# Data Structures in Java

Lecture 17: Introduction to Graphs.

11/13/2019

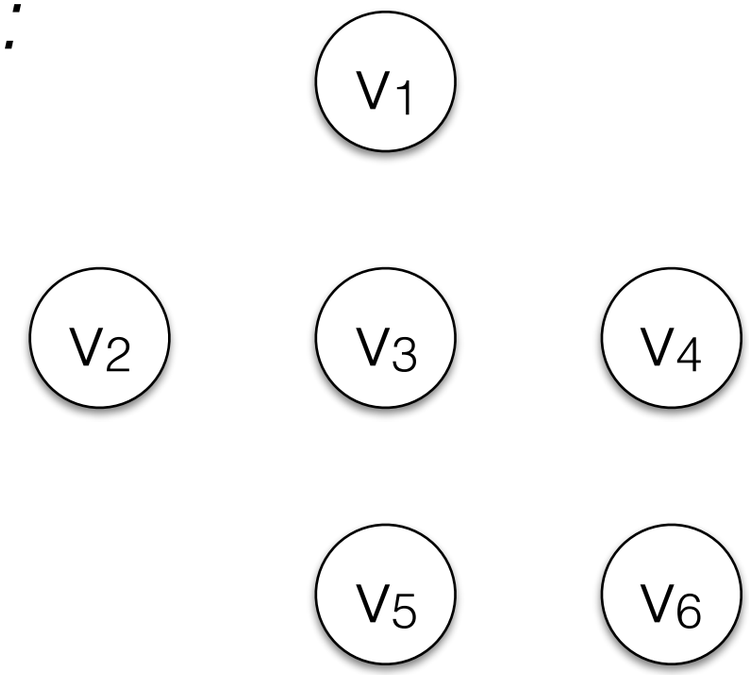
Daniel Bauer

# Graphs

- A **Graph** is a pair of two sets *G=(V,E):*
  - V: the set of **vertices** (or **nodes**)
  - E: the set of **edges**.
    - each edge is a pair (v,w) where v,w ∈ V

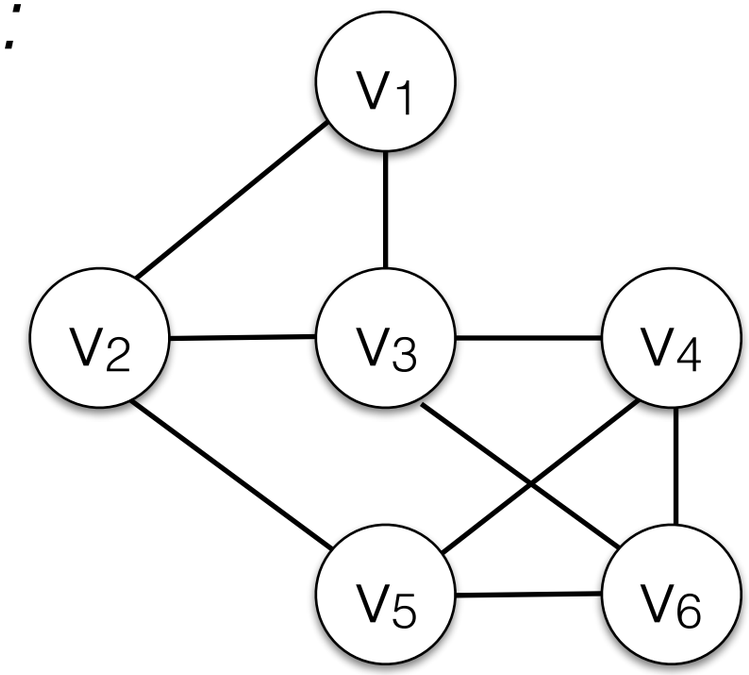
# Graphs

- A **Graph** is a pair of two sets *G=(V,E):*
  - V: the set of **vertices** (or **nodes**)
  - E: the set of **edges**.
    - each edge is a pair (v,w) where v,w ∈ V



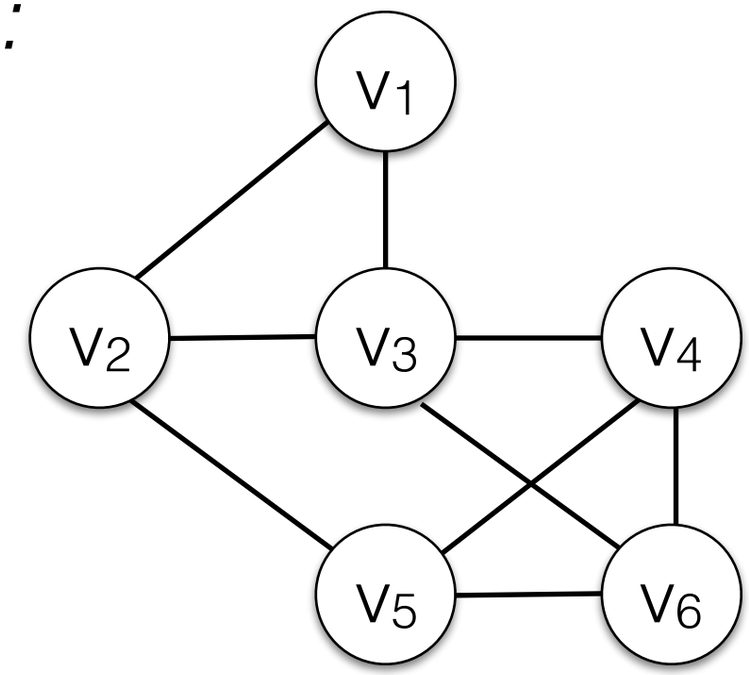
# Graphs

- A **Graph** is a pair of two sets *G=(V,E):*
  - V: the set of **vertices** (or **nodes**)
  - E: the set of **edges**.
    - each edge is a pair (v,w) where v,w ∈ V



# Graphs

- A **Graph** is a pair of two sets *G=(V,E):*
  - V: the set of **vertices** (or **nodes**)
  - E: the set of **edges**.
    - each edge is a pair (v,w) where v,w ∈ V

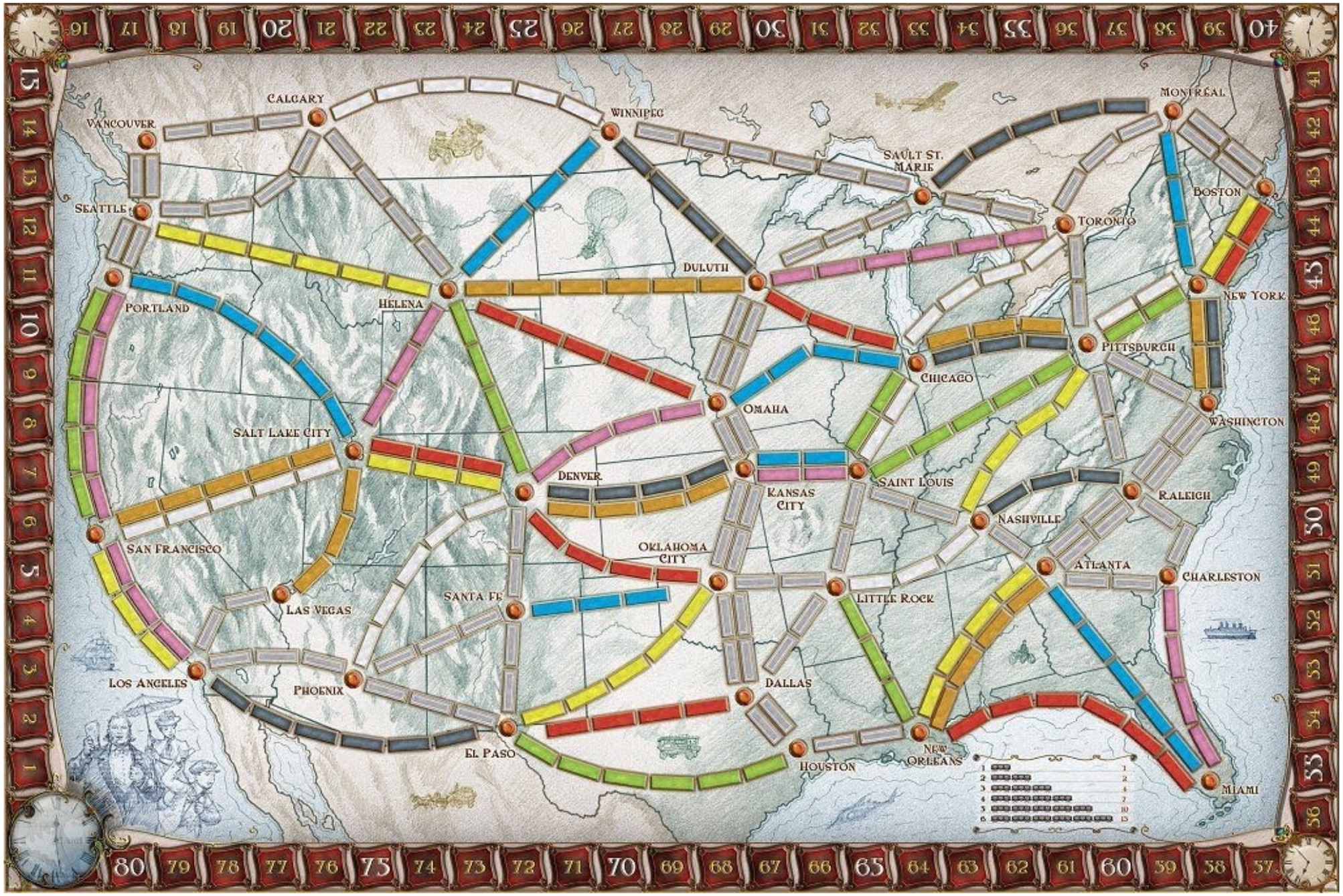



$$V = \{v_1, v_2, v_3, v_4, v_5, v_6\}$$
$$E = \{(v_1, v_2), (v_1, v_3), (v_2, v_3), (v_2, v_5), (v_3, v_4), (v_3, v_6), (v_4, v_5), (v_4, v_6), (v_5, v_6)\}$$

# Graphs in Computer Science

- Graphs are used to model all kinds of relational data.
- General purpose algorithms make it possible to solve problems on these models.
  - Shortest Paths, Spanning Tree, Finding Cliques, Strongly Connected Components, Network Flow, Graph Coloring, Minimum Edge/Vertex Cover, Graph Partitioning, …

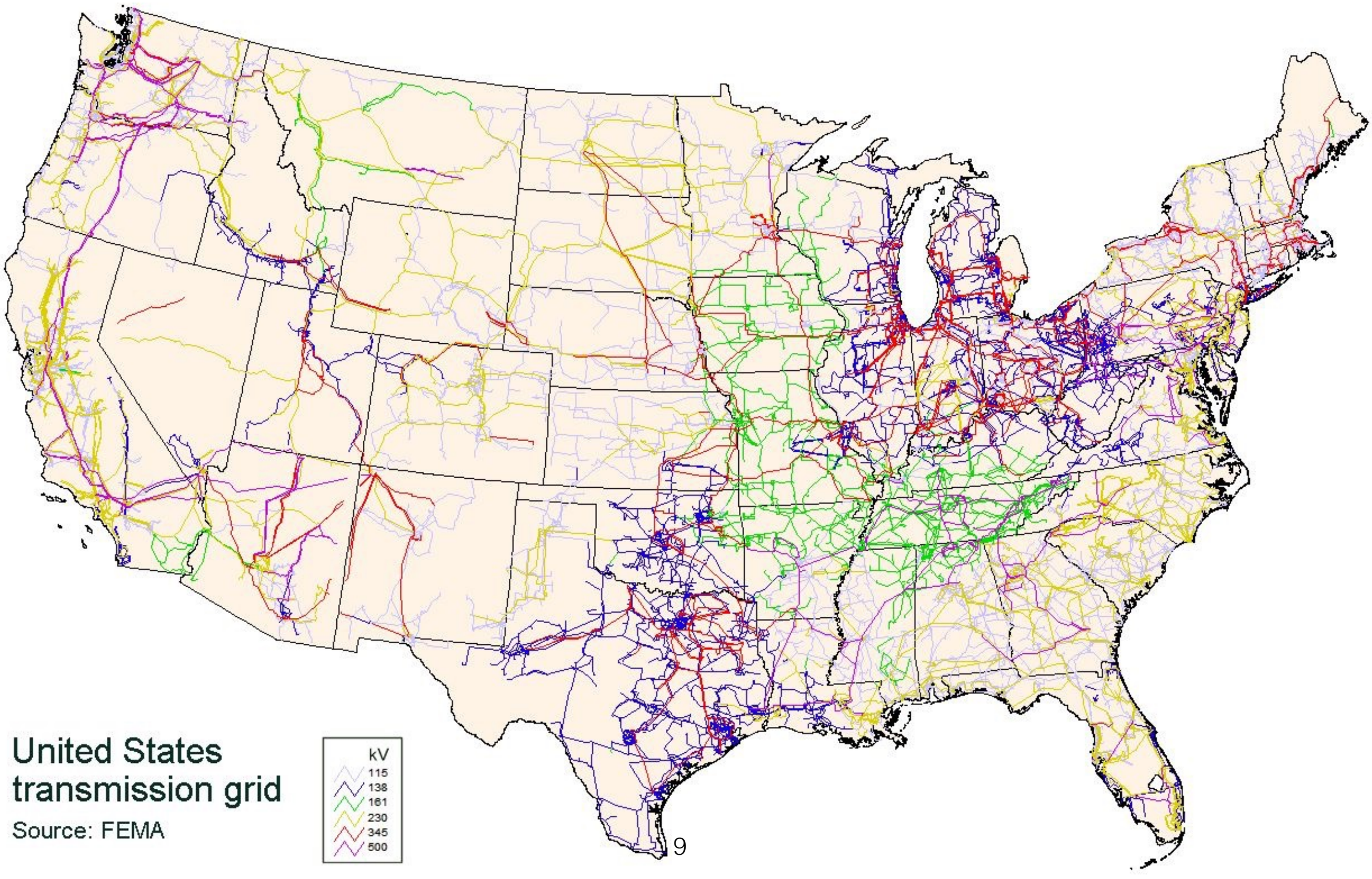
# Rail Network




Source: Days of WonderVideo Games

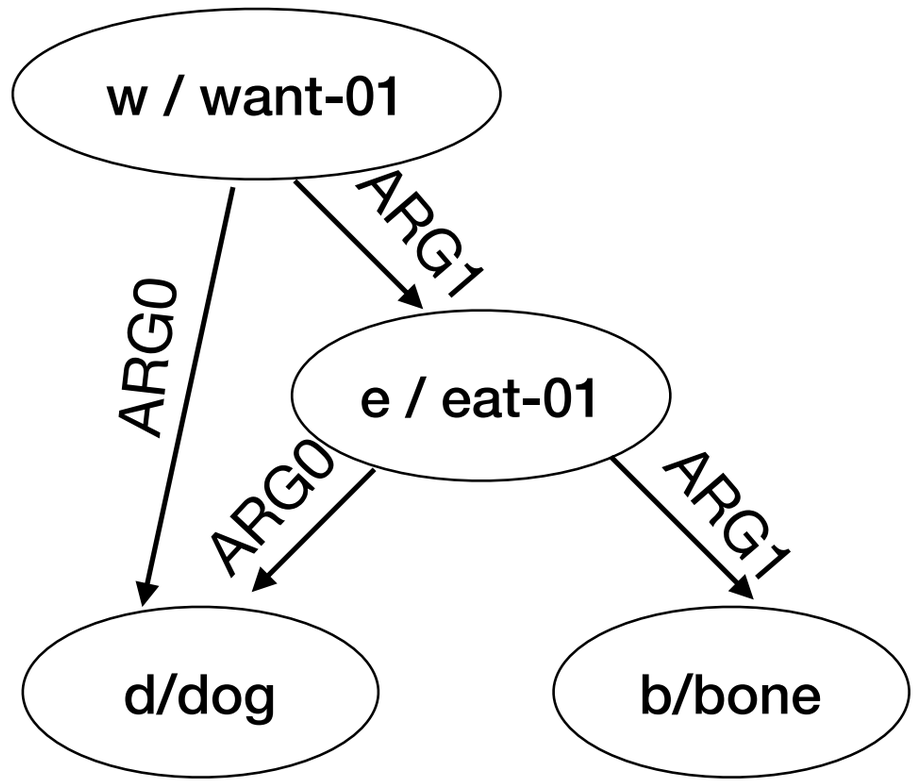
# Social Networks

# US Power Grid



# Graph-Based Representation of Sentence Meaning

*The dog wants to eat a bone.*



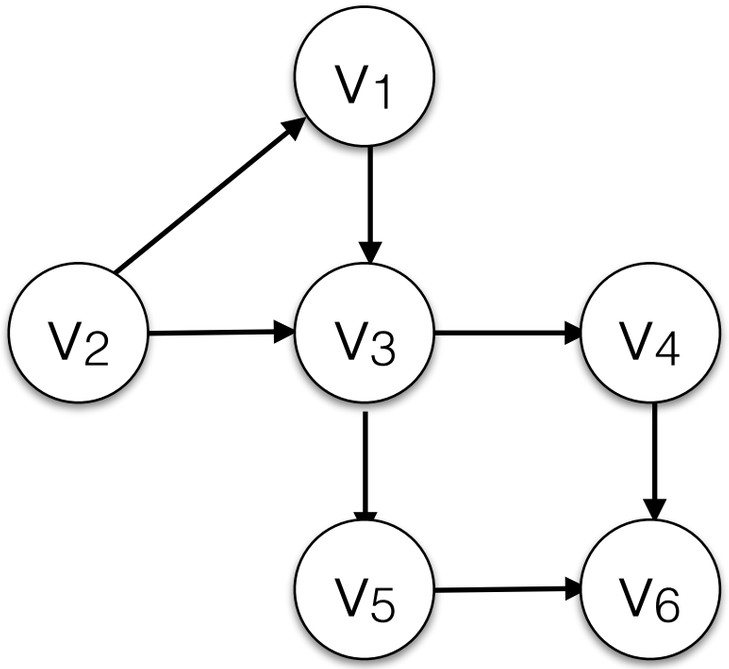

- E.g. "Abstract Meaning Representation"

# Edges

- Graphs may be **directed** or **undirected**.
  - In directed graphs, the edge pairs are ordered.

$V = \{v_1, v_2, v_3, v_4, v_5, v_6\}$

$E = \{(v_1, v_3), (v_2, v_1), (v_2, v_3), (v_3, v_4), (v_3, v_5), (v_4, v_6), (v_5, v_6)\}$




directed graph

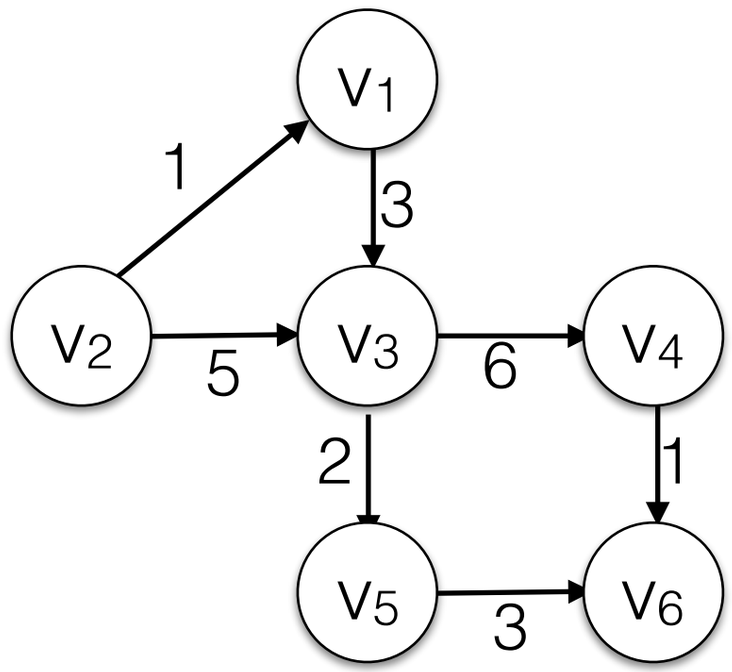
# Edges

- Graphs may be **directed** or **undirected**.
  - In directed graphs, the edge pairs are ordered.

- Edges often have some weight or cost associated with them (**weighted** graphs).

$V = \{v_1, v_2, v_3, v_4, v_5, v_6\}$

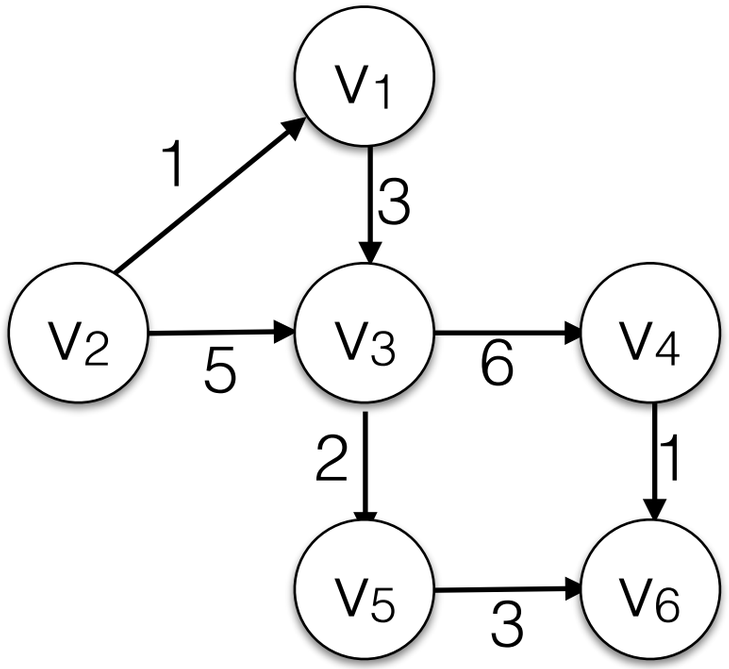
$E = \{(v_1, v_3), (v_2, v_1), (v_2, v_3), (v_3, v_4), (v_3, v_5), (v_4, v_6), (v_5, v_6)\}$




directed and weighted graph

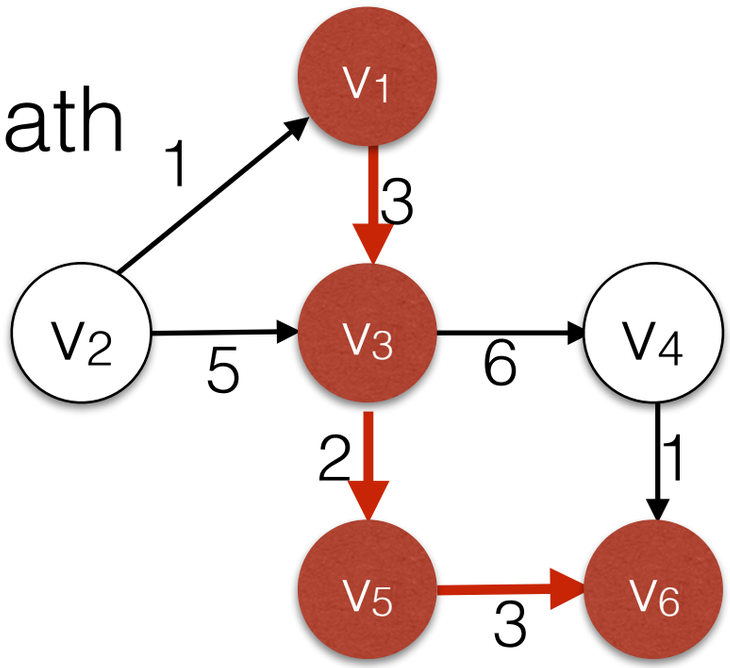
# Paths

- Vertex w is **adjacent** to vertex v iff $(w,v) \in E$.
- A **path** is a sequence of vertices $w_1, w_2, \ldots, w_k$ such that $(w_i, w_{i+1}) \in E$.



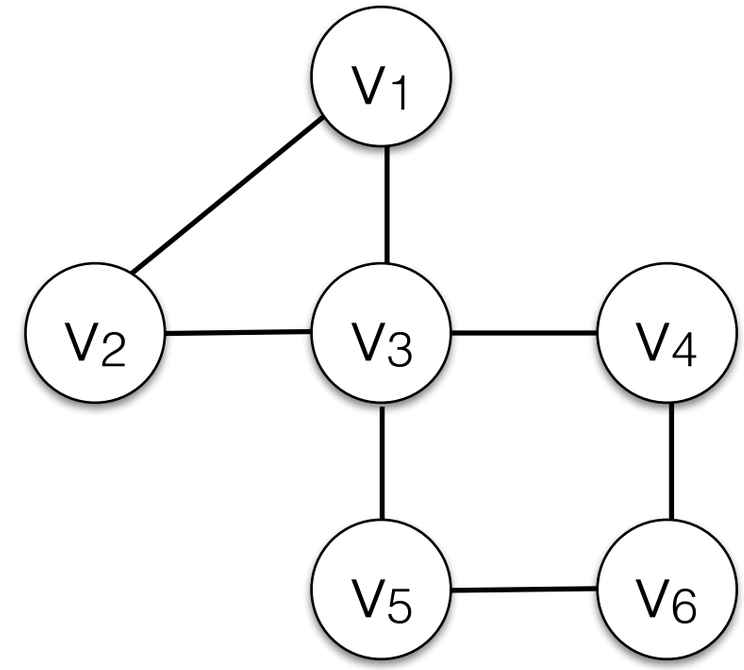
# Paths

- Vertex w is **adjacent** to vertex v iff $(w,v) \in E$.
- A **path** is a sequence of vertices $w_1, w_2, \ldots, w_k$ such that $(w_i, w_{i+1}) \in E$.
- **length** of a path:
  k-1 = number of edges on path
- **cost** of a path:
  Sum of all edge costs.



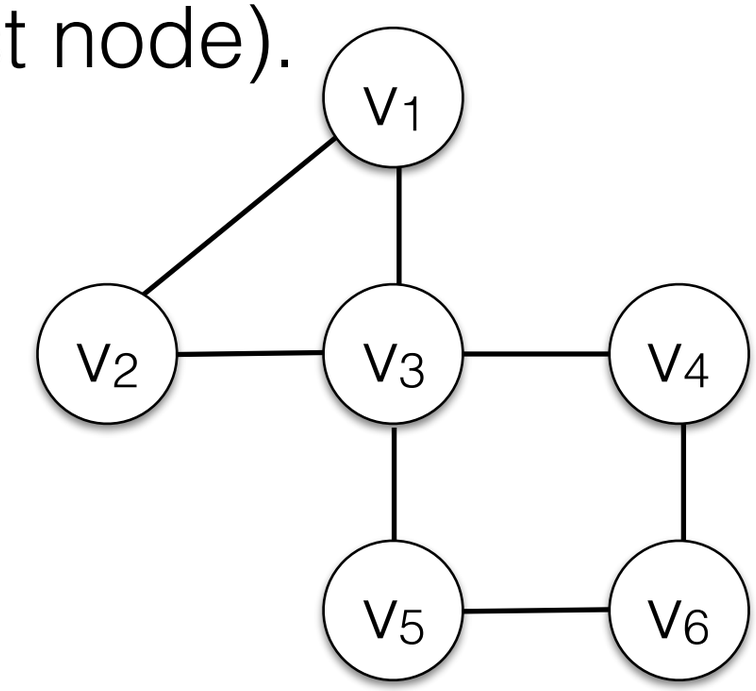

Path from $v_1$ to $v_6$, length 3, cost 8
$(v_1, v_3), (v_3, v_5), (v_5, v_6)$

# Simple Paths



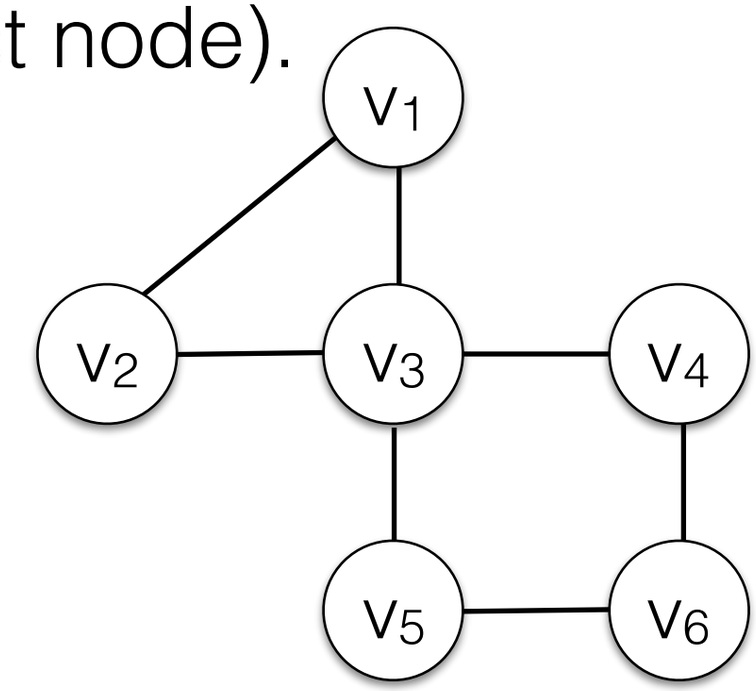
# Simple Paths

- A **simple path** is a path that contains every node only once (except possibly the first and last node).



# Simple Paths

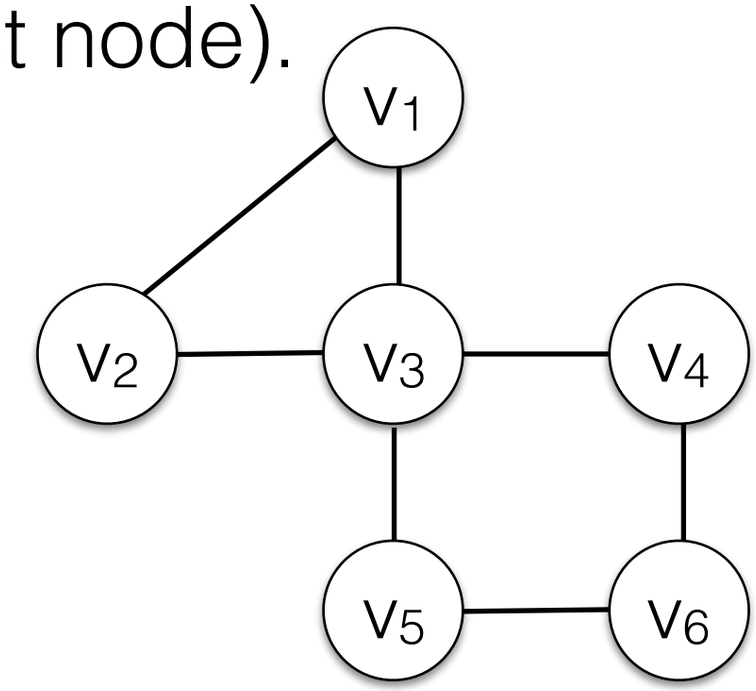
- A **simple path** is a path that contains every node only once (except possibly the first and last node).




- ($v_2$, $v_3$, $v_4$, $v_6$, $v_5$,$v_3$, $v_1$) is a path but not a simple path.

# Simple Paths

- A **simple path** is a path that contains every node only once (except possibly the first and last node).

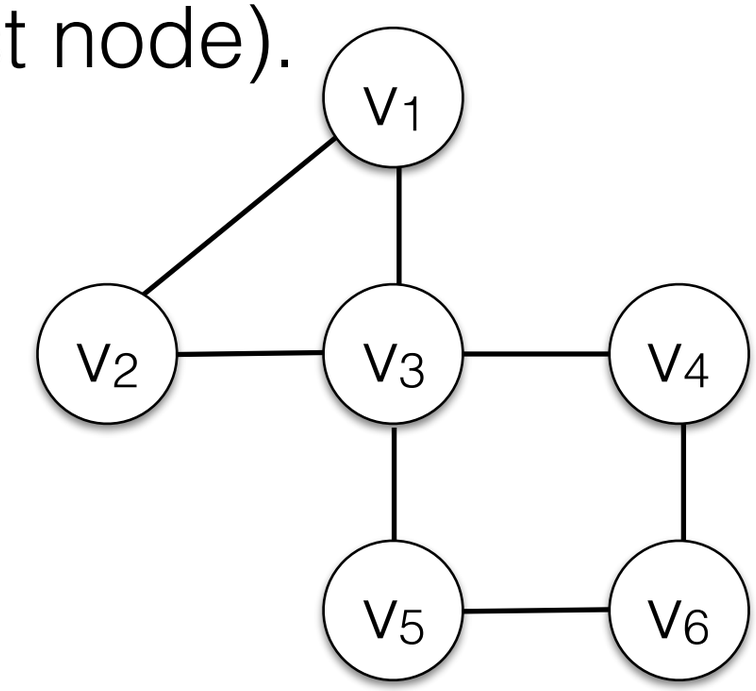



- ($v_2$, $v_3$, $v_4$, $v_6$, $v_5$,$v_3$, $v_1$) is a path but not a simple path.

- There are only two simple paths between $v_2$ and $v_1$: ($v_2$, $v_1$) and ($v_2$, $v_3$, $v_1$)

# Simple Paths

- A **simple path** is a path that contains every node only once (except possibly the first and last node).




- ($v_2$, $v_3$, $v_4$, $v_6$, $v_5$,$v_3$, $v_1$) is a path but not a simple path.

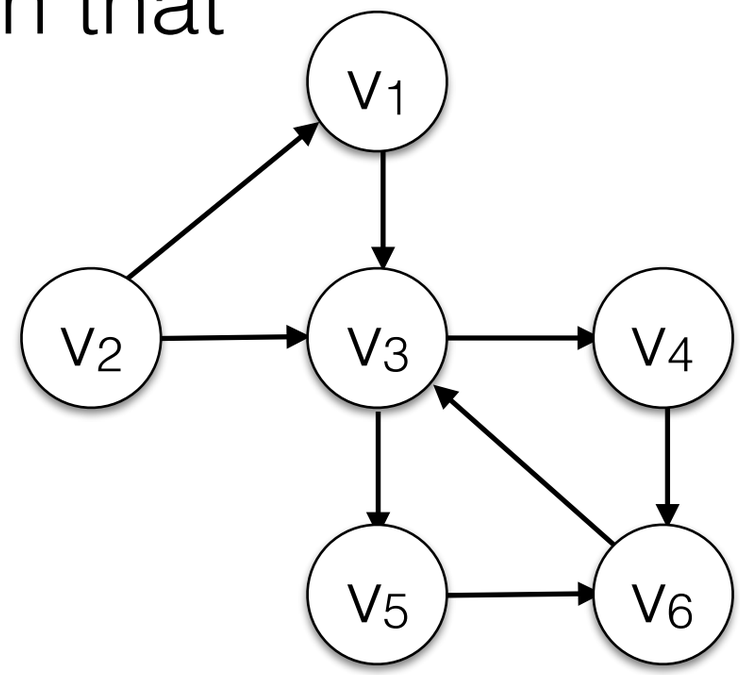
- There are only two simple paths between $v_2$ and $v_1$: ($v_2$, $v_1$) and ($v_2$, $v_3$, $v_1$)

- ($v_1$, $v_3$, $v_2$, $v_1$) is a simple path.

# Cycles in Directed Graphs

- A **cycle** is a path (of length > 1) such that $w_1 = w_k$

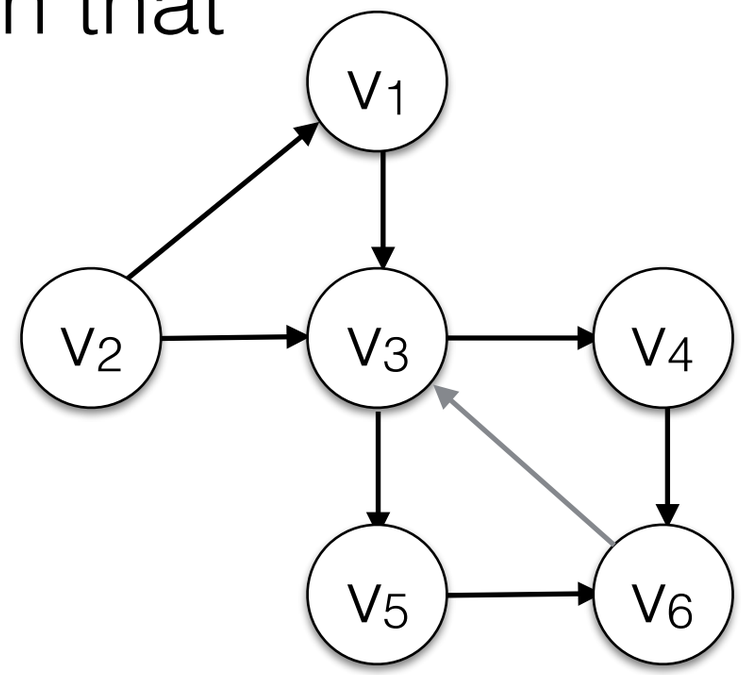
- $(v_3, v_4, v_6, v_3)$ is a cycle.



# Cycles in Directed Graphs

- A **cycle** is a path (of length > 1) such that $w_1 = w_k$

- $(v_3, v_4, v_6, v_3)$ is a cycle.

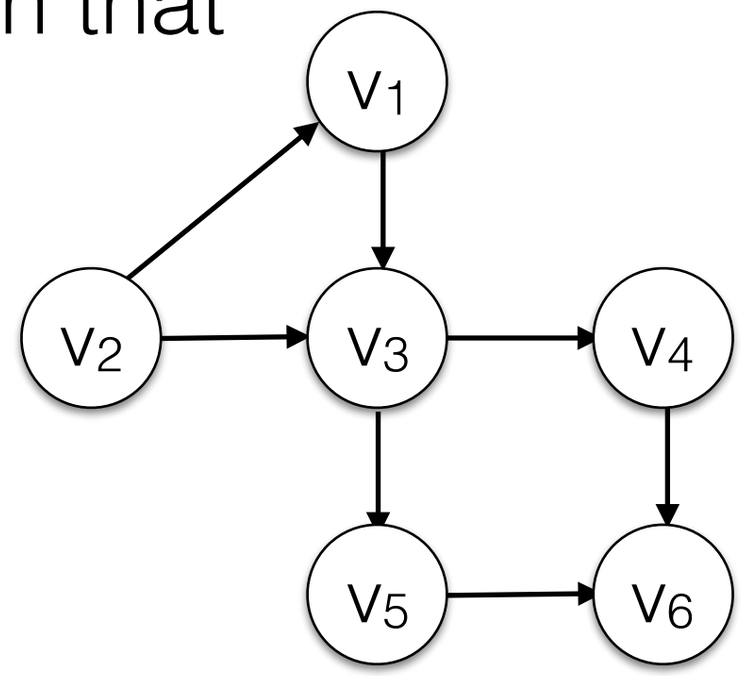



- A **Directed Acyclic Graph (DAG)** is a directed graph that contains no cycles.

# Cycles in Directed Graphs

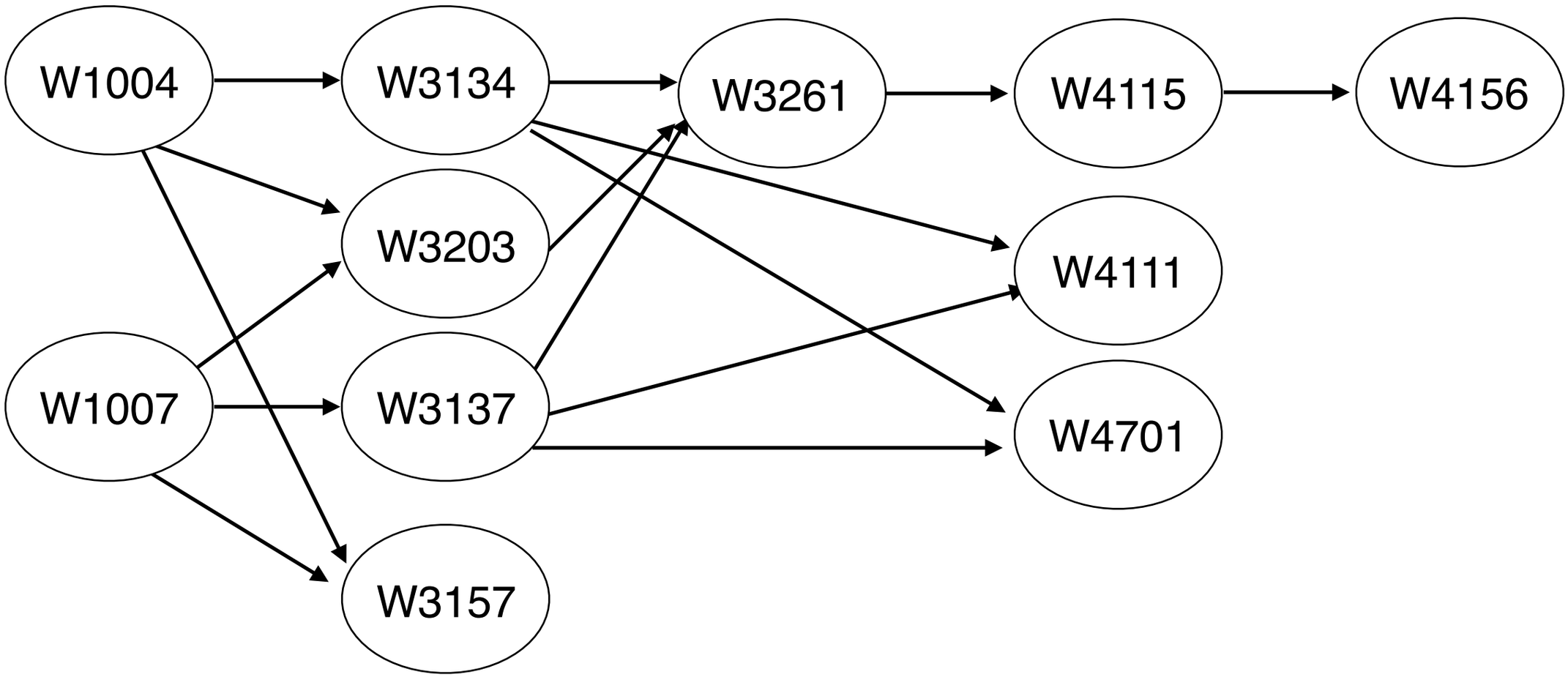
- A **cycle** is a path (of length > 1) such that $w_1 = w_k$

- $(v_3, v_4, v_6, v_3)$ is a cycle.




- A **Directed Acyclic Graph (DAG)** is a directed graph that contains no cycles.

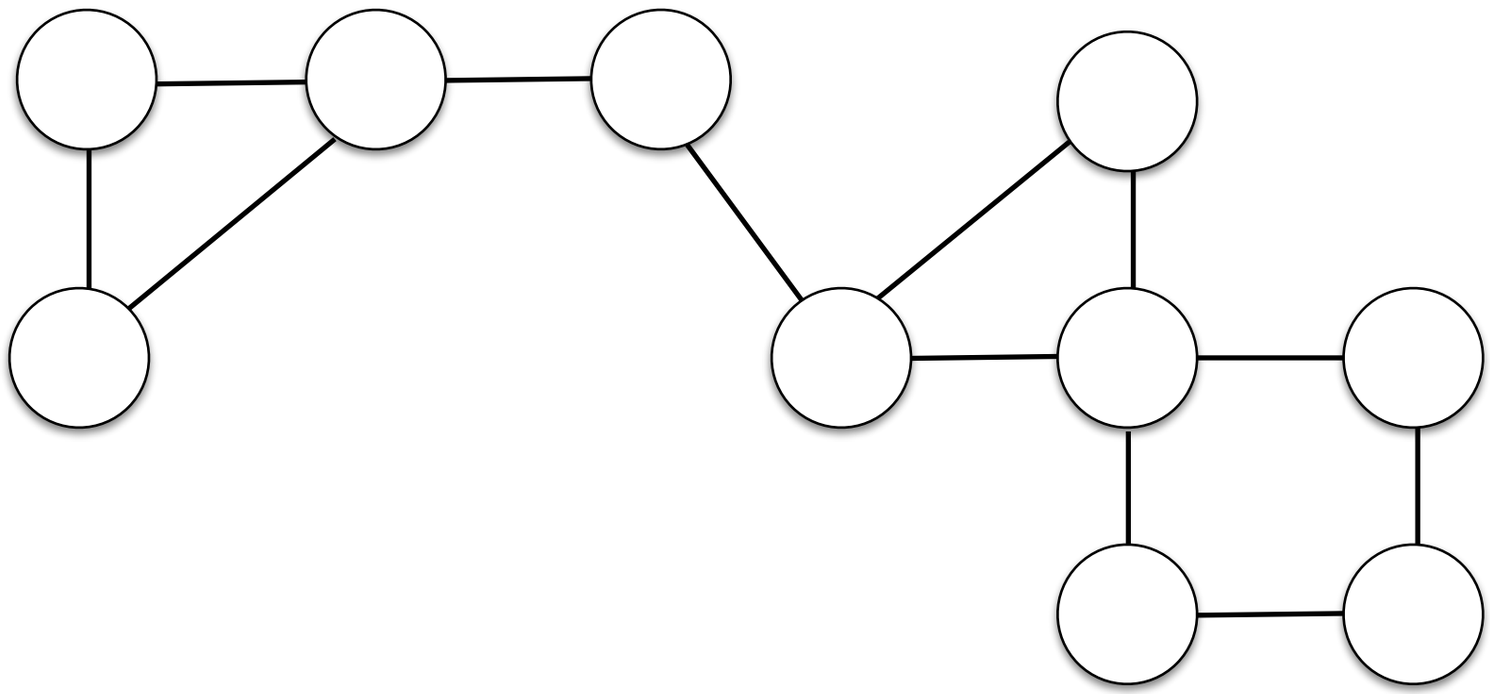
# Columbia CS Course Prerequisites as a DAG




Please do not use this figure for program planning!
It's inaccurate!

# Connectivity

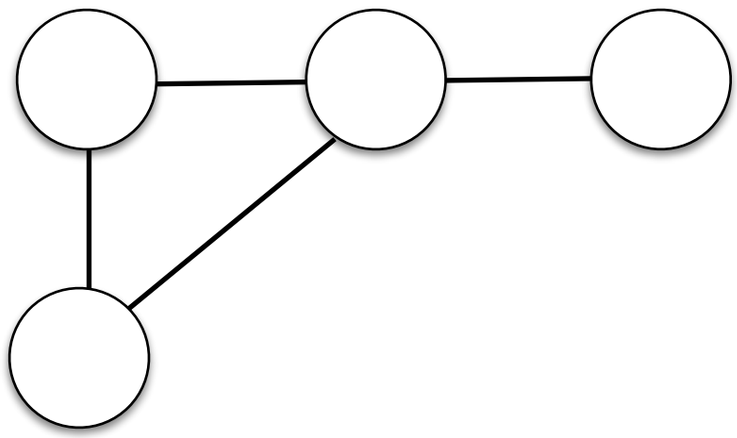
- An undirected graph is **connected** if there is a path from every vertex to every other vertex.



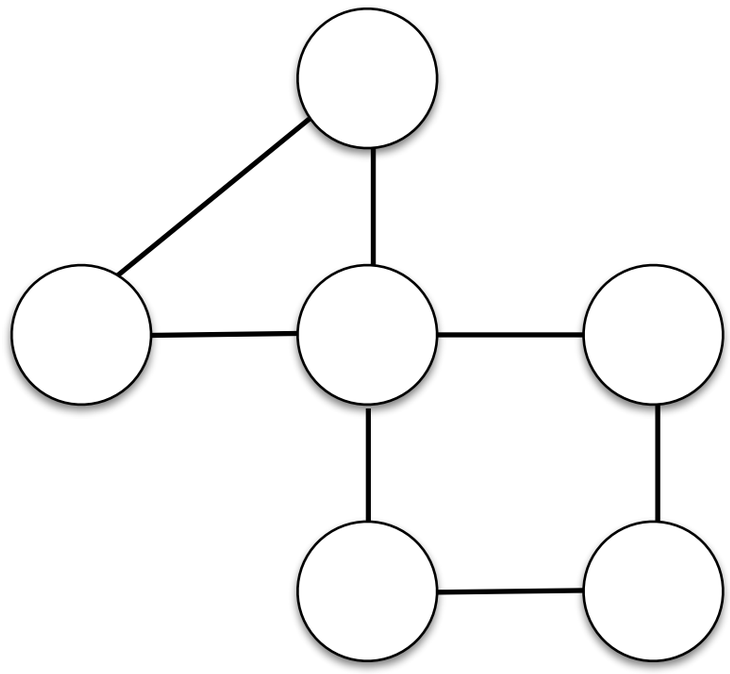
connected graph

# Connectivity

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

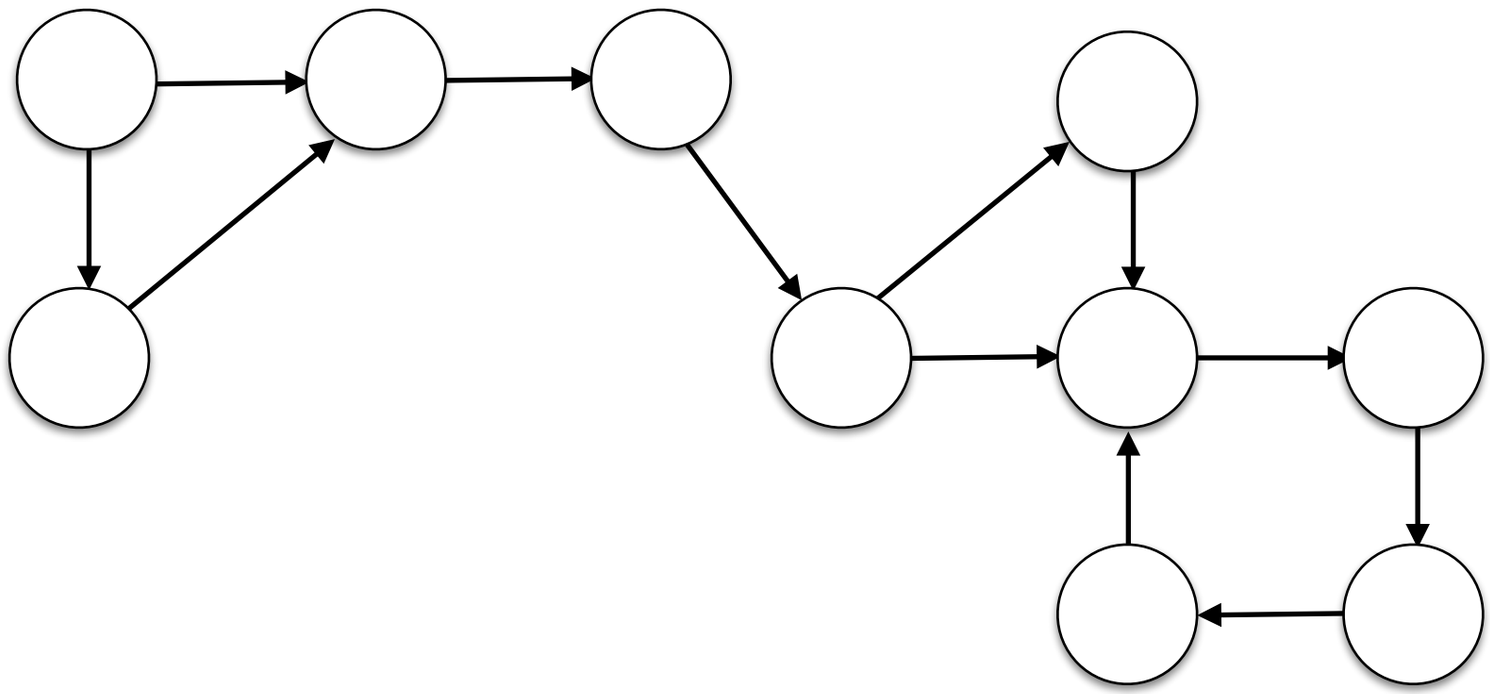


unconnected graph



# Connectivity in Directed Graphs

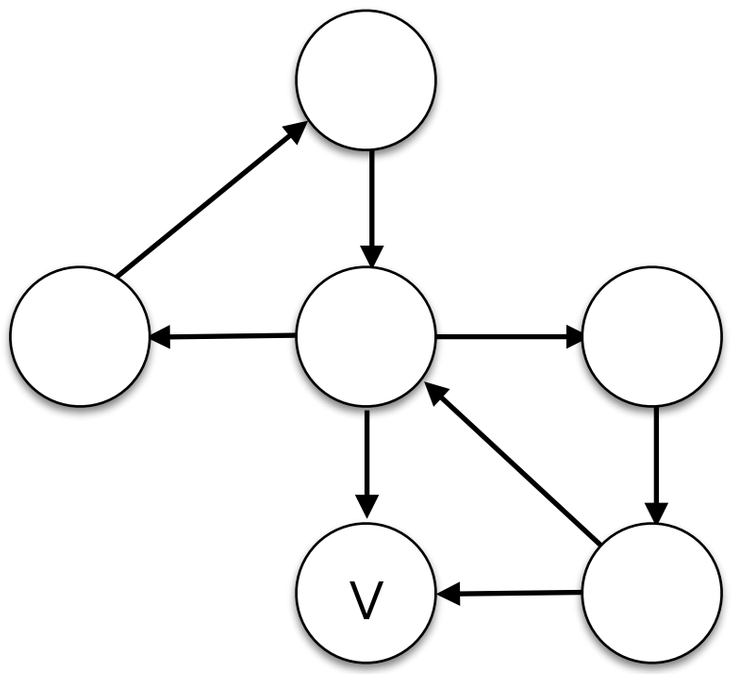
- A directed graph is **weakly connected** if there is an *undirected* path from every vertex to every other vertex.



weakly connected graph

# Strongly Connected Graphs

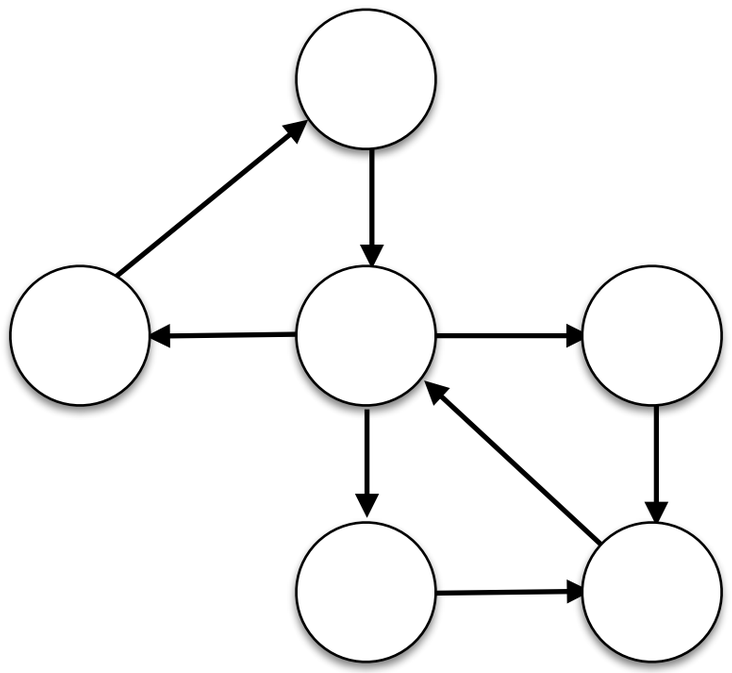
- A directed graph is **strongly connected** if there is a path from every vertex to every other vertex.




Weakly connected, but not strongly connected (no other vertex can be reached from v).

# Strongly Connected Graphs

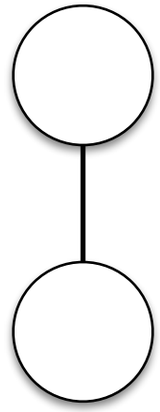
- A directed graph is **strongly connected** if there is a path from every vertex to every other vertex.



strongly connected

# Complete Graphs

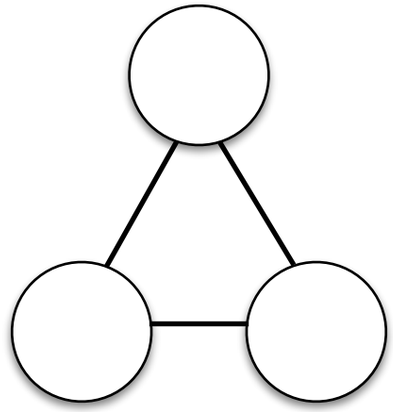
- A **complete graph** has edges between every pair of vertices.



N=2

# Complete Graphs

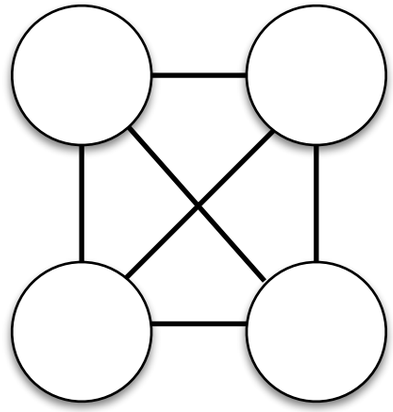
- A **complete graph** has edges between every pair of vertices.



N=3

# Complete Graphs

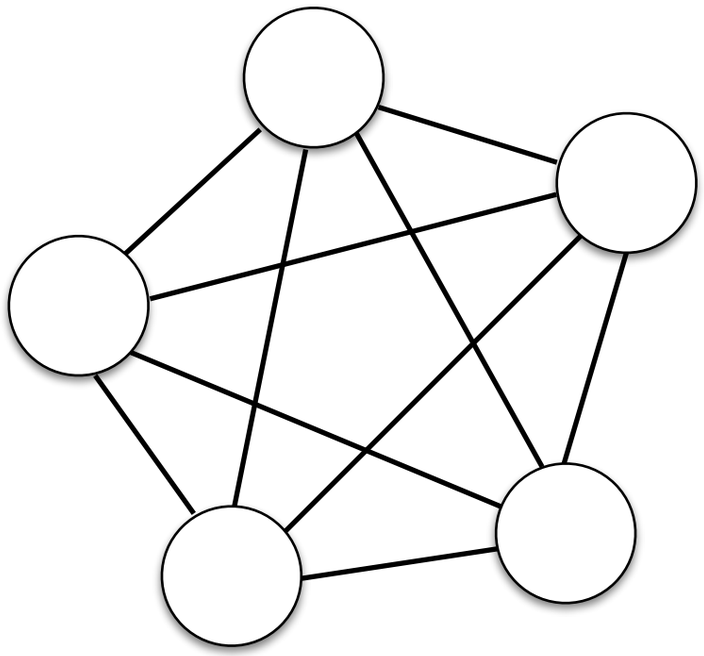
- A **complete graph** has edges between every pair of vertices.



N=4

# Complete Graphs

- A **complete graph** has edges between every pair of vertices.

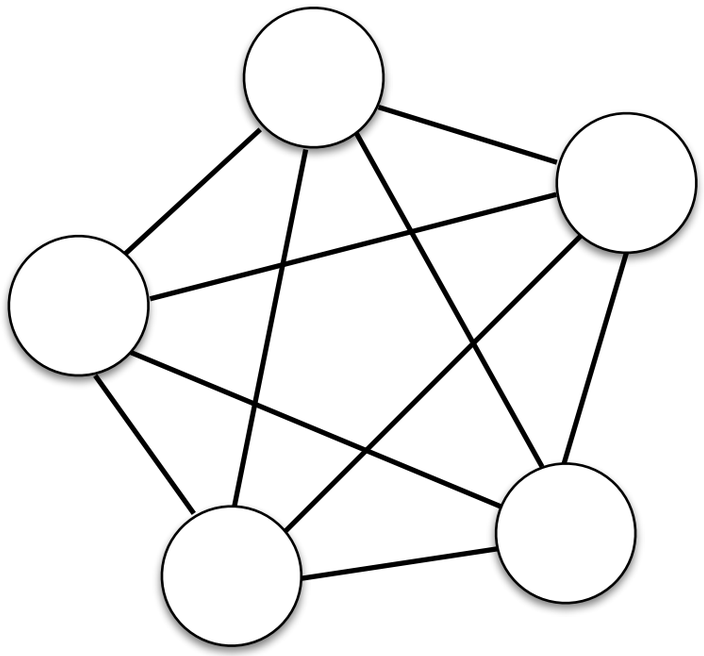


N=5

How many edges are there in a complete graph of size N?

# Complete Graphs

- A **complete graph** has edges between every pair of vertices.



N=5

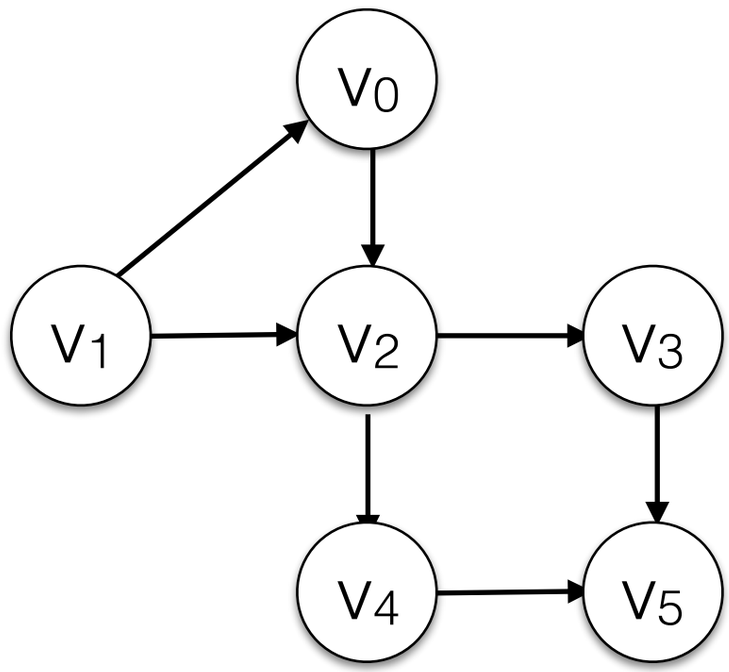
How many edges are there in a complete graph of size N?

$$\sum_{i=1}^{N-1} i = \frac{N \cdot (N-1)}{2}$$

# Representing Graphs

- Represent graph G = (E,V), option 1:
  - N x N **Adjacency Matrix** represented as 2-dimensional `Boolean[][]`.
  - A[u][v] = `true` if (u,v) ∈ E, else false

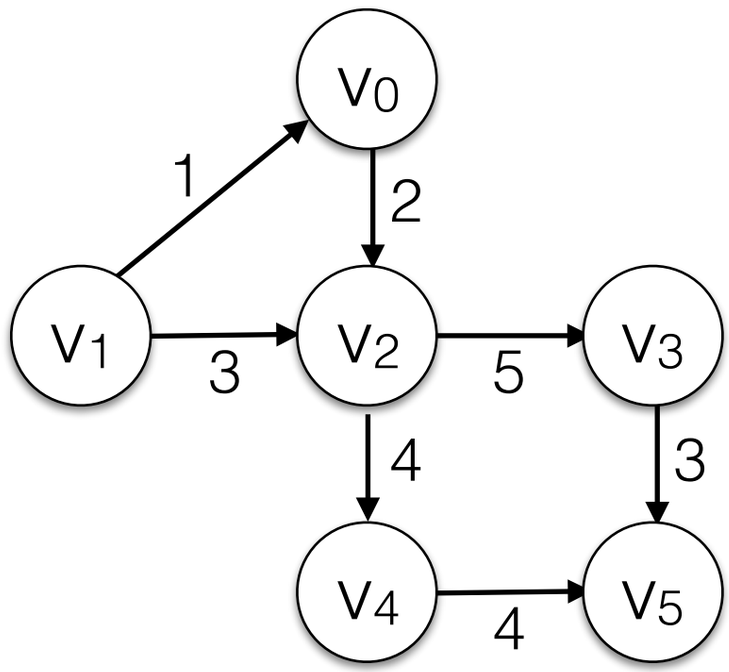
| | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| 0 | f | f | t | f | f | f |
| 1 | t | f | t | f | f | f |
| 2 | f | f | f | t | t | f |
| 3 | f | f | f | f | f | t |
| 4 | f | f | f | f | f | t |
| 5 | f | f | f | f | f | f |



# Representing Graphs

- Represent graph G = (E,V), option 1:
  - N x N **Adjacency Matrix** represented as 2-dimensional `Integer[][]`.
  - A[u][v] = cost(u,v) if (u,v) ∈ E, else ∞

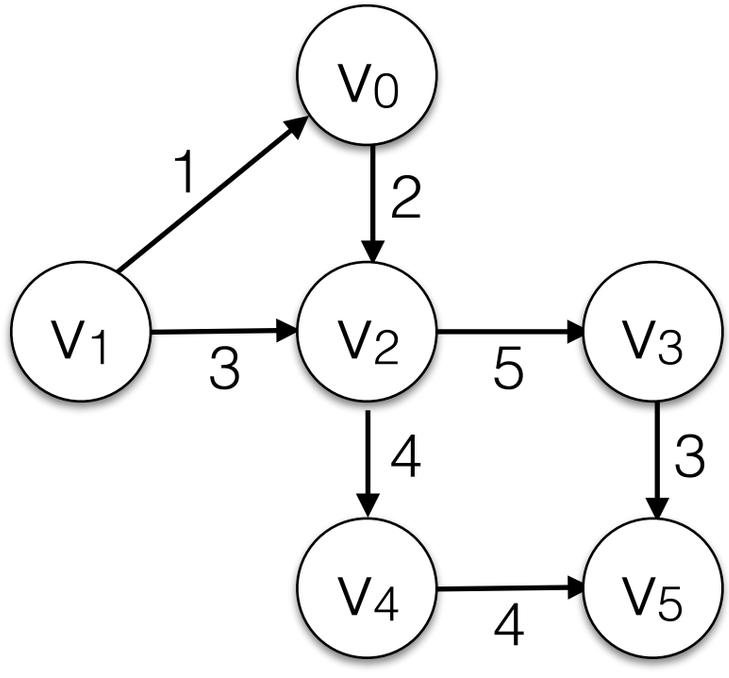
| | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| 0 | ∞ | ∞ | 2 | ∞ | ∞ | ∞ |
| 1 | 1 | ∞ | 3 | ∞ | ∞ | ∞ |
| 2 | ∞ | ∞ | ∞ | 5 | 4 | ∞ |
| 3 | ∞ | ∞ | ∞ | ∞ | ∞ | 3 |
| 4 | ∞ | ∞ | ∞ | ∞ | ∞ | 4 |
| 5 | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ |



# Representing Graphs

- Problem of Adjacency Matrix representation:
  - For **sparse** graphs (that contain much less than $|V|^2$ edges), a lot of array space is wasted.

| | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| 0 | ∞ | ∞ | 2 | ∞ | ∞ | ∞ |
| 1 | 1 | ∞ | 3 | ∞ | ∞ | ∞ |
| 2 | ∞ | ∞ | ∞ | 5 | 4 | ∞ |
| 3 | ∞ | ∞ | ∞ | ∞ | ∞ | 3 |
| 4 | ∞ | ∞ | ∞ | ∞ | ∞ | 4 |
| 5 | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ |



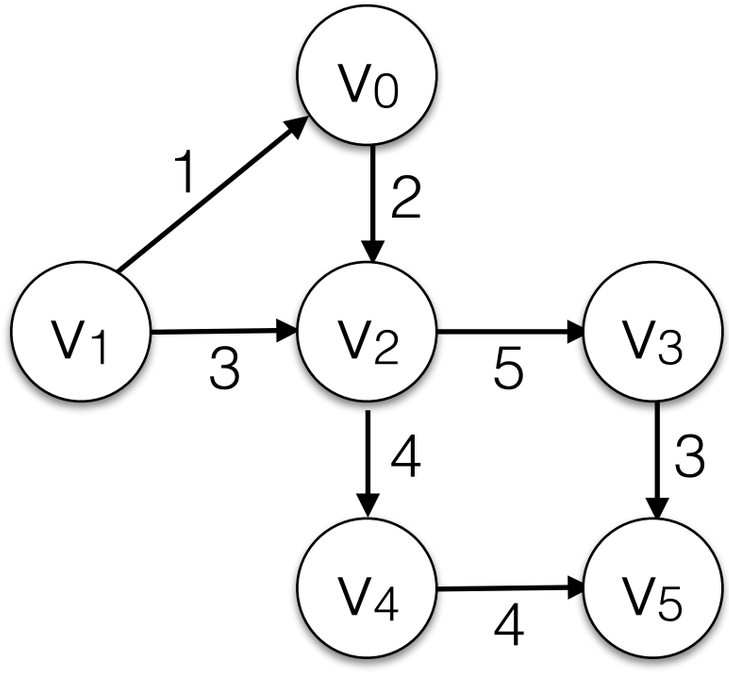
# Representing Graphs

- Problem of Adjacency Matrix representation:

  Space requirement: $\Theta(|V|^2)$

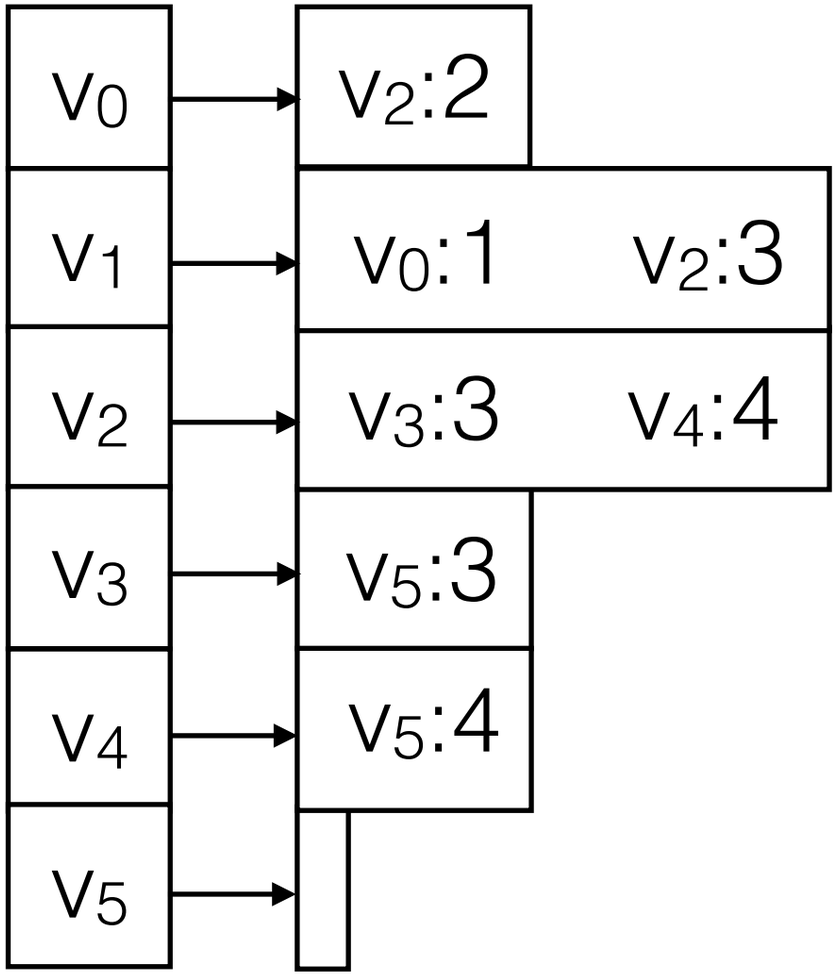
  - For **sparse** graphs (that contain much less than $|V|^2$ edges), a lot of array space is wasted.

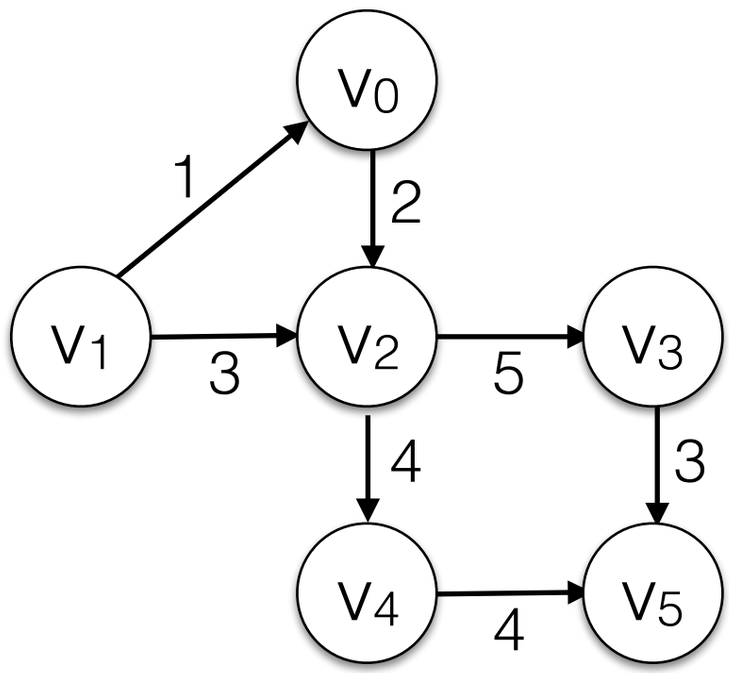
| | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| 0 | ∞ | ∞ | 2 | ∞ | ∞ | ∞ |
| 1 | 1 | ∞ | 3 | ∞ | ∞ | ∞ |
| 2 | ∞ | ∞ | ∞ | 5 | 4 | ∞ |
| 3 | ∞ | ∞ | ∞ | ∞ | ∞ | 3 |
| 4 | ∞ | ∞ | ∞ | ∞ | ∞ | 4 |
| 5 | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ |



# Representing Graphs

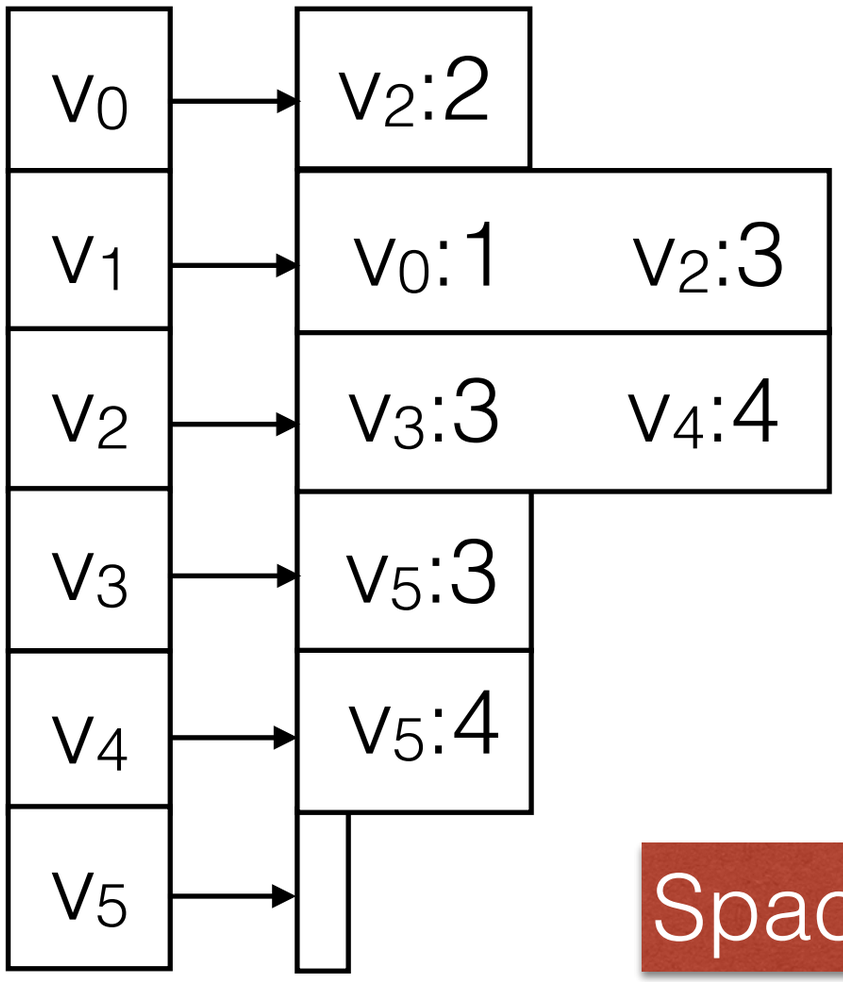
- Represent graph G = (E,V), option 2: **Adjacency Lists**
  - For each vertex, keep a list of all adjacent vertices.

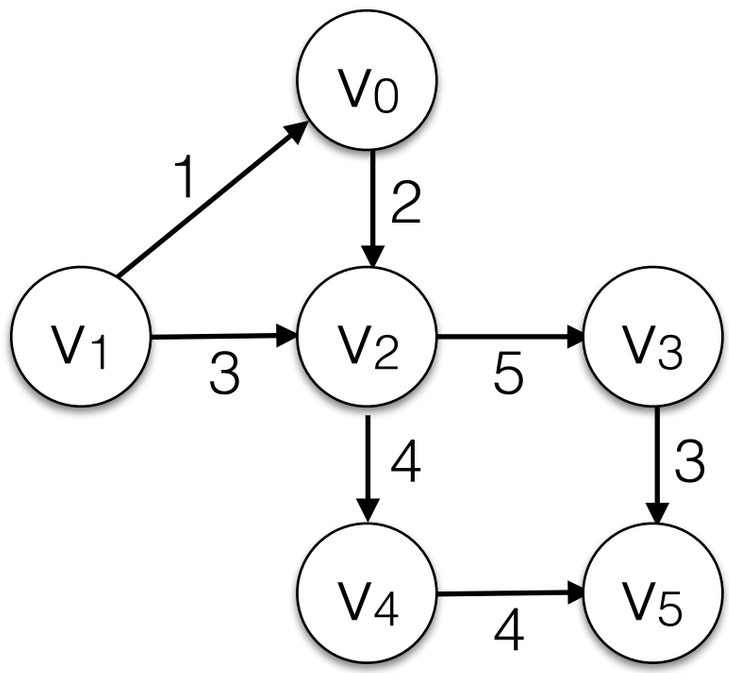


# Representing Graphs

- Represent graph G = (E,V), option 2: **Adjacency Lists**
  - For each vertex, keep a list of all adjacent vertices.

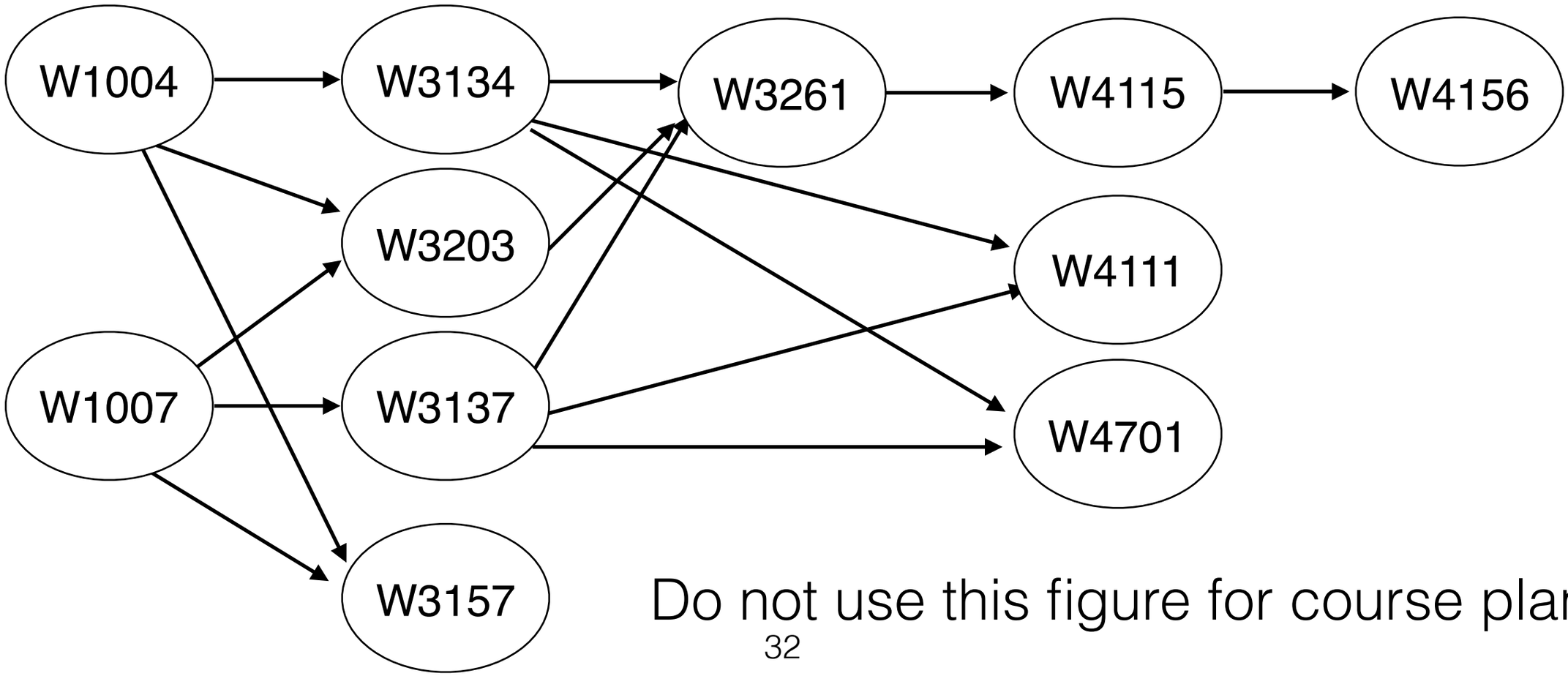



Space requirement: $\Theta(|V| + |E|)$

# Topological Sort in DAGs

A **topological sort** of a DAG is an ordering of its vertices such that if there is a path from *u* to *w*, *u* appears before *w* in the ordering.

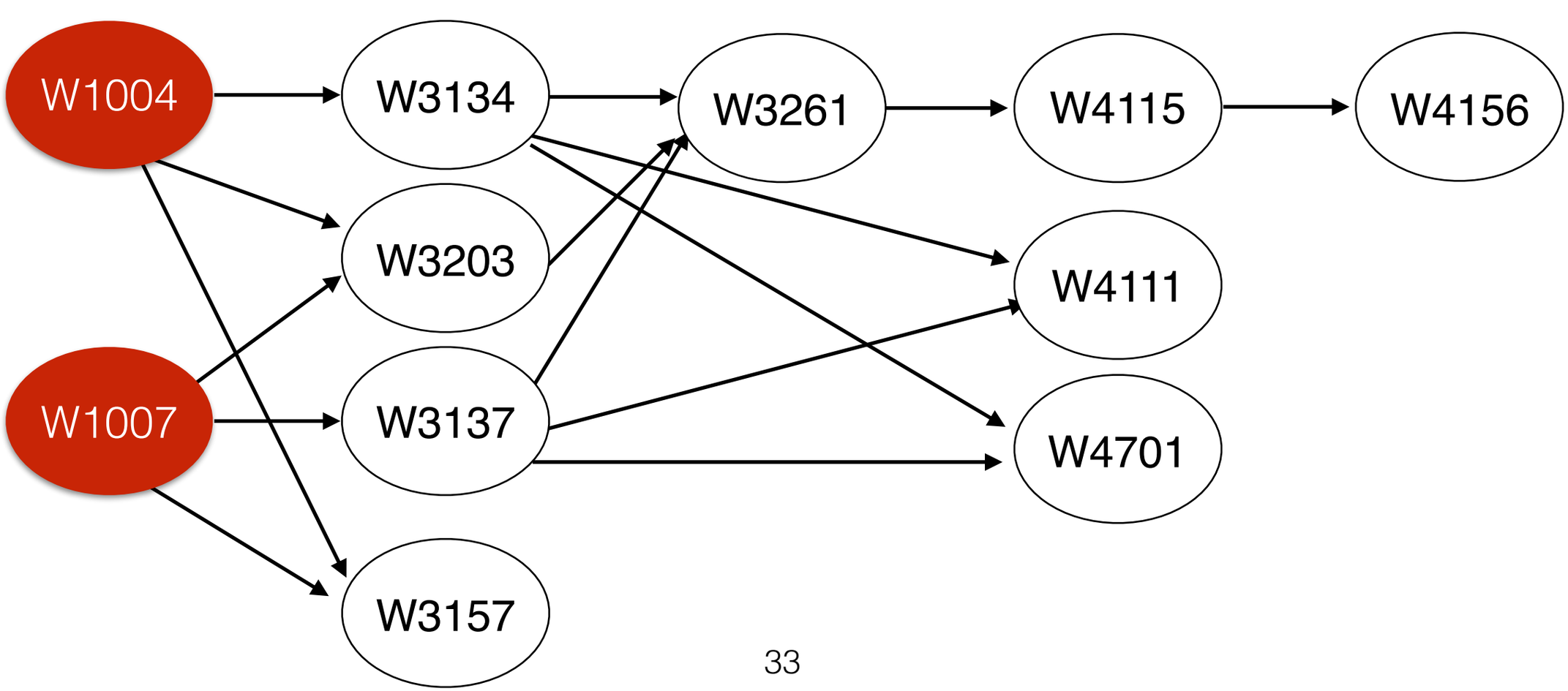



Do not use this figure for course planning!

# Topological Sort in DAGs

A topological sort of a DAG is an ordering of its vertices such that if there is a path from *u* to *w*, *u* appears before *w* in the ordering.

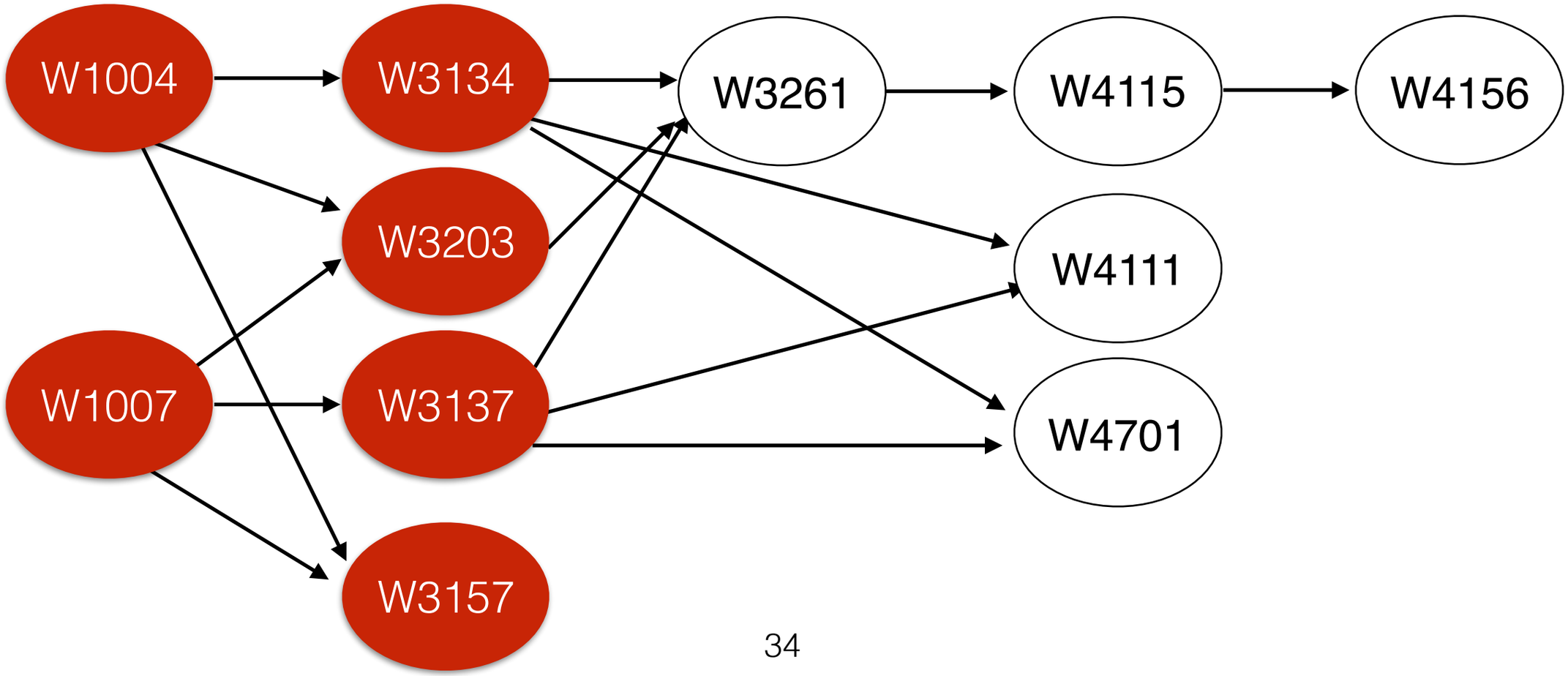
W1007 W1004



# Topological Sort in DAGs

A topological sort of a DAG is an ordering of its vertices such that if there is a path from *u* to *w*, *u* appears before *w* in the ordering.

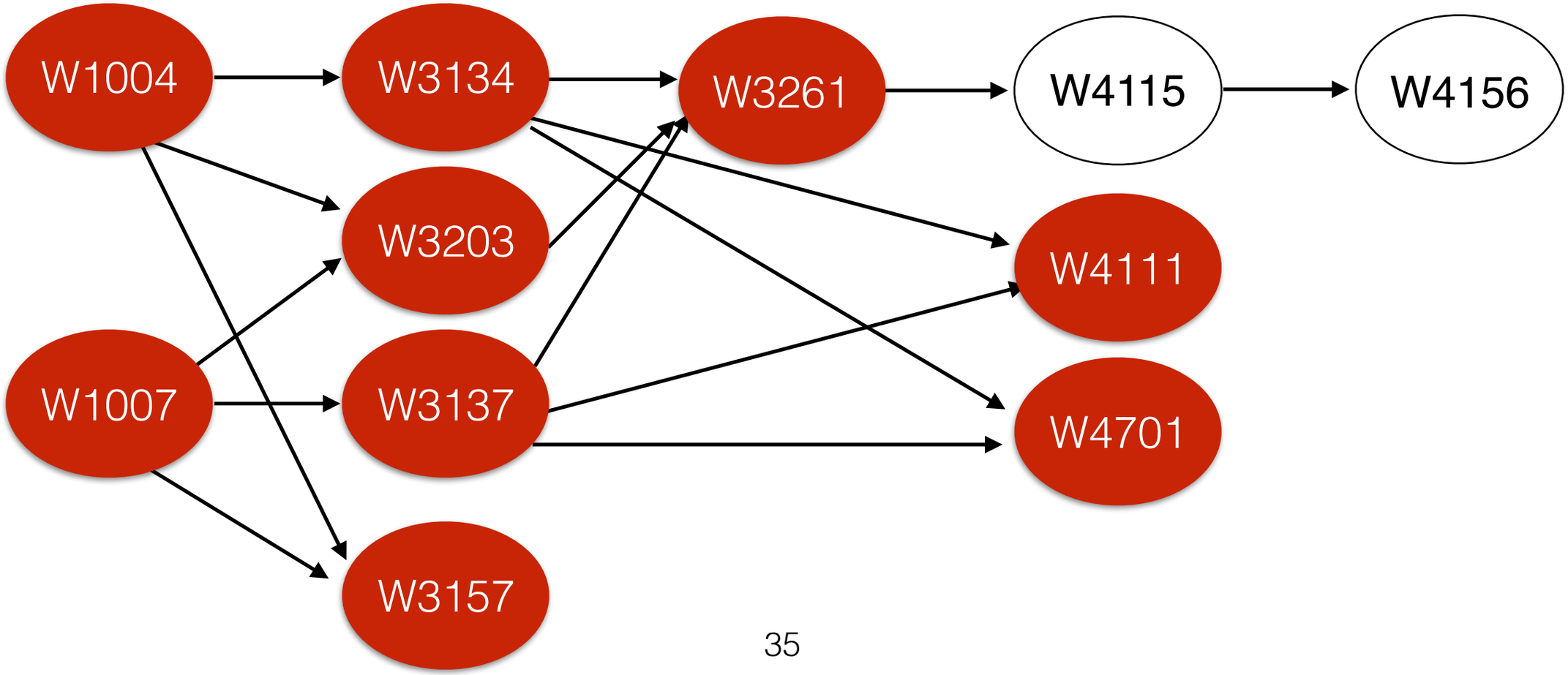
W1007 W1004 W3134 W3203 W3137 W3157



# Topological Sort in DAGs

A topological sort of a DAG is an ordering of its vertices such that if there is a path from *u* to *w*, *u* appears before *w* in the ordering.

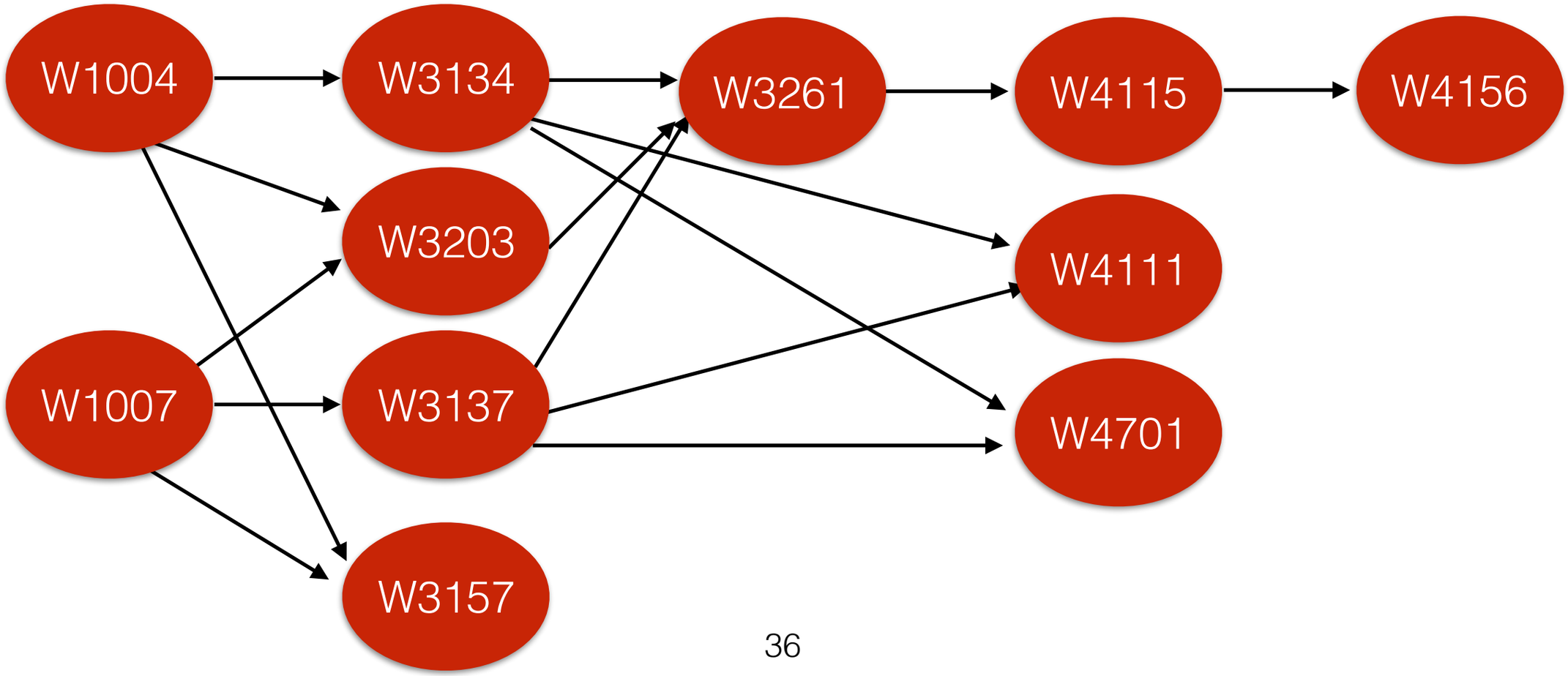
W1007 W1004 W3134 W3203 W3137 W3157 W3261 W4111 W4701



# Topological Sort in DAGs

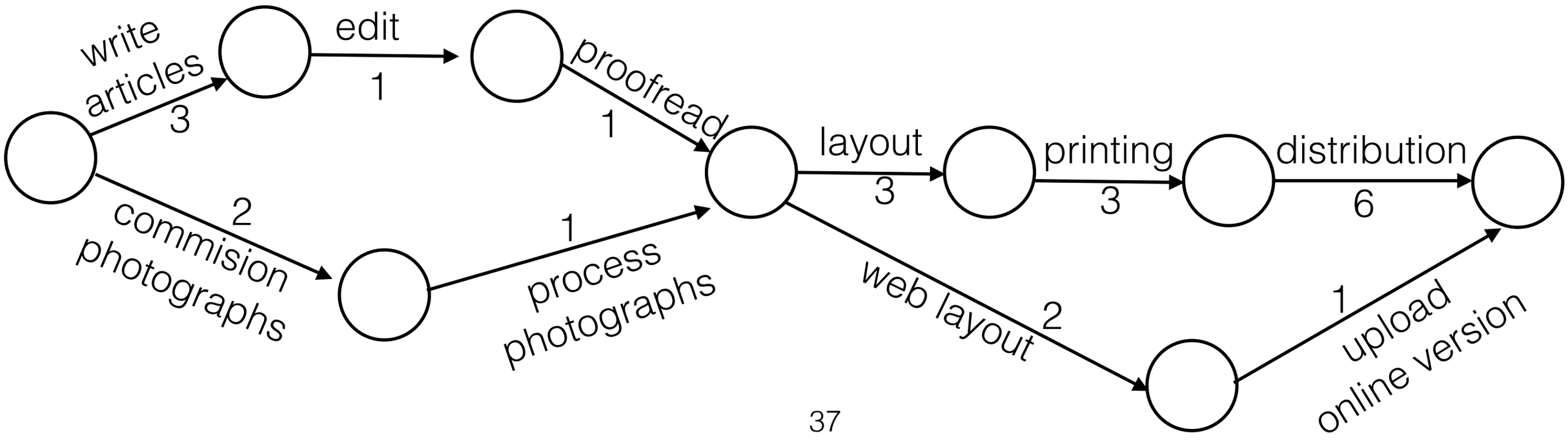
A topological sort of a DAG is an ordering of its vertices such that if there is a path from *u* to *w*, *u* appears before *w* in the ordering.

W1007 W1004 W3134 W3203 W3137 W3157 W3261 W4111 W4701 W4115 W4156



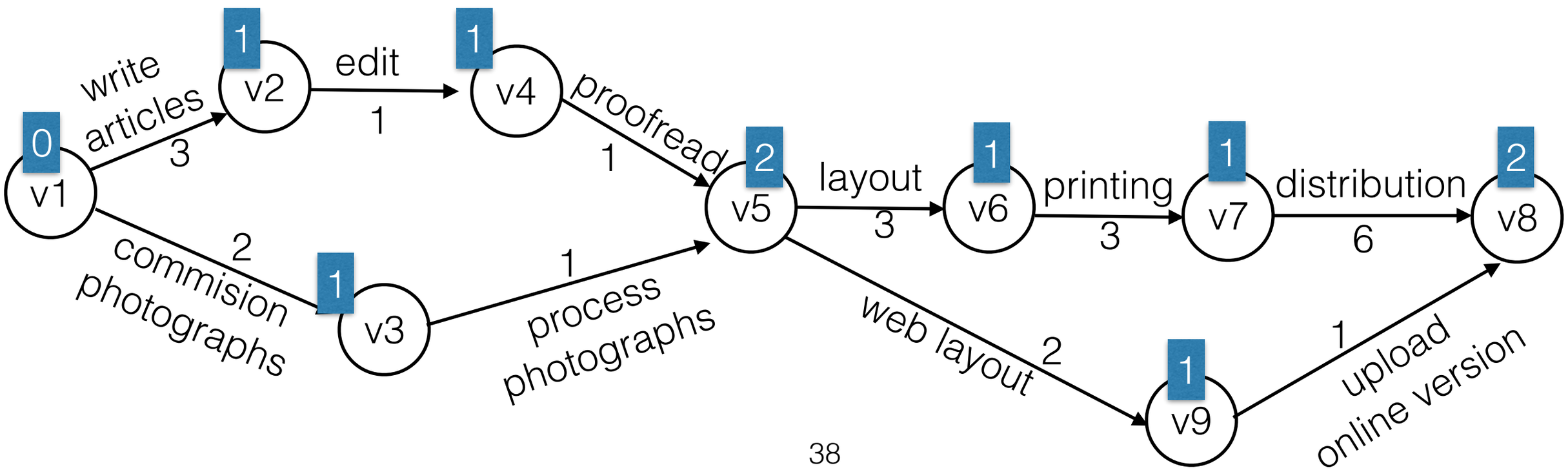
# Application: Critical Path Analysis

- An **Event-Node Graph** is a DAG in which
  - Edges represent tasks, weights represents the time it takes to complete the task.
  - Vertices represent the event of completing a set of tasks.



# Computing Topological Sort

- First annotate each vertex with the number of incoming edges (the **indegree**).

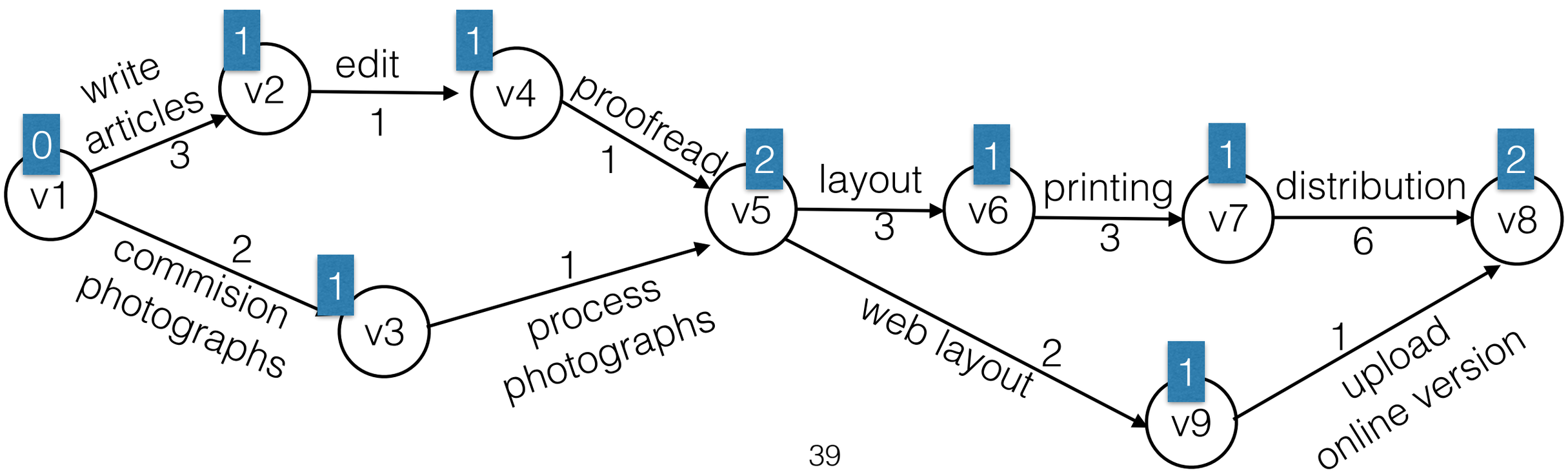


# Computing Topological Sort

- While the queue is not empty, dequeue a vertex, print it and decrement the indegree of its adjacent nodes.
- If the indegree of any new vertex becomes 0, enqueue it.

Queue: v1

Output:

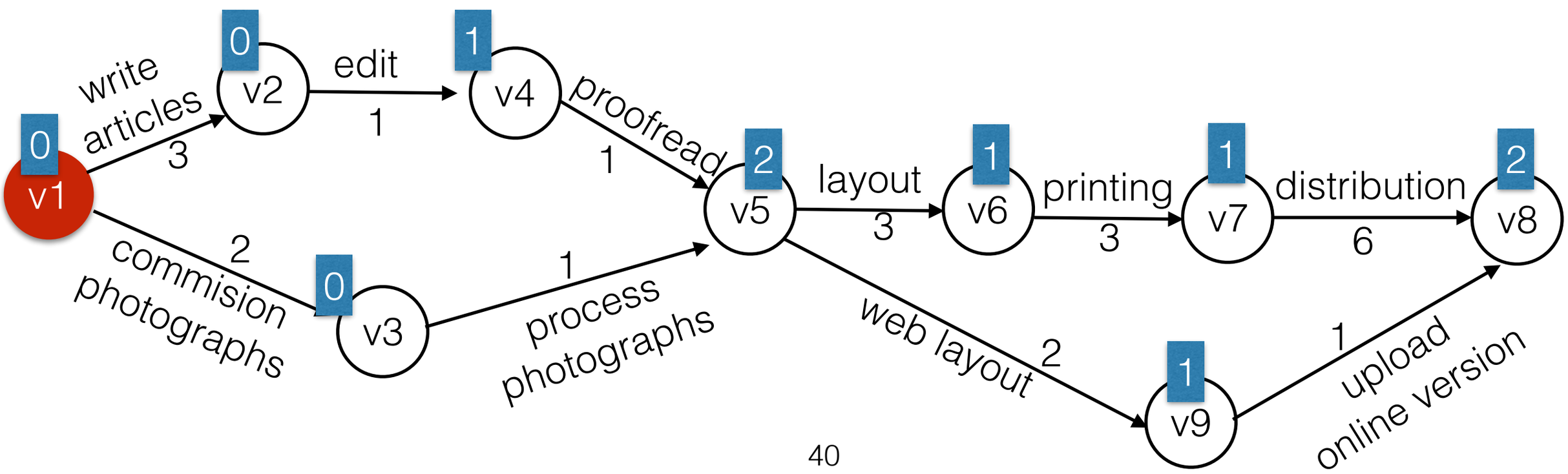


# Computing Topological Sort

- While the queue is not empty, dequeue a vertex, print it and decrement the indegree of its adjacent nodes.
- If the indegree of any new vertex becomes 0, enqueue it.

Queue: v2 v3

Output: v1

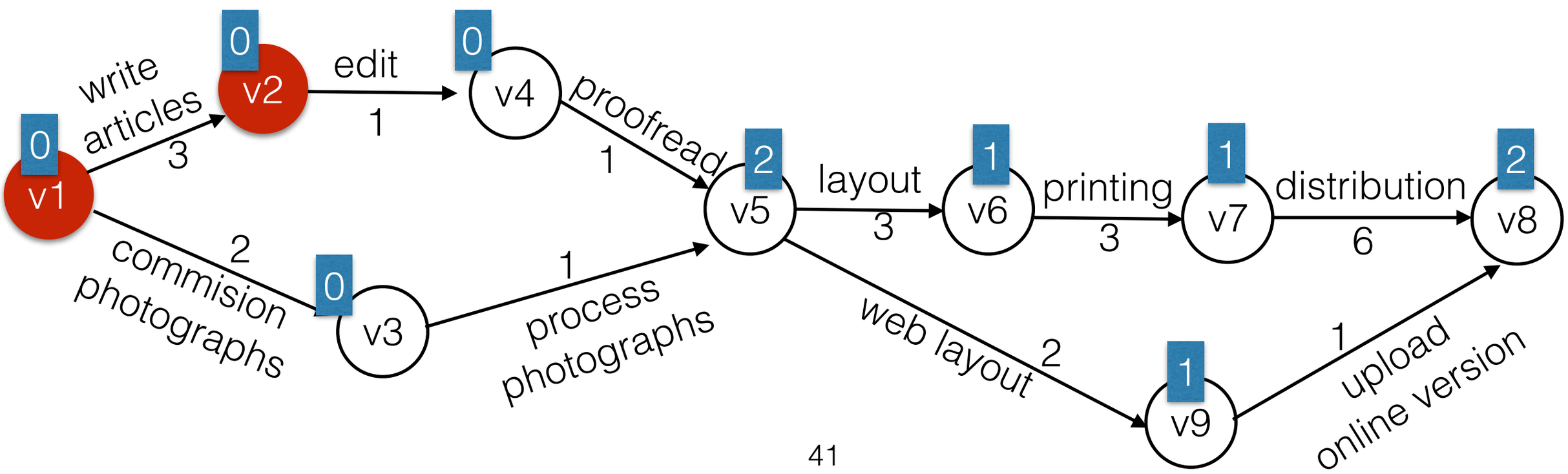


# Computing Topological Sort

- While the queue is not empty, dequeue a vertex, print it and decrement the indegree of its adjacent nodes.
- If the indegree of any new vertex becomes 0, enqueue it.

Queue: v3 v4

Output: v1 v2

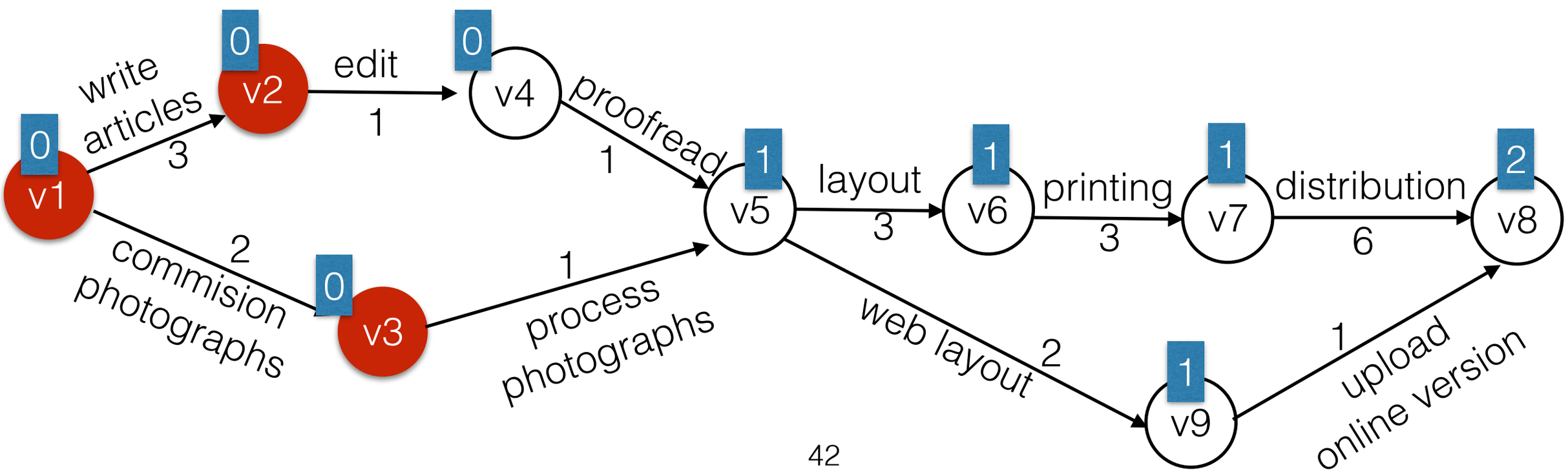


# Computing Topological Sort

- While the queue is not empty, dequeue a vertex, print it and decrement the indegree of its adjacent nodes.
- If the indegree of any new vertex becomes 0, enqueue it.

Queue: v4

Output: v1 v2 v3

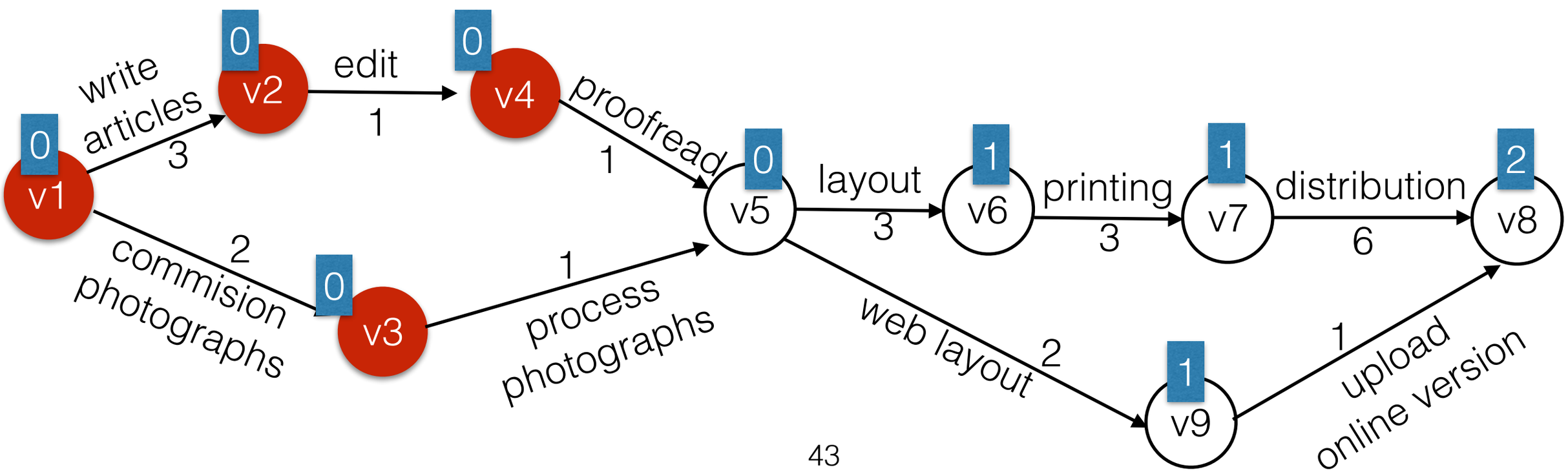


# Computing Topological Sort

- While the queue is not empty, dequeue a vertex, print it and decrement the indegree of its adjacent nodes.
- If the indegree of any new vertex becomes 0, enqueue it.

Queue: v5

Output: v1 v2 v3  v4

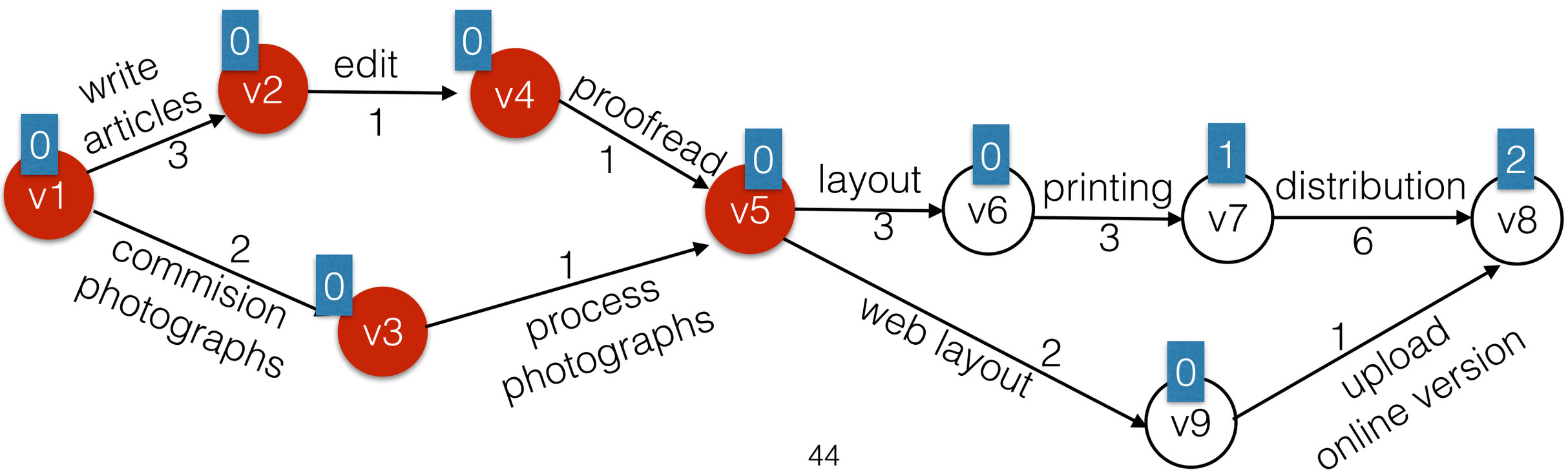


# Computing Topological Sort

- While the queue is not empty, dequeue a vertex, print it and decrement the indegree of its adjacent nodes.
- If the indegree of any new vertex becomes 0, enqueue it.

Queue: v6 v9

Output: v1 v2 v3 v4 v5

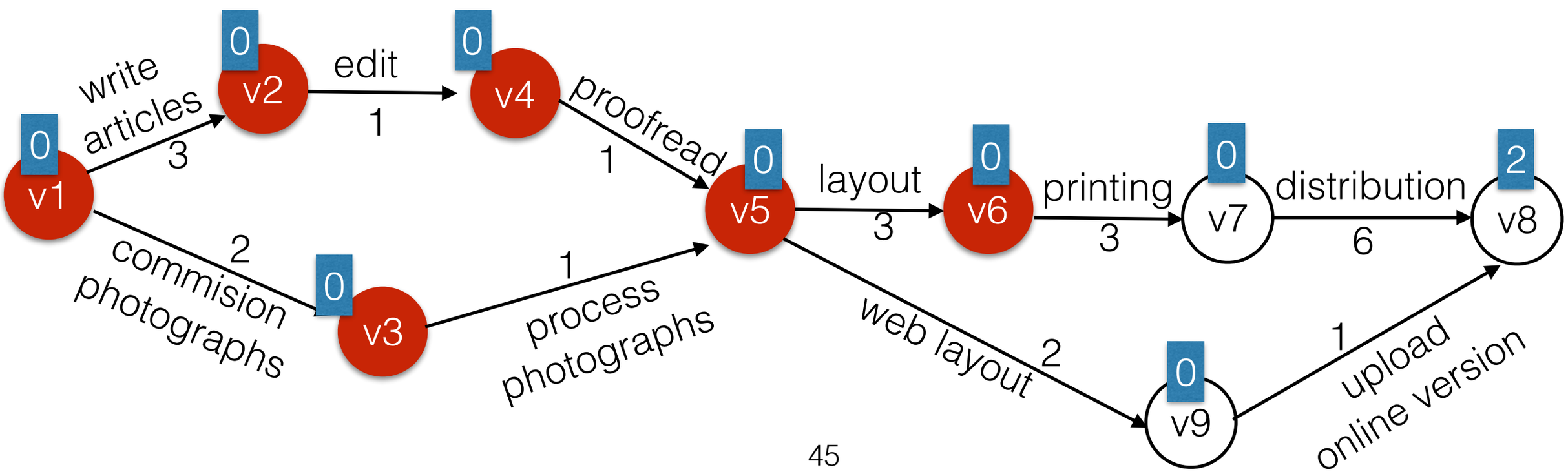


# Computing Topological Sort

- While the queue is not empty, dequeue a vertex, print it and decrement the indegree of its adjacent nodes.
- If the indegree of any new vertex becomes 0, enqueue it.

Queue: v9 v7

Output: v1 v2 v3 v4 v5 v6

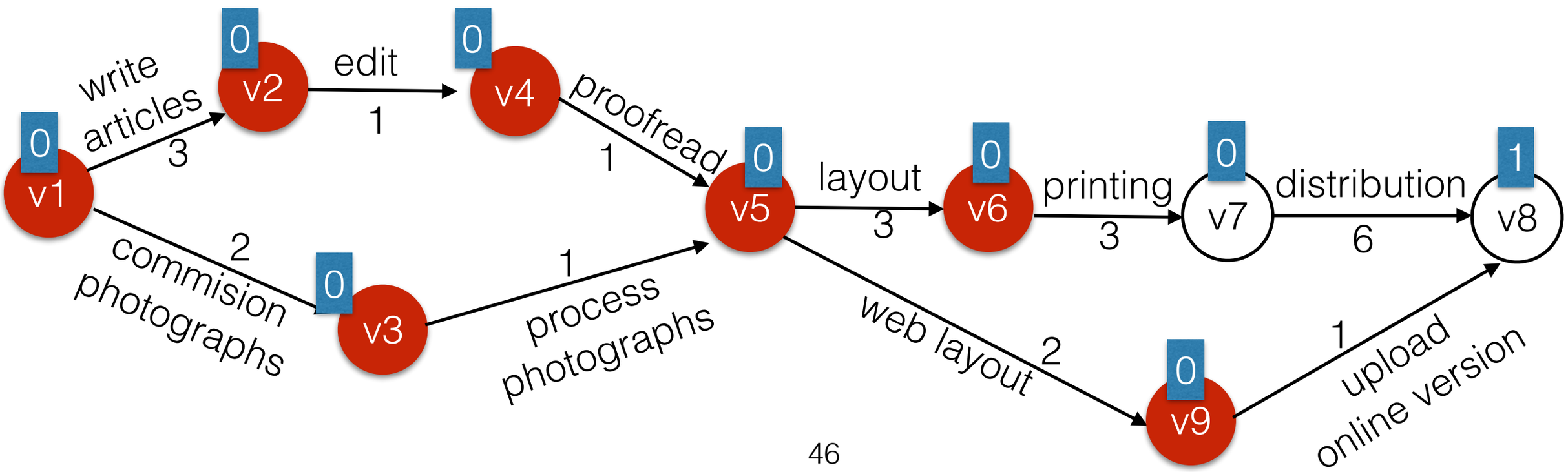


# Computing Topological Sort

- While the queue is not empty, dequeue a vertex, print it and decrement the indegree of its adjacent nodes.
- If the indegree of any new vertex becomes 0, enqueue it.

Queue: v7

Output: v1 v2 v3 v4 v5 v6 v9

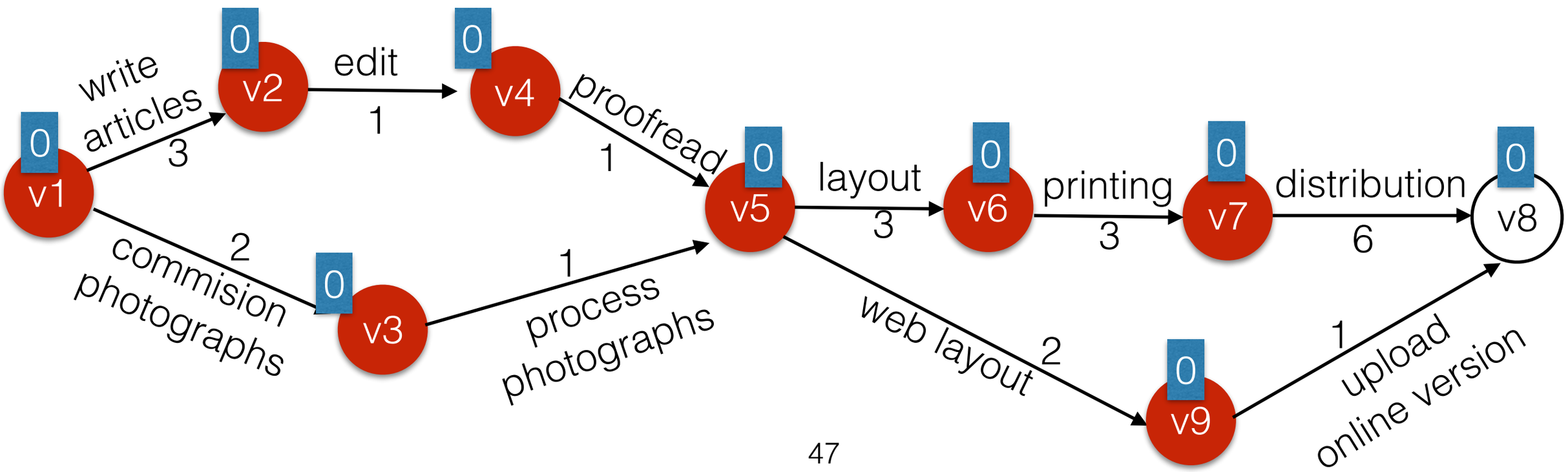


# Computing Topological Sort

- While the queue is not empty, dequeue a vertex, print it and decrement the indegree of its adjacent nodes.
- If the indegree of any new vertex becomes 0, enqueue it.

Queue: v8

Output: v1 v2 v3 v4 v5 v6 v9 v7

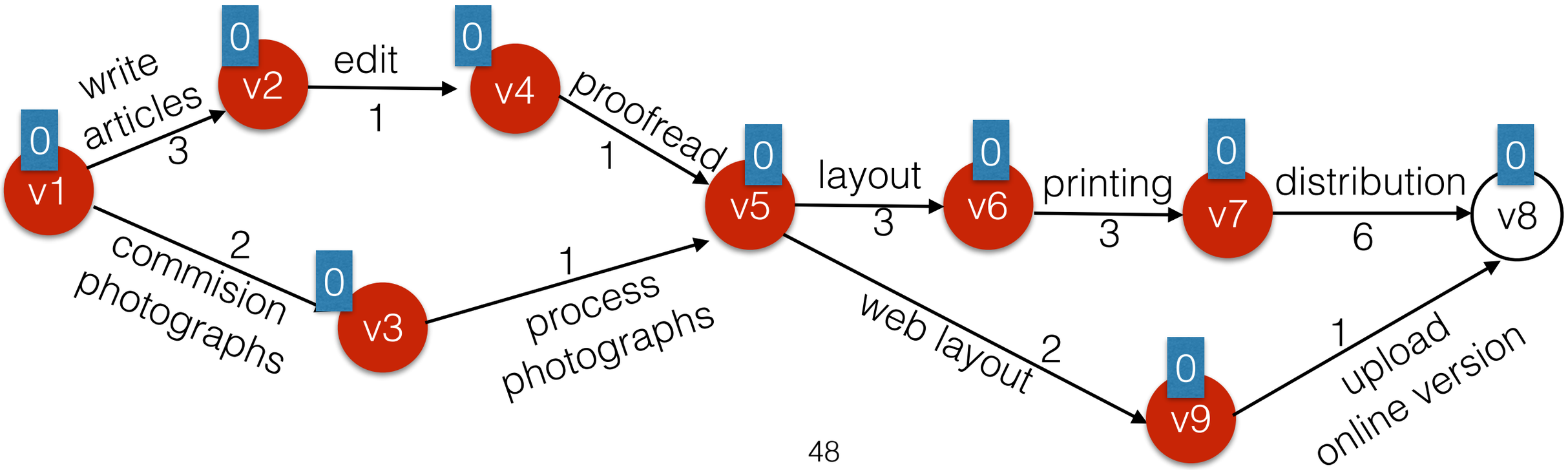


# Computing Topological Sort

- While the queue is not empty, dequeue a vertex, print it and decrement the indegree of its adjacent nodes.
- If the indegree of any new vertex becomes 0, enqueue it.

Queue:

Output: v1 v2 v3 v4 v5 v6 v9 v7 v8



# Topological Sort - Algorithm

- First annotate each vertex with its indegree.
- Enqueue all vertices with indegree 0.
- While the queue is not empty
  - dequeue a vertex.
  - add it to the topological sort.
  - decrement the indegree of its adjacent vertices.
  - if the indegree of any new vertex becomes 0, enqueue it.

# Topological Sort - Algorithm

- First annotate each vertex with its indegree.
- Enqueue all vertices with indegree 0.
- While the queue is not empty
  - dequeue a vertex.
  - add it to the topological sort.
  - decrement the indegree of its adjacent vertices.
  - if the indegree of any new vertex becomes 0, enqueue it.

O(|V|+|E|)

# Topological Sort - Algorithm

- First annotate each vertex with its indegree.
- Enqueue all vertices with indegree 0.
- While the queue is not empty
  - dequeue a vertex.
  - add it to the topological sort.
  - decrement the indegree of its adjacent vertices.
  - if the indegree of any new vertex becomes 0, enqueue it.

O(|V|+|E|)

O(|E|) edges visited

# Topological Sort - Algorithm

- First annotate each vertex with its indegree.
- Enqueue all vertices with indegree 0.
- While the queue is not empty
  - dequeue a vertex.
  - add it to the topological sort.
  - decrement the indegree of its adjacent vertices.
  - if the indegree of any new vertex becomes 0, enqueue it.

O(|V|+|E|)

O(|E|) edges visited

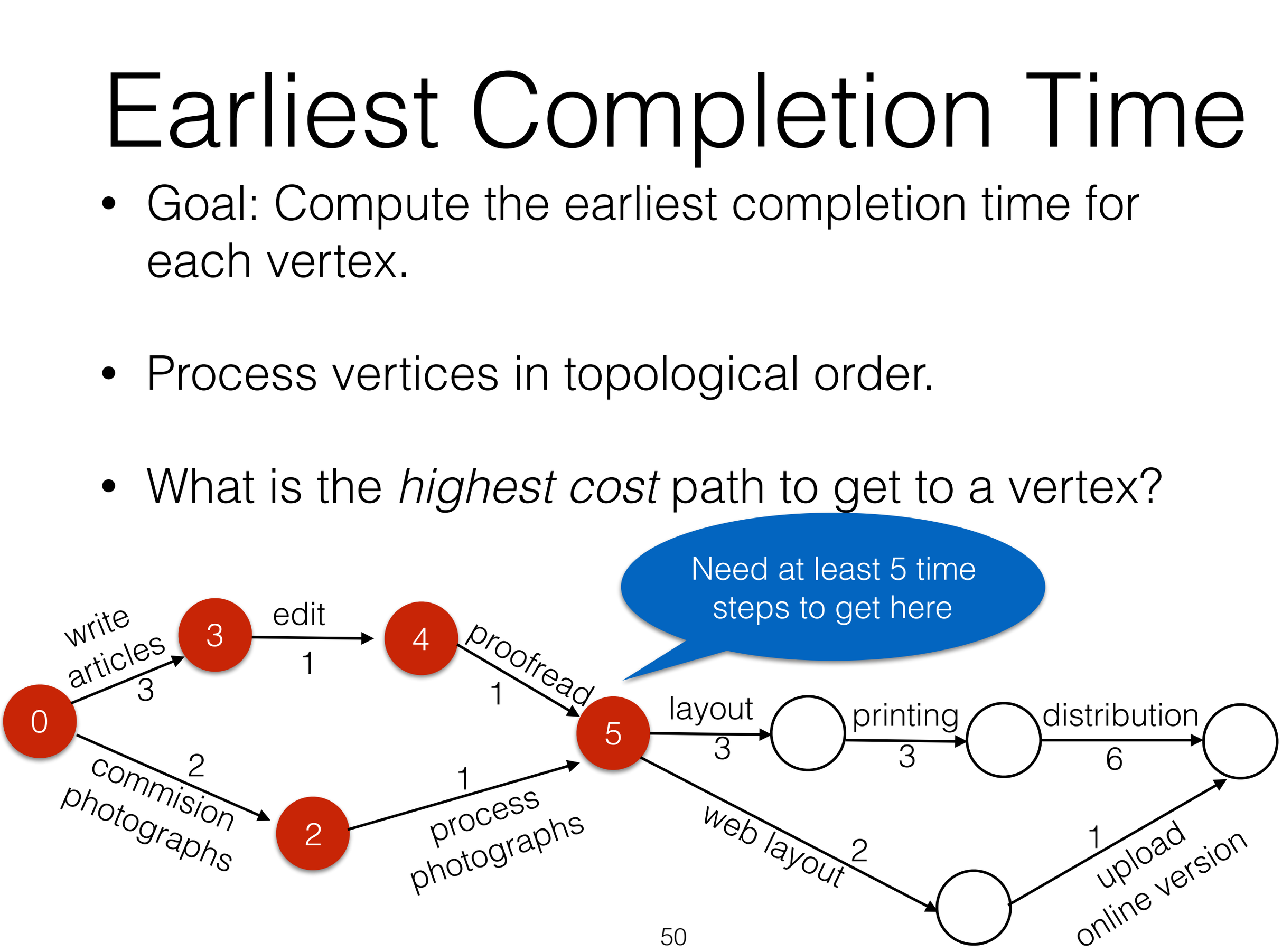
Total: O(|V|+|E|)

# Earliest Completion Time

- Goal: Compute the earliest completion time for each vertex.

- Process vertices in topological order.

- What is the *highest cost* path to get to a vertex?

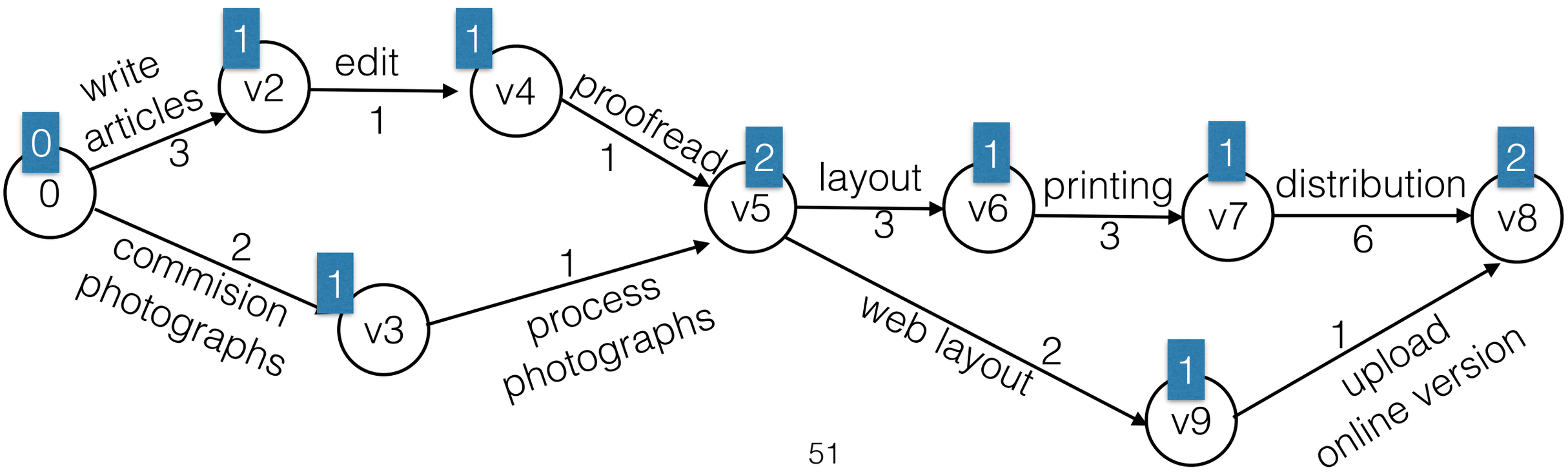


# Earliest Completion Time

- While the queue is not empty, dequeue a vertex, print it and decrement the indegree of its adjacent nodes. **Update earliest completion time for each adjacent node.**
- If the indegree of any new vertex becomes 0, enqueue it.

Queue: v1

Output:

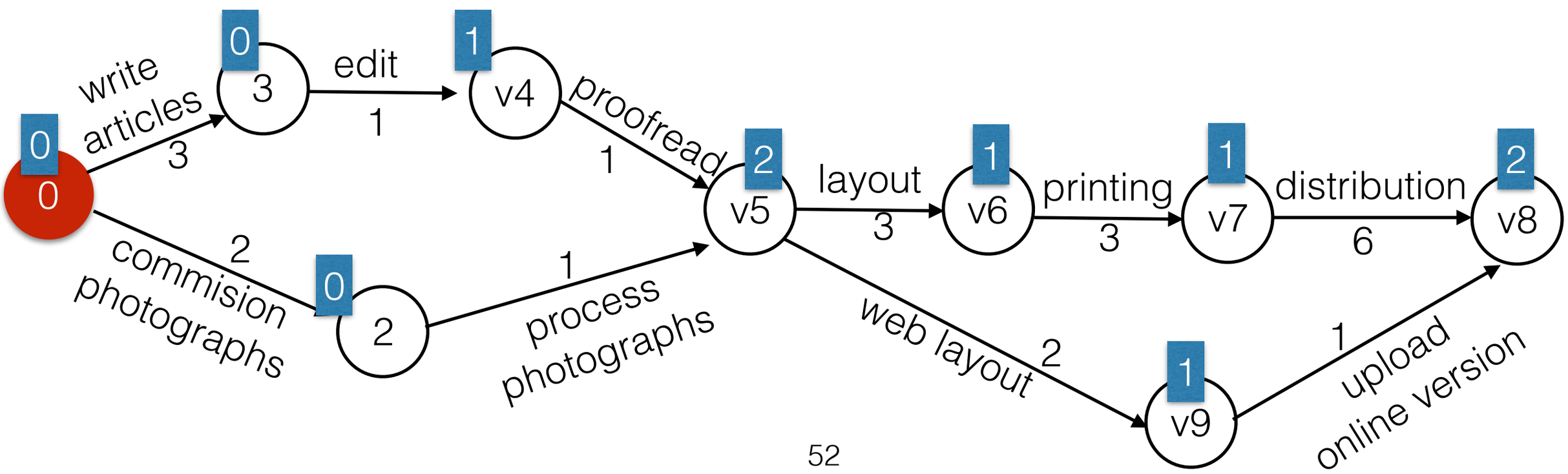


# Earliest Completion Time

- While the queue is not empty, dequeue a vertex, print it and decrement the indegree of its adjacent nodes. **Update earliest completion time for each adjacent node.**
- If the indegree of any new vertex becomes 0, enqueue it.

Queue: v2 v3

Output: v1

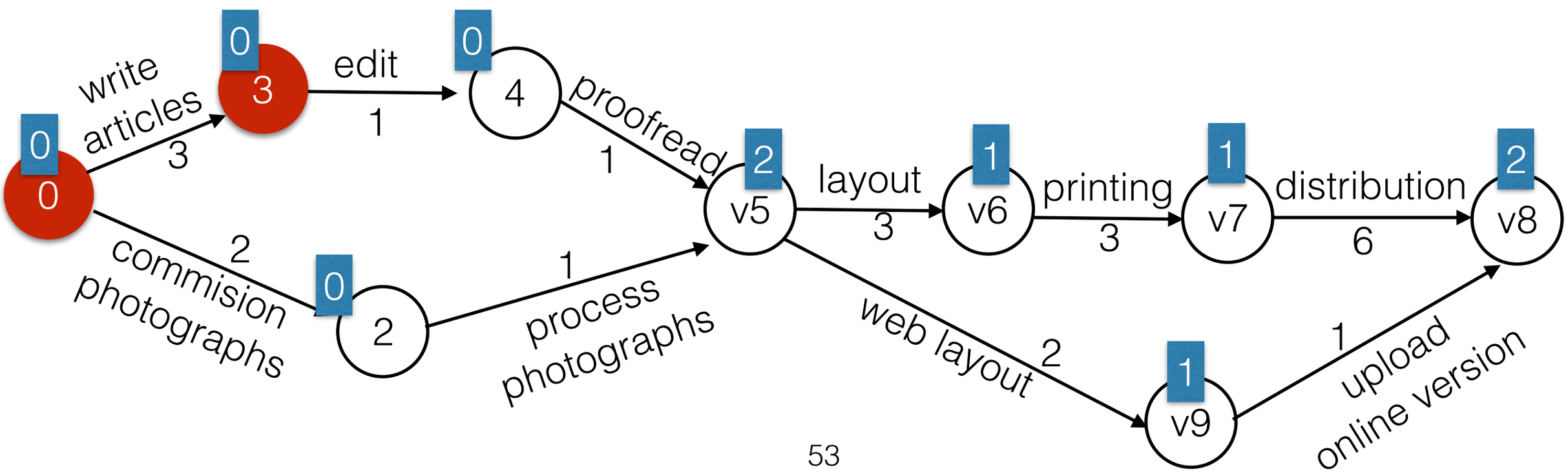


# Earliest Completion Time

- While the queue is not empty, dequeue a vertex, print it and decrement the indegree of its adjacent nodes. **Update earliest completion time for each adjacent node.**
- If the indegree of any new vertex becomes 0, enqueue it.

Queue: v3 v4

Output: v1 v2

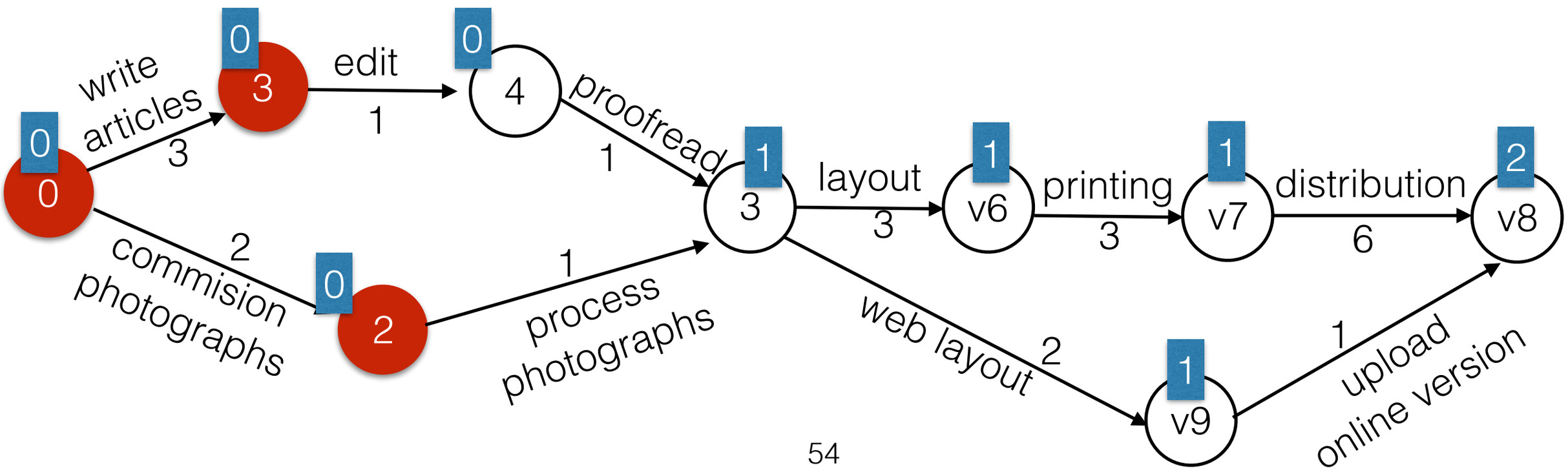


# Earliest Completion Time

- While the queue is not empty, dequeue a vertex, print it and decrement the indegree of its adjacent nodes. **Update earliest completion time for each adjacent node.**
- If the indegree of any new vertex becomes 0, enqueue it.

Queue: v4

Output: v1 v2 v3

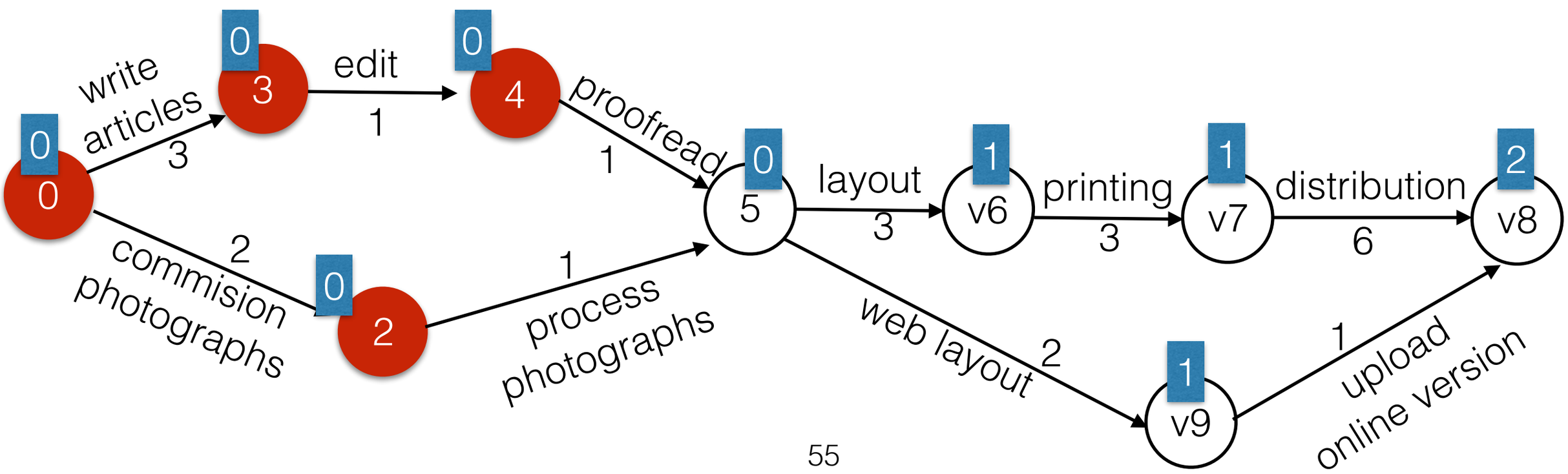


# Earliest Completion Time

- While the queue is not empty, dequeue a vertex, print it and decrement the indegree of its adjacent nodes. **Update earliest completion time for each adjacent node.**
- If the indegree of any new vertex becomes 0, enqueue it.

Queue: v5

Output: v1 v2 v3  v4

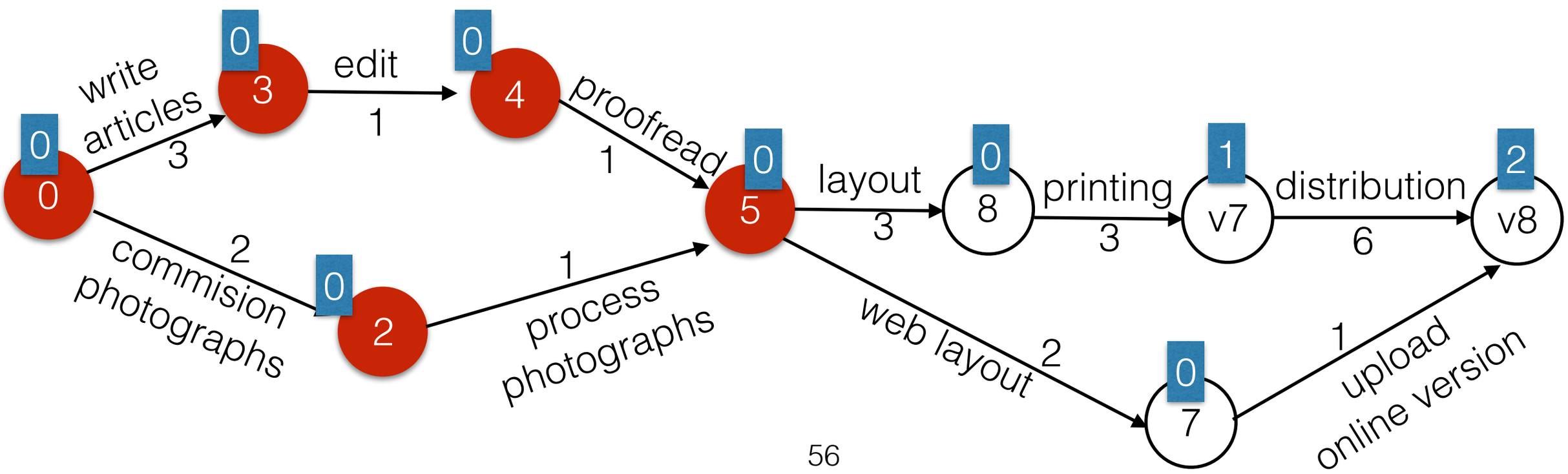


# Earliest Completion Time

- While the queue is not empty, dequeue a vertex, print it and decrement the indegree of its adjacent nodes. **Update earliest completion time for each adjacent node.**
- If the indegree of any new vertex becomes 0, enqueue it.

Queue: v6 v9

Output: v1 v2 v3 v4 v5

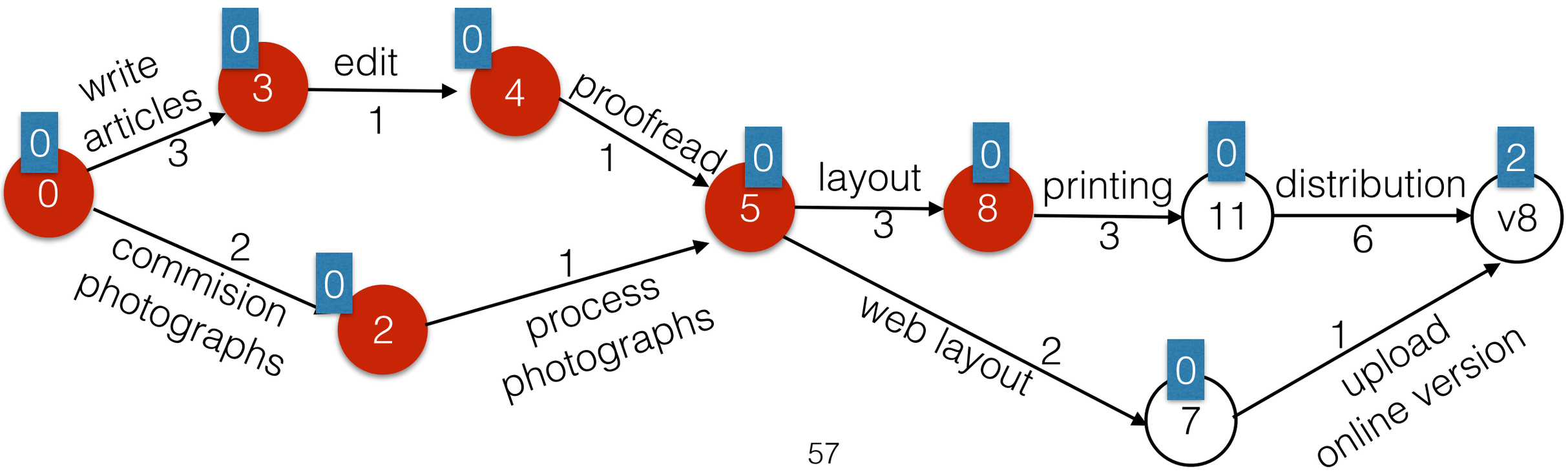


# Earliest Completion Time

- While the queue is not empty, dequeue a vertex, print it and decrement the indegree of its adjacent nodes. **Update earliest completion time for each adjacent node.**
- If the indegree of any new vertex becomes 0, enqueue it.

Queue: v9 v7

Output: v1 v2 v3 v4 v5 v6

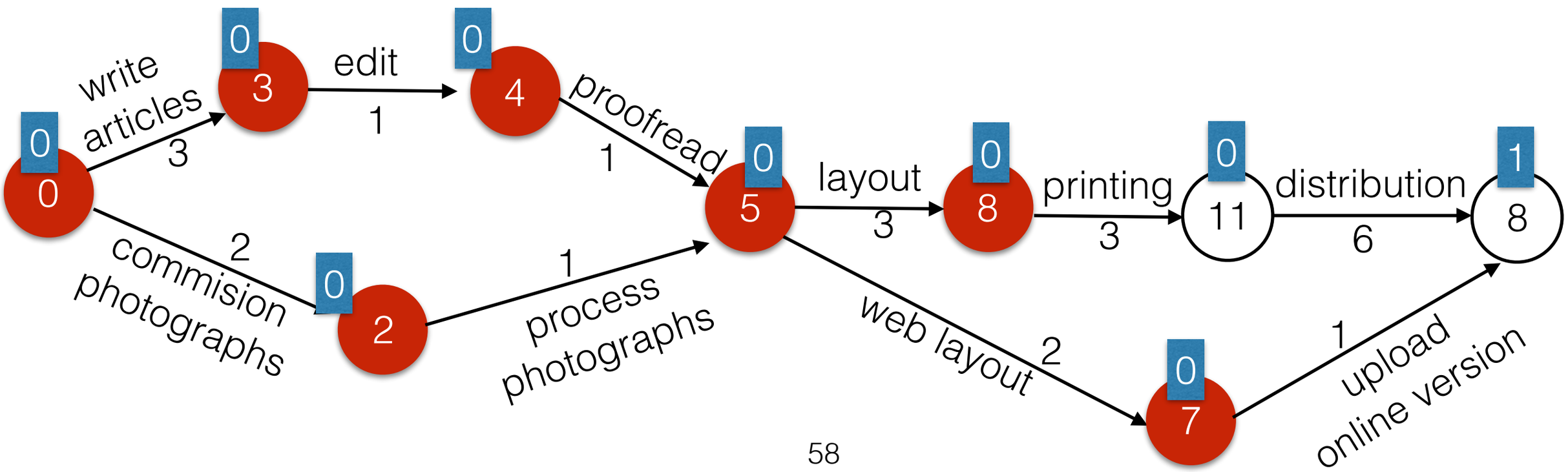


# Earliest Completion Time

- While the queue is not empty, dequeue a vertex, print it and decrement the indegree of its adjacent nodes. **Update earliest completion time for each adjacent node.**
- If the indegree of any new vertex becomes 0, enqueue it.

Queue: v7

Output: v1 v2 v3 v4 v5 v6 v9



# Earliest Completion Time

- While the queue is not empty, dequeue a vertex, print it and decrement the indegree of its adjacent nodes. **Update earliest completion time for each adjacent node.**
- If the indegree of any new vertex becomes 0, enqueue it.

Queue: v8

Output: v1 v2 v3 v4 v5 v6 v9 v7