Data Structure

Lecture#23: Graphs (Chapter 11)

U Kang
Seoul National University

In This Lecture

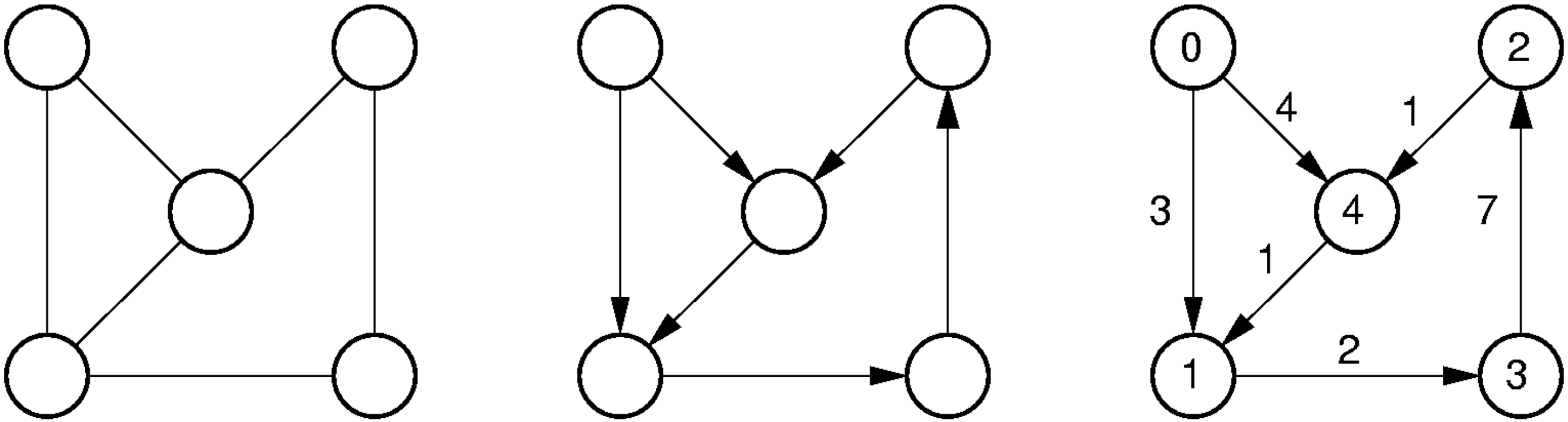
- Basic terms and definitions of graphs
- How to represent graphs
- Graph traversal methods

Graphs

- A graph **G** = (**V**, **E**) consists of a set of vertices (or nodes) **V**, and a set of edges **E**, such that each edge in **E** is a connection between a pair of vertices in **V**.
  - Example: Social network, phone call graph, computer network, …

- The number of vertices is written |**V**|, and the number edges is written |**E**|.

Graphs (2)



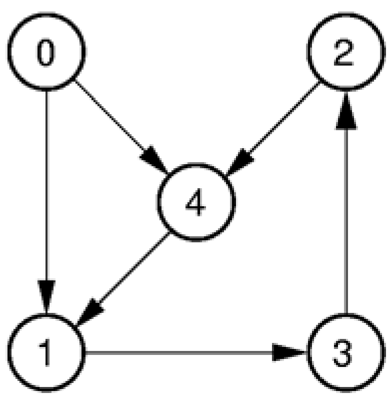
(a) Undirected Graph

(b) Directed Graph

(c) Weighted Graph

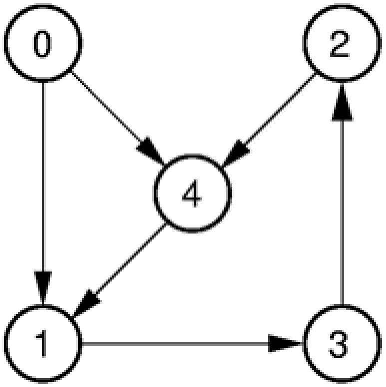
Paths and Cycles

- Path: A sequence of vertices $v_1, v_2, \ldots, v_n$ of length n-1 with an edge from v_i to v_{i+1} for $1 \leq i < n$.
  - E.g., 0, 4, 1, 3, 2, 4 in the graph below is a path
- A path is simple if all vertices on the path are distinct.
  - E.g., 0, 4, 1, 3, 2 in the graph below is a simple path



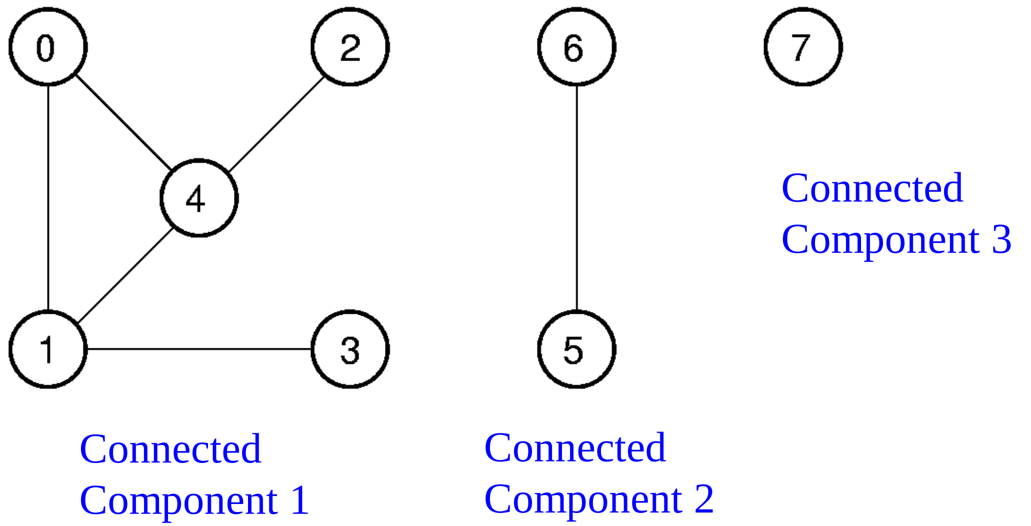
Paths and Cycles

- A <u>cycle</u> is a path of length 3 or more that connects v_i to itself.
  - E.g., 1, 3, 2, 4, 1, 3, 2, 4, 1 in the graph below is a cycle
- A cycle is <u>simple</u> if all vertices on the path are distinct, except the first and the last vertices
  - E.g., 1, 3, 2, 4, 1 in the graph below is a simple cycle

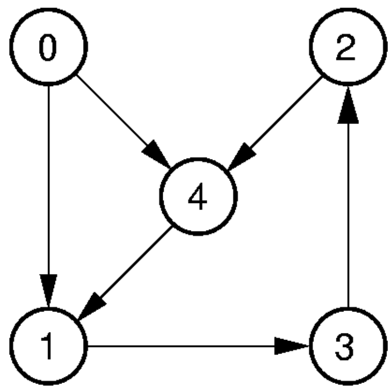


Connected Components

- An undirected graph is connected if there is at least one path from any vertex to any other.
- The maximum connected subgraphs of an undirected graph are called connected components.



Directed Representation

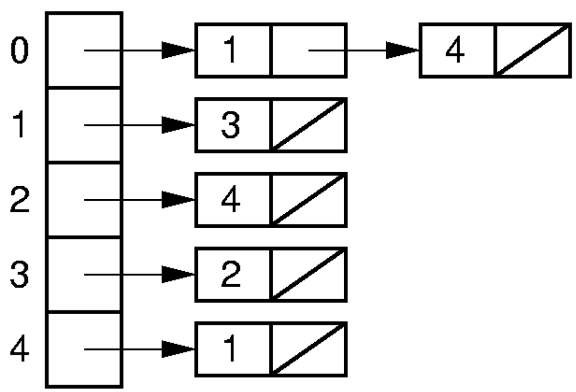


(a)

	0	1	2	3	4
0		1			1
1				1	
2					1
3			1		
4		1			

(b)

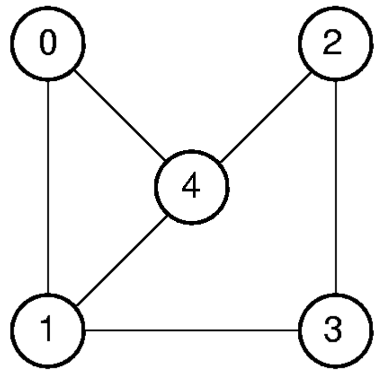
Adjacency Matrix



(c)

Adjacency List

Undirected Representation

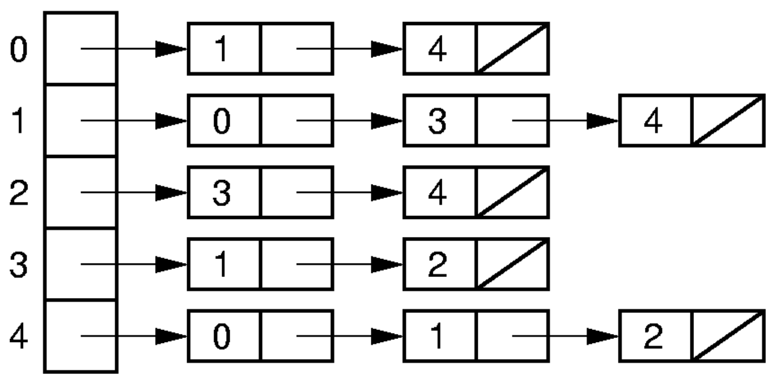


(a)

	0	1	2	3	4
0		1			1
1	1			1	1
2				1	1
3		1	1		
4	1	1	1		

(b)

Adjacency Matrix



(c)

Adjacency List

Representation Costs

- Adjacency Matrix:
  - $\Theta(|\mathbf{V}|^2)$ space
- Adjacency List:
  - $\Theta(|\mathbf{V}| + |\mathbf{E}|)$ space.
  - What is the maximum size of $|\mathbf{E}|$?
    - Answer: $|\mathbf{V}|^2$
  - When is Adjacency List more space-efficient than Adjacency Matrix?
    - For sparse graphs ($|\mathbf{E}| << |\mathbf{V}|^2$)
  - When is Adjacency Matrix more space-efficient than Adjacency List?
    - For dense graphs ($|\mathbf{E}| \sim |\mathbf{V}|^2$)

Graph ADT

```
interface Graph {              // Graph class ADT
  public void Init(int n); // Initialize
  public int n();              // # of vertices
  public int e();              // # of edges
  public int first(int v); // First neighbor
  public int next(int v, int w); // Neighbor
  public void setEdge(int i, int j, int wght);
  public void delEdge(int i, int j);
  public boolean isEdge(int i, int j);
  public int weight(int i, int j);
  public void setMark(int v, int val);
  public int getMark(int v);  // Get v's Mark
}
```

Graph Traversals

- Some applications require visiting every vertex in the graph exactly once.
- The application may require that vertices be visited in some special order based on graph topology.
- Examples: artificial intelligence search, shortest paths problems
- Important Traversals
  - Depth First Search (DFS)
  - Breadth First Search (BFS)
  - Topological Sort

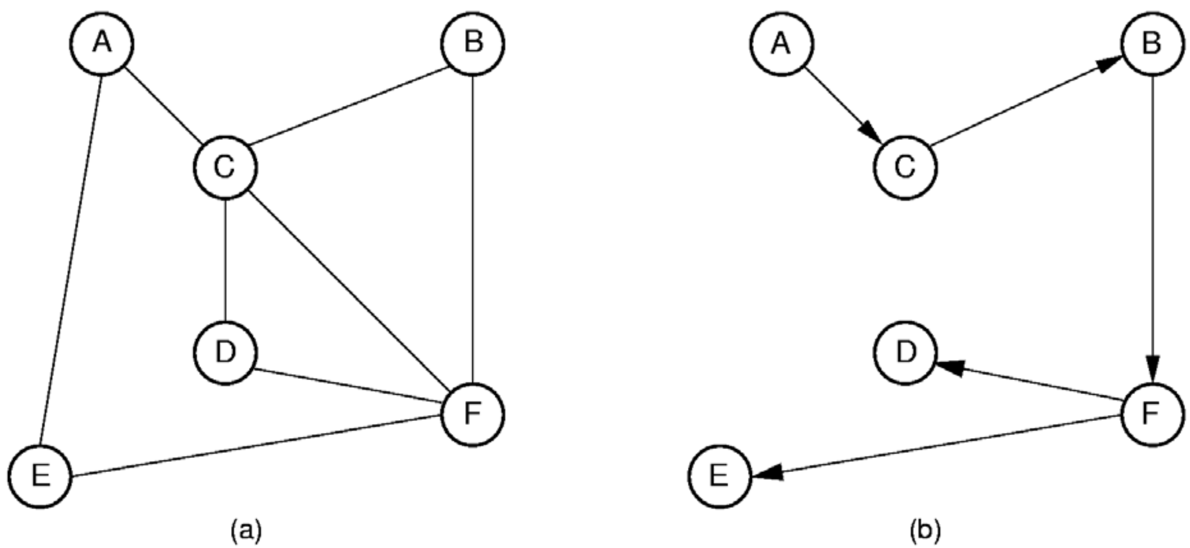
Graph Traversals (2)

- To insure visiting all vertices:

```
void graphTraverse(Graph G) {
  int v;
  for (v=0; v<G.n(); v++)
    G.setMark(v, UNVISITED); // Initialize
  for (v=0; v<G.n(); v++)
    if (G.getMark(v) == UNVISITED)
      doTraverse(G, v);
}
```

Depth First Search (1)

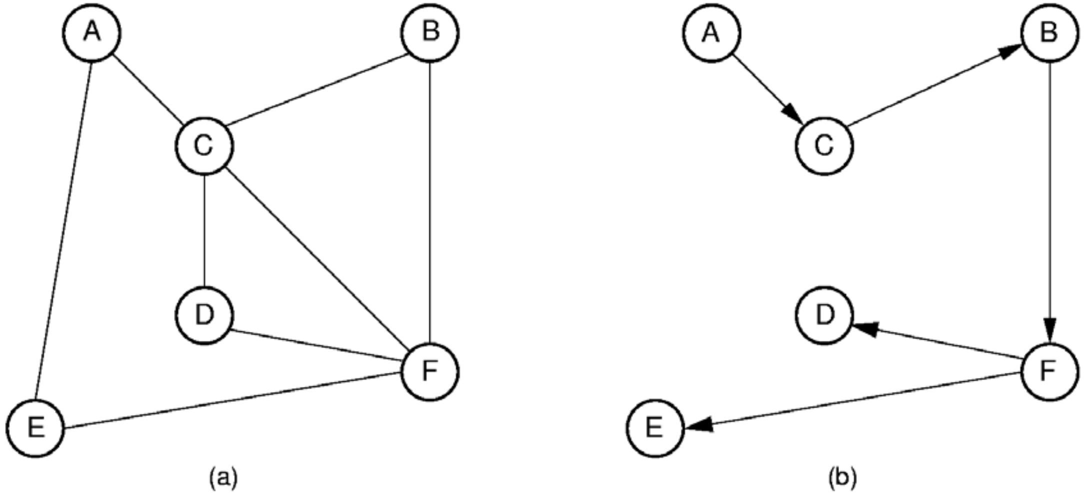
- Main Idea
  - Start from a vertex s
  - Visit an unvisited neighbor v of s
  - Visit an unvisited neighbor v’ of v
  - … continue until all vertices are visited



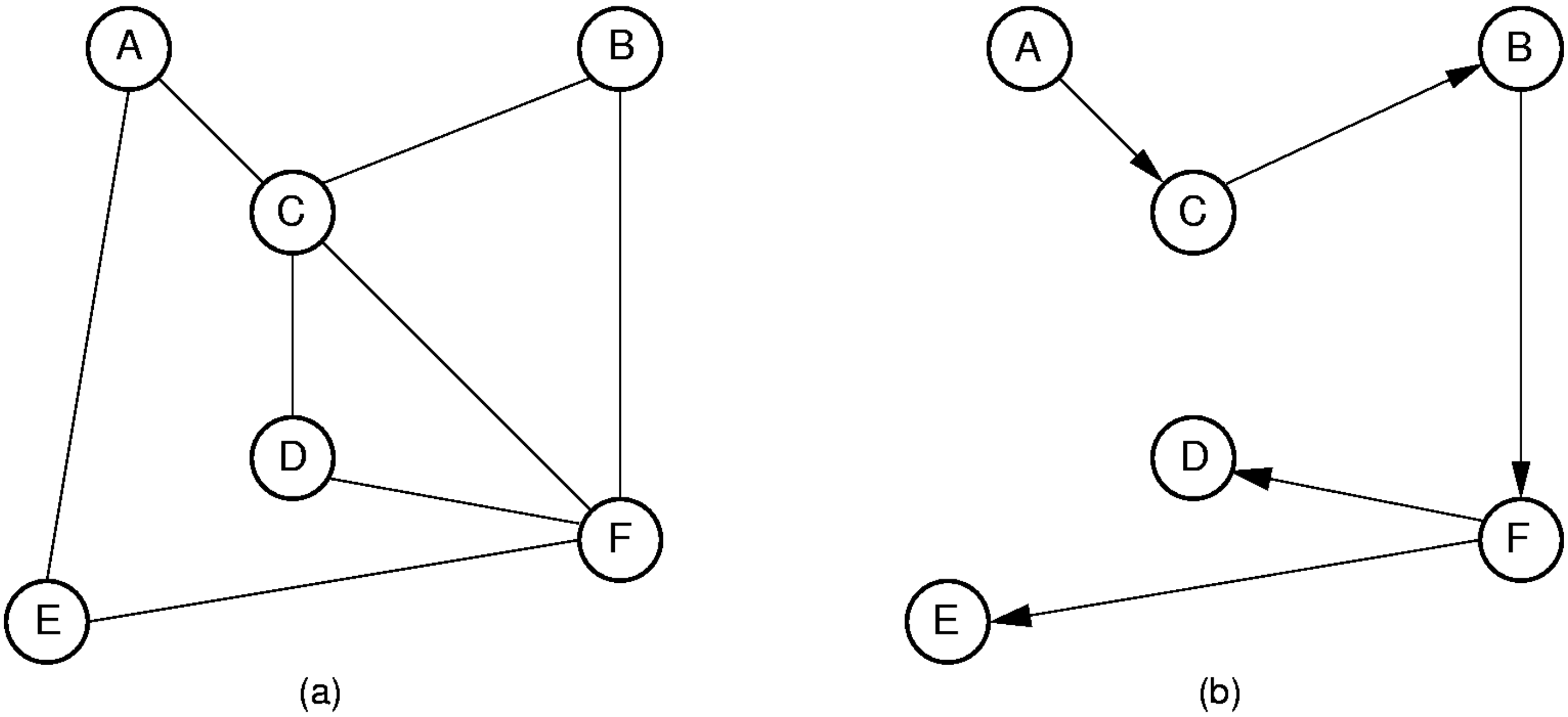
Starting Vertex: A

Depth First Search (2)

```
// Depth first search
void DFS(Graph G, int v) {
  PreVisit(G, v);  // Take appropriate action
  G.setMark(v, VISITED);
  for (int w = G.first(v); w < G.n();
                           w = G.next(v, w))
    if (G.getMark(w) == UNVISITED)
      DFS(G, w);
  PostVisit(G, v); // Take appropriate action
}
```



Depth First Search (3)



Cost: $\Theta(|\mathbf{V}| + |\mathbf{E}|)$.

Breadth First Search (1)

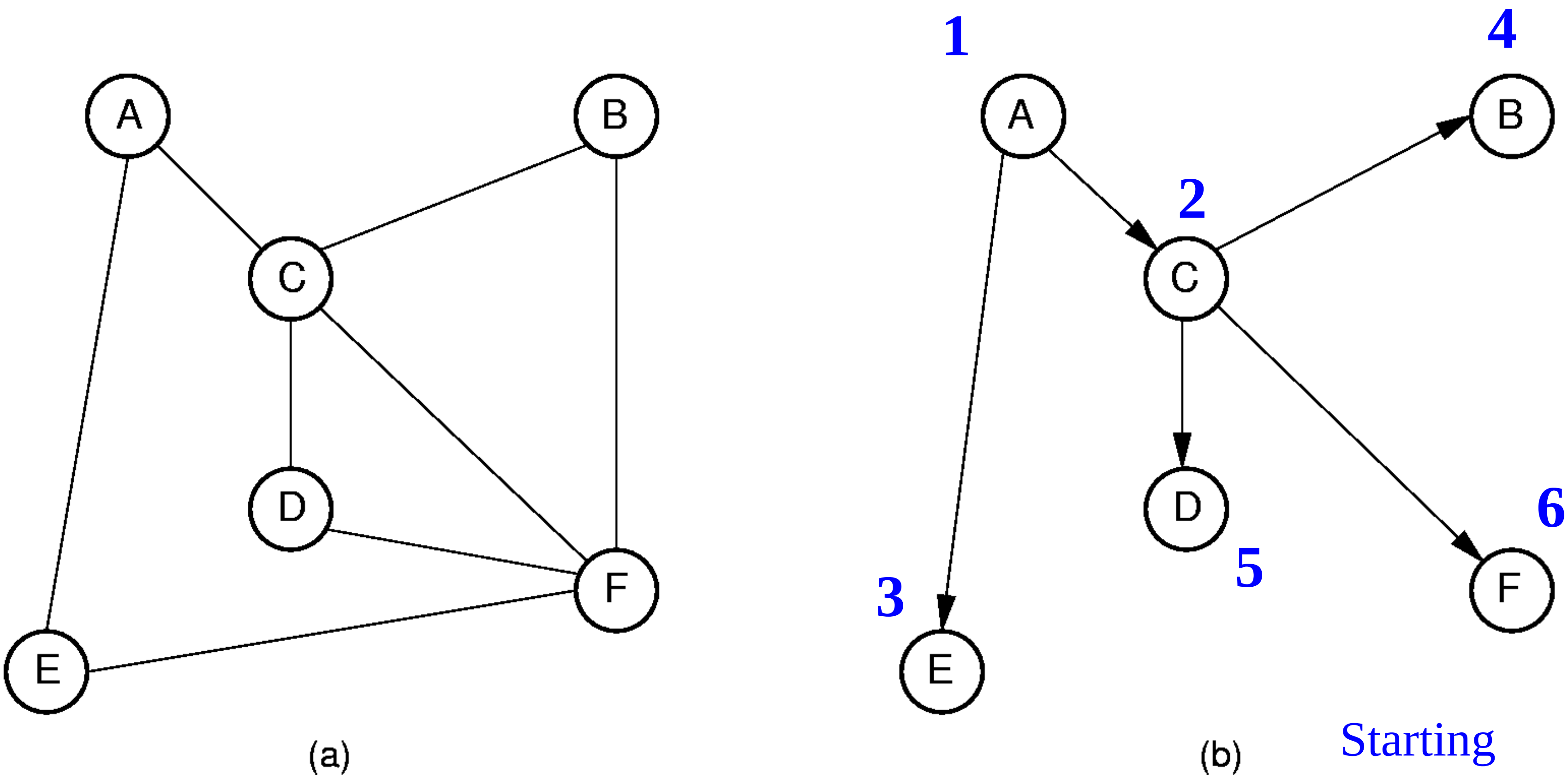
- Breadth First Search (BFS)
  - Like DFS, but replace stack with a queue.
  - Visit vertex's neighbors before continuing deeper in the tree.

- BFS Algorithm
  - Start from a vertex s
  - Visit all neighbors of s
  - Visit all neighbors of neighbors of s
  - … continue until all vertices are visited

Breadth First Search (2)

```
void BFS(Graph G, int start) {
  Queue<Integer> Q = new AQueue<Integer>(G.n());
  Q.enqueue(start);
  G.setMark(start, VISITED);
  while (Q.length() > 0) { // For each vertex
    int v = Q.dequeue();
    PreVisit(G, v); // Take appropriate action
    for (int w = G.first(v); w < G.n();
                             w = G.next(v, w))
      if (G.getMark(w) == UNVISITED) {
        // Put neighbors on Q
        G.setMark(w, VISITED);
        Q.enqueue(w);
      }
    PostVisit(G, v); // Take appropriate action
  }
}
```

Breadth First Search (3)

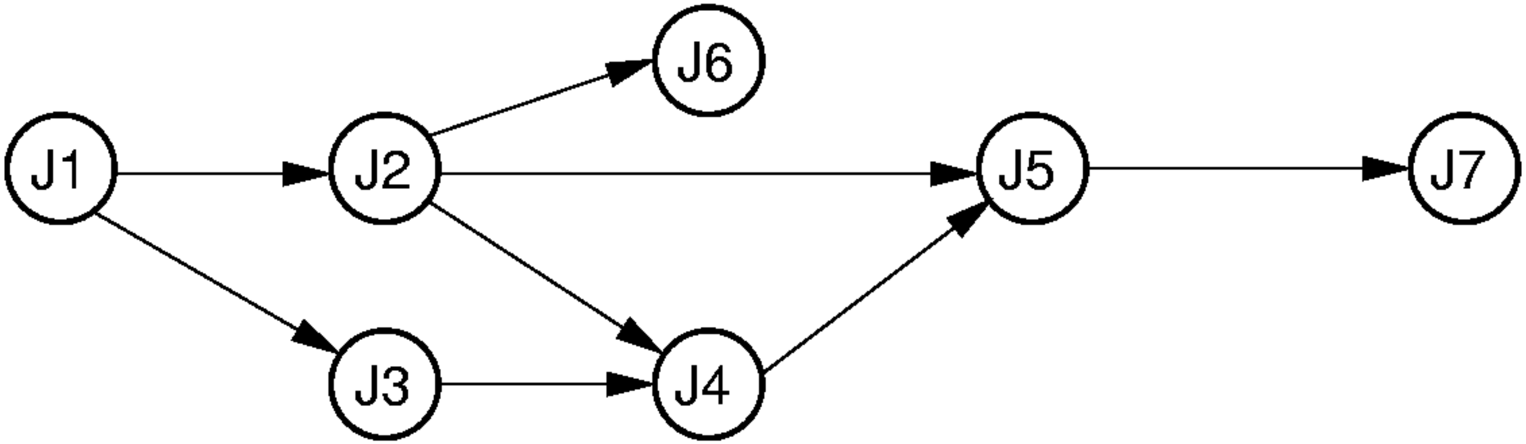


Starting Vertex: A

Cost: $\Theta(|\mathbf{V}| + |\mathbf{E}|)$.

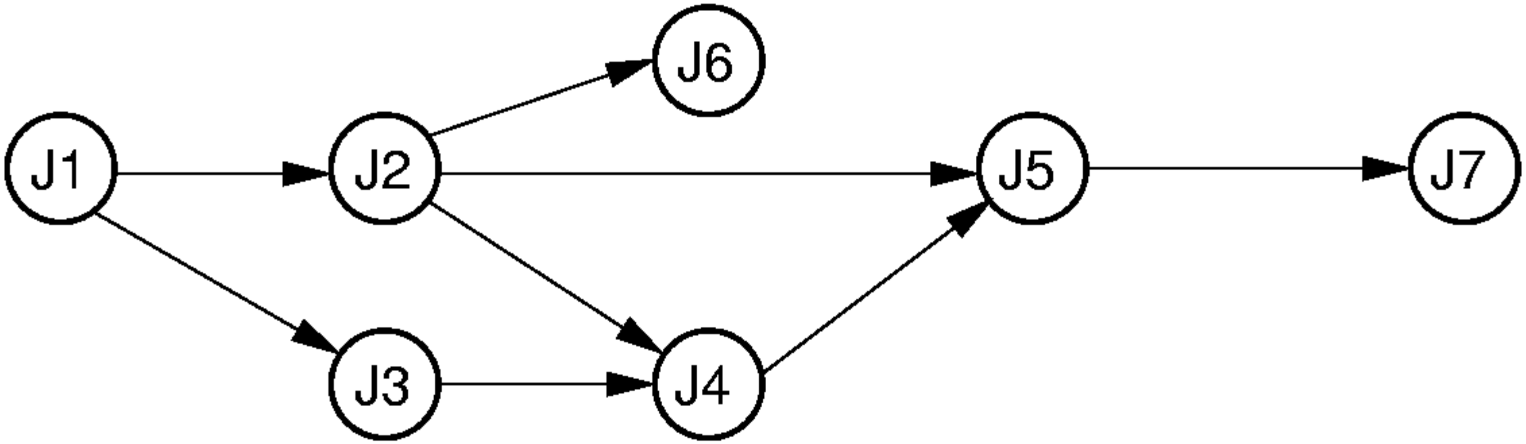
Topological Sort (1)

- Problem: Given a set of jobs, courses, etc., with prerequisite constraints, output the jobs in an order that does not violate any of the prerequisites.
  - J2 cannot start before J1; J4 cannot start before J2 and J3; …



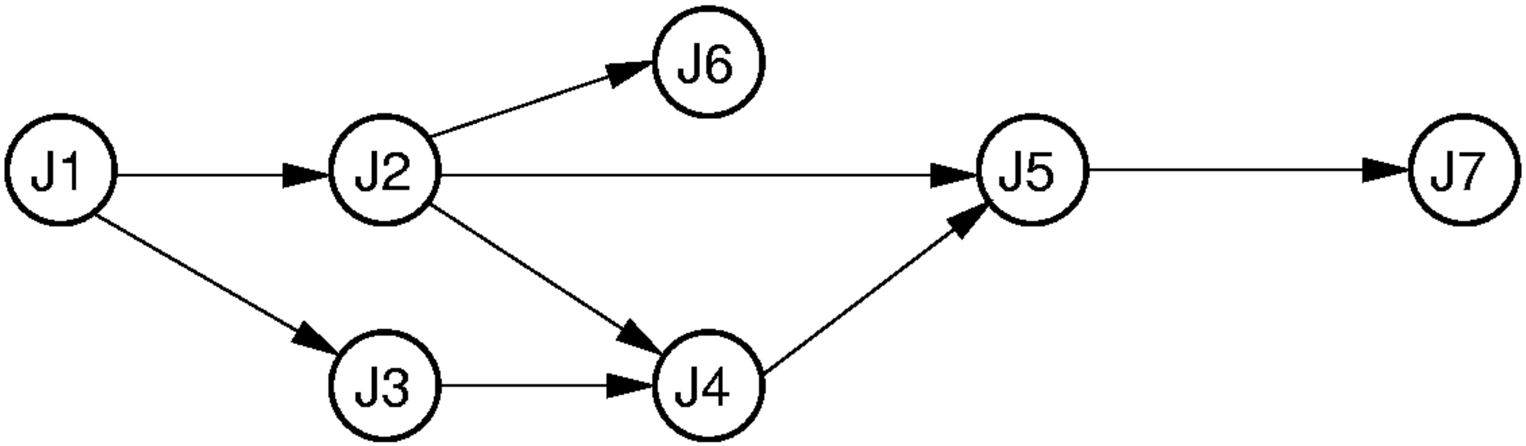
Topological Sort (2)

- May have several solutions
  - E.g., It doesn't matter which of J4 or J6 comes first; same for J2 or J3
  - J1, J3, J2, J4, J6, J5, J7 is a valid solution
  - J1, J2, J3, J6, J4, J5, J7 is a valid solution, too
- Algorithm
  - Based on DFS
  - Based on Queue



Topological Sort with DFS (1)

- Main Idea (reverse topological sort)
  - Perform DFS from each of the vertices, visiting unvisited vertices; print out a vertex v in PostVisit for v
  - It prints out vertices in reverse topological sort order
  - Correctness: Assume a dependency from v1 to v2. Can v1 be printed before v2? Why?

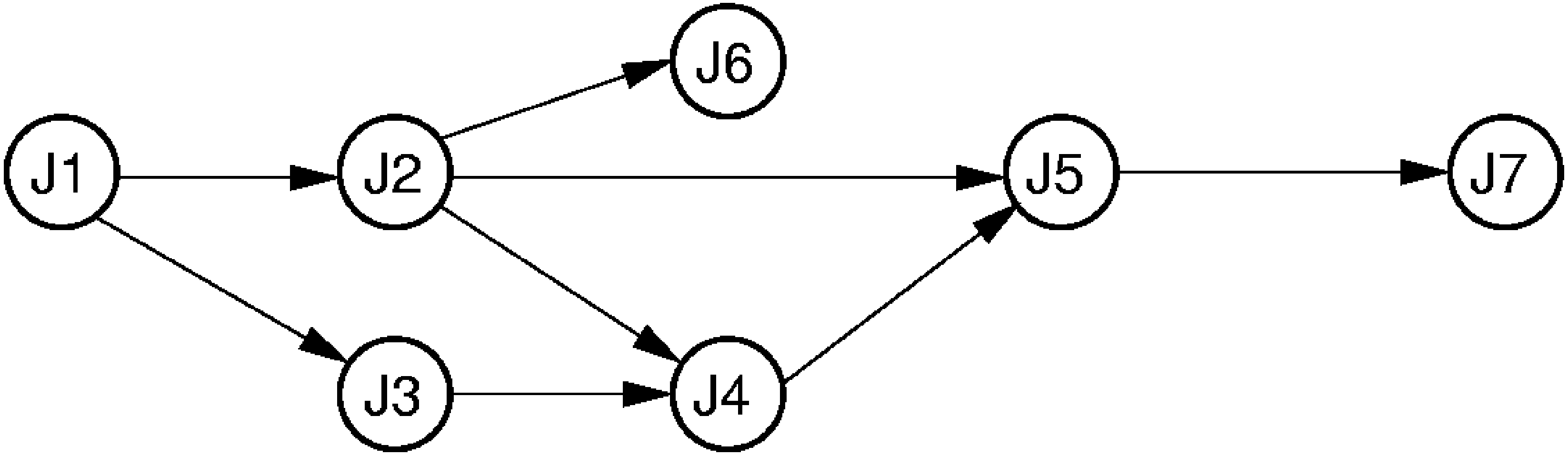


Topological Sort with DFS (2)

```
void topsort(Graph G) {
  for (int i=0; i<G.n(); i++)
    G.setMark(i, UNVISITED);
  for (int i=0; i<G.n(); i++)
    if (G.getMark(i) == UNVISITED)
      tophelp(G, i);
}

void tophelp(Graph G, int v) {
  G.setMark(v, VISITED);
  for (int w = G.first(v); w < G.n();
                           w = G.next(v, w))
    if (G.getMark(w) == UNVISITED)
      tophelp(G, w);
  printout(v);
}
```

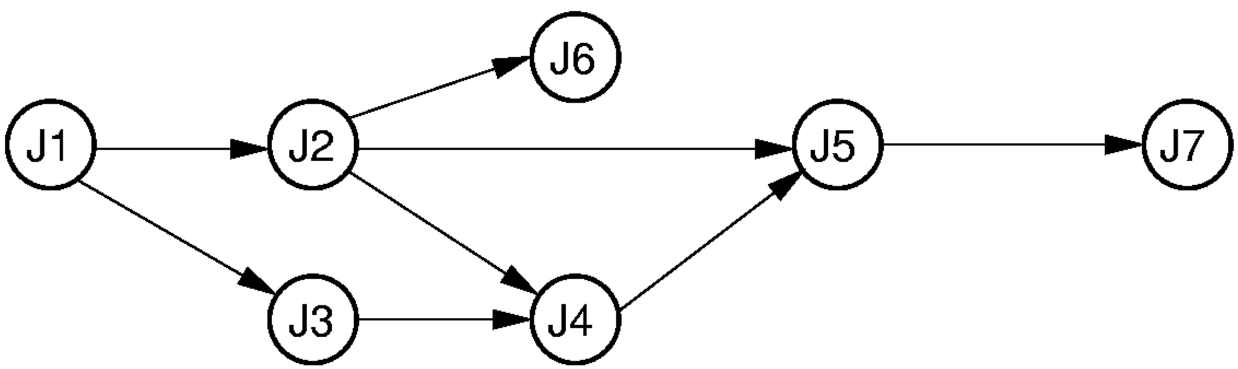
Topological Sort with DFS (3)



- Is the order of calling vertices important?
  - No. Why?

Topological Sort with Queue (1)

- Main Idea
  - Visit all edges, counting the number of incoming edges for each vertex
  - All vertices with no incoming edges are on the queue
  - Process the queue
    - When v is taken off the queue, print v, and all outgoing neighbors of v's counts decrement by one
    - Place on the queue any neighbor of v with count zero.



Topological Sort with Queue (2)

```
void topsort(Graph G) {
  Queue<Integer> Q = new AQueue<Integer>(G.n());
  int[] Count = new int[G.n()];
  int v, w;
  for (v=0; v<G.n(); v++) Count[v] = 0;
  for (v=0; v<G.n(); v++)
    for (w=G.first(v); w<G.n(); w=G.next(v, w))
       Count[w]++;
  for (v=0; v<G.n(); v++)
    if (Count[v] == 0) Q.enqueue(v);
  while (Q.length() > 0) {
    v = Q.dequeue().intValue();
    printout(v);
    for (w=G.first(v); w<G.n(); w=G.next(v, w)) {
      Count[w]--;
      if (Count[w] == 0)
        Q.enqueue(w);
    }
  }
}
```

What You Need to Know

- Basic terms and definitions of graphs
- How to represent graphs
  - When to use adjacency matrix or adjacency list
- Graph traversal methods
  - Main ideas and costs of DFS, BFS, and topological sort

Questions?