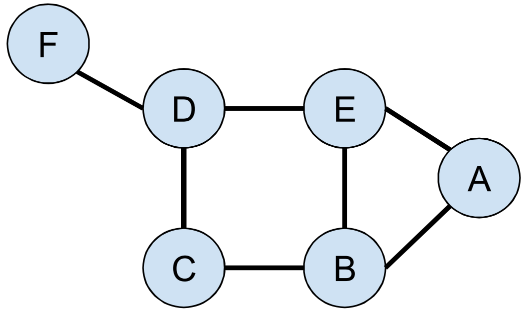
COMP2421—DATA STRUCTURES AND ALGORITHMS

Graphs

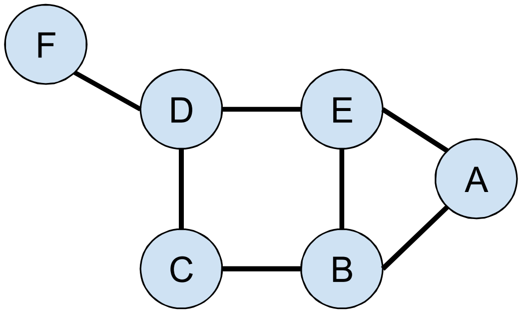
Dr. Radi Jarrar
Department of Computer Science
Birzeit University

Graphs



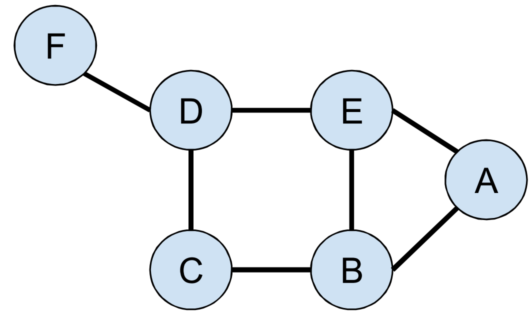
- Graphs are mathematical concepts that have many applications in computer science.
- They have many applications in real-life applications such as social networks, locations and routers in GPS, …
- A graph consists of a finite set of vertices (i.e., nodes) and a set of edges connecting these vertices.
- Two vertices are called adjacent if they are connected to each other by the same edge.

Graphs



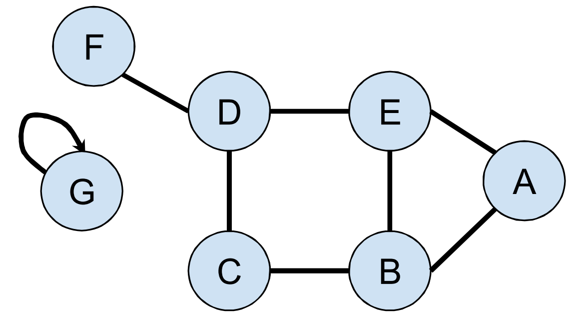
- A graph G=(V, E), is a data structure that consists of a finite set of vertices (or nodes) V, and a set of edges, E.
- Each edge is a pair (v, w) where v and w are nodes from V.

Graphs



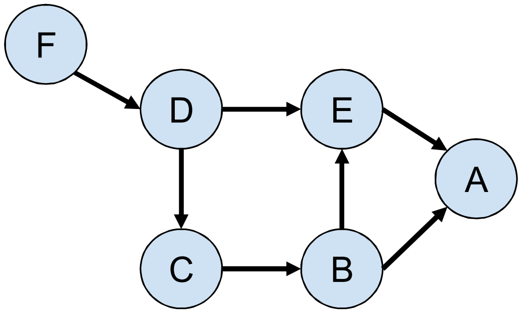
- If the pairs are ordered in the graph, then the graph is called **directed graph**(diagraphs).
- Vertex w is **adjacent** to v if and only if (v, w) ∈ E. In an undirected graph with edge (v, w), and hence (w, v), w is adjacent to v and v is adjacent to w.

Graphs - Definitions



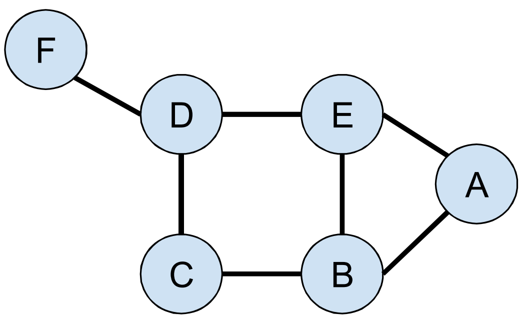
- Order: is the number of vertices in a graph
- Size: is the number of edges in a graph
- Vertex degree: is the number of edges that are connected to a vertex
- Isolated vertex: is the vertex that is not connected to any other vertex in the graph
- Self-loop: an edge from a vertex to itself

Graphs - Definitions

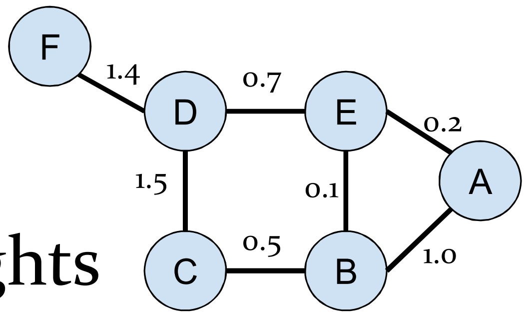


- Directed graph: is a graph where all edges have directions indicating what is the start vertex and what is the end vertex

- Undirected graph: is a graph with edges that have no directions

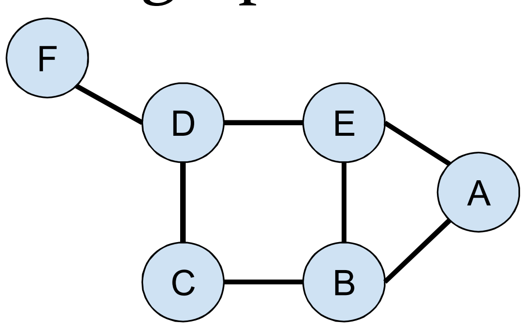


Graphs - Definitions

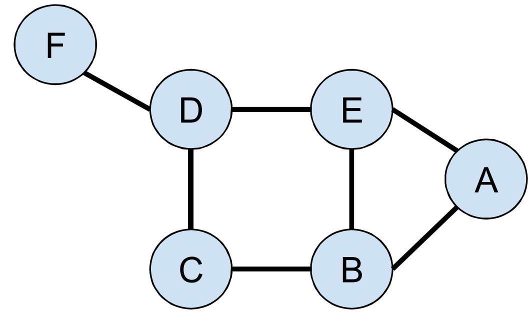


- Weighted graph: edges of a graph have weights

- Unweighted graph: edges of a graph have no weights

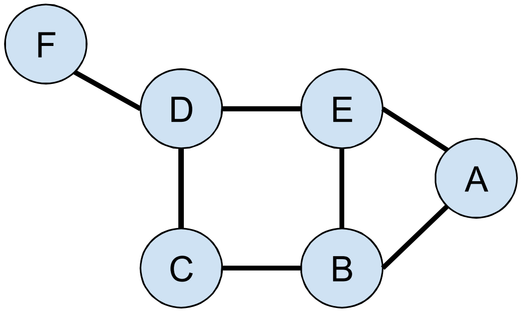


Graphs - Definitions



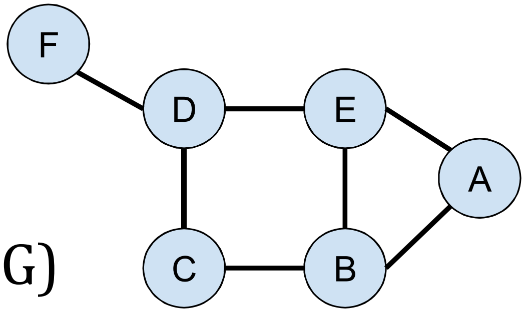
- A **path** in a graph is a sequence of vertices $w_1, w_2, w_3, \ldots, w_N$, such that $(wi,wi+1) \in E$ for $1 \le i < N$. The **length** of such a path is the number of edges on the path, which is equal to $N - 1$.
- A path from a vertex to itself is allowed. If it does not contain edges, then the path length is 0. If edge (v,v), then the path v (which is also referred to as a loop).
- Cycle: a path $w_1, w_2, w_3, \ldots, w_N$ for which N > 2, the first N - 1 vertices are all different, and $w_1 = w_N$. For example, the sequence D, E, A, B, C, D is a cycle in the graph above.

Graphs - Definitions



- A simple path is a path such that all vertices are distinct (except that the first and last might be the same).
- A cycle in a directed graph is a path of length at least one such that $w_1 = w_n$.
- The path v, u, v is cyclic. However, it is not in undirected graph because (v,u) and (u,v) is the same path.

Graphs - Definitions



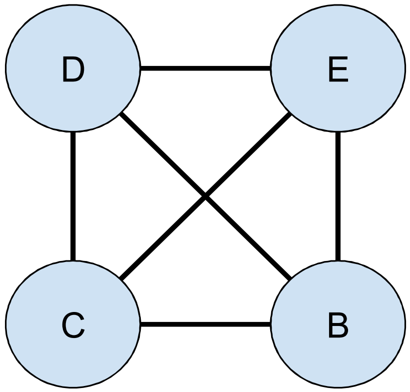
- A directed graph is called acyclic if it has no cycles (DAG)

  - Acyclic directed graph.

- An undirected graph is called connected if there is a path from every node to every other node. A directed graph with this property is called strongly connected.

Graphs - Definitions

- A complete graph is a graph in which there is an edge between every pair of vertices.



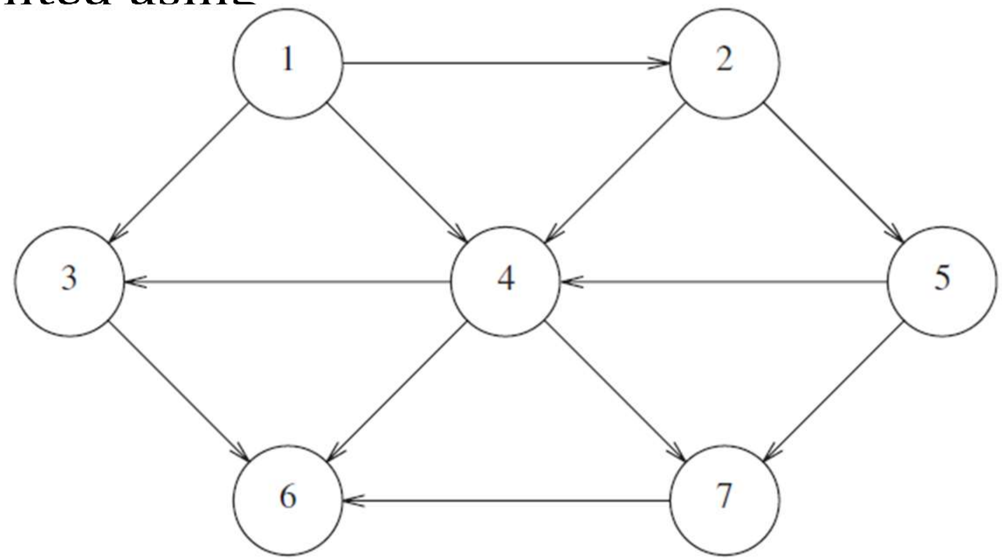
Examples of using graphs

- Airport System
- Graphs are used to represent networks. The networks may include paths in a city or telephone network or circuit network.
- Graphs are also used in social networks like LinkedIn, Facebook. For example, in Facebook, each person is represented with a vertex(or node). Each node is a structure and contains information like person id, name, gender, and locale.

REPRESENTATION OF GRAPHS

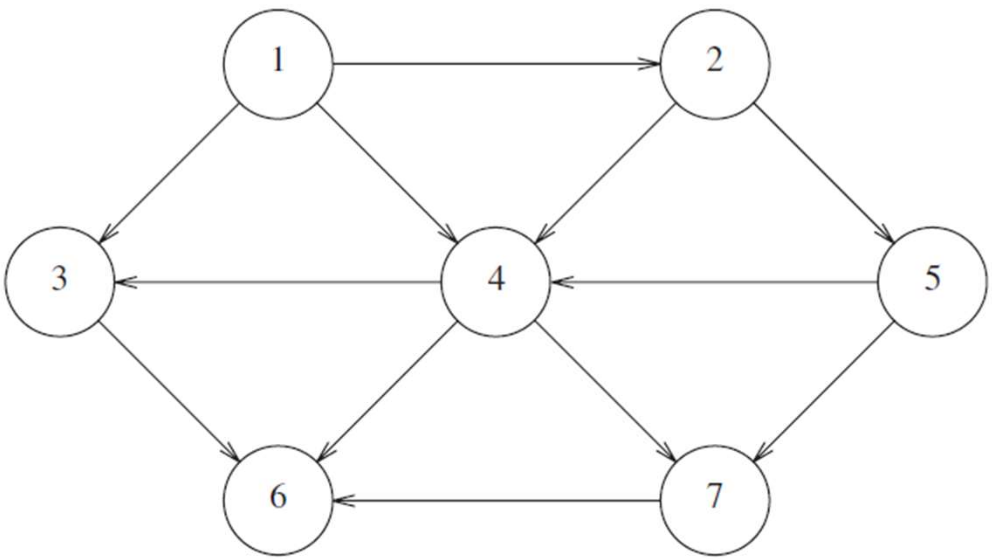
Graph Representation

- A graph is a data structure that consists of two main components: a finite set of vertices (i.e., nodes); and a finite set of ordered pairs called edges
- Graphs are most commonly represented using
  - Adjacency matrix
  - Adjacency list



Graph Representation

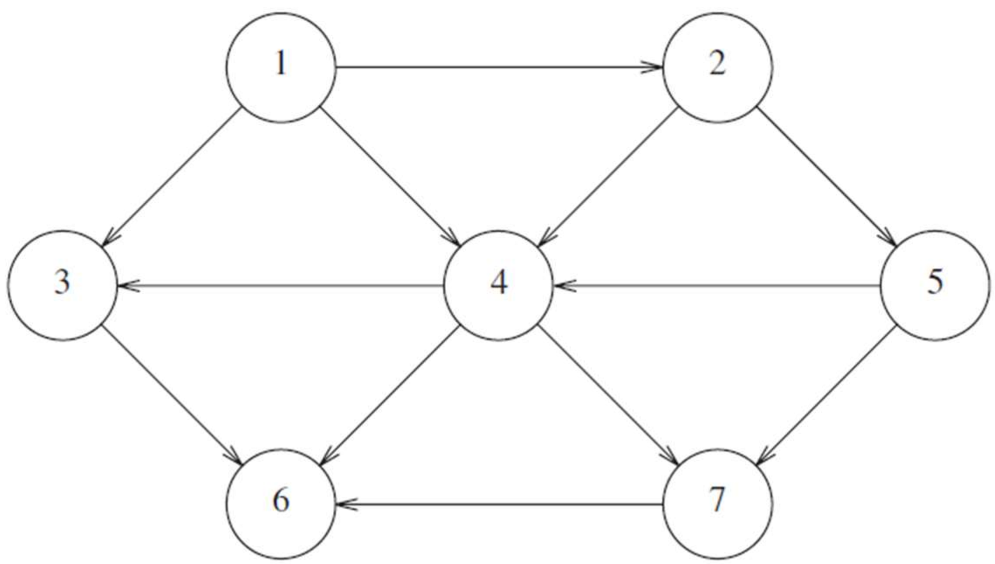
- Consider the following directed graph (the undirected graph is represented the same way)
- Suppose that we can number the vertices starting at 1. This graph has 7 vertices and 12 edges.
- One method is to represent a graph using a 2D array (adjacency matrix)



Adjacency Matrix

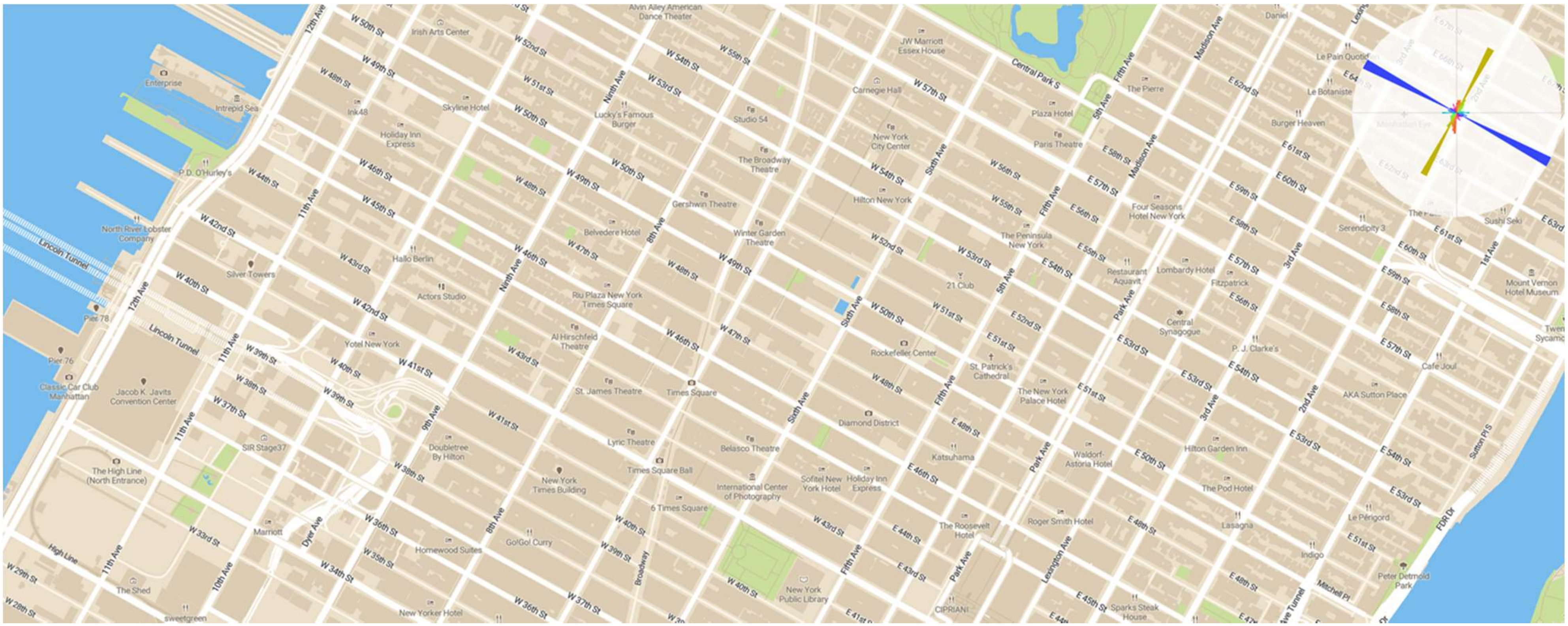
- Adjacency Matrix: maintain a 2D-Boolean array of size v * v where v is the number of vertices in the graph.
- Let the adjacency matrix adj, each edge is represented with the value true: adj[v][w] = true for the edge (v, w)
- The boolean value can be replaced with a weight to represent a weighted graph
- For undirected graph, the adjacency matrix is symmetric

Adjacency Matrix



	1	2	3	4	5	6	7
1		1	1	1			
2				1	1		
3						1	
4			1			1	1
5				1			1
6							
7						1	

Adjacency Matrix



Adjacency Matrix

Advantages:

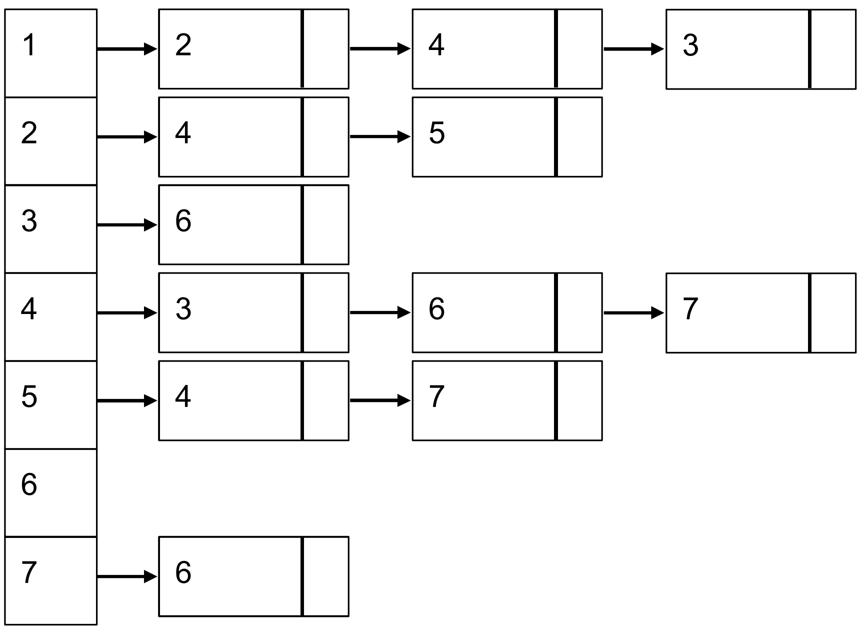
- Easy to implement and follow
- Removing and edge/checking if an edge exists in the graph takes O(1)

Disadvantages:

- Requires more space $O(n^2)$ if the graph has a few number of edges between vertices
- Adding a vertex will consume $O(n^2)$
- Very slow to iterate over all edges

Adjacency List

- Is a better solution if the graph is sparse (not dense)
- For each vertex, we keep a list of all adjacent vertices
- The space requirement is then O(|E| + |V|), which is linear in the size of the graph



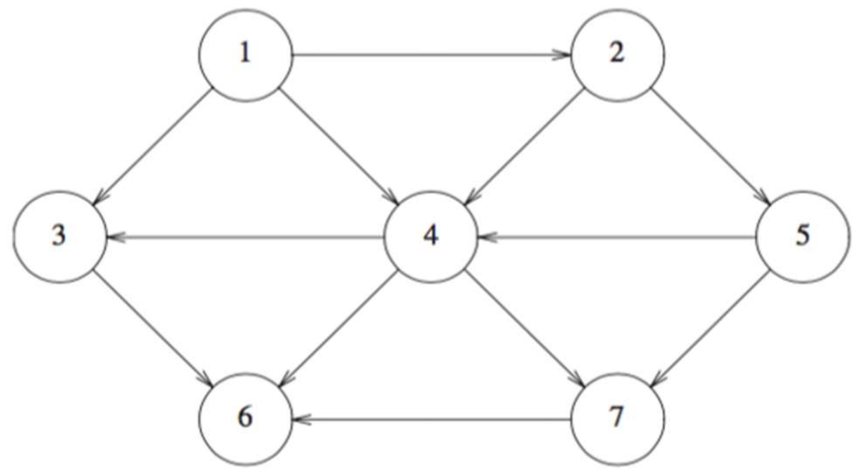


Figure 9.1 A directed graph

Adjacency List

- Adjacency lists are the standard way to represent graphs
- Undirected graphs can be similarly represented; each edge (u, v) appears in two lists, so the space usage essentially are doubled
- A common requirement in graph algorithms is to find all vertices adjacent to some given vertex v, and this can be done in time proportional to the number of such vertices found, by a simple scan down the appropriate adjacency list

Adjacency List

Advantages:

- Fast to iterate over all edges
- Fast to add/delete a node (vertix)
- Fast to add a new edge O(1)
- Memory depends more on the number of edges (and less on the number of nodes), which saves more memory if the adjacency matrix is sparse

Disadvantages:

- Finding a specific edge between any two nodes
  is slightly slower than the matrix O(k); where k is the number of neighbors nodes

SORTING GRAPHS

Topological Sort

- A linear order of the vertices in a directed graph
- A topological sort is an ordering of vertices in a directed acyclic graph, such that if there is a path from v_i to v_j, then v_j appears after vi in the ordering
- An example is the a directed graph that represents the prerequisite of courses in the figure below

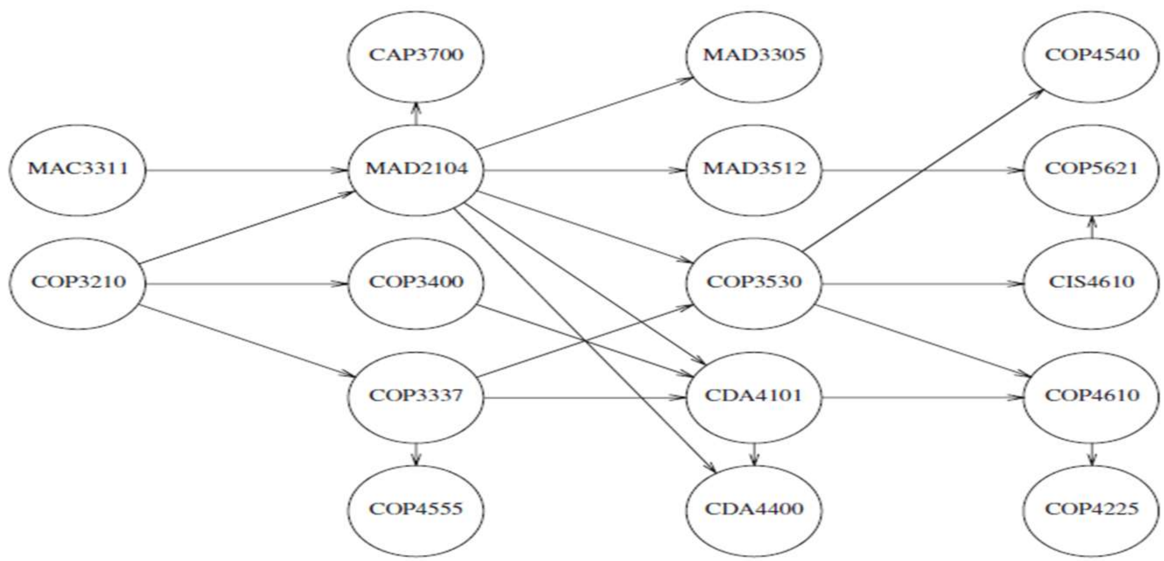
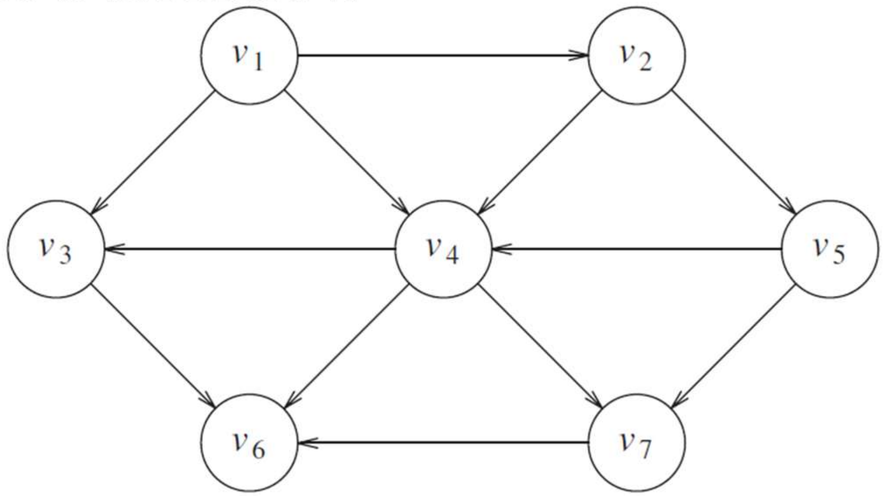


Figure 9.3 An acyclic graph representing course prerequisite structure

Topological Sort

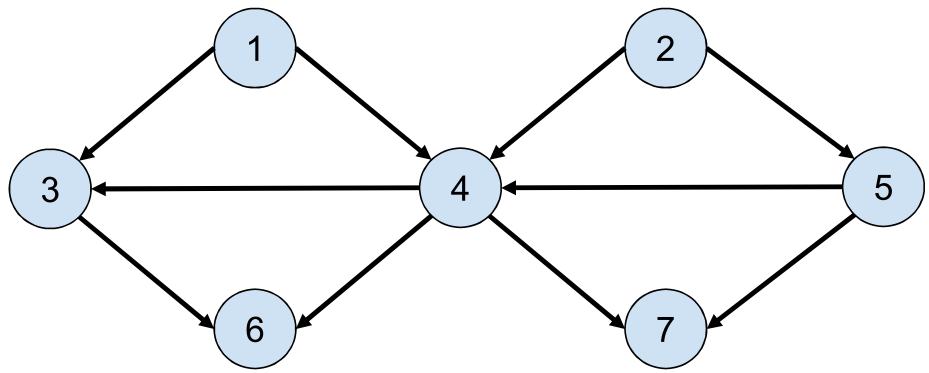
- A directed edge (v, w) indicates that course v must be completed before course w may be attempted
- A topological ordering of these courses is any course sequence that does not violate the prerequisite requirement
- Topological ordering is not possible if the graph has a cycle, since for two vertices v and w on the cycle, v precedes w and w precedes v.
- The ordering is not necessarily unique; any legal ordering will work.
- In this graph, $v_1, v_2, v_5, v_4, v_3, v_7, v_6$ and $v_1, v_2, v_5, v_4, v_7, v_3, v_6$ are both topological orderings.



Topological Sort

- Main idea: find a vertex with nothing going into it (i.e., Starting point). Write it down. Remove it and go through the other vertices and check for anyone with nothing coming into it. Repeat.
- scan all vertices to find the starting point
- * if edge (A, B) exists, A must precede B in the final order.
- Algorithm:
- Assume indegree is sorted with each node
- Repeat until no nodes remain
  - Choose a node of zero indegree and output it
  - Remove the node and all its edges and update indegree

Topological Sort - Example



- Indegree:

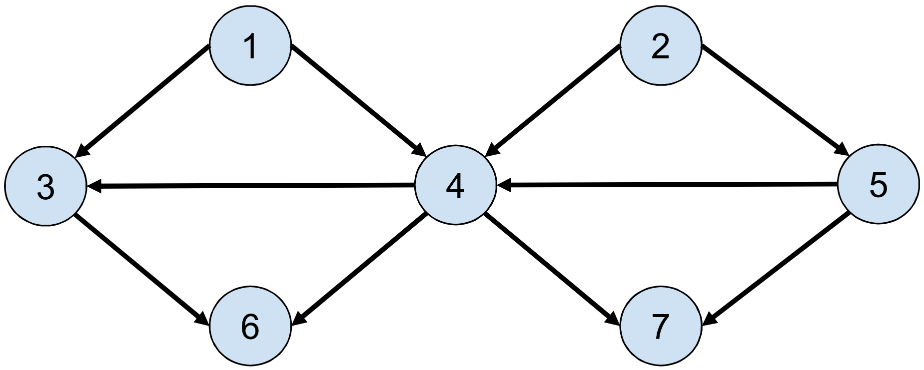
0:

1:

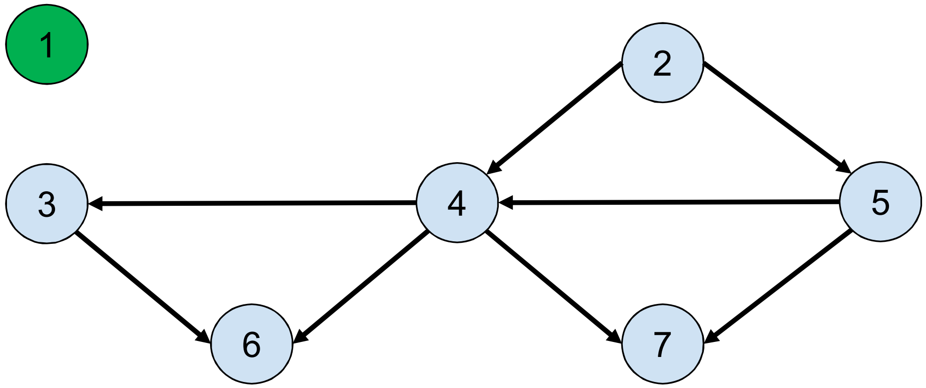
2:

3:

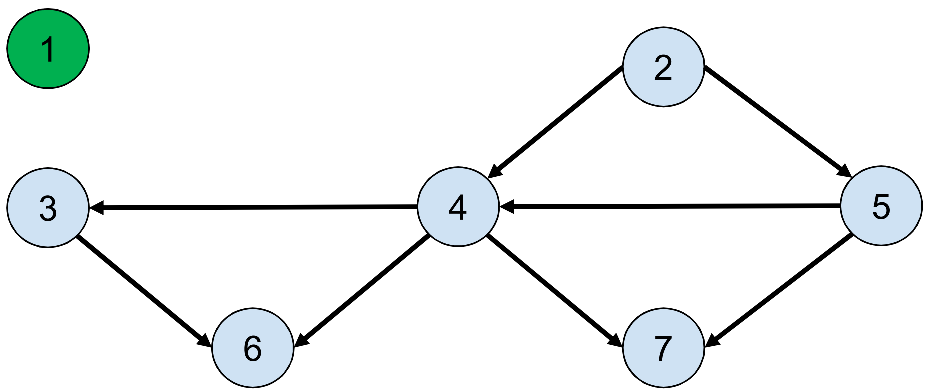
Topological Sort - Example



- Pick 1 and then update:



Topological Sort - Example



- Indegree:

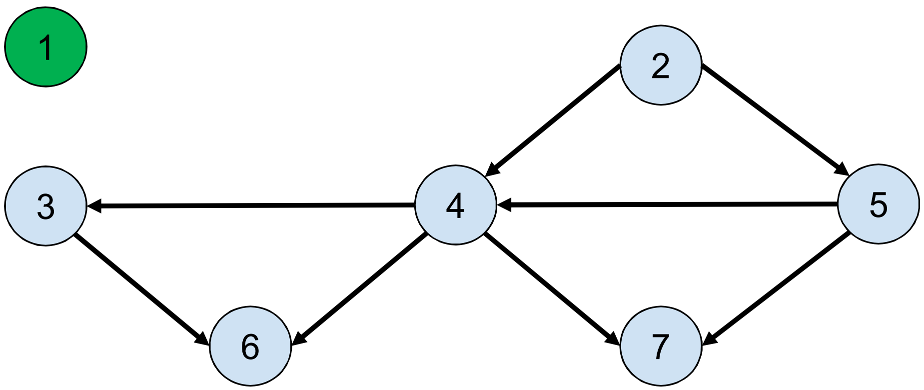
0:

1:

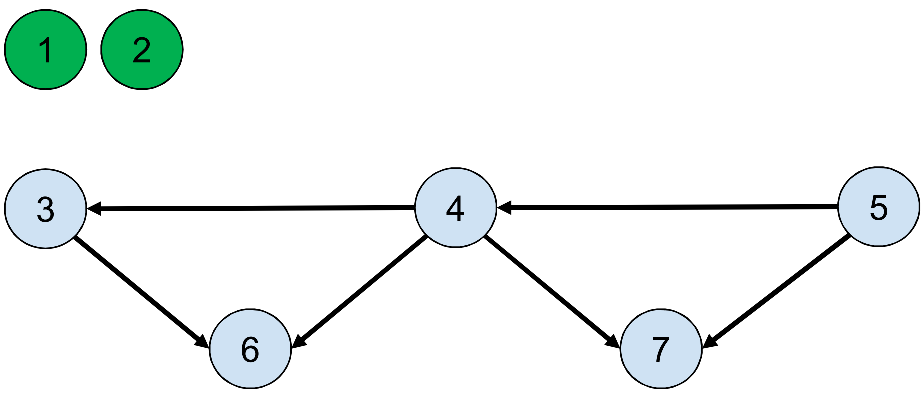
2:

3:

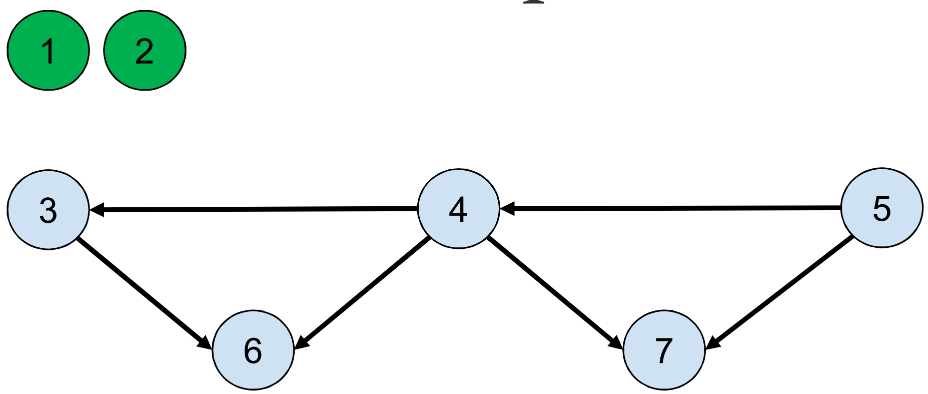
Topological Sort - Example



- Pick 2 and then update:



Topological Sort - Example



- Indegree:

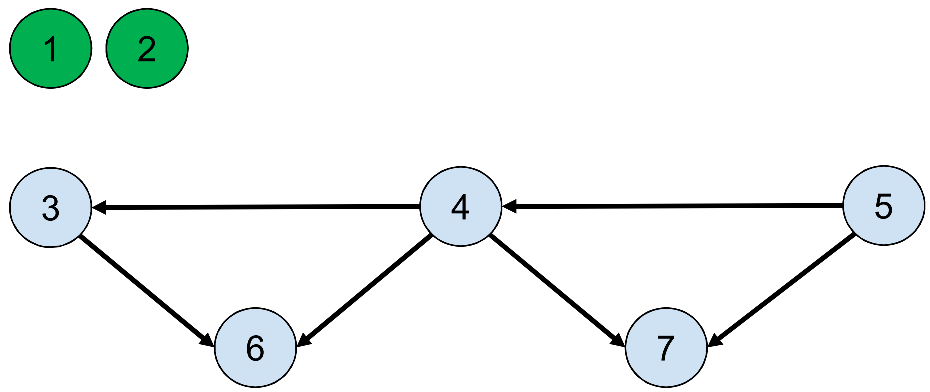
0:

1:

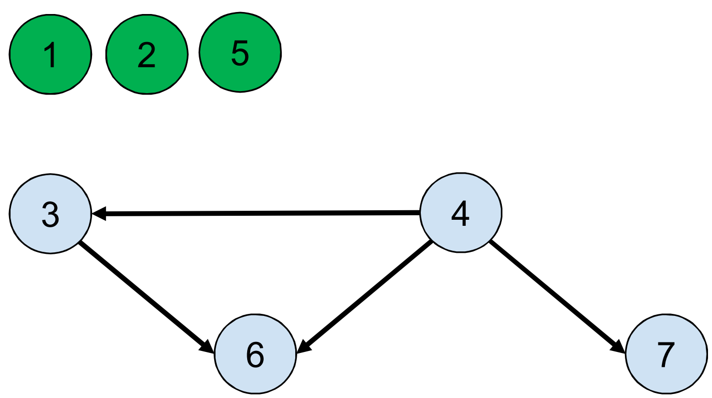
2:

3:

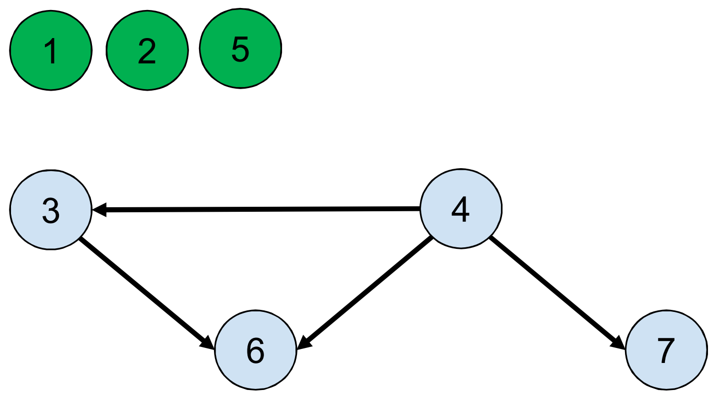
Topological Sort - Example



- Pick 5 and then update:



Topological Sort - Example



- Indegree:

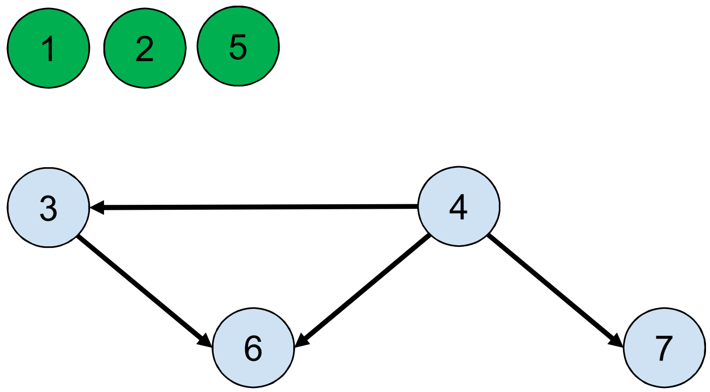
0:

1:

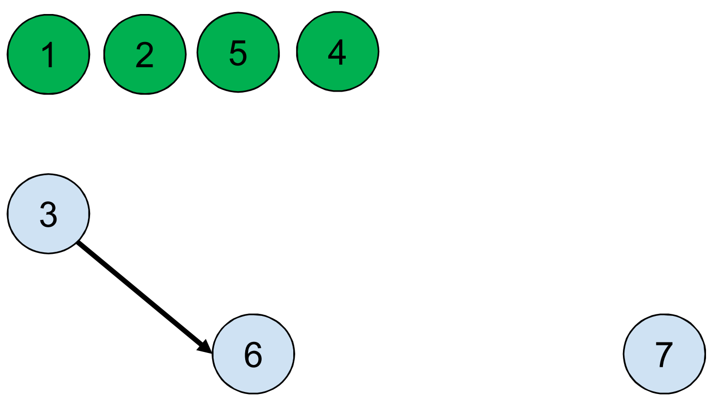
2:

3:

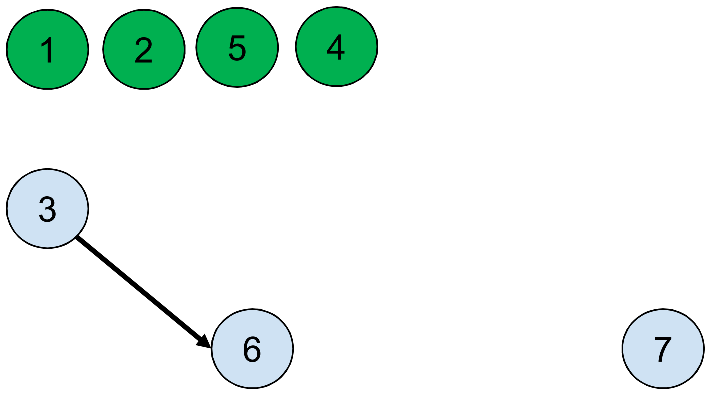
Topological Sort - Example



- Pick 4 and then update:



Topological Sort - Example



- Indegree:

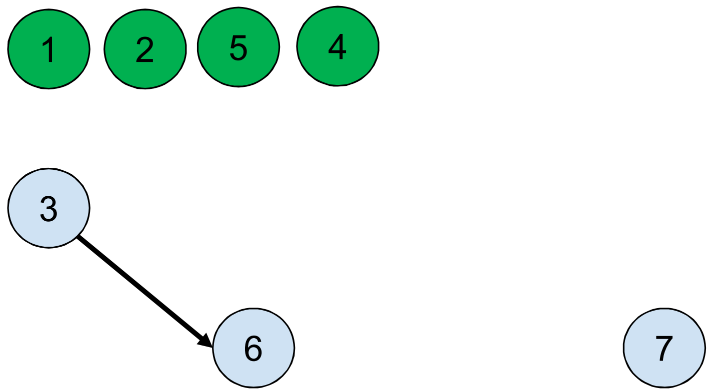
0:

1:

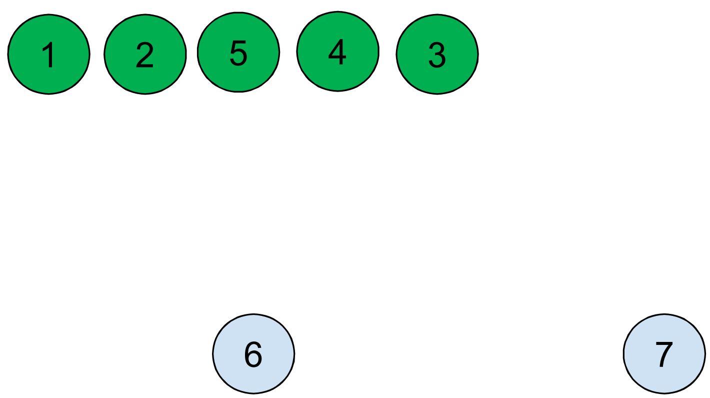
2:

3:

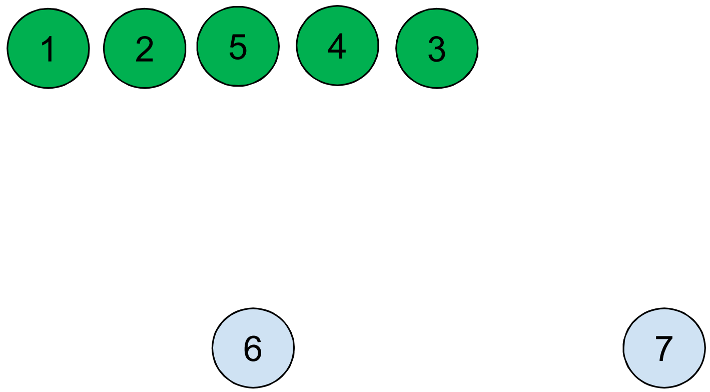
Topological Sort - Example



- Pick 3 and then update:



Topological Sort - Example



- Indegree:

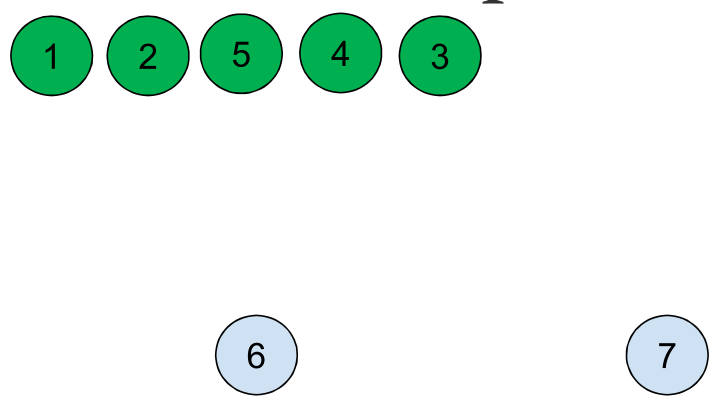
0:

1:

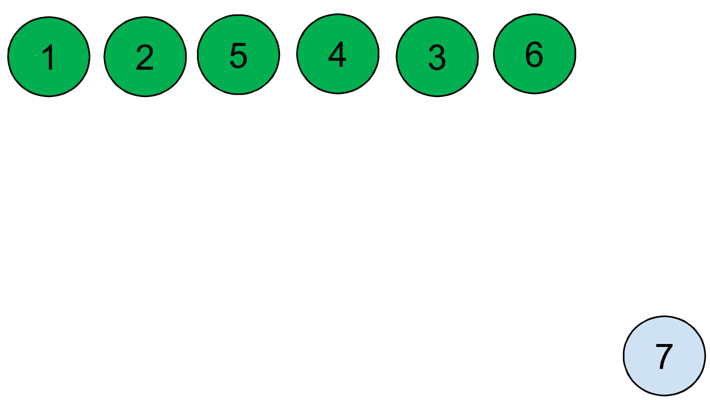
2:

3:

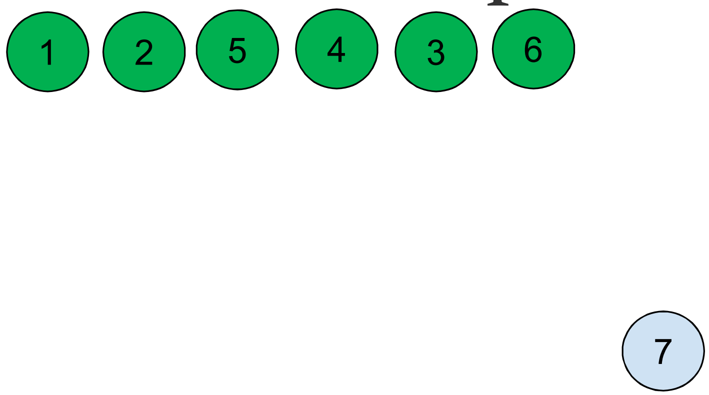
Topological Sort - Example



- Pick 6 and then update:



Topological Sort - Example



- Indegree:

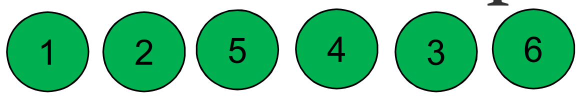
0:

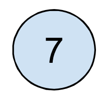
1:

2:

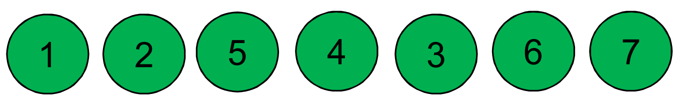
3:

Topological Sort - Example





- Pick 7 and then update:



Topological Sort

- First we find the nodes with no predecessors.
- Then, using a queue, we can keep the nodes with no predecessors and on each dequeue we can remove the edges from the node to all other nodes.

Topological Sort

- **Pseudocode:**

1. Represent the graph with two lists on each vertex (incoming edges and outgoing edges)
2. Make an empty queue Q;
3. Make an empty topologically sorted list T;
4. Push all items with no predecessors in Q;
5. While Q is not empty
    Dequeue from Q into u;
    Push u in T;
    Remove all outgoing edges from u;
6. Return T;

Topological Sort

- This approach will give us a running time complexity is $O(|V| + |E|)$.
- The problem is that we need additional space and an operational queue.

Topological Sort

- To find a topological ordering is first to find any vertex with no incoming edges. We can then print this vertex, and remove it, along with its edges, from the graph. Then we apply this same strategy to the rest of the graph.
- To formalize this, we define the indegree of a vertex v as the number of edges (u, v). We compute the indegrees of all vertices in the graph. Assuming that the indegree for each vertex is stored and that the graph is read into an adjacency list, we can then apply the algorithm in Figure 9.5 to generate a topological ordering.

Topological Sort - Pseudocode

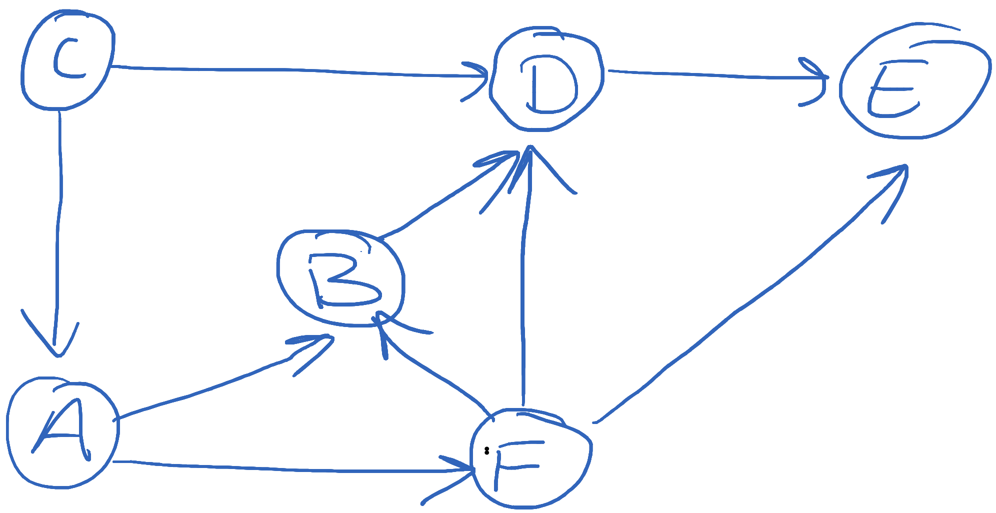
```
void topsort( Graph g)
{
	int counter;
	Vertex w, v;
	for( counter = 0; counter < NUM_VERTICES; counter++ )
	{
		v = find_new_vrtex_of_indegree_zero( );
		if( v == null ){
			printf("Graph has a cycle!\n");
			break;
		}
		topNum[v] = counter;

		for each Vertex w adjacent to v
		indegree[w]--;
	}
}
```

Topological Sort

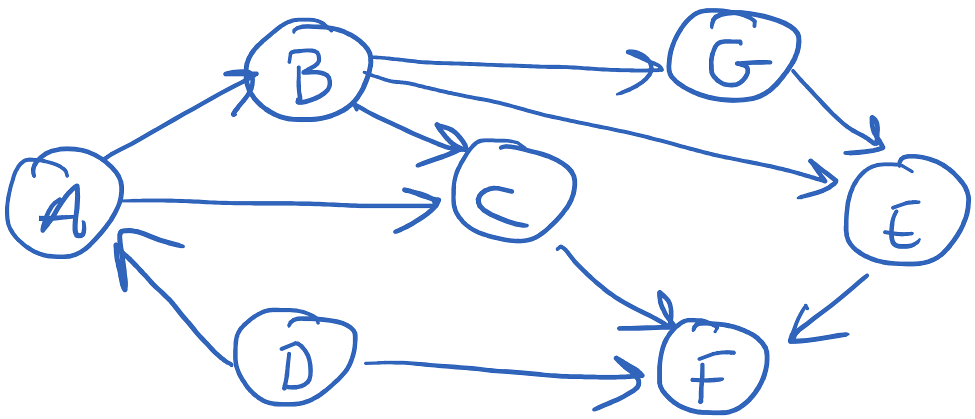
The method findNewVertexOfIndegreeZero scans the array looking for a vertex with indegree 0 that has not already been assigned a topological number. It returns null if no such vertex exists; this indicates that the graph has a cycle.

findNewVertexOfIndegreeZero is a simple sequential scan of the array of vertices, each call to it takes $O(|V|)$ time. Since there are $|V|$ such calls, the running time of the algorithm is $O(|V|^2)$.

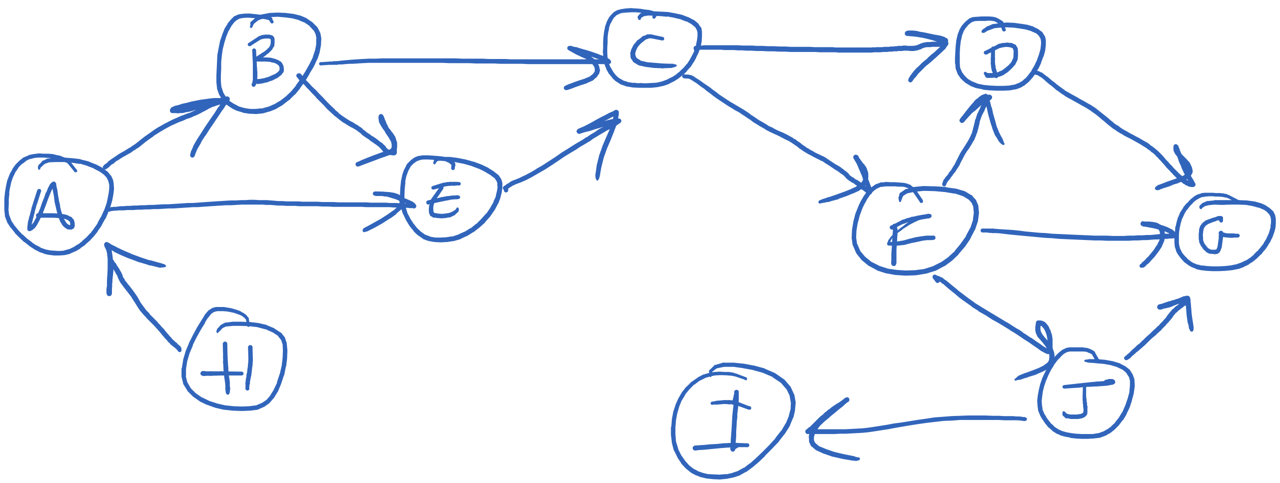
Topological Sort - Example



Topological Sort - Example



Topological Sort - Example



SEARCH ALGORITHMS

Shortest-Path Algorithms

- Shortest-path algorithms aim at finding the shortest path between nodes in a graph
- The input is a weighted graph: associated with each edge (v_i, v_j) is a cost $c_{i,j}$ to traverse the edge
- The cost of a path $v_1 v_2 \ldots v_N$ is $\sum_{i=1}^{N-1} c_{i,\, i+1}$
- This is referred to as the **weighted path length**
- The unweighted path length is the number of edges on the path, namely, N – 1

Single-Source Shortest-Path Algorithms

- Given as input a weighted graph, $G = (V, E)$, and a distinguished vertex, s, find the shortest weighted path from s to every other vertex in G.
- For example, the shortest weighted path from v_1 to v_6 has a cost of 6 and goes from v_1 to v_4 to v_7 to v_6
- The shortest unweighted path between these vertices is 2

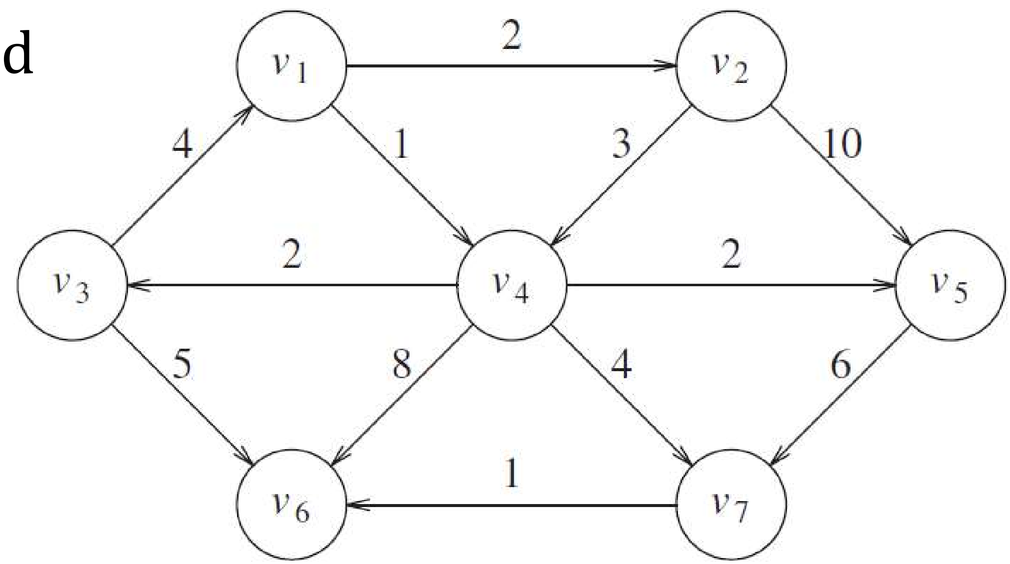


Figure 9.8 A directed graph G

Single-Source Shortest-Path Algorithms

- The shortest unweighted path between these vertices is 2
- Generally, when it is not specified whether we are referring to a weighted or an unweighted path, the path is weighted if the graph is.

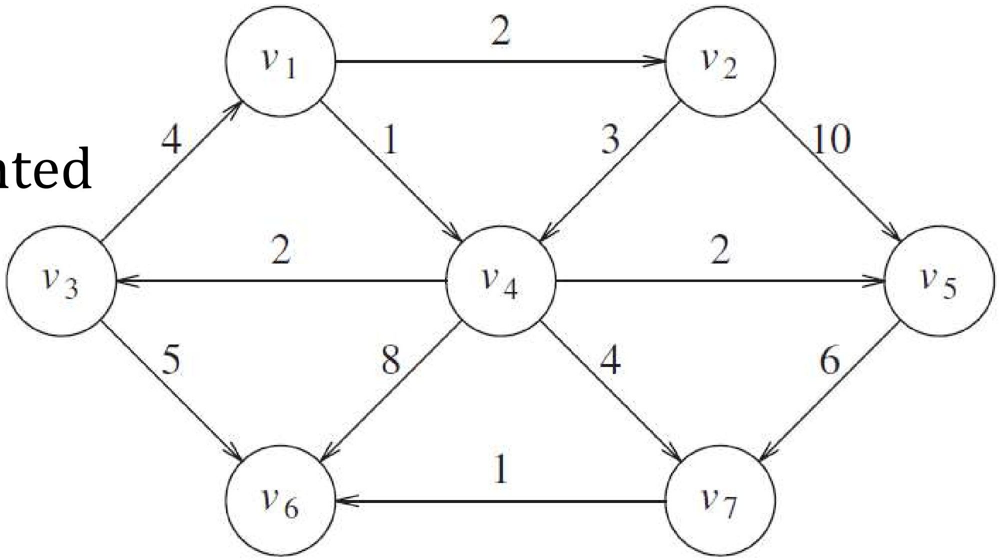


Figure 9.8 A directed graph G

Single-Source Shortest-Path Algorithms

- Having negative weights in the graph may cause some problems.
- The path from v_5 to v_4 has cost 1, but a shorter path exists by following the loop v_5, v_4, v_2, v_5, v_4, which has cost -5
- This path is still not the shortest, because we could stay in the loop arbitrarily long.
- Thus, the shortest path between these two points is **undefined**.

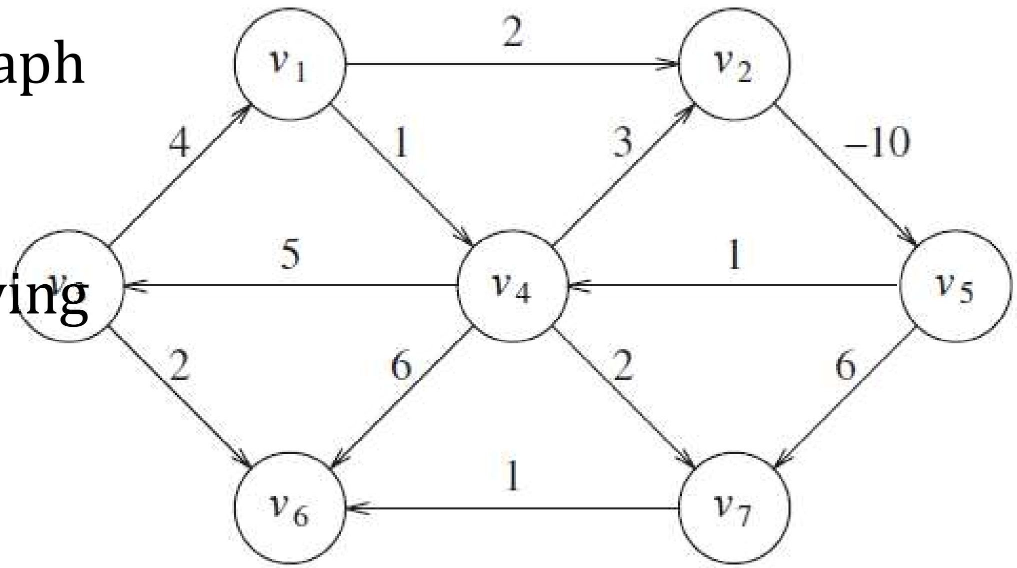


Figure 9.9 A graph with a negative-cost cycle

Single-Source Shortest-Path Algorithms

- Another example, the shortest path
- from v_1 to v_6 is undefined, because we can get into the same loop.
- This loop is known as a
- **negative-cost cycle;** when one is present in the graph, the shortest paths are not defined.

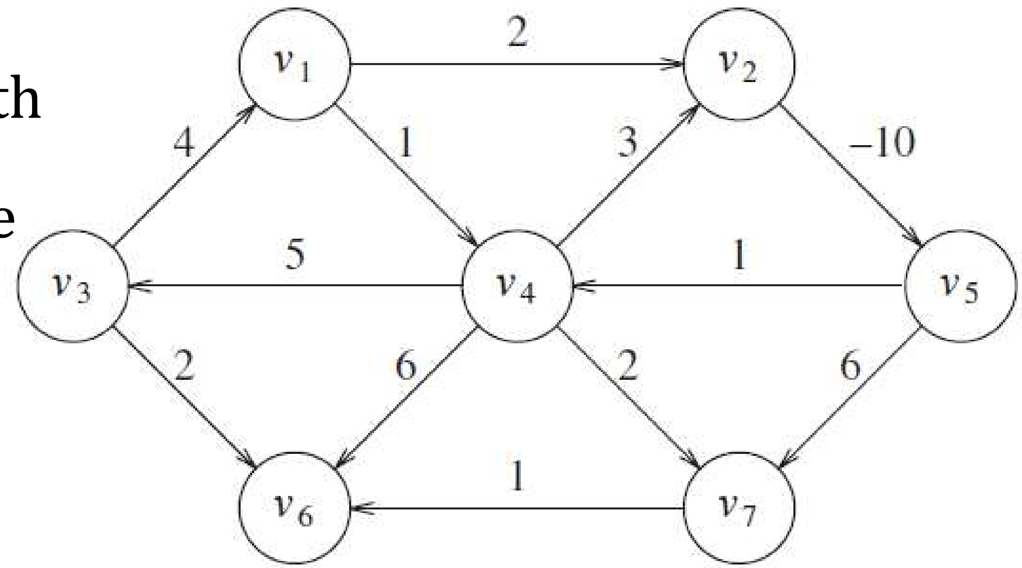


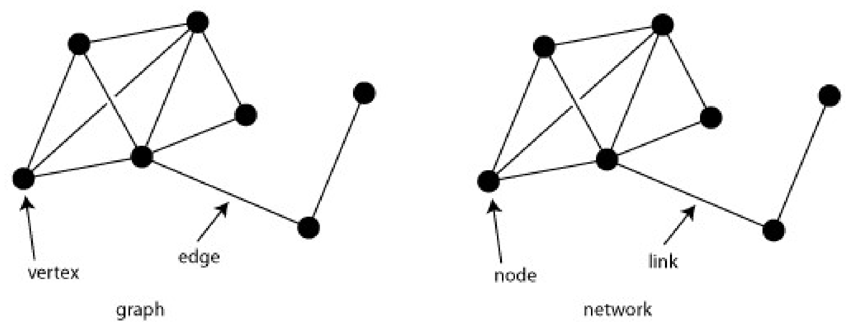
Figure 9.9 A graph with a negative-cost cycle

Single-Source Shortest-Path Algorithms

- Negative-cost edges are not necessarily bad, as the cycles are, but their presence seems to make the problem harder.
- For convenience, in the absence of a negative-cost cycle, the shortest path from s to s is zero.

Single-Source Shortest-Path Algorithms

- There are many examples where we might want to solve the shortest-path problem.
- If the vertices represent computers; the edges represent a link between computers; and the costs represent communication costs (phone bill per megabyte of data), delay costs (number of seconds required to transmit a megabyte), or a combination of these and other factors, then we can use the shortest-path algorithm to find the cheapest way to send electronic news from one computer to a set of other computers.



Single-Source Shortest-Path Algorithms

- Another example is to model an airplane (or transportation routes) by graphs and use a shortest path algorithm to compute the best route between two points.
- In this and many practical applications, we might want to find the shortest path from one vertex, *s*, to only one other vertex, *t*.
- Currently there are no algorithms in which finding the path from *s* to one vertex is any faster (by more than a constant factor) than finding the path from *s* to all vertices.
- We will solve 4 variations of this problem

Unweighted Shortest Paths

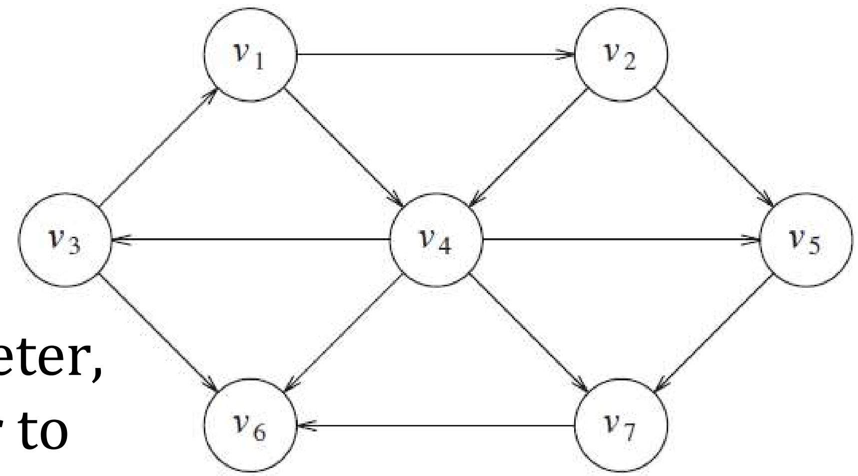


Figure 9.10 An unweighted directed graph G

- Given an unweighted graph, G. Using some vertex, s, which is an input parameter, we want to find the shortest path from s to all other vertices.
- We are only interested in the number of edges contained on the path (because there are no weights).
- This is clearly a special case of the weighted shortest-path problem, since we could assign all edges a weight of 1.

Unweighted Shortest Paths

- Suppose we are interested in the length of the shortest path not in the
- actual paths themselves. Keeping track of the actual paths will turn out to be a matter of simple bookkeeping.

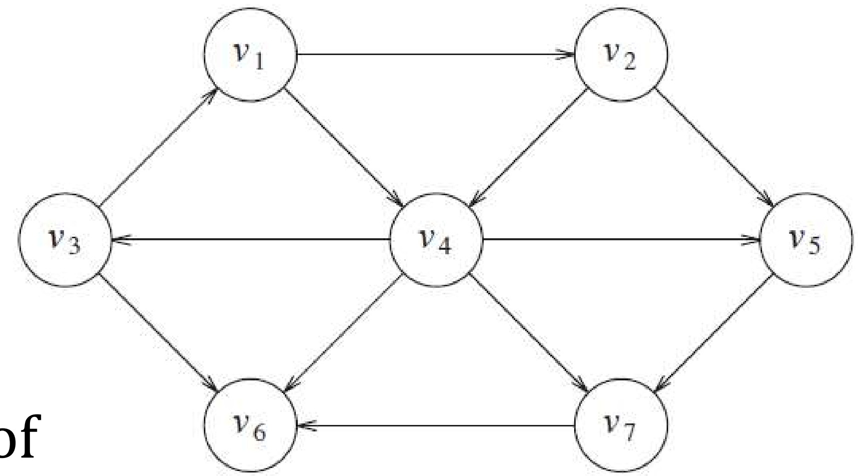


Figure 9.10 An unweighted directed graph G

Weighted Shortest-Path

Breadth-First Search (BFS)

Dijkstra's Algorithm

Minimum Spanning Tree