Graphs and Traversals

Reminders/ Check in

- **Assignment 01** due tonight at 10 pm
- Assignment 02 will be released later today
- If you haven't done so already, check out Problem Set Advice
- Take advantage of office hours today:
  - Mine: 1.30-3 pm, TAs: 3-5pm, 7-10 pm
- Questions?

- Announcements?

Today's Outline

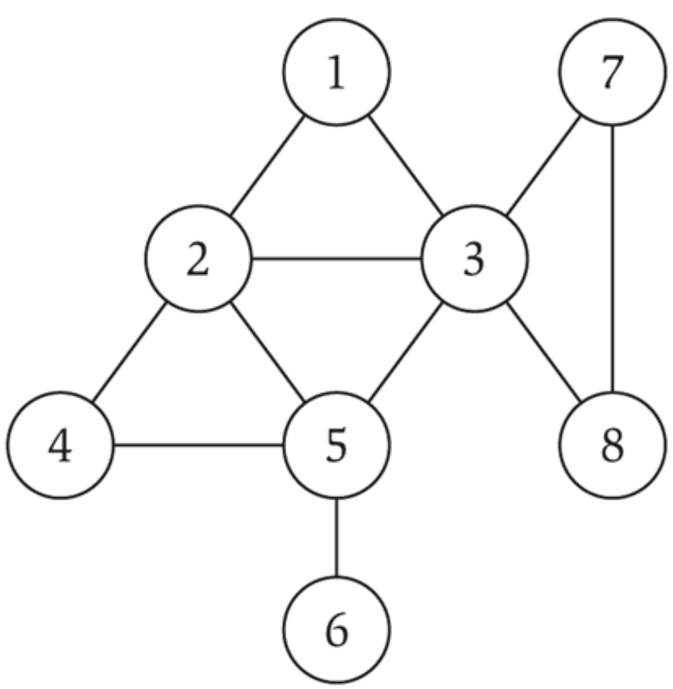
- Formal definitions of graph terms
- Review common approaches for graph representation
- Review breadth-first search
- Review depth-first search
- Search Proofs (runtime, correctness)

Review: Undirected Graphs

An undirected graph $G = (V, E)$

- V is the set of nodes, E is the set of edges
- Graph size parameters: $n = |V|$, $m = |E|$
- Sometimes we consider weighted graphs, where each edge e has a weight $w(e)$

Unweighted,
Undirected Graph



$V = \{1,2,3,4,5,6,7,8\}$

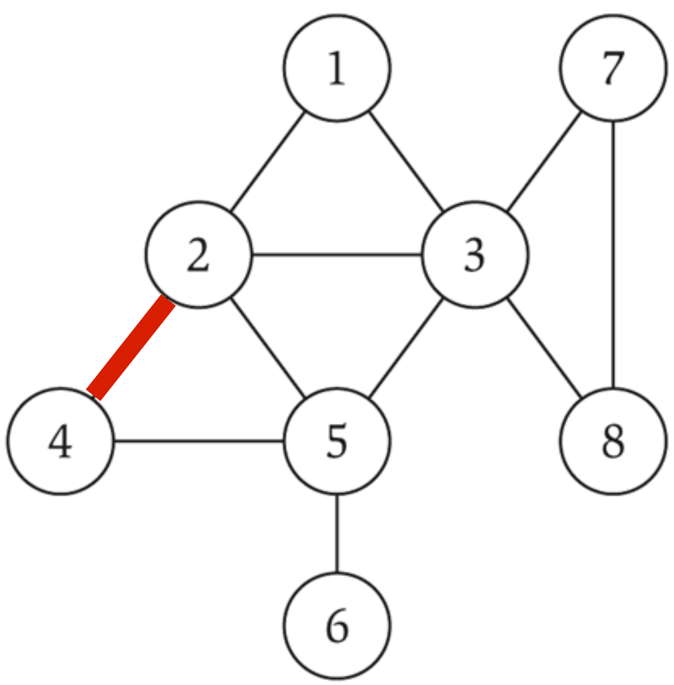
$E = \{(1,2), (1,3), (2,3), (2,4), (2,5), (3,5), (3,7), (3,8), (4,5), (5,6), (7,8)\}$

$n = 8$, $m = 11$

Representing Graphs (Review)

Option 1a: **Adjacency matrix.**

- n-by-n matrix where $A[u][v] = 1$ if $(u, v) \in E$



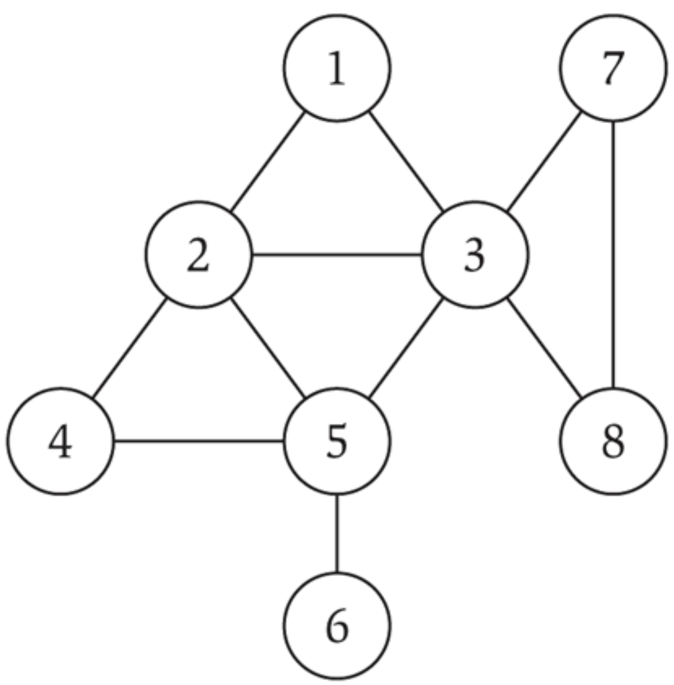
$n = |V|,\ m = |E|$

$$
\begin{array}{c|cccccccc}
 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
\hline
1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\
2 & 1 & 0 & 1 & \boxed{1} & 1 & 0 & 0 & 0 \\
3 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 1 \\
4 & 0 & \boxed{1} & 0 & 0 & 1 & 0 & 0 & 0 \\
5 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 0 \\
6 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
7 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\
8 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0
\end{array}
$$

Representing Graphs (Review)

Option 1a: **Adjacency matrix.**

- n-by-n matrix where $A[u][v] = 1$ if $(u, v) \in E$
- Space $\underline{O(n^2)}$?
- Checking if $(u, v) \in E$ takes $\underline{O(1)}$ time?



$n = |V|, m = |E|$

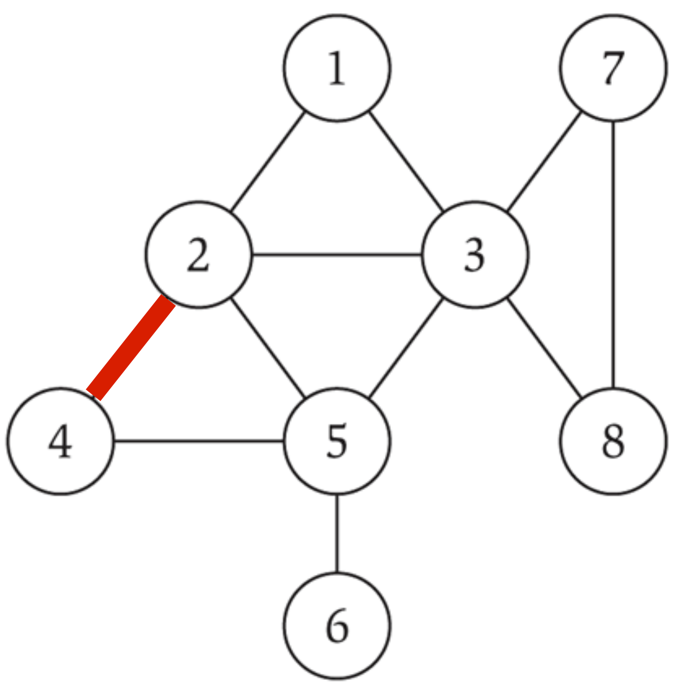
	1	2	3	4	5	6	7	8
1	0	1	1	0	0	0	0	0
2	1	0	1	1	1	0	0	0
3	1	1	0	0	1	0	1	1
4	0	1	0	0	1	0	0	0
5	0	1	1	1	0	1	0	0
6	0	0	0	0	1	0	0	0
7	0	0	1	0	0	0	0	1
8	0	0	1	0	0	0	1	0

Representing Graphs (Review)

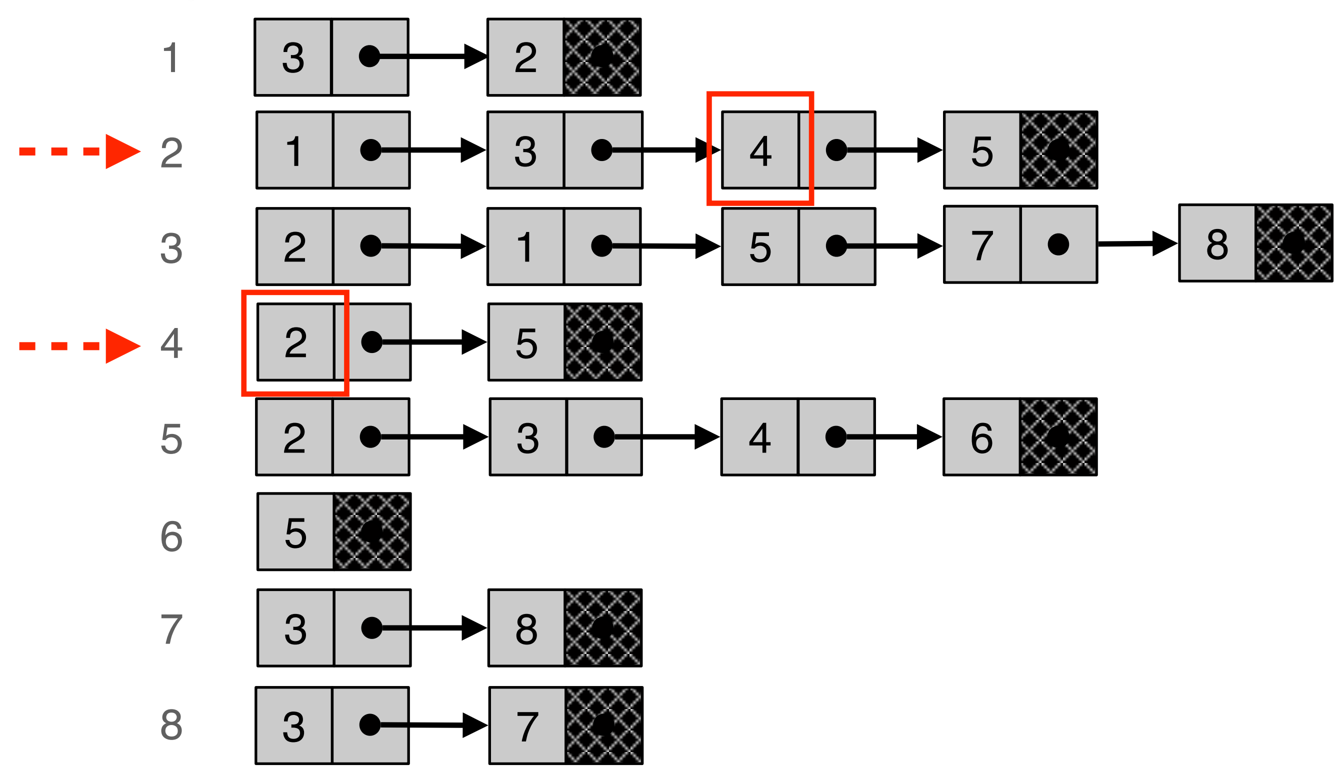
Option 1b: **Adjacency list.**

- Array of lists, where each list stores the neighbors of a given node

One list per vertex $v_i \in V$.
List i contains v_i's neighbors



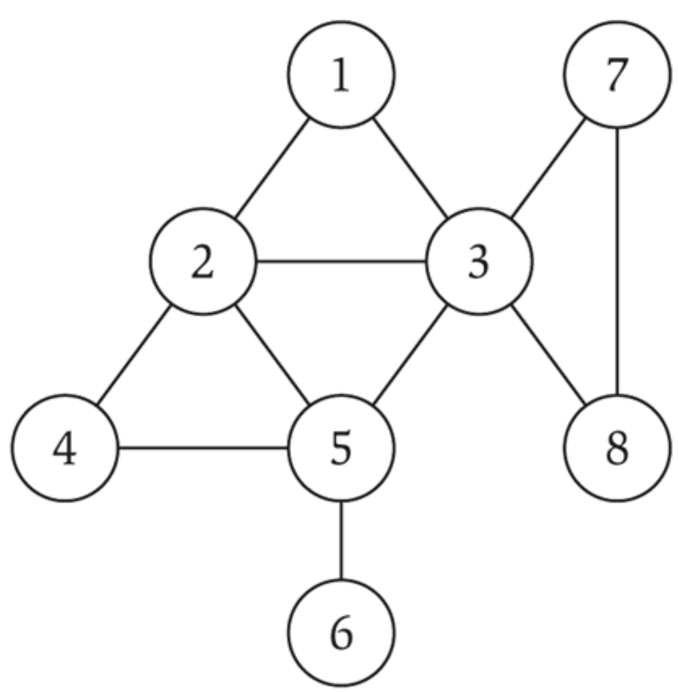
$n = |V|, m = |E|$



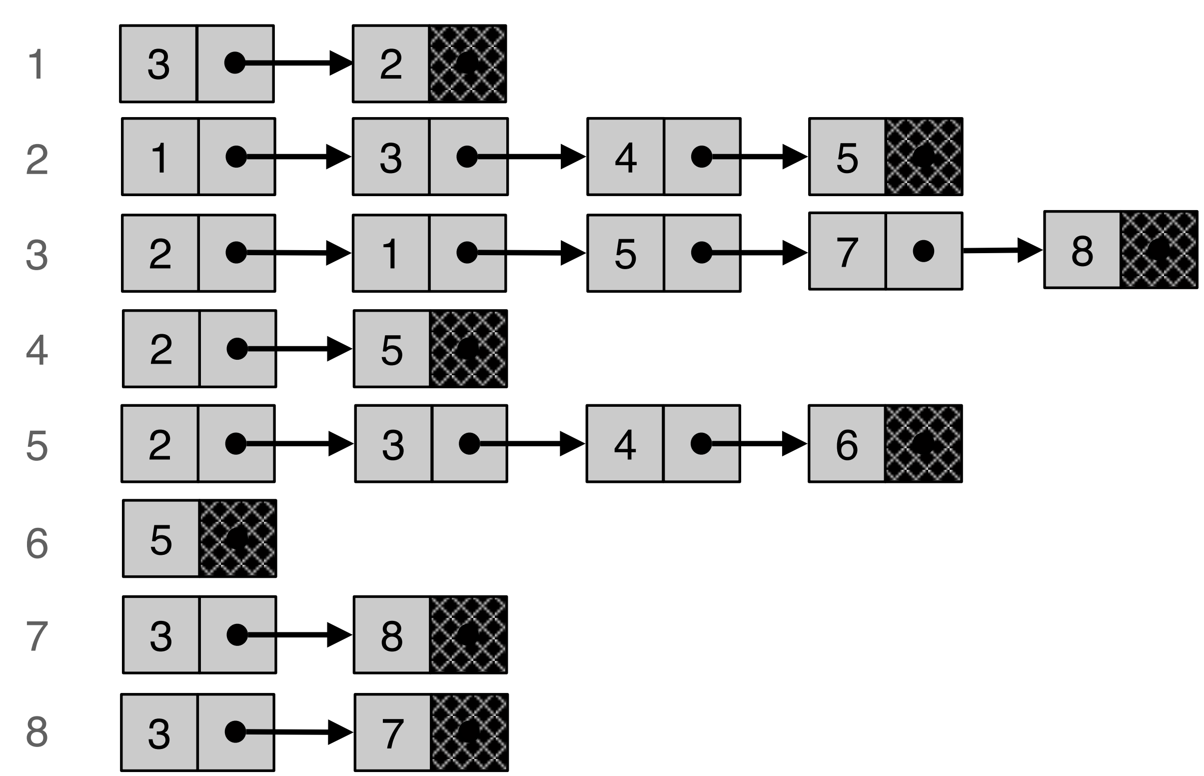
Representing Graphs (Review)

Option 1b: **Adjacency list.**

- Array of lists, where each list stores the neighbors of a given node
- Space $\underline{O(n+m)}$?
- Checking if $(u, v) \in E$ takes $\underline{O(\text{degree}(u))}$ time?



$n = |V|, m = |E|$



Graph Terminology (Review)

- A **path** in an undirected graph $G = (V, E)$ is a sequence of nodes $u_1, u_2, \ldots, u_k$ such that every pair $(u_{i-1}, u_i) \in E$.
- A path is **simple** if all nodes are distinct.
- The **length** of a path is **the number of edges on the** path

- An undirected graph is **connected** if for every pair of nodes u and v, there is a path between u and v (every node is **reachable** from all other nodes)
  - A connected component is the set of all vertices/edges reachable from some vertex v
  - A connected graph has 1 connected component.

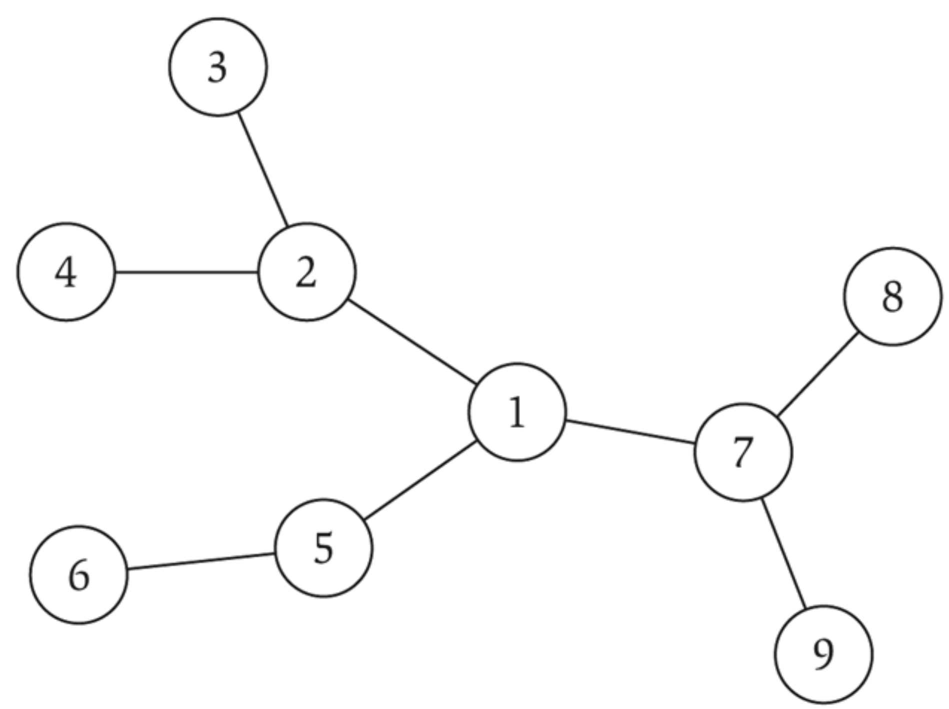
- A **cycle** is path $u_1, u_2, \ldots, u_k$ where $u_1 = u_k$ $(k \geq 2)$
- A cycle is **simple** if all internal nodes are distinct

Trees (Review)

An undirected graph is a **tree** if it is connected and acyclic (i.e, it does not contain a cycle)

Lemma. Let G be an undirected graph with n nodes. Then any two of these conditions imply the third

- G is connected
- G does not contain a cycle
- G has $n-1$ edges



Graph Traversals

A few common questions we ask about a graph $G = (V, E)$:

- **Connectivity.** How do we verify if a graph is connected?
- **Reachability.** Given $s, t \in V$, is there a path between them?

Answers can be determined by "traversing the graph"

- Two classic graph traversal algorithms:
    - Breadth-first search (BFS)

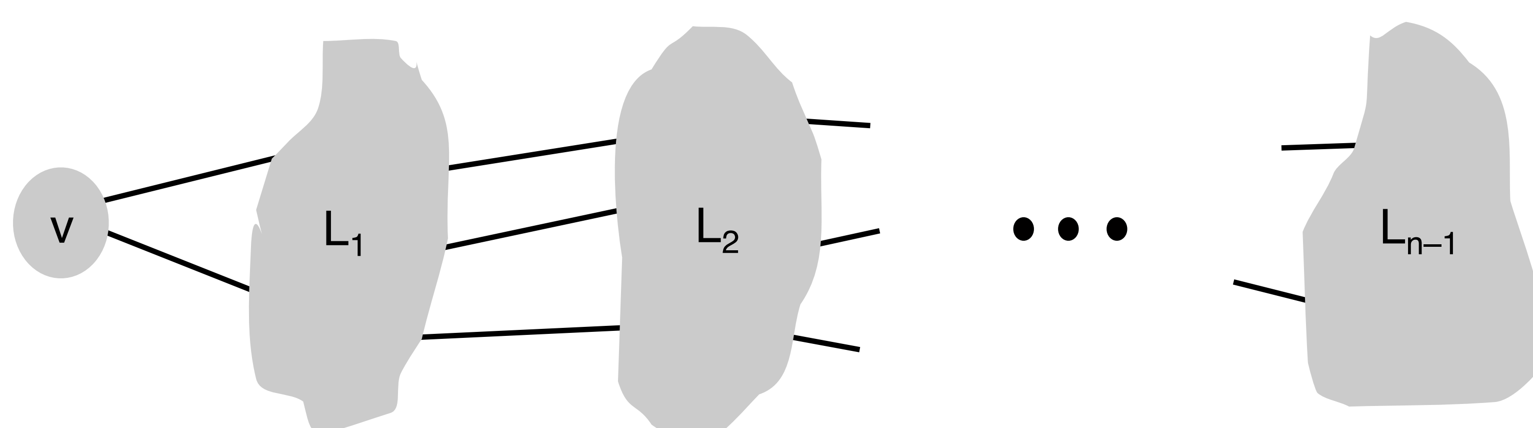
      Start at some node and radiate outward
    - Depth-first search (DFS)

      Start at some node and keep going until you hit a dead end
    - BFS & DFS are remarkably similar algorithms that merely differ in the data structure used

Breadth-first Search

Explore outwards in all possible directions from starting point, peeling "one layer after another"

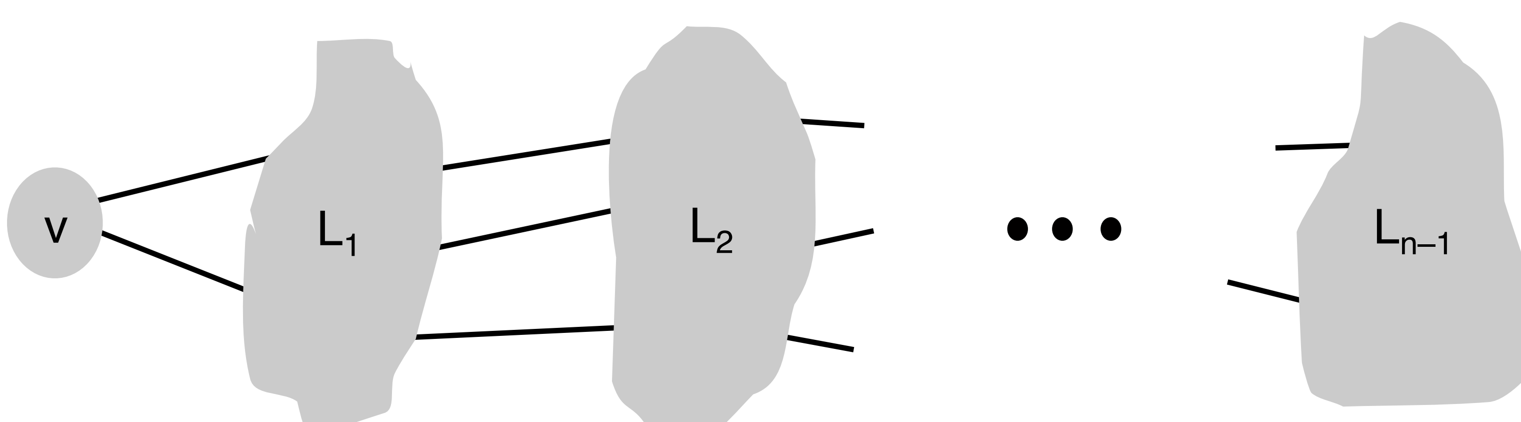
- BFS algorithm: Initialize $L_0 = \{v\}$
    - $L_1 =$ all neighbors of L_0
    - $L_2 =$ all nodes that do not belong to L_0 or L_1 that are adjacent to a node in L_1
    - $\ldots$
    - $L_{i+1} =$ all nodes that do not belong an earlier layer that are adjacent to a node in L_i



BFS Implementation

We need data structures to represent:

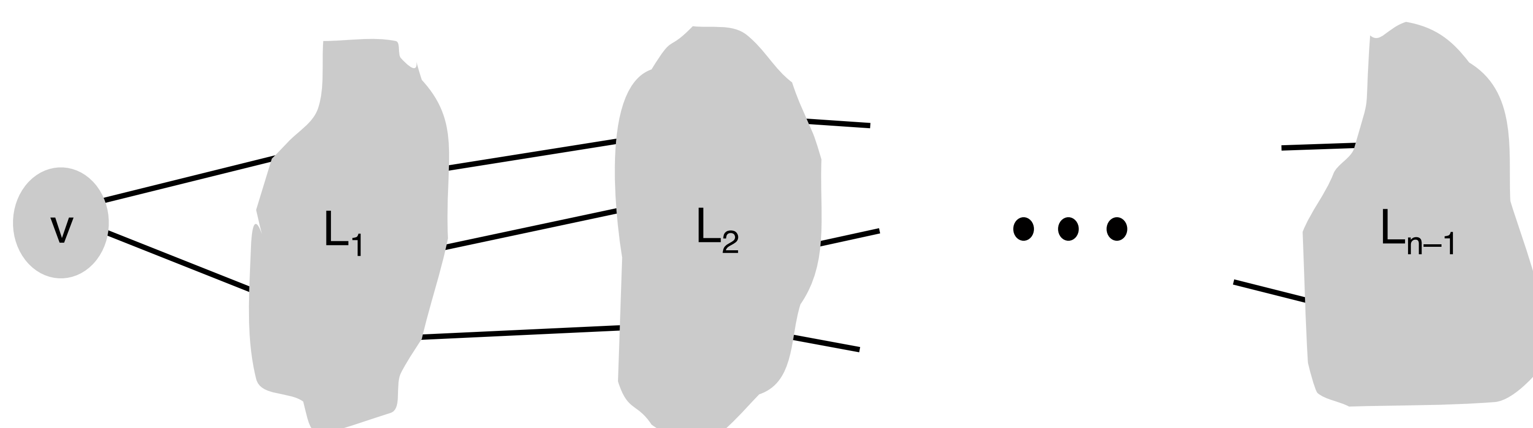
- Nodes that we have not encountered yet
- Nodes that we have encountered but not yet “explored”
- Nodes that have been “fully explored” (encountered all its neighbors as well)



BFS Implementation

Suppose we are currently exploring node u

- Its neighbors will be marked “encountered”, but when will they be explored compared to other encountered but unexplored nodes?
- **BFS Idea**: Explore all nodes at level i (same distance from initial node) before moving on to level $i + 1$
    - Rule: first encountered node should be first node to be explored
- Which data structure should we use?
    - Queue! First-in-first-out



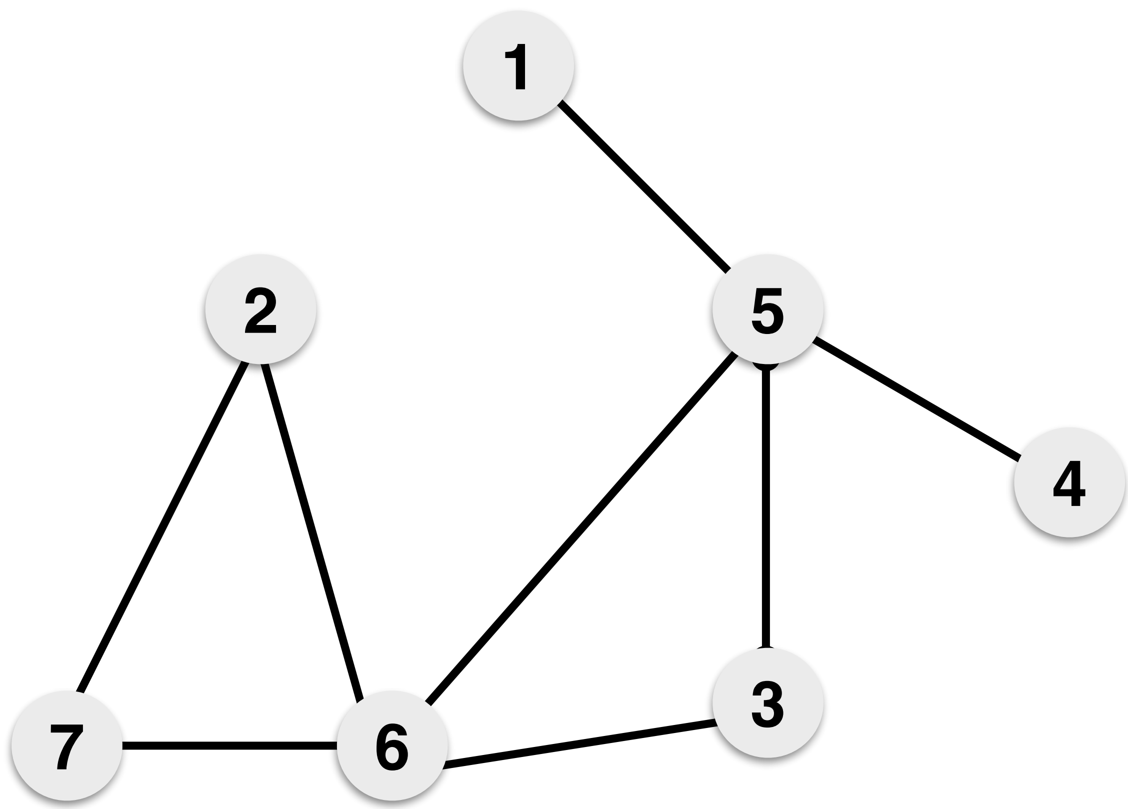
BFS Implementation: Queue

```
BFS (G, s):
 Set status of all nodes to unmarked
 Place s into the queue Q
 While Q is not empty
   Extract v from Q
     If v is unmarked
       Mark v
       For each edge (v, w):
          Put w into the queue Q
```

Observations:

- Nodes that we have not encountered have never been added to Q
- When a node u is marked (after extraction from Q), all u's neighbors are then enqueued, so the next time we see u we can ignore it —its already been explored!
- We may enqueue some nodes multiple times, but we only explore them once (if a marked node is extracted, it is skipped)

BFS Example



Tracing the Traversal: BFS Tree

- We can remember parent nodes (the node at level i that lead us to a given node at level $i+1$)
- Keeping track of these relationships produces a tree rooted at s

```
BFS-Tree(G, s):
 Put (∅, s) in the queue Q
 While Q is not empty
   Extract (p, v) from Q
     If v is unmarked
       Mark v
       parent(v) = p
       For each edge (v, w):
          Put (v, w) into the queue Q       (*)
```

BFS Analysis

- Inserting and extracting an edge from a queue: $\underline{O(1)}$ time
- For each marked node v, we run the for loop for its edges: $\underline{O(n)}$ times
- Overall running time? $\underline{O(n^2)}$
  - Can we do better?
- Yes! We can improve our analysis to $O(n+m)$
  - Node u has degree(u) incident edges (u, v)
  - Total time processing edges: $\sum_{u \in V} \text{degree(u)} = 2m$

each edge (u, v) is counted exactly twice
in sum: once in degree(u) and once in degree(v)

Depth-First Search

Stack Instead of Queue

If we change how we store the visited vertices (the data structure we use), it changes how we traverse the graph

```
BFS (G, s):
 Set status of all nodes to unmarked
 Place s into the queue Q
 While Q is not empty
   Extract v from Q
       For each edge (v, w):
          If w is unmarked
              Mark w
              Put w into the queue Q
```

Stack Instead of Queue

If we change how we store the visited vertices (the data structure we use), it changes how we traverse the graph

```
DFS (G, s):
 Set status of all nodes to unmarked
 Place s into the stack S
 While S is not empty
   Extract v from S
       For each edge (v, w):
          If w is unmarked
              Mark w
              Put w into the stack S
```

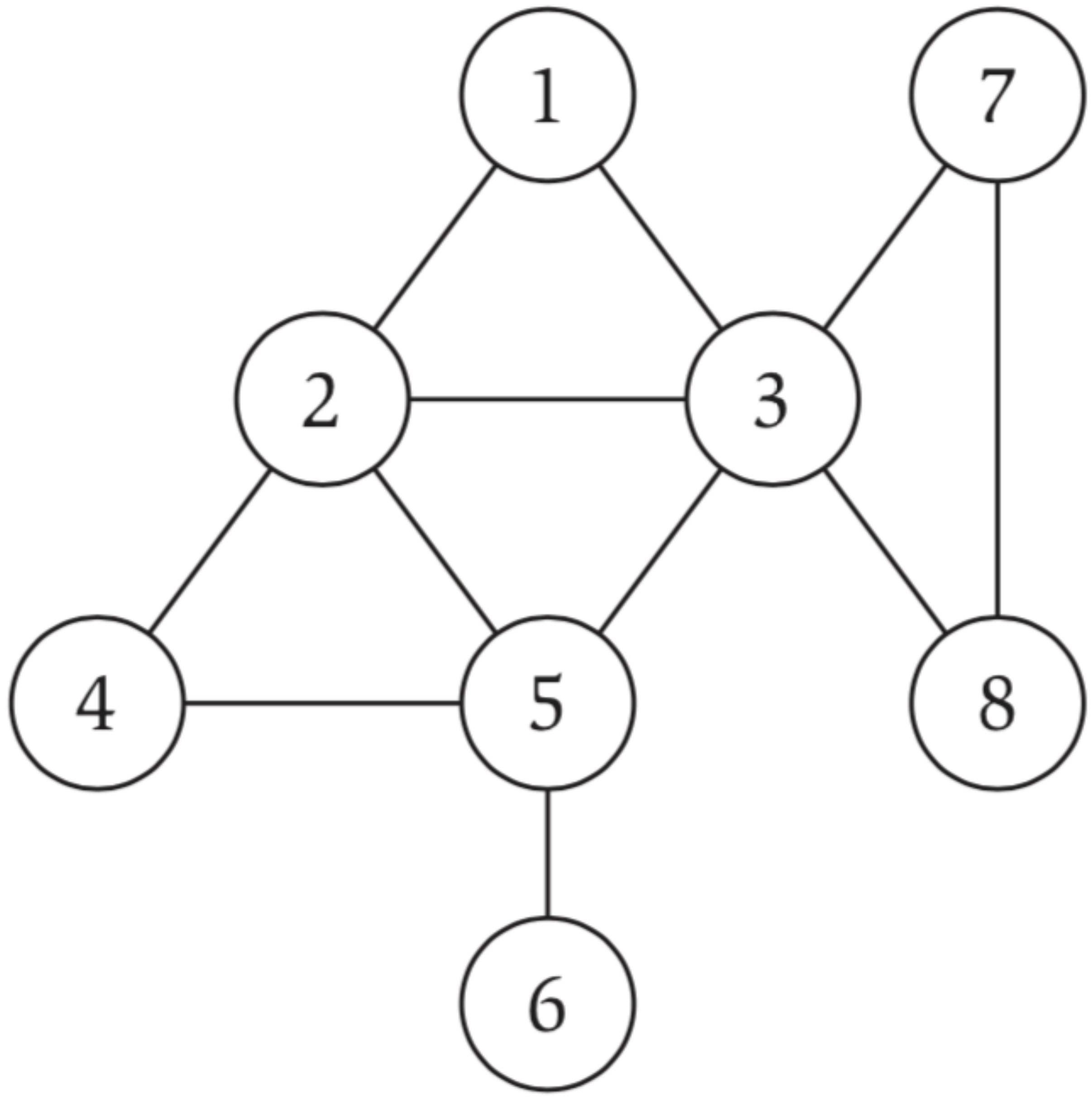
Depth-First Search: Recursive

DFS is perhaps the more natural traversal algorithm to write.

- Can be written **iteratively** or **recursively**
- Both DFS versions are the same; can actually see the “recursion stack” in the iterative version

```
Recursive-DFS(u):
   Set status of u to marked # encountered u
   for each edge (u, v):
      if v's status is unmarked:
         DFS(v)
   # done exploring neighbors of u
```

Example Graph



DFS Running Time

We can apply the same analysis as we did for BFS.

- Inserts and extracts to a stack: $O(1)$ time
- Setting status of each node to unmarked: $O(n)$
- Each node is set marked at most once; equivalently DFS(u) is called at most once for each node
- For every node v, explore degree(v) edges
  - $\sum_{v}$ degree(v) $= 2m$
- Overall, running time $O(n + m)$

Depth-First Search Tree

DFS returns a spanning tree, similar to BFS

```
DFS-Tree(G, s):
 Put (∅, s) in the stack S
 While S is not empty
   Extract (p, v) from S
     If v is unmarked
       Mark v
       parent(v) = p
       For each edge (v, w):
          Put (v, w) into the stack S
```

The spanning tree formed by parent edges in a DFS are usually long and skinny

Proving Correctness

DFS Correctness

- DFS finds precisely the set of nodes reachable from start node s
- That is, DFS(s) marks node x iff node x is reachable from s
- **Proof**. $(\Rightarrow)$
    - Since x is marked, $(x,$ parent$(x))$) is an edge in the graph
    - *Claim*. $x \to$ parent(x) $\to$ parent(parent(x)) $\to \cdots$ leads to s
    - Induction on the order in which vertices are marked
    - Suppose claim holds for all vertices before some vertex u
    - Consider u: parent(u) must be discovered before u, and thus the claim holds for it, since ($u,$ parent(u)) is an edge, we have a path from u to s

DFS Correctness

- DFS finds precisely the set of nodes reachable from start node s
- That is, DFS(s) marks node x iff node x is reachable from s
- **Proof**. $(\Leftarrow)$
    - Suppose node x is reachable from s via path P, but x is not marked by DFS
    - Since s is marked by DFS and x is not, there must be a first node v on P that is not marked by DFS
    - Thus, there is an edge $(u, v) \in P$ such that u is marked and v is not marked
    - But this cannot happen, since when u is marked, all its neighbors are also marked $\Rightarrow \Leftarrow$ ∎

BFS Correctness

- Breadth first search finds precisely the set of nodes reachable from s
- That is, BFS(s) marks node x iff node x is reachable from s
- **Proof**. $(\Rightarrow)$
  - Since x is marked, $(x,$ parent$(x)))$ is an edge in the graph
  - *Claim*. $x \rightarrow$ parent(x) $\rightarrow$ parent(parent$(x)) \rightarrow \cdots$ leads to s
  - Induction on the order in which vertices are marked
  - Let $u_1, u_2, \ldots, u_k, \ldots, u_n$ denote the order in which vertices are marked, suppose claim holds all vertices with index less than k
  - Consider u_k: parent(u_k) must be discovered before u_k, and thus the claim holds for it, since $(u_k,$ parent$(u_k))$ is an edge, we have a path from u_k to s

BFS Correctness

- Breadth first search finds precisely the set of nodes reachable from s
- That is, BFS(s) marks node x iff node x is reachable from s
- **Proof**. $(\Leftarrow)$
    - Suppose node x is reachable from s via path P, but x is not marked by BFS
    - Since s is marked by BFS and x is not, there must be a first node $v \neq s$ on P that is not marked by BFS
    - Thus, there is an edge $(u, v) \in P$ such that u is marked and v is not marked
    - But this cannot happen, since when u is marked, all its neighbors are also marked $\Rightarrow \Leftarrow$ ∎