Chapter 7
Elementary Graph Algorithms

References:

[KT 3]

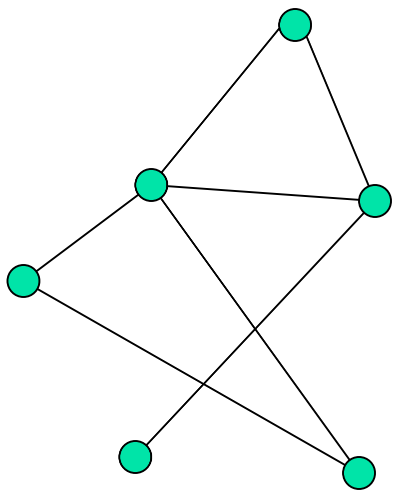
[CLRS 22.1-22.3, 22.5]

[DPV 3, 4.1-4.2]

[SSS 5]

What is a Graph?

- A graph G = (V, E) consists of:
  - V: a set of *vertices*
  - E: a set of *edges* joining the vertices
- Can model many situations
  - Road networks
  - Computer networks
  - Hyperlink relation between web pages
  - Social acquaintances
  - …

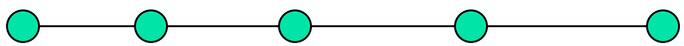


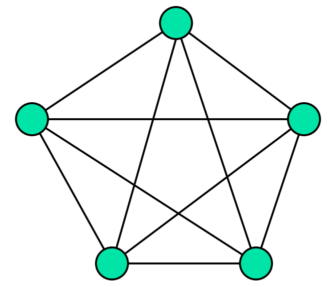
Graph Terminologies (Revision)

- n = number of vertices, m = number of edges
- *Undirected graphs* (edge no direction) vs. *directed graphs*
- *Degree* of a vertex: number of adjacent edges (in-degree, out-degree for directed graphs)
- *Path:* a sequence of edges between two vertices
- *Cycles:* A path with same starting and finishing vertex
- *Connected graph:* a (undirected) graph where any two vertices reachable by a path
- *Tree:* a connected graph with no cycles

Graph Properties

- For a connected graph, $n - 1 \leq m \leq n(n - 1)/2$
  - Minimum when it is a path
  - Maximum when it is a complete graph

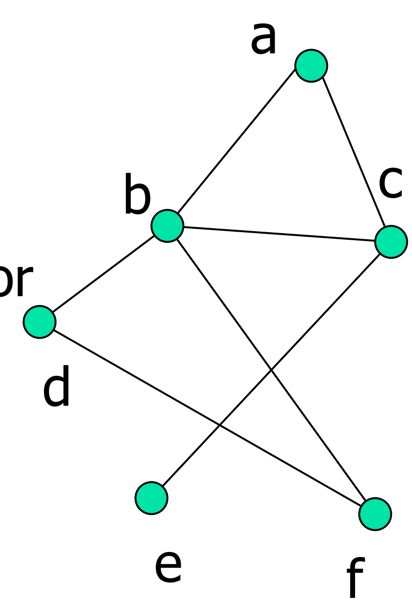


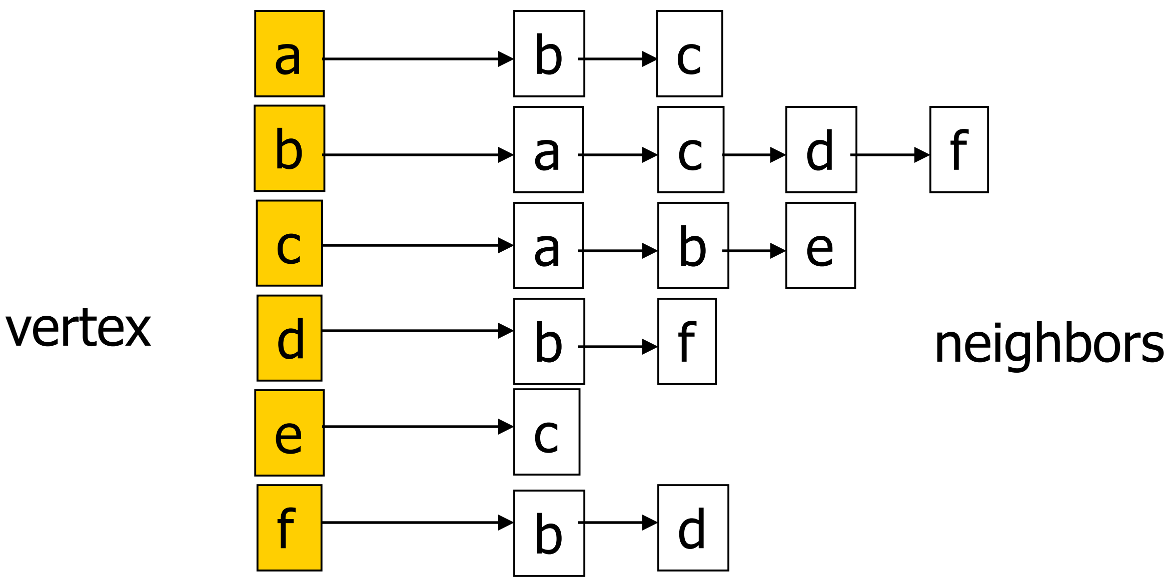


- $\Sigma \deg(v) = 2m$
  - Proof: each edge counted exactly twice

Representation of Graphs

- How to represent a graph in computers?
- (1) *Adjacency list*
  - Space: O(n+m); good for sparse graphs
  - Listing neighbors of a node: O(1) per neighbor
  - Checking two nodes are adjacent: O(deg)

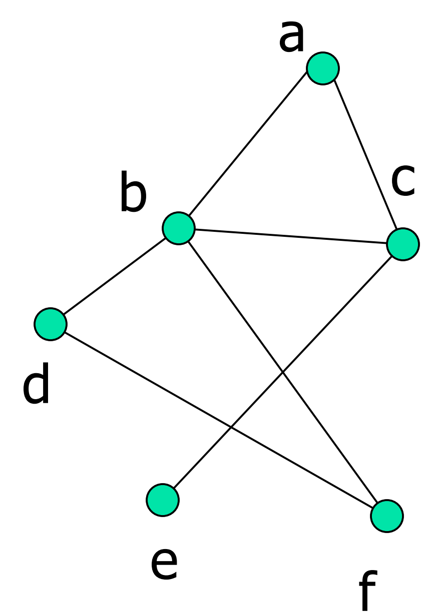




Representation of Graphs

- (2) *Adjacency matrix*
  - Space: $O(n^2)$; good for dense graphs
  - Listing neighbors: $O(n)$
  - Checking two nodes are adjacent: $O(1)$

$$
\begin{array}{c|cccccc}
 & a & b & c & d & e & f \\
\hline
a & 0 & 1 & 1 & 0 & 0 & 0 \\
b & 1 & 0 & 1 & 1 & 0 & 1 \\
c & 1 & 1 & 0 & 0 & 1 & 0 \\
d & 0 & 1 & 0 & 0 & 0 & 1 \\
e & 0 & 0 & 1 & 0 & 0 & 0 \\
f & 0 & 1 & 0 & 1 & 0 & 0
\end{array}
$$



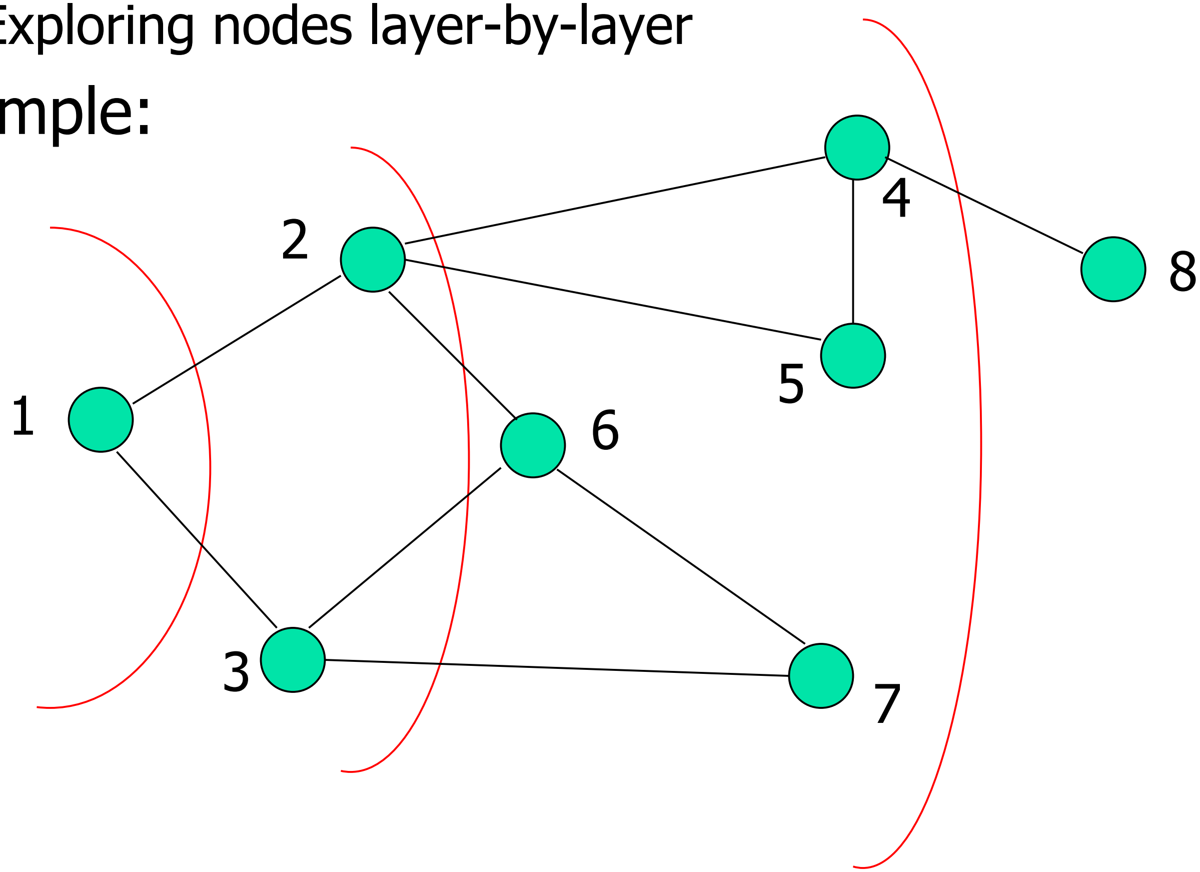
- Both representations can be naturally generalised to directed graphs and weighted graphs

Graph Traversal

- A fundamental operation: exploring a graph (visit all nodes and edges)
  - Do it in a systematic way, avoid repeating or missing
  - Many applications (e.g. game tree search in AI)
- Different approaches:
  - Breadth first search (BFS)
  - Depth first search (DFS)

Breadth First Search (BFS)

- Idea: “extending wavefront”
  - Exploring nodes layer-by-layer
- Example:



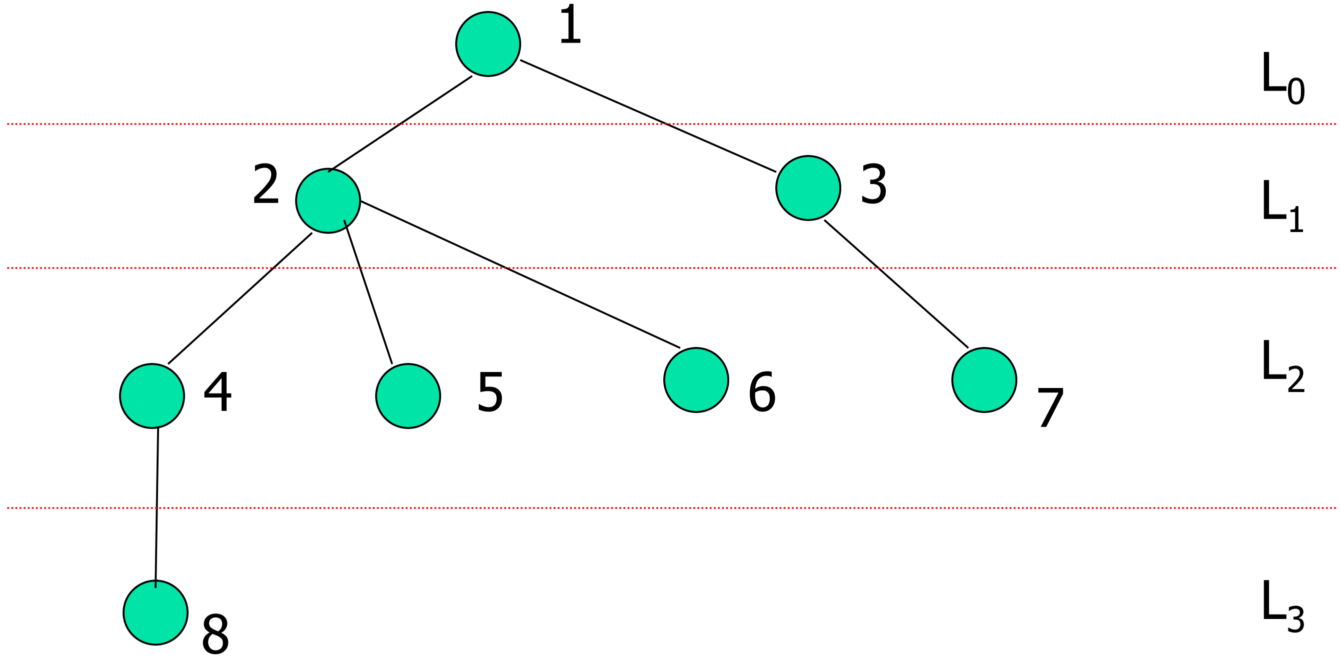
BFS (High-level)

- High-level pseudocode:
  - Note that no particular order is specified for nodes in the next level

```
/* BFS of graph G starting at vertex s */
BFS(G, s) {
  i := 0
  L_i := {s} // layer 0
  while (not all nodes explored) {
    for each edge (u, v) where u in L_i and
          v not in L_j, for some j <= i {
        add v to L_i+1
    }
    i++
  }
}
```

BFS Tree

- BFS tree: a tree showing the levels of nodes of a BFS traversal
  - On previous example:



Implementing BFS Using a Queue

- We need additional implementation details (data structure)
  - Recall: Queue is a FIFO data structure

```
BFS(G, s)
  Set Discovered[u] := false for all nodes
  Discovered[s] := true
  Enqueue(Q, s)
  while Q not empty {
    u := Dequeue(Q) // remove from head
    for each edge (u,v) {
      if (Discovered[v] == false) {
        Discovered[v] := true; Enqueue(Q,v)
      }
    }
  }
```

Running Time of BFS

- Example operation:

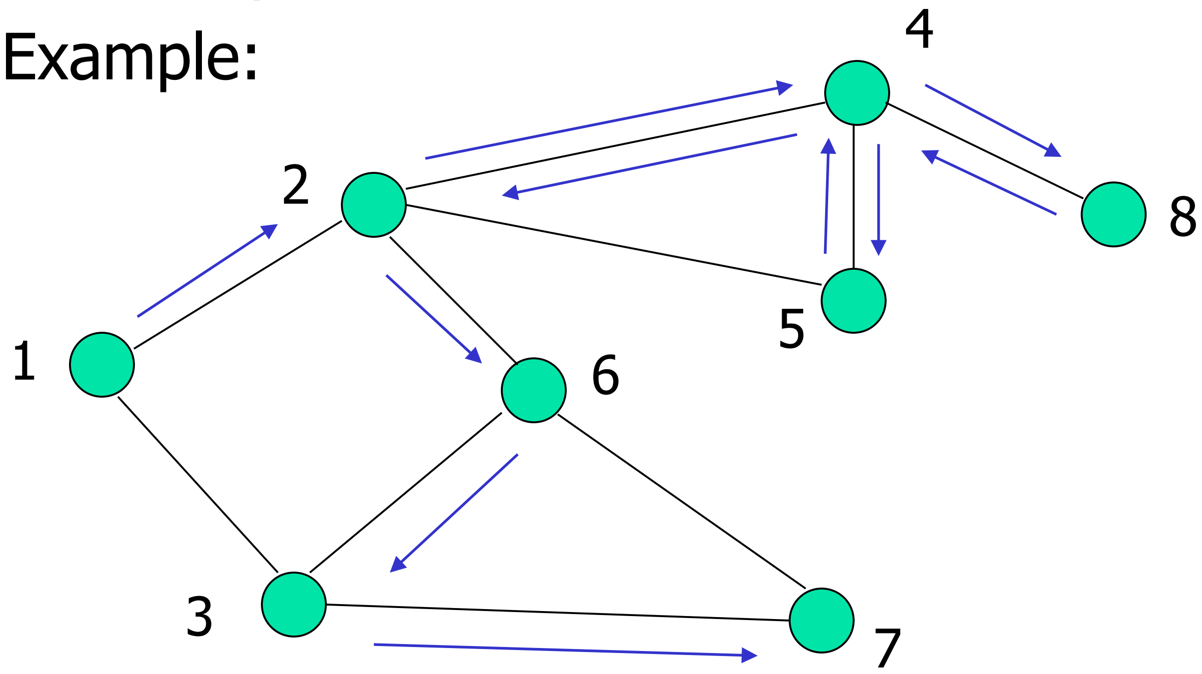
1 → 2 3 → 3 4 5 6 → 4 5 6 7 → ...

Children of 2 = {4, 5, 6}

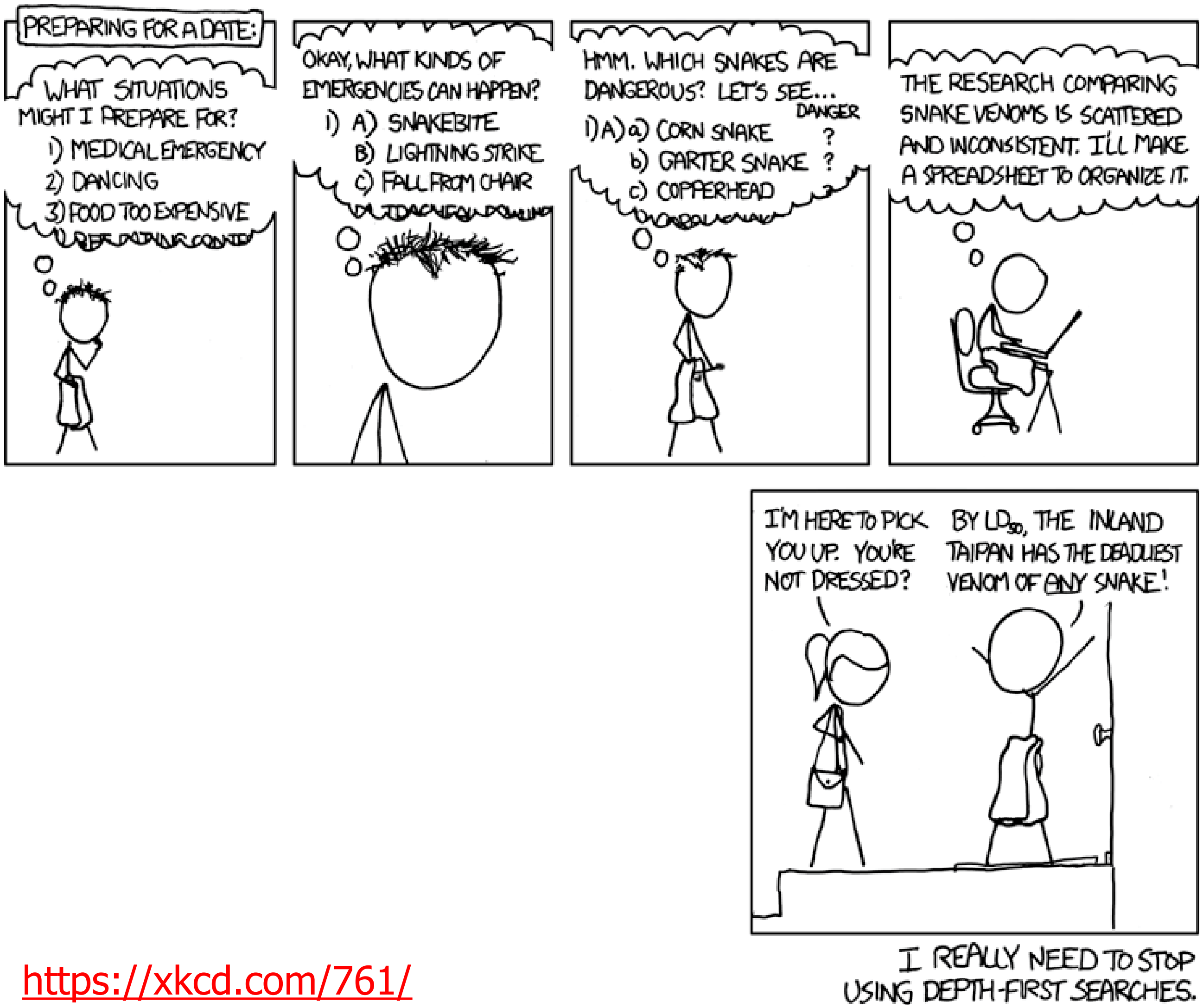
- Running time?
  - While loop executed at most n times (each node enqueued once and dequeued once)
  - All for loops: goes through each of deg(v) neighbors (using adjacency list), total Σ deg(v)
  - But Σ deg(v) = 2m
  - Therefore total O(n + m) time

Depth First Search (DFS)

- Idea: explore as deep as possible, only retreat if necessary
- Example:



DFS: Another Example



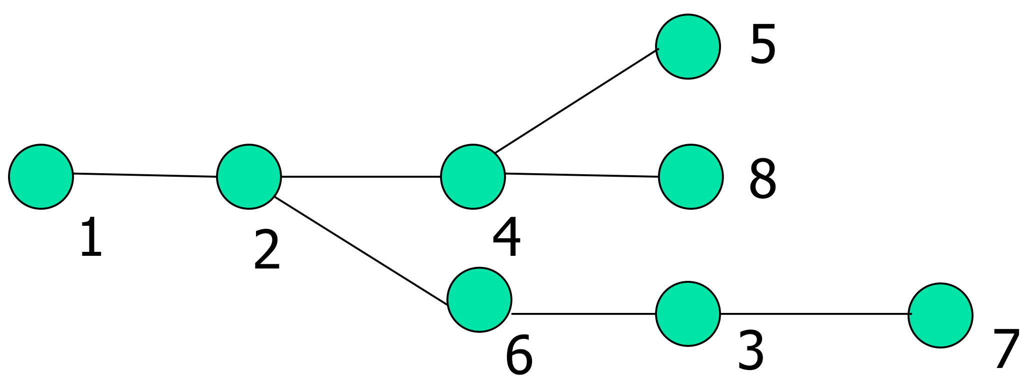
https://xkcd.com/761/

DFS algorithm, DFS tree

- A recursive formulation:

```
Mark all nodes as unexplored
DFS(G, s)
  Mark s as explored
  for each edge (s,v) {
    if (v is unexplored) DFS(G, v)
  }
```

- Similar to BFS, we can construct a DFS tree:



Implementing DFS using a Stack

- Recall: a stack is a LIFO data structure

```
DFS(G, s) {
  Set Explored[u] := false for all nodes
  Push(S, s)  // S is a stack
  while S is not empty {
    u := Pop(S) // remove from top of S
    if (Explored[u] == false) {
      Explored[u] := true
      for each edge (u,v) {
        if (Explored[v] == false) Push(S, v)
      }
    }
  }
}
```

Running Time of DFS

- Example operation:

1	→	3 2	→	3 6 5 4	→	3 6 5 8 5	→ ...

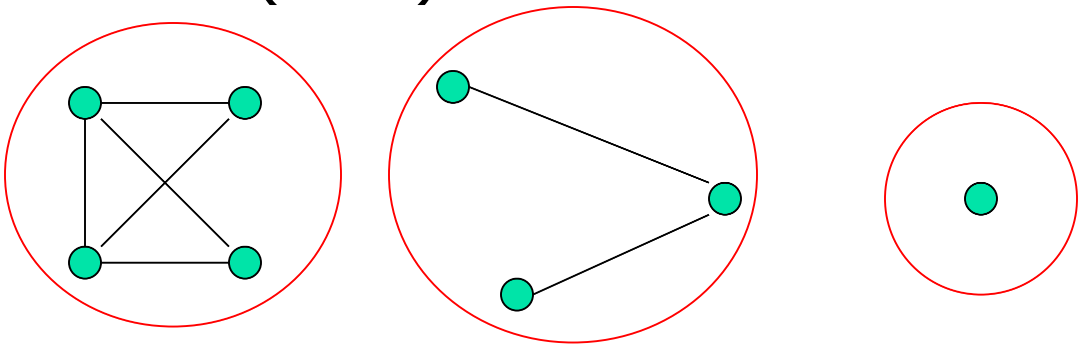
- Running time of DFS:
  - Similar analysis to BFS
  - Each push of v when u is considered correspond to an edge (u,v)
  - Total number of pushes = O(m)
  - Each stack operation takes O(1) time
- Total runtime: O(m + n)

Comparing BFS and DFS

- Similar:
  - In each step, take a vertex and find all its neighbors
  - $O(m + n)$ time
- Only difference:
  - BFS use a queue (newly discovered vertices add to end, explored last)
  - DFS uses a stack (newly discovered vertices add to front, explored first)

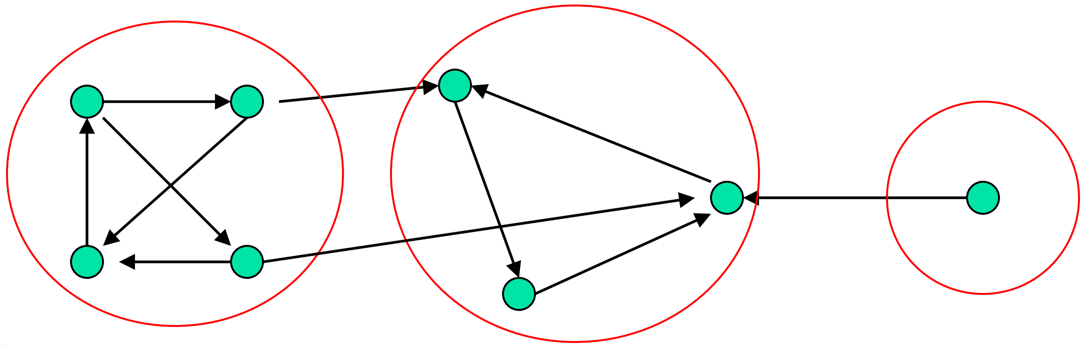
Finding Connected Components

- If the graph is connected, a BFS or DFS starting at a vertex will reach all other vertices
- However, if the graph is not connected, the traversal will only reach all vertices within the same *connected component* as the starting vertex
- Repeat for remaining vertices to identify other remaining components
- Total time O(m+n)



Strongly Connected Components

- A directed graph is *strongly connected* if any two vertices u and v are mutually reachable by some paths
- The *strongly connected components* (SCC) of a graph are the subgraphs all of which are strongly connected
- Example:



- We want to:
  - Test whether a given graph is strongly connected
  - Find all SCC of a graph

SCC using Graph Traversals

- Naïve approach: perform n traversals, one starting from each vertex
  - Strongly connected iff all vertices reachable in each traversal
  - Slow
- Instead, with some tricks, two traversals are sufficient!
- Algorithm for testing strong connectivity:
  - 1) Pick any node s, run BFS on G starting from s
  - 2) Repeat above on G^{rev} (a graph obtained by reversing every edge of G)
  - 3) Report no if some nodes not reachable in either traversals, yes otherwise

More on the SCC Algorithm

- Why is this correct?
  - If it reports “no”: obvious (some vertex non-reachable)
  - If it reports “yes”, then any u and v must be mutually reachable
    - {s, u} mutually reachable, {s, v} mutually reachable
    - So u can go to v via s, and v can go to u via s
- Runtime: O(m + n)
  - Just do BFS twice
  - Reversing the graph can also be done in O(m+n) time
- Finding the actual SCC:
  - The set of nodes reachable by both traversals is a SCC containing s
  - Repeat for remaining subgraph