Northern Illinois University

Graphs

Dr. Maoyuan Sun – smaoyuan@niu.edu

Graphs – Terms

- **Graph** – a data structure that consists of a set of nodes and a set of edges that relate the nodes to one another; G = (V, E)
- **Vertex** – a node in a graph; V(G) is a finite, nonempty set of vertices
- **Edge** (**arc**) – a pair of vertices representing a connection between two nodes in a graph; E(G) is a set of edges (written as pairs of vertices)
- **Undirected graph** – a graph in which the edges have no direction
- **Directed graph** (**digraph**) – a graph in which each edge is directed from one vertex to another (or the same) vertex

Graphs – Terms

- **Adjacent vertices** – two vertices in a graph that connected by an edge
- **Path** – a sequence of vertices that connects two nodes in a graph
- **Complete graph** – a graph in which every vertex is directly connected to every other vertex
- **Weighted graph**– a graph in which each edge carries a value

Depth-First Search

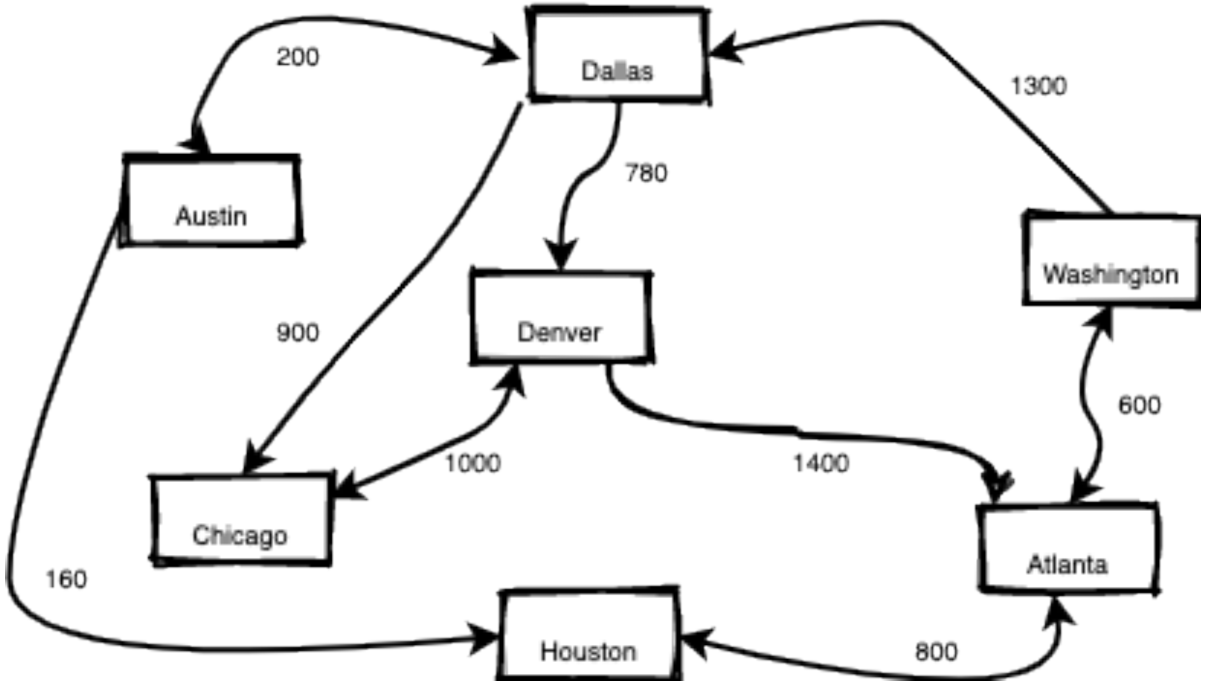
- **Depth-first search** follows a path as far as possible before backtracking
- It is useful in graphs for checking if a path exists between two nodes

Depth-First Search Algorithm

- Add the nodes adjacent to the start node to the stack
- Pop a node off the stack and examine it
- Add all the nodes adjacent to the popped node to the stack
- Continue until the target node is found or the stack is empty (no more nodes to check)

Depth-First Search Algorithm

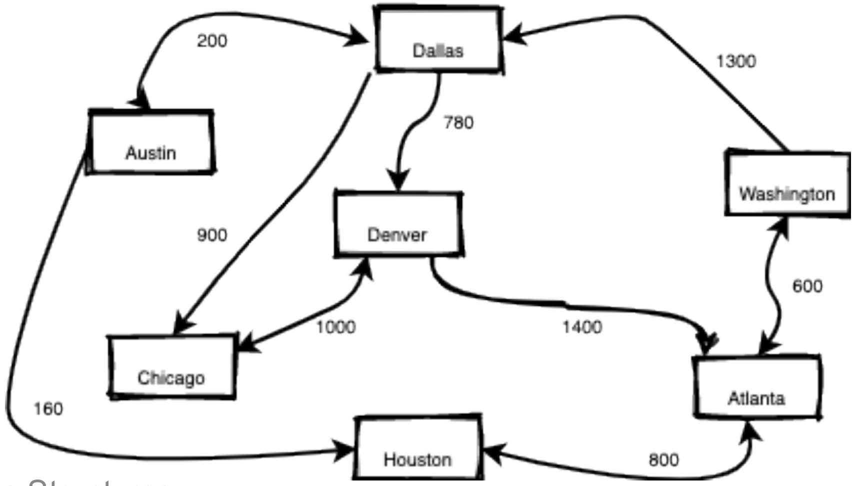
```
set found to false
stack.Push(startVertex)
do
    stack.Pop(vertex)
    if vertex == endVertex
        write endVertex
        set found to true
    else
        write vertex
        push all adjacent vertices onto stack
while !stack.IsEmpty() && !found
if(!found)
    write "Path does not exits"
```



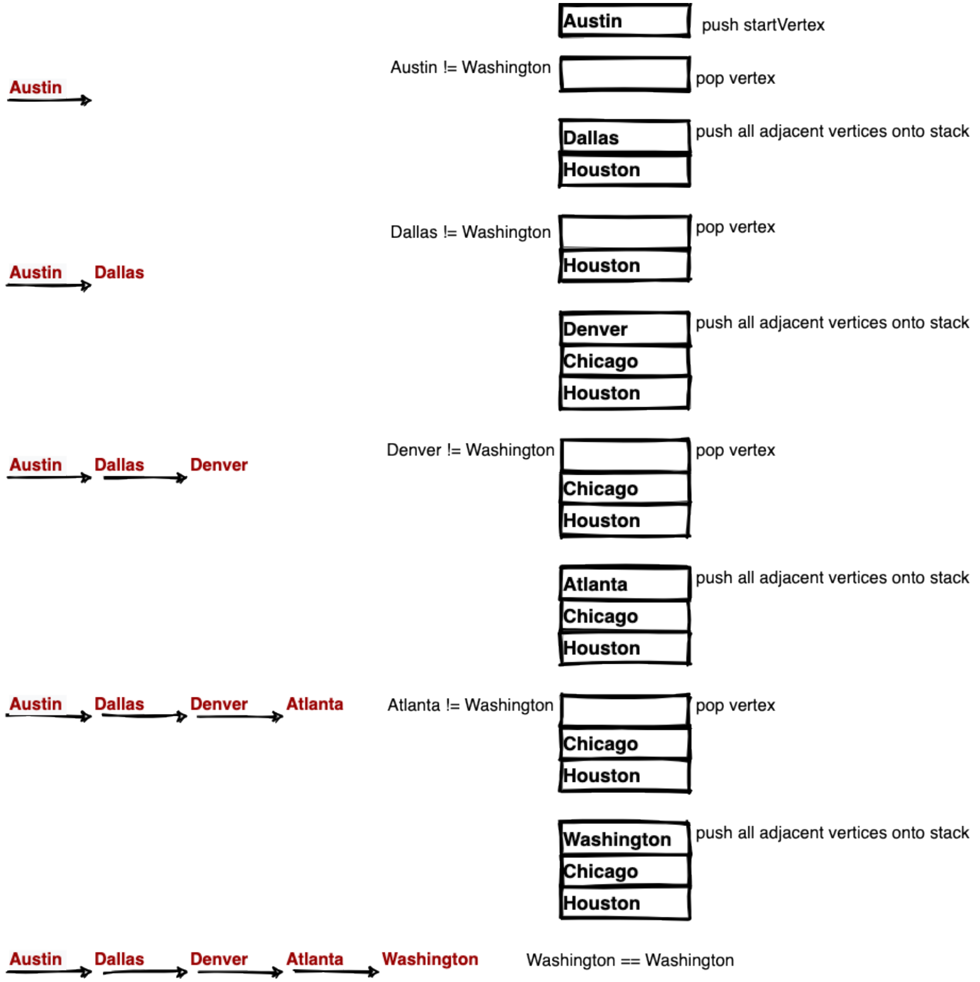
Is there a path from Austin to Washington?

Depth-First Search Algorithm

```
set found to false
stack.Push(startVertex)
do
    stack.Pop(vertex)
    if vertex == endVertex
        write endVertex
        set found to true
    else
        write vertex
        push all adjacent vertices onto stack
while !stack.IsEmpty() && !found
if(!found)
    write "Path does not exits"
```



Is there a path from Austin to Washington?

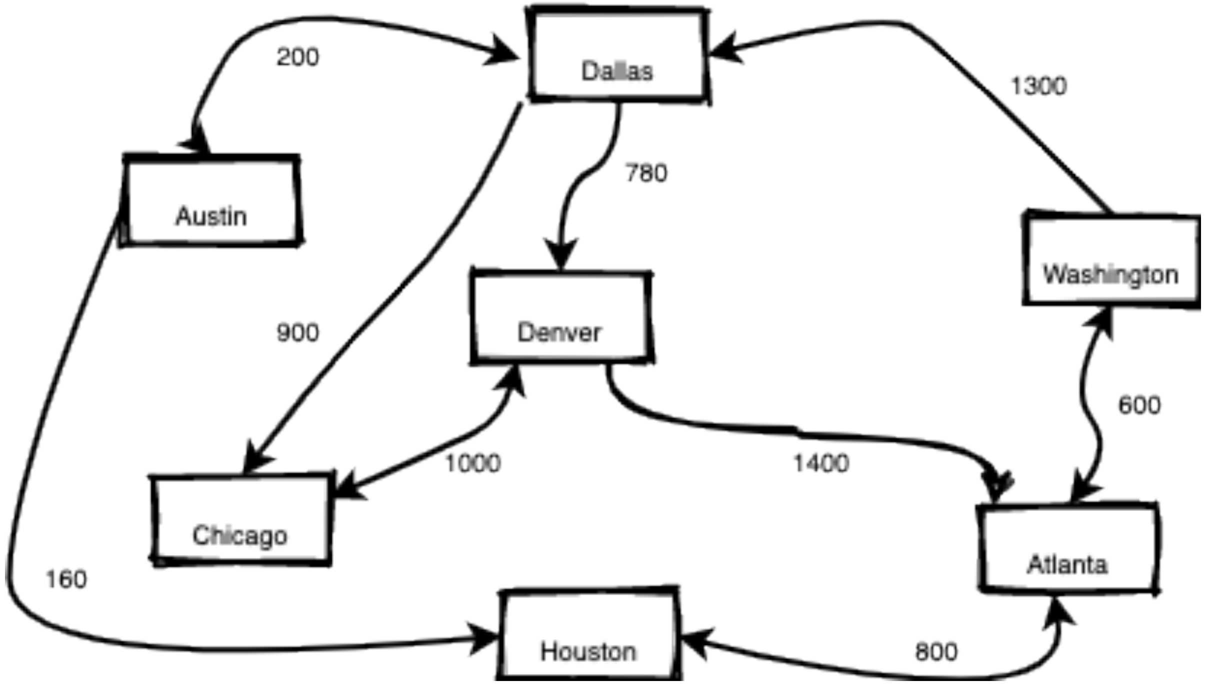


Breadth-First Search

- **Breadth-first search** examines all possible paths of the same length before going further
- DFS backtracks as little as possible, while BFS backtracks as far as possible
- In a binary tree, BFS would explore all the nodes at a particular level before exploring any nodes on the next level
- BFS uses a queue to keep track of nodes

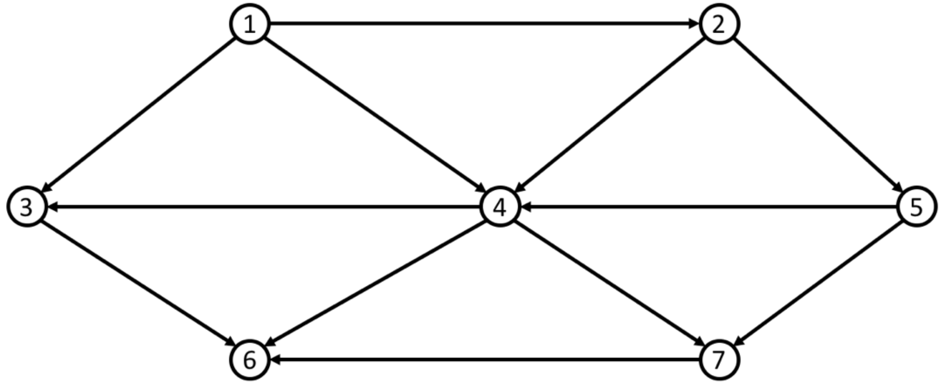
Breadth-First Search Algorithm

```
set found to false
queue.enqueue(startVertex)
do
    queue.dequeue(vertex)
    if vertex == endVertex
        write endVertex
        set found to true
    else
        write vertex
        enqueue all adjacent vertices into queue
while !queue.IsEmpty() && !found
if(!found)
    write "Path does not exits"
```



Is there a path from Austin to Washington?

Representing Graphs



v

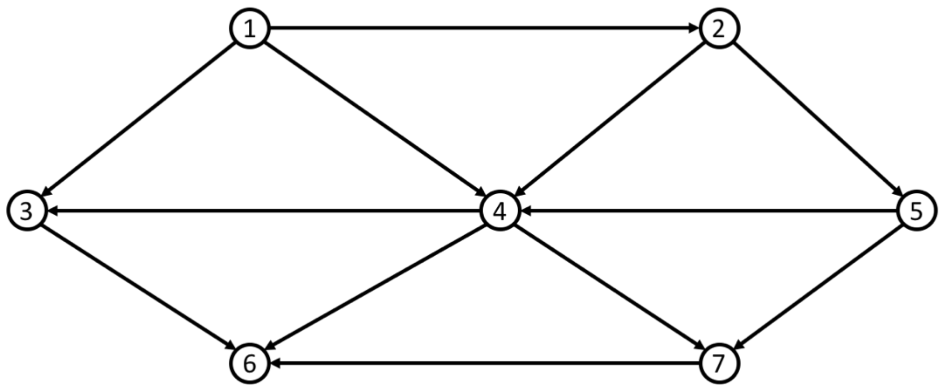
u

	1	2	3	4	5	6	7
1	T	T	T	T	F	F	F
2	F	T	F	T	T	F	F
3	F	F	T	F	F	T	F
4	F	F	T	T	F	T	T
5	F	F	F	T	T	F	T
6	F	F	F	F	F	T	F
7	F	F	F	F	F	T	T

A two-dimensional array (matrix) to represent a graph, $u \rightarrow v$.

Representing Graphs

A two-dimensional array (matrix) to represent a graph, $u \rightarrow v$.



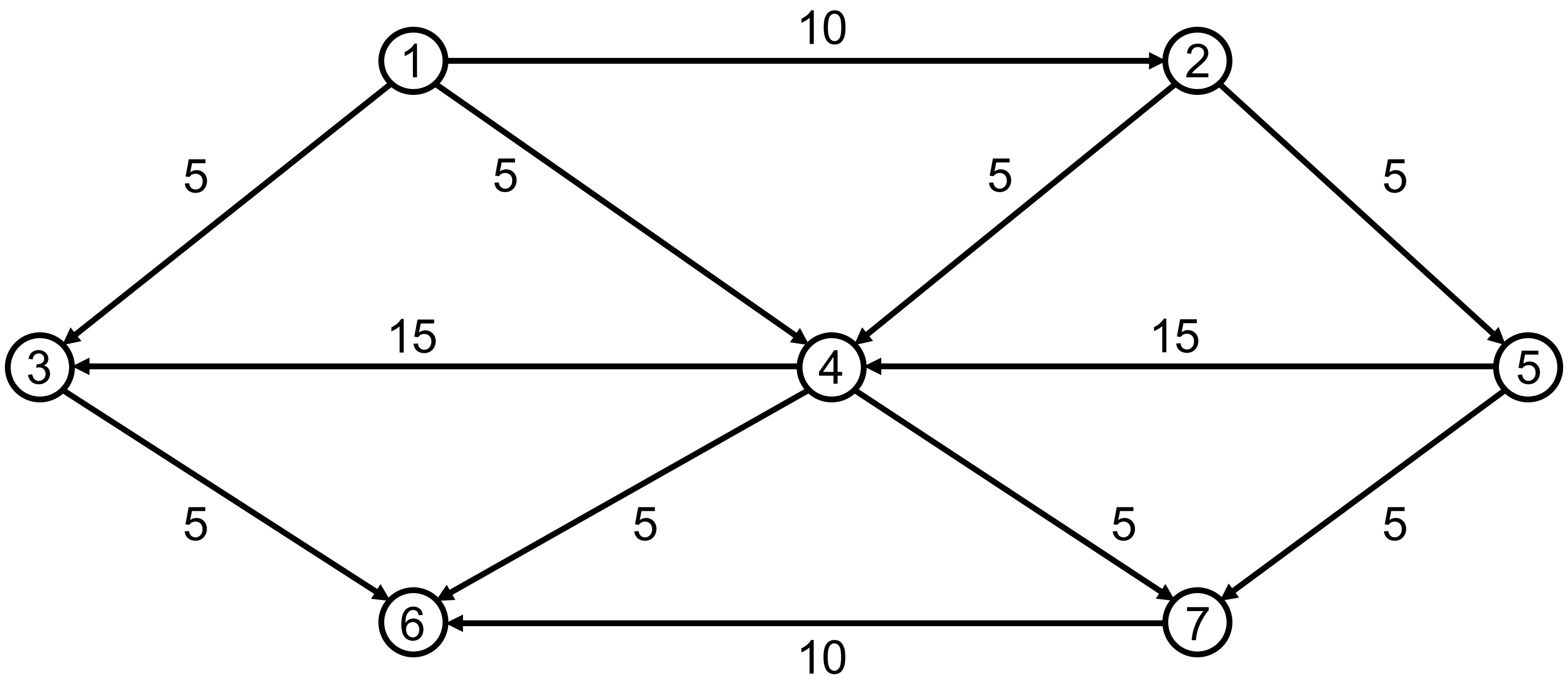
v

u

	1	2	3	4	5	6	7
1	T	T	T	T	F	F	F
2	F	T	F	T	T	F	F
3	F	F	T	F	F	T	F
4	F	F	T	T	F	T	T
5	F	F	F	T	T	F	T
6	F	F	F	F	F	T	F
7	F	F	F	F	F	T	T

Put a True(T) for edge and False(F) for no edge, it's a choice on how to handle self-connection. For this class assume every vertex is self connected so put a T.

Representing Graphs with Weights



Representing Graphs with Weights

- Same thing as before but instead of T/F we put in the weights, with special mark for no edges. Weight on self connection is 1.

Representing Graphs with Weights

- Same thing as before but instead of T/F we put in the weights, with special mark for no edges. Weight on self connection is 1.

v

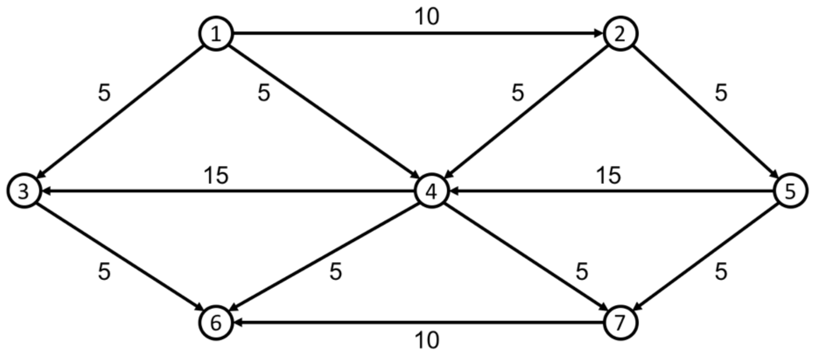
u

	1	2	3	4	5	6	7
1	1	10	5	5	∞	∞	∞
2	∞	1	∞	5	5	∞	∞
3	∞	∞	1	∞	∞	5	∞
4	∞	∞	15	1	∞	5	5
5	∞	∞	∞	15	1	∞	5
6	∞	∞	∞	∞	∞	1	∞
7	∞	∞	∞	∞	∞	10	1

Representing Graphs with Weights

$u \rightarrow v$

(u, v)



v

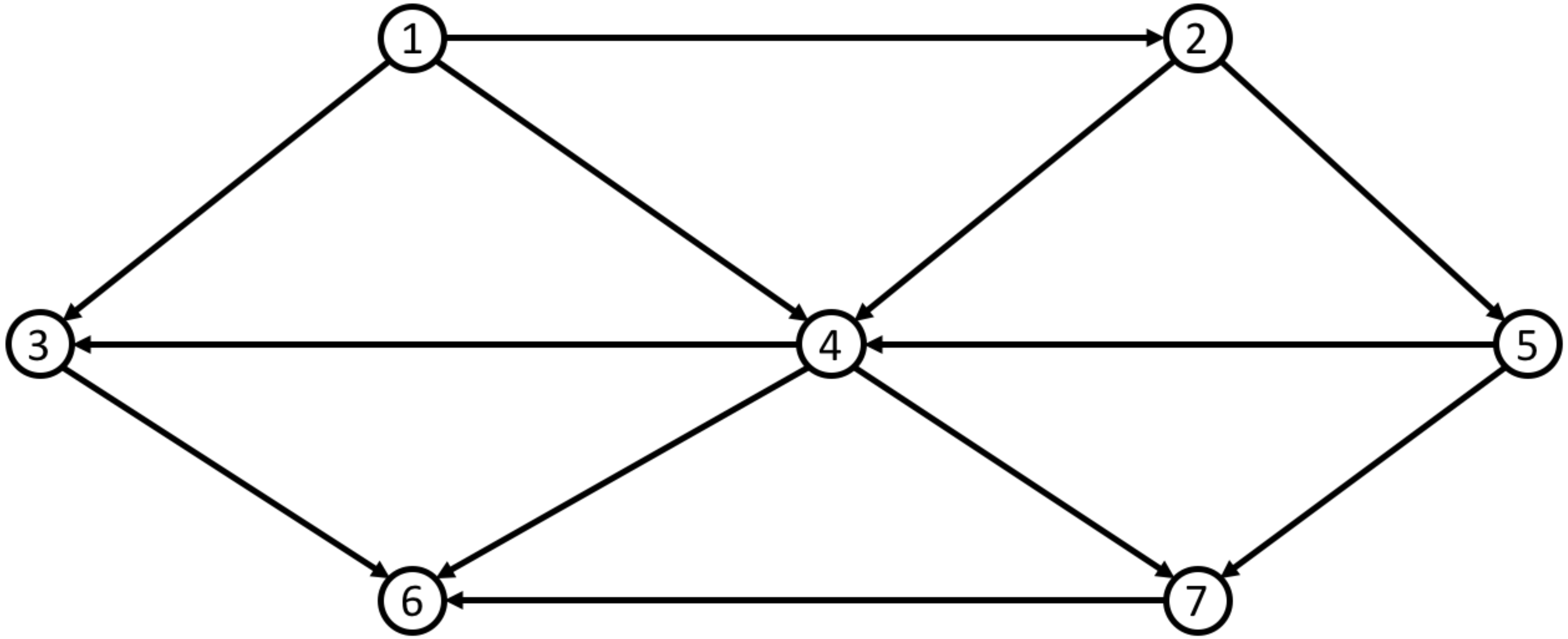
u

	1	2	3	4	5	6	7
1	1	10	5	5	∞	∞	∞
2	∞	1	∞	5	5	∞	∞
3	∞	∞	1	∞	∞	5	∞
4	∞	∞	15	1	∞	5	5
5	∞	∞	∞	15	1	∞	5
6	∞	∞	∞	∞	∞	1	∞
7	∞	∞	∞	∞	∞	10	1

Representing Graphs

Adjacency List

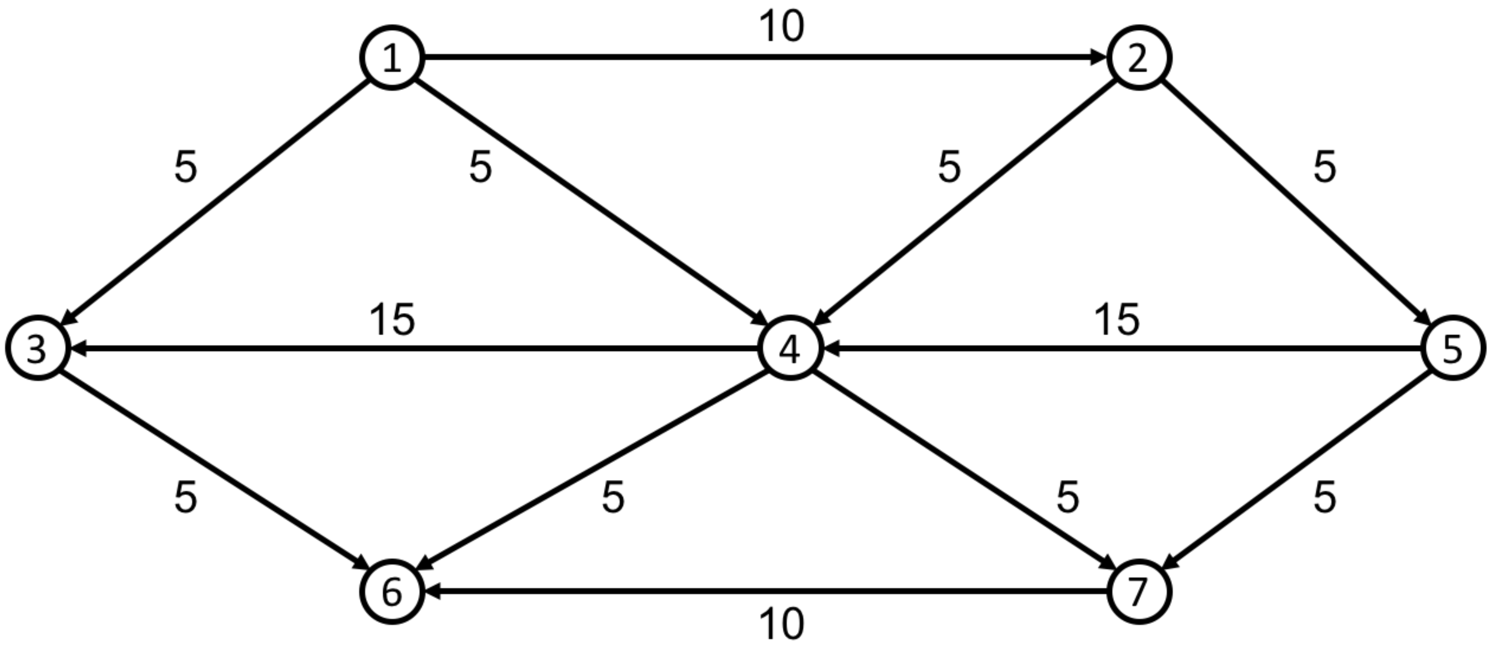
1	2, 3, 4
2	4, 5
3	6
4	3, 6, 7
5	4, 7
6	-
7	6



Representing Graphs with Weights

Adjacency List

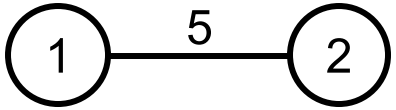
1	2[10], 3[5], 4[5]
2	4[5], 5[5]
3	6[5]
4	3[15], 6[5], 7[5]
5	4[15], 7[5]
6	-
7	6[10]



Definitions

- **weight** (cost) – optional third component to an edge, numerical value assigned as a label to the edge

Vertices: 1, 2 Edge: (1, 2) no weight



Vertices: 1, 2 Edge: (1, 2) weight of 5

- **path** – in a graph is a sequence of vertices $w_1, w_2, w_3, \dots, w_N$ such that $(w_i, w_{i+1}) \in E$ for $1 \leq i \leq N$

Vertices: w_1, w_2, w_3, w_4 Edges: (w_1, w_2), (w_2, w_3) with a path from w_1 to w_3, where $N = 3$

Definitions

- **Length**: is the number of edges on a path, it is equal to $N - 1$

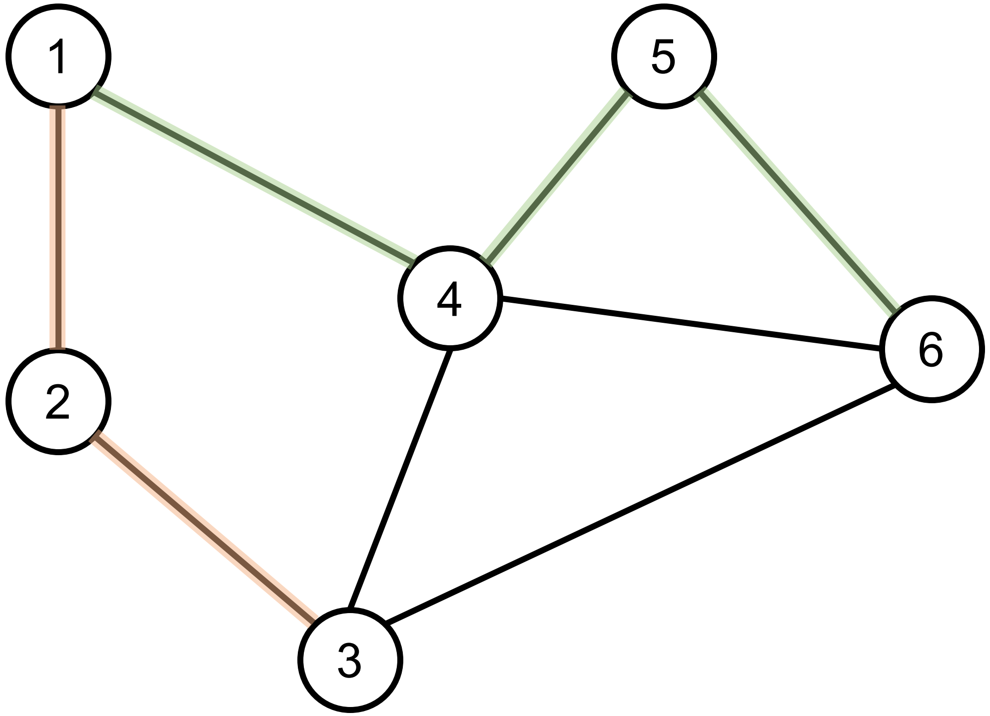
Vertices: w_1, w_2, w_3, w_4 Edges: (w_1, w_2), (w_2, w_3)
with a path from w_1 to w_3, where $N = 3$ and the length is 2
If a path contains no edges, then its path length is 0

- **Loop**: if there is an edge (v, v) from a vertex to itself then this path is known as a *loop*, we will consider graphs in general will be loopless
- **simple path:** is a *path* that all vertices are distinct, except the first and last could be the same

Definitions

- **cycle** – in a directed graph, a path with a length of at least 1, such that $w_1 = w_N$, the *cycle* is simple if the path is simple; in an undirected graph the edges must be distinct
- **acyclic** (DAG) – is a directed graph with no cycles
- **connected** – in an undirected graph, the graph is connected if there is a path from every vertex to every other vertex
- **strongly connected** – a **connected** directed graph is known as a *strongly connected* graph
- **weakly connected** – a graph is *weakly connected* if the directed graph is **connected** when direction of the edges is ignored
- **complete** – is a graph where there is an edge between every pair of vertices
- **Indegree** – the number of incoming edges in directed graph
- **Outdegree** – the number of outgoing edges in directed graph

Summary of Definitions



Undirected Graph

Vertices: 1, 2, 3, 4, 5, 6

Edges: [8] (1, 2), (1, 4), (2, 3),
(3, 4), (3, 6), (4, 5),
(4, 6), (5, 6)

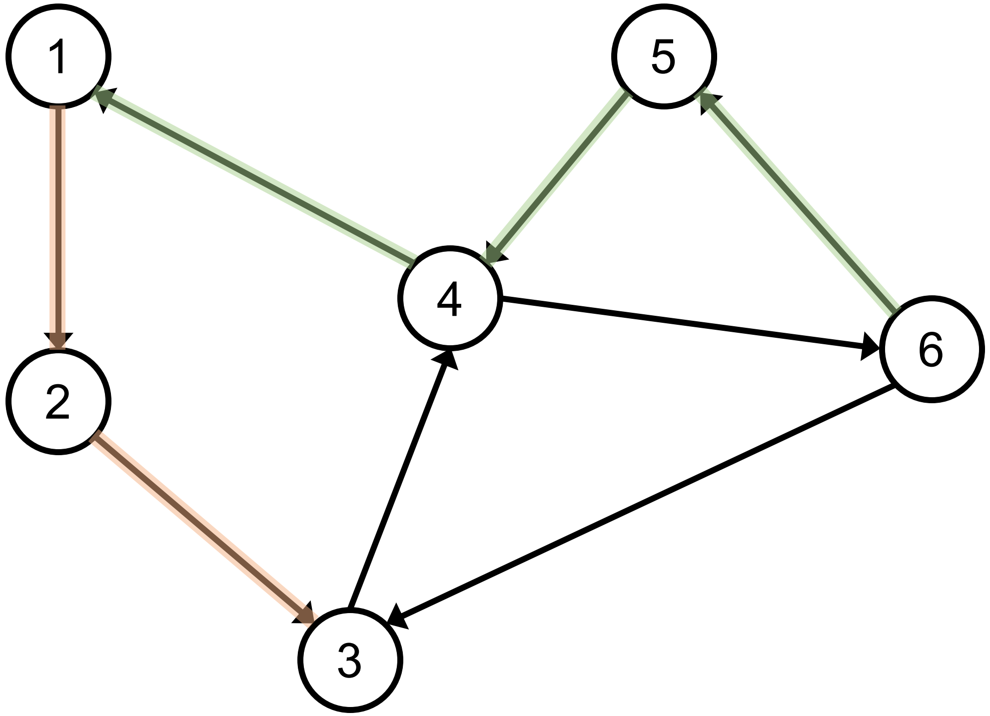
Adjacent: 1 is adjacent to 2 and 4

Path: 1 to 4 to 5 to 6 or 1 to 2 to 3

Length: 1 to 4 to 5 to 6 is 3 and 1 to 2 to 3 is 2

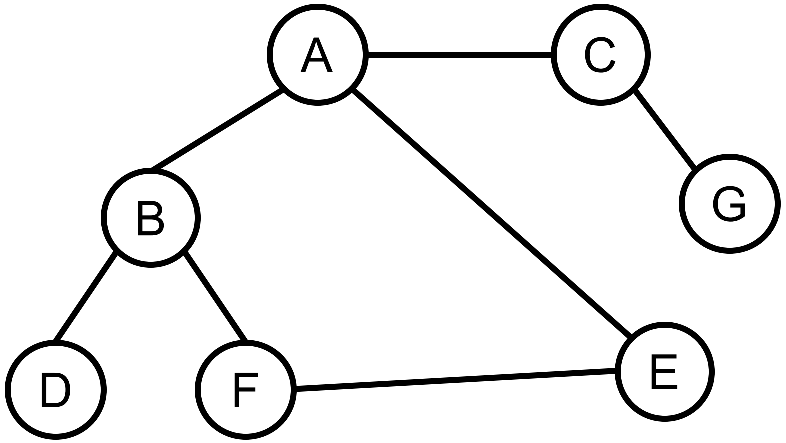
Is this graph connected? Yes

Summary of Definitions



Directed Graph
Vertices: 1, 2, 3, 4, 5, 6
Edges: [8] (1, 2), (2, 3), (3, 4),
(4, 1), (4, 6), (5, 4),
(6, 3), (6, 5)
Adjacent: 1 is adjacent to 2
Path: 6 to 5 to 4 to 1 or 1 to 2 to 3
Length: 6 to 5 to 4 to 1 is 3 and 1 to 2 to 3 is 2
Is this graph strongly connected? Yes

Depth-First Search



Starting at A

A, B

A, B, D *pop back to B*

A, B, D, F

A, B, D, F, E *has path to A* pop back to F, then to B, then to A looking for path to a vertex that has not be visited*

A, B, D, F, E, C

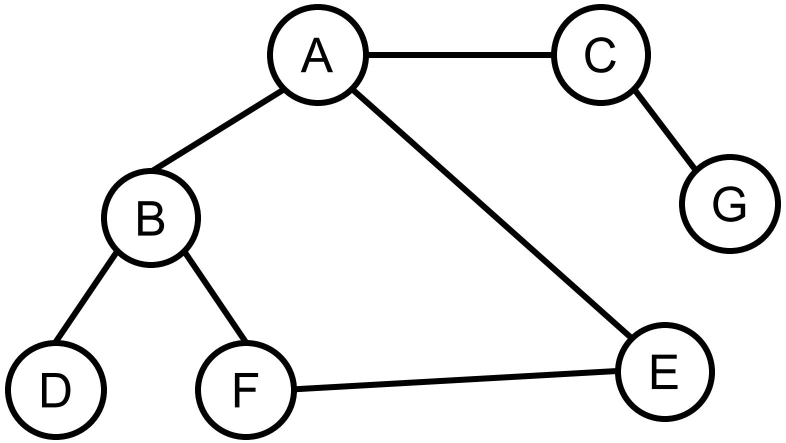
A, B, D, F, E, C, G *Finished*

*Coding wise this why its important to keep a list of vertices visited

Depth-First Search for Cycle Detection

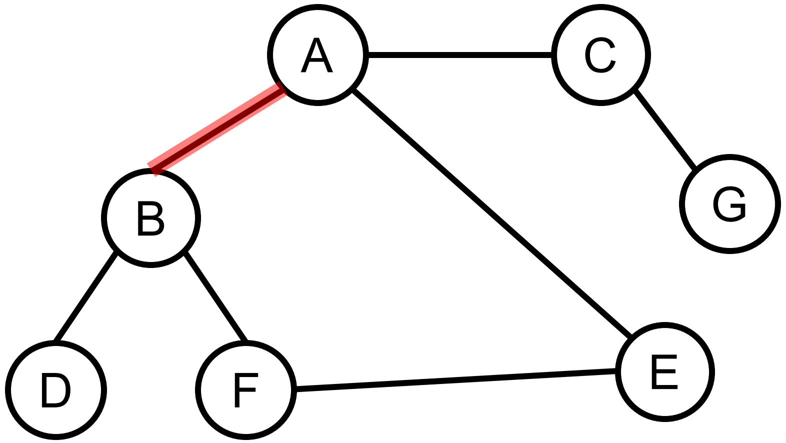
We can use DFS to detect cycle

Starting at A



List of visited vertices: A

Depth-First Search for Cycle Detection



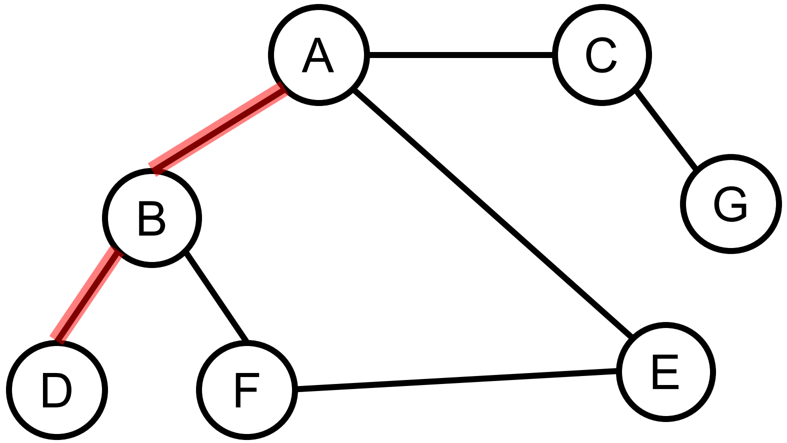
We can use DFS to detect cycle

Starting at A

Visit B

List of visited vertices: A, B

Depth-First Search for Cycle Detection



We can use DFS to detect cycle

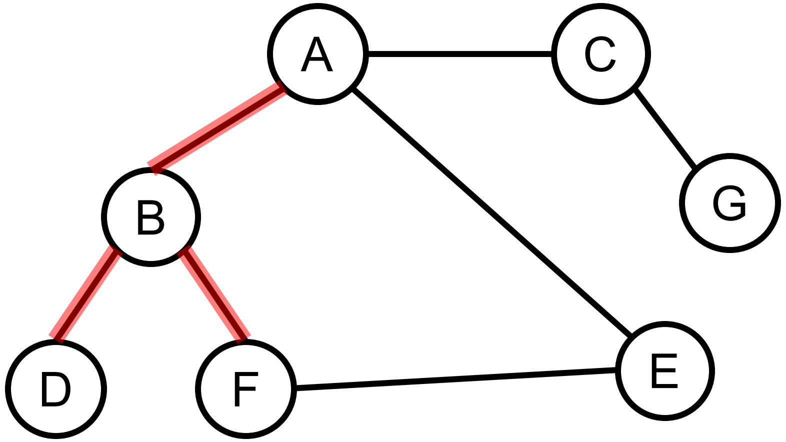
Starting at A

Visit B

Visit D – nothing is adjacent to D so pop back to B

List of visited vertices: A, B, D

Depth-First Search for Cycle Detection



We can use DFS to detect cycle

Starting at A

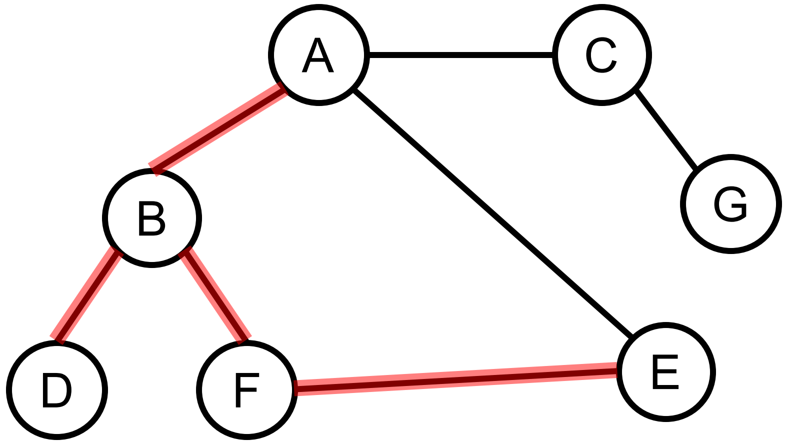
Visit B

Visit D – nothing is adjacent to D so pop back to B

Visit F

List of visited vertices: A, B, D, F

Depth-First Search for Cycle Detection



We can use DFS to detect cycle

Starting at A

Visit B

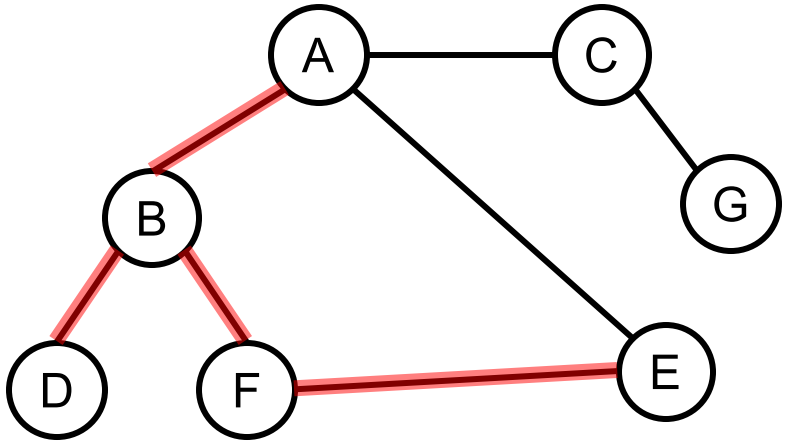
Visit D – nothing is adjacent to D so pop back to B

Visit F

Visit E

List of visited vertices: A, B, D, F, E

Depth-First Search for Cycle Detection



List of visited vertices: A, B, D, F, E

We can use DFS to detect cycle

Starting at A

Visit B

Visit D – nothing is adjacent to D so pop back to B

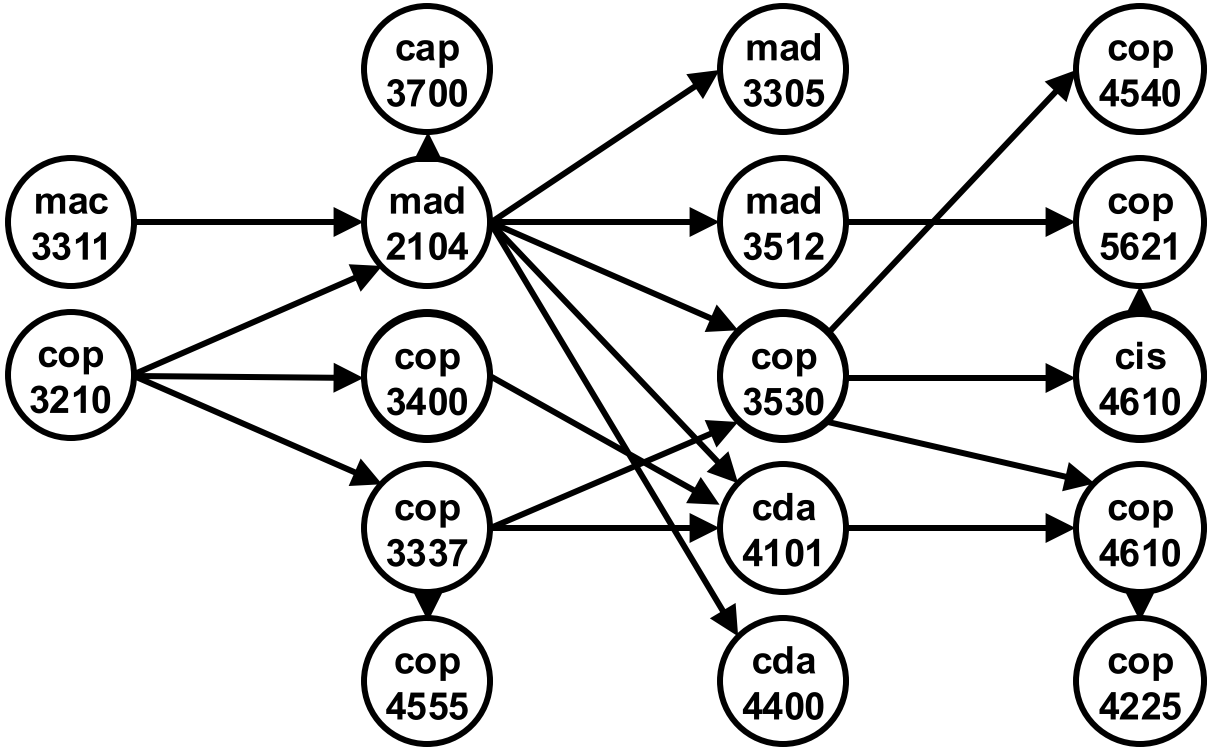
Visit F

Visit E

Visit A – but A is visited list, this means there is an alternative path from A to E in addition to the path E to A and a **cycle is found**!!!!

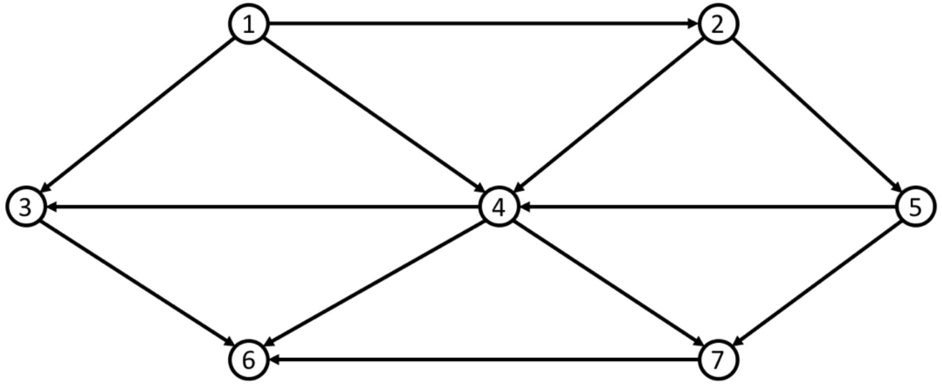
Topological Sort

- Topological sort: is the 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 v_i in the ordering
- Reminder: acyclic (DAG) – is a directed graph with no cycles



Topological Sort

- The 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 v_i in the ordering



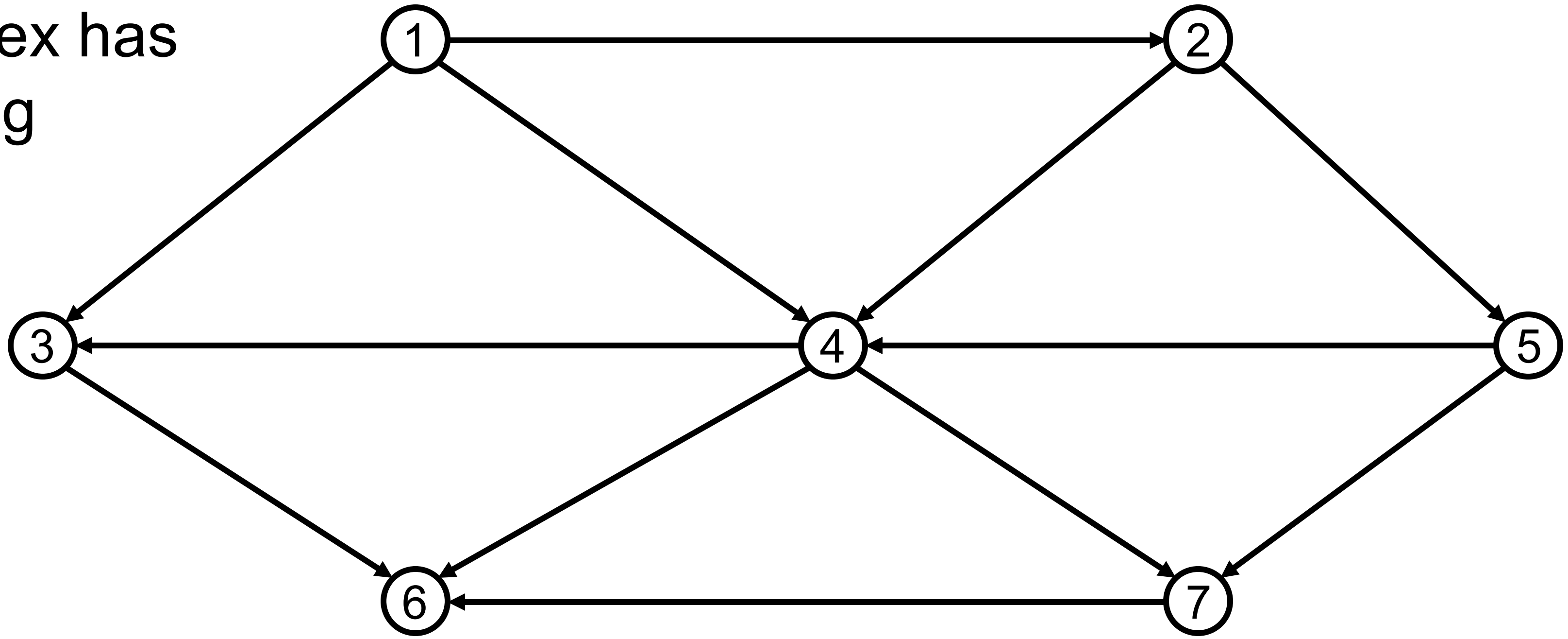
- A **single** topological ordering is not possible if the graph has a cycle, as for the two vertices v and w on the cycle, v precedes w and w precedes v
- Valid topological ordering:

$v_1, v_2, v_5, v_4, v_3, v_7, v_6$ or $v_1, v_2, v_5, v_4, v_7, v_3, v_6$

Topological Sort

- A simple algorithm to find a topological ordering
  - find any vertex with no incoming edges
  - remove it and its edges

Which vertex has no incoming edges?

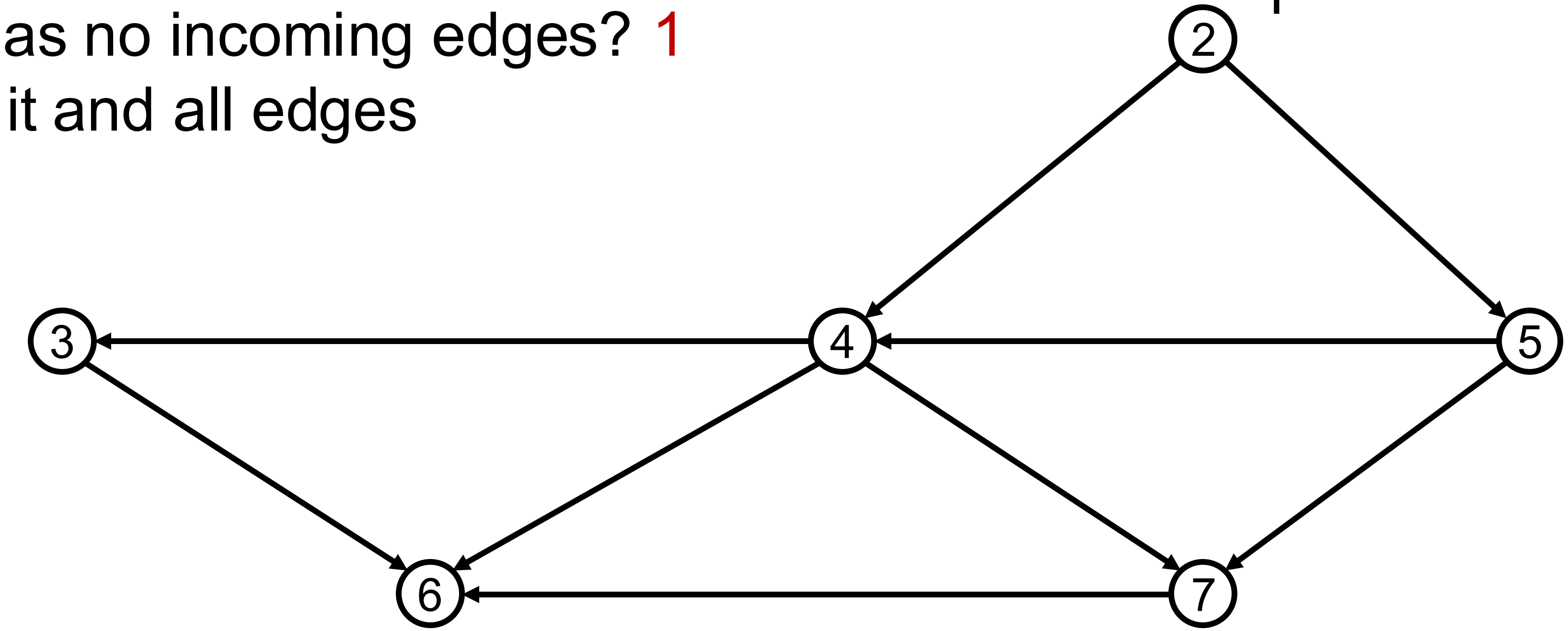


Topological Sort

- A simple algorithm to find a topological ordering
  - find any vertex with no incoming edges
  - remove it and its edges

Topological Ordering:
1

Which vertex has no incoming edges? 1
So we remove it and all edges

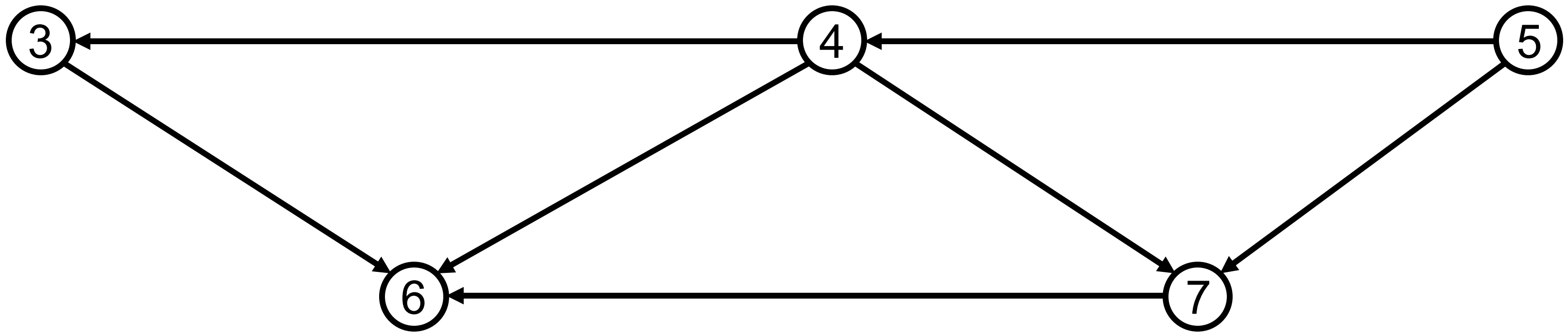


Topological Sort

- A simple algorithm to find a topological ordering
  - find any vertex with no incoming edges
  - remove it and its edges

Topological Ordering:
1, 2

Which vertex has no incoming edges? 2
So we remove it and all edges

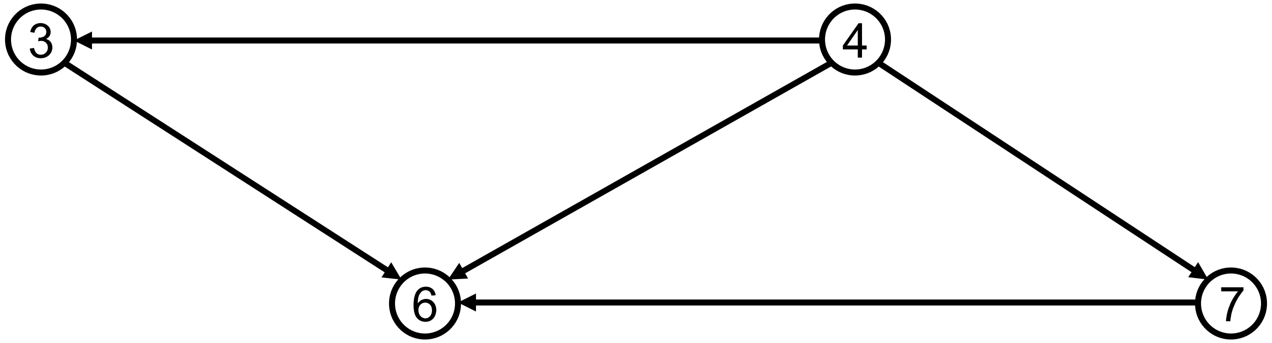


Topological Sort

- A simple algorithm to find a topological ordering
  - find any vertex with no incoming edges
  - remove it and its edges

Topological Ordering:
1, 2, 5

Which vertex has no incoming edges? 5
So we remove it and all edges

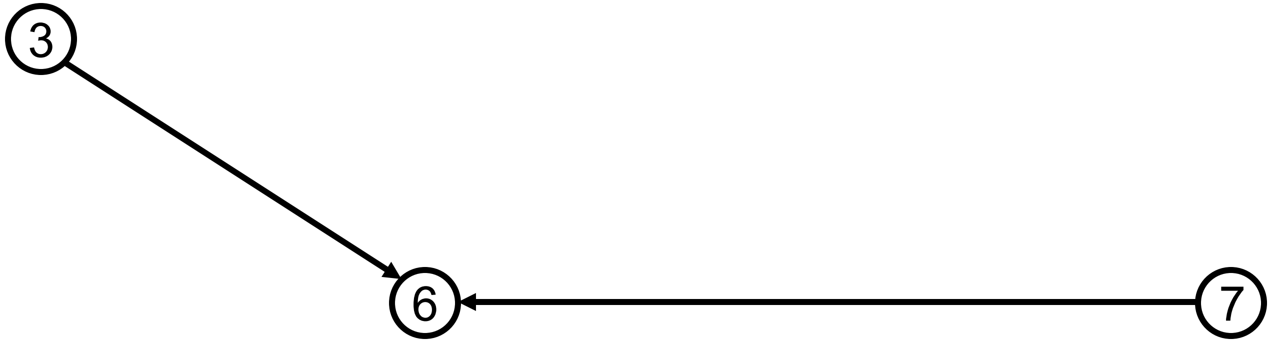


Topological Sort

- A simple algorithm to find a topological ordering
  - find any vertex with no incoming edges
  - remove it and its edges

Which vertex has no incoming edges? 4
So we remove it and all edges

Topological Ordering:
1, 2, 5, 4



Topological Sort

- A simple algorithm to find a topological ordering
  - find any vertex with no incoming edges
  - remove it and its edges

Topological Ordering:
1, 2, 5, 4, {3, 7}

Which vertex has no incoming edges? 3 or 7
So we remove it and all edges

Topological Sort

- A simple algorithm to find a topological ordering
  - find any vertex with no incoming edges
  - remove it and its edges

Which vertex has no incoming edges? 3 or 7
So we remove it and all edges

Topological Ordering:
1, 2, 5, 4, {3, 7}

6

Topological Sort

- A simple algorithm to find a topological ordering
  - find any vertex with no incoming edges
  - remove it and its edges

Which vertex has no incoming edges? 6
So we remove it and all edges

Topological Ordering:
1, 2, 5, 4, {3, 7}, 6

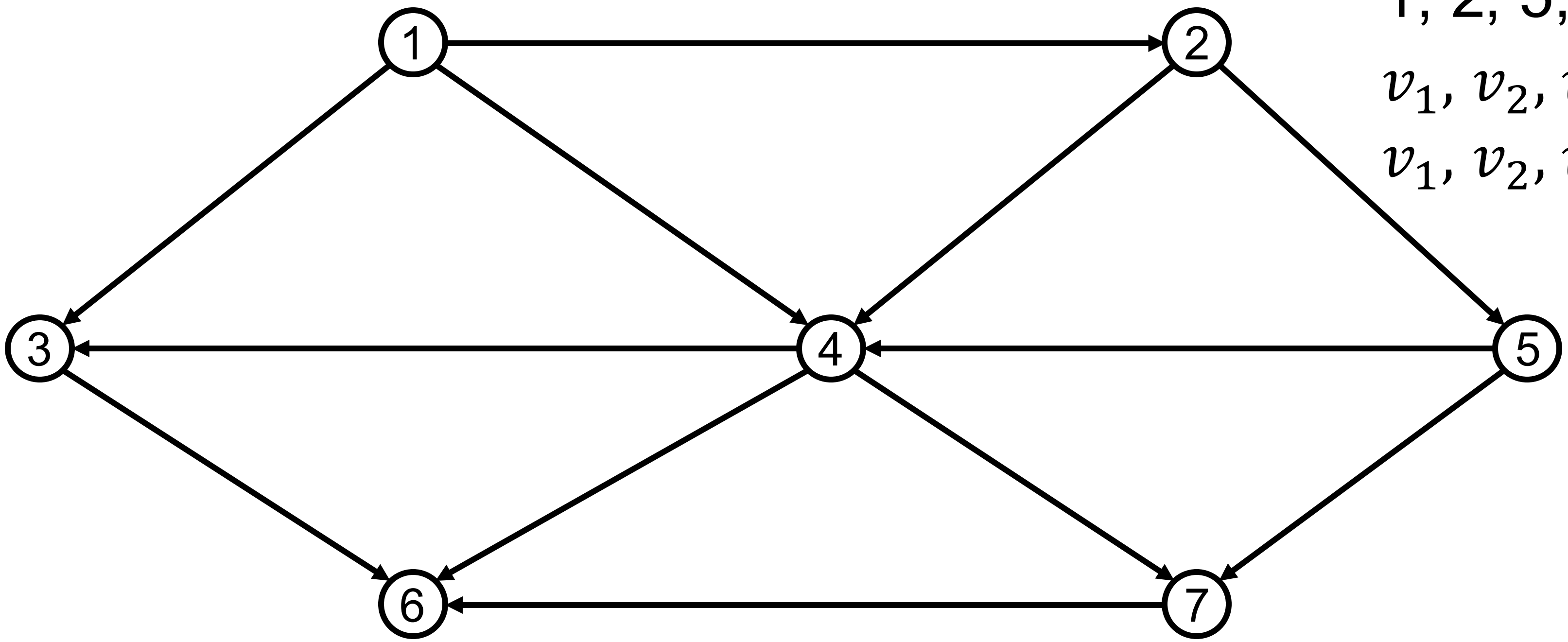
Topological Sort

- A simple algorithm to find a topological ordering
  - find any vertex with no incoming edges
  - remove it and its edges

Topological Ordering:
1, 2, 5, 4, {3, 7}, 6

$v_1, v_2, v_5, v_4, v_3, v_7, v_6$

$v_1, v_2, v_5, v_4, v_7, v_3, v_6$

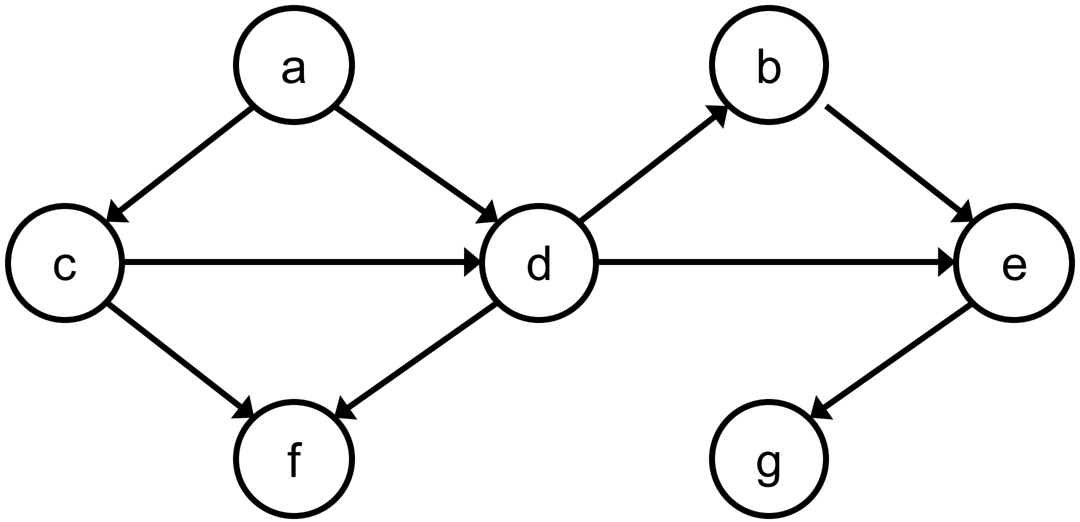


Topological Sort

- A simple algorithm to find a topological ordering
  - find any vertex with no incoming edges
  - remove it and its edges

Topological Ordering:

Which vertex has no incoming edges?
So we remove it and all edges

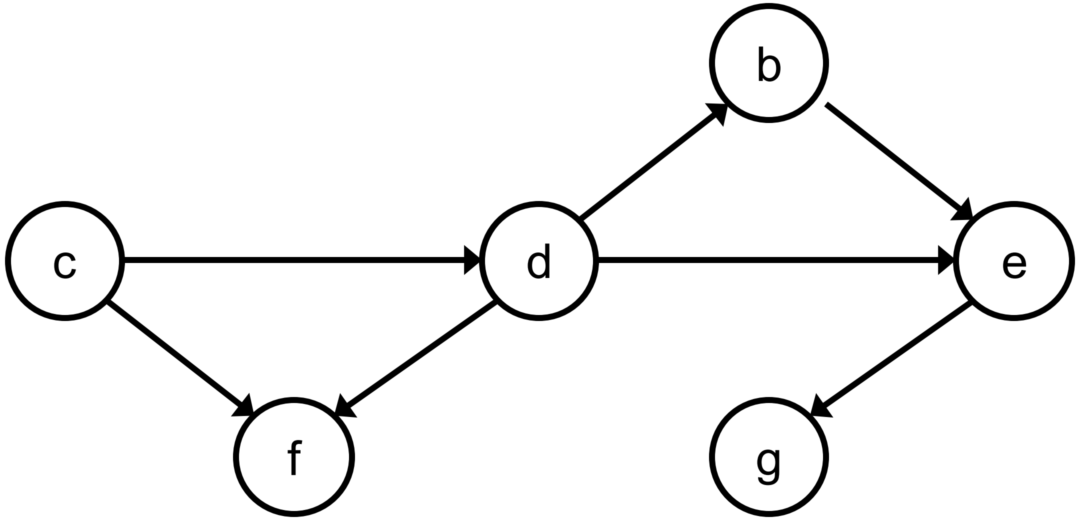


Topological Sort

- A simple algorithm to find a topological ordering
  - find any vertex with no incoming edges
  - remove it and its edges

Topological Ordering:
a

Which vertex has no incoming edges? **a**
So we remove it and all edges

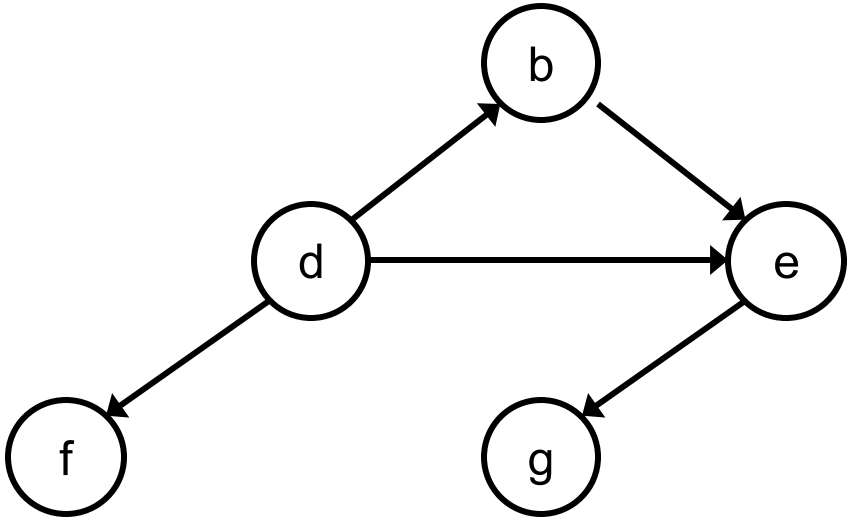


Topological Sort

- A simple algorithm to find a topological ordering
  - find any vertex with no incoming edges
  - remove it and its edges

Topological Ordering:
a, c

Which vertex has no incoming edges? **c**
So we remove it and all edges

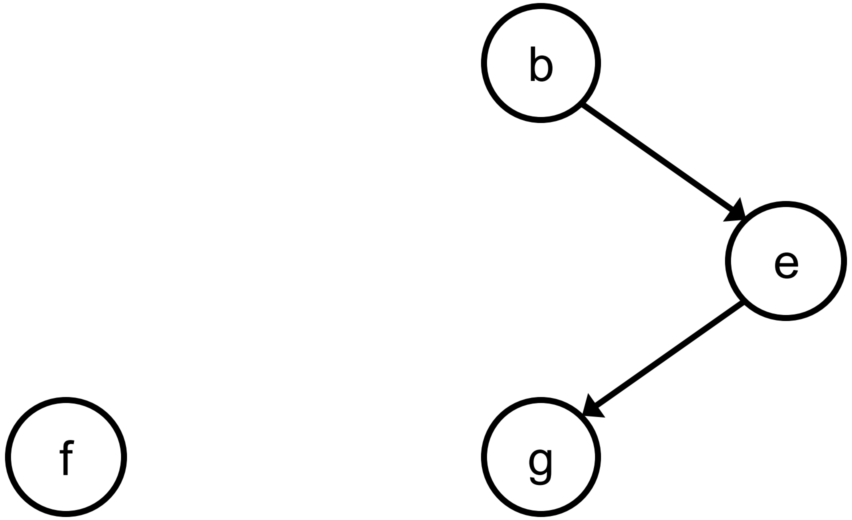


Topological Sort

- A simple algorithm to find a topological ordering
  - find any vertex with no incoming edges
  - remove it and its edges

Topological Ordering:
a, c, d

Which vertex has no incoming edges? **d**
So we remove it and all edges

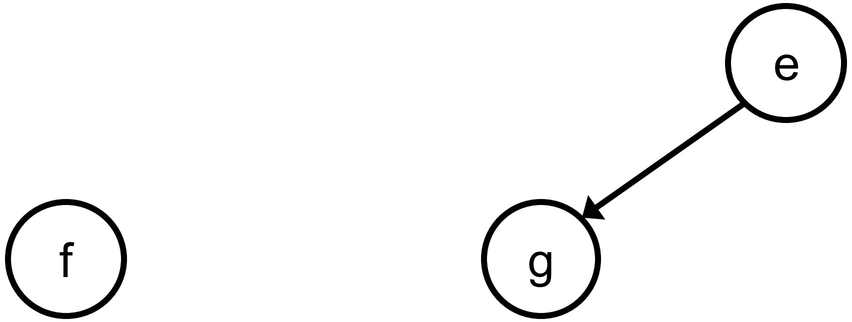


Topological Sort

- A simple algorithm to find a topological ordering
  - find any vertex with no incoming edges
  - remove it and its edges

Topological Ordering:
a, c, d, {b, f}

Which vertex has no incoming edges? **b or f**
So we remove it
and all edges

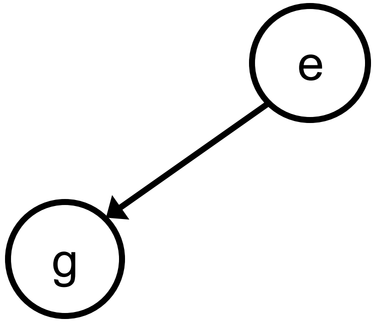


Topological Sort

- A simple algorithm to find a topological ordering
  - find any vertex with no incoming edges
  - remove it and its edges

Topological Ordering:
a, c, d, {b, f}

Which vertex has no
incoming edges? **b or f**
So we remove it
and all edges

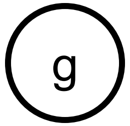


Topological Sort

- A simple algorithm to find a topological ordering
  - find any vertex with no incoming edges
  - remove it and its edges

Topological Ordering:
a, c, d, {b, f}, e

Which vertex has no incoming edges? **e**
So we remove it and all edges



Topological Sort

- A simple algorithm to find a topological ordering
  - find any vertex with no incoming edges
  - remove it and its edges

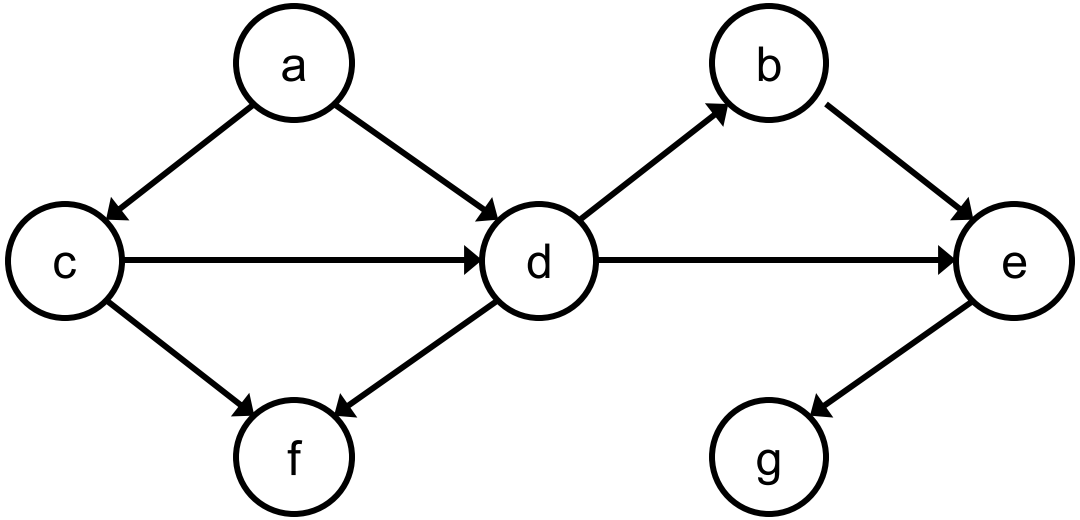
Topological Ordering:
a, c, d, {b, f}, e, g

Which vertex has no incoming edges? **g**
So we remove it
and all edges

Topological Sort

- A simple algorithm to find a topological ordering
  - find any vertex with no incoming edges
  - remove it and its edges

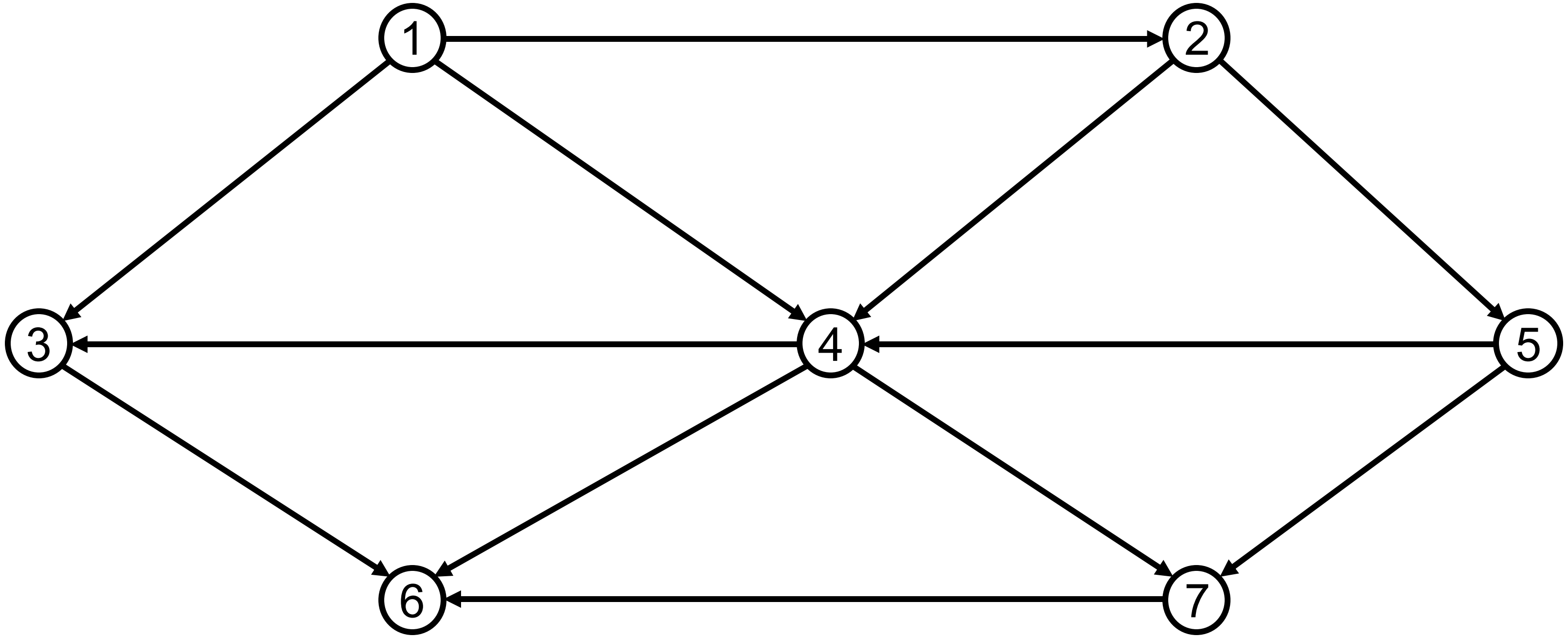
Topological Ordering:
a, c, d, {b, f}, e, g



Topological Sort

- Formally, we use our definition of **indegree** of a vertex v as the number of edges (u, v). Compute the indegree of all vertices in the graph and keep in adjacency list to generate a topological order

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



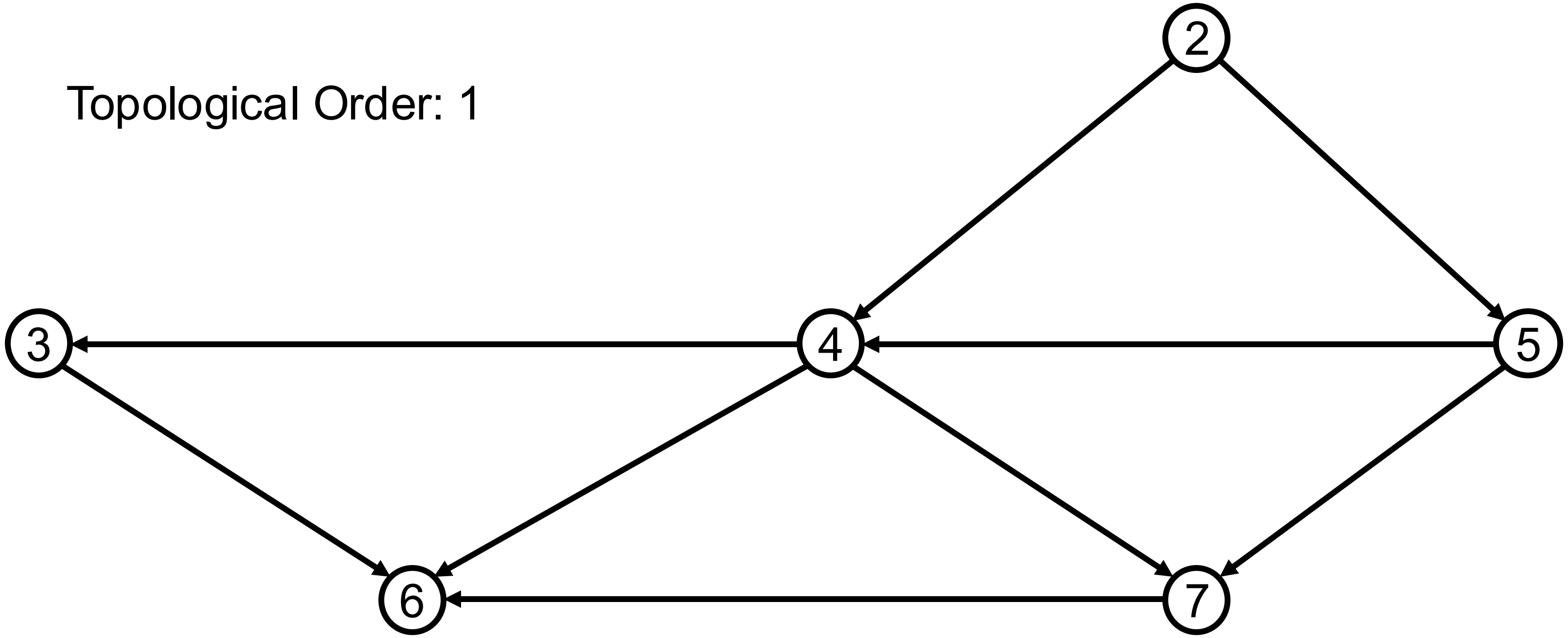
Pick one with Zero (0)

Topological Sort

- Formally, we use our definition of **indegree** of a vertex v as the number of edges (u, v). Compute the indegree of all vertices in the graph and keep in adjacency list to generate a topological order

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

Topological Order: 1



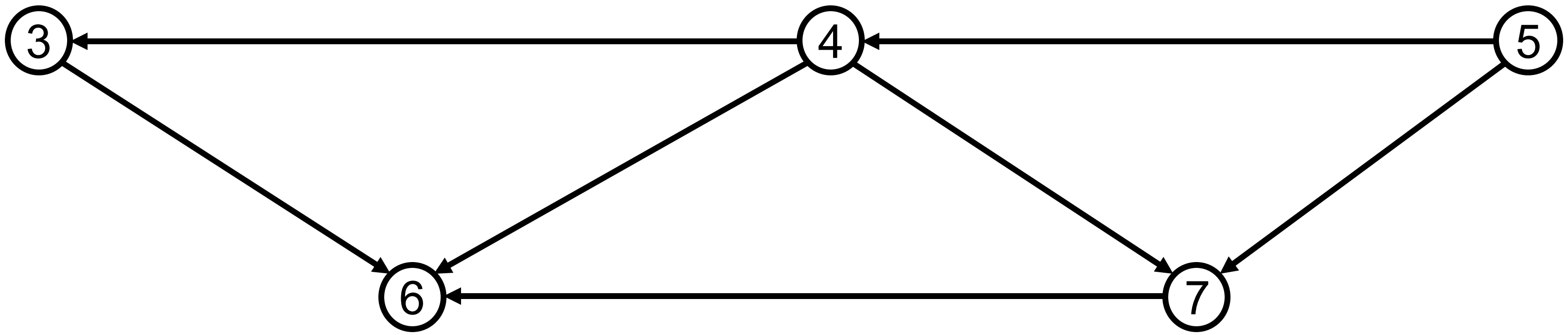
Pick one with Zero (0)

Topological Sort

- Formally, we use our definition of **indegree** of a vertex v as the number of edges (u, v). Compute the indegree of all vertices in the graph and keep in adjacency list to generate a topological order

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

Topological Order: 1, 2



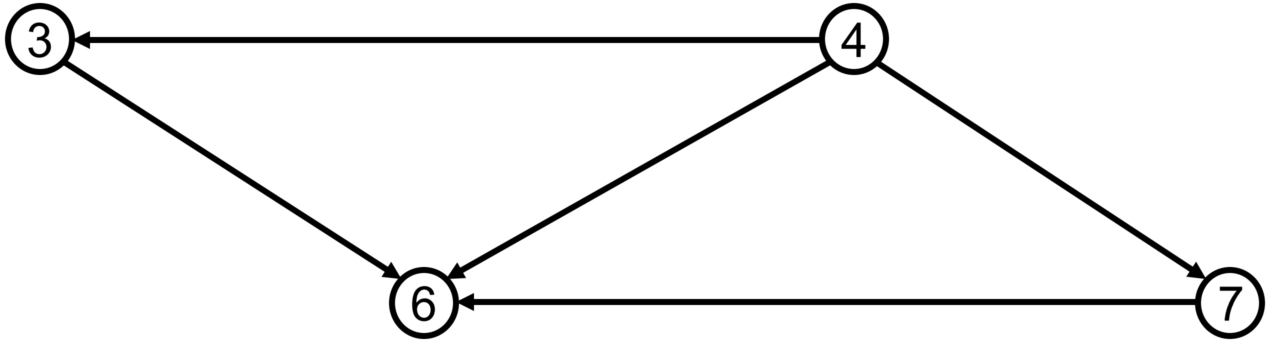
Pick one with Zero (0)

Topological Sort

- Formally, we use our definition of **indegree** of a vertex v as the number of edges (u, v). Compute the indegree of all vertices in the graph and keep in adjacency list to generate a topological order

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

Topological Order: 1, 2, 5



Pick one with Zero (0)

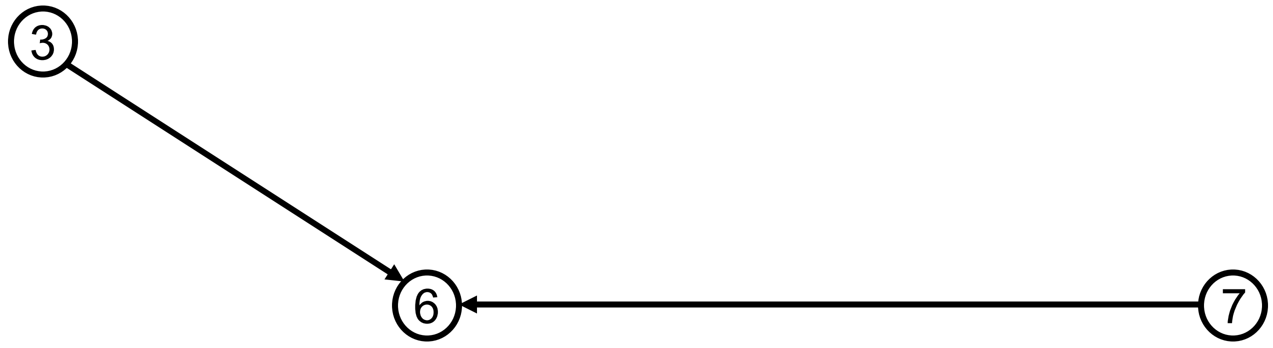
Topological Sort

- Formally, we use our definition of **indegree** of a vertex v as the number of edges (u, v). Compute the indegree of all vertices in the graph and keep in adjacency list to generate a topological order

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

Pick one with Zero (0)

Topological Order: 1, 2, 5, 4



Topological Sort

- Formally, we use our definition of **indegree** of a vertex v as the number of edges (u, v). Compute the indegree of all vertices in the graph and keep in adjacency list to generate a topological order

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

Topological Order: 1, 2, 5, 4, {3, 7}

6

Pick one with Zero (0)

Acknowledgement

These slides have been adapted and borrowed from books on the right as well as the CS340 notes of NIU CS department (Professors: Alhoori, Hou, Lehuta, and Winans) and many google searches.