

# Announcements

- Homework 2 is released, due next Tuesday at 3:30pm
- The grading for Programming Project 1 will be delayed
- The office hour schedule for the rest of the semester has been posted on Piazza and the course website

# Topic 3: Graphs

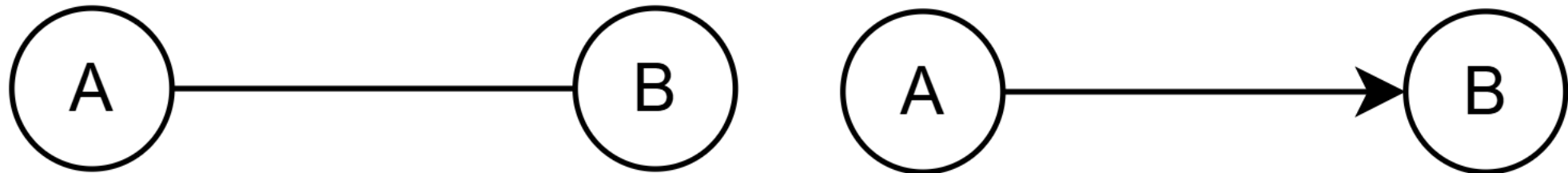
By Professor Hudson Lynam

# Graph Definition

- Definition: *Graph*
- *A graph  $G=(V,E)$  where  $V$  is a finite non-empty set of vertices (a.k.a. nodes) and  $E$  is a binary relation on  $V$ .*
- Note that definitions of “graph” vary: this is a common definition we’ll be using for this class

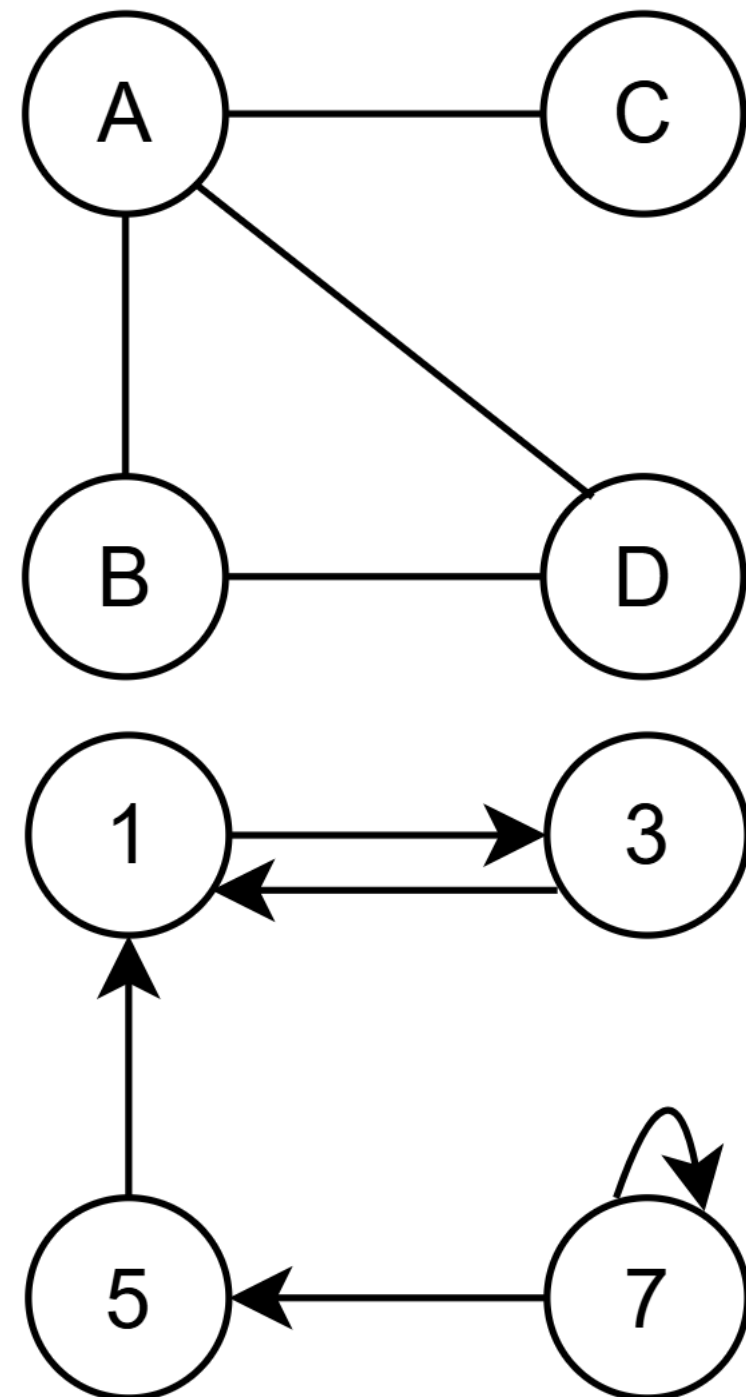
# Undirected versus Directed Graphs

- Undirected:
  - The pairs of vertices in  $E$  are **unordered**. In other words, each edge is a set, like  $\{A,B\}$ .
  - When we say “ $G$  is a graph,” the default assumption is that  $G$  is undirected
- Directed
  - Each edge has an order, like a tuple  $(A,B)$ .
  - Because directed graphs are common, we usually abbreviate to “digraph.”



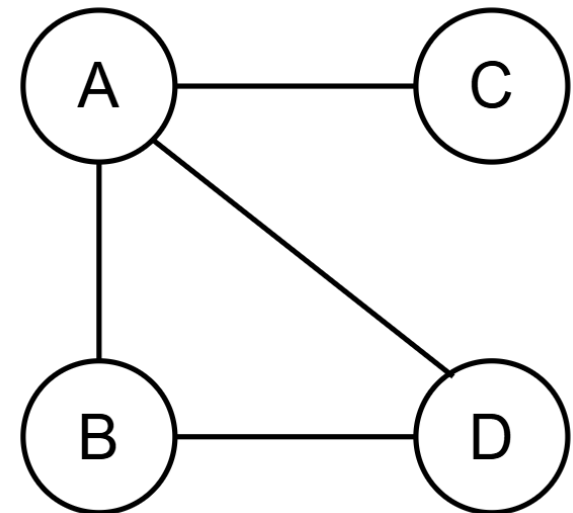
# Graphs Example

- What would  $V$  be for this graph?
- $V = \{A, B, C, D\}$
- $E = \{\{A, C\}, \{A, B\}, \{A, D\}, \{B, D\}\}$
- What about  $V$  and  $E$  for this digraph?
- $V = \{1, 3, 5, 7\}$
- $E = \{(1, 3), (3, 1), (5, 1), (7, 7), (7, 5)\}$
- Note: Don't use bidirectional arrows for undirected graphs, and always use arrows to show two edges between vertices in a digraph



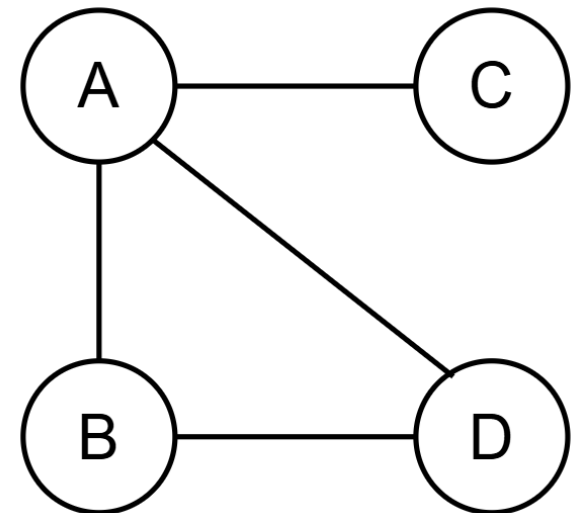
# Graph Terminology

- Definition: *Adjacent Vertices*
- *2 vertices are adjacent when an edge exists between them*
- In our example graph, C and D are not adjacent, B and C are not adjacent, every other pair of vertices are adjacent
- Definition: *Path (a.k.a. Walk)*
- *A path is a sequence of adjacent vertices*
- Example: C-A-B-A-D-B.
- A **simple** path's vertices appear exactly once. The above example is *not* simple. What would be a simple path here?



# Graph Terminology

- Definition: *Cycle*
- *A cycle is a path of length  $\geq 1$  that begins and ends with the same vertex*
- In our example graph, A-C-A is a cycle. D-B-A-C-A-D is a cycle.
- A cycle is **simple** if all the vertices in the path only occur *once* (except for the first and last vertices)
- Notes: “the cycle ABD” means A-B-D-A
- Common assumption: simple cycles in undirected graphs must have length  $\geq 3$

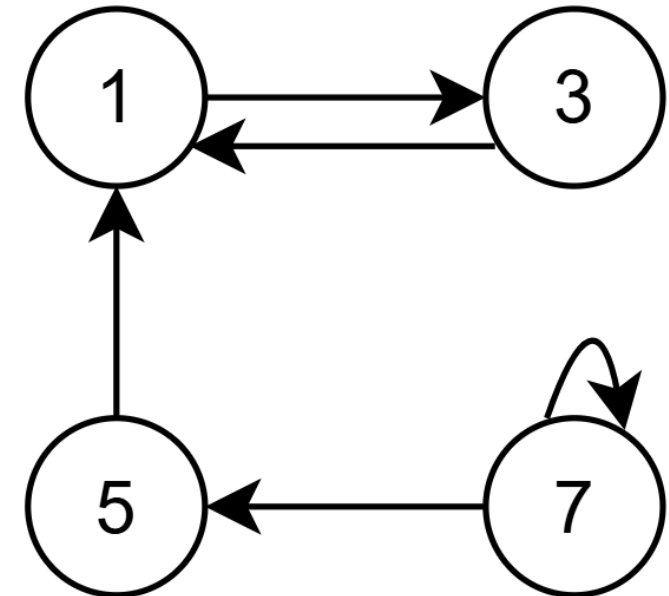


# Graph Terminology

- Definition: *Incident Edges*
- *In any graph, an edge is incident on both participating vertices*
- *In digraphs, an edge is **incident from** the source and **incident to** the destination vertex*

- Definition: *Degree*
- *The degree of a vertex is its # of incident edges*

- Note: a self-loop adds to a vertex's degree
- What is the degree of 1? In versus out?
- In-degree of 1: 2, out-degree of 1: 1

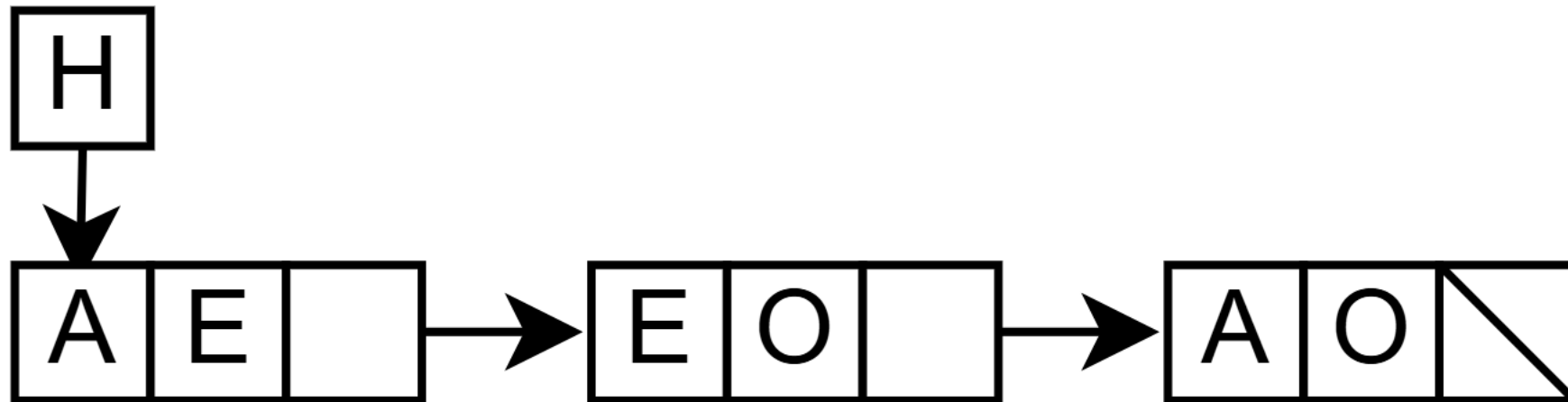
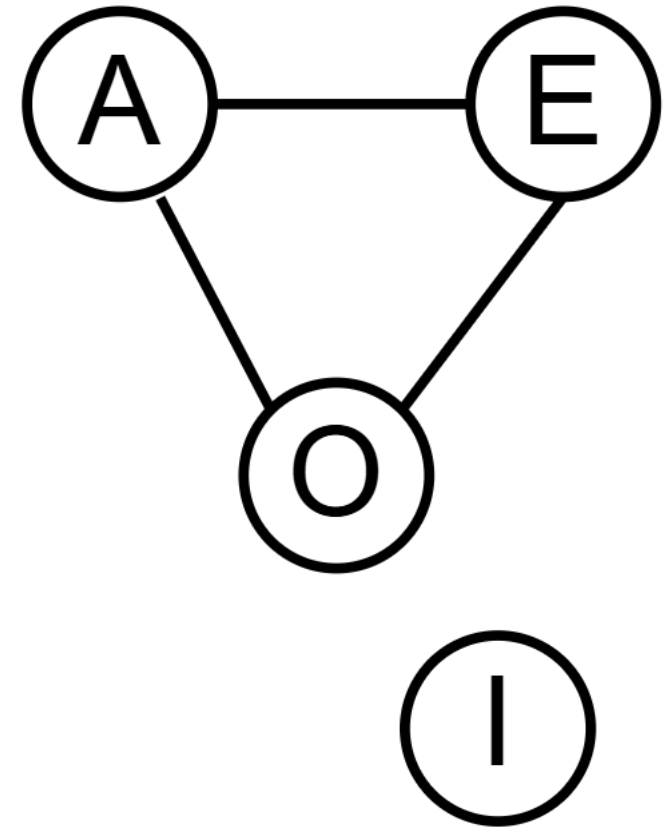


# Graph Representations

- How can we store all of a graph's vertices and edges? Three common representations:
  1. Edge Lists
  2. Adjacency Matrices
  3. Adjacency Lists
- Which should we choose? Depends on the operations we want to be efficient.

# Edge Lists

- An edge list is a collection of edges within a graph. Example:
- I is an isolate in this graph. Obviously, not an ideal representation for graphs with isolates
- Note: In a digraph, just allow 1<sup>st</sup> vertex in each edge to the source or “from” vertex

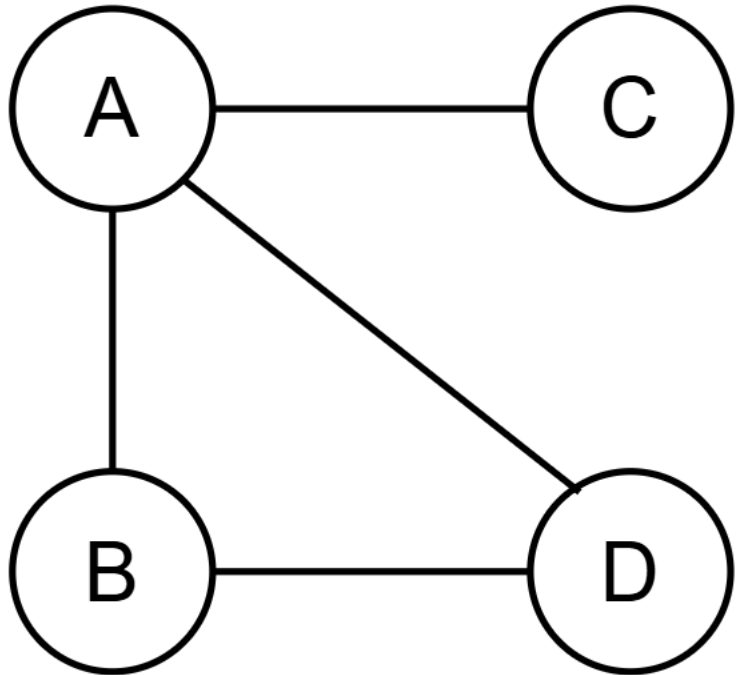


# Adjacency Matrices

- Adjacency Matrices are a good choice when the quantity of vertices is bounded and the graph has a relatively large quantity of edges.
- The **adjacency matrix** of a graph  $G=(V, E)$  is a  $|V| \times |V|$  matrix  $M$  in which element  $M[i][j] = 1$  iff  $\{v_i, v_j\}$  or  $(v_i, v_j) \in E$ . Otherwise,  $M[i][j] = 0$ .

# Adjacency Matrices

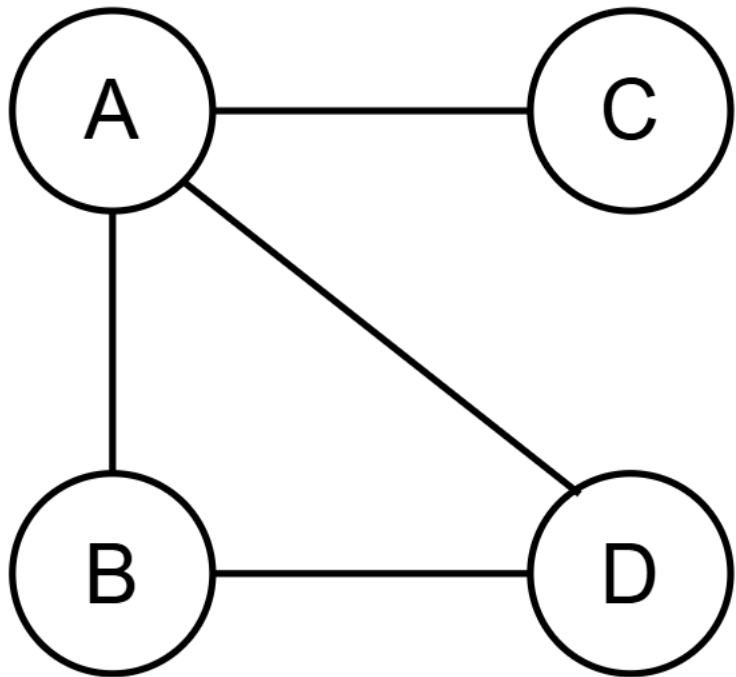
- Let's take a look at our example graph:



	A	B	C	D
A	0	1	1	1
B	1	0	0	1
C	1	0	0	0
D	1	1	0	0

# Adjacency Matrices

- Did we even need the full matrix? Could we represent the edges more efficiently? How?

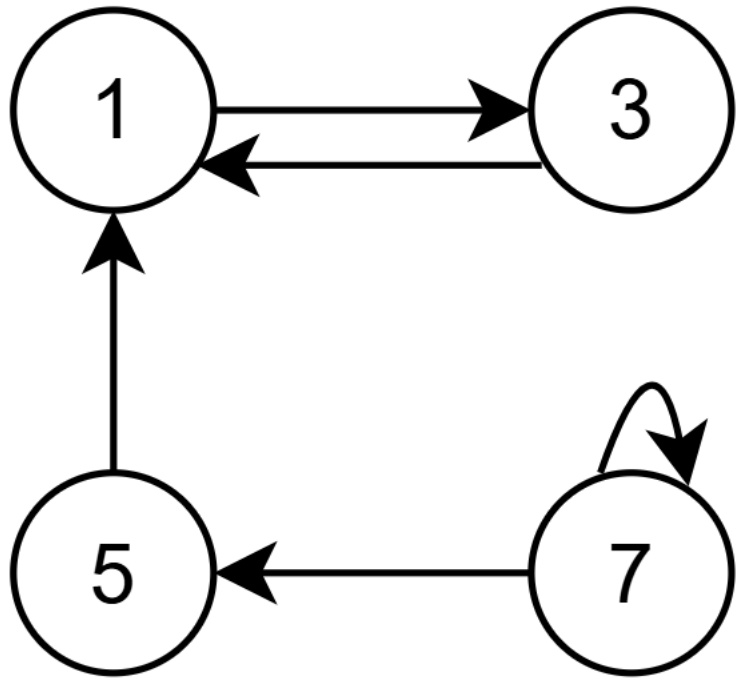


	A	B	C	D
A	0	1	1	1
B	1	0	0	1
C	1	0	0	0
D	1	1	0	0

- Note: this is a lower left triangular matrix

# Adjacency Matrices

- Try to make an adjacency matrix for this digraph. Cols are To, Rows are From



	1	3	5	7
1	0	1	0	0
3	1	0	0	0
5	1	0	0	0
7	0	0	1	1

# Adjacency Matrices

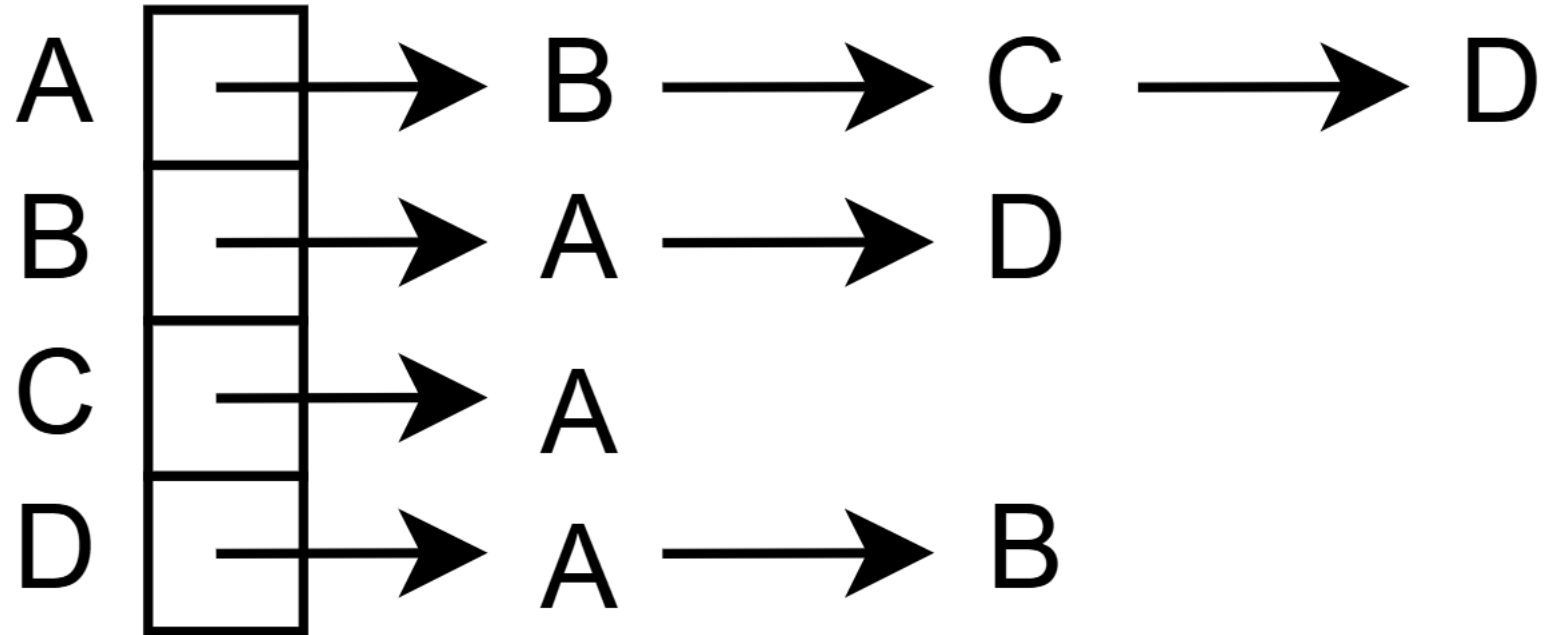
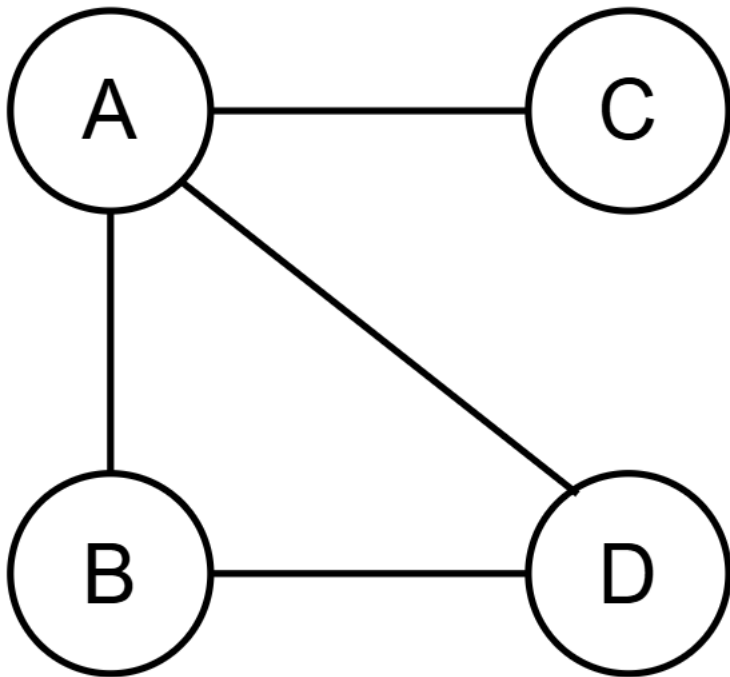
- Adjacency matrices make many operations easy. For example...
- How would we find the degree of a vertex in an AM?
- Count the row or col for the vertex if it's a graph, count the row for out-degree in a digraph, or col for the in-degree
- Which edges are incident to a given vertex?
- Same solution as finding degree!
- But there's a downside:
- We need  $O(|V|^2)$  space to store an AM. Which is why AM is a good choice when the number of vertices is low

# Adjacency Lists

- The **adjacency list** of a graph  $G=(V, E)$  is an array of  $|V|$  lists of vertices, with each list holding the vertices to which the vertex is adjacent (for a graph) or the vertices incident from a vertex (for a digraph).

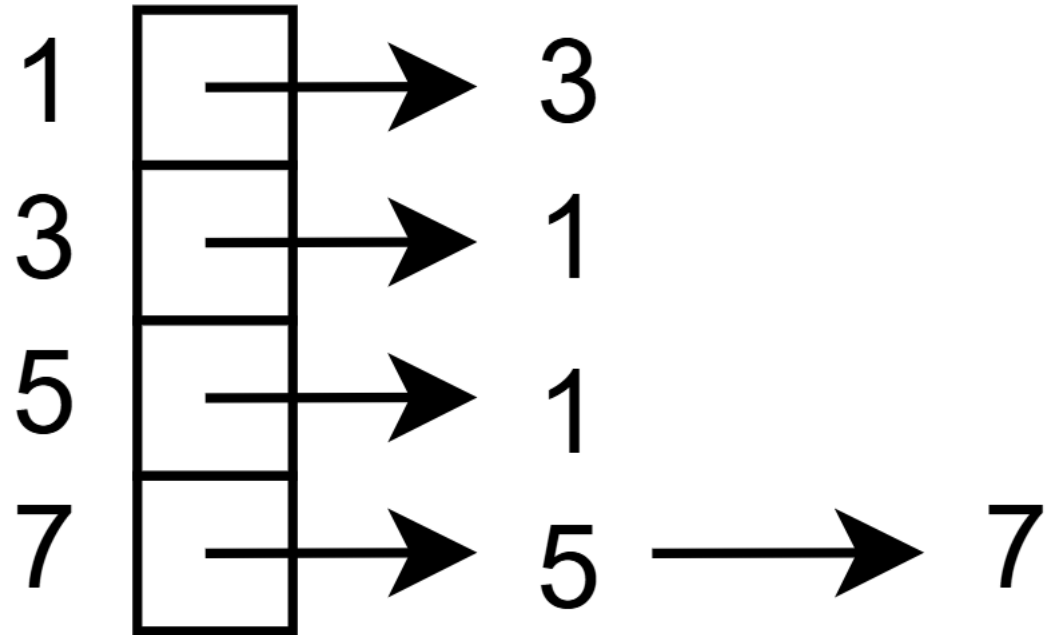
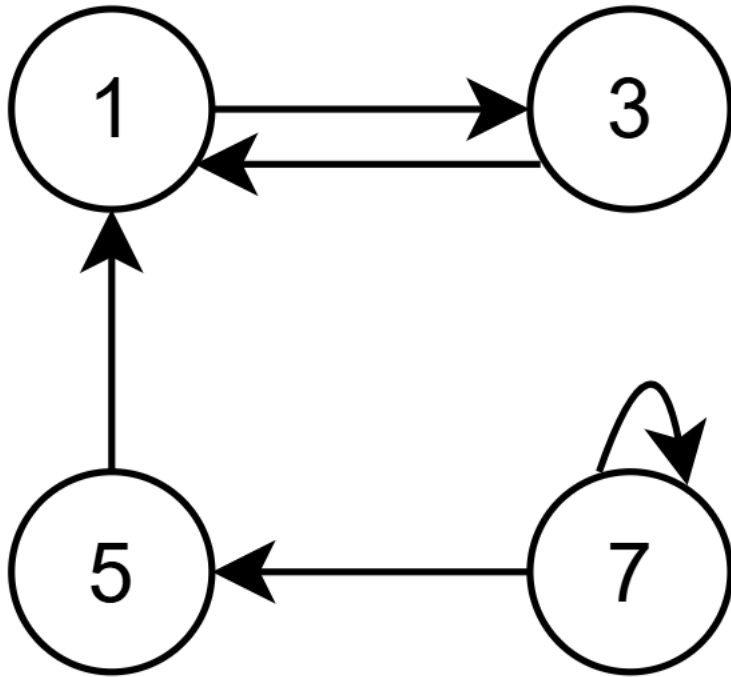
# Adjacency Lists

- Let's take a look at our example graph:



# Adjacency Lists

- Now go ahead and try to construct the AL for our digraph example.



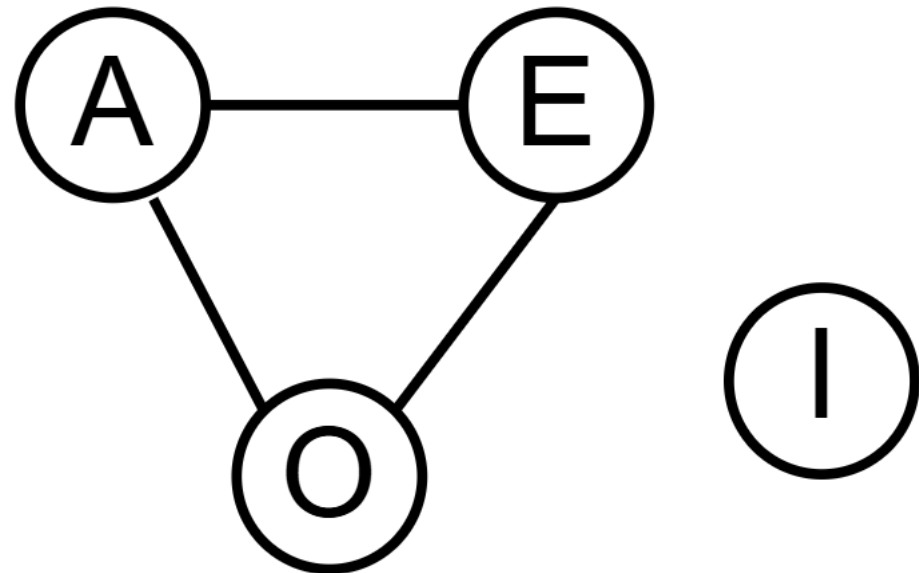
# Adjacency Lists

- ALs are efficient for some operations, but not others
- How about finding the degree for a given vertex?
- Easy for graphs, but digraphs require searching the entire AL!
- What's the storage requirement for AL?
- $O(|V|+|E|)$  ; this is an example of having 2 instance characteristics!
- In summary, when choosing a graph representation, consider the graph (# of vertices vs edges) and the operations you expect to do on the graph (and how expensive they're likely to be)

# Connectedness of Graphs

- Definition: *Connected Graph*
- *A graph  $G$  is connected when every pair of vertices in  $G$  is connected.*

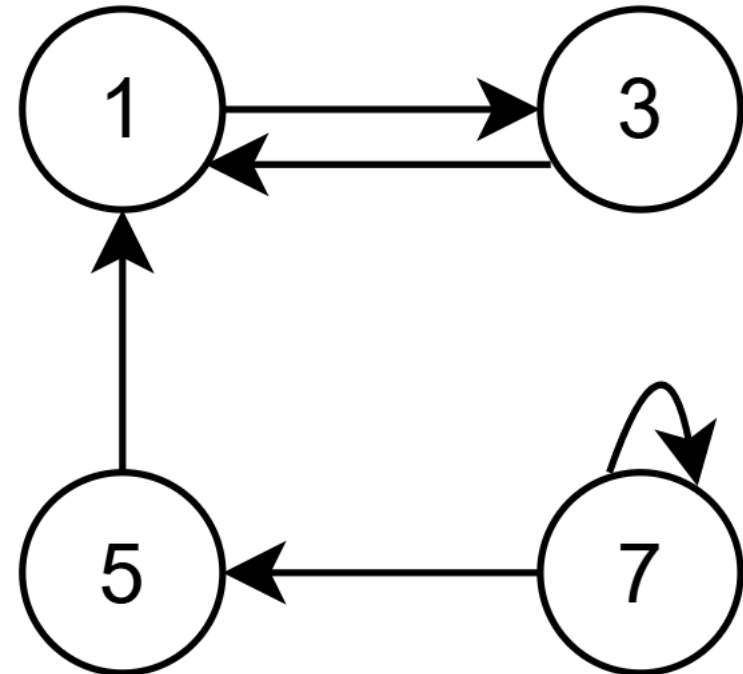
- Is this graph connected?
- How about this graph?
- You can't get to I from any other vertex, so no, not a connected graph



# Connectedness of Digraphs

- Definition: *Weakly Connected Digraph*
- A digraph  $G$  is a weakly connected digraph if it is connected when edge direction is ignored.

- Definition: *Strongly Connected Digraph*
- A digraph  $G$  is a strongly connected digraph if, for every pair of vertices there is a path between them in both directions.



# Determining that a Graph is Connected

- What are some of the tree traversals you've learned?
- Preorder, Postorder, Inorder, and Level Order
- Would any of these work for graphs?
- No; because graphs can have cycles
- We need a different way to traverse (or search through) graphs!

# Breadth-First Search (BFS)



# Breadth-First Search (BFS)

- Like the spread of liquid after popping a water balloon, BFS starts at some “center” and fans in all directions
- BFS relies on a **queue**

## **BFS:**

Enqueue the source vertex

While the queue is not empty:

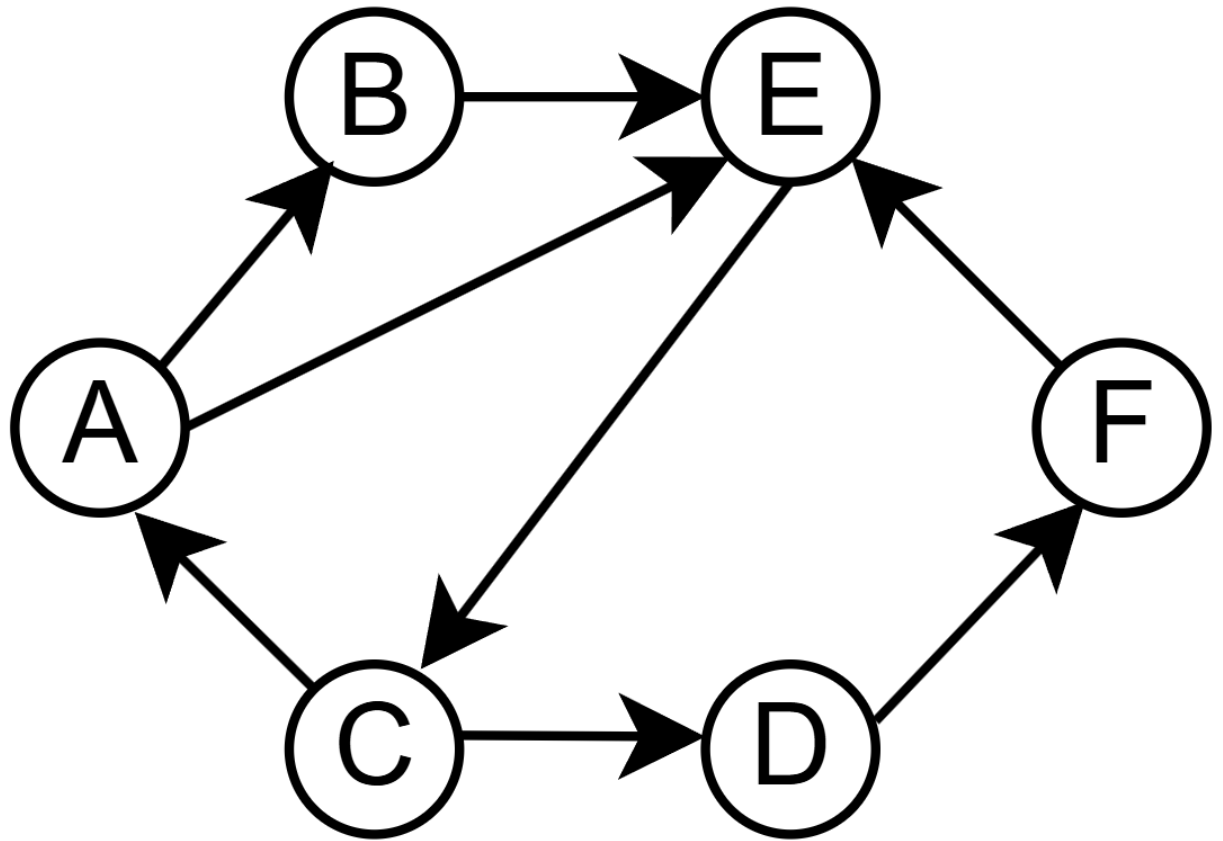
    Dequeue the first vertex

    Act on the vertex

    Enqueue the adjacent but previously undiscovered vertices

# Breadth-First Search (BFS)

- Let's do a BFS on this graph, with A as the source vertex
- Queue: A, prev seen: A. Dequeue A.
- Queue: B, E prev seen: A, B, E. Dequeue B.
- Queue: E, prev seen: A, B, E. Dequeue E.
- Queue: C, prev seen A, B, E, C. Dequeue C.
- Queue D, prev seen all but F. Dequeue D.
- Queue: F, prev seen all. Dequeue F, queue is empty, finished.



# Breadth-First Search (BFS)

- Some notes on BFS:
- If vertices and edges are not reachable from the source vertex, BFS cannot find them.
  - To deal with this, we can re-apply BFS to the remaining portions of the graph to find additional connected components of the graph
- How efficient is BFS? Depends on the graph representation.  
Discuss: Adjacency Matrix versus Adjacency List?
- AM:  $O(|V|^2)$
- AL:  $O(|V| + |E|)$

# Depth-First Search (DFS)

- The idea is: when we see a new vertex, we visit that vertex!
- DFS relies on a **stack**

## **DFS:**

Push the source vertex

While the stack is not empty:

    Pop the first vertex

    Act on the popped vertex

    Push its adjacent but previously undiscovered vertices