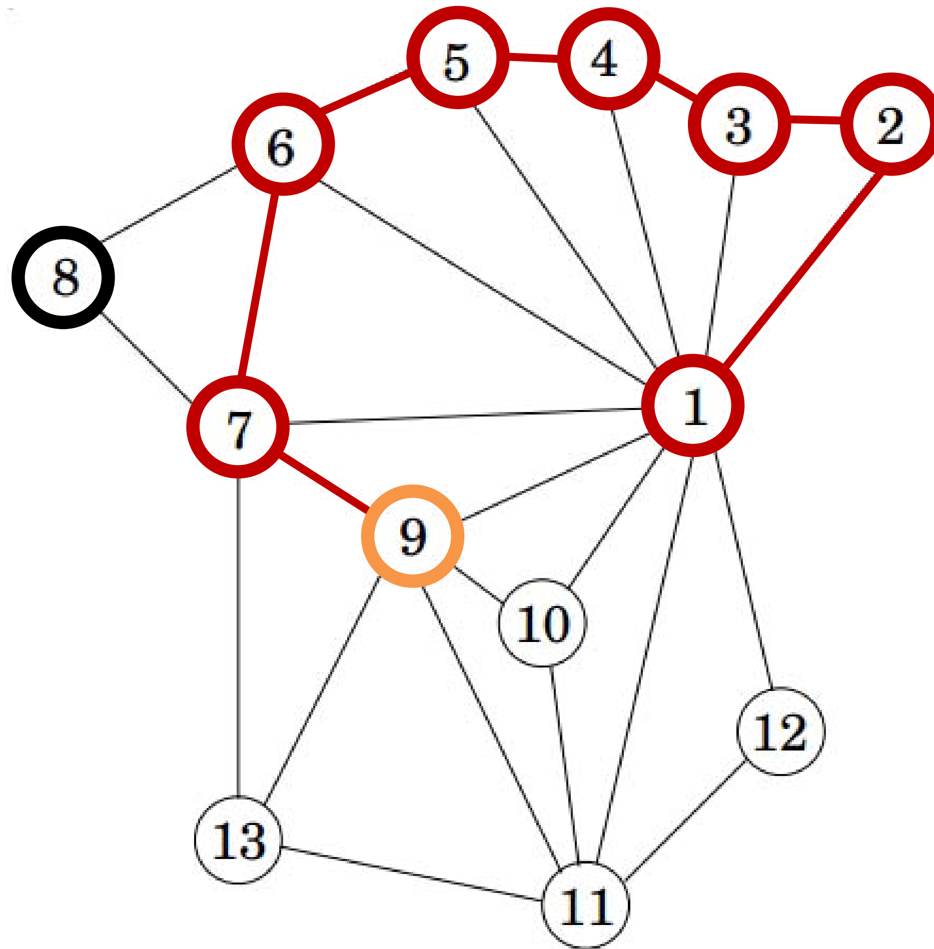
DFS Example



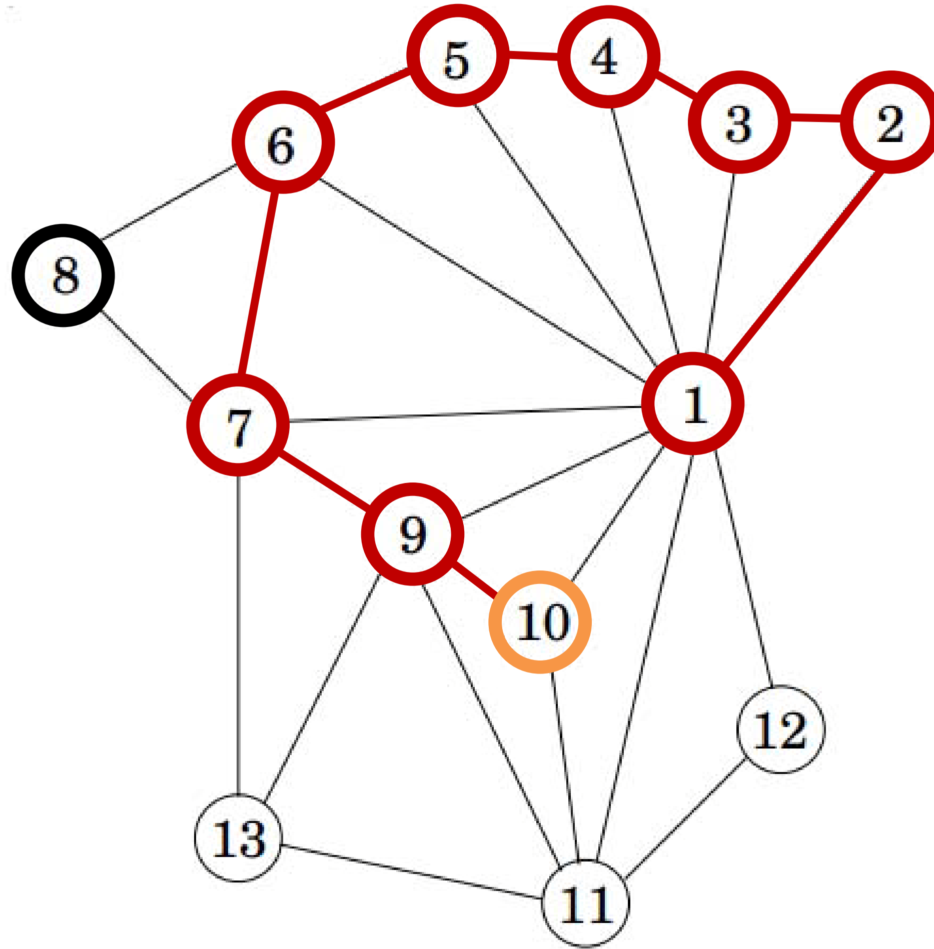
DFS Example



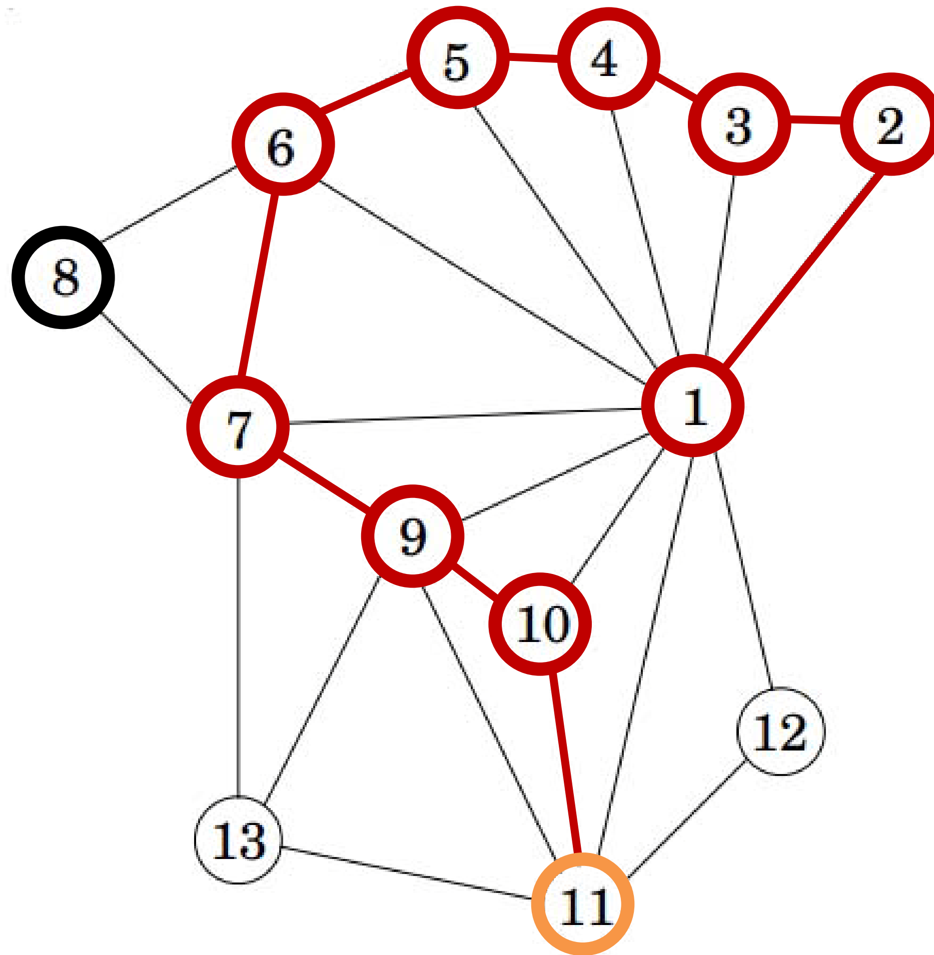
DFS Example



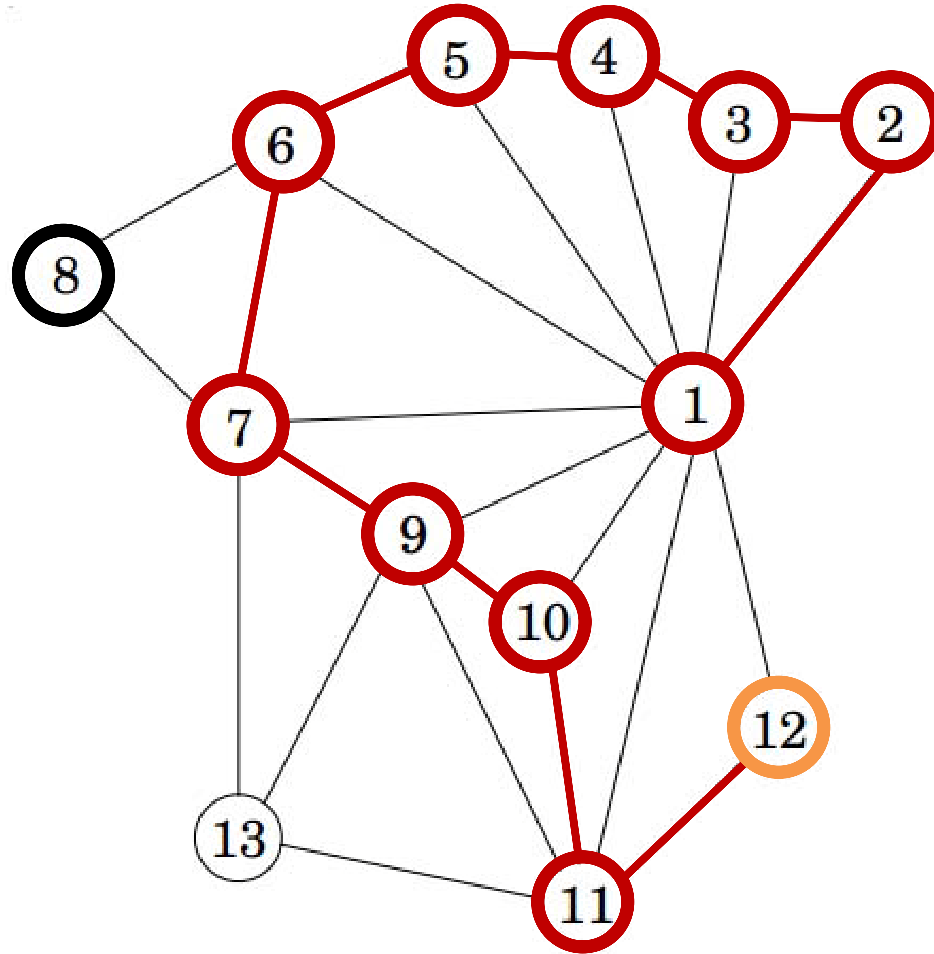
DFS Example



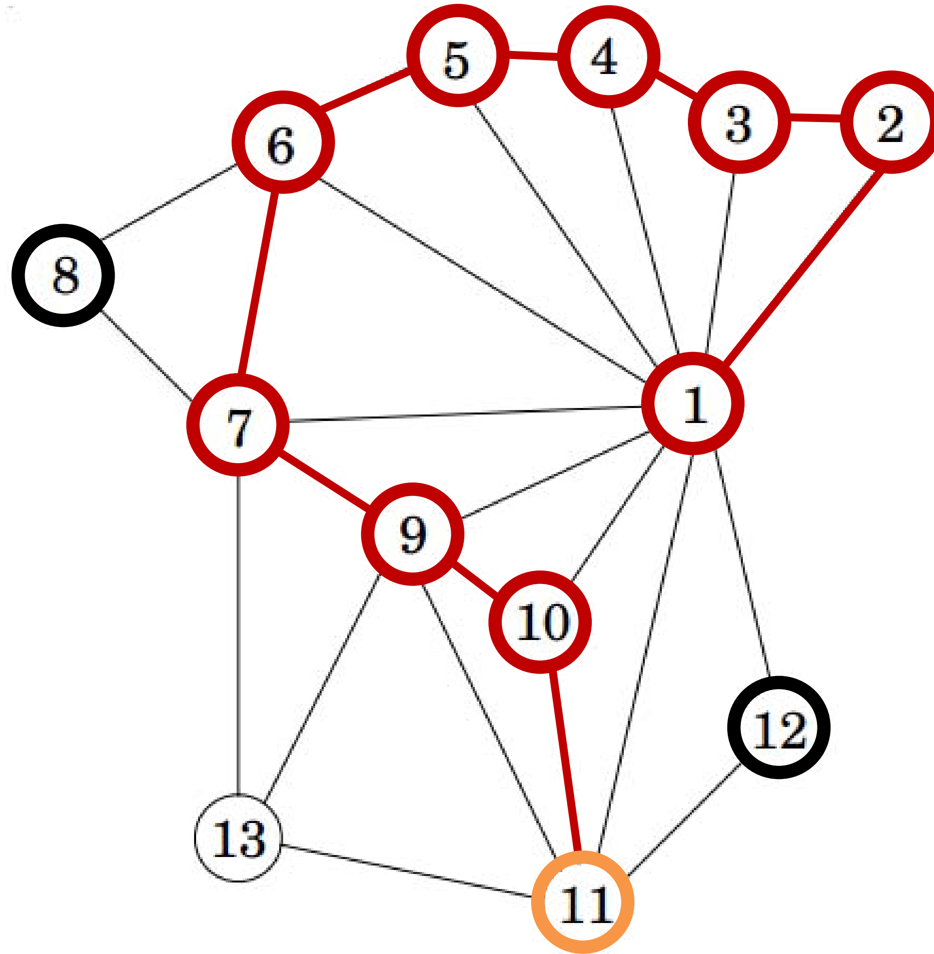
DFS Example



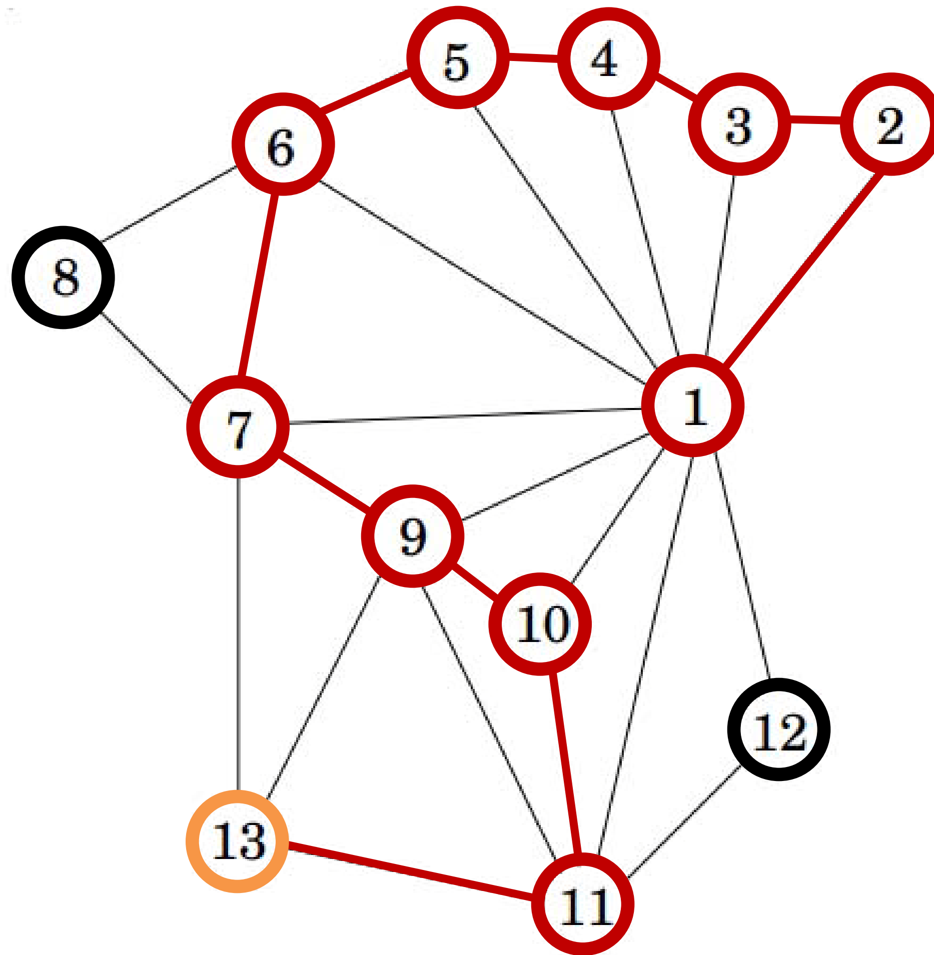
DFS Example



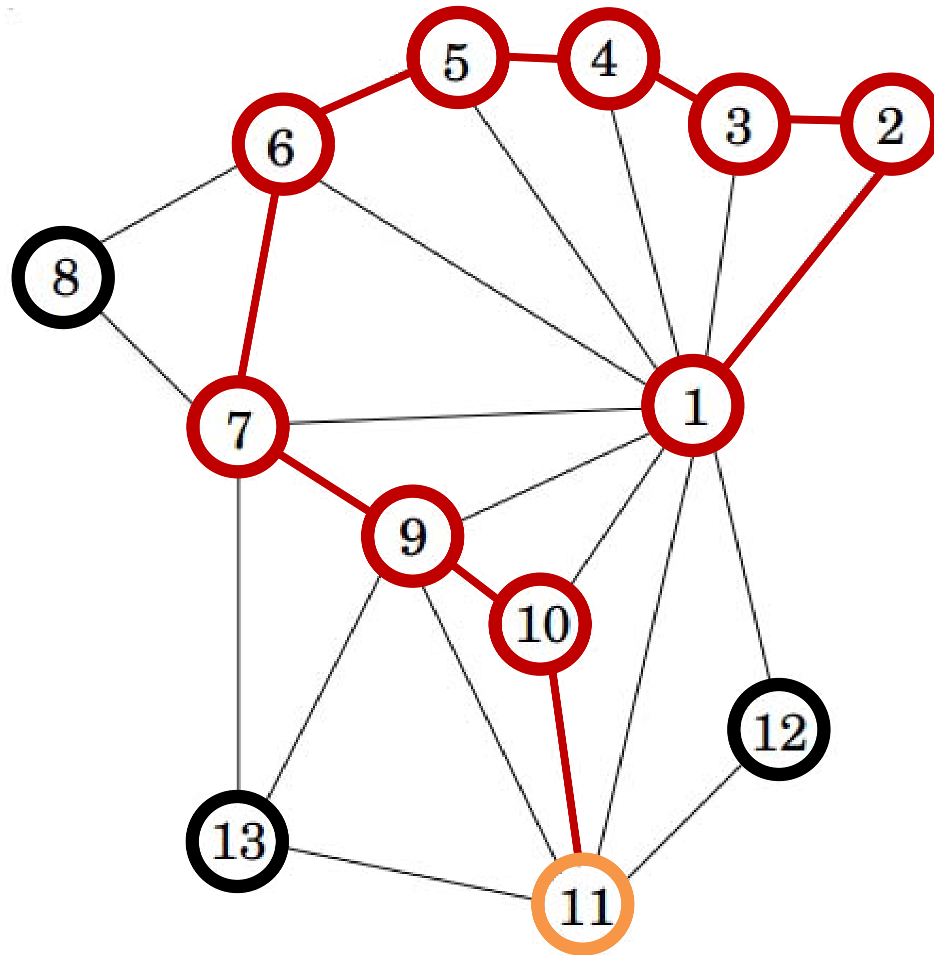
DFS Example



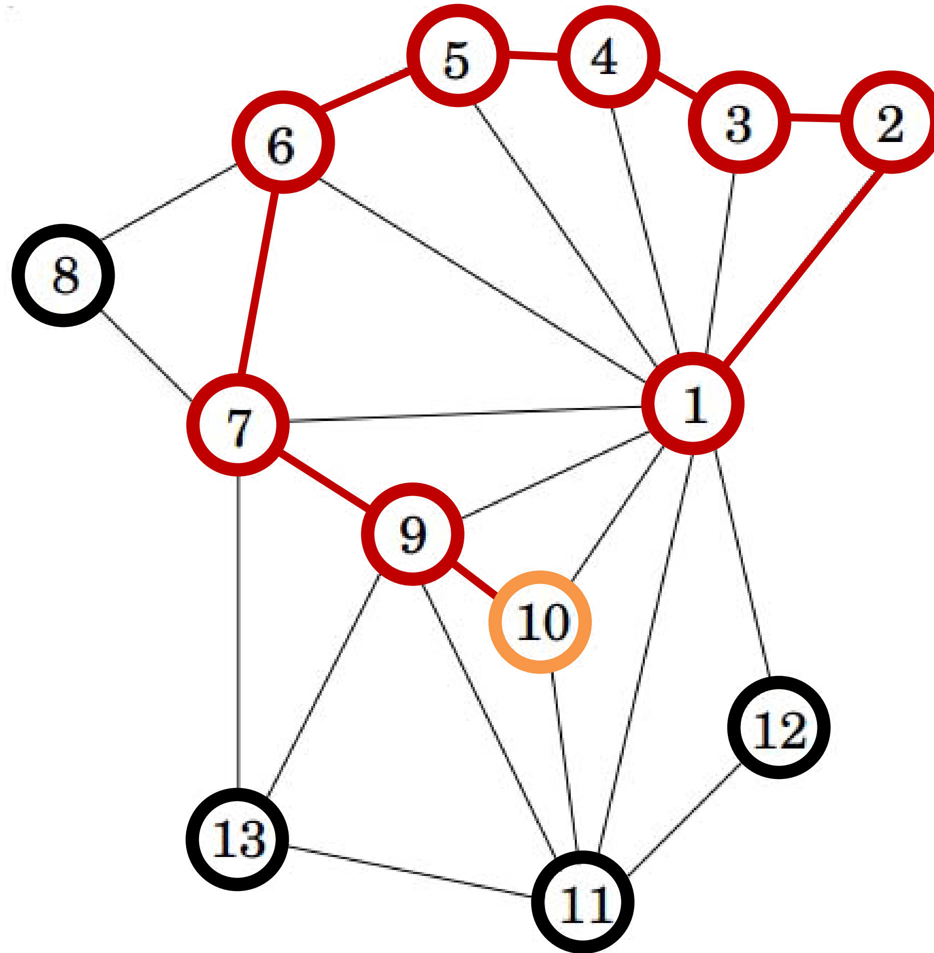
DFS Example



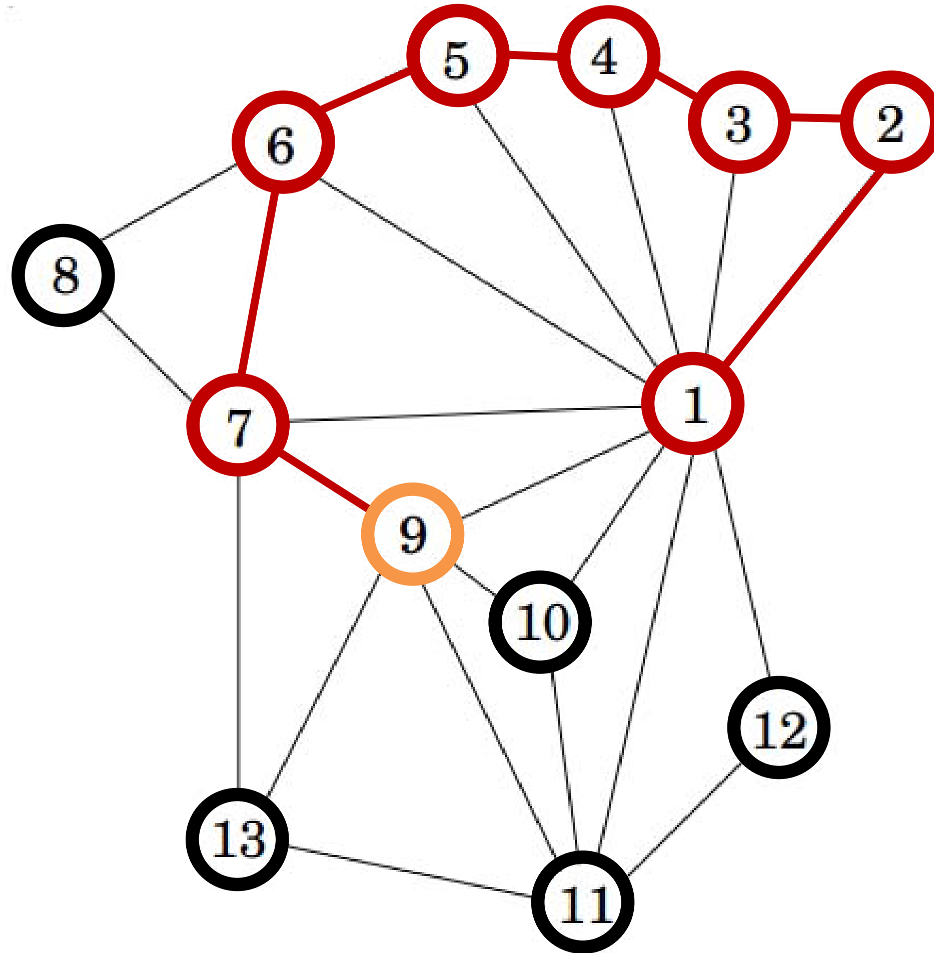
DFS Example



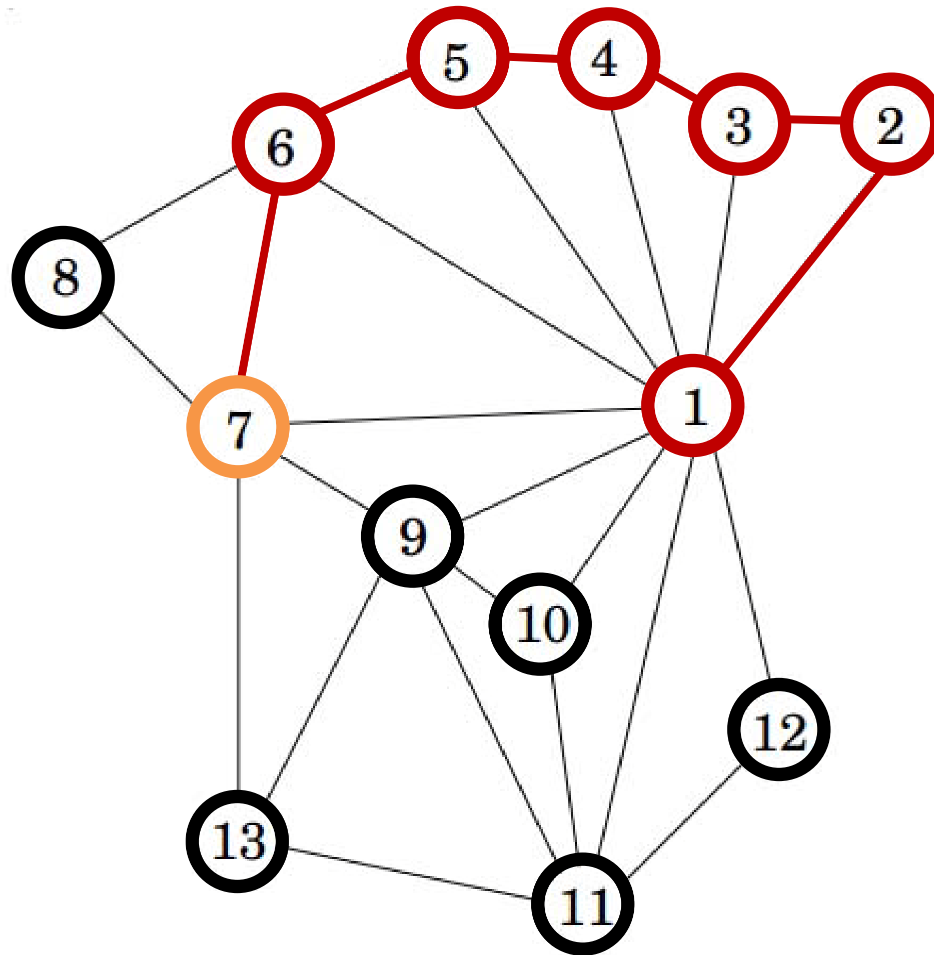
DFS Example



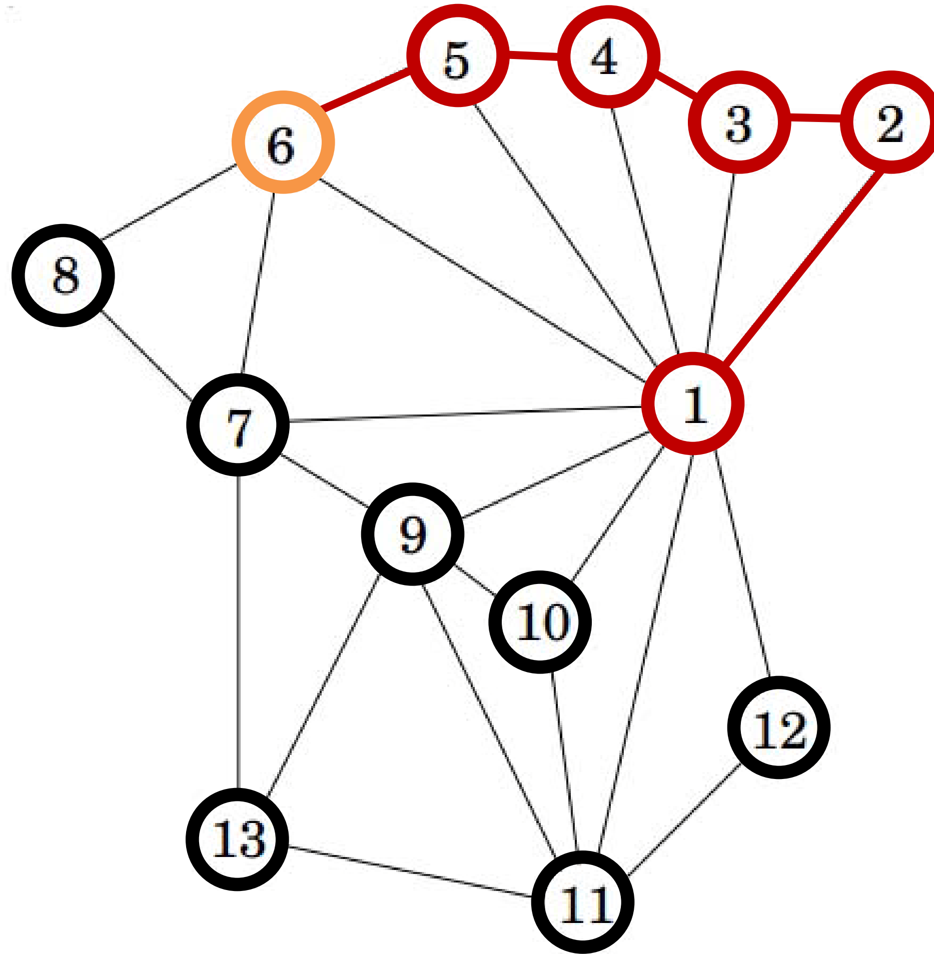
DFS Example



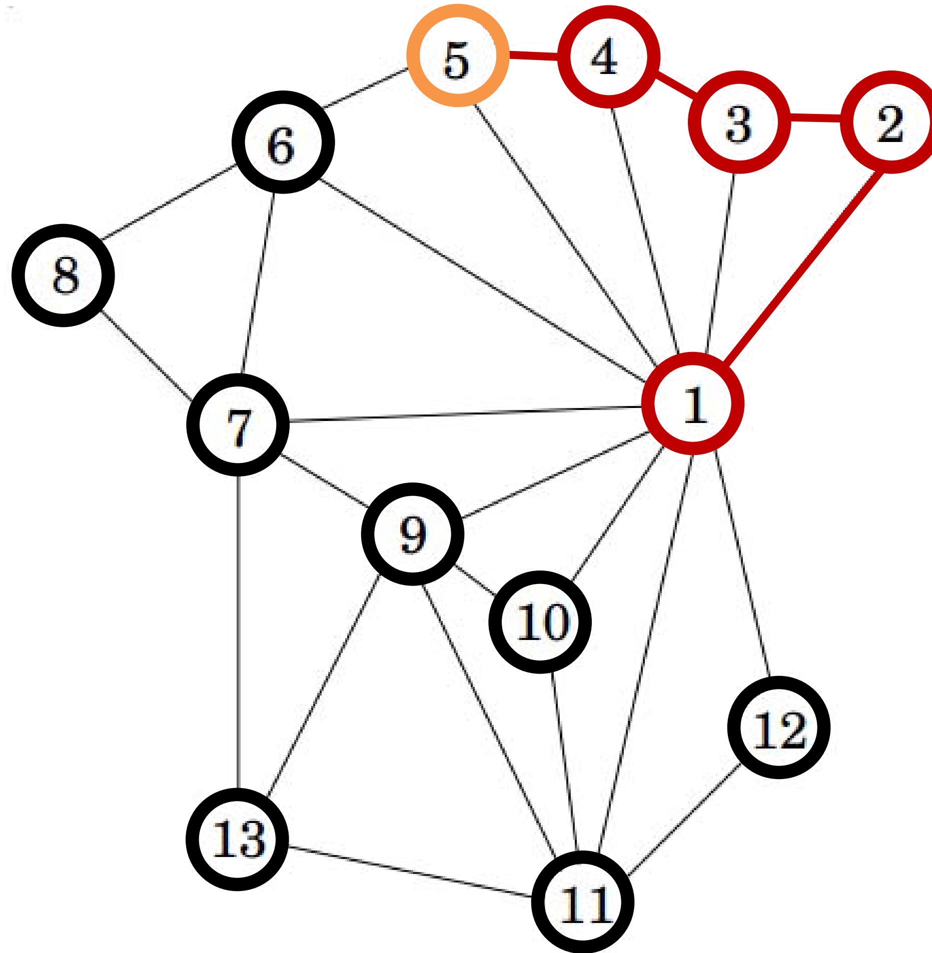
DFS Example



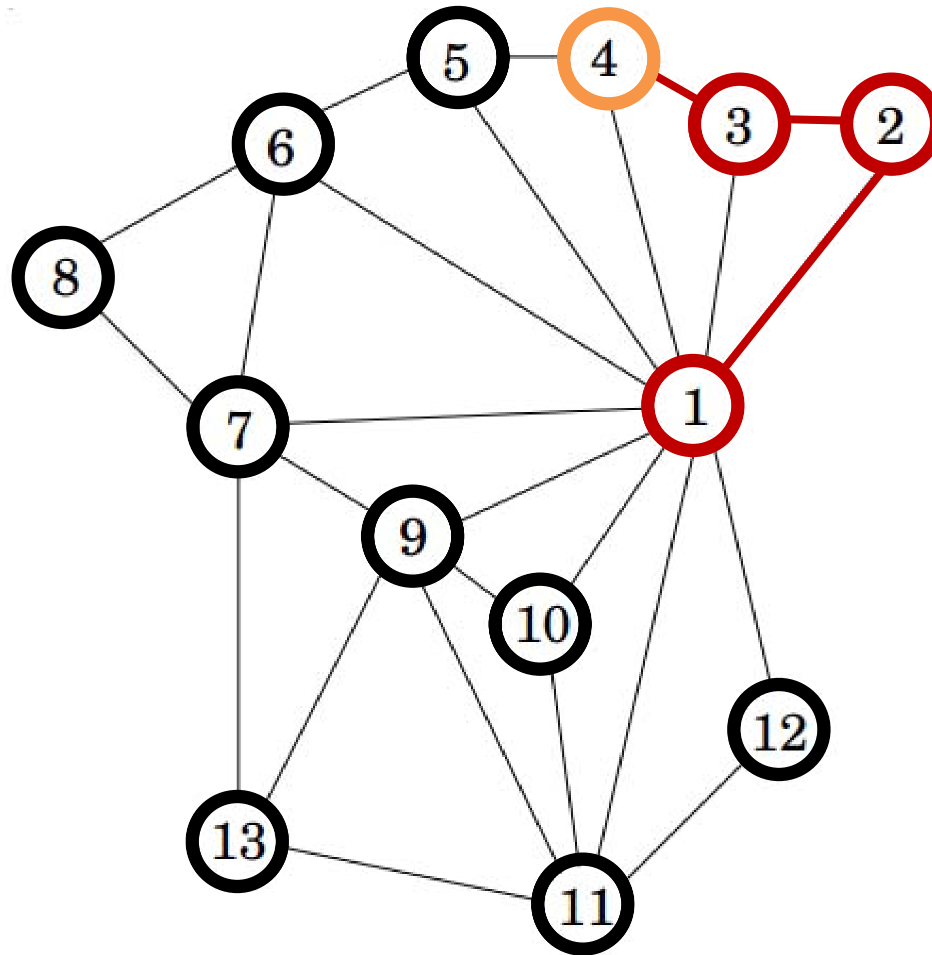
DFS Example



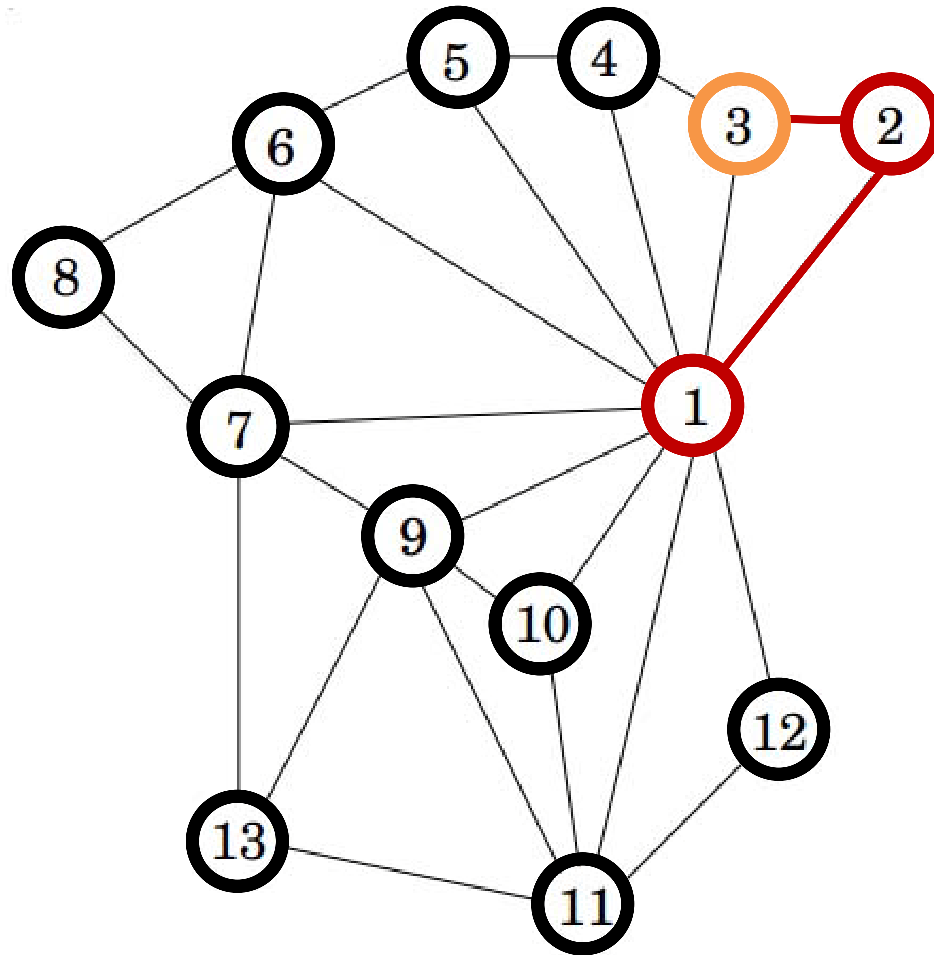
DFS Example



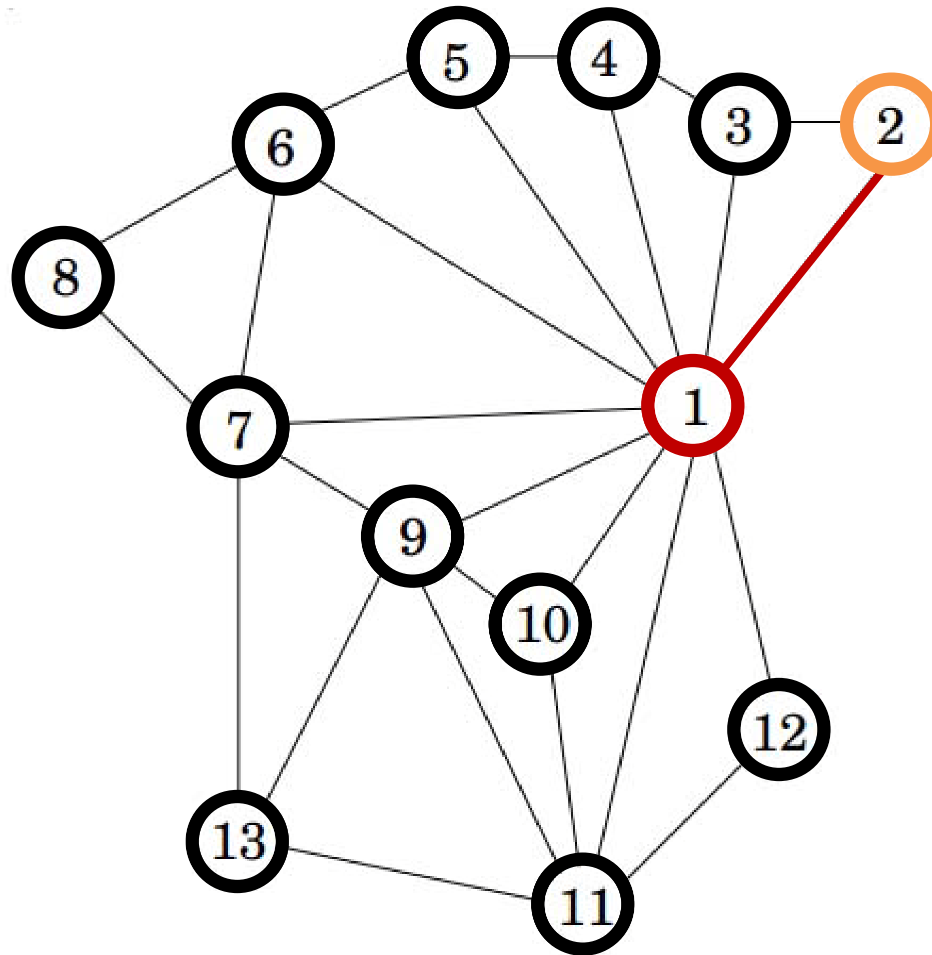
DFS Example



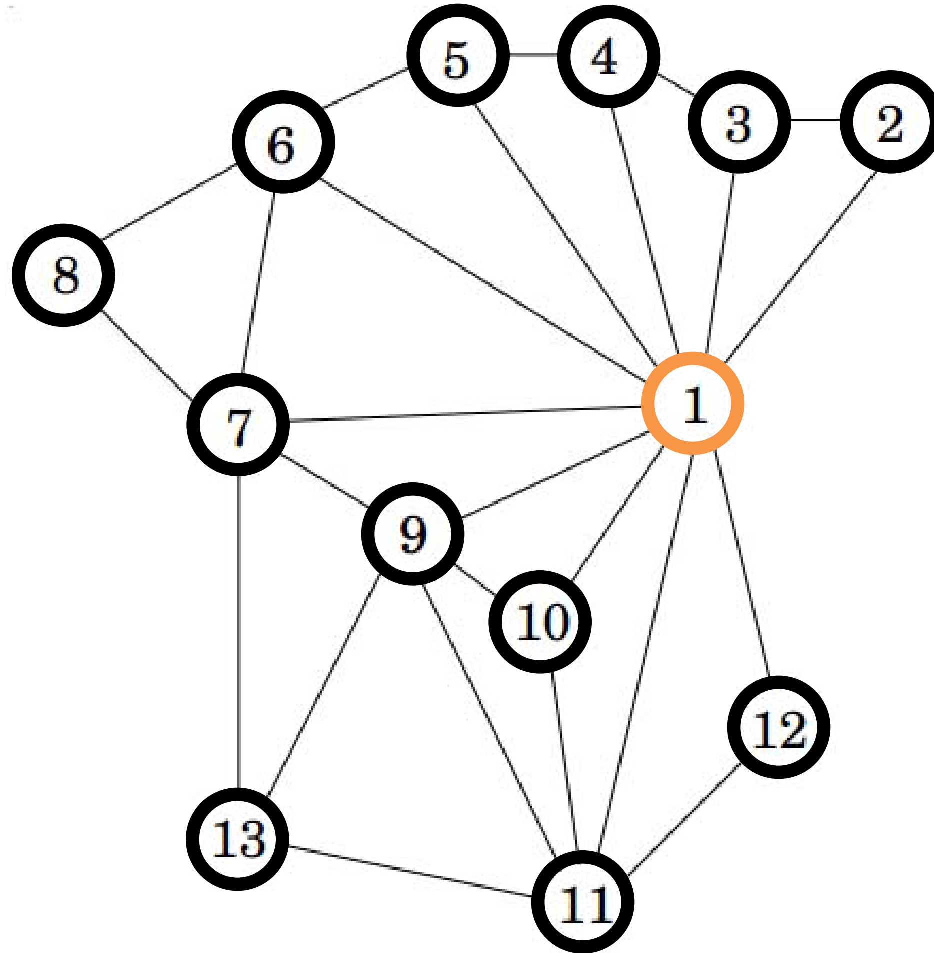
DFS Example



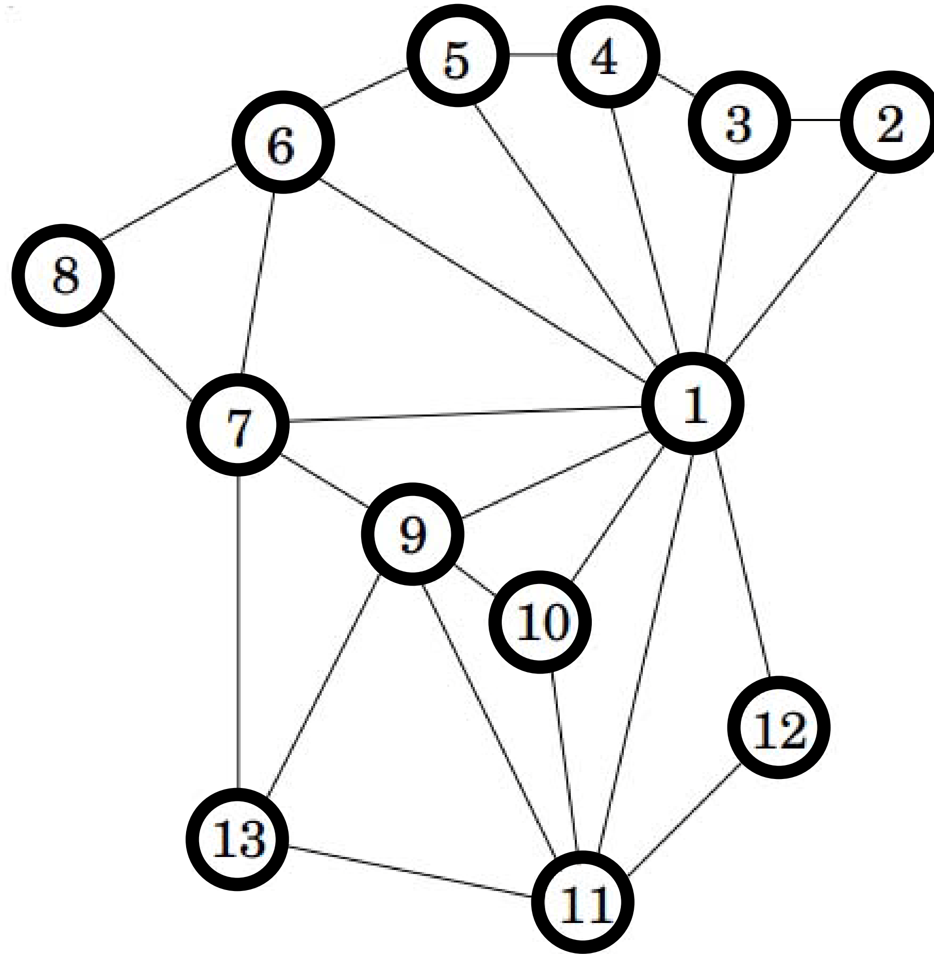
DFS Example



DFS Example



DFS Example



What does DFS do?

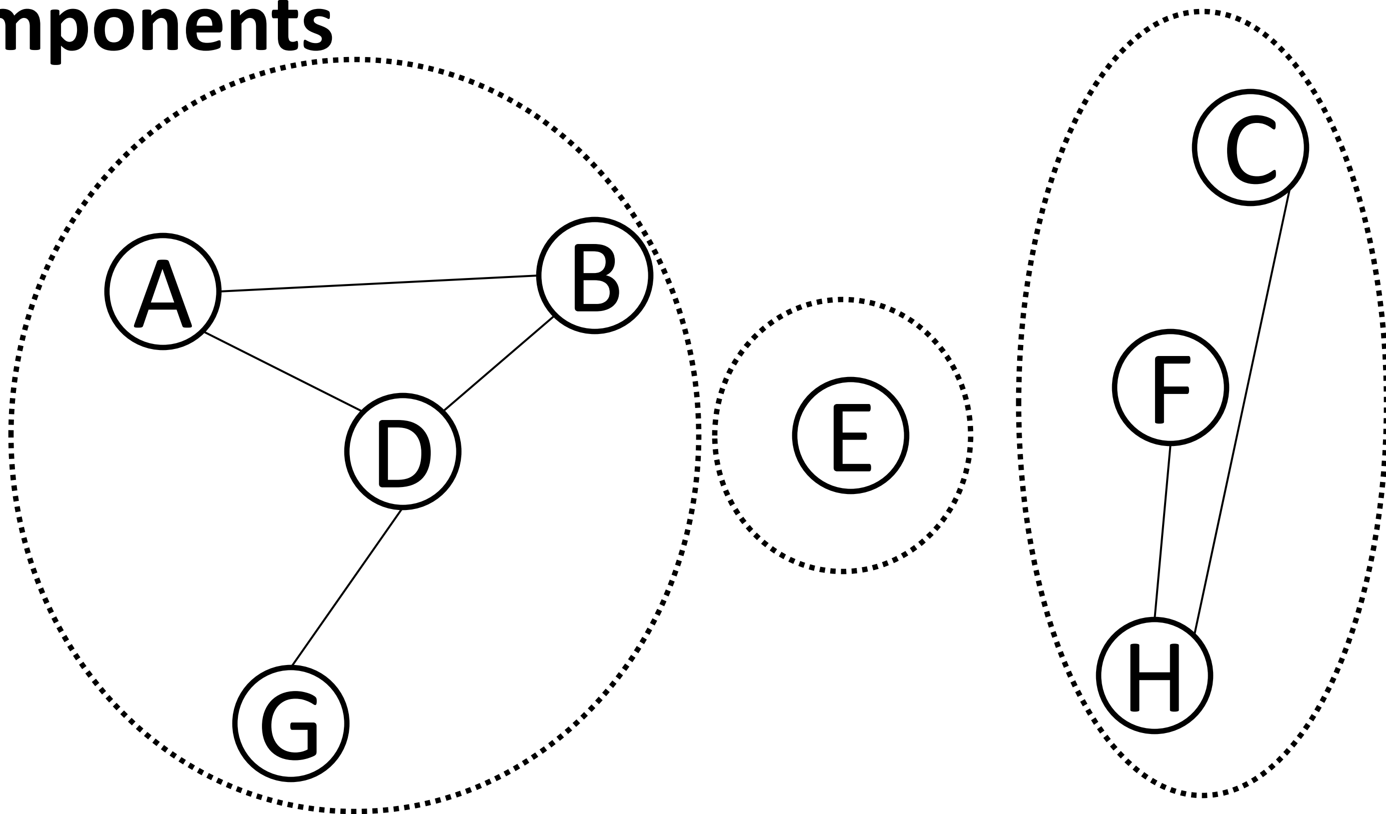
- Depends on what previsit and postvisit do!

Undirected Connectivity

- Say two nodes u, v are **connected** if there is a path from u to v
- Equivalence relation:
  - u is connected to u
  - If u is connected to v, v is also connected to u
  - If u is connected to v and v is connected to w, then u is connected to w

Undirected Connectivity

- The equivalence relation induces equivalence classes on graph. Called **connected components**



What does Explore Do?

- Visits exactly the nodes in the connected component containing v
- How do we get all of the connected components?
  - Run explore from all nodes

DFS

- DFS(G) =

  initialize()

  visited(v) = false for all v

  For all v,

      If not visited(v):

        update()

        explore(G,v)

- How do we set initialize(), update(), previsit(v), and postvisit(v) to get connected components?

Running Time

- Assume initialize is O(|E|+|V|), update() is O(1)
- Call explore once per node.
- Each call to explore visits all the edges from that node
- Each node visited once, each edge twice
- Constant time per visit to node/edge
- Total running time O(|V|+|E|)

How to Find Connected Components?

- initialize(): cc = 0
- update(): cc = cc + 1
- previsit(v): ccnum(v) = cc
- postvisit(v): do nothing