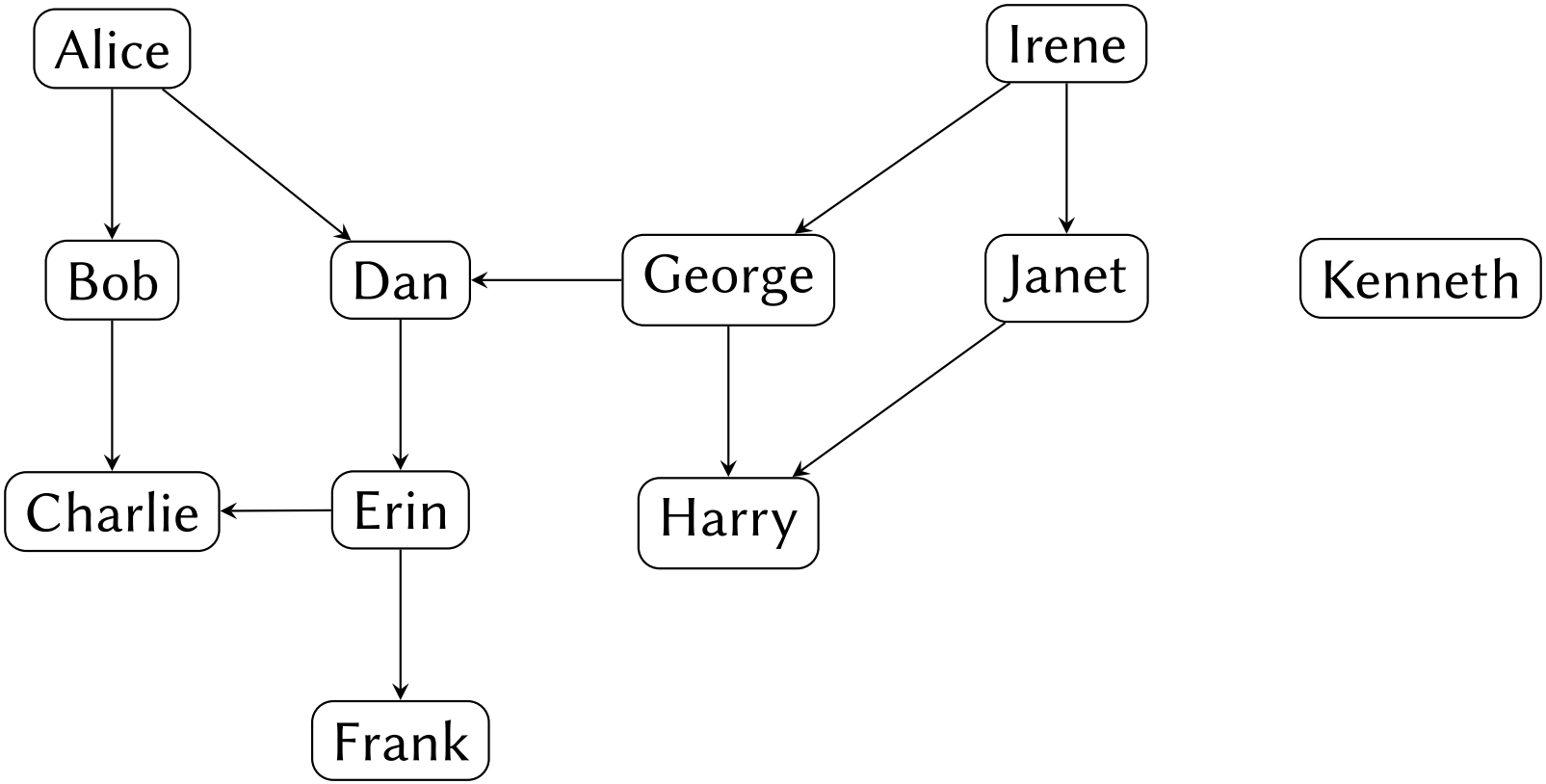
Chapter 6:
Tasks in Order

Panos Louridas

Athens University of Economics and Business
Real World Algorithms
A Beginners Guide
The MIT Press

Outline

Cast of Characters

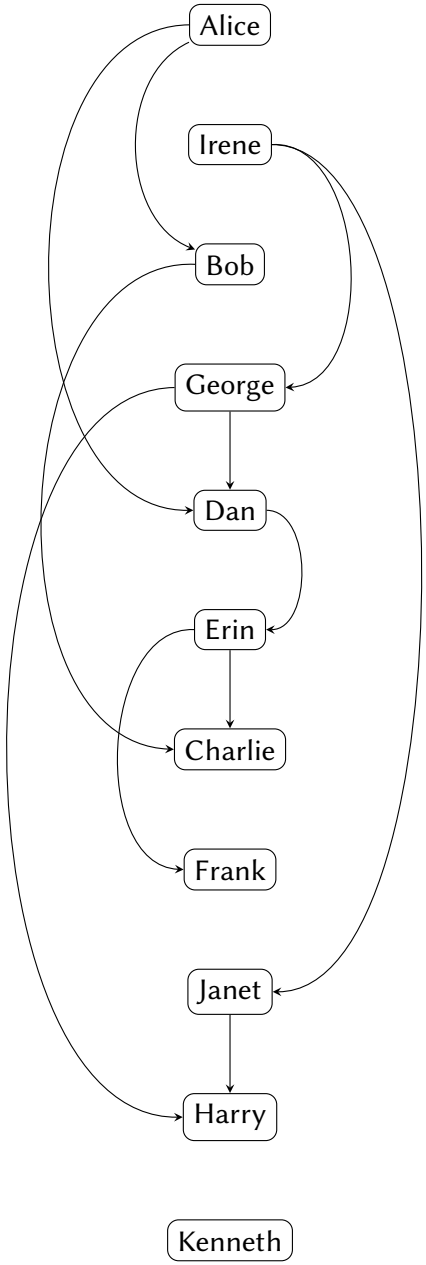


Topological Sort

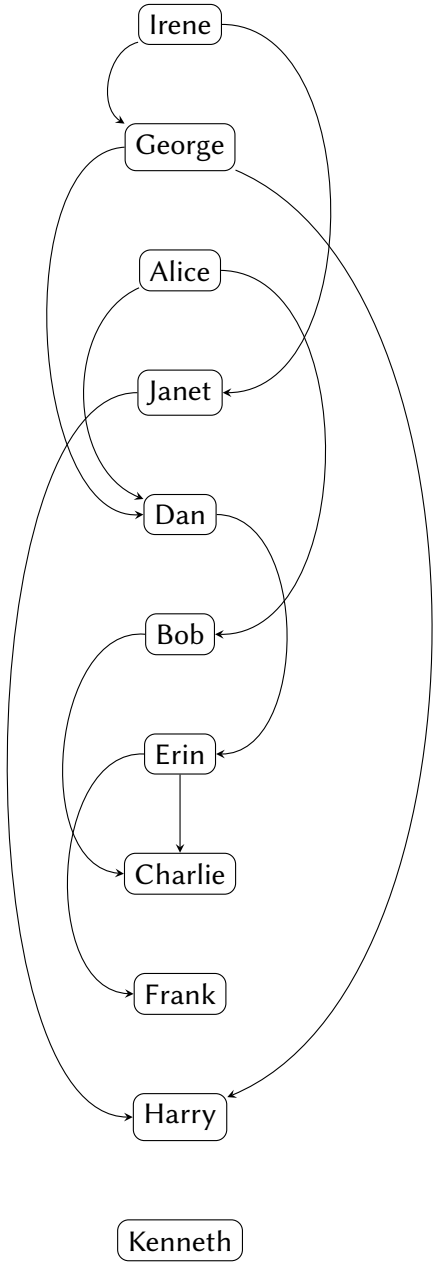
Definition

A topological sort of a directed acyclic graph (dag) $G = (V, E)$ is an ordering of the vertices V of the graph such that for every edge (u, v) that appears in E, the vertex u appears in the ordering before vertex v.

An Ordering


Alice
Irene
Bob
George
Dan
Erin
Charlie
Frank
Janet
Harry
Kenneth

Another Ordering



Topological Sort: Basic Idea

- Instead of trying to find the first vertex in the ordering, let's start by trying to find the last.
- The last will be the one that cannot lead us anywhere we have not already visited (dead end).
- Then, the one before the last will be again a vertex that cannot lead us anywhere we have not already visited, and so on.
- To explore the graph in this way, we only need to do a depth-first search: every time we backtrack, we add a vertex from the end of the ordering to the front.

Depth-first Search for Topological Sort

Algorithm: DFS for topological sort.

`DFSTopologicalSort`(G, *node*)

Input: $G = (V, E)$, a dag
node, a node in G

Data: *visited*, an array of size $|V|$
sorted, a list

Result: *visited*[i] is TRUE if node i is reachable from *node*
sorted contains in its start, in reverse order, the dead-ends we reached using depth-first search starting from *node*

```
1  visited[node] ← TRUE
2  foreach v in AdjacencyList(G, node) do
3      if not visited[v] then
4          DFSTopologicalSort(G, v)
5  InsertInList(sorted, NULL, node)
```

Topological Sort Algorithm

Algorithm: Topological sort on a dag.

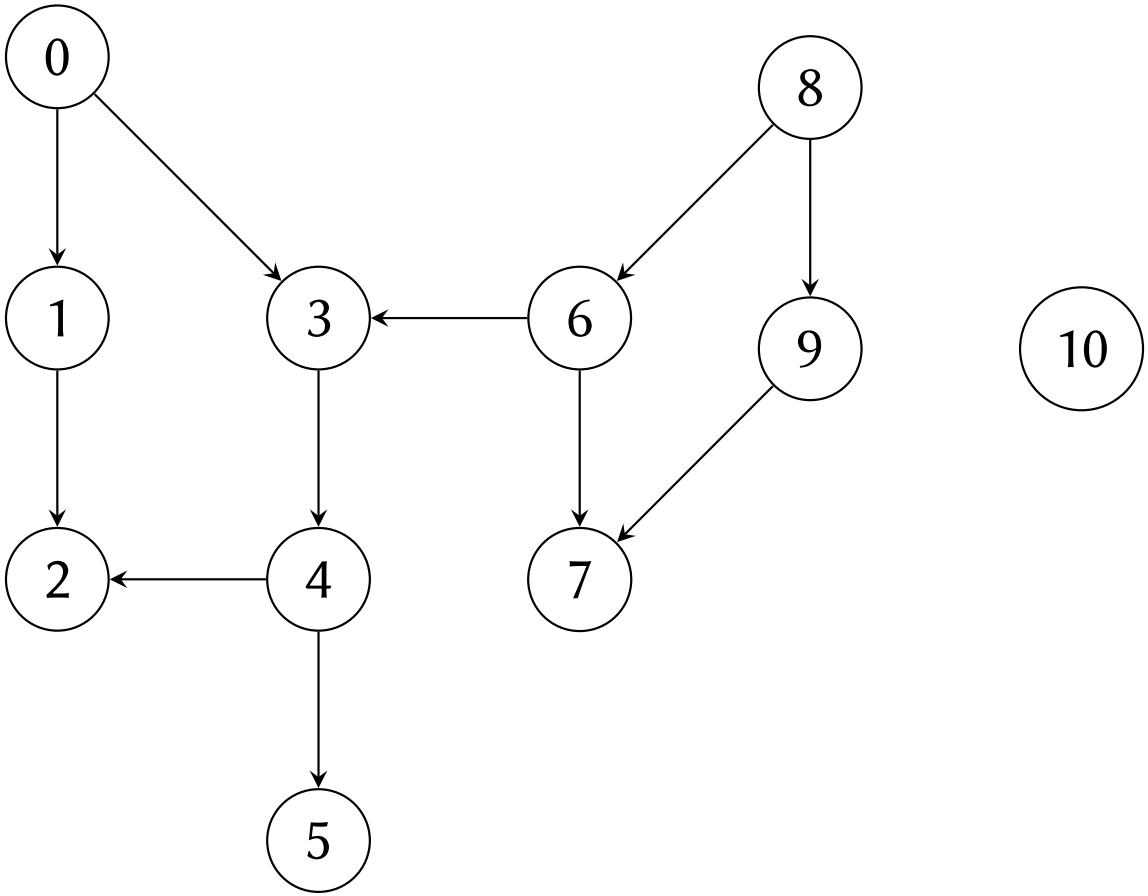
`TopologicalSort`$(G) \rightarrow$ *sorted*

Input: $G = (V, E)$, a dag

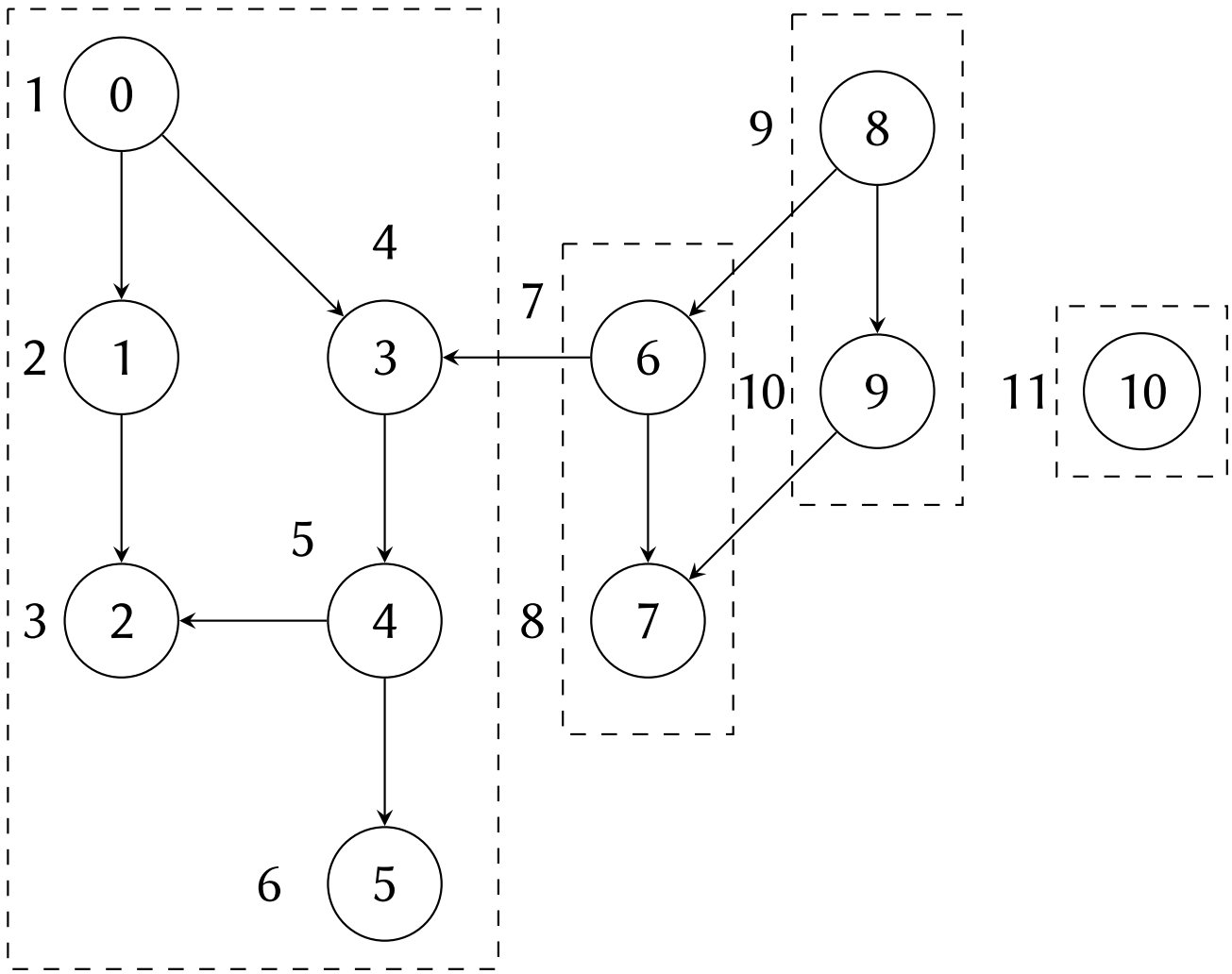
Output: *sorted*, a list of size $|V|$ with the nodes of the graph in topological order

```
1  visited ← CreateArray(|V|)
2  sorted ← CreateList()
3  for i ← 0 to |V| do
4      visited[i] ← FALSE
5  for i ← 0 to |V| do
6      if not visited[i] then
7          DFSTopologicalSort(G, i)
8  return sorted
```

Topological Sort Example (1)

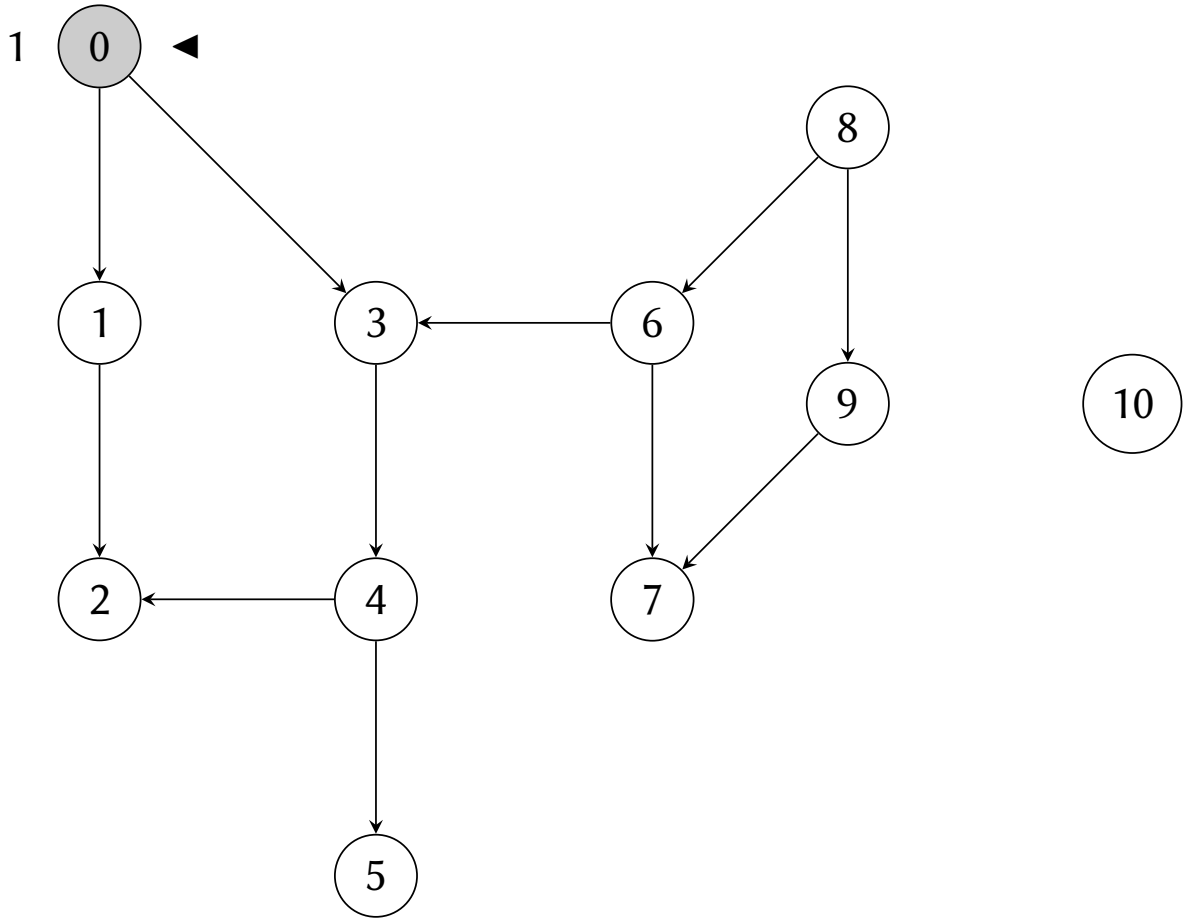


Topological Sort Example (2)



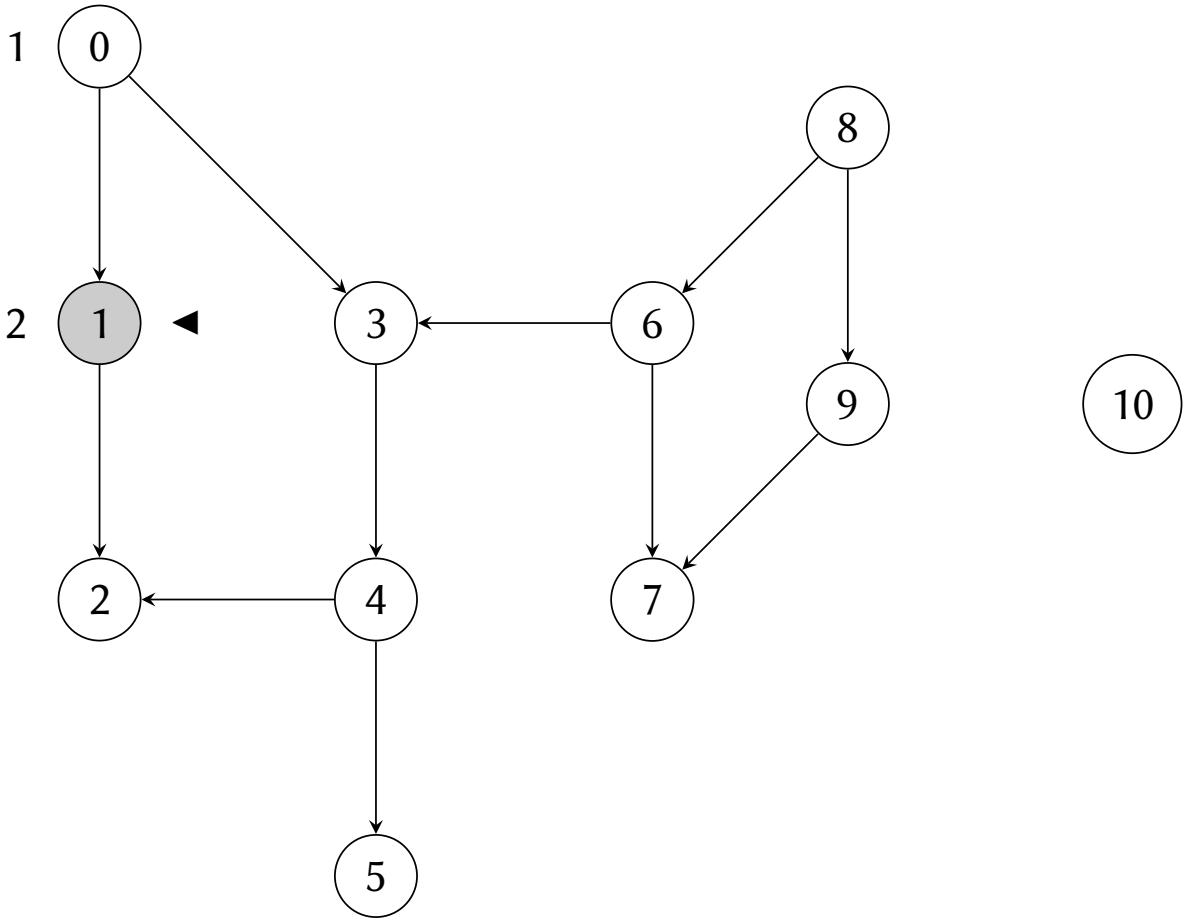
Depth-first graph traversal.

Topological Sort Example (3)



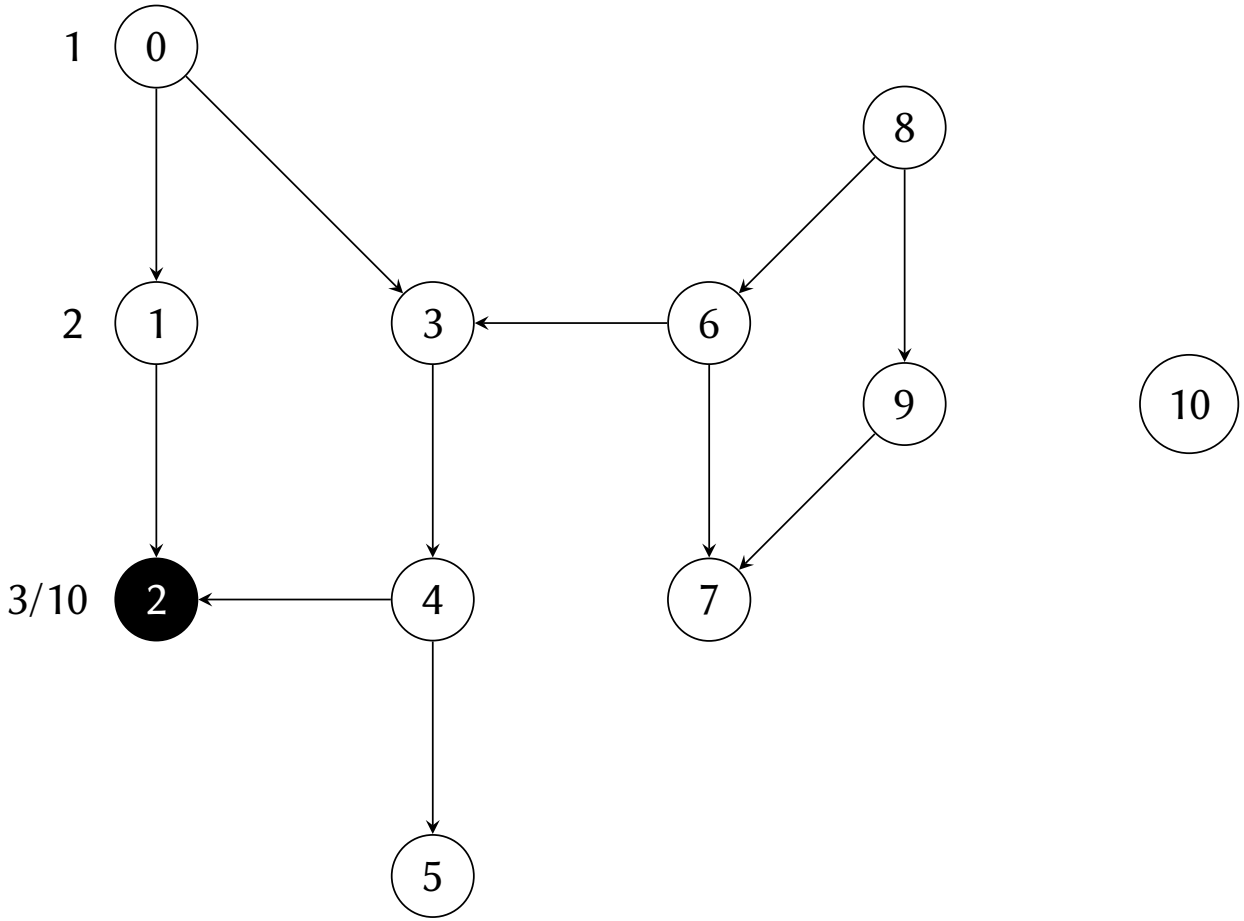
$$sorted = [\quad\quad]$$

Topological Sort Graph Traversal (4)



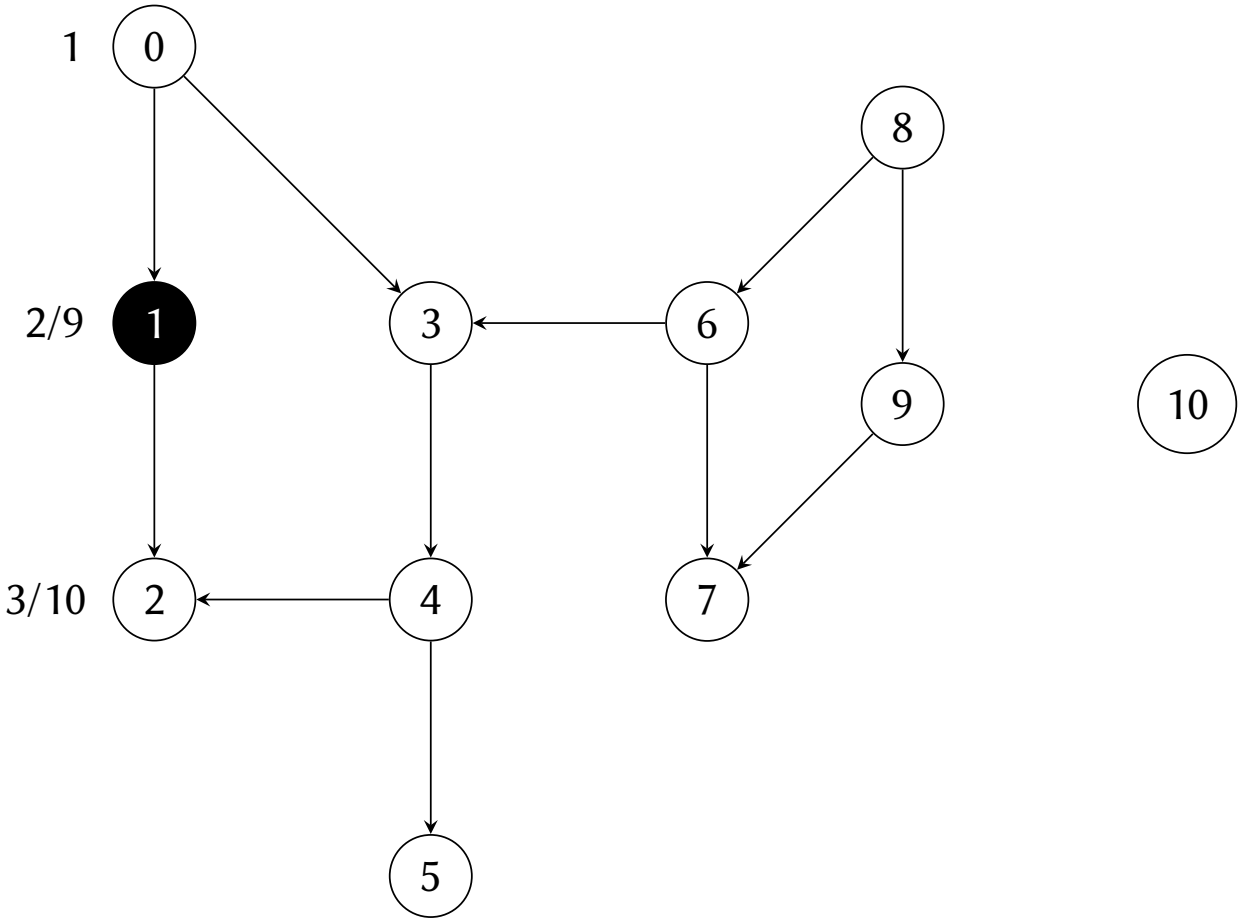
$$sorted = [\quad\quad]$$

Topological Sort Graph Traversal (5)



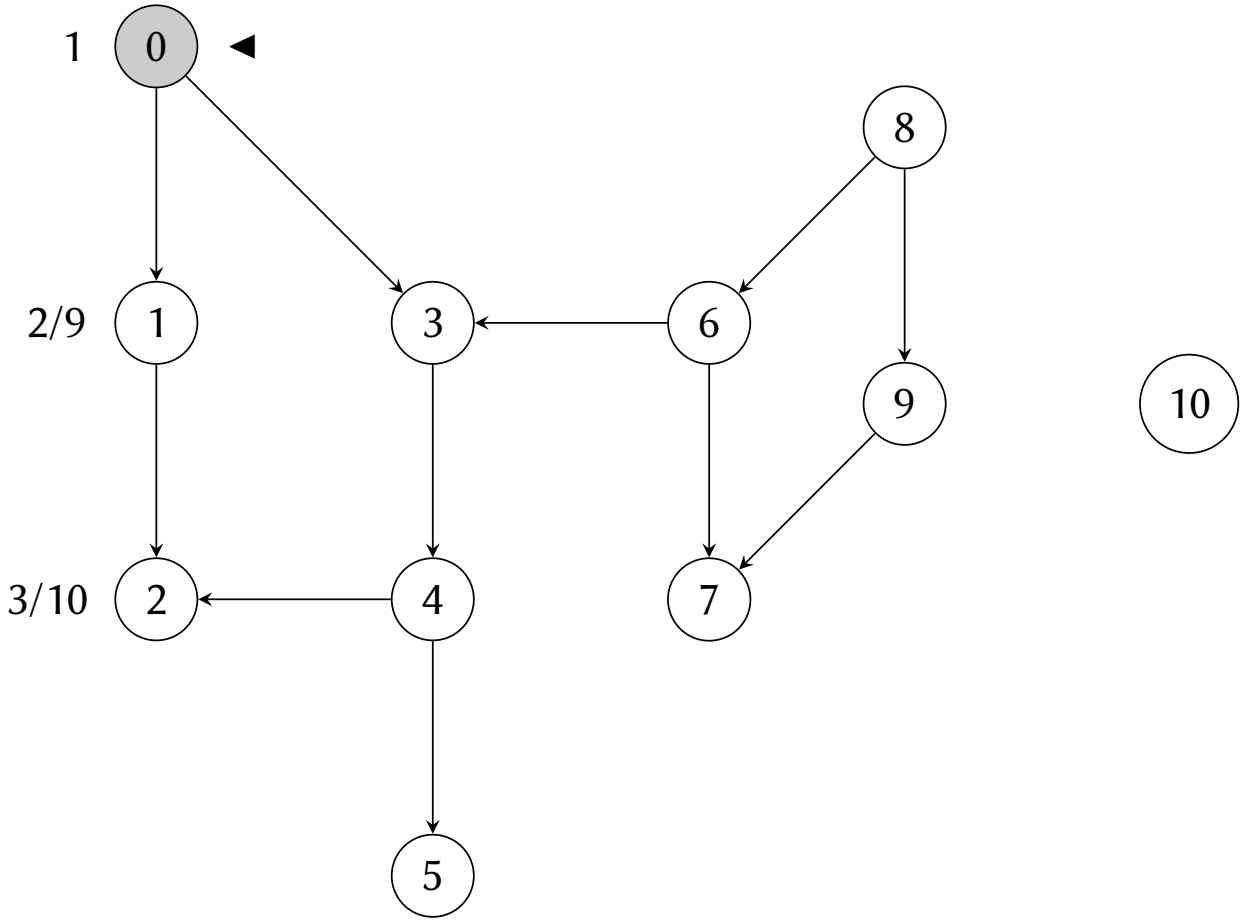
$$sorted = [2]$$

Topological Sort Graph Traversal (6)



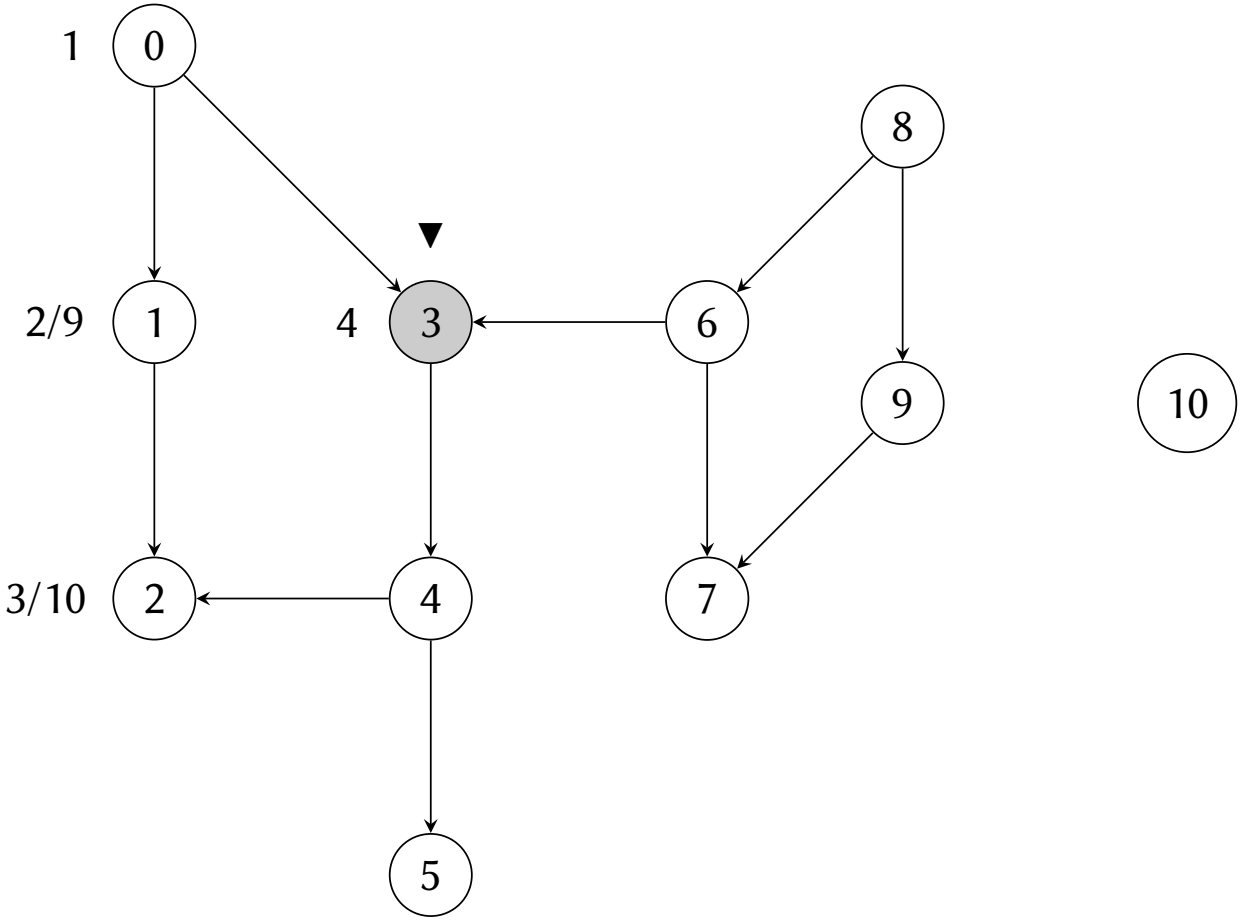
$$sorted = [1, 2]$$

Topological Sort Graph Traversal (7)



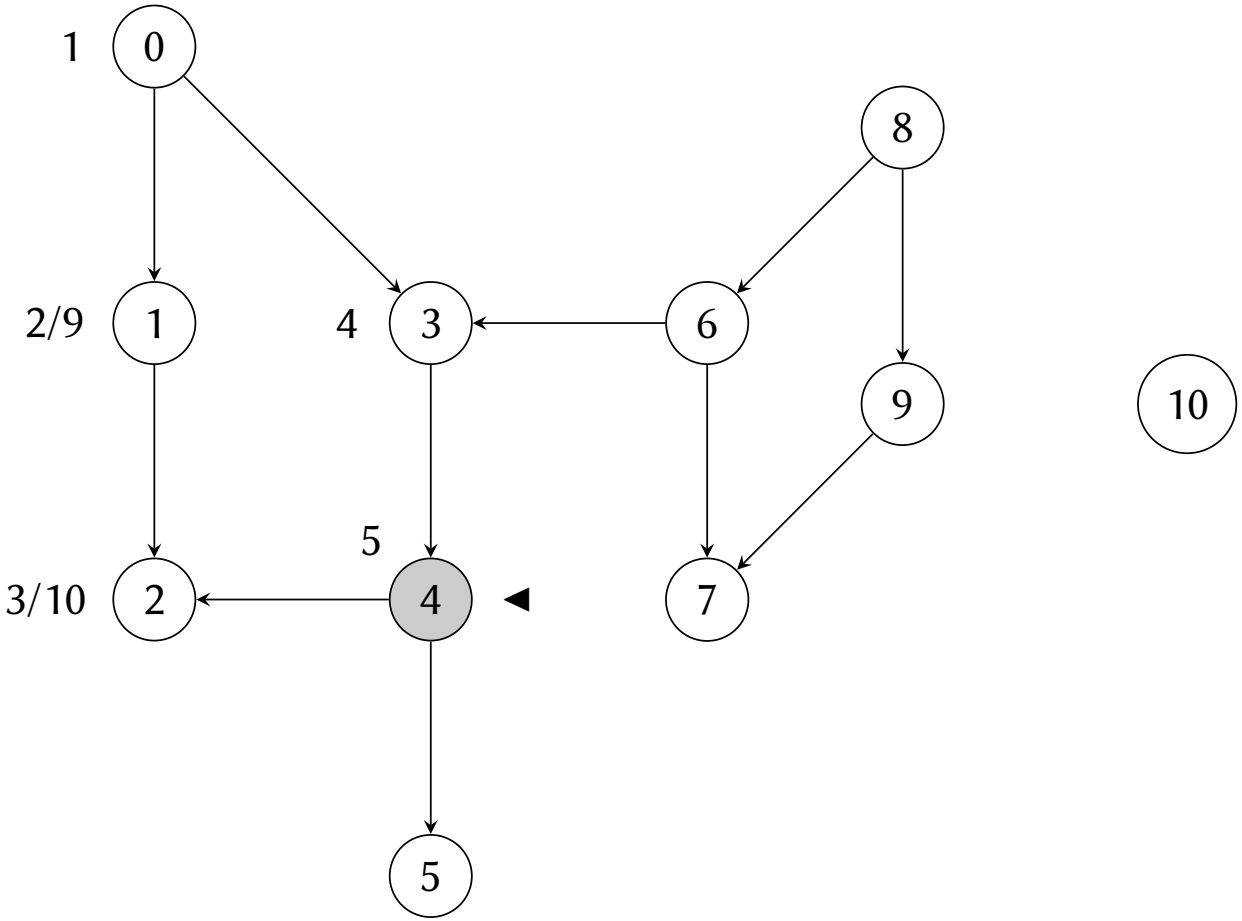
$$sorted = [1, 2]$$

Topological Sort Graph Traversal (8)



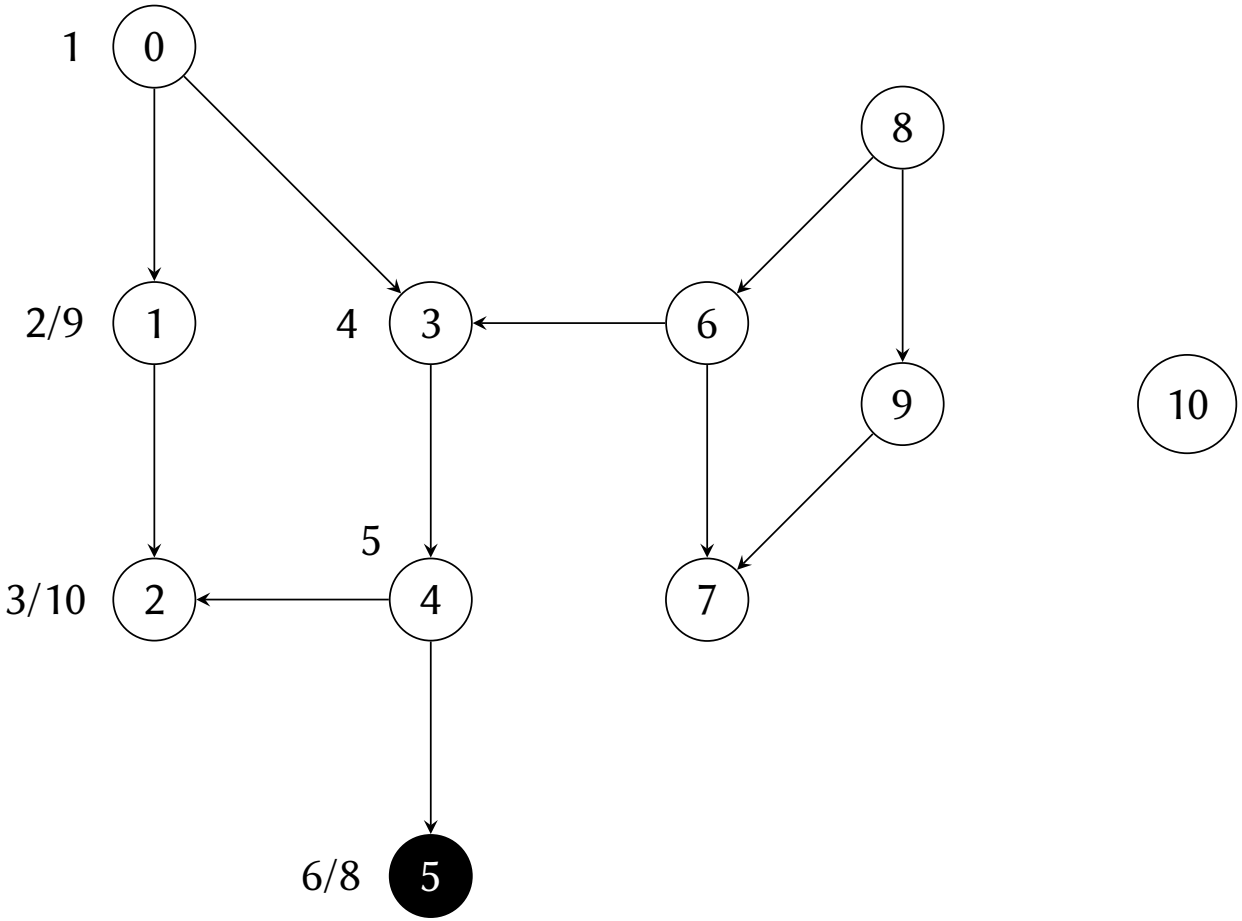
$$sorted = [1, 2]$$

Topological Sort Graph Traversal (9)



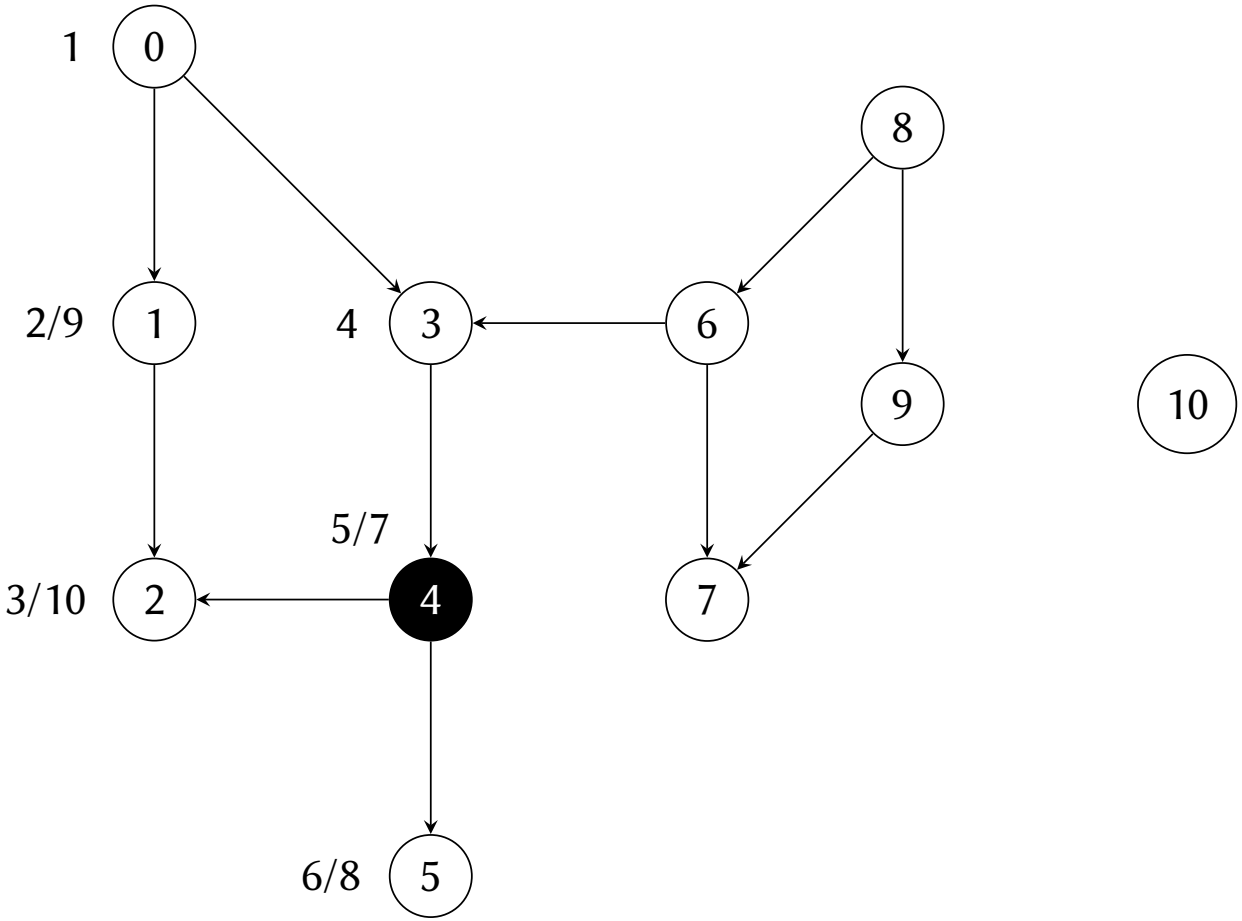
$$sorted = [1, 2]$$

Topological Sort Graph Traversal (10)



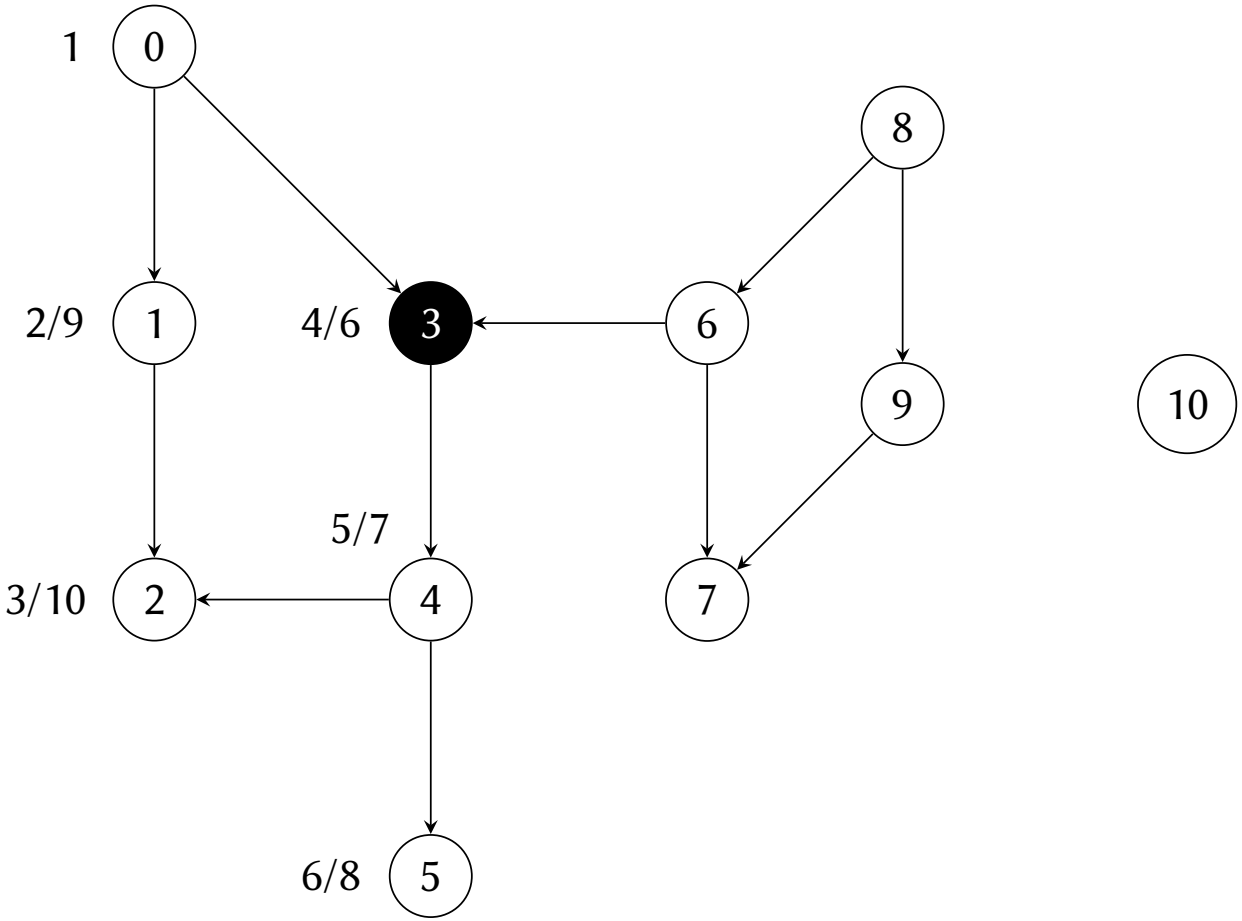
$$sorted = [5, 1, 2]$$

Topological Sort Graph Traversal (11)



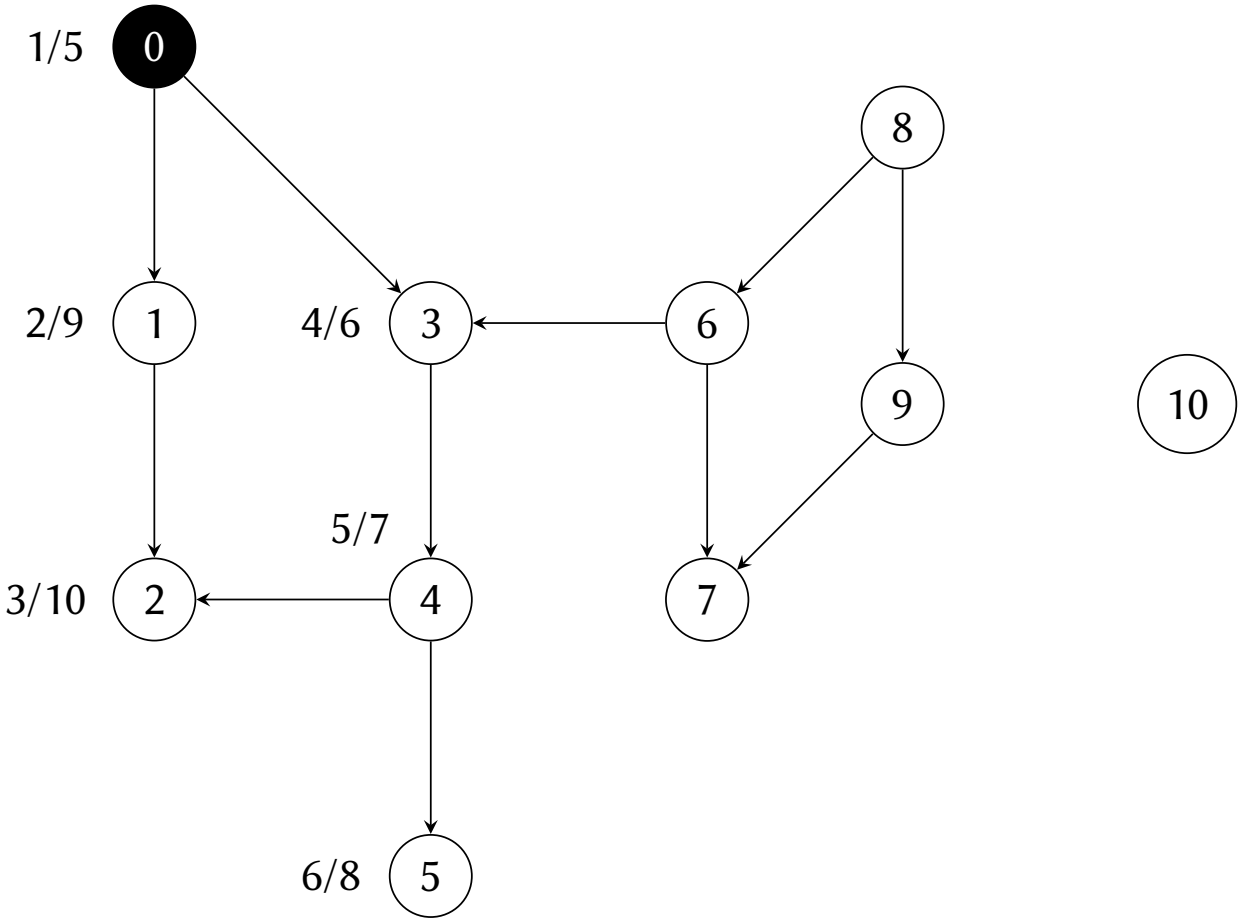
$$sorted = [4, 5, 1, 2]$$

Topological Sort Graph Traversal (12)



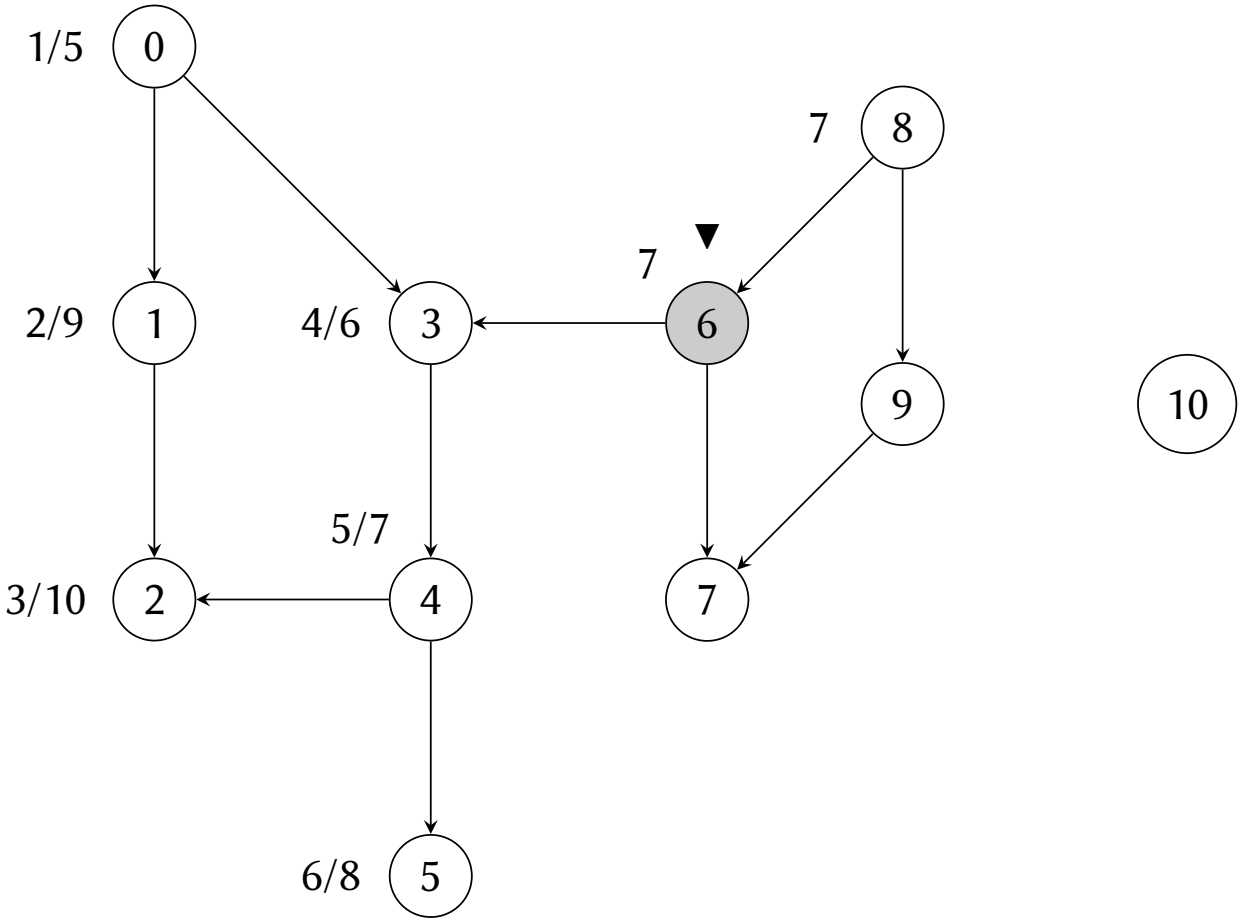
$$sorted = [3, 4, 5, 1, 2]$$

Topological Sort Graph Traversal (13)



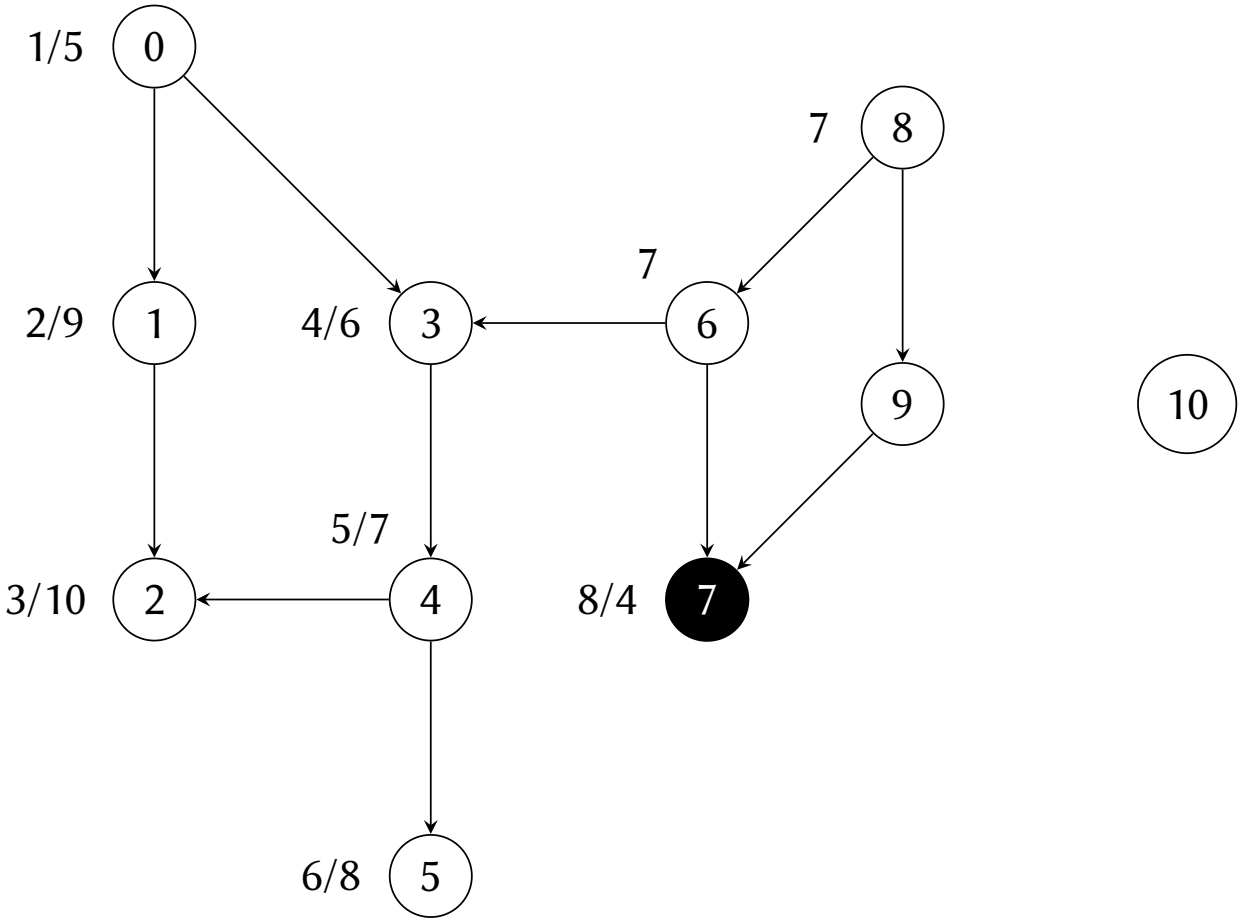
$$sorted = [0, 3, 4, 5, 1, 2]$$

Topological Sort Graph Traversal (14)



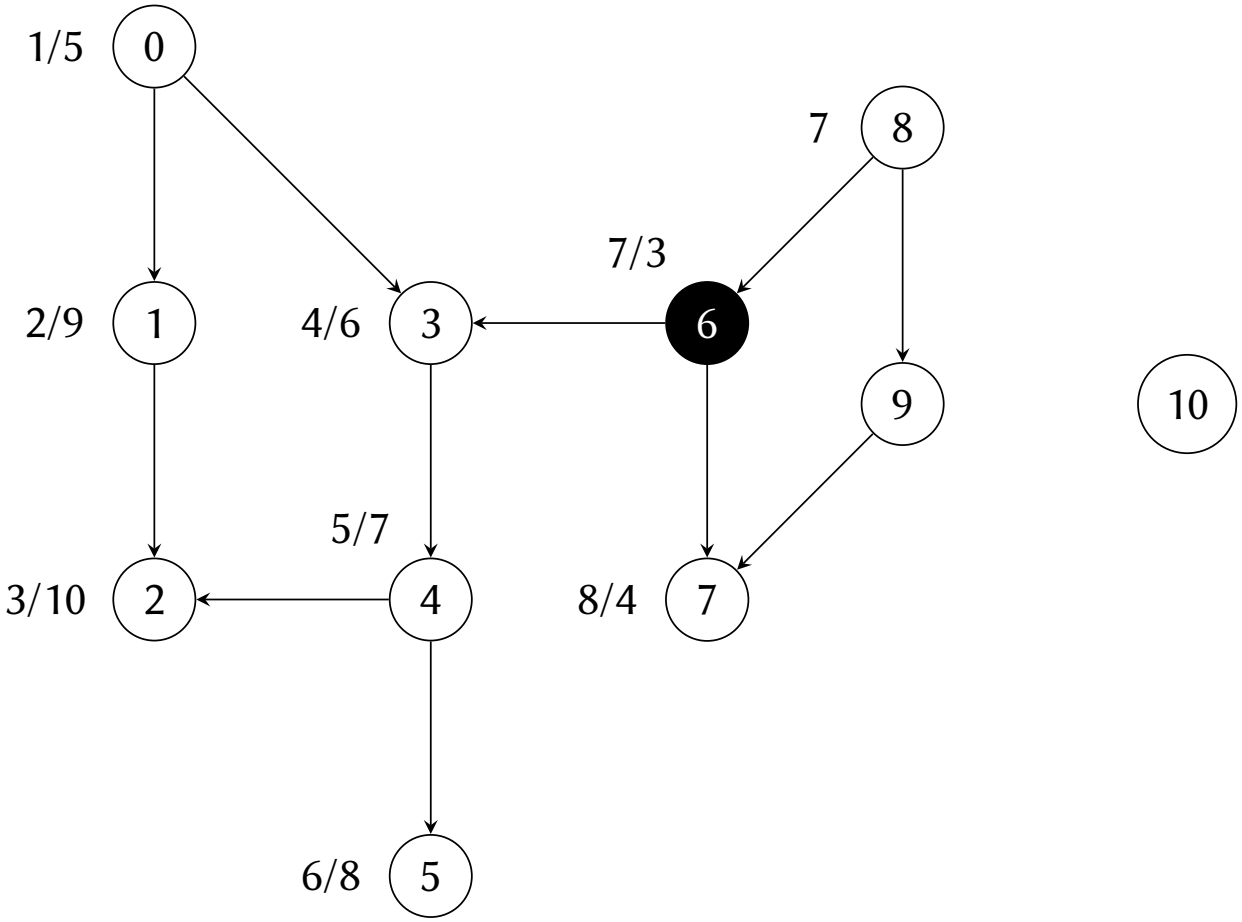
$$sorted = [0, 3, 4, 5, 1, 2]$$

Topological Sort Graph Traversal (15)



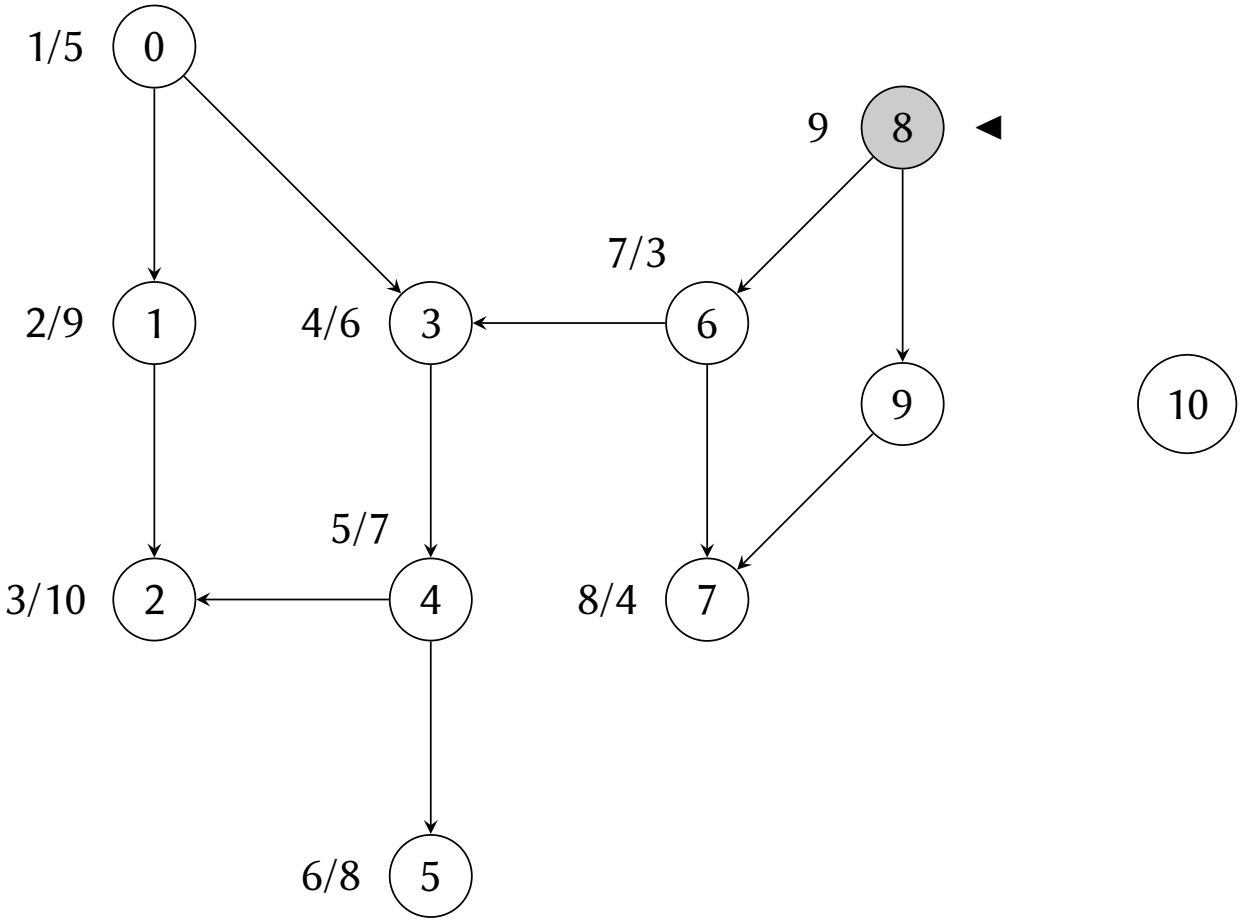
$$sorted = [7, 0, 3, 4, 5, 1, 2]$$

Topological Sort Graph Traversal (16)



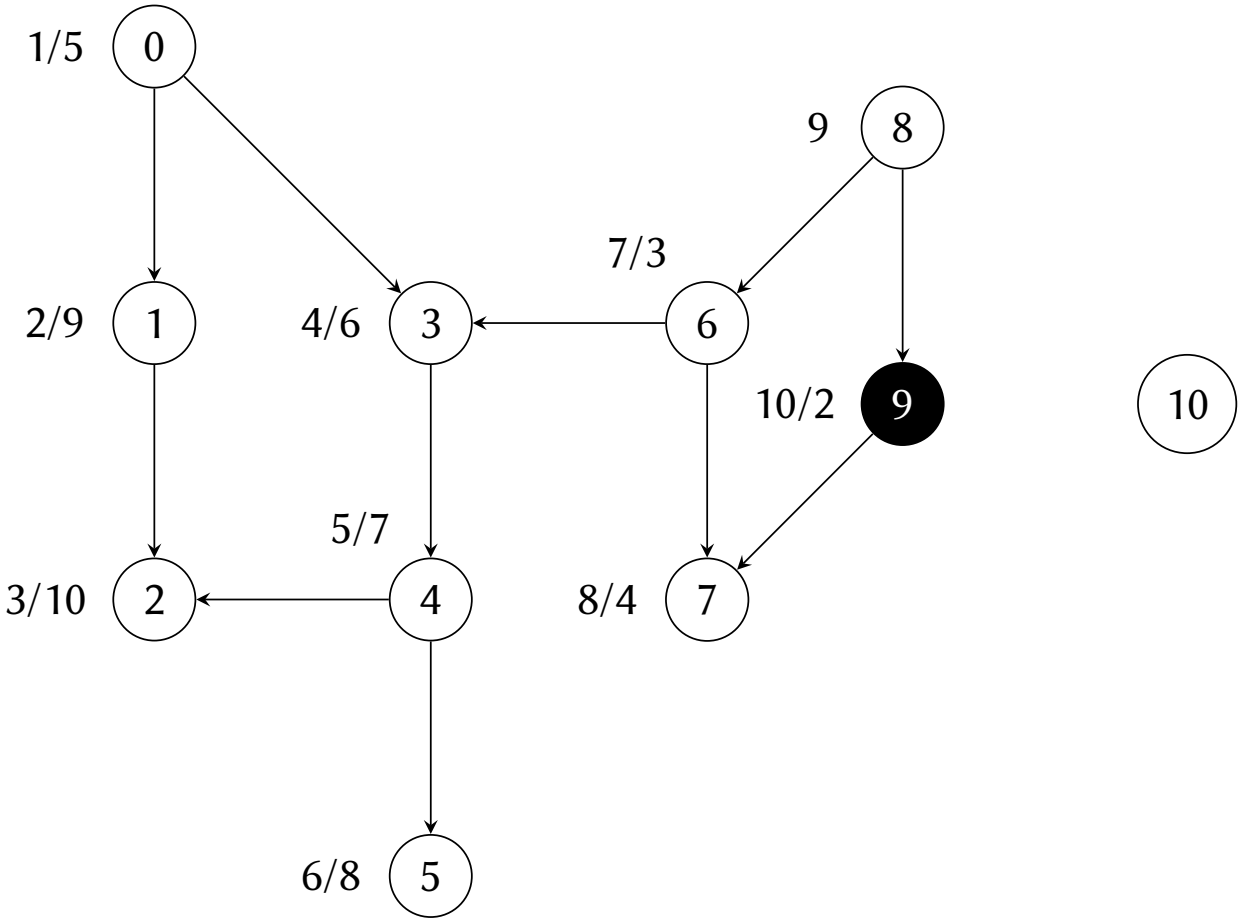
$$sorted = [6, 7, 0, 3, 4, 5, 1, 2]$$

Topological Sort Graph Traversal (17)



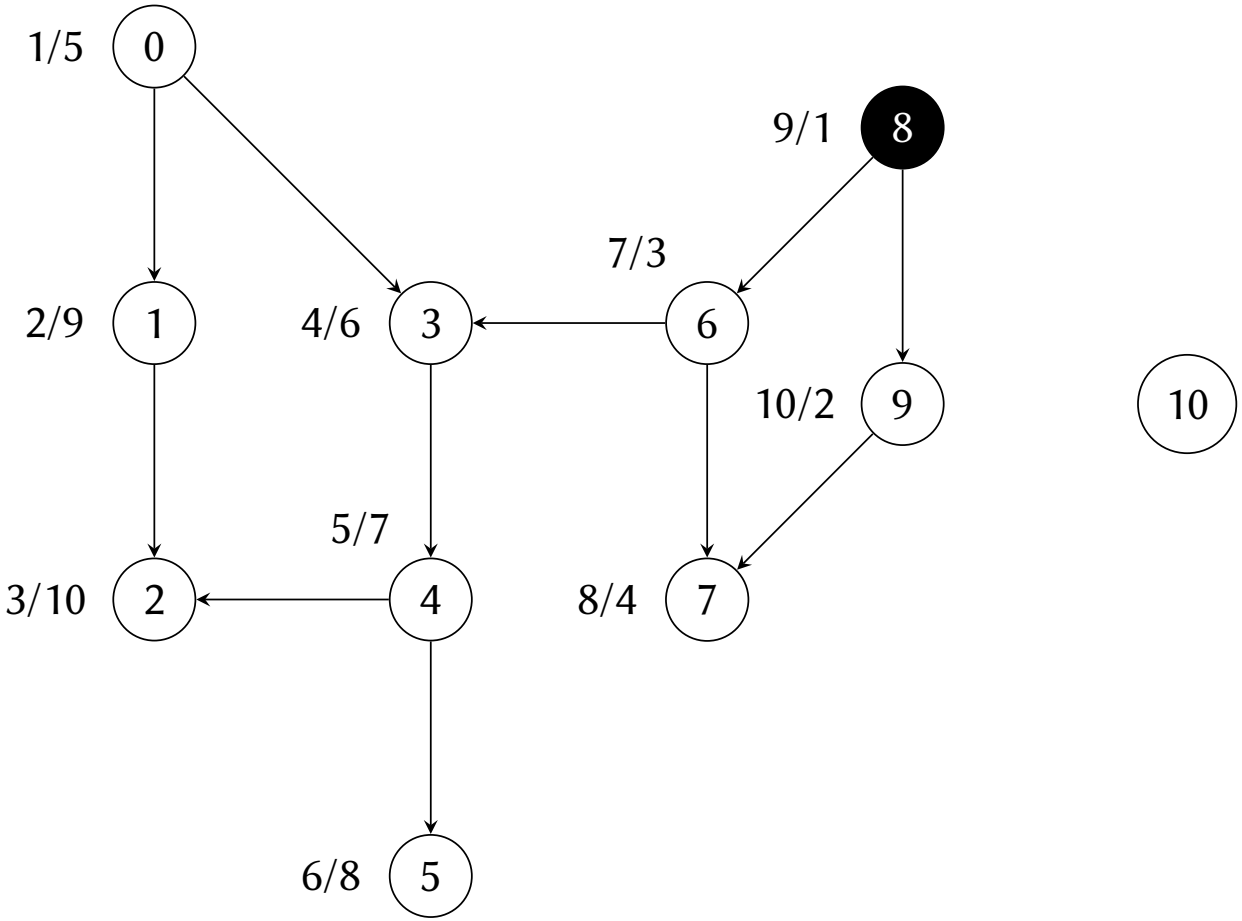
$$sorted = [6, 7, 0, 3, 4, 5, 1, 2]$$

Topological Sort Graph Traversal (18)



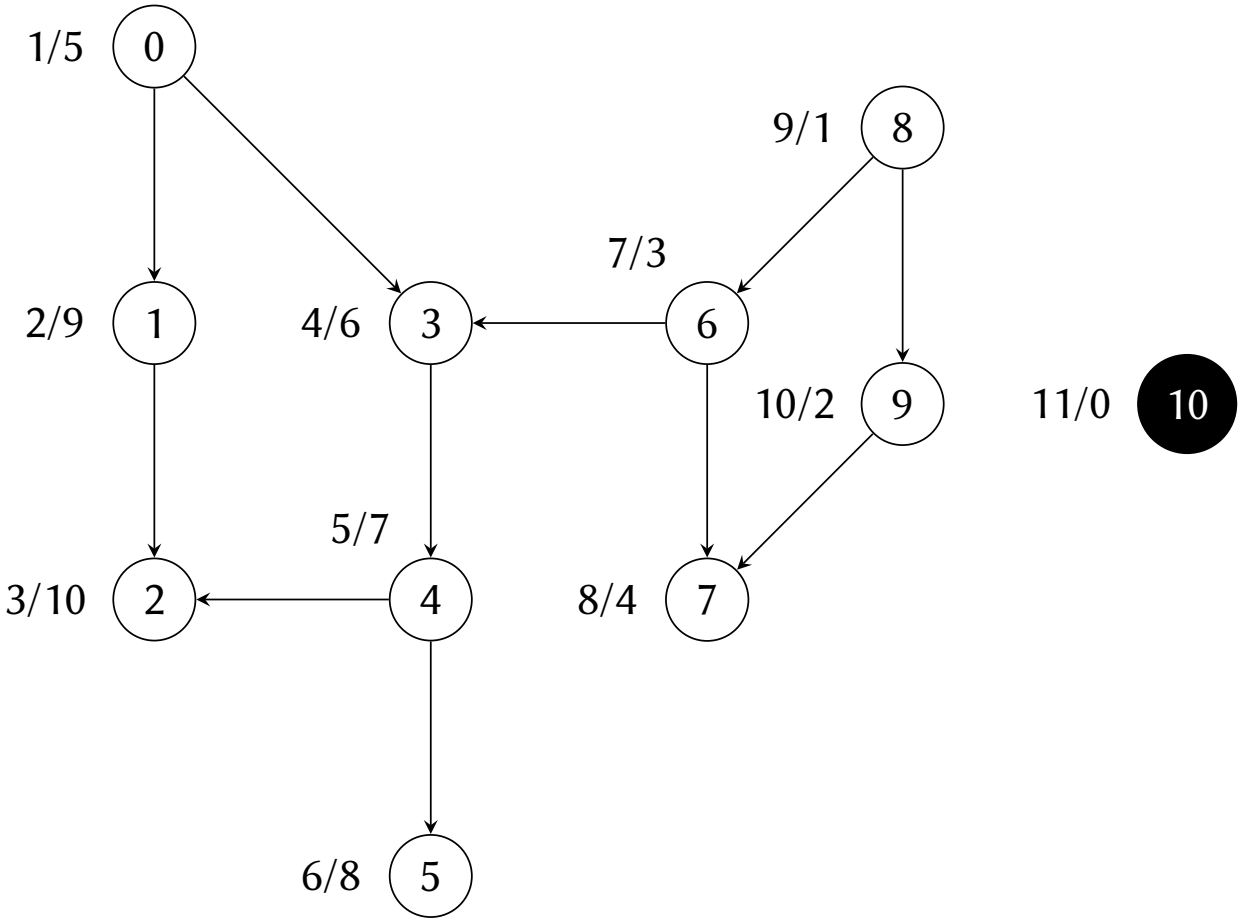
$$sorted = [9, 6, 7, 0, 3, 4, 5, 1, 2]$$

Topological Sort Graph Traversal (19)



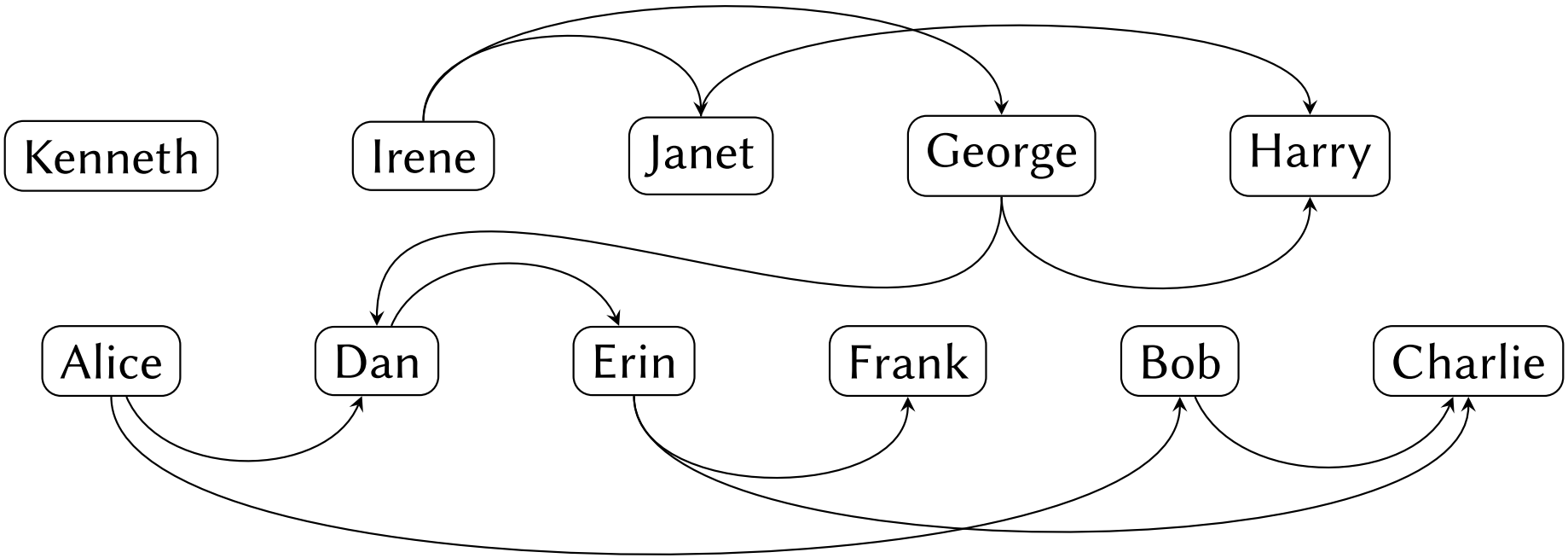
$$sorted = [8, 9, 6, 7, 0, 3, 4, 5, 1, 2]$$

Topological Sort Graph Traversal (20)



$$sorted = [10, 8, 9, 6, 7, 0, 3, 4, 5, 1, 2]$$

Topological Sort of Cast of Characters



Complexity Analysis

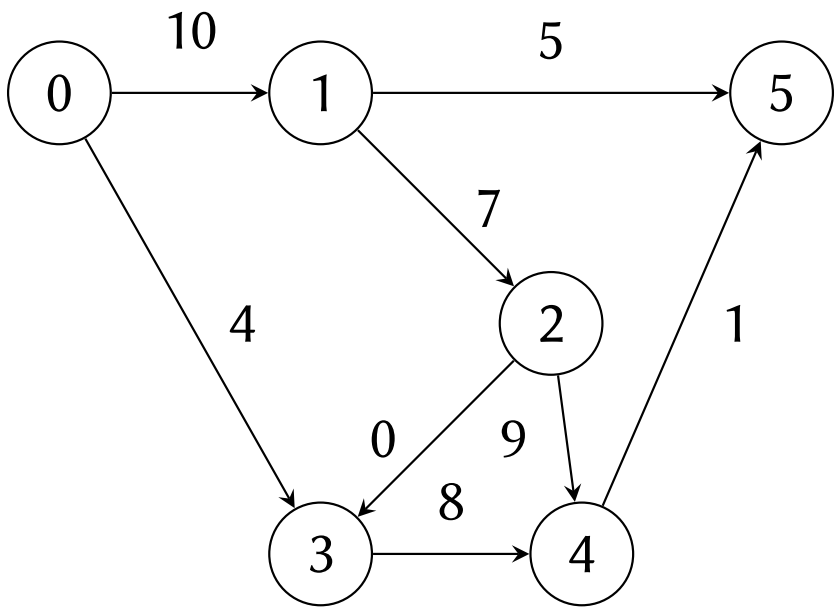
- The algorithm is, in essence, a depth-first traversal of the graph.
- Depth-first traversal requires time $\Theta(|V| + |E|)$.
- Therefore, topological sort requires time $\Theta(|V| + |E|)$.

Outline

Weighted Graphs

- We can extend graphs so that we assign a number, called *weight*, to each edge.
- In fact, we can view unweighted graphs as weighted graphs with all weights equal to one.
- We will write $w(u, v)$ for the weight of the edge (u, v).
- We use weights to represent more information. For example, if the graph is road network, the weights can be distances or time required between two points.

Weighted Graph Example



Weights

- The weights need not be positive integers.
- They can also be negative, representing, for instance, some penalty or cost.
- They can also be real numbers; the application dictates the kind of weight to use.
- In a weighted graph, the length of a path is not the number of edges, but the sum of the weights of the edges.
- So, in the previous graph, the graph from 0 to 2 has length 17, not 2.

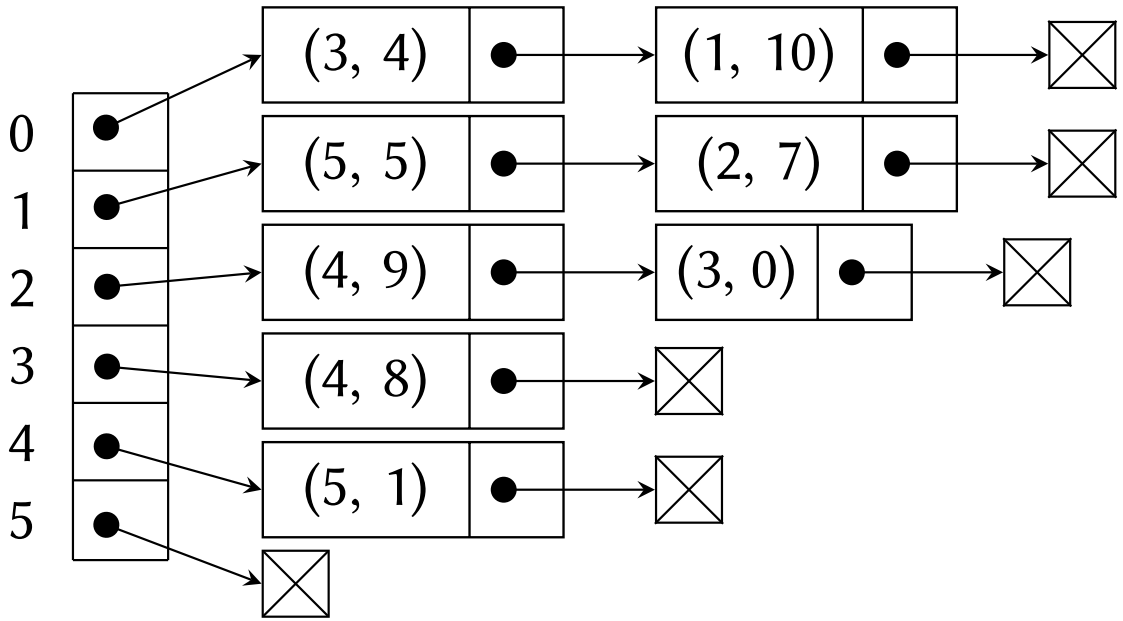
Weighted Graph Representation

- In the adjacency matrix representation we put at each cell the corresponding weight, instead of 1. When two vertices are not connected, we put a special value, such as ∞ or -1, depending on the application.
- In the adjacency list representation we can put the weight as an extra piece of data in each node in a list.

Adjacency Matrix for Weighted Graph

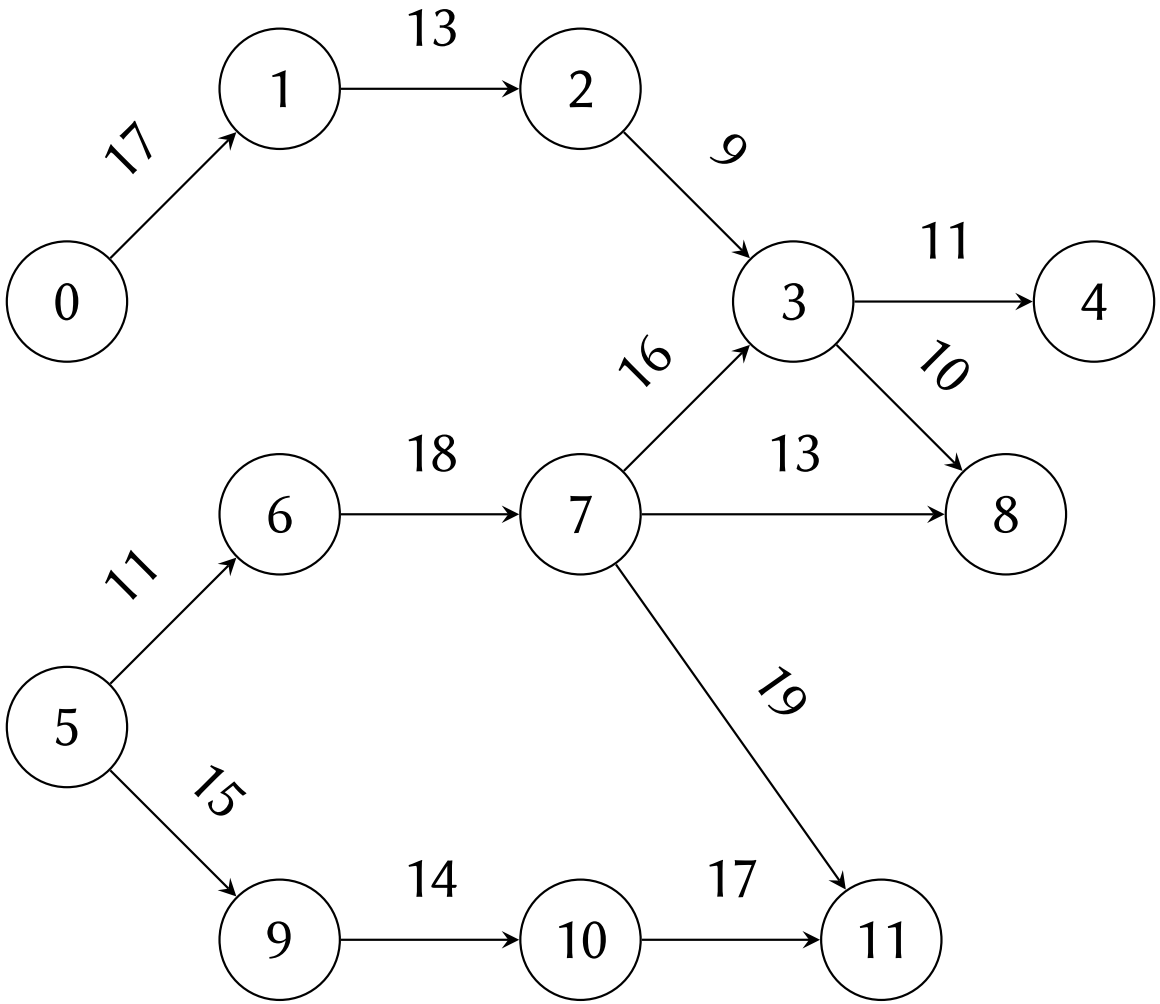
	0	1	2	3	4	5
0	∞	10	∞	4	∞	∞
1	∞	∞	7	∞	∞	5
2	∞	∞	∞	0	9	∞
3	∞	∞	∞	∞	8	∞
4	∞	∞	∞	∞	∞	1
5	∞	∞	∞	∞	∞	∞

Adjacency Lists for Weighted Graph

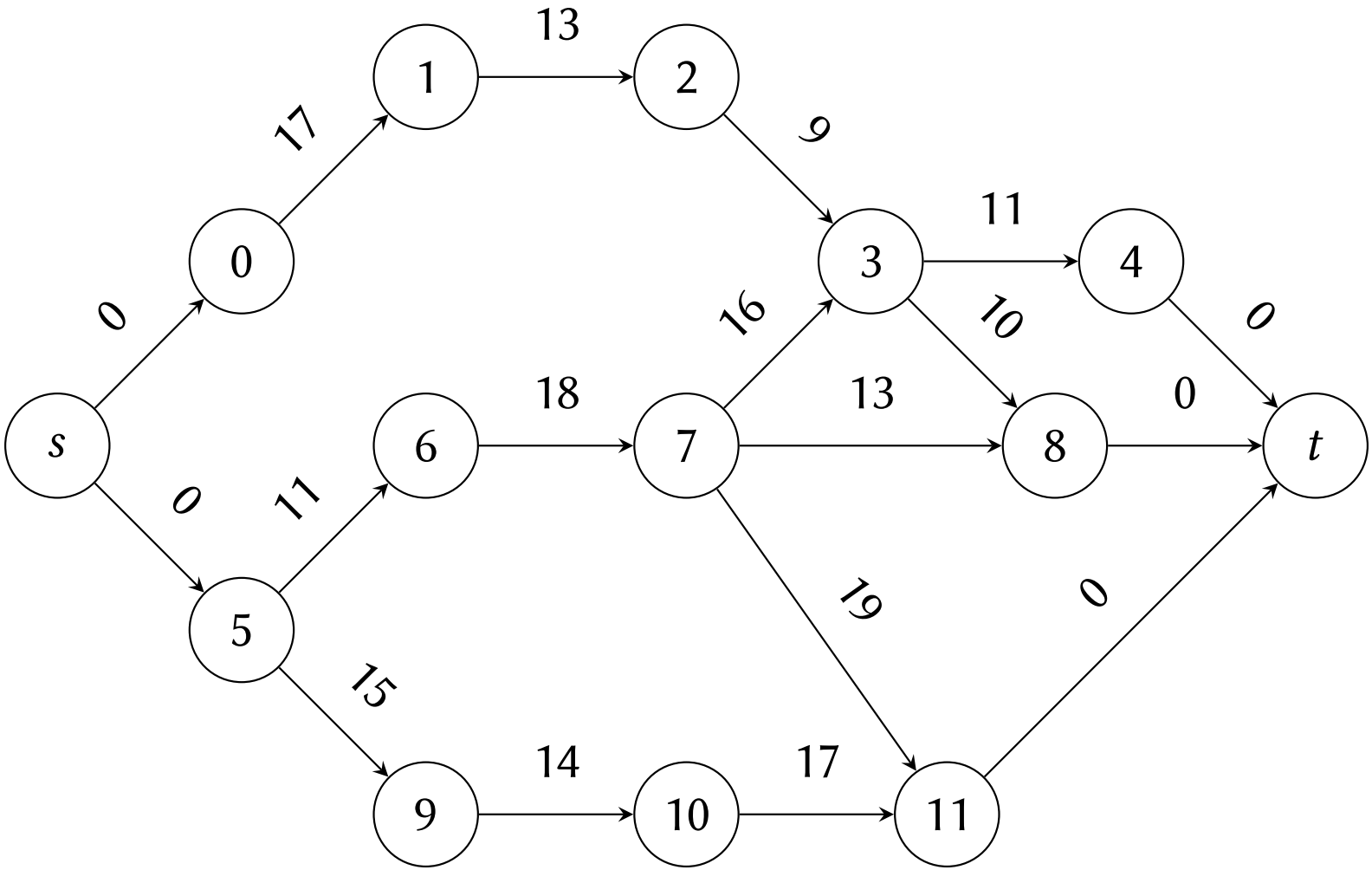


Outline

Task Scheduling Graph

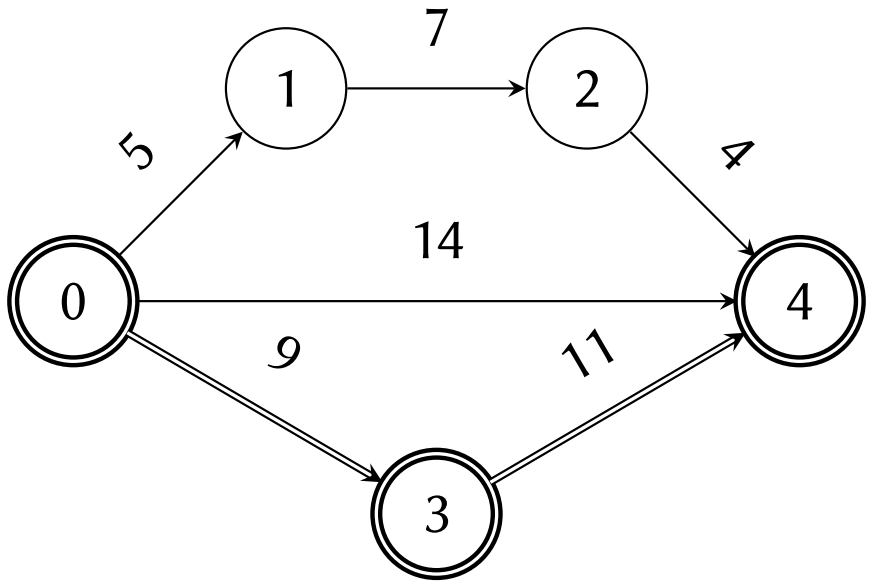


Source and Sink Nodes



Outline

Critical Path



Definition

The longest path between two vertices in a scheduling graph is a a *critical path*. That is because we cannot finish before we complete the path that requires that longest amount of time.

Importance of the Critical Path

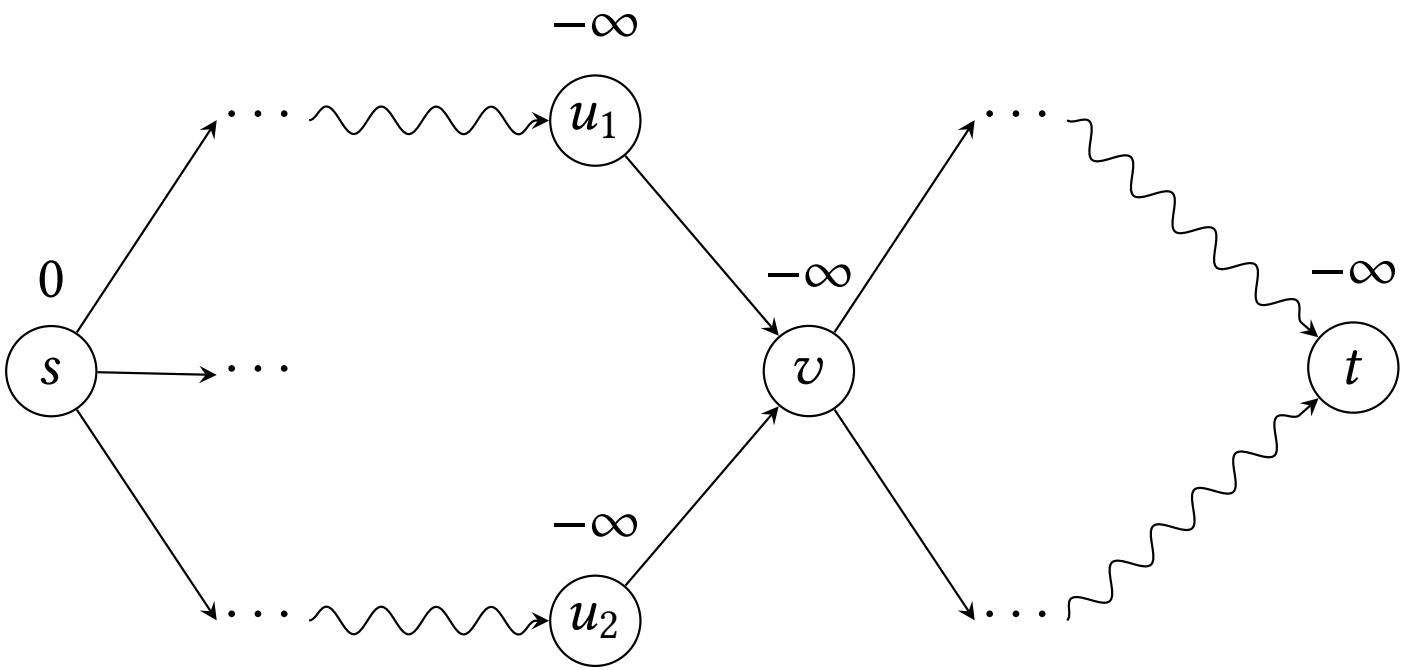
- In a scheduling graph with source and sink we calculate the critical path between them.
- We must give particular importance to the processes in the critical path, as we will not be able to finish before the time required by the critical path.
- Speeding up the processes outside the critical path will not affect the overall completion time.
- Slowing down any process in the critical path will slow down the whole project.

Relaxation

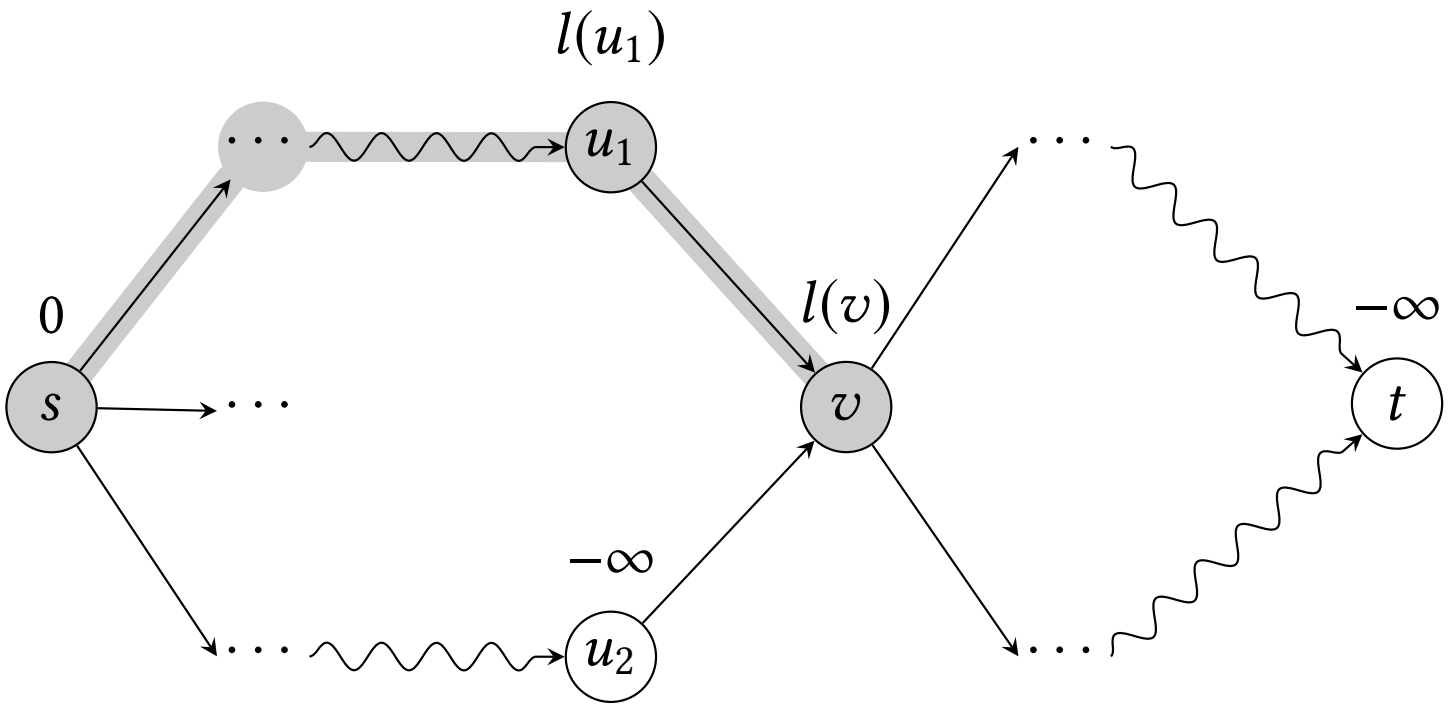
Definition

In many graph algorithms we start by giving some extreme initial values to a metric and then we gradually improve our estimate of those values. This technique is called *relaxation*.

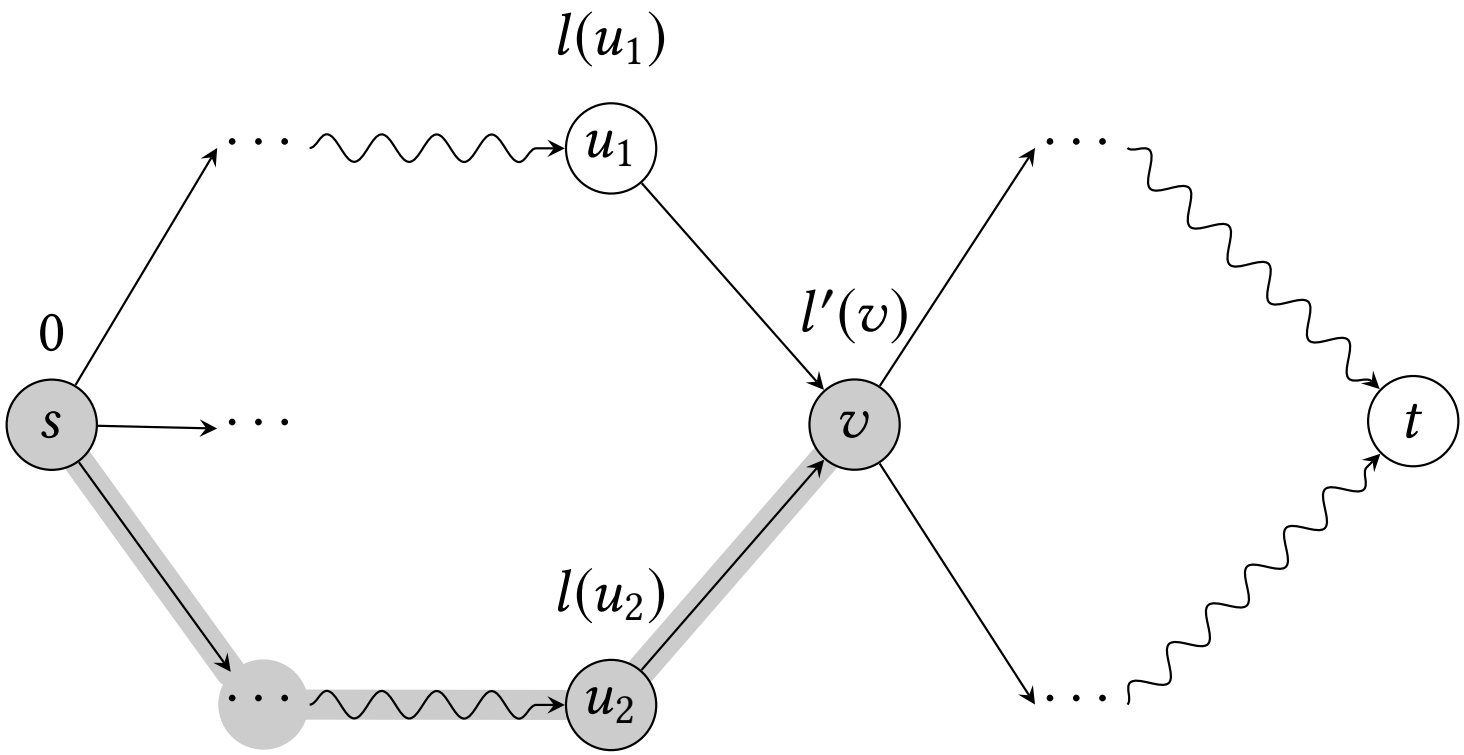
Relaxation Example (1)



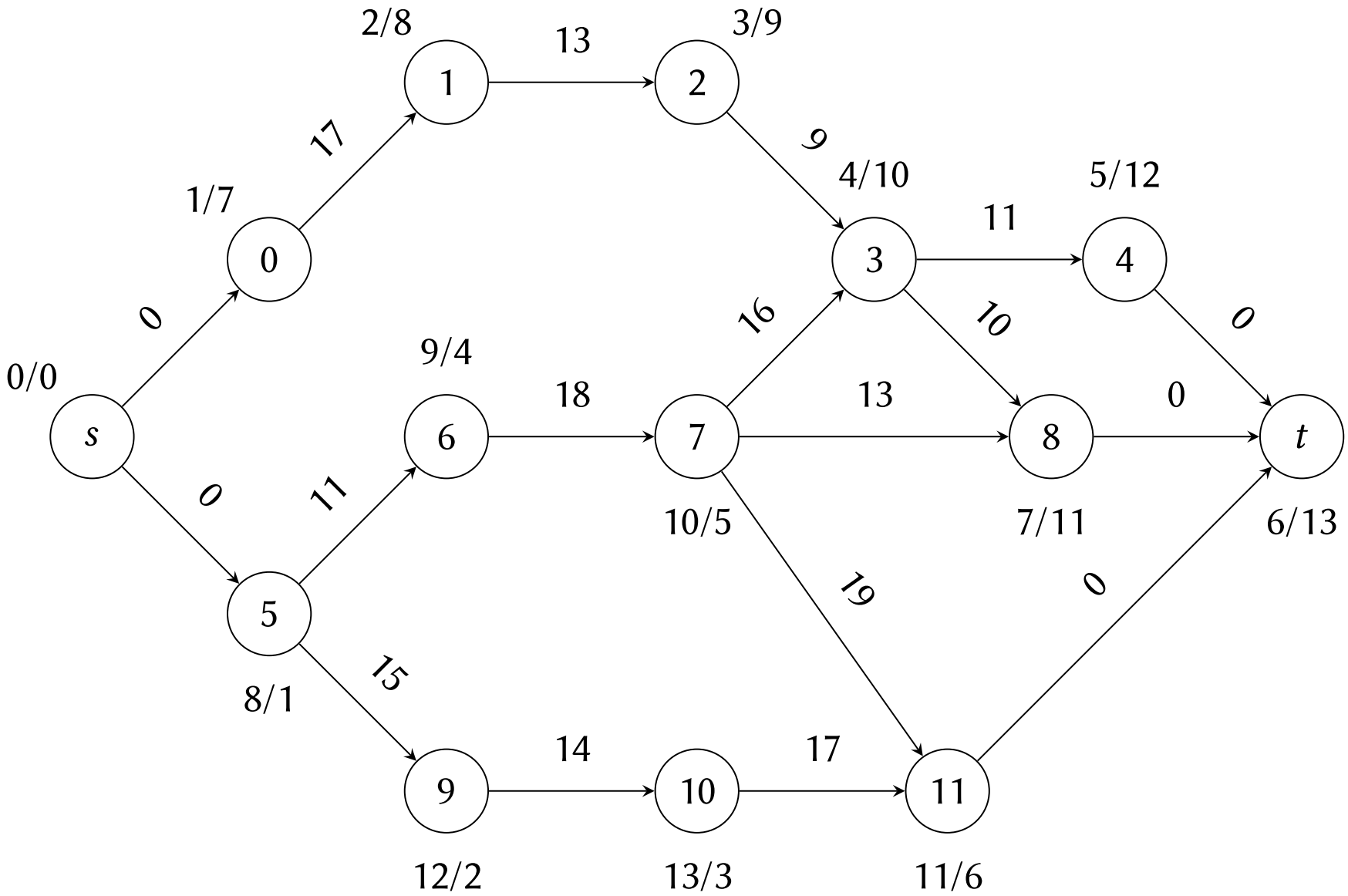
Relaxation Example (2)



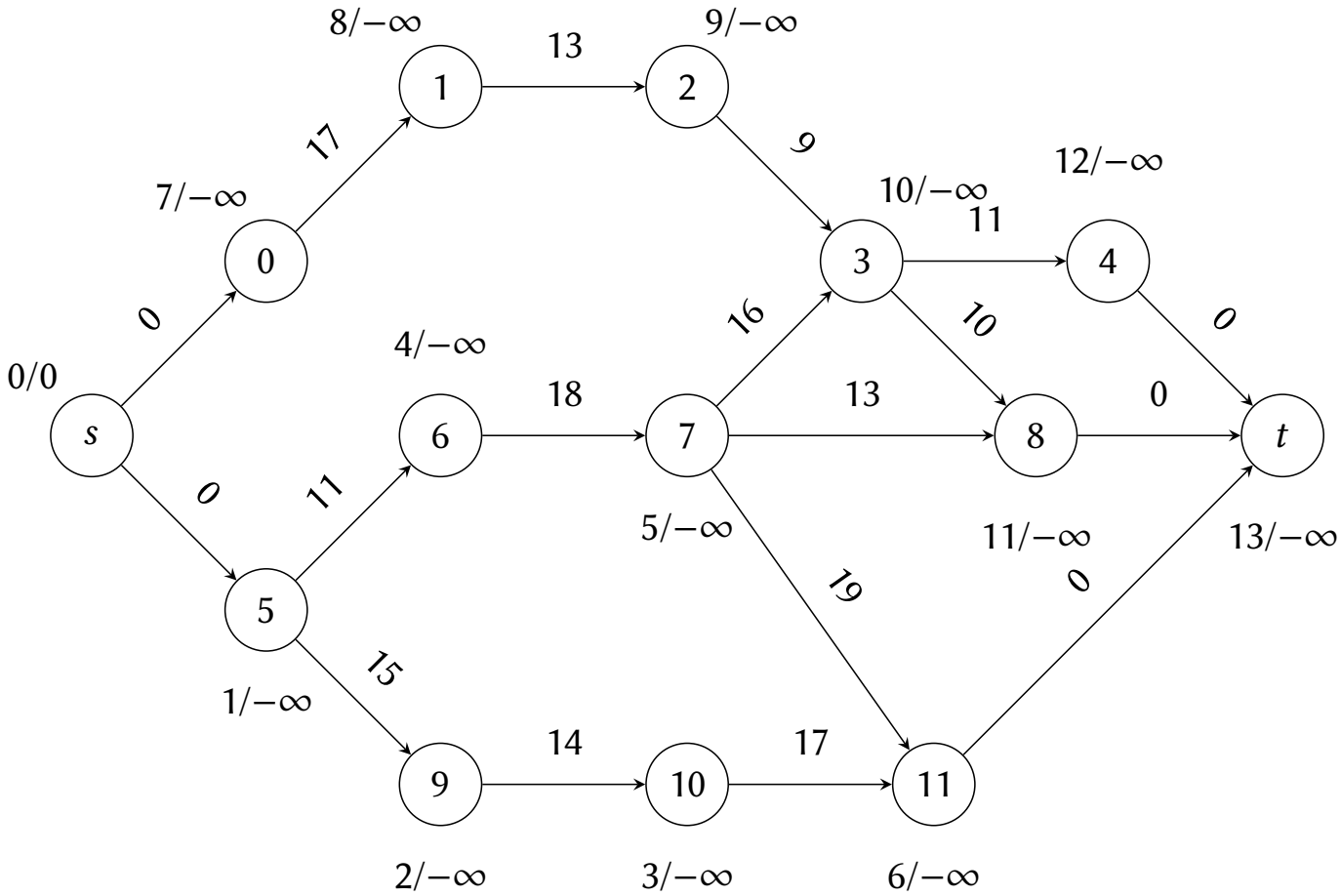
Relaxation Example (3)



The Task Scheduling Graph in Topological Order

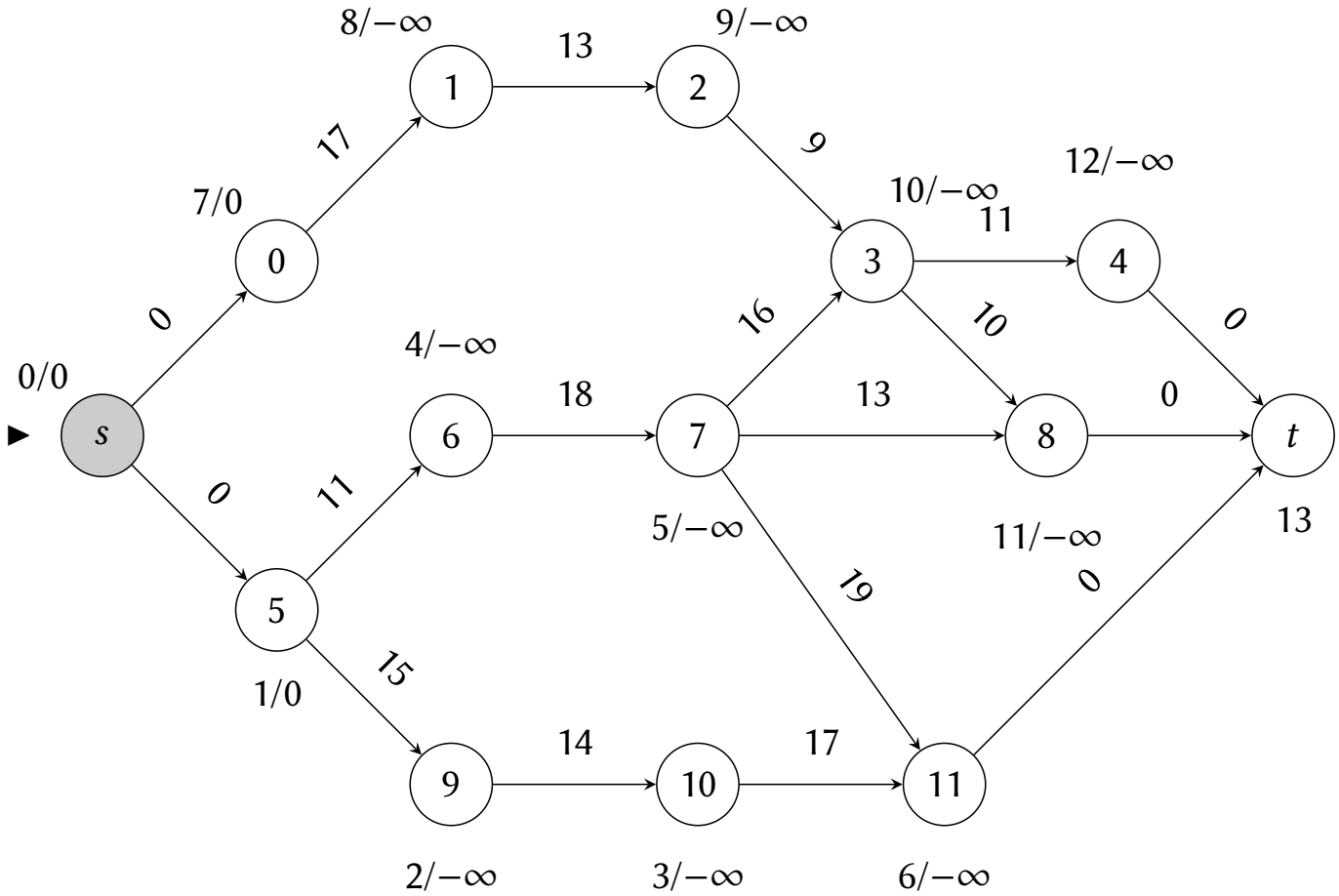


Critical Path Example (1)



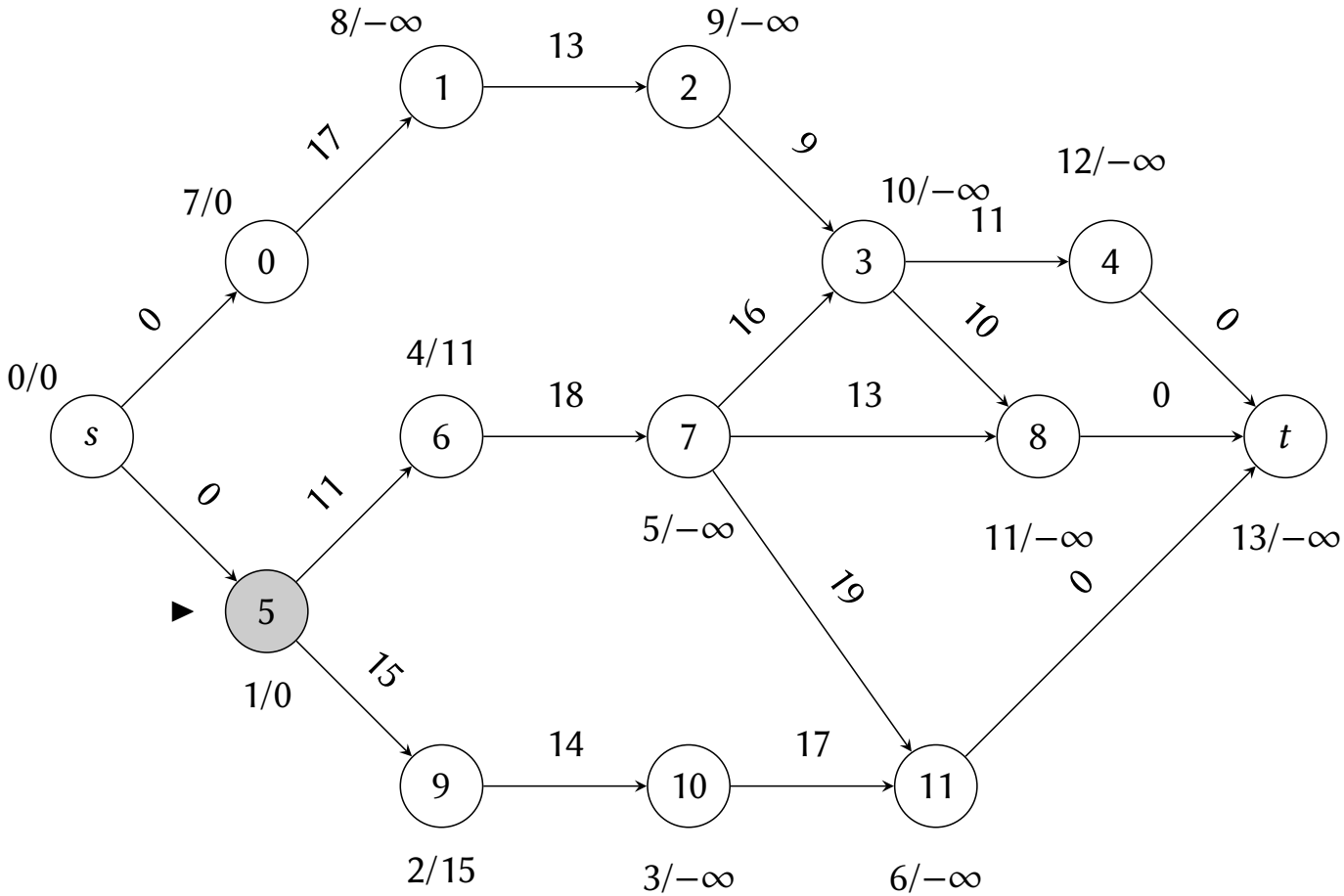
s	0	1	2	3	4	5	6	7	8	9	10	11	t

Critical Path Example (2)



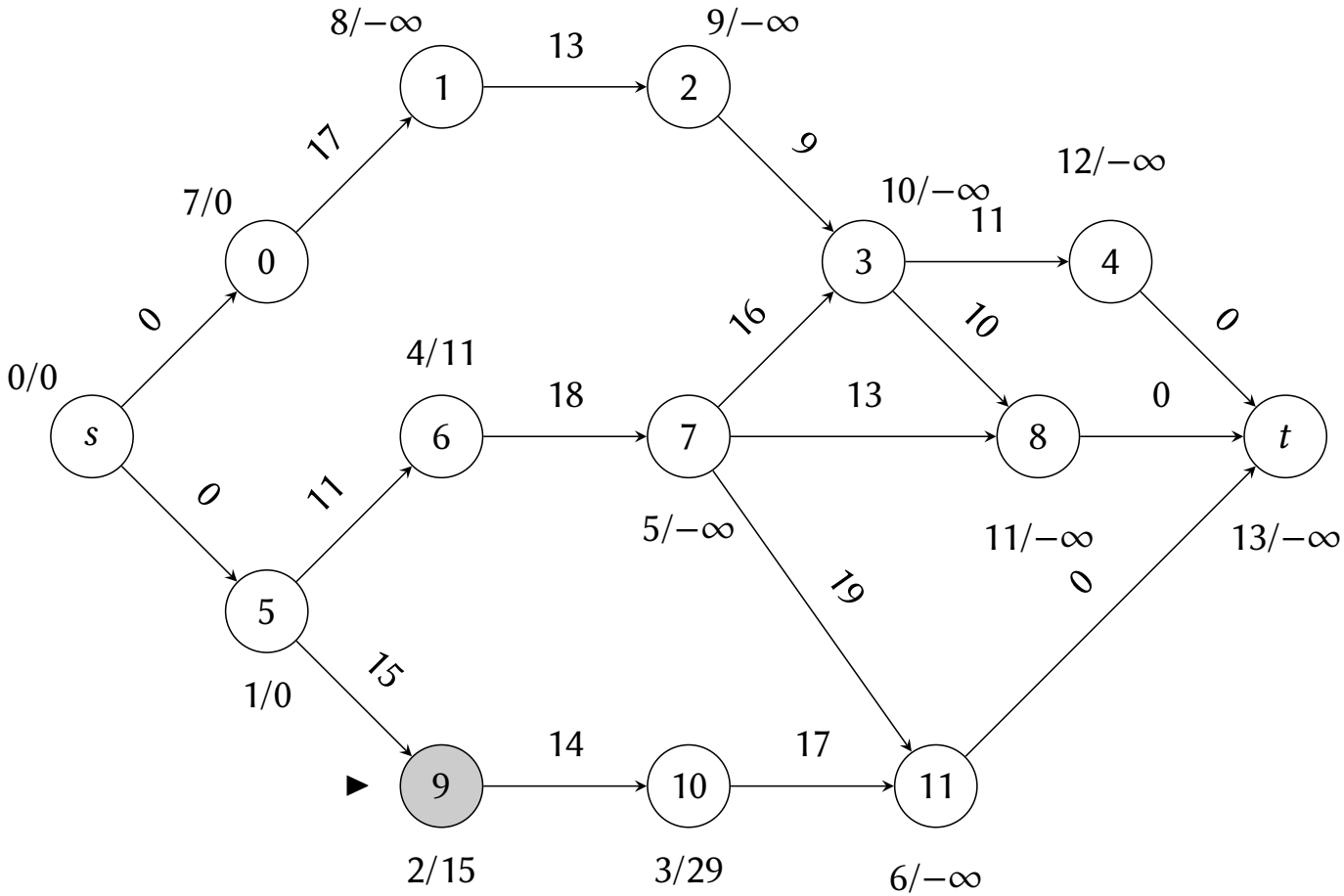
s	0	1	2	3	4	5	6	7	8	9	10	11	t
	s					s							

Critical Path Example (3)



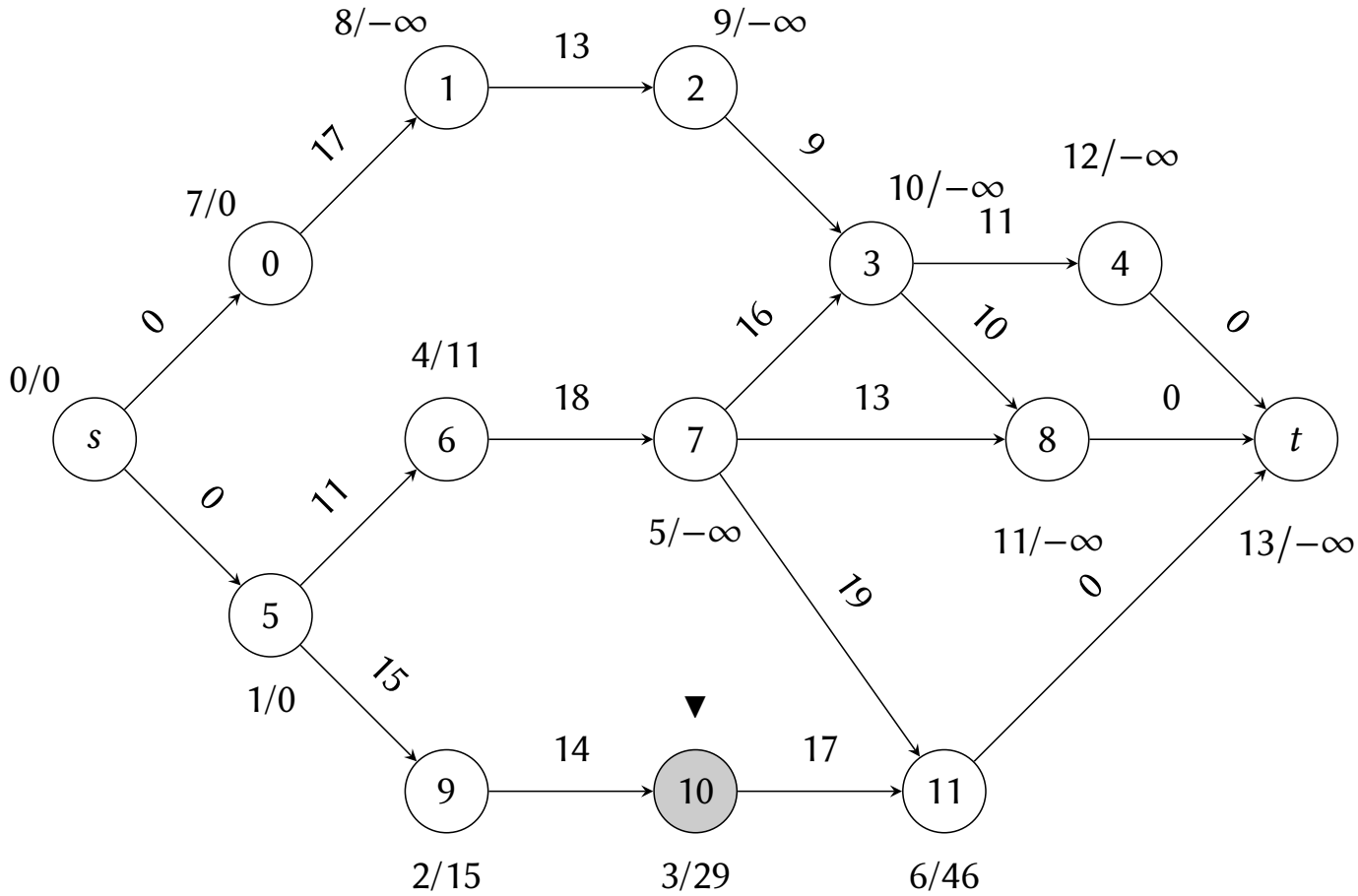
s	0	1	2	3	4	5	6	7	8	9	10	11	t
	s					s	5			5			

Critical Path Example (4)



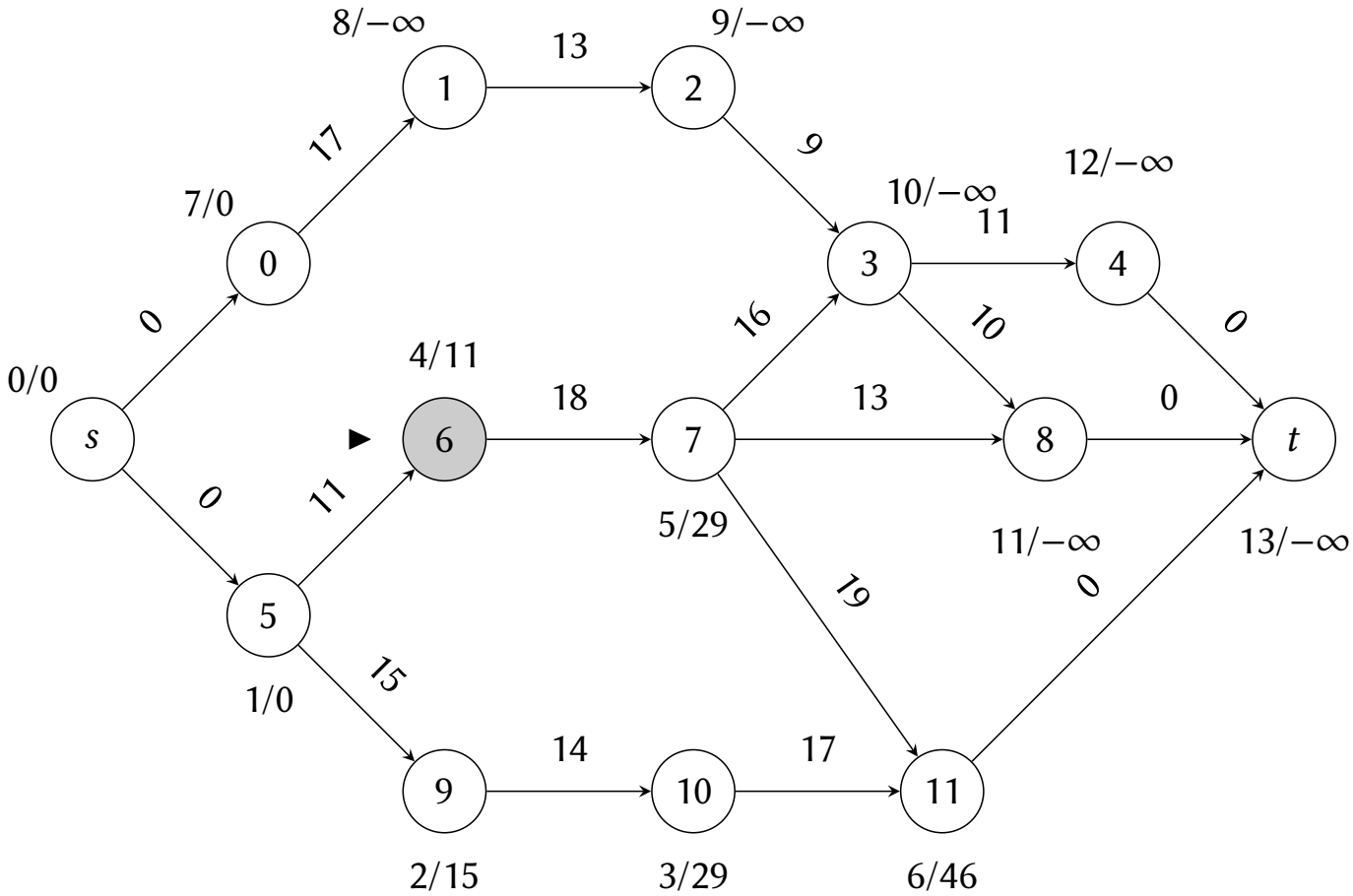
s	0	1	2	3	4	5	6	7	8	9	10	11	t
	s					s	5			5	9		

Critical Path Example (5)



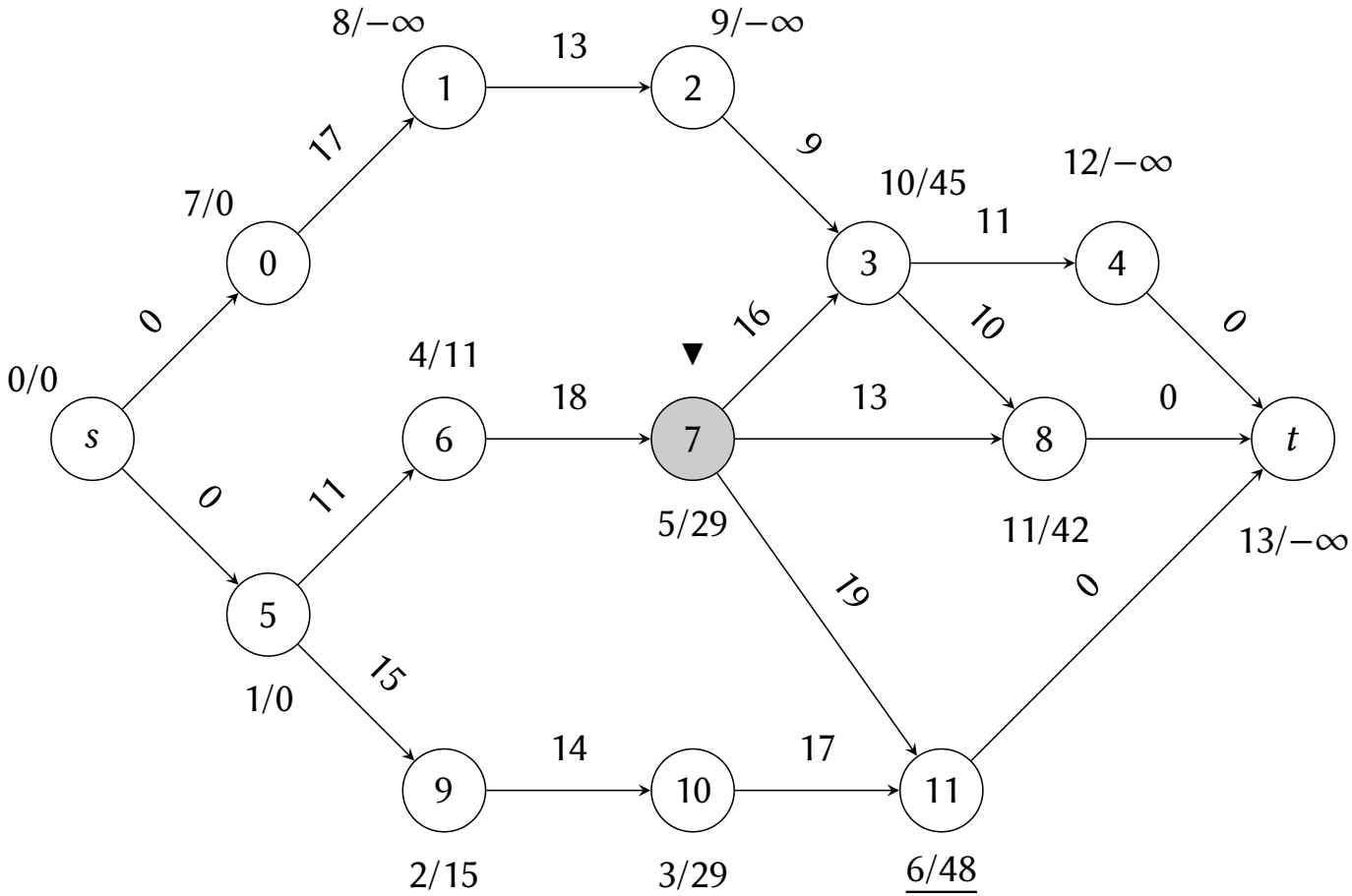
s	0	1	2	3	4	5	6	7	8	9	10	11	t
	s					s	5			5	9	10	

Critical Path Example (6)



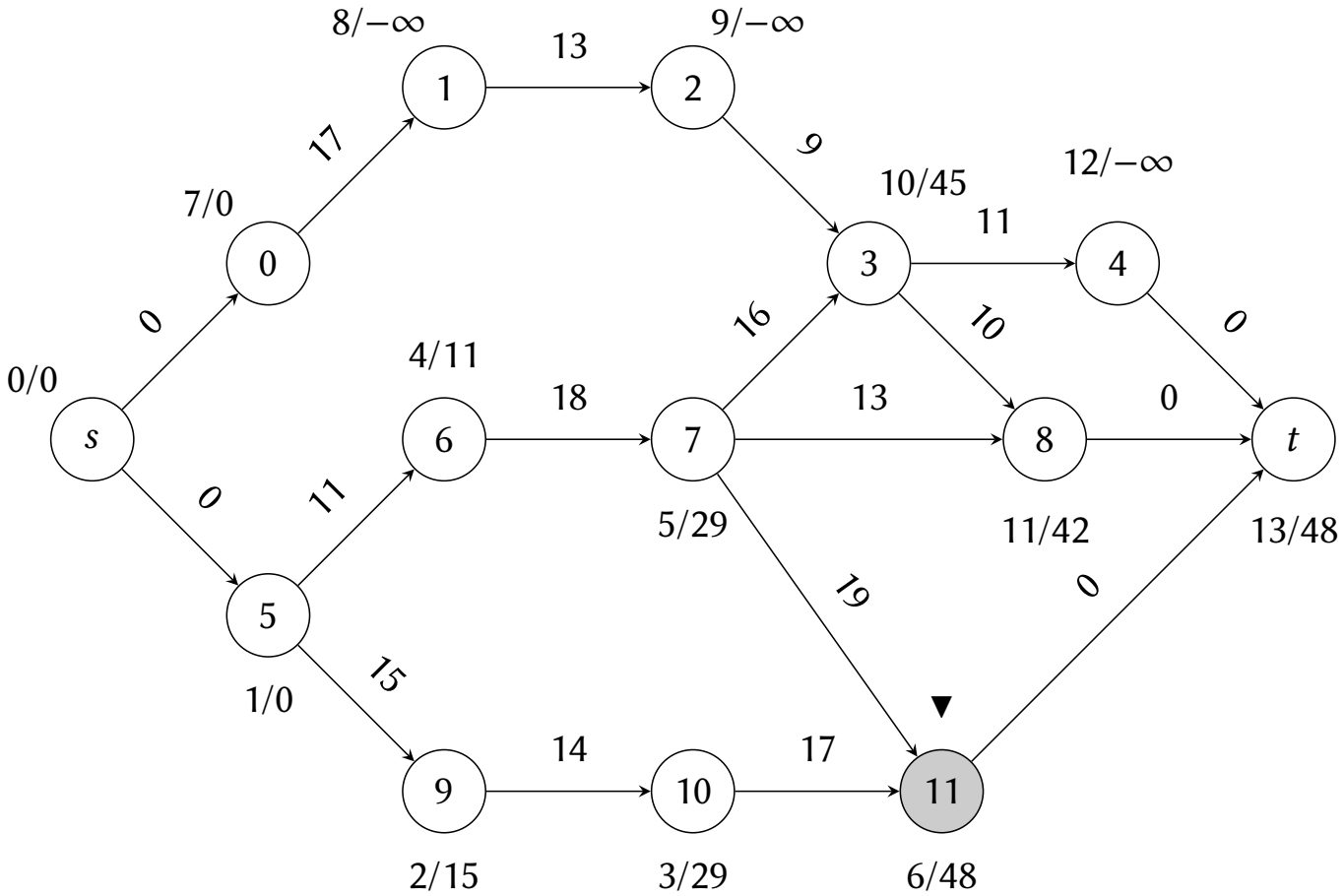
s	0	1	2	3	4	5	6	7	8	9	10	11	t
	s					s	5	6		5	9	10	

Critical Path Example (7)



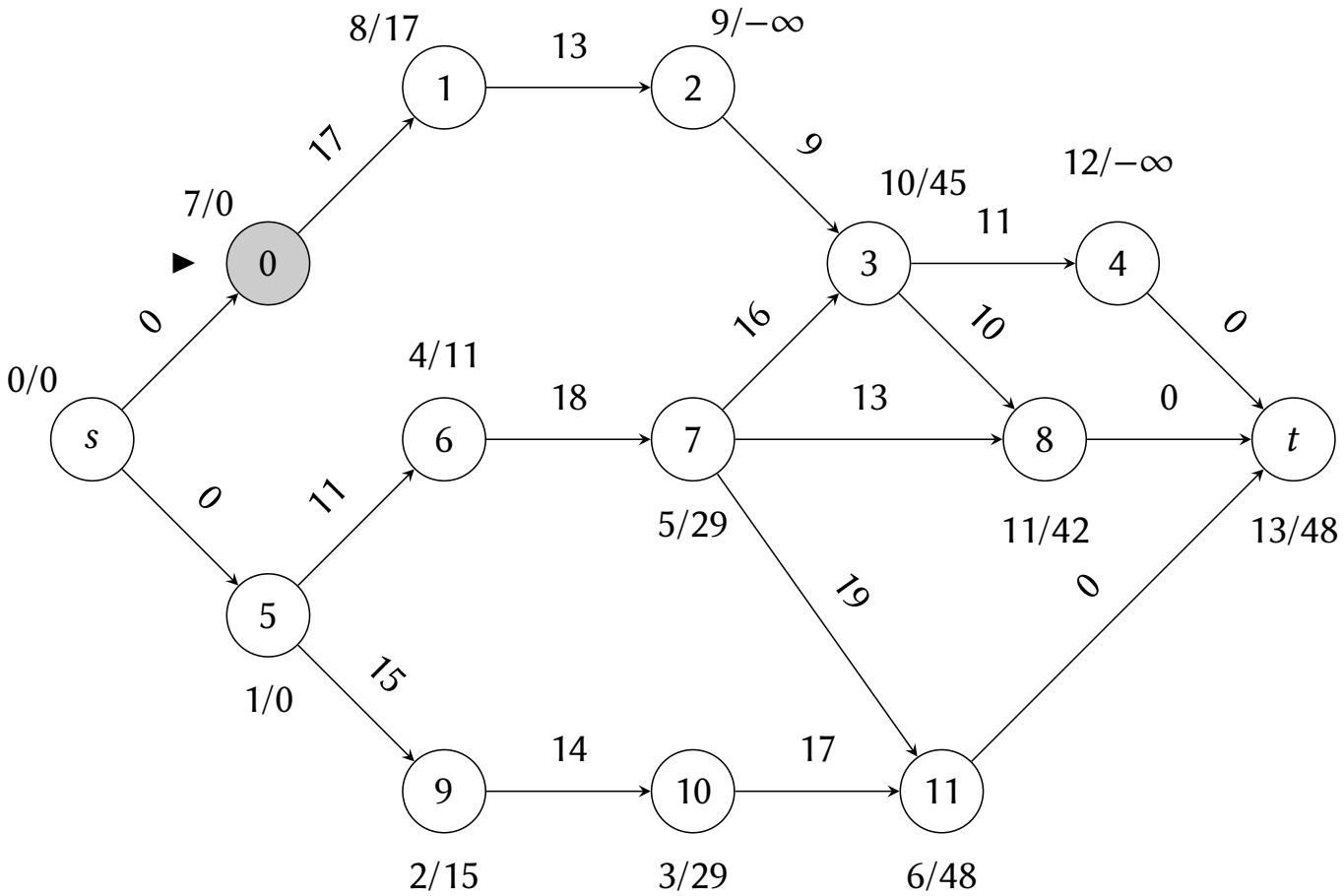
s	0	1	2	3	4	5	6	7	8	9	10	11	t
	s			7		s	5	6	7	5	9	7	

Critical Path Example (8)



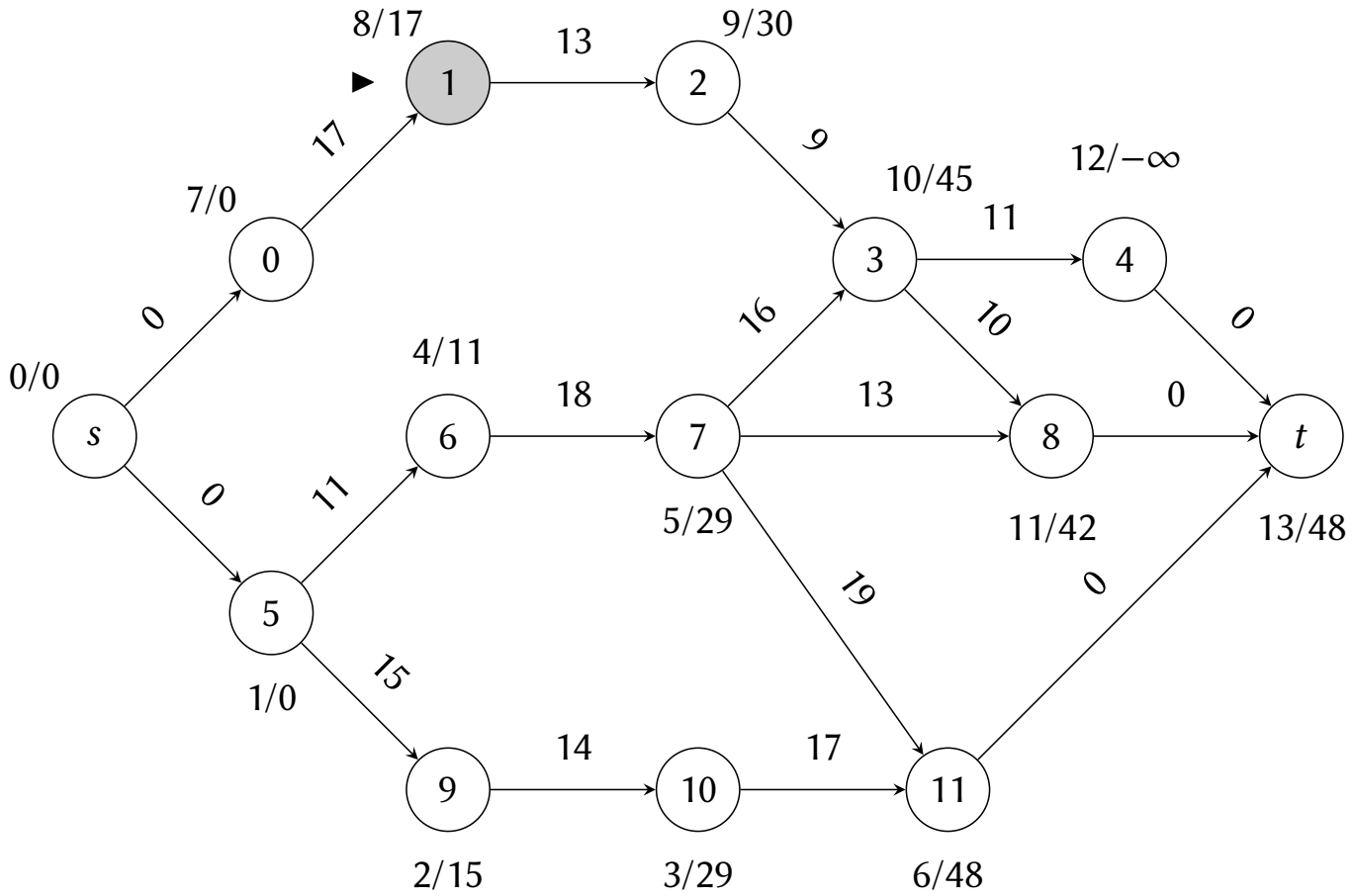
s	0	1	2	3	4	5	6	7	8	9	10	11	t
	s			7		s	5	6	7	5	9	7	11

Critical Path Example (10)



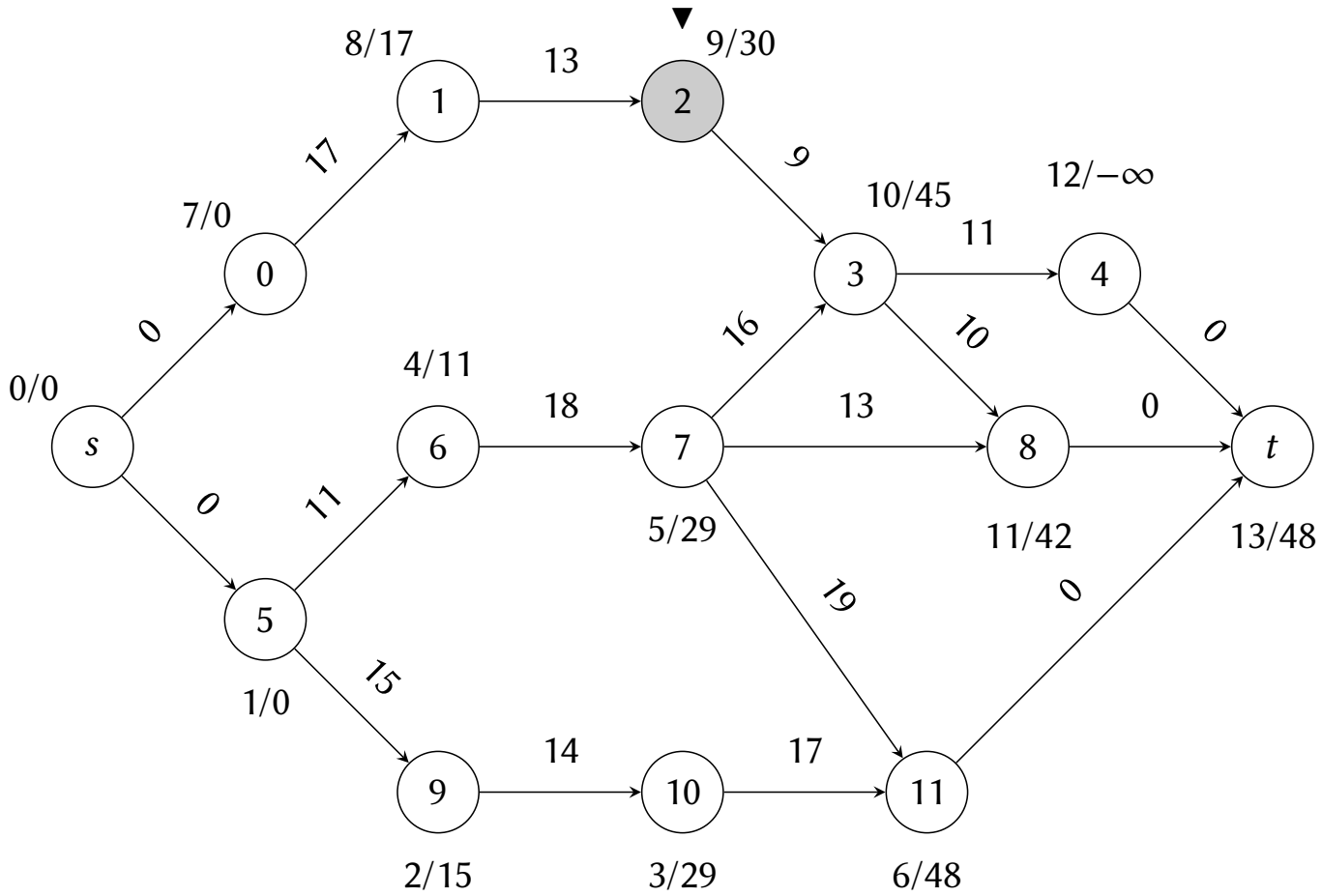
s	0	1	2	3	4	5	6	7	8	9	10	11	t
	s	0		7		s	5	6	7	5	9	7	11

Critical Path Example (11)



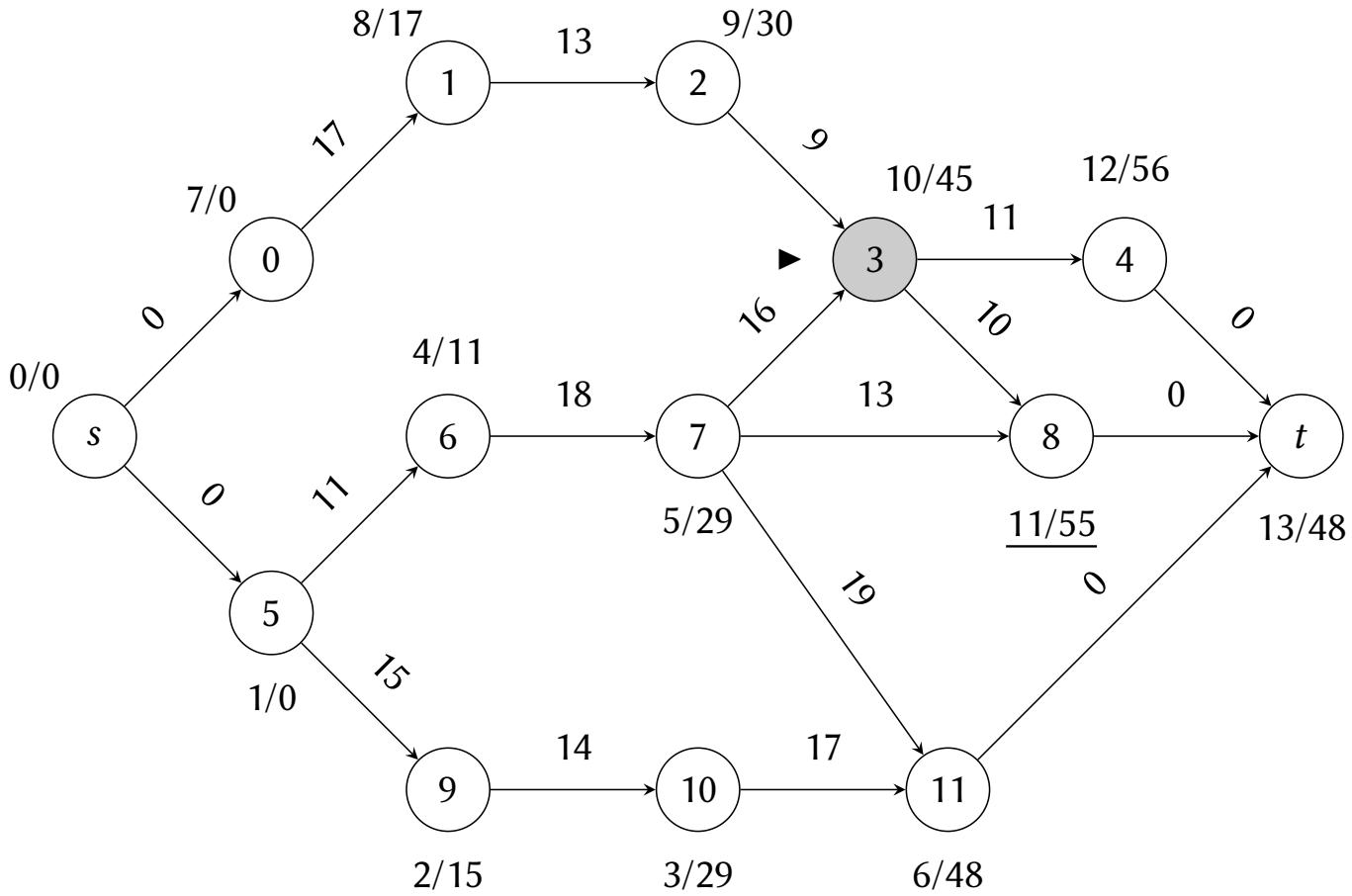
s	0	1	2	3	4	5	6	7	8	9	10	11	t
	s	0	1	7		s	5	6	7	5	9	7	11

Critical Path Example (12)



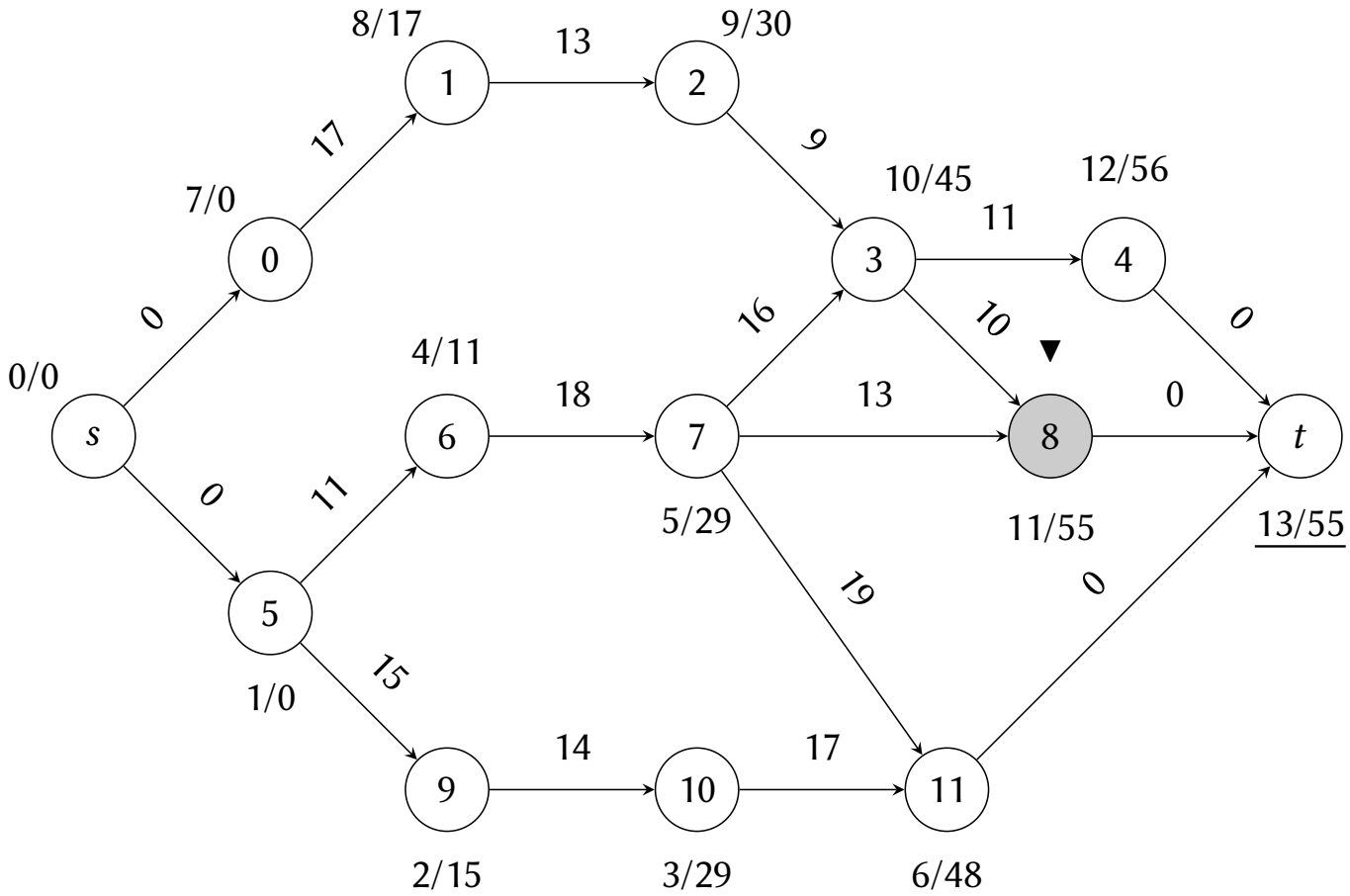
s	0	1	2	3	4	5	6	7	8	9	10	11	t
	s	0	1	7		s	5	6	7	5	9	7	11

Critical Path Example (13)



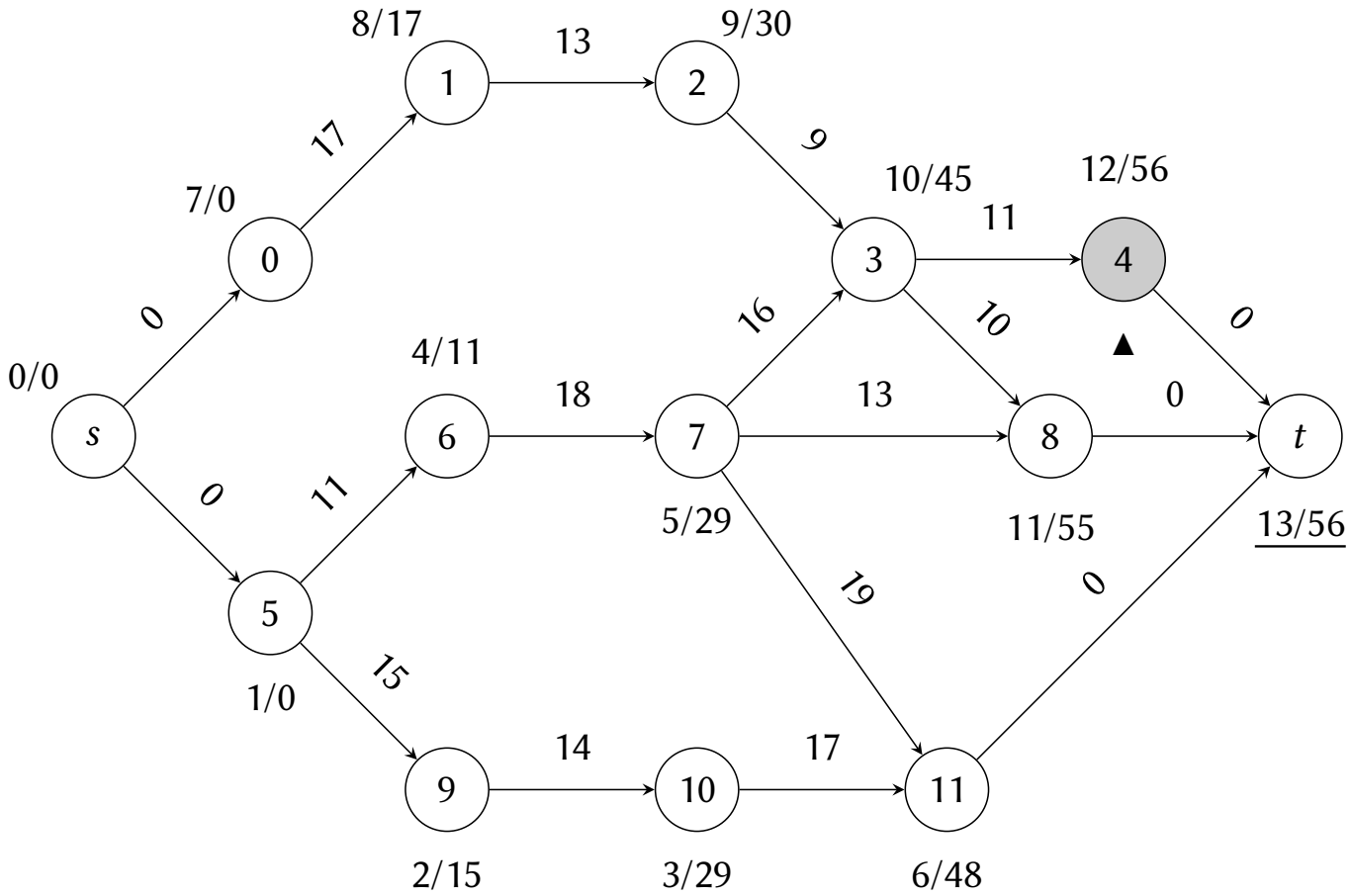
s	0	1	2	3	4	5	6	7	8	9	10	11	t
	s	0	1	7	3	s	5	6	3	5	9	7	11

Critical Path Example (14)



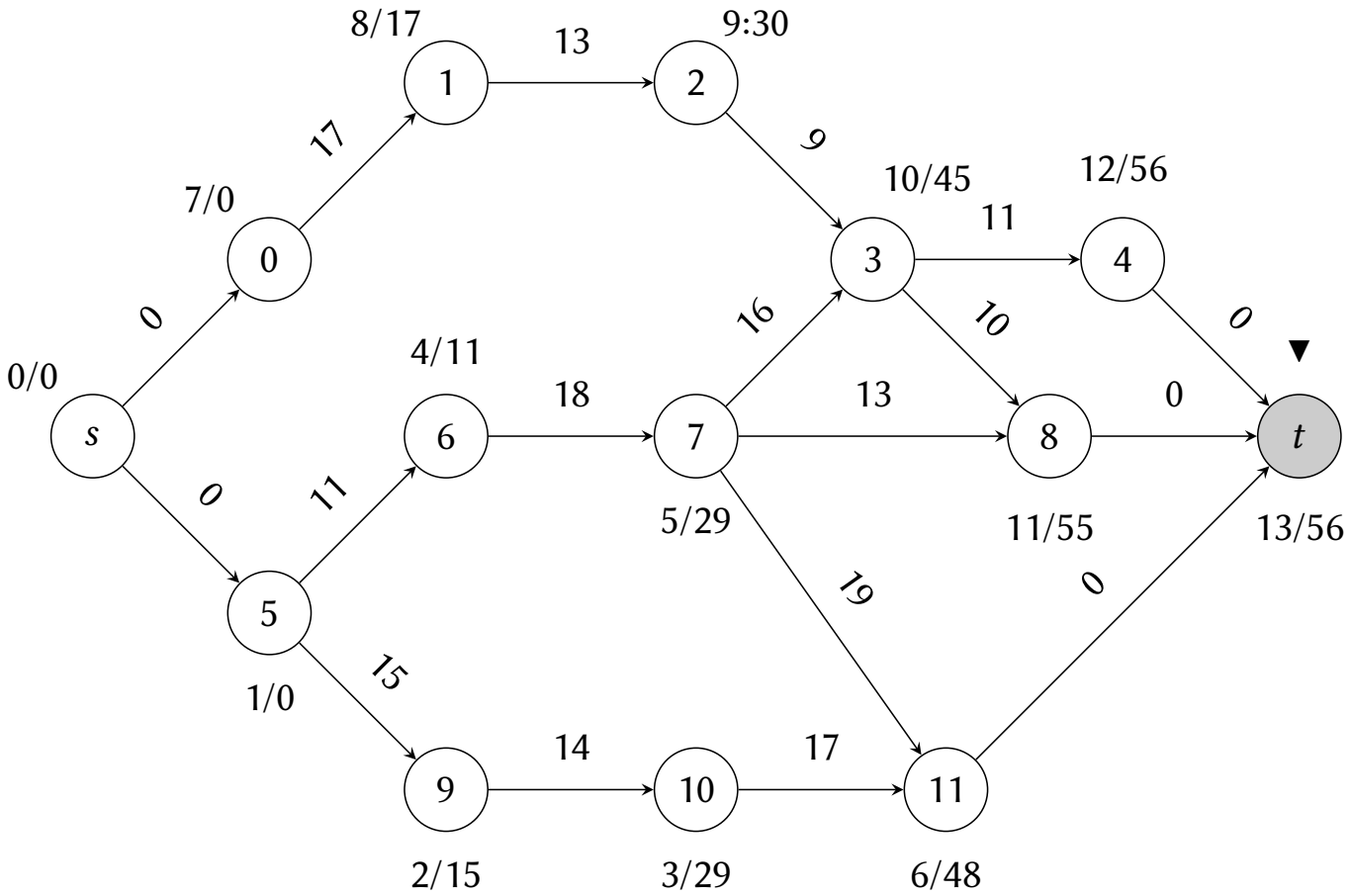
s	0	1	2	3	4	5	6	7	8	9	10	11	t
	s	0	1	7	3	s	5	6	3	5	9	7	8

Critical Path Example (15)



s	0	1	2	3	4	5	6	7	8	9	10	11	t
	s	0	1	7	3	s	5	6	3	5	9	7	4

Critical Path Example (16)



s	0	1	2	3	4	5	6	7	8	9	10	11	t
	s	0	1	7	3	s	5	6	3	5	9	7	4

Critical Path Algorithm

Algorithm: Critical path.

`CriticalPath`$(G) \rightarrow (pred, dist)$

Input: $G = (V, E, s)$, a weighted dag with a source s

Output: $pred$, an array of size $|V|$ such that $pred[i]$ is the predecessor of node i along the critical path from s to i
$dist$, an array of size $|V|$ such that $dist[i]$ contains the length of the critical path from s to i

```
1  pred ← CreateArray(|V|)
2  dist ← CreateArray(|V|)
3  for i ← 0 to |V| do
4      pred[i] ← −1
5      dist[i] ← −∞
6  dist[s] ← 0
7  sorted ← TopologicalSort(V, E)
8  foreach u in sorted do
9      foreach v in AdjacencyList(G, u) do
10         if dist[v] < dist[u] + Weight(G, u, v) then
11             dist[v] ← dist[u] + Weight(G, u, v)
12             pred[v] ← u
13 return (pred, dist)
```

Complexity Analysis

- The topological sort requires time $\Theta(|V| + |E|)$.
- The loop of lines 3–5 requires time $\Theta(|V|)$.
- The loop of lines 8–12 go through lines 11–12 once per edge; each edge is relaxed once. So, each loop requires time $\Theta(|E|)$.
- In total, the algorithm requires time:

$$\Theta(|V| + |E| + |V| + |E|) = \Theta(|V| + |E|)$$