WASHINGTON STATE UNIVERSITY

CPTS 223 Advanced Data Structure C/C++

Graph

Overview

- Definitions
- Terminology
- Graph representation
- Topological sort
- Shortest-path algorithms
  - Unweighted shortest path
  - Weighted shortest path: Dijkstra’s algorithm

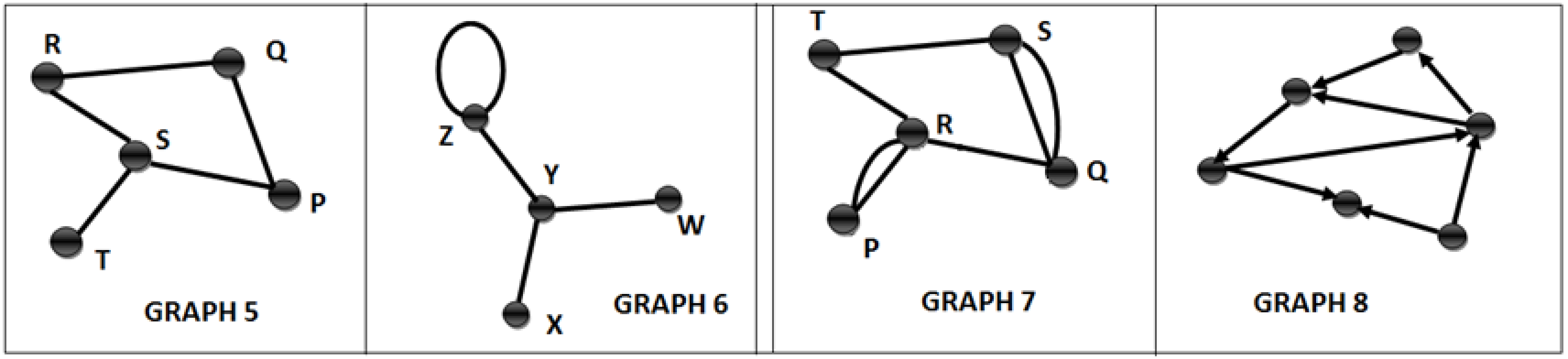
Single-source shortest path problem

Graph and graph algorithms

- What is a graph?
  - A graph G = (V, E) consists of a set of vertices V, and a set of edges E.
- Each edge is a pair (v, w), where v, w ∈ V

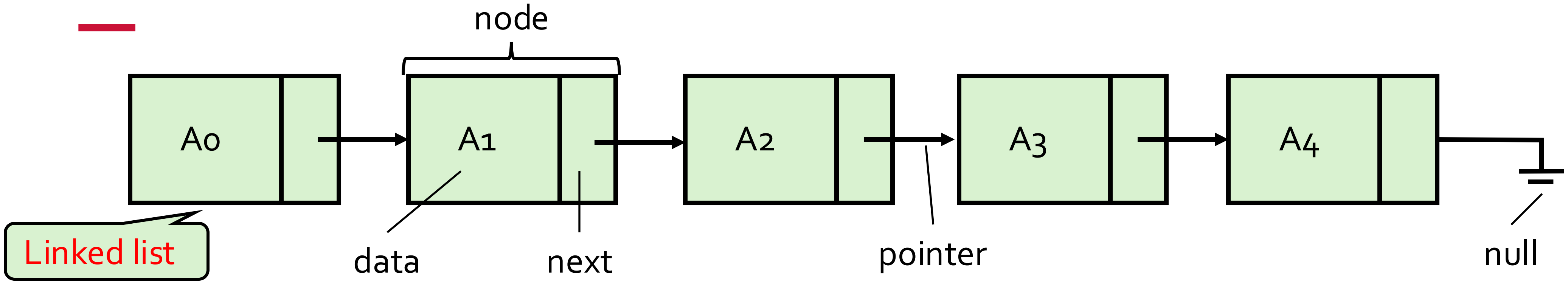
Both v and w are vertices

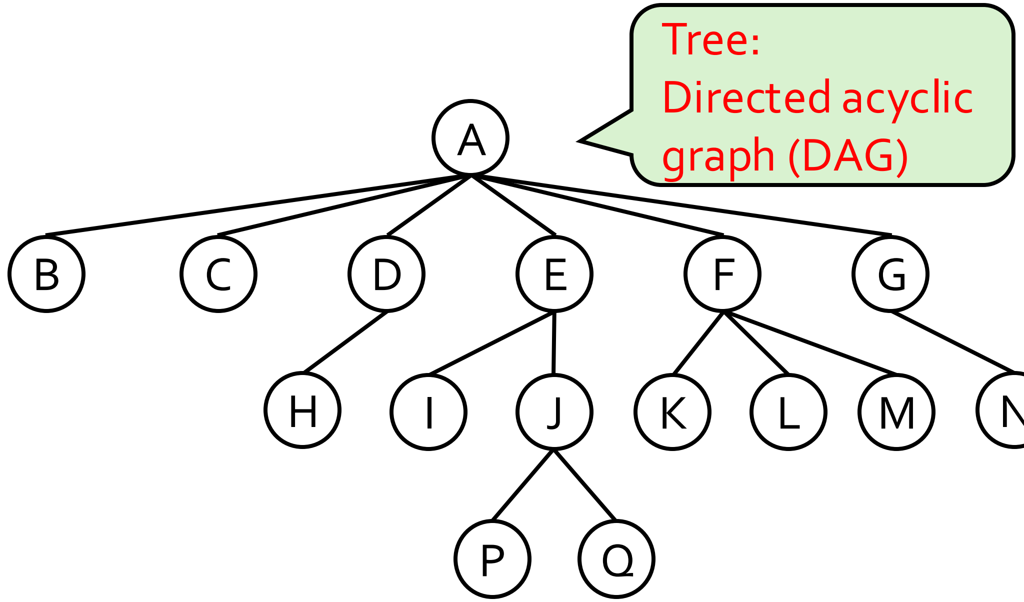
- Graph algorithm?
  - An algorithm designed to work with data kept in a graph structure

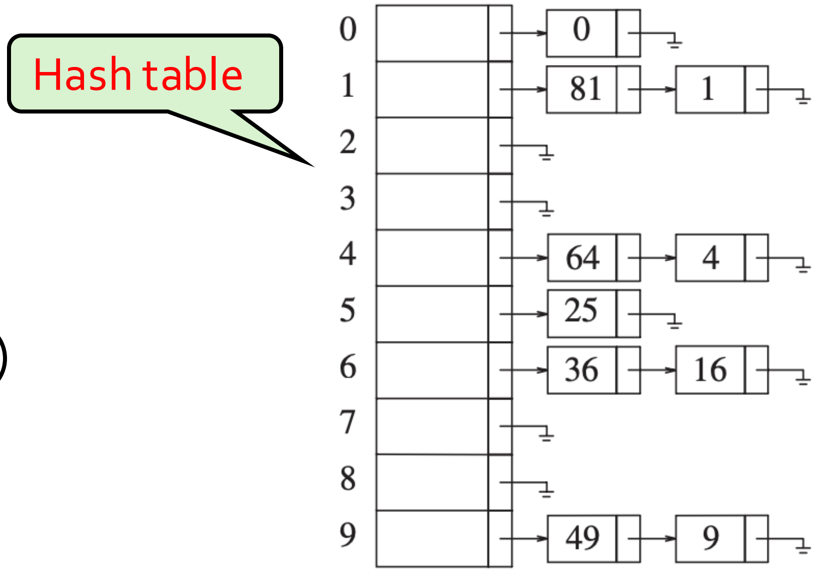


WSU

Where have we seen graphs?







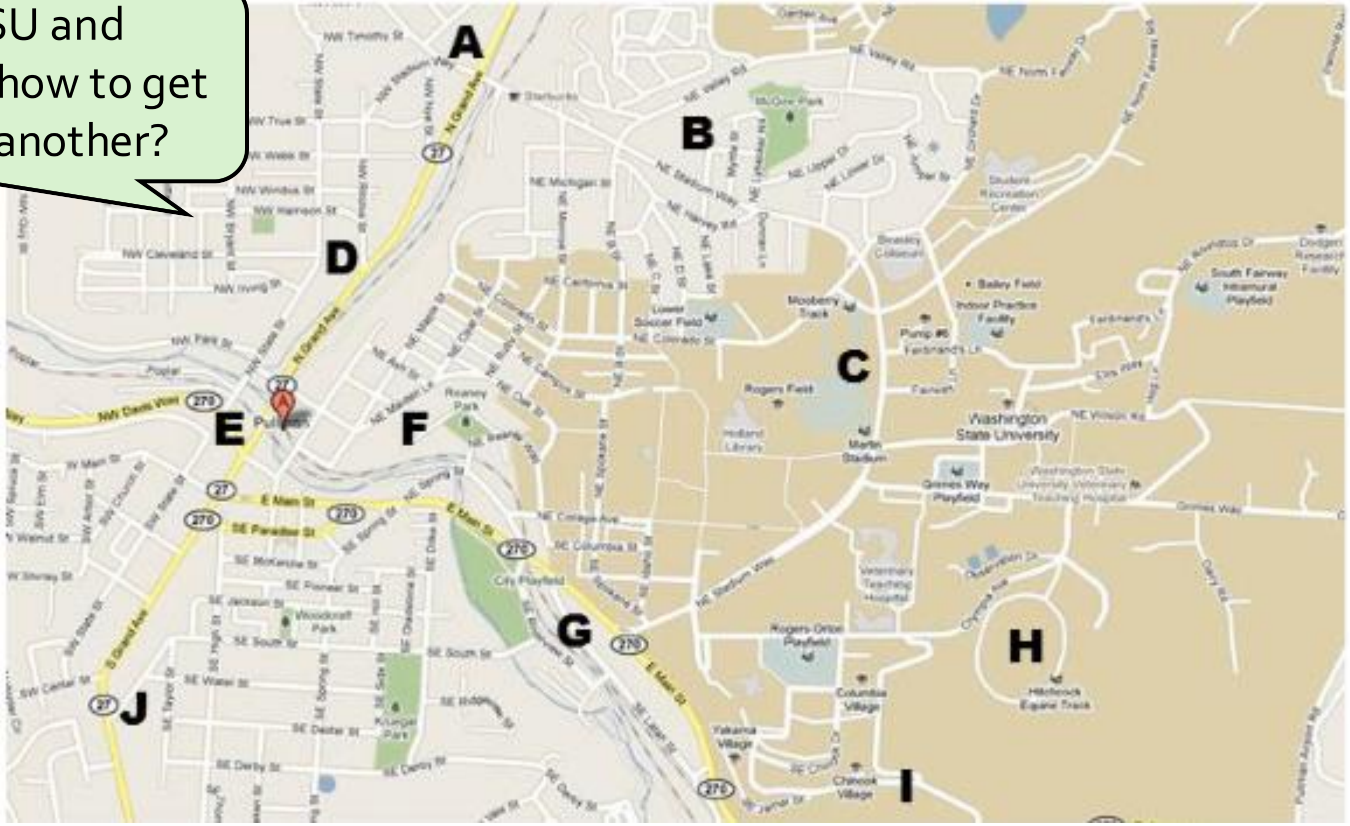
Big-O for graphs

- In trees, heaps, hashing, stacks
  - Number of updates or operations to complete algorithm
  - Push == add node to head of list
  - Insert (tree) == traversal to bottom of tree, then node create & update
- In sorting:
  - Number of swaps/moves and comparisons done
- For graphs, big-O is based on two things:
  - Edges traversed
  - nodes (vertices) inspected

|E| == number of edges
|V| == number of vertices

A traversal graph example

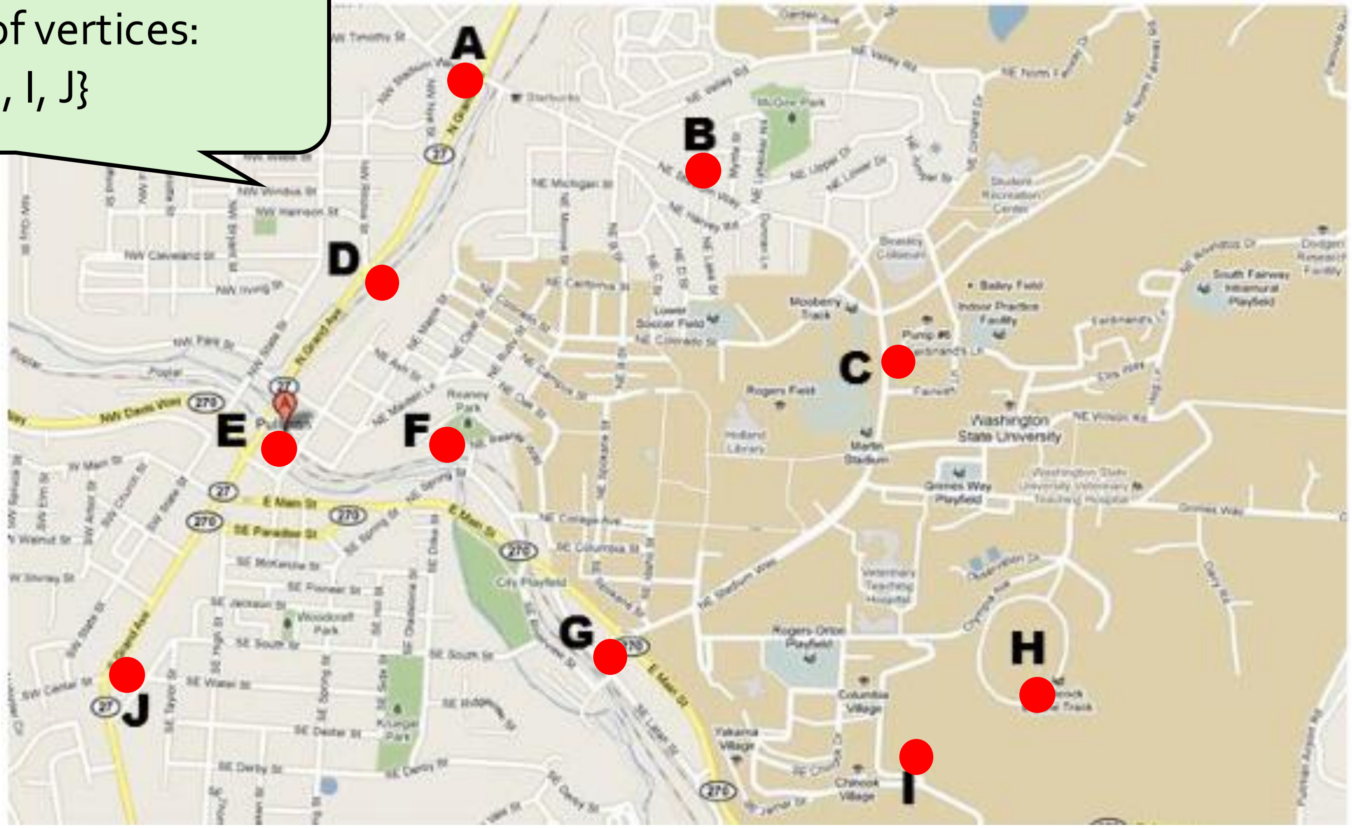
Given a map of WSU and surrounding area, how to get from one place to another?



Graph

A traversal graph example

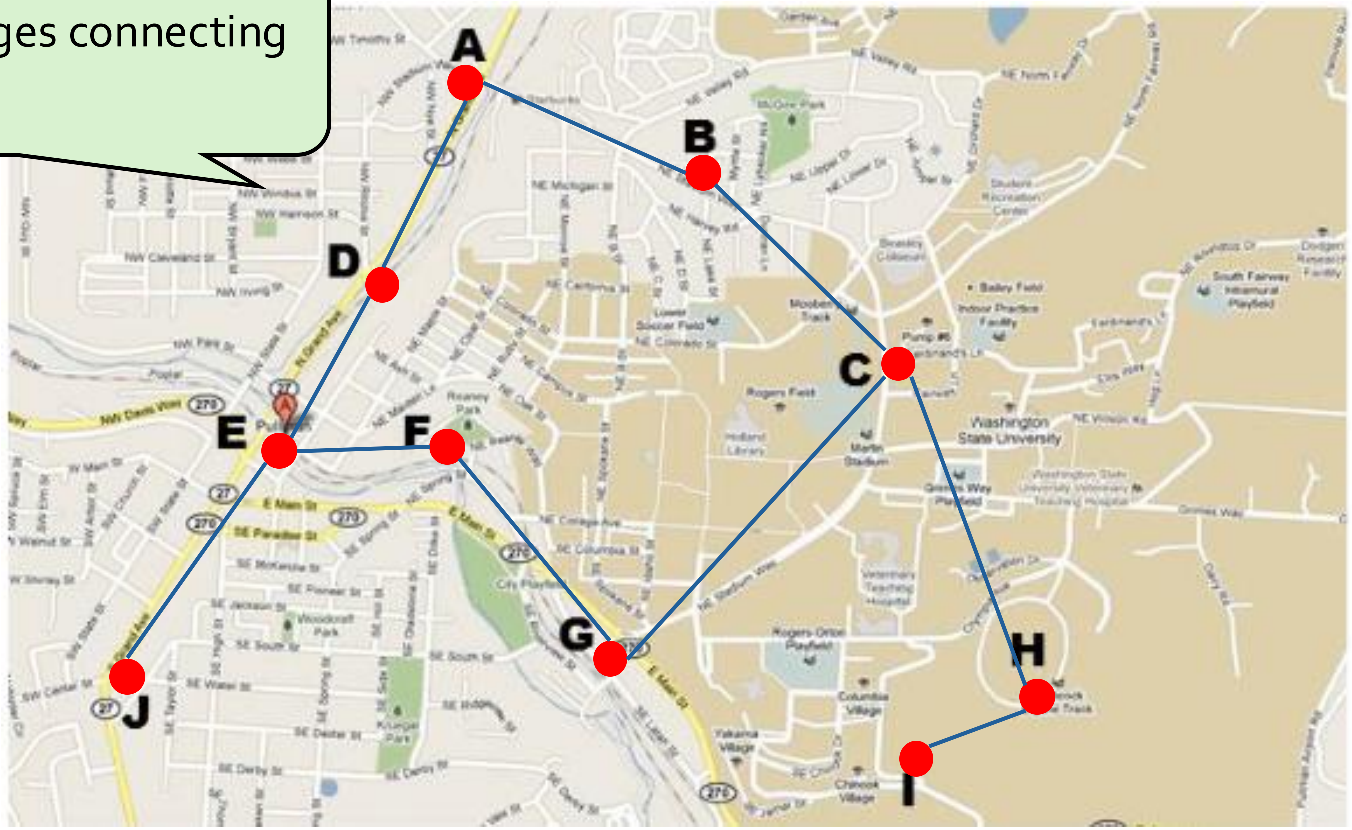
We now have a set of vertices:
{A, B, C, D, E, F, G, H, I, J}



WSU

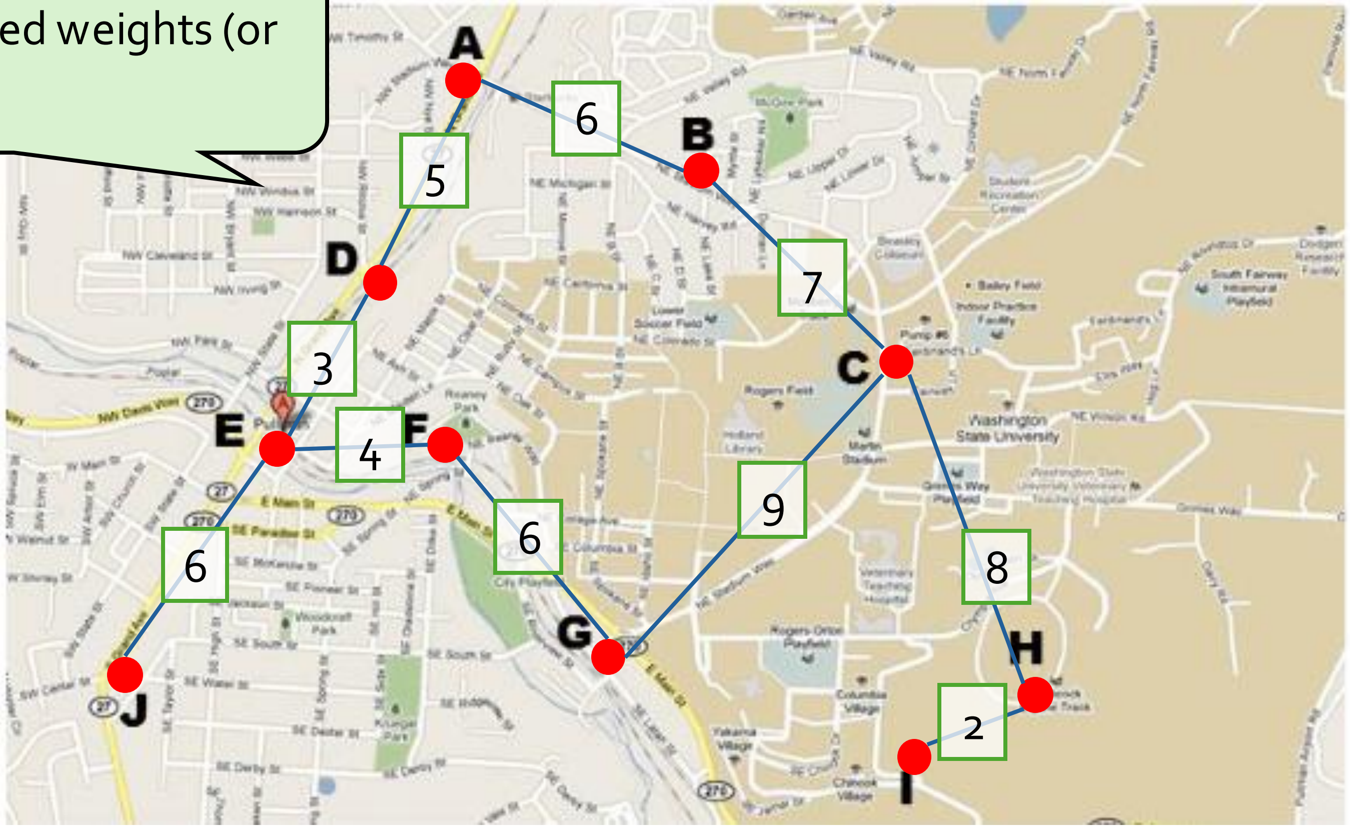
A traversal graph example

We have a set of edges connecting the vertices



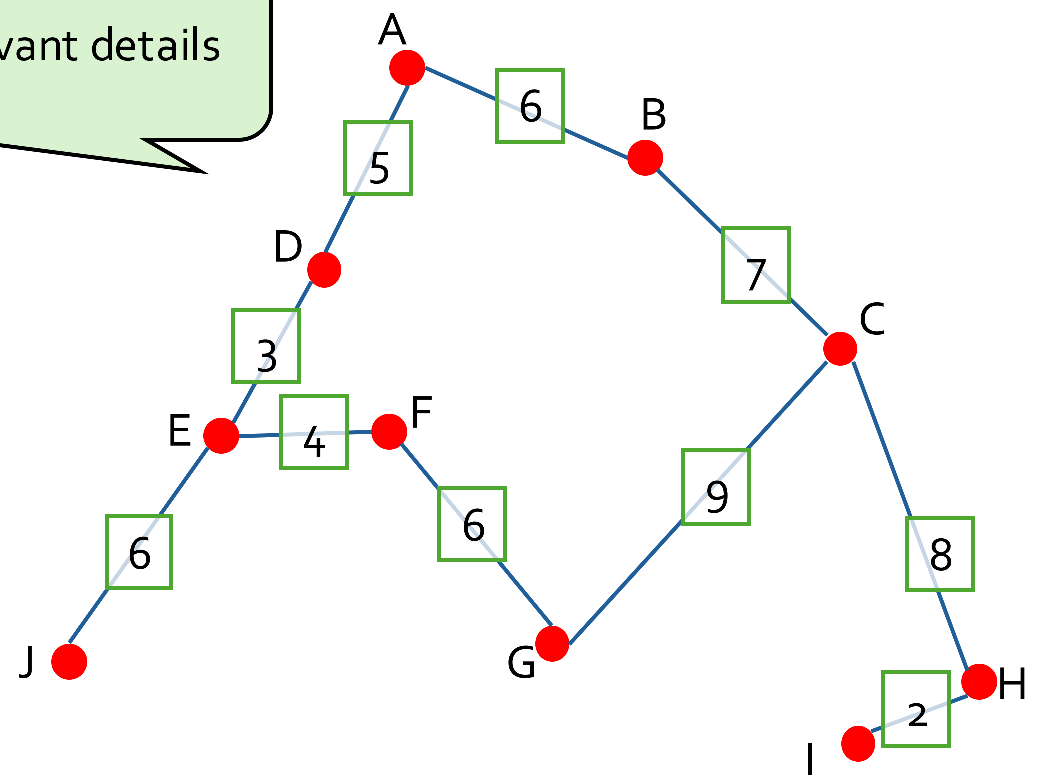
A traversal graph example

Edges can be assigned weights (or costs)



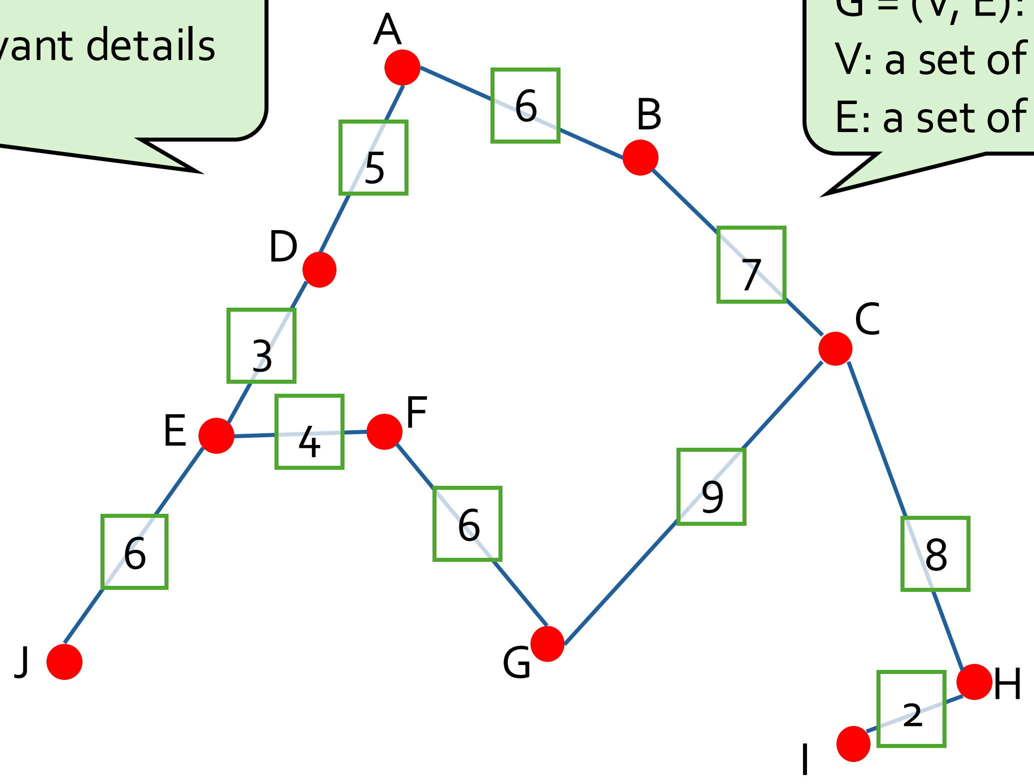
A traversal graph example

Let us strip away irrelevant details



A traversal graph example

Let us strip away irrelevant details



G = (V, E): a graph
V: a set of vertices
E: a set of edges

Graph representation: how?

- Determining what you need to represent
- Find points of interest
- How can an object traverse between those points?
    - Is it on the physical world?
- Once you have the nodes and a way to connect them as edges, you have a graph

Terminology in graphs

- Due to their flexibility graphs have many terms to keep track of
- Ensuring you are fluent in the terminology is a huge part of being able to converse, reason about, and leverage graph algorithms
- Be prepared to learn terms as we go along

Undirected graphs

- Edges do not have a front and back, normally shown with a line (no arrow)
  - Edges can be traversed in either direction
  - Think of it being the difference between one-way streets and two-way streets
  - Both nodes are adjacent to each other
  - (B, D) = (D, B)

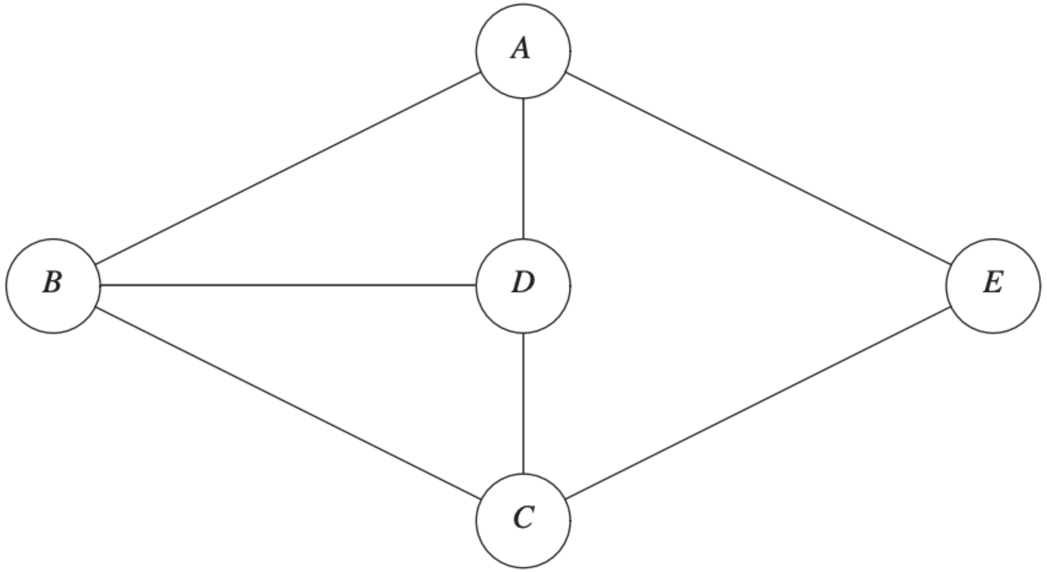
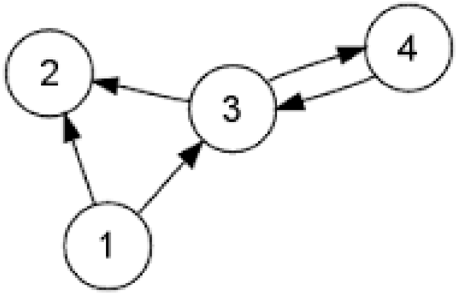


Figure 9.62 An undirected graph

WSU

Directed graphs

- Directed graphs are when the edges are "directed."
- This means they have a front and a back, normally shown as an arrow
- (v, w) ∈ E
  - Edge from v to w
  - (1, 2) ≠ (2, 1)



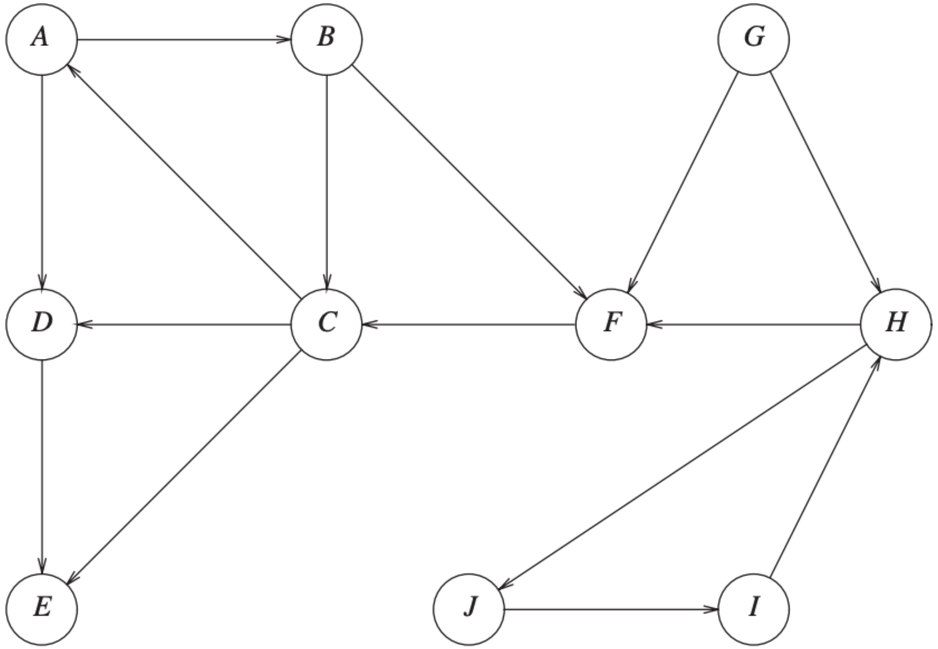
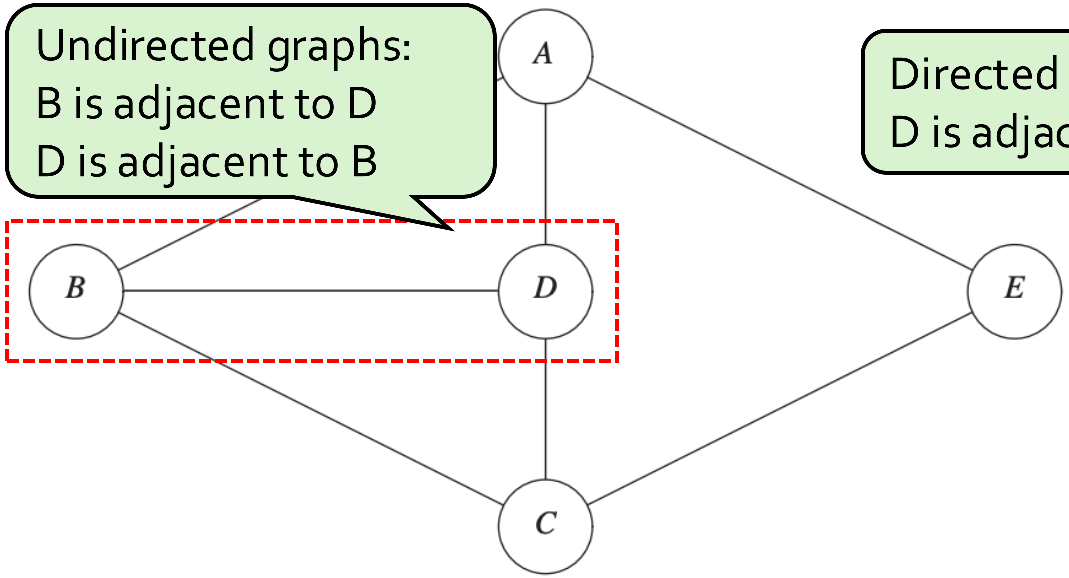


Figure 9.76 A directed graph

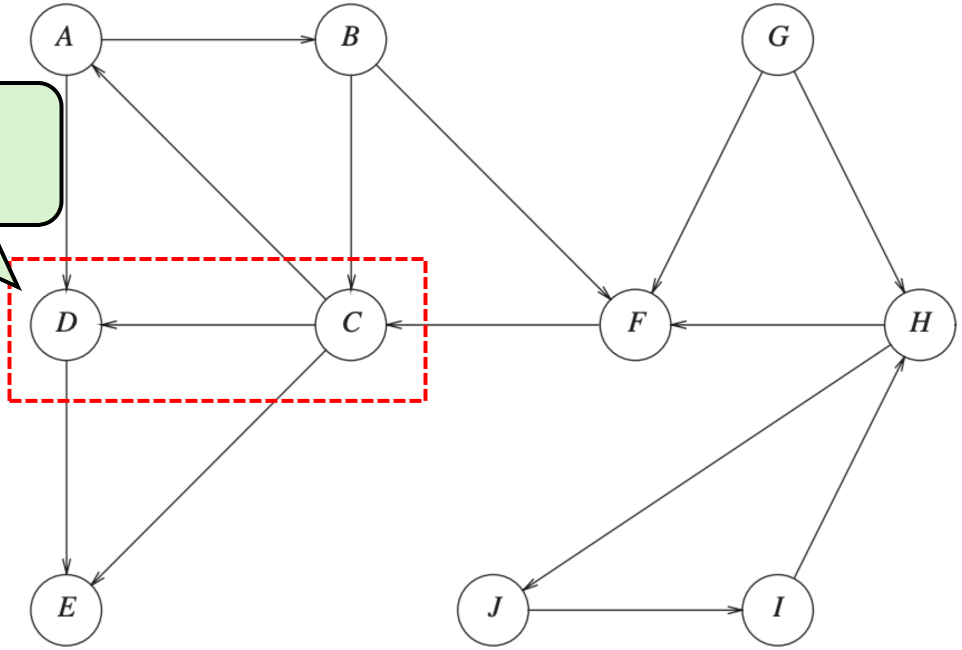
Graphs: adjacency

- Vertex w is adjacent to v if and only if (v, w) ∈ E
  - This means you can traverse the edge from v to w
- When using undirected edges, (v, w) means (w, v) so w and v are both adjacent to each other

Undirected graphs:
B is adjacent to D
D is adjacent to B



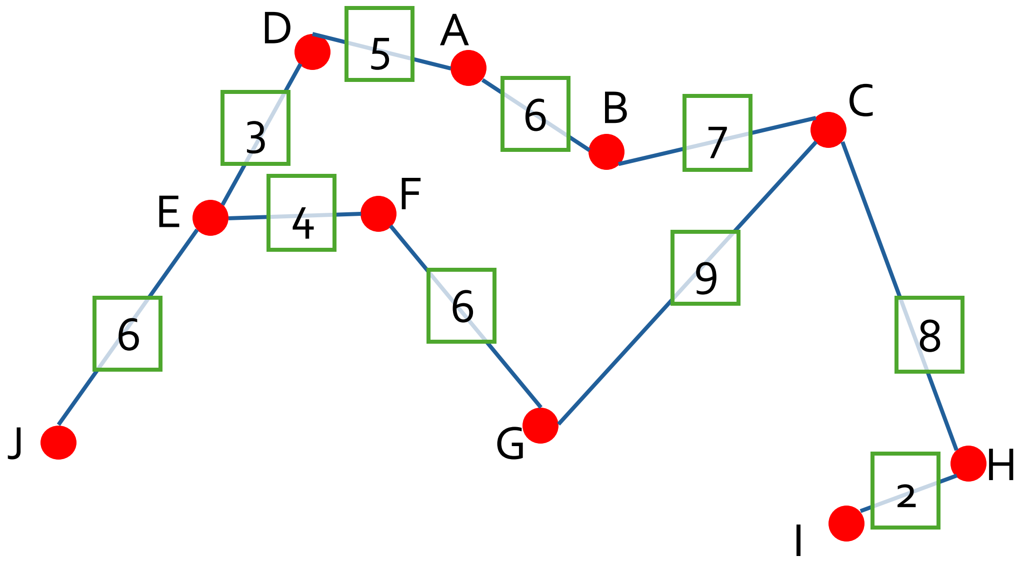
Directed graphs:
D is adjacent to C



WSU

Weight or “cost” of an edge

- Edges can carry a “cost” to traverse them
  - For example, two intersections are connected and the cost is how many meters long the connecting road is
  - What the cost means is entirely dependent upon what the graph is representing



WSU

Degree of a vertex

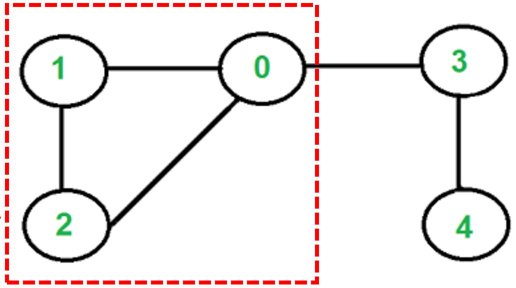
- The number of vertices adjacent to a vertex
- Indegree is the number of incoming edges
- Outdegree is the number of outgoing edges

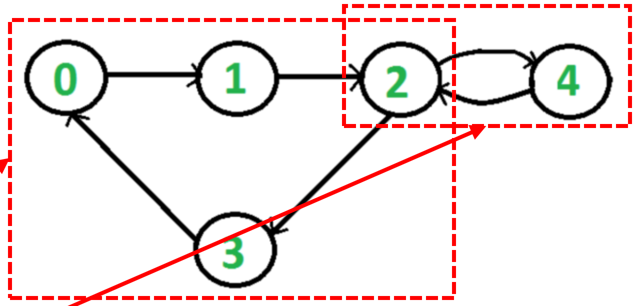
Paths

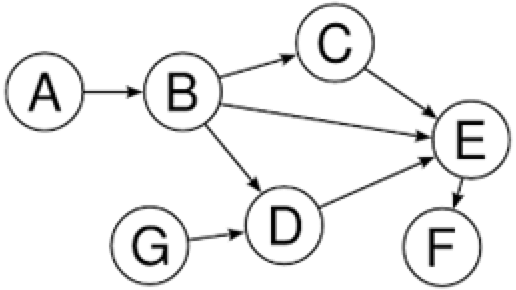
- A path is a sequence of vertices
  - $w_1, w_2, w_3, \ldots, w_N$ such that $(w_i, w_{i+1}) \in E$ for $1 \le i < N$
- The length of a path is the number of edges on the path (not vertices)
- The path can go from a vertex to itself (a special case)
  - If that path has no edges, it has a length of zero.
- A path can be (v, v): a loop
  - Normally loops don't happen in most algorithm traversals, but can happen
- Simple paths: all vertices are distinct (no repeated vertices)
  - Exception: First and last can be the same if it's a path and a loop.

Cycles

- Cycle in a graph is a non-empty path in which all vertices are distinct except the first and last one
  - 0 → 1 → 2 → 0
- Directed cycle: cycle in a directed graph
  - Cycle 1: 0 → 1 → 2 → 3 → 0
  - Cycle 2: 2 → 4 → 2
- Directed Acyclic Graph
  - Directed graphs with no cycles

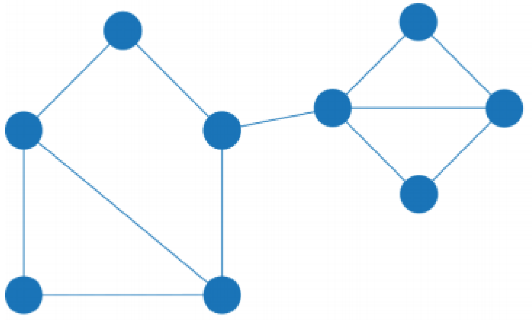




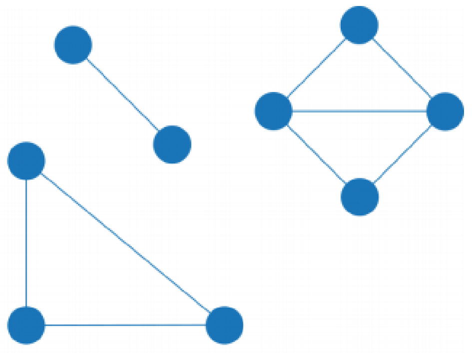


Connected and disconnected

- An undirected graph is a connected graph if there is a path from every vertex to every other vertex

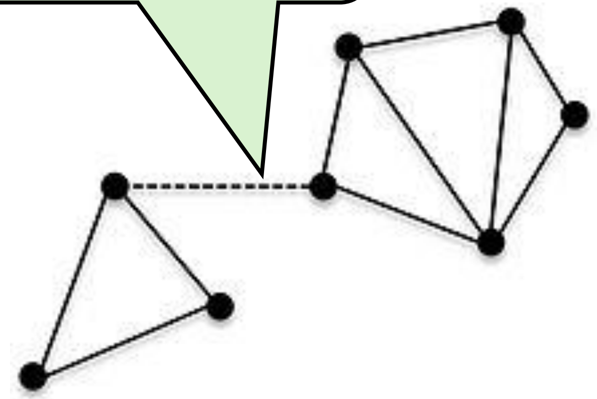


Connected Graph



Disconnected Graph
Includes 3 components.

This graph becomes disconnected when the dashed edge is removed

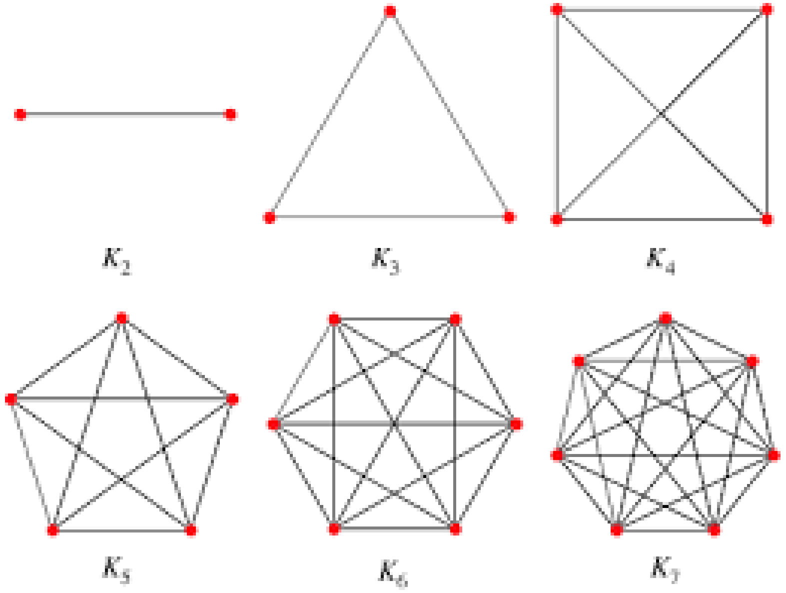


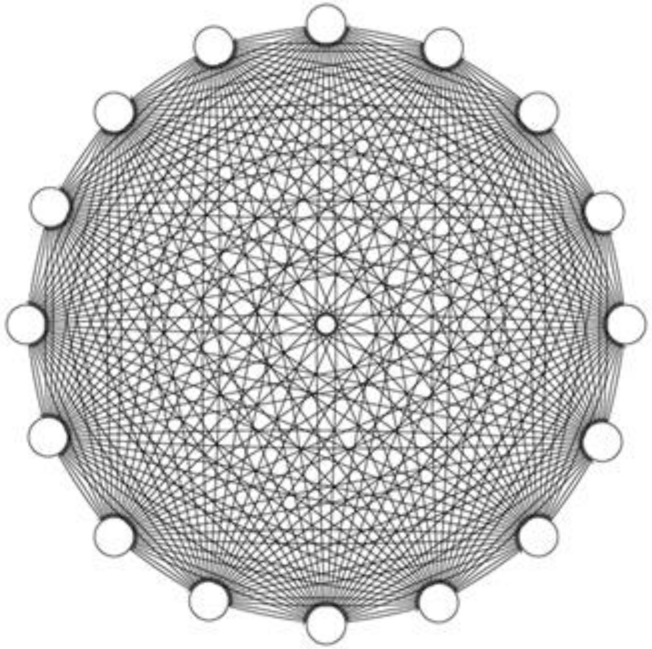
Connected and disconnected

- A directed graph is weakly connected if the underlying undirected graph is a connected graph
- A directed graph is strongly connected if it contains a directed path from x to y (and from y to x) for every pair of vertices (x, y)

Complete graph

- When there is an edge between every pair of vertices



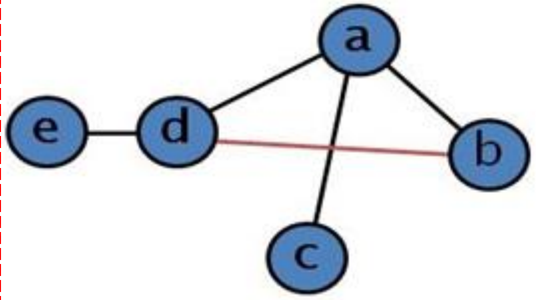


Graph: examples

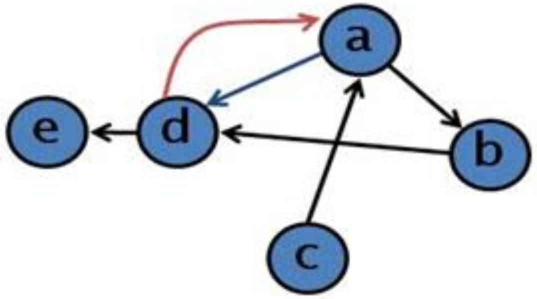
- Airport connections
- Road trip route planning
- Traffic flow
- Networking
- LinkedIn
- Course prerequisites: a DAG
- What else?

Storing and representing graphs

- Adjacency matrix
  - A 2-dimension array where each dimension contains all vertices
  - For each edge (u, v), we set A[u][v] to true; otherwise the entry in the array is false
  - If edges have weights, set A[u][v] equal to the weight and use either a very large or a very small weight to indicate nonexistent edges (e.g., INF or -INF)

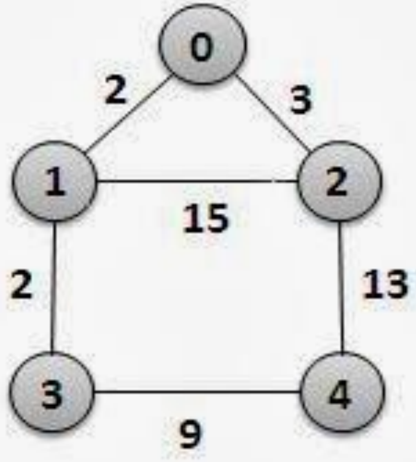


	a	b	c	d	e
a	0	1	1	1	0
b	1	0	0	1	0
c	1	0	0	0	0
d	1	1	0	0	1
e	0	0	0	1	0



	a	b	c	d	e
a	0	1	0	1	0
b	0	0	0	1	0
c	1	0	0	0	0
d	1	0	0	0	1
e	0	0	0	0	0

Weights in adjacency matrix



	0	1	2	3	4
0	0	2	3	0	0
1	2	0	15	2	0
2	3	15	0	0	13
3	0	2	0	0	9
4	0	0	13	9	0

Adjacency Matrix Representation of Weighted Graph

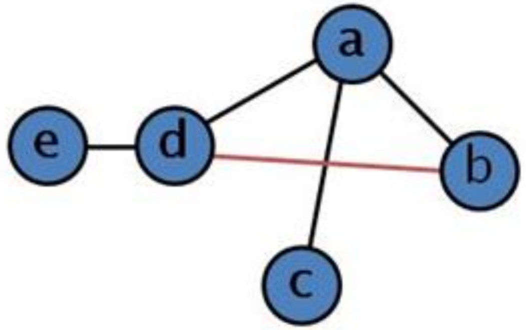
Storing and representing graphs

- Adjacency matrix
  - A 2-dimension array where each dimension contains all vertices
  - For each edge (u, v), we set A[u][v] to true; otherwise the entry in the array is false
  - If edges have weights, set A[u][v] equal to the weight and use either a very large or a very small weight to indicate nonexistent edges (e.g., INF or -INF)
  - Disadvantage?
    - Requires $\Theta(|V|^2)$ space
    - only appropriate if $|E| = \Theta(|V|^2)$
    - Wasteful if $|E| << O(|V|^2)$

Dense v.s. sparse

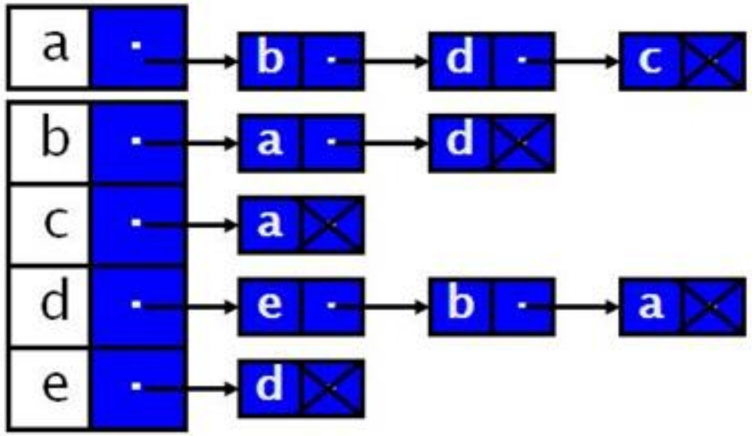
- Most graphs are sparse
  - This means |E| << |V|
- Better solution for sparse graphs is an adjacency list representation
  - Keep a list of all adjacent vertices for each vertex
  - Space requirement becomes O(|E| + |V|)
    - Instead of Θ(|V|^2) with the matrix
  - Weights can be kept with edges in adjacency list
  - Standard way to represent graphs

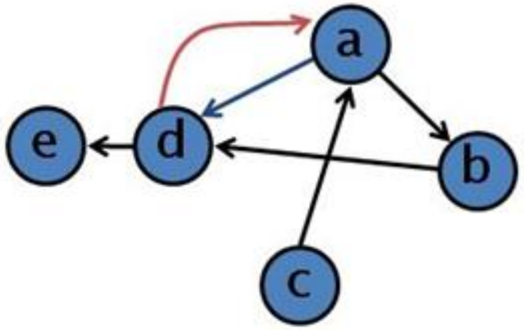
Dense v.s. sparse



	a	b	c	d	e
a	0	1	1	1	0
b	1	0	0	1	0
c	1	0	0	0	0
d	1	1	0	0	1
e	0	0	0	1	0

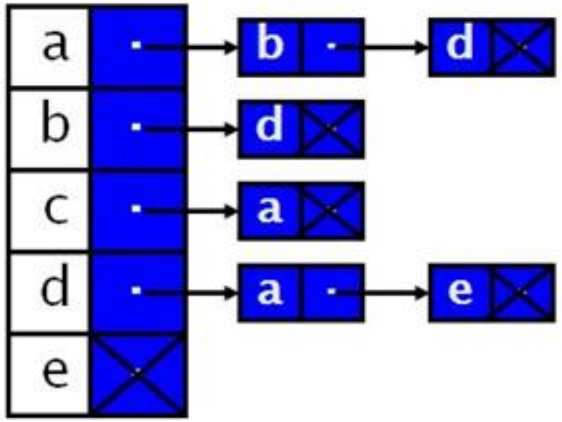
v.s.





	a	b	c	d	e
a	0	1	0	1	0
b	0	0	0	1	0
c	1	0	0	0	0
d	1	0	0	0	1
e	0	0	0	0	0

v.s.



WSU

Topological sort

- 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 v_i in the ordering
- Not work if there is a cycle in the graph
- Does not guarantee a unique ordering
- Used for deciding scheduling of work units
  - Edges represent the dependency of work units
  - Only those with an indegree of 0 can be “done” next

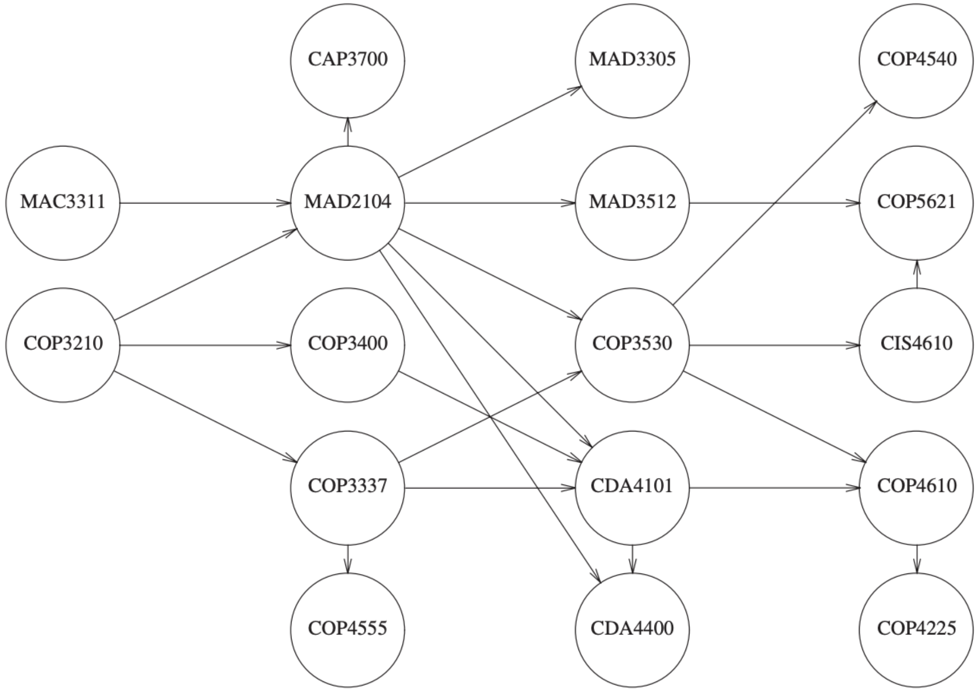


Figure 9.3 An acyclic graph representing course prerequisite structure

WSU

Topographical sorting example

- {v1 , v2 , v5 , v4 , v3 , v7 , v6} and {v1 , v2 , v5 , v4 , v7 , v3 , v6} are both valid topological orderings
  1. Find node with no "in-edges"
     1) In-degree of zero
     2) Called a "source node"
  2. Print out (process) node && remove it (and edges implicitly) from graph
- 3) Repeat

Start from v1:
in-degree = 0

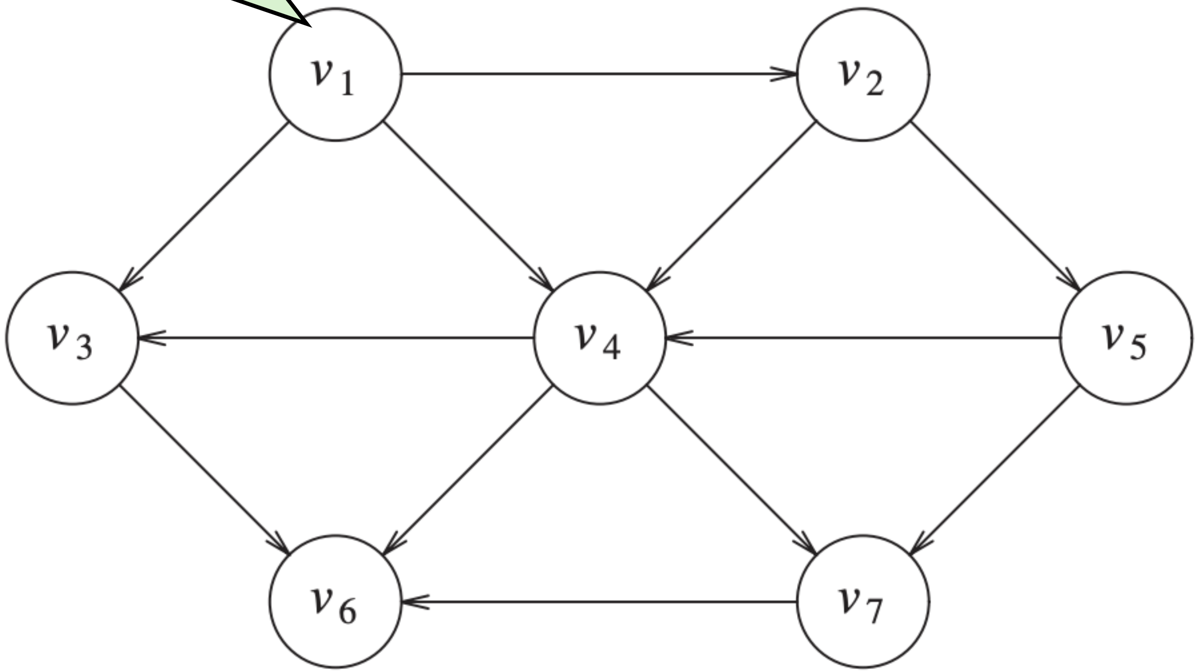
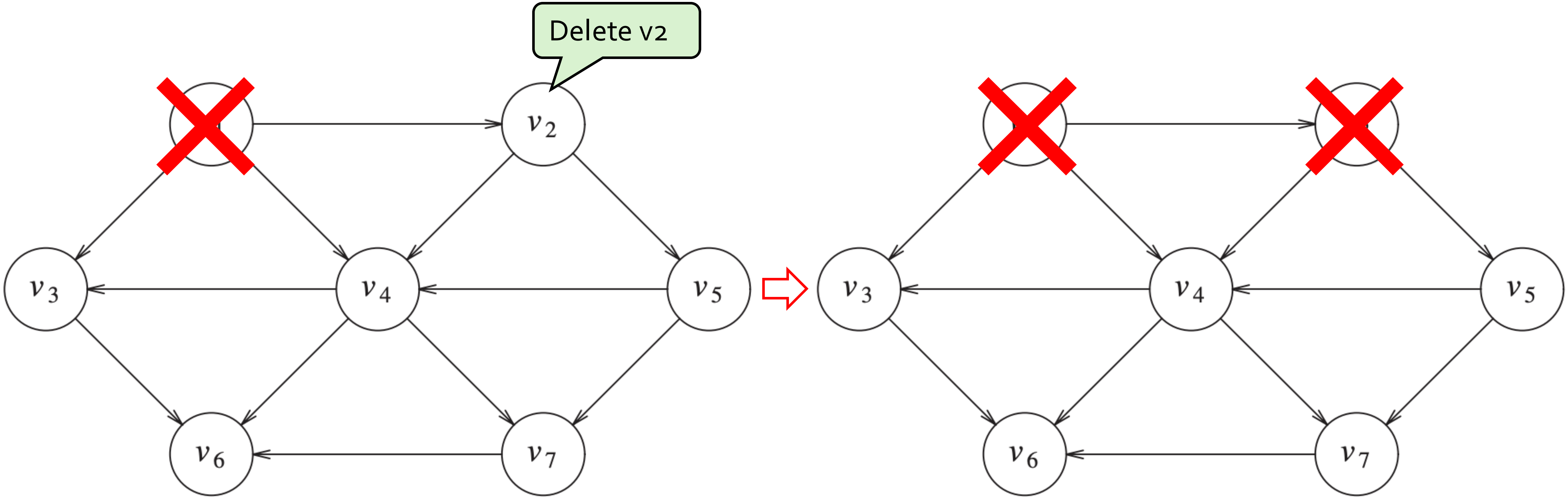
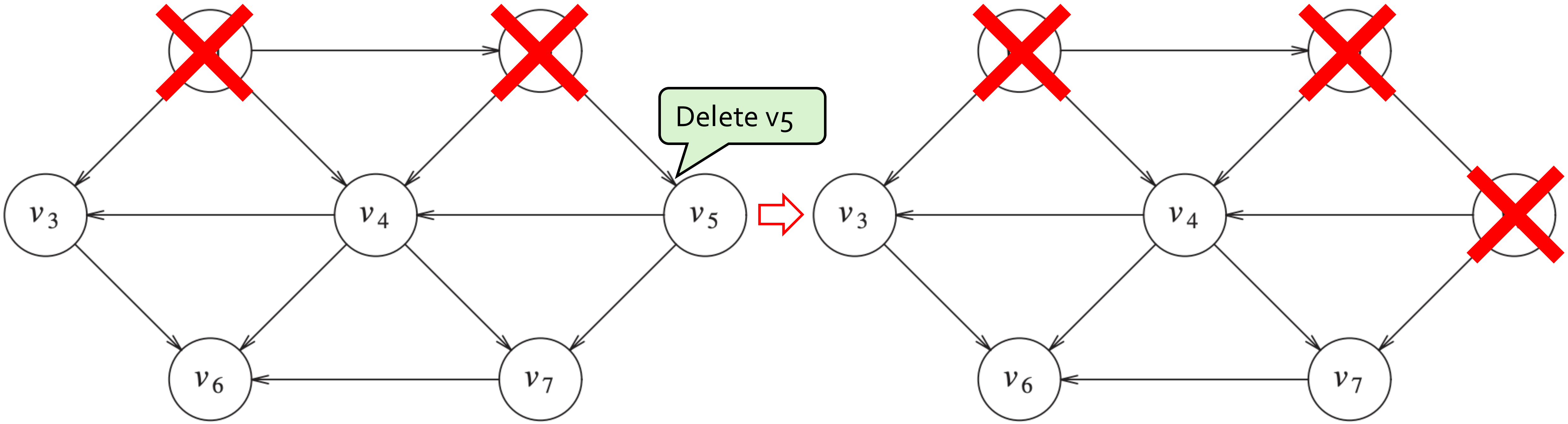


Figure 9.4 An acyclic graph

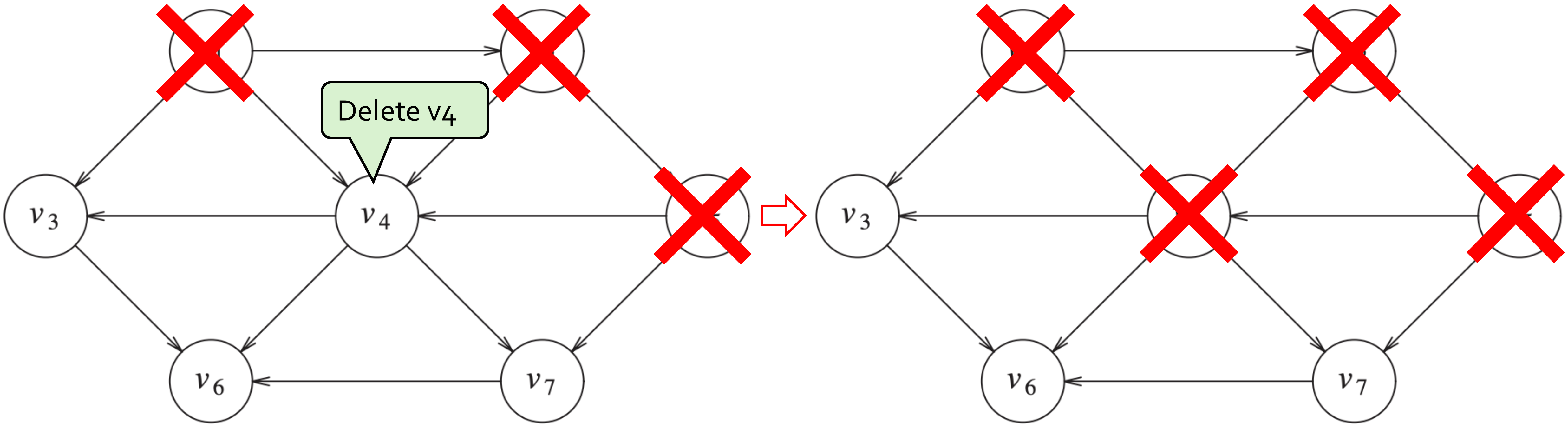
Topographical sorting example



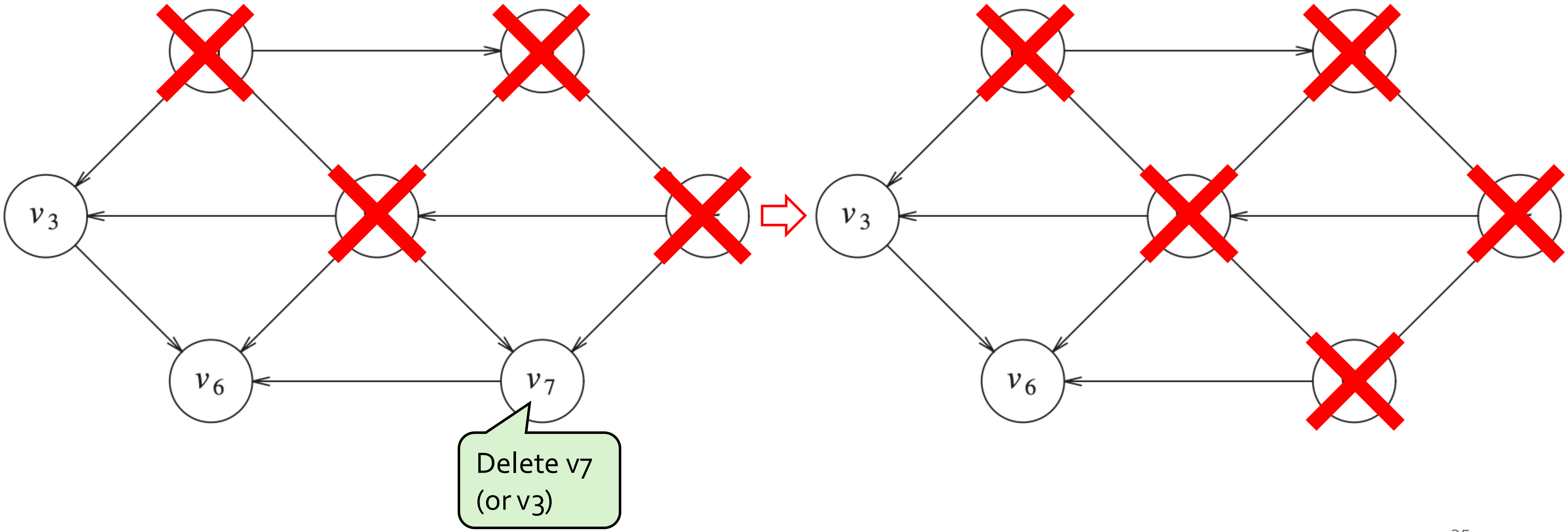
Topographical sorting example



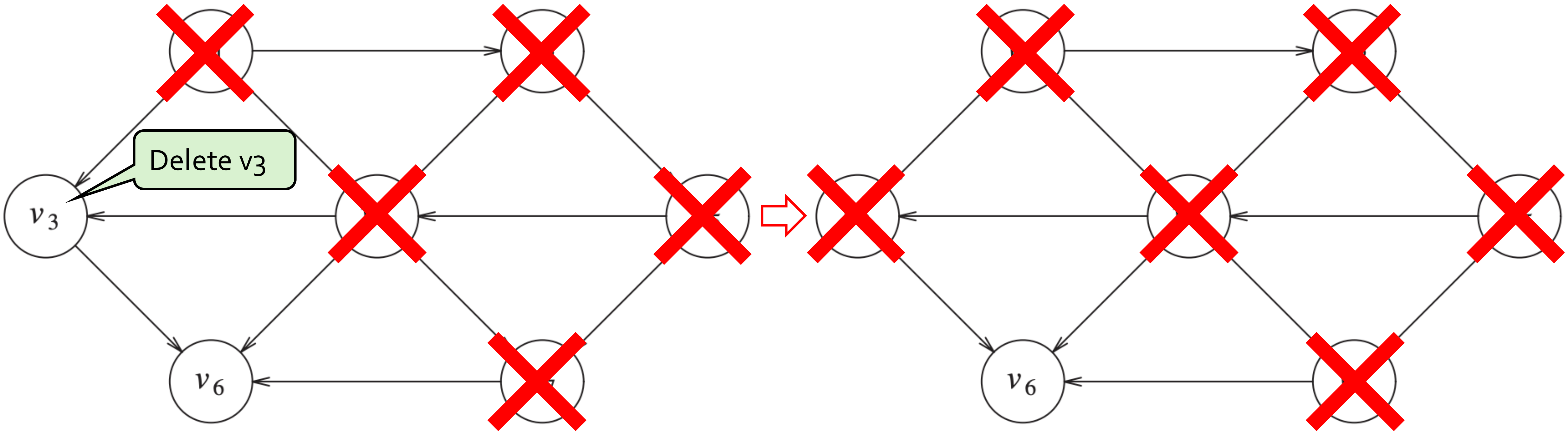
Topographical sorting example



Topographical sorting example

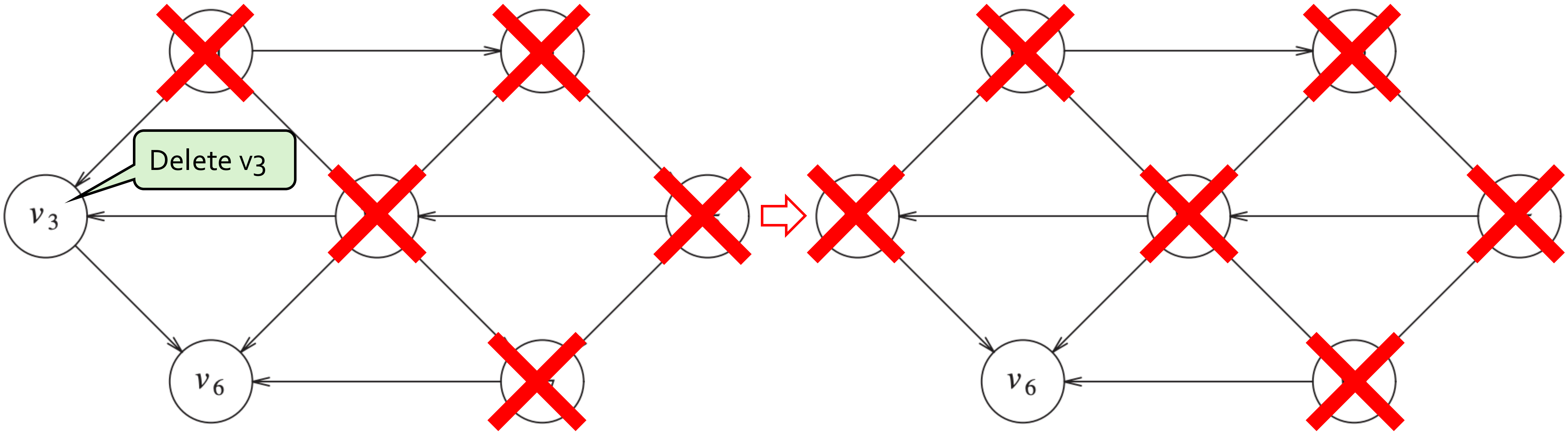


Topographical sorting example



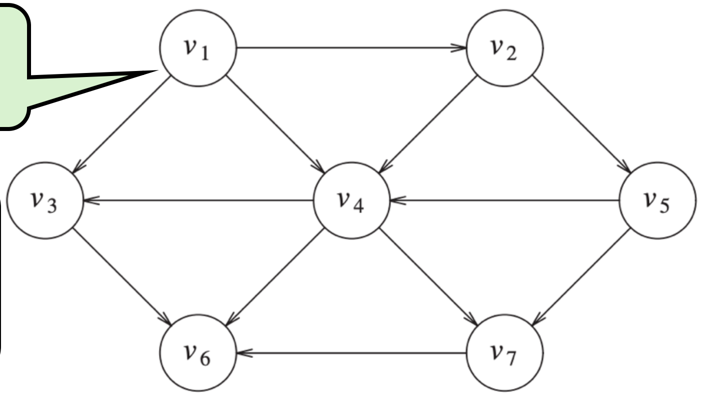
Graph

Topographical sorting example



Delete order: v1, v2, v5, v4, v7, v3, v6 → topographical sorting order

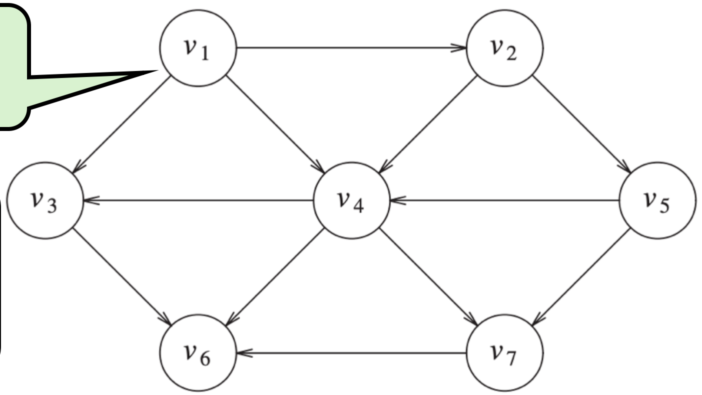
Start from v1:
in-degree = 0



Implementation:
Save nodes in a heap
by their in-degrees

V1 - 0	V2 - 1	V5 - 1	V3 - 2	V7 - 2	V4 - 3	V6 - 3

Start from v1:
in-degree = 0

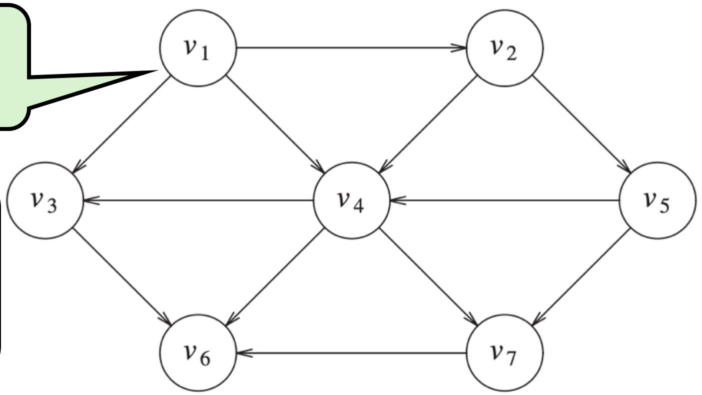


Implementation:
Save nodes in a heap
by their in-degrees

deleteMin
and update
in-degrees

V1 - 0	V2 - 1	V5 - 1	V3 - 2	V7 - 2	V4 - 3	V6 - 3
V2 - 0	V3 - 1	V5 - 1	V4 - 2	V7 - 2	V6 - 3	

Start from v1:
in-degree = 0

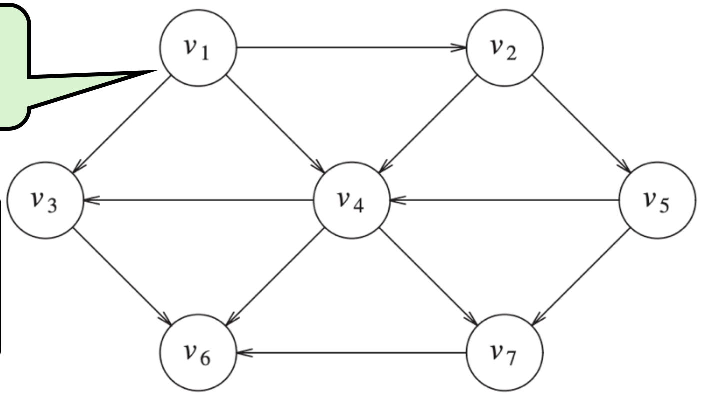


Implementation:
Save nodes in a heap
by their in-degrees

deleteMin
and update
in-degrees

V1 - 0	V2 - 1	V5 - 1	V3 - 2	V7 - 2	V4 - 3	V6 - 3
V2 - 0	V3 - 1	V5 - 1	V4 - 2	V7 - 2	V6 - 3	
V5 - 0	V3 - 1	V6 - 3	V4 - 1	V7 - 2		

Start from v1:
in-degree = 0

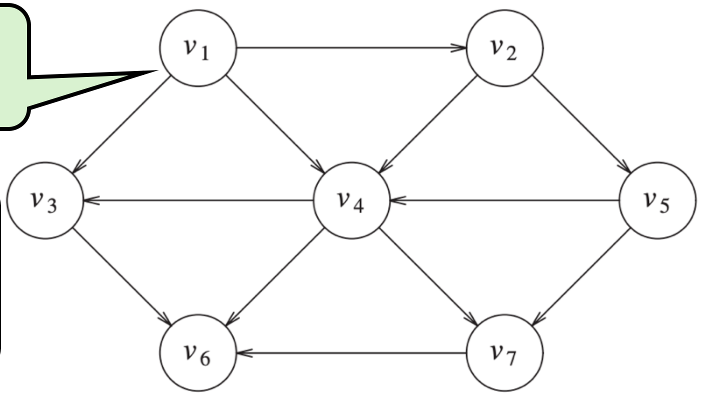


Implementation:
Save nodes in a heap
by their in-degrees

deleteMin
and update
in-degrees

V1 - 0	V2 - 1	V5 - 1	V3 - 2	V7 - 2	V4 - 3	V6 - 3
V2 - 0	V3 - 1	V5 - 1	V4 - 2	V7 - 2	V6 - 3	
V5 - 0	V3 - 1	V6 - 3	V4 - 1	V7 - 2		
V4 - 0	V3 - 1	V6 - 3	V7 - 1			

Start from v1:
in-degree = 0

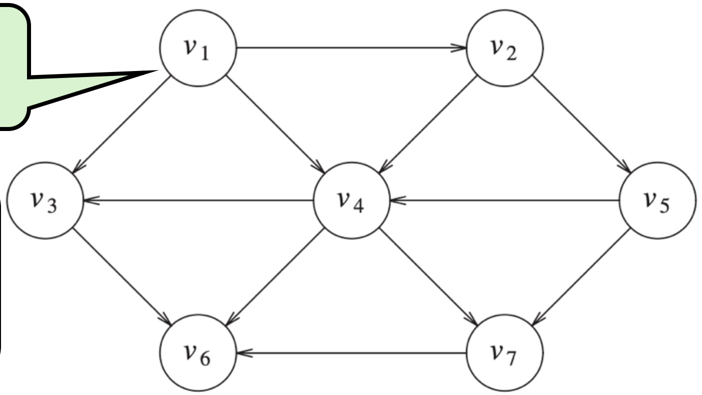


Implementation:
Save nodes in a heap
by their in-degrees

deleteMin
and update
in-degrees

V1 - 0	V2 - 1	V5 - 1	V3 - 2	V7 - 2	V4 - 3	V6 - 3
V2 - 0	V3 - 1	V5 - 1	V4 - 2	V7 - 2	V6 - 3	
V5 - 0	V3 - 1	V6 - 3	V4 - 1	V7 - 2		
V4 - 0	V3 - 1	V6 - 3	V7 - 1			
V7 - 0	V3 - 0	V6 - 2				

Start from v1:
in-degree = 0



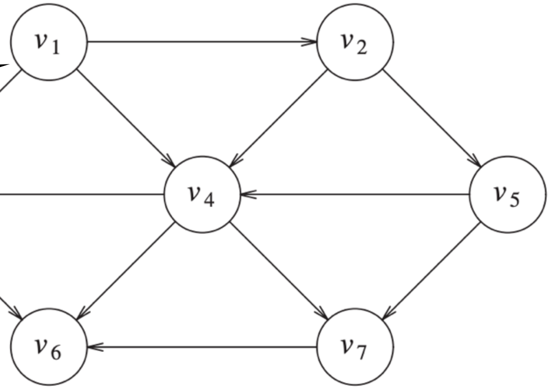
WSU

Implementation:
Save nodes in a heap
by their in-degrees

deleteMin
and update
in-degrees

V1 - 0	V2 - 1	V5 - 1	V3 - 2	V7 - 2	V4 - 3	V6 - 3
V2 - 0	V3 - 1	V5 - 1	V4 - 2	V7 - 2	V6 - 3	
V5 - 0	V3 - 1	V6 - 3	V4 - 1	V7 - 2		
V4 - 0	V3 - 1	V6 - 3	V7 - 1			
V7 - 0	V3 - 0	V6 - 2				
V3 - 0	V6 - 1					

Start from v1:
in-degree = 0

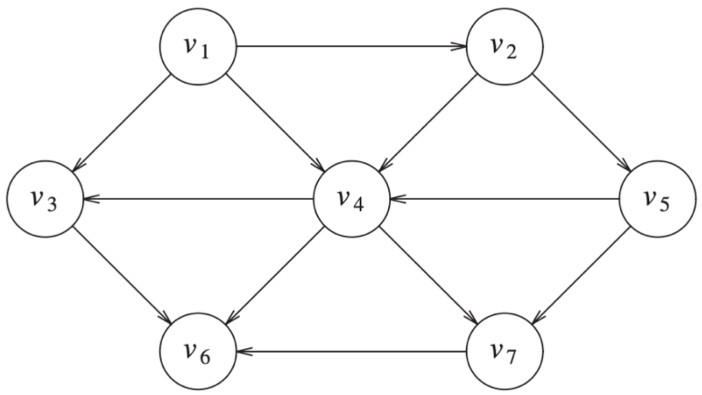


Implementation:
Save nodes in a heap
by their in-degrees

deleteMin
and update
in-degrees

V1 - 0	V2 - 1	V5 - 1	V3 - 2	V7 - 2	V4 - 3	V6 - 3
V2 - 0	V3 - 1	V5 - 1	V4 - 2	V7 - 2	V6 - 3	
V5 - 0	V3 - 1	V6 - 3	V4 - 1	V7 - 2		
V4 - 0	V3 - 1	V6 - 3	V7 - 1			
V7 - 0	V3 - 0	V6 - 2				
V3 - 0	V6 - 1					
V6 - 0						

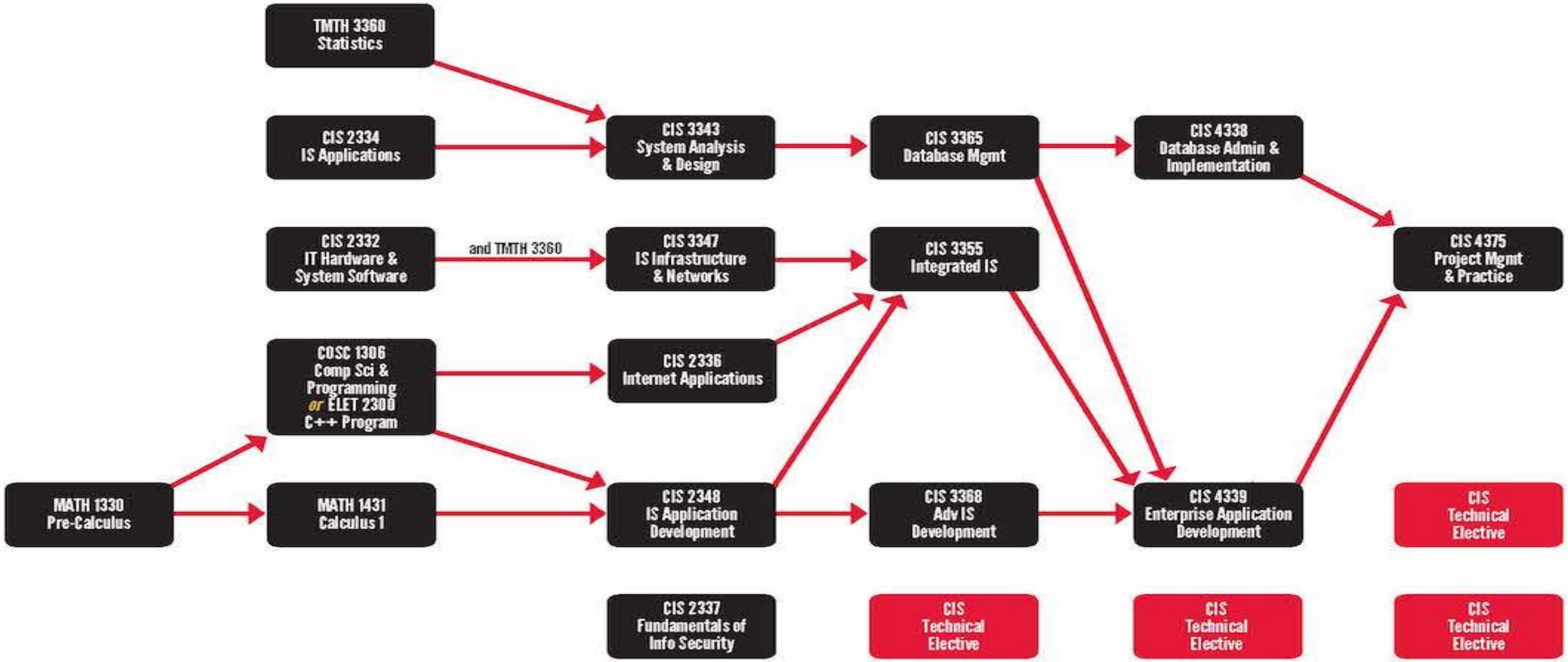
WSU



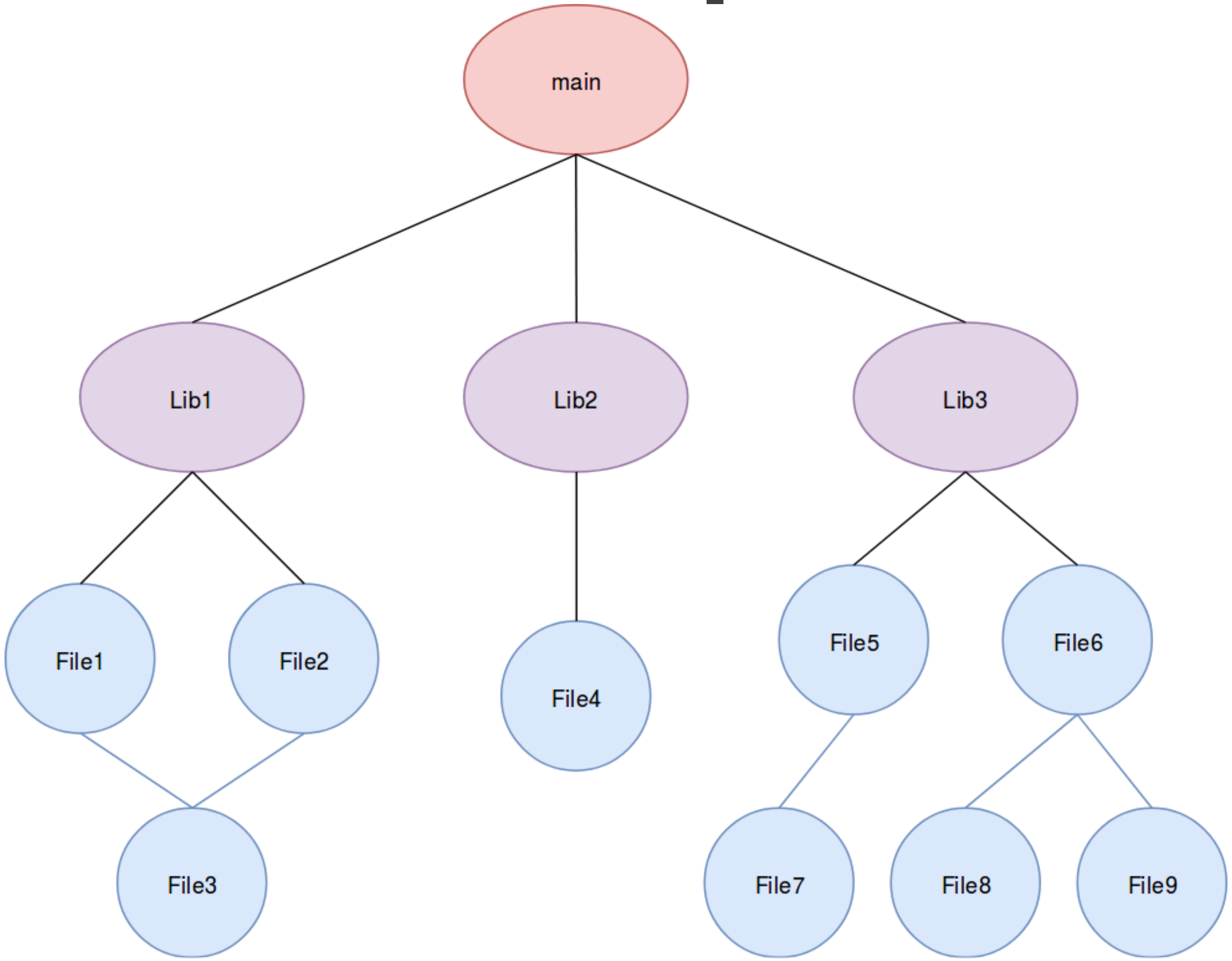
This is the output of topological sort

V1 - 0	V2 - 1	V5 - 1	V3 - 2	V7 - 2	V4 - 3	V6 - 3
V2 - 0	V3 - 1	V5 - 1	V4 - 2	V7 - 2	V6 - 3	
V5 - 0	V3 - 1	V6 - 3	V4 - 1	V7 - 2		
V4 - 0	V3 - 1	V6 - 3	V7 - 1			
V7 - 0	V3 - 0	V6 - 2				
V3 - 0	V6 - 1					
V6 - 0						

Application of topo sort



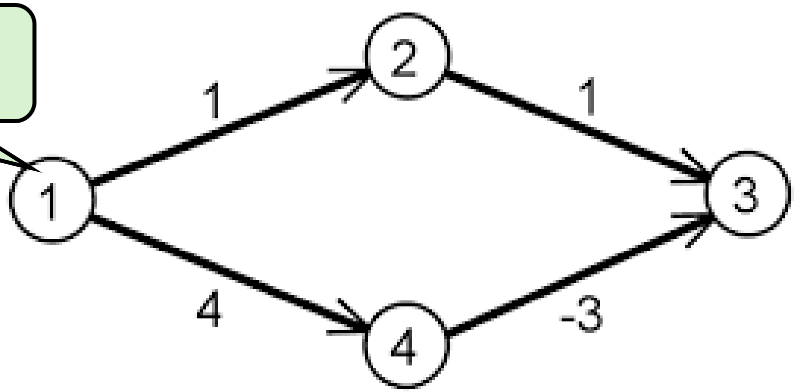
Application of topo sort



Shortest path algorithms

- How to go across graph at the lowest cost from A to B?
  - “Short” can be defined in lots of ways - entirely application dependent
  - This is where the cost of an edge truly starts to matter

What is the shortest path from 1 to 3?



Shortest path: formal definition

- Single-source shortest-path problem:
  - 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.
- Cost of a path is:
  - Associated with each edge (v_i , v_j) is a cost c{i,j} to traverse the edge. The cost of a path from V_1 to V_N:

$$\sum_{i=1}^{N-1} c_{i,i+1}$$

WSU

Shortest path: positive edges

- v1 to v6?

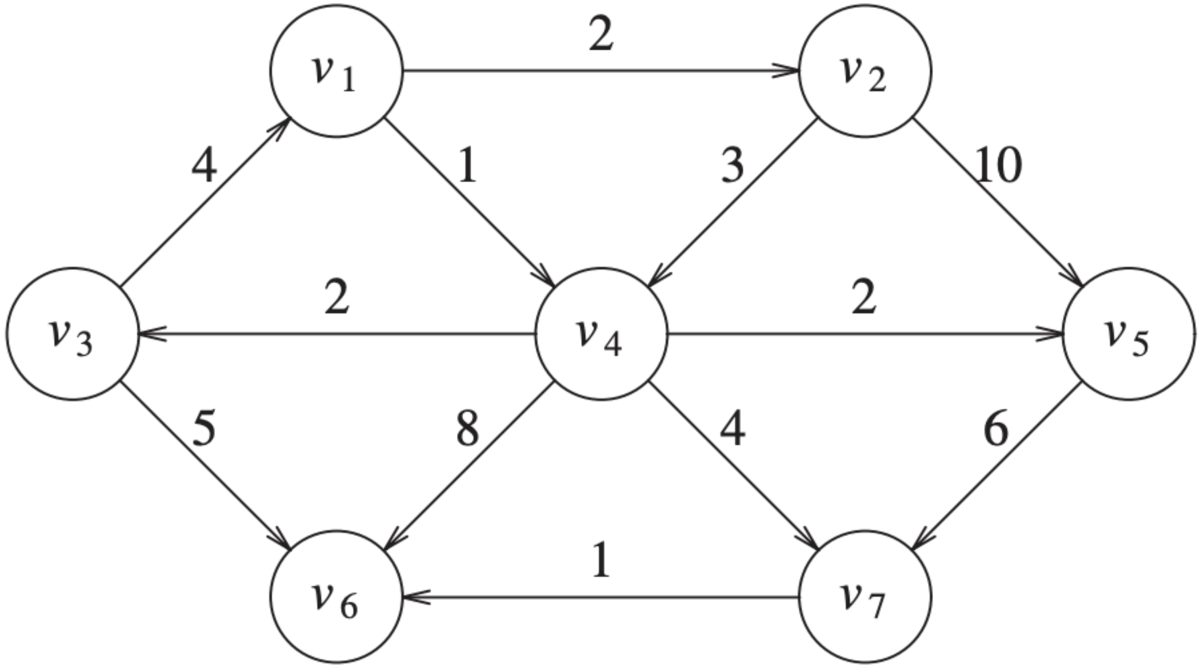


Figure 9.8 A directed graph *G*

WSU

Shortest path: positive edges

- v1 to v6?
  - v1 → v4 → v7 → v6
  - Sum cost: 6

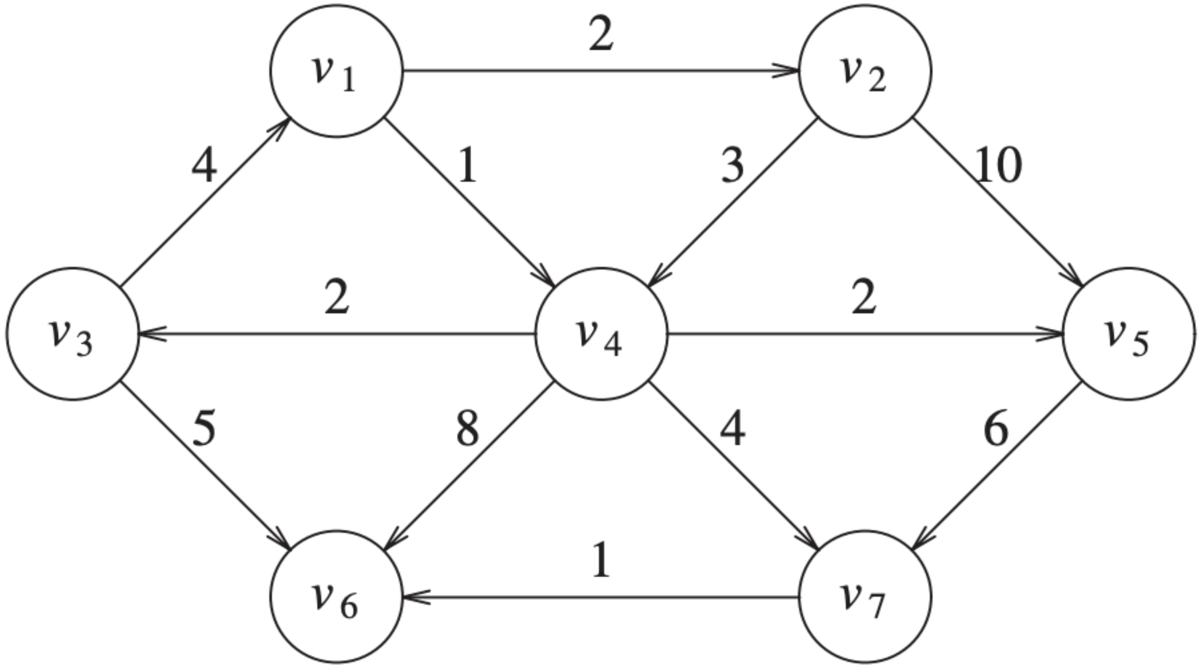


Figure 9.8 A directed graph G

Shortest path: negative edges

- Go: v5 -> v4
  - Takes 1 right?
- What about:
  v5 -> v4 -> v2 -> v5 -> v4?
  Perhaps you go around again?
- Extreme case: If I add 1 more edge (v5, v2), cost: -5, will you ever find a shortest path for v2 -> v5?
- When a negative cycle exists, shortest paths are not defined!
  -- Cost can go to -INF

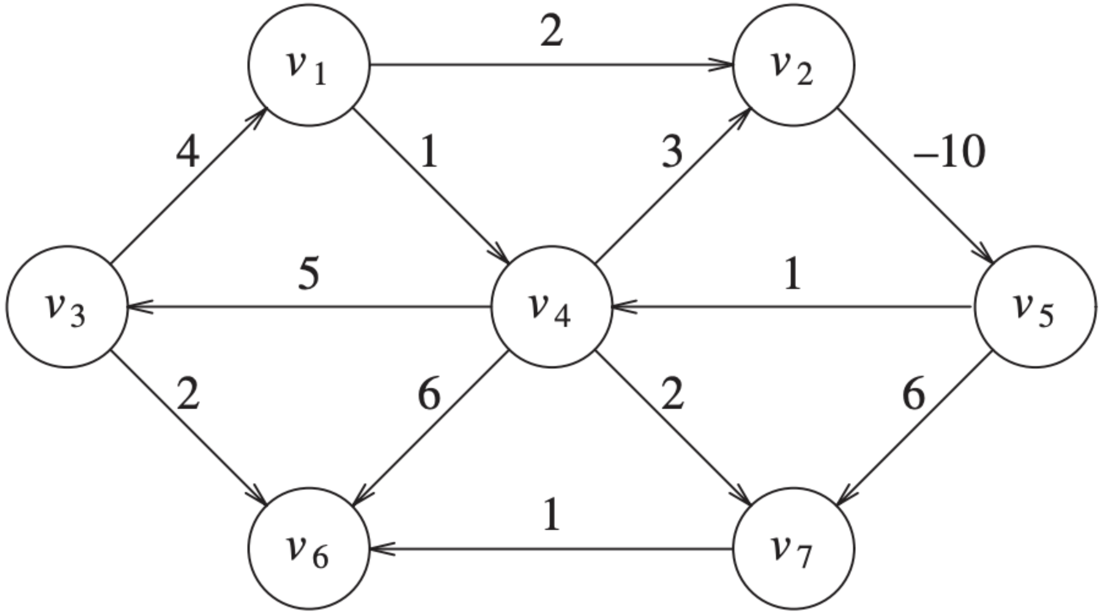


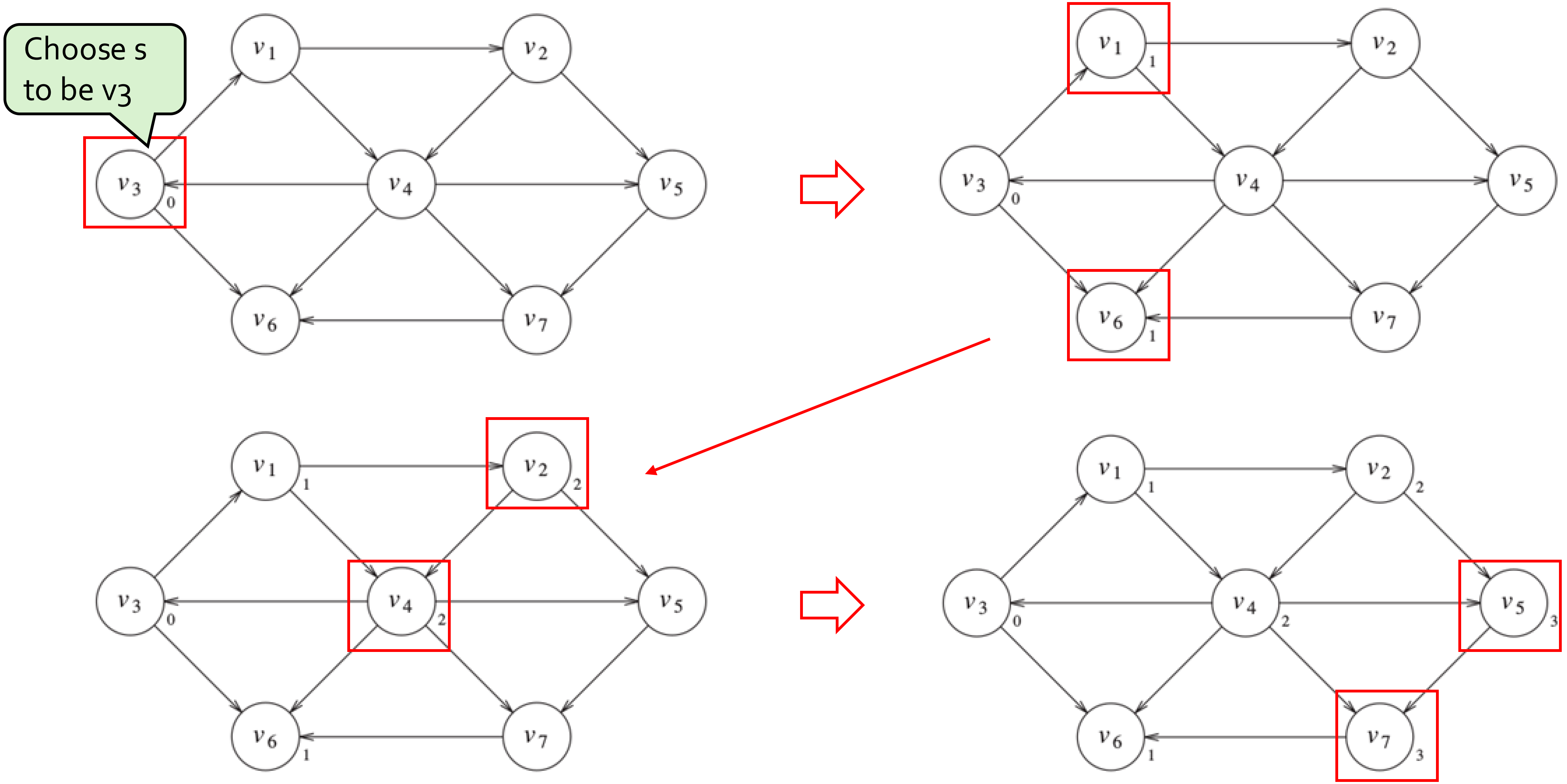
Figure 9.9 A graph with a negative-cost cycle

Unweighted shortest path

- Find shortest path from a vertex s to all other vertices
- Only care about number of edges in path, not their costs (cost == 1)
  1. Mark starting node (s) with length 0
  2. Look at all adjacent vertices with distance 1 from s
  3. Repeat for all vertices at distance 2, then 3, etc
  4. Once all nodes are marked, you are finished
- This is a breadth first search (BFS): the network is examined in layers, starting from a root node. Basically, level order traversal for trees
- Final result is all vertices are marked with distance from initial (s)
- Done in O(|E| + |V|) time

Find shortest path from a vertex s to all other vertices

WSU



```
void Graph::unweighted( Vertex s )
{
    for each Vertex v
    {
        v.dist = INFINITY;
        v.known = false;
    }

    s.dist = 0;

    for( int currDist = 0; currDist < NUM_VERTICES; currDist++ )
        for each Vertex v
            if( !v.known && v.dist == currDist )
            {
                v.known = true;
                for each Vertex w adjacent to v
                    if( w.dist == INFINITY )
                    {
                        w.dist = currDist + 1;
                        w.path = v;
                    }
            }
}
```

Initialize the distance of all vertices as INFINITY

Initialize the distance of s as 0

Figure 9.16 Pseudocode for unweighted shortest-path algorithm

WSU

	Initial State			v_3 Dequeued			v_1 Dequeued			v_6 Dequeued		
v	*known*	d_v	p_v	*known*	d_v	p_v	*known*	d_v	p_v	*known*	d_v	p_v
v_1	F	∞	0	F	1	v_3	T	1	v_3	T	1	v_3
v_2	F	∞	0	F	∞	0	F	2	v_1	F	2	v_1
v_3	F	0	0	T	0	0	T	0	0	T	0	0
v_4	F	∞	0	F	∞	0	F	2	v_1	F	2	v_1
v_5	F	∞	0	F	∞	0	F	∞	0	F	∞	0
v_6	F	∞	0	F	1	v_3	F	1	v_3	T	1	v_3
v_7	F	∞	0	F	∞	0	F	∞	0	F	∞	0
Q:	v_3			v_1, v_6			v_6, v_2, v_4			v_2, v_4		

	v_2 Dequeued			v_4 Dequeued			v_5 Dequeued			v_7 Dequeued		
v	*known*	d_v	p_v	*known*	d_v	p_v	*known*	d_v	p_v	*known*	d_v	p_v
v_1	T	1	v_3	T	1	v_3	T	1	v_3	T	1	v_3
v_2	T	2	v_1	T	2	v_1	T	2	v_1	T	2	v_1
v_3	T	0	0	T	0	0	T	0	0	T	0	0
v_4	F	2	v_1	T	2	v_1	T	2	v_1	T	2	v_1
v_5	F	3	v_2	F	3	v_2	T	3	v_2	T	3	v_2
v_6	T	1	v_3	T	1	v_3	T	1	v_3	T	1	v_3
v_7	F	∞	0	F	3	v_4	F	3	v_4	T	3	v_4
Q:	v_4, v_5			v_5, v_7			v_7			empty		

```
void Graph::unweighted( Vertex s )
{
    for each Vertex v
    {
        v.dist = INFINITY;
        v.known = false;
    }

    s.dist = 0;

    for( int currDist = 0; currDist < NUM_VERTICES; currDist++ )
        for each Vertex v
            if( !v.known && v.dist == currDist )
            {
                v.known = true;
                for each Vertex w adjacent to v
                    if( w.dist == INFINITY )
                    {
                        w.dist = currDist + 1;
                        w.path = v;
                    }
            }
}
```

Largest distance = |V|

	Initial State			v_3 Dequeued			v_1 Dequeued			v_6 Dequeued		
v	*known*	d_v	p_v	*known*	d_v	p_v	*known*	d_v	p_v	*known*	d_v	p_v
v_1	F	∞	0	F	1	v_3	T	1	v_3	T	1	v_3
v_2	F	∞	0	F	∞	0	F	2	v_1	F	2	v_1
v_3	F	0	0	T	0	0	T	0	0	T	0	0
v_4	F	∞	0	F	∞	0	F	2	v_1	F	2	v_1
v_5	F	∞	0	F	∞	0	F	∞	0	F	∞	0
v_6	F	∞	0	F	1	v_3	F	1	v_3	T	1	v_3
v_7	F	∞	0	F	∞	0	F	∞	0	F	∞	0
Q:	v_3			v_1, v_6			v_6, v_2, v_4			v_2, v_4		

	v_2 Dequeued			v_4 Dequeued			v_5 Dequeued			v_7 Dequeued		
v	*known*	d_v	p_v	*known*	d_v	p_v	*known*	d_v	p_v	*known*	d_v	p_v
v_1	T	1	v_3	T	1	v_3	T	1	v_3	T	1	v_3
v_2	T	2	v_1	T	2	v_1	T	2	v_1	T	2	v_1
v_3	T	0	0	T	0	0	T	0	0	T	0	0
v_4	F	2	v_1	T	2	v_1	T	2	v_1	T	2	v_1
v_5	F	3	v_2	F	3	v_2	T	3	v_2	T	3	v_2
v_6	T	1	v_3	T	1	v_3	T	1	v_3	T	1	v_3
v_7	F	∞	0	F	3	v_4	F	3	v_4	T	3	v_4
Q:	v_4, v_5			v_5, v_7			v_7			empty		

Figure 9.16 Pseudocode for unweighted shortest-path algorithm

```
void Graph::unweighted( Vertex s )
{
    for each Vertex v
    {
        v.dist = INFINITY;
        v.known = false;
    }

    s.dist = 0;

    for( int currDist = 0; currDist < NUM_VERTICES; currDist++ )
        for each Vertex v
            if( !v.known && v.dist == currDist )
            {
                v.known = true;
                for each Vertex w adjacent to v
                    if( w.dist == INFINITY )
                    {
                        w.dist = currDist + 1;
                        w.path = v;
                    }
            }
}
```

Largest distance = |V|

v: known and at current distance

	Initial State			v_3 Dequeued			v_1 Dequeued			v_6 Dequeued		
v	*known*	d_v	p_v	*known*	d_v	p_v	*known*	d_v	p_v	*known*	d_v	p_v
v_1	F	∞	0	F	1	v_3	T	1	v_3	T	1	v_3
v_2	F	∞	0	F	∞	0	F	2	v_1	F	2	v_1
v_3	F	0	0	T	0	0	T	0	0	T	0	0
v_4	F	∞	0	F	∞	0	F	2	v_1	F	2	v_1
v_5	F	∞	0	F	∞	0	F	∞	0	F	∞	0
v_6	F	∞	0	F	1	v_3	F	1	v_3	T	1	v_3
v_7	F	∞	0	F	∞	0	F	∞	0	F	∞	0
Q:	v_3			v_1, v_6			v_6, v_2, v_4			v_2, v_4		

	v_2 Dequeued			v_4 Dequeued			v_5 Dequeued			v_7 Dequeued		
v	*known*	d_v	p_v	*known*	d_v	p_v	*known*	d_v	p_v	*known*	d_v	p_v
v_1	T	1	v_3	T	1	v_3	T	1	v_3	T	1	v_3
v_2	T	2	v_1	T	2	v_1	T	2	v_1	T	2	v_1
v_3	T	0	0	T	0	0	T	0	0	T	0	0
v_4	F	2	v_1	T	2	v_1	T	2	v_1	T	2	v_1
v_5	F	3	v_2	F	3	v_2	T	3	v_2	T	3	v_2
v_6	T	1	v_3	T	1	v_3	T	1	v_3	T	1	v_3
v_7	F	∞	0	F	3	v_4	F	3	v_4	T	3	v_4
Q:	v_4, v_5			v_5, v_7			v_7			empty		

Figure 9.16 Pseudocode for unweighted shortest-path algorithm

WSU

```
void Graph::unweighted( Vertex s )
{
    for each Vertex v
    {
        v.dist = INFINITY;
        v.known = false;
    }

    s.dist = 0;

    for( int currDist = 0; currDist < NUM_VERTICES; currDist++ )
        for each Vertex v
            if( !v.known && v.dist == currDist )
            {
                v.known = true;
                for each Vertex w adjacent to v
                    if( w.dist == INFINITY )
                    {
                        w.dist = currDist + 1;
                        w.path = v;
                    }
            }
}
```

Largest distance = |V|

v: known and at current distance

Set to known

Update adjacent vertices

	Initial State			v_3 Dequeued			v_1 Dequeued			v_6 Dequeued		
v	*known*	d_v	p_v	*known*	d_v	p_v	*known*	d_v	p_v	*known*	d_v	p_v
v_1	F	∞	0	F	1	v_3	T	1	v_3	T	1	v_3
v_2	F	∞	0	F	∞	0	F	2	v_1	F	2	v_1
v_3	F	0	0	T	0	0	T	0	0	T	0	0
v_4	F	∞	0	F	∞	0	F	2	v_1	F	2	v_1
v_5	F	∞	0	F	∞	0	F	∞	0	F	∞	0
v_6	F	∞	0	F	1	v_3	F	1	v_3	T	1	v_3
v_7	F	∞	0	F	∞	0	F	∞	0	F	∞	0
Q:	v_3			v_1, v_6			v_6, v_2, v_4			v_2, v_4		

	v_2 Dequeued			v_4 Dequeued			v_5 Dequeued			v_7 Dequeued		
v	*known*	d_v	p_v	*known*	d_v	p_v	*known*	d_v	p_v	*known*	d_v	p_v
v_1	T	1	v_3	T	1	v_3	T	1	v_3	T	1	v_3
v_2	T	2	v_1	T	2	v_1	T	2	v_1	T	2	v_1
v_3	T	0	0	T	0	0	T	0	0	T	0	0
v_4	F	2	v_1	T	2	v_1	T	2	v_1	T	2	v_1
v_5	F	3	v_2	F	3	v_2	T	3	v_2	T	3	v_2
v_6	T	1	v_3	T	1	v_3	T	1	v_3	T	1	v_3
v_7	F	∞	0	F	3	v_4	F	3	v_4	T	3	v_4
Q:	v_4, v_5			v_5, v_7			v_7			empty		

Figure 9.16 Pseudocode for unweighted shortest-path algorithm

WSU

```
void Graph::unweighted( Vertex s )
{
    for each Vertex v
    {
        v.dist = INFINITY;
        v.known = false;
    }

    s.dist = 0;

    for( int currDist = 0; currDist < NUM_VERTICES; currDist++ )
        for each Vertex v
            if( !v.known && v.dist == currDist )
            {
                v.known = true;
                for each Vertex w adjacent to v
                    if( w.dist == INFINITY )
                    {
                        w.dist = currDist + 1;
                        w.path = v;
                    }
            }
}
```

Largest distance = |V|

v: known and at current distance

Set to known

Update adjacent vertices

	Initial State			v_3 Dequeued			v_1 Dequeued			v_6 Dequeued		
v	*known*	d_v	p_v	*known*	d_v	p_v	*known*	d_v	p_v	*known*	d_v	p_v
v_1	F	∞	0	F	1	v_3	T	1	v_3	T	1	v_3
v_2	F	∞	0	F	∞	0	F	2	v_1	F	2	v_1
v_3	F	0	0	T	0	0	T	0	0	T	0	0
v_4	F	∞	0	F	∞	0	F	2	v_1	F	2	v_1
v_5	F	∞	0	F	∞	0	F	∞	0	F	∞	0
v_6	F	∞	0	F	1	v_3	F	1	v_3	T	1	v_3
v_7	F	∞	0	F	∞	0	F	∞	0	F	∞	0
Q:	v_3			v_1, v_6			v_6, v_2, v_4			v_2, v_4		

	v_2 Dequeued			v_4 Dequeued			v_5 Dequeued			v_7 Dequeued		
v	*known*	d_v	p_v	*known*	d_v	p_v	*known*	d_v	p_v	*known*	d_v	p_v
v_1	T	1	v_3	T	1	v_3	T	1	v_3	T	1	v_3
v_2	T	2	v_1	T	2	v_1	T	2	v_1	T	2	v_1
v_3	T	0	0	T	0	0	T	0	0	T	0	0
v_4	F	2	v_1	T	2	v_1	T	2	v_1	T	2	v_1
v_5	F	3	v_2	F	3	v_2	T	3	v_2	T	3	v_2
v_6	T	1	v_3	T	1	v_3	T	1	v_3	T	1	v_3
v_7	F	∞	0	F	3	v_4	F	3	v_4	T	3	v_4
Q:	v_4, v_5			v_5, v_7			v_7			empty		

Figure 9.16 Pseudocode for unweighted shortest-path algorithm

```
void Graph::unweighted( Vertex s )
{
    for each Vertex v
    {
        v.dist = INFINITY;
        v.known = false;
    }

    s.dist = 0;

    for( int currDist = 0; currDist < NUM_VERTICES; currDist++ )
        for each Vertex v
            if( !v.known && v.dist == currDist )
            {
                v.known = true;
                for each Vertex w adjacent to v
                    if( w.dist == INFINITY )
                    {
                        w.dist = currDist + 1;
                        w.path = v;
                    }
            }
}
```

Largest distance = |V|

v: known and at current distance

Set to known

Update adjacent vertices

	Initial State			v_3 Dequeued			v_1 Dequeued			v_6 Dequeued		
v	*known*	d_v	p_v	*known*	d_v	p_v	*known*	d_v	p_v	*known*	d_v	p_v
v_1	F	∞	0	F	1	v_3	T	1	v_3	T	1	v_3
v_2	F	∞	0	F	∞	0	F	2	v_1	F	2	v_1
v_3	F	0	0	T	0	0	T	0	0	T	0	0
v_4	F	∞	0	F	∞	0	F	2	v_1	F	2	v_1
v_5	F	∞	0	F	∞	0	F	∞	0	F	∞	0
v_6	F	∞	0	F	1	v_3	F	1	v_3	T	1	v_3
v_7	F	∞	0	F	∞	0	F	∞	0	F	∞	0
Q:	v_3			v_1, v_6			v_6, v_2, v_4			v_2, v_4		

	v_2 Dequeued			v_4 Dequeued			v_5 Dequeued			v_7 Dequeued		
v	*known*	d_v	p_v	*known*	d_v	p_v	*known*	d_v	p_v	*known*	d_v	p_v
v_1	T	1	v_3	T	1	v_3	T	1	v_3	T	1	v_3
v_2	T	2	v_1	T	2	v_1	T	2	v_1	T	2	v_1
v_3	T	0	0	T	0	0	T	0	0	T	0	0
v_4	F	2	v_1	T	2	v_1	T	2	v_1	T	2	v_1
v_5	F	3	v_2	F	3	v_2	T	3	v_2	T	3	v_2
v_6	T	1	v_3	T	1	v_3	T	1	v_3	T	1	v_3
v_7	F	∞	0	F	3	v_4	F	3	v_4	T	3	v_4
Q:	v_4, v_5			v_5, v_7			v_7			empty		

Figure 9.16 Pseudocode for unweighted shortest-path algorithm

General problem: setting weight=1 for each edge
→ unweighted shortest path problem

Weighted shortest path

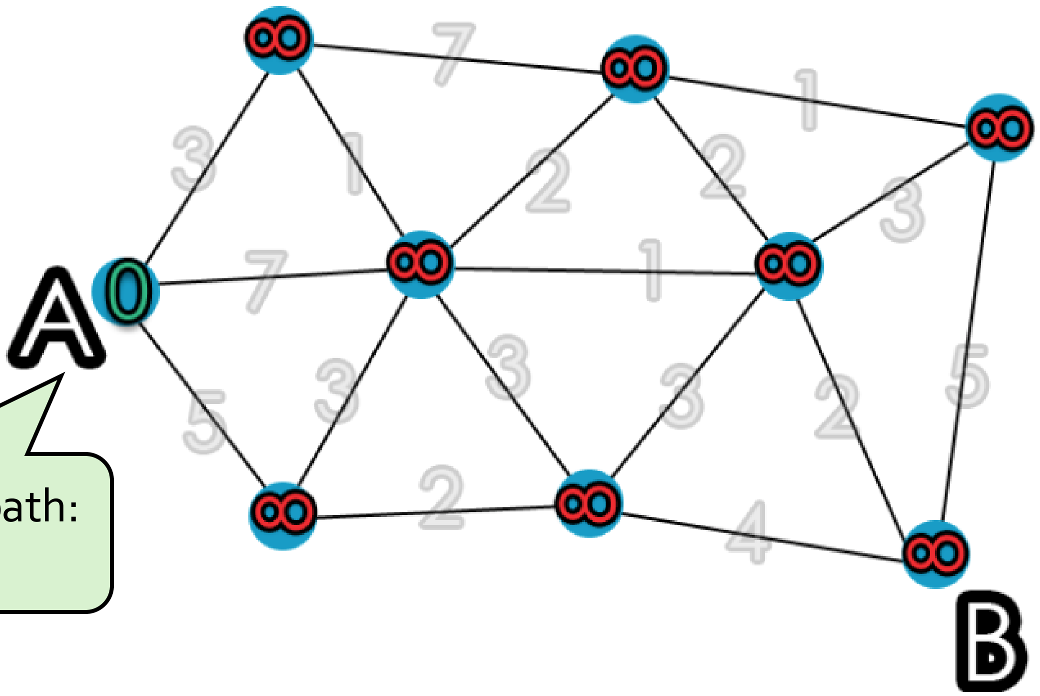
- BFS cannot solve this issue
  - A path with more edges may have lower cost
- Dijkstra's Algorithm takes into account edge weights for finding paths
- Don't just keep the raw distance, but sum up the cost to get there
- Greedy algorithm - follow lowest cost path first every time.
  - Queue is sorted by shortest path of the nodes not explored so far (priority queue time!)
- Again, keep a table of the vertices and their costs. Start them at INF
  - If a node popped off of the queue shortens another node's path
    then benefits cascade down the chain automagically
- Heavily used in network routing and shortest path network choices

Weighted shortest path

- Algorithm
- Put all vertices in a priority queue and the initial distance is INF except the source
- Select vertex v from the queue which has the smallest distance to v (denoted by d_v) among unknown vertices
  - Declares shortest path from s to v is known
  - For each adjacent node (denoted by w) of v
    - If unweighted graph:
      - set d_w = d_v + 1 (if d_w = INF, aka not visited yet), thus this lowers value of d_w if v was shorter path
    - If weighted graph:
      - Set d_w = d_v + c{v,w} (if this can reduce dw)

Weighted shortest path

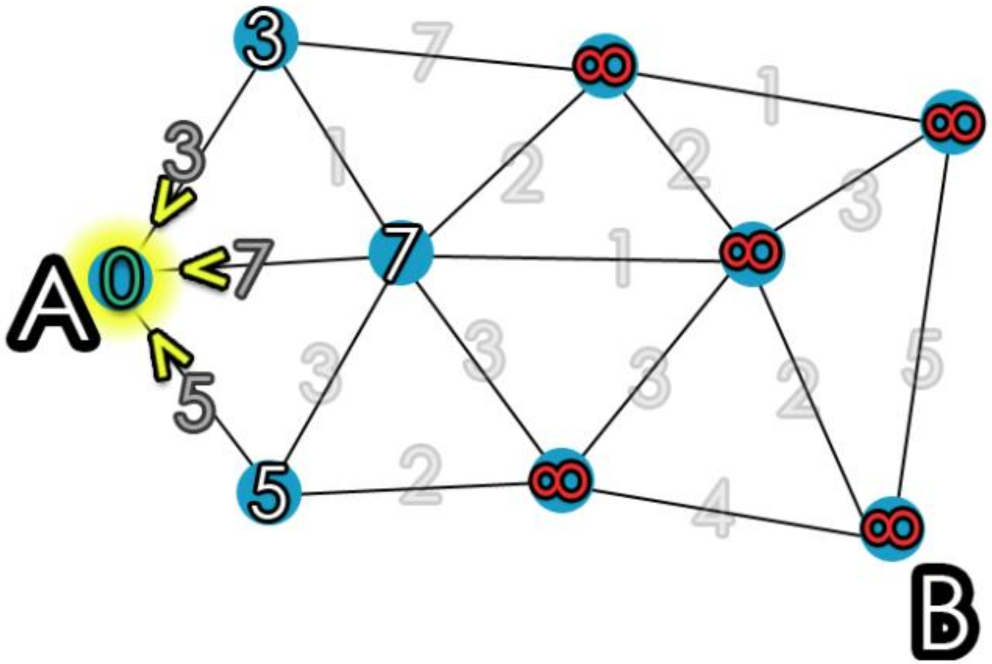
- 1. Initialize distances according to the algorithm.
  - All vertexes are stored in a min-heap (priority queue) according to the current distance to A. The initial distance is INF



Find the shortest path:
A → B

Weighted shortest path

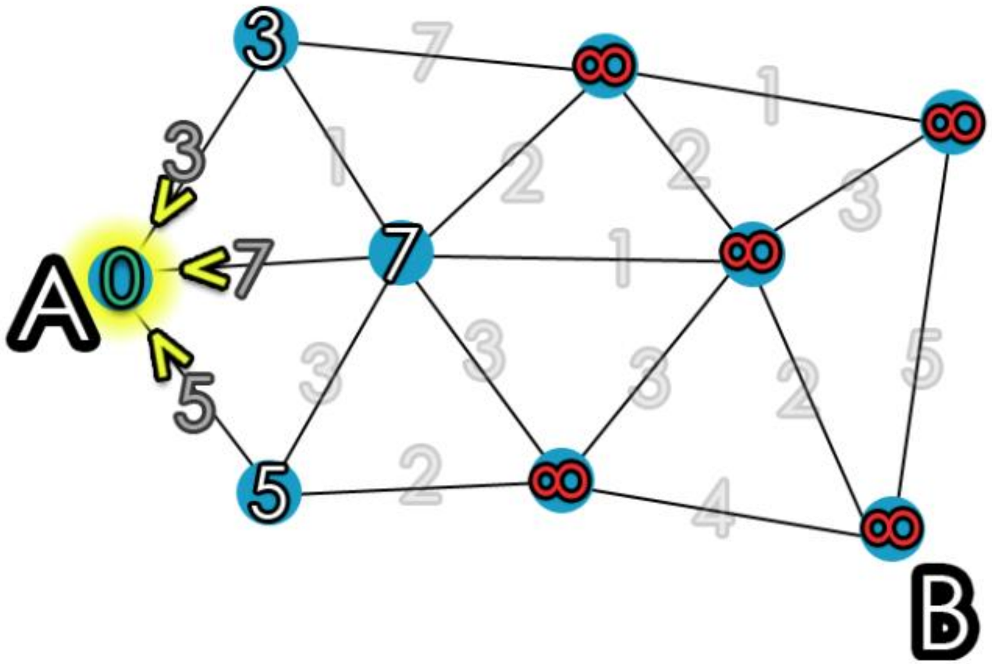
- 2. Pick first node and calculate distances to adjacent nodes.



WSU

Weighted shortest path

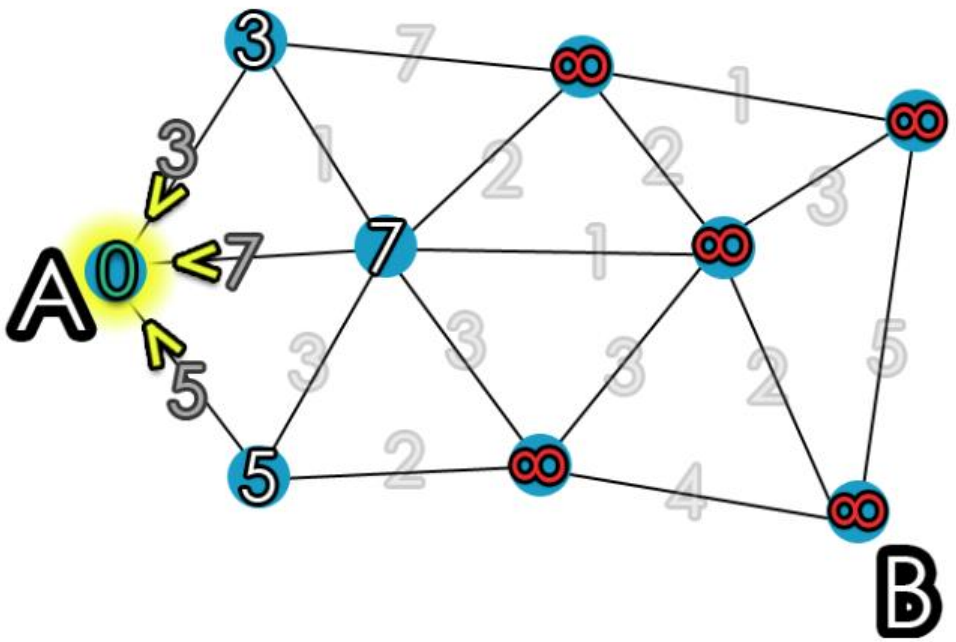
- 3. Pick next node with minimal distance (deleteMin()); repeat adjacent node distance calculations (decrease()).

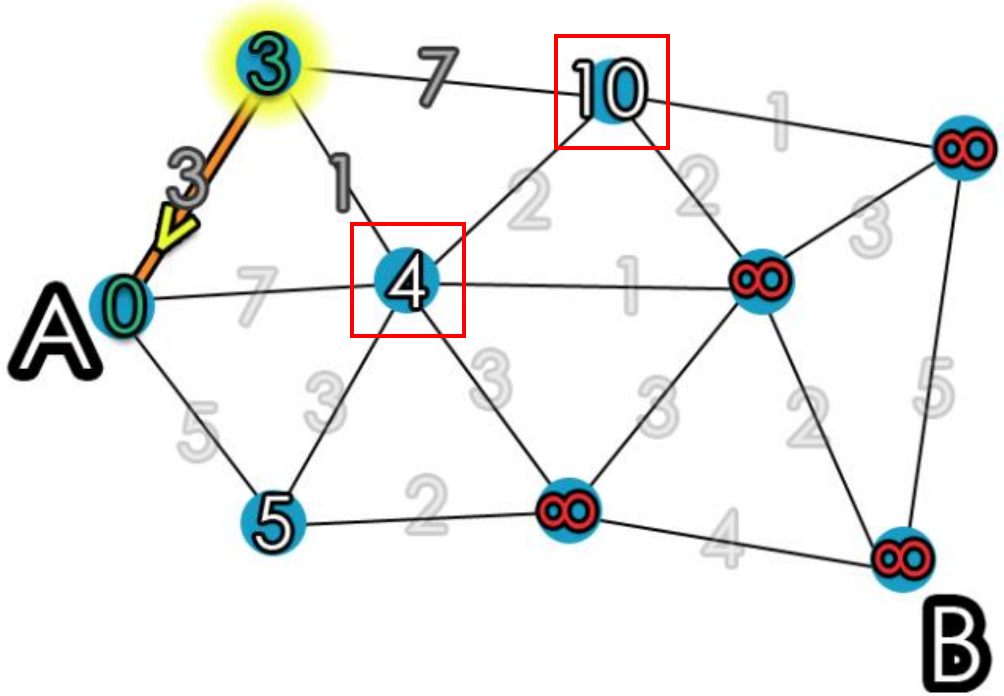


Graph

Weighted shortest path

- 3. Pick next node with minimal distance (deleteMin()); repeat adjacent node distance calculations (decrease()).

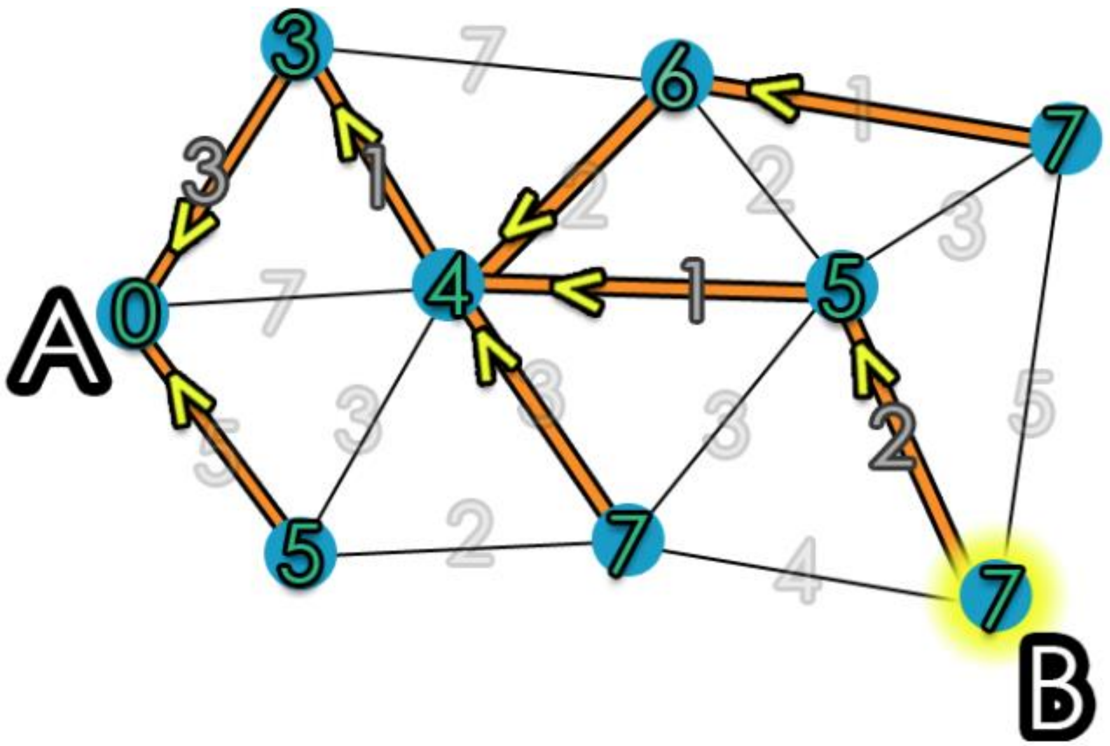




WSU

Weighted shortest path

- 4. Repeat Step 3, until no unknown vertices left in the queue



Code for Dijkstra’s algorithm

General problem: setting weight=1 for each edge
→ unweighted shortest path problem

```
struct Vertex
{
    List      adj;      // Adjacency list
    bool      known;
    DistType  dist;     // DistType is probably int
    Vertex    path;     // Probably Vertex *, as mentioned above
        // Other data and member functions as needed
};
```

Figure 9.29 Vertex class for Dijkstra’s algorithm (pseudocode)

Code for orithm

WSU

```
void Graph::dijkstra( Vertex s )
{
    for each Vertex v
    {
        v.dist = INFINITY;
        v.known = false;
    }

    s.dist = 0;

    while( there is an unknown distance vertex )
    {
        Vertex v = smallest unknown distance vertex;

        v.known = true;

        for each Vertex w adjacent to v
            if( !w.known )
            {
                DistType cvw = cost of edge from v to w;

                if( v.dist + cvw < w.dist )
                {
                    // Update w
                    decrease( w.dist to v.dist + cvw );
                    w.path = v;
                }
            }
    }
}
```

Initialization

Expand to vertices layer by layer to the farthest one

Figure 9.31 Pseudocode for Dijkstra's algorithm

Dijkstra’s summary

- Shortest path algorithms are incredibly valuable
- Dijkstra’s Algorithm is fast and efficient
  - By default, cannot work if there’s a negative cycle in the graph
  - Works in $O(|E| + |V| \log |V|)$ time
    - Explore all $|E|$ edges
    - Each round in the loop causes a percolation-up in the priority queue (minheap) of $|V|$ vertices - or $\log |V|$
    - $O(|E|)$ space needed to store all of the edges in the queue