

# Shortest path algorithms

Data Structures and Algorithms for Computational Linguistics III  
(ISCL-BA-07)

Çağrı Çöltekin

`ccoltekin@sfs.uni-tuebingen.de`

University of Tübingen  
Seminar für Sprachwissenschaft

Winter Semester 2020/21

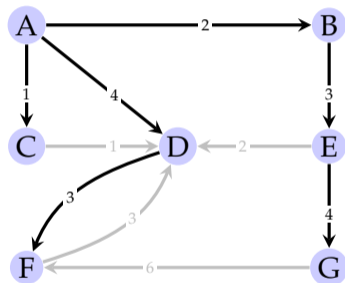
# Shortest path

- Finding shortest paths on a weighted (directed) graph is one of the most common problems in many fields
- Applications include
  - Navigation
  - Routing in computer networks
  - Optimal construction of electronic circuits, VLSI chips
  - Robotics, transportation, finance, ...

# Shortest paths on unweighted graphs

## BFS

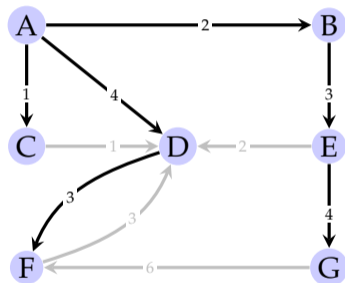
- A BFS search tree gives the shortest path from the source node to all other nodes



# Shortest paths on unweighted graphs

## BFS

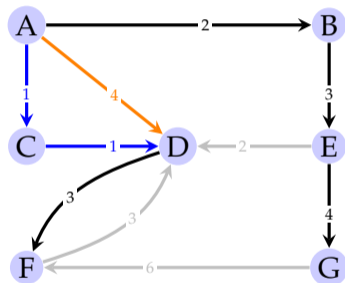
- A BFS search tree gives the shortest path from the source node to all other nodes
- The BFS is not enough on weighted graphs



# Shortest paths on unweighted graphs

## BFS

- A BFS search tree gives the shortest path from the source node to all other nodes
- The BFS is not enough on weighted graphs
- Shortest-cost path may be longer in terms of nodes visited



# Shortest paths on weighted graphs

## variation of the problem

- Different versions of the problem:
  - Single source shortest path: find shortest path from a source node to all others
  - Single target (sometimes called sink) shortest path: find shortest path from all nodes to a target node
  - Source to target: from a particular source node to a particular target node
  - All pairs: shortest paths between all pairs of nodes
- Restrictions on weights:
  - Euclidean weights
  - Non-negative weights
  - Arbitrary weights

# Dijkstra's algorithm

## intro

- Dijkstra's algorithm is a 'weighted' version of the BFS
- The algorithm finds shortest path from a single source node to all connected nodes
- Weights has to be non-negative
- It is a greedy algorithm that grows a 'cloud' of nodes for which we know the shortest paths from the source node
- The new nodes are included in the cloud in order of their shortest paths from the source node
- The algorithm is also similar to Prim-Jarník algorithm used for finding MST

# Dijkstra's algorithm

## the algorithm

- We maintain a list  $D$  of minimum known distances to each node
- At each step
  - we take closest node out of  $Q$
  - update the distances of all nodes
- Can be more efficient if  $Q$  is implemented using a (adaptable) priority queue

```

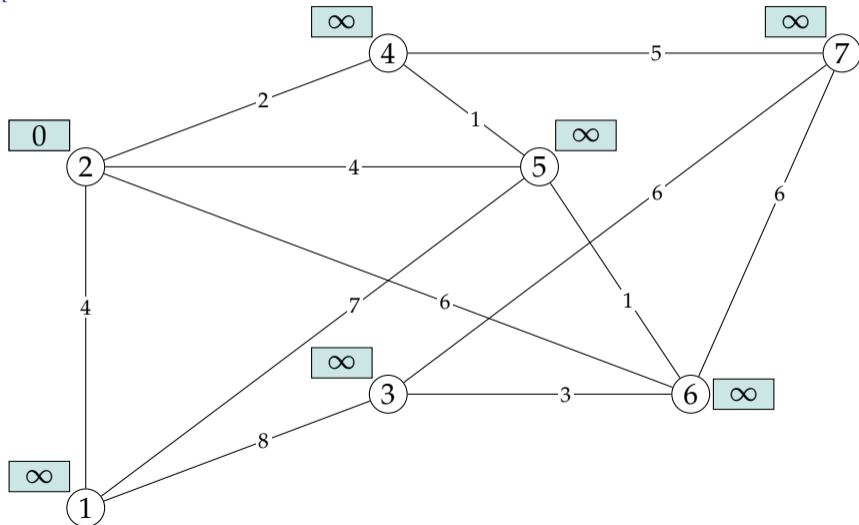
1:  $D[s] \leftarrow 0$ 
2: for each node  $v \neq s$  do
3:    $D[v] \leftarrow \infty$ 
4:  $Q \leftarrow \text{nodes}$ 
5: while  $Q$  is not empty do
6:   Find the node  $v$  with  $\min D[v]$ 
7:   for each edge  $(v, w)$  do
8:     if  $D[v] + w[(v, w)] < D[w]$  then
9:        $D[w] \leftarrow D[v] + w[(v, w)]$ 
10:  $D$  contains the shortest distances from  $s$ 

```



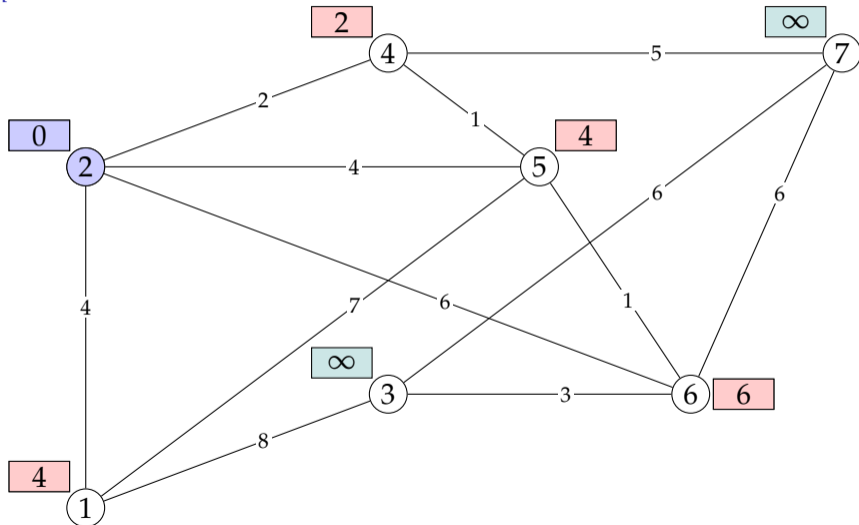
# Dijkstra's algorithm

## demonstration



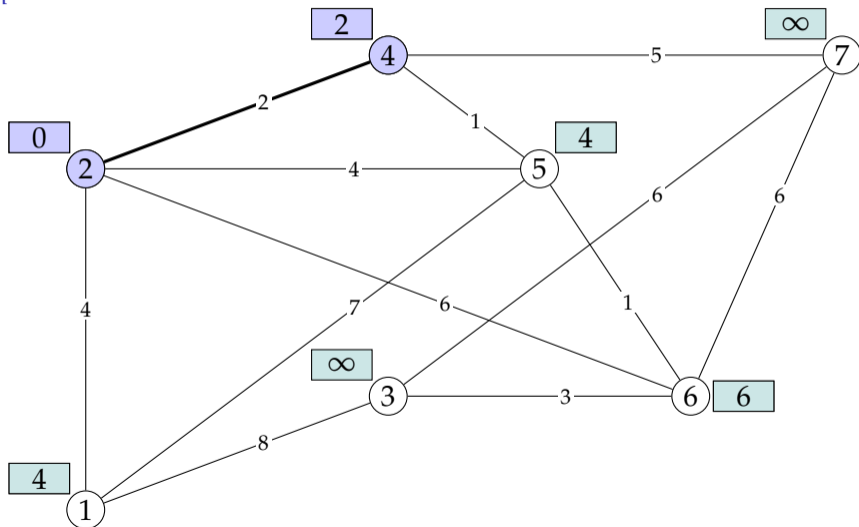
# Dijkstra's algorithm

## demonstration



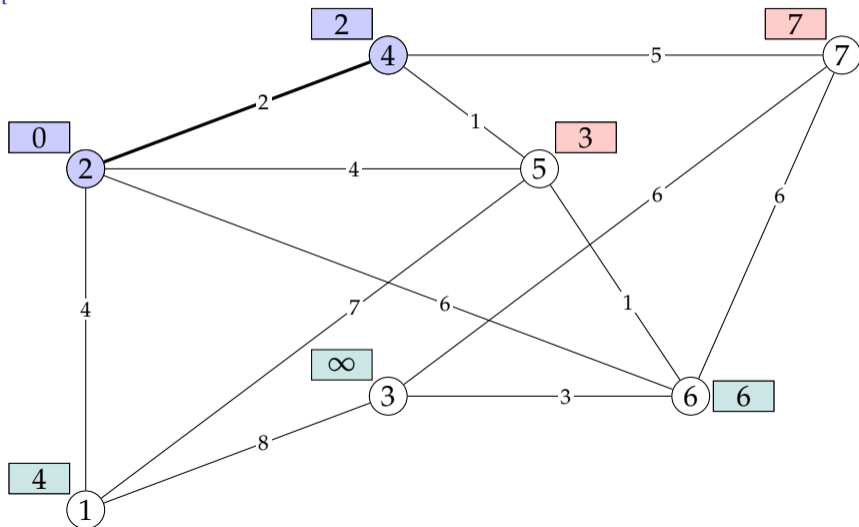
# Dijkstra's algorithm

## demonstration



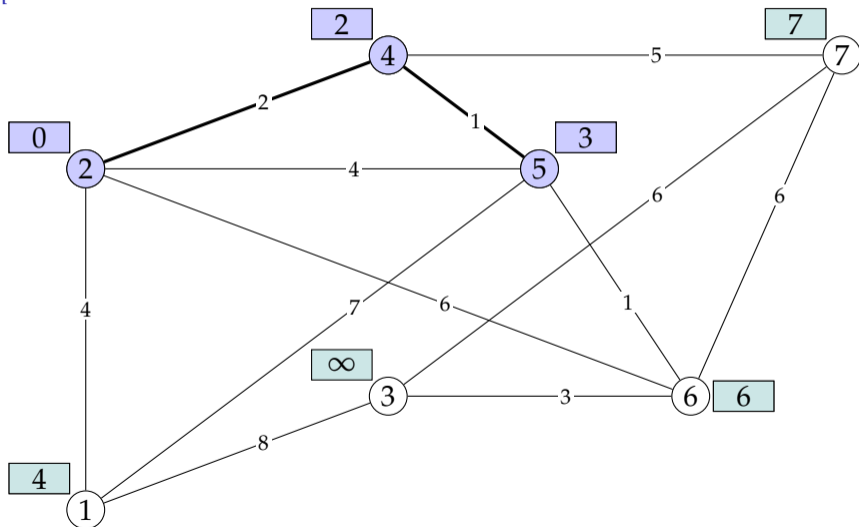
# Dijkstra's algorithm

## demonstration



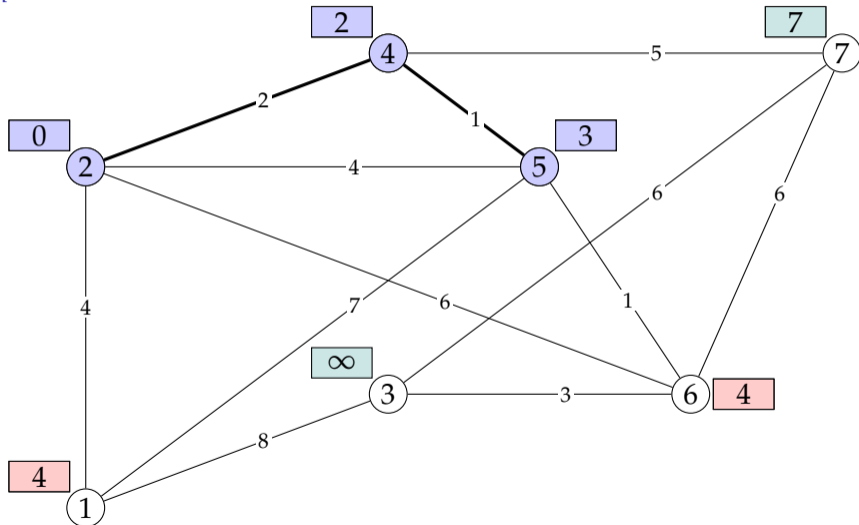
# Dijkstra's algorithm

demonstration



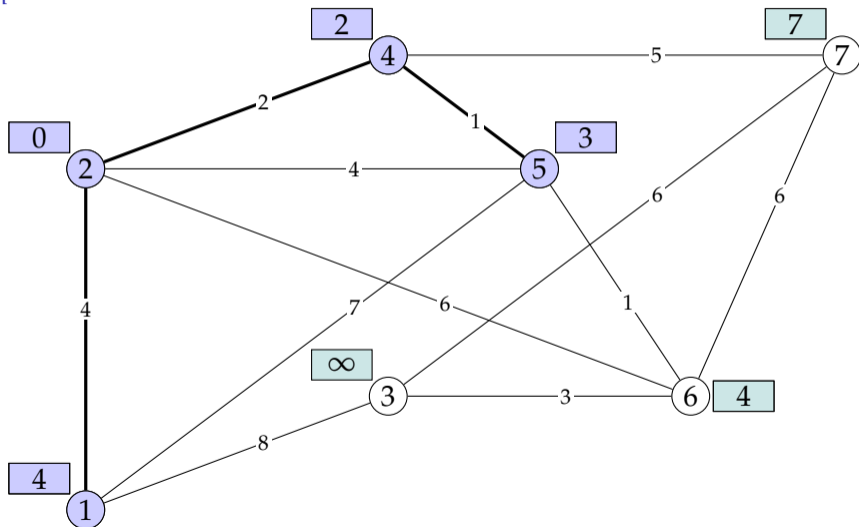
# Dijkstra's algorithm

## demonstration



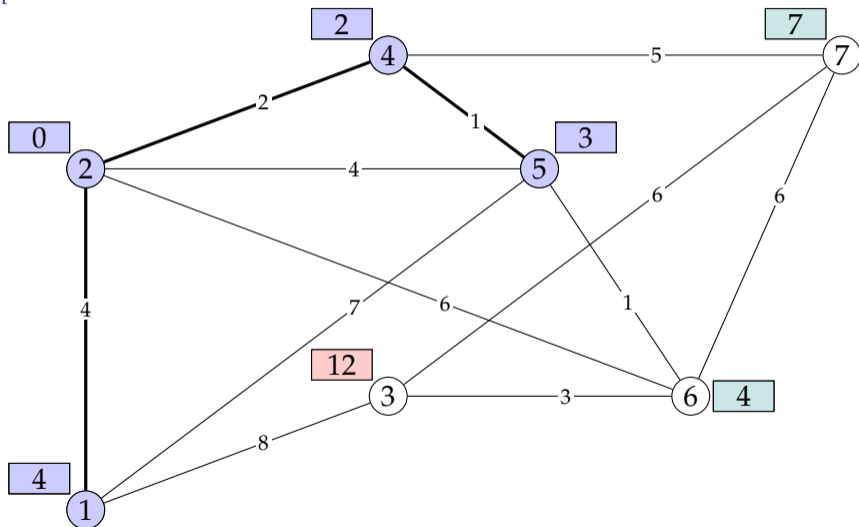
# Dijkstra's algorithm

## demonstration



# Dijkstra's algorithm

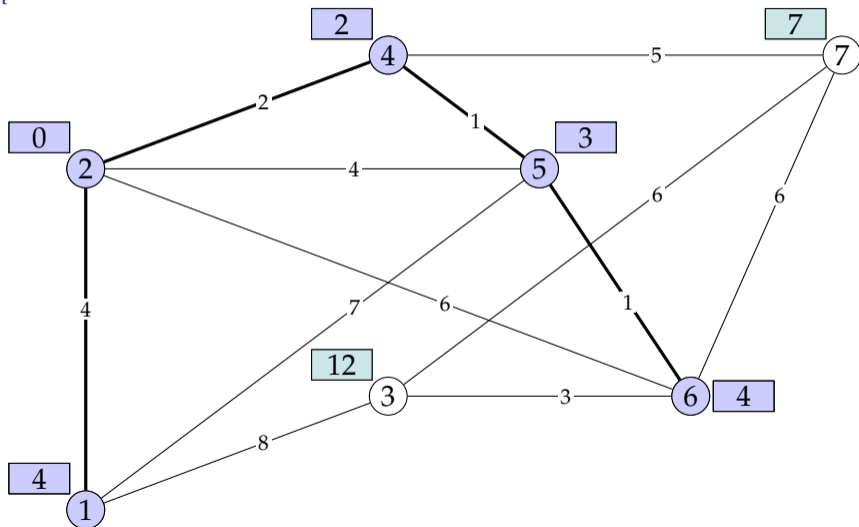
demonstration





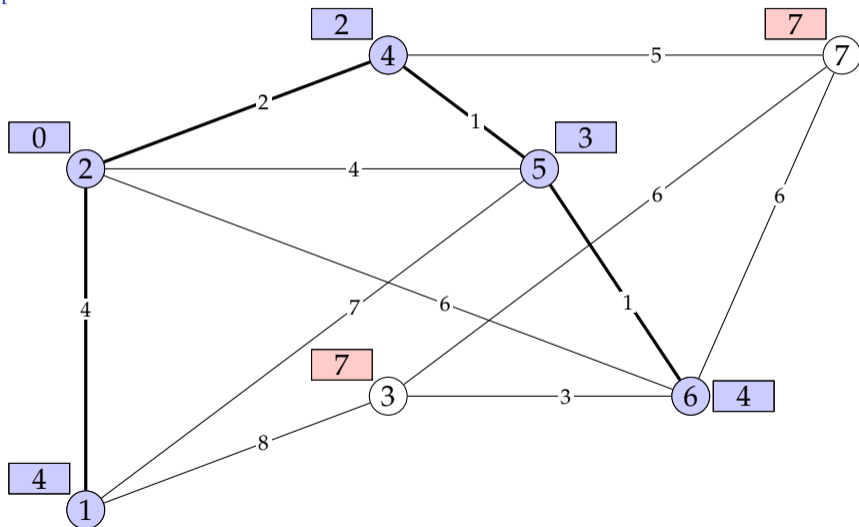
# Dijkstra's algorithm

demonstration



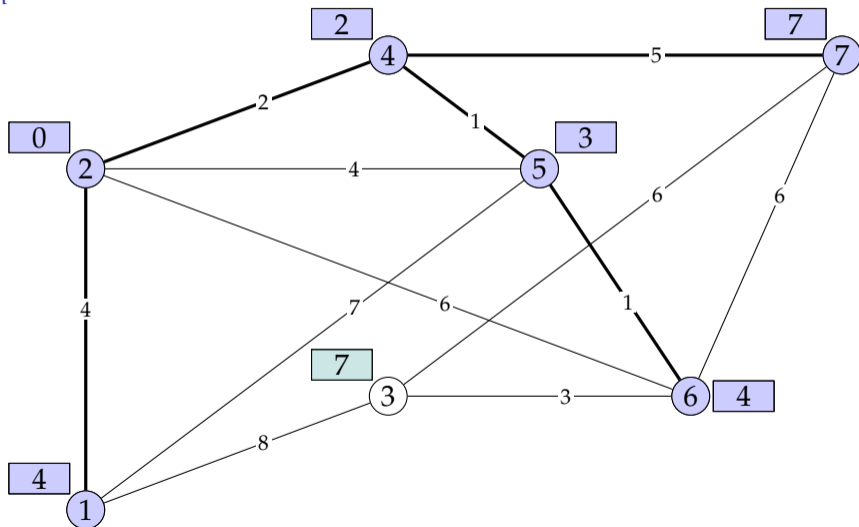
# Dijkstra's algorithm

demonstration



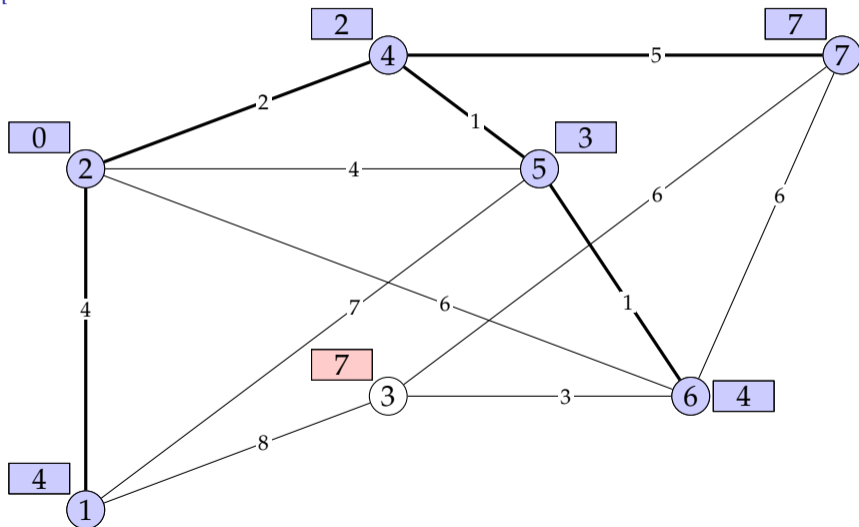
# Dijkstra's algorithm

demonstration



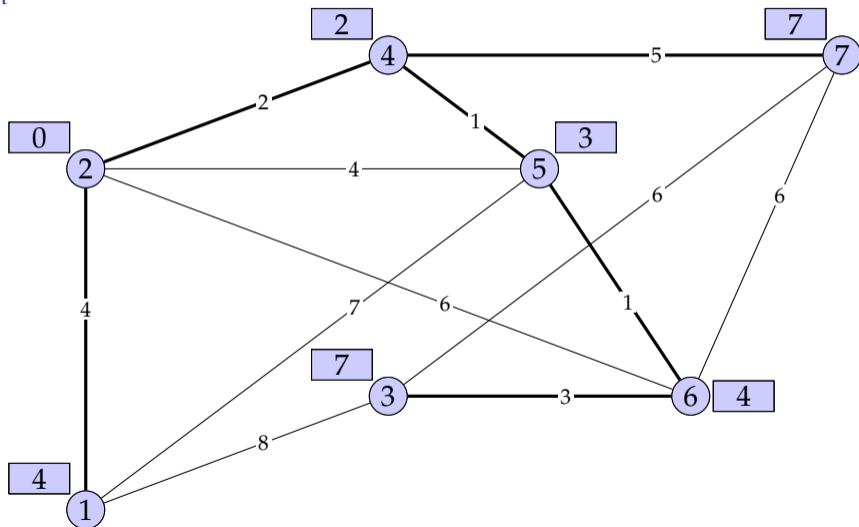
# Dijkstra's algorithm

demonstration

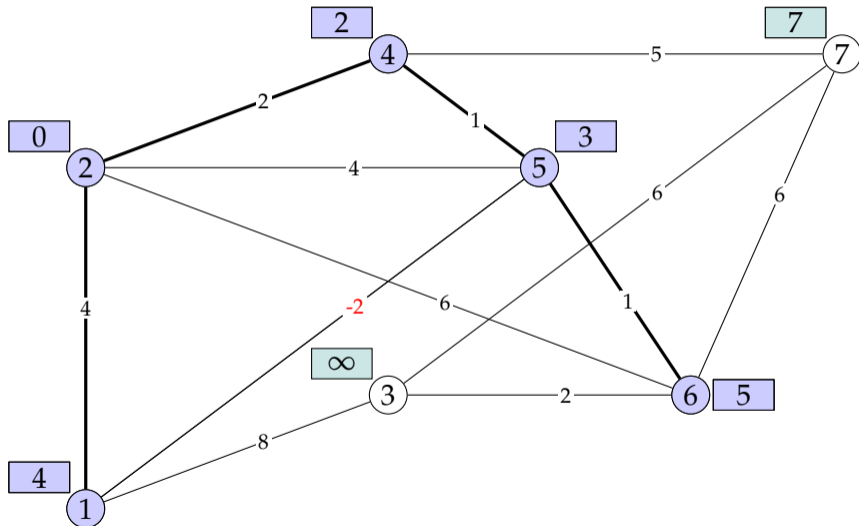


# Dijkstra's algorithm

demonstration



# Dijkstra's algorithm and negative weights



# Dijkstra's algorithm

## the algorithm

- In general, complexity is  $O(t_{\text{find\_min}}n + t_{\text{update\_key}}m)$
- With list-based implementation of Q:  $O(m + n^2) = O(n^2)$
- With a priority queue:  $O((m + n) \log n)$

```

1: D[s] ← 0
2: for each node v ≠ s do
3:   D[v] ← ∞
4: Q ← nodes
5: while Q is not empty do
6:   Find the node v with min D[v]
7:   for each edge (v, w) do
8:     if D[v] + w[(v, w)] < D[w] then
9:       D[w] ← D[v] + w[(v, w)]
10: D contains the shortest distances from s

```

# Shortest-path tree

- The way we introduced, the Dijkstra's algorithm does not give the shortest-path tree
- Similar to traversal algorithms, we can extract it from distances  $D$
- Running time is  $O(n^2)$  (or  $O(n + m)$ )

```

1:  $T \leftarrow \emptyset$ 
2: for  $v \in D - \{s\}$  do
3:   for each edge  $(w, v)$  do
4:     if  $D[v] == D[u] + \text{weight}(w, v)$  then
5:        $T \leftarrow T \cup (w, v)$ 

```

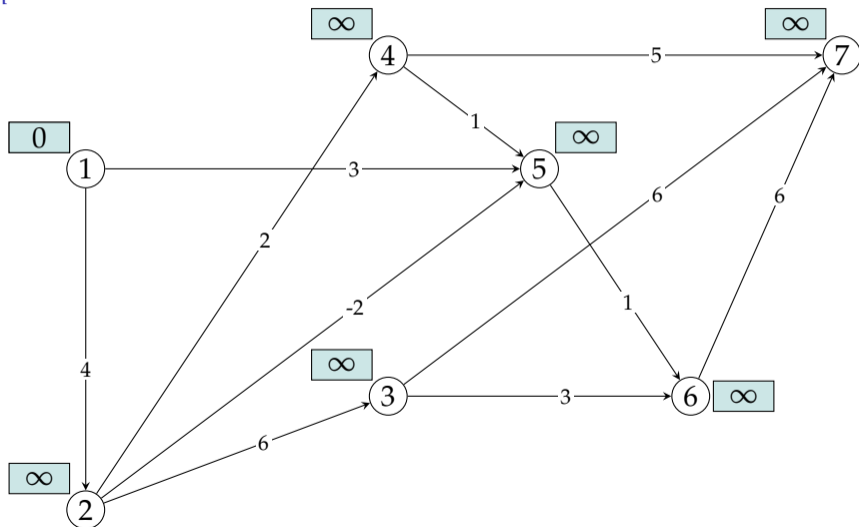


# Shortest-paths on DAGs

- The shortest path can be found more efficiently, if the graph is a DAG
- The algorithm is similar to Dijkstra's, but simpler and faster
- Only difference is we follow a topological order
- The algorithm will also work with negative edge weights

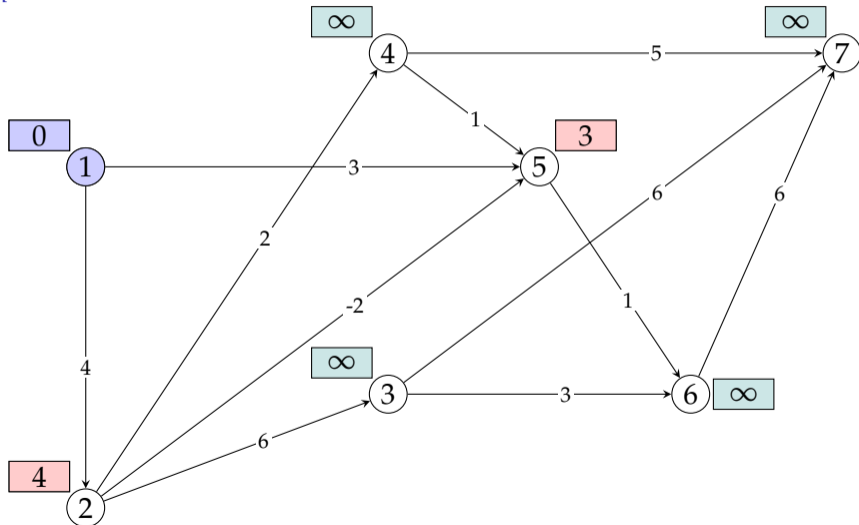
# Shortest-paths on DAGs

demonstration



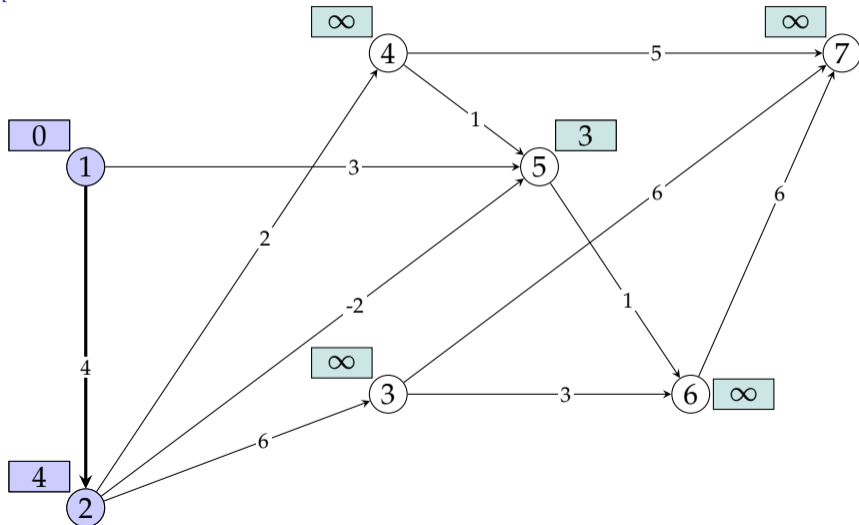
# Shortest-paths on DAGs

demonstration



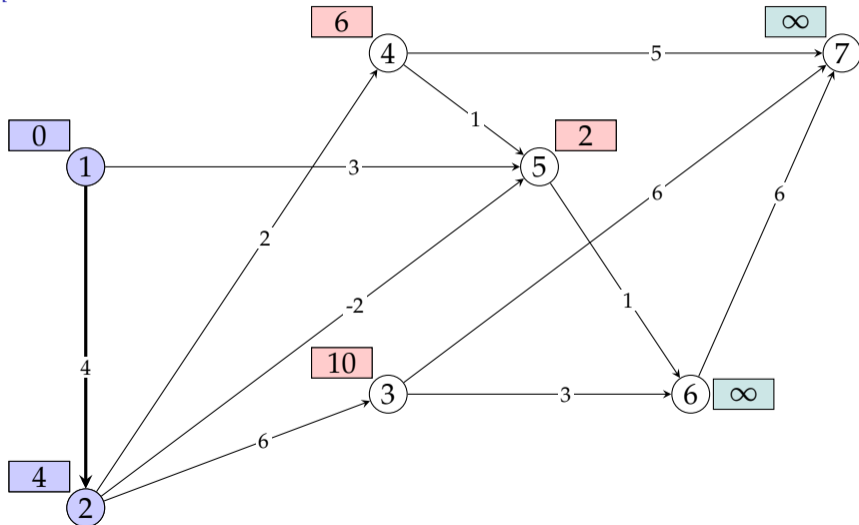
# Shortest-paths on DAGs

demonstration



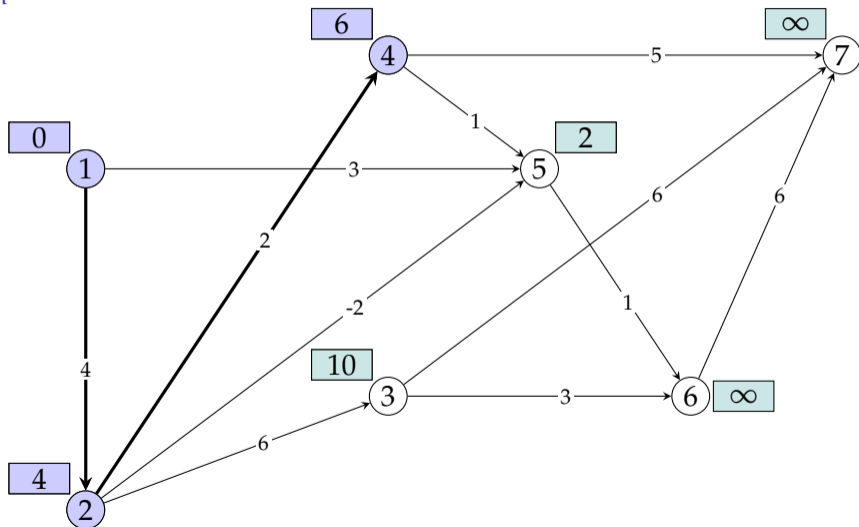
# Shortest-paths on DAGs

demonstration



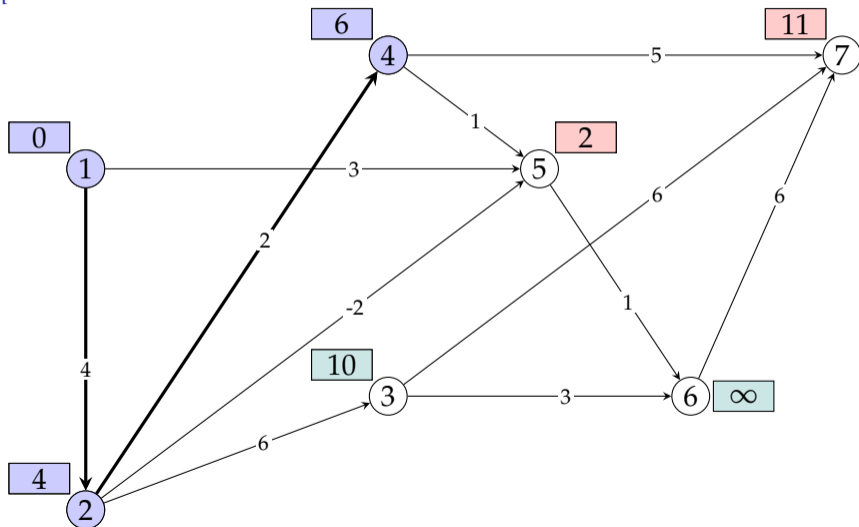
# Shortest-paths on DAGs

demonstration



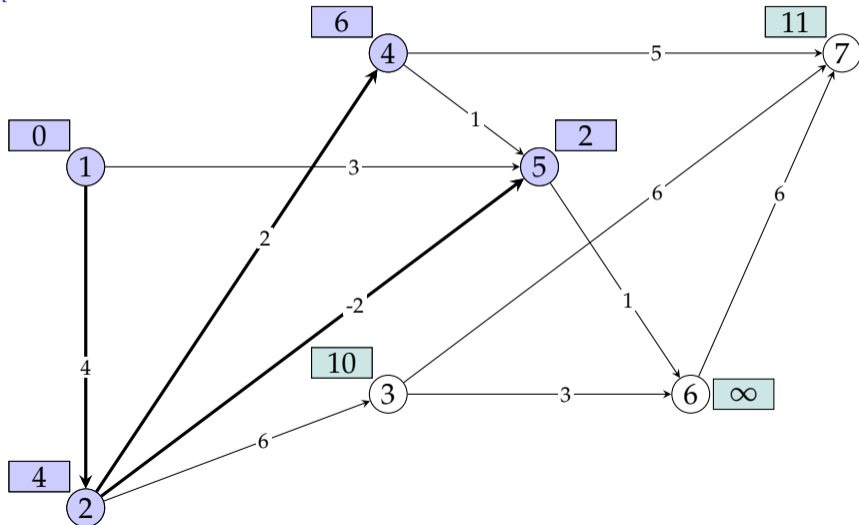
# Shortest-paths on DAGs

demonstration



# Shortest-paths on DAGs

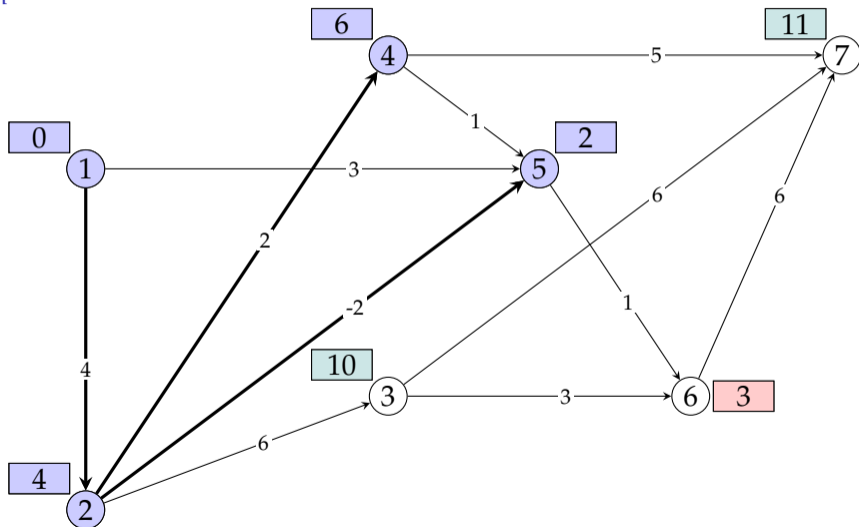
demonstration





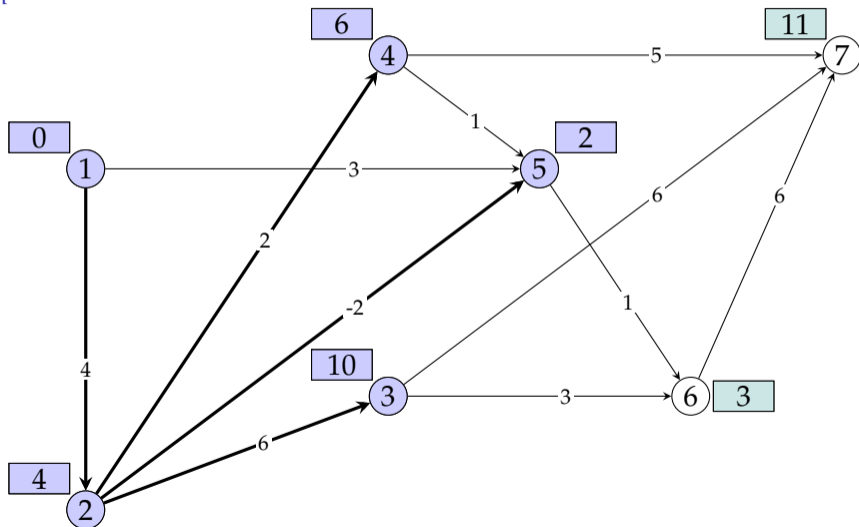
# Shortest-paths on DAGs

demonstration



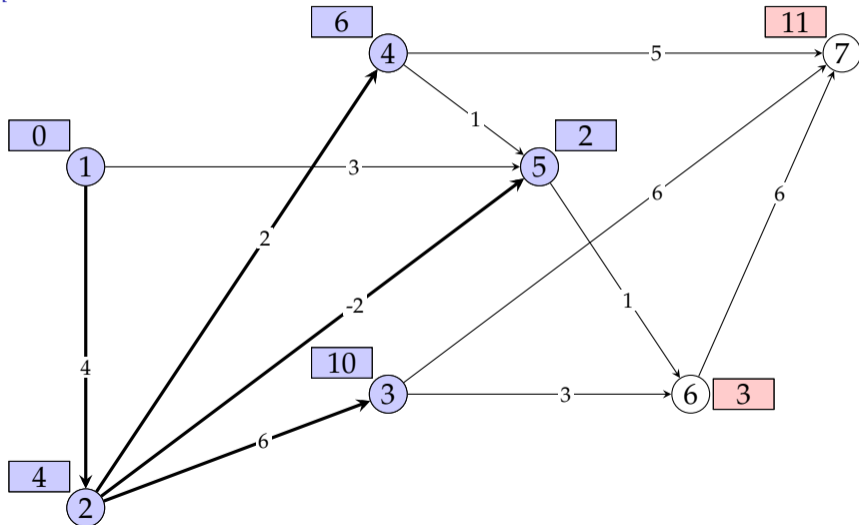
# Shortest-paths on DAGs

demonstration



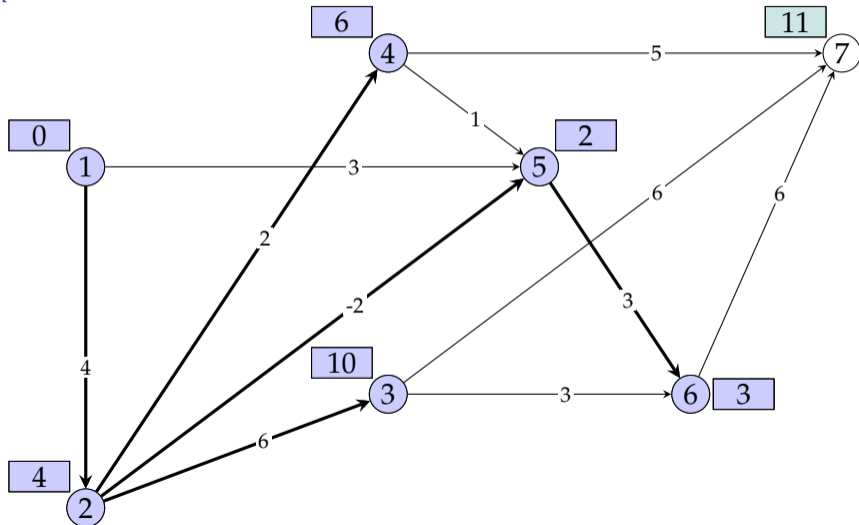
# Shortest-paths on DAGs

demonstration



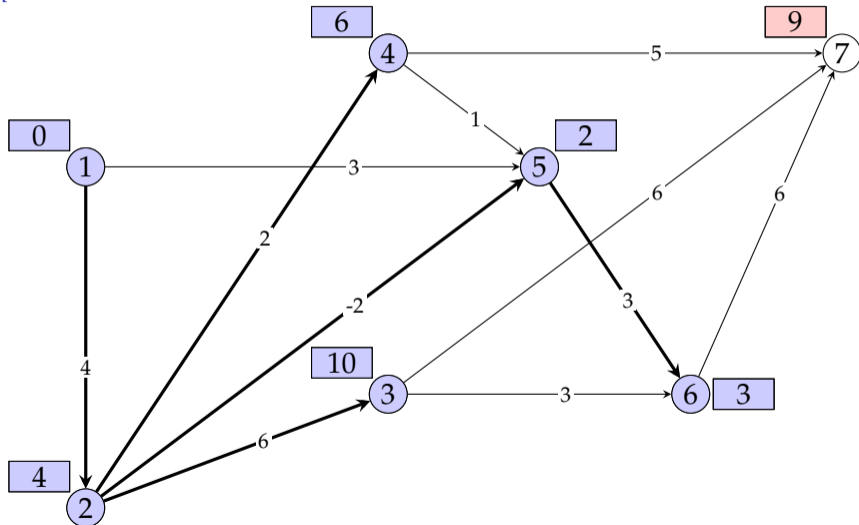
# Shortest-paths on DAGs

demonstration



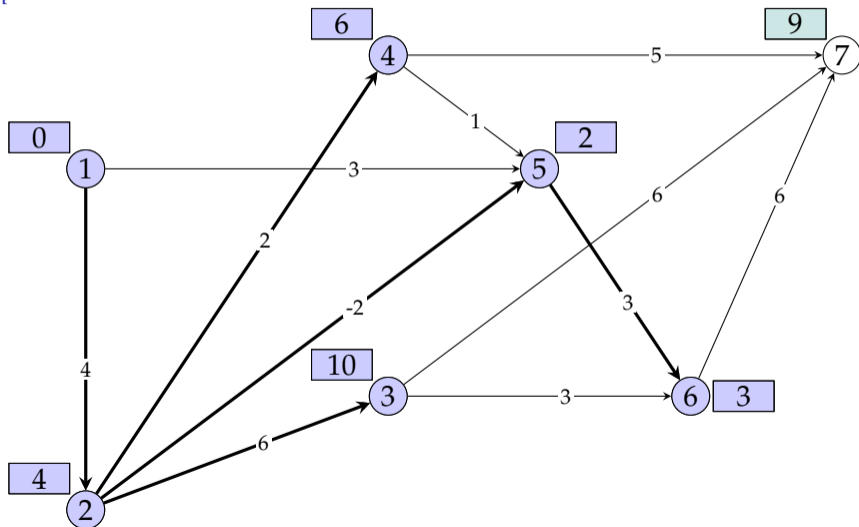
# Shortest-paths on DAGs

demonstration



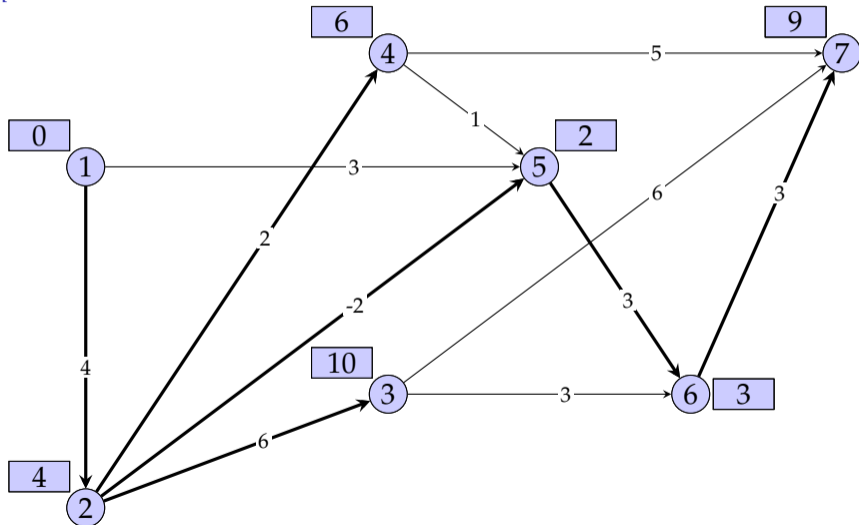
# Shortest-paths on DAGs

demonstration



# Shortest-paths on DAGs

demonstration



# Shortest-paths on directed graphs

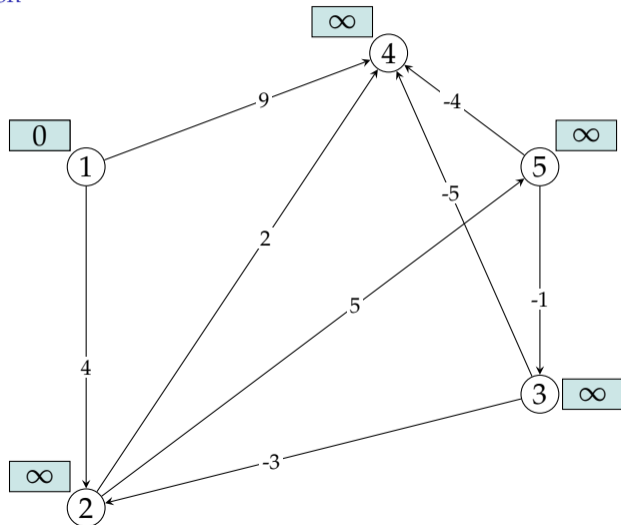
with negative weights – without negative cycles

- Single-source shortest path problem can also be solved efficiently for any directed graph
  - including cycles (no DAG requirement)
  - including negative weights
  - *excluding* negative cycles
- The algorithm is known as Bellman-Ford algorithm
  - Similar to earlier, initialize  $D[s] = 0$ ,  $D[v] = \infty$
  - Make  $n$  passes over the edges
    - Update distances for each edge (relax edges)
    - Stop if there were no changes at the end of a pass



# Bellman-Ford algorithm

## demonstration

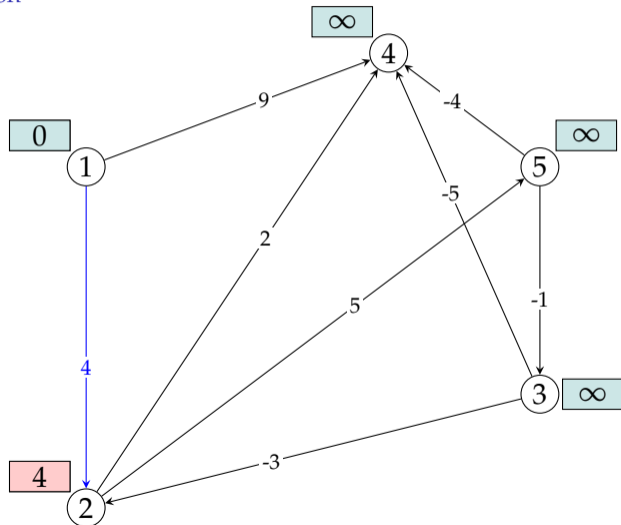


*edges*

1 → 2
1 → 4
2 → 4
2 → 5
3 → 2
3 → 4
5 → 3
5 → 4

# Bellman-Ford algorithm

## demonstration

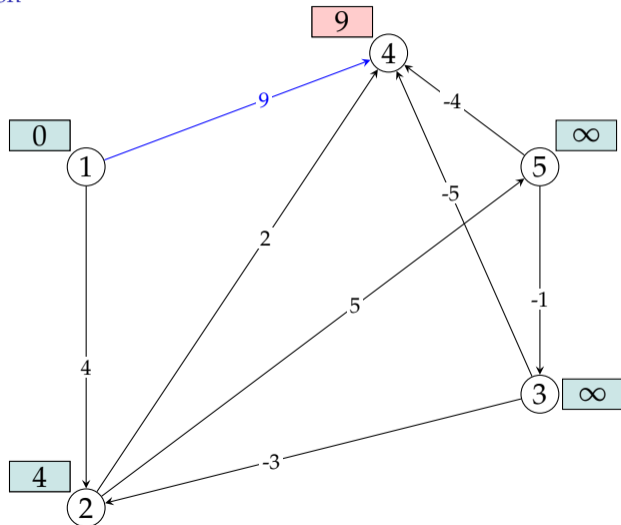


*edges*

1 → 2
1 → 4
2 → 4
2 → 5
3 → 2
3 → 4
5 → 3
5 → 4

# Bellman-Ford algorithm

## demonstration

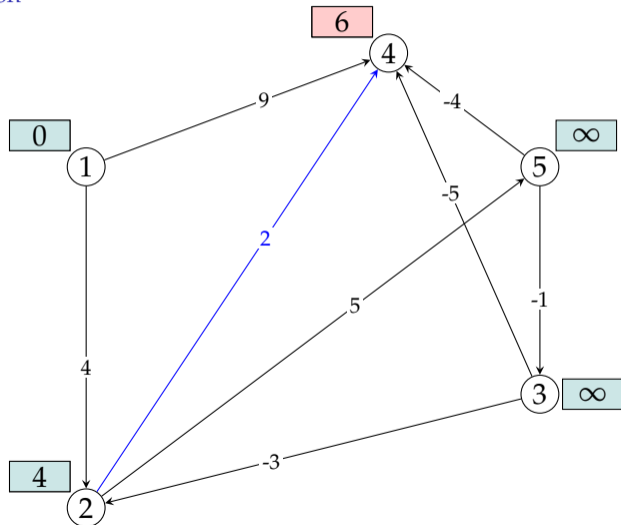


*edges*

1 $\rightarrow$ 2
1 $\rightarrow$ 4
2 $\rightarrow$ 4
2 $\rightarrow$ 5
3 $\rightarrow$ 2
3 $\rightarrow$ 4
5 $\rightarrow$ 3
5 $\rightarrow$ 4

# Bellman-Ford algorithm

## demonstration

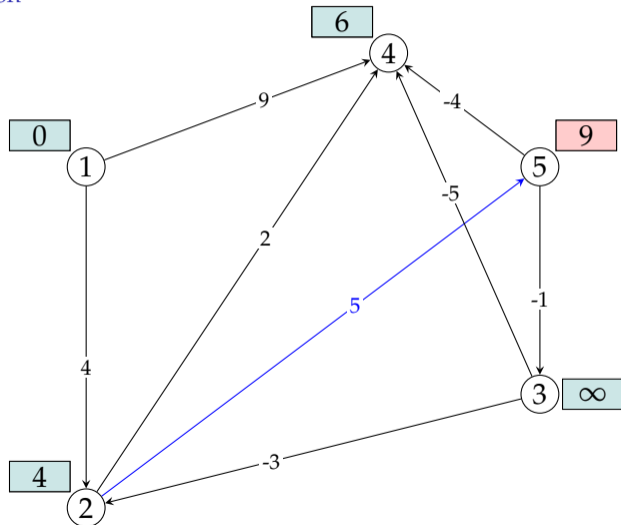


*edges*

1 → 2
1 → 4
2 → 4
2 → 5
3 → 2
3 → 4
5 → 3
5 → 4

# Bellman-Ford algorithm

## demonstration

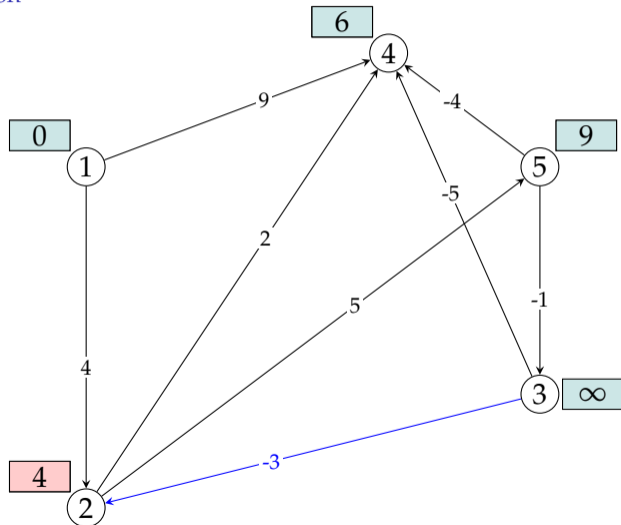


*edges*

1 → 2
1 → 4
2 → 4
2 → 5
3 → 2
3 → 4
5 → 3
5 → 4

# Bellman-Ford algorithm

## demonstration

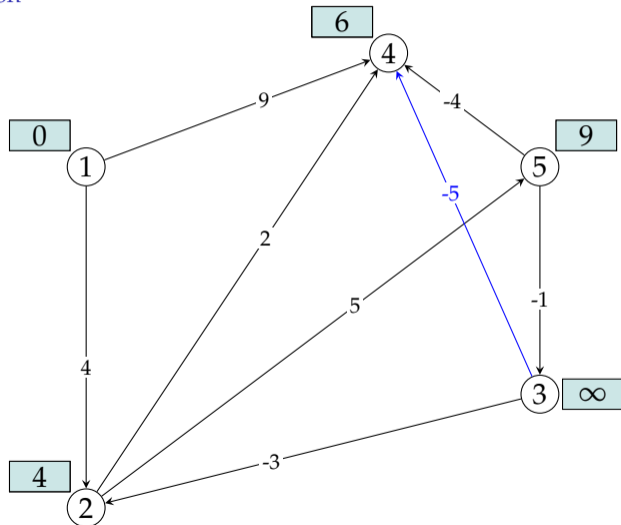


*edges*

1 → 2
1 → 4
2 → 4
2 → 5
3 → 2
3 → 4
5 → 3
5 → 4

# Bellman-Ford algorithm

demonstration

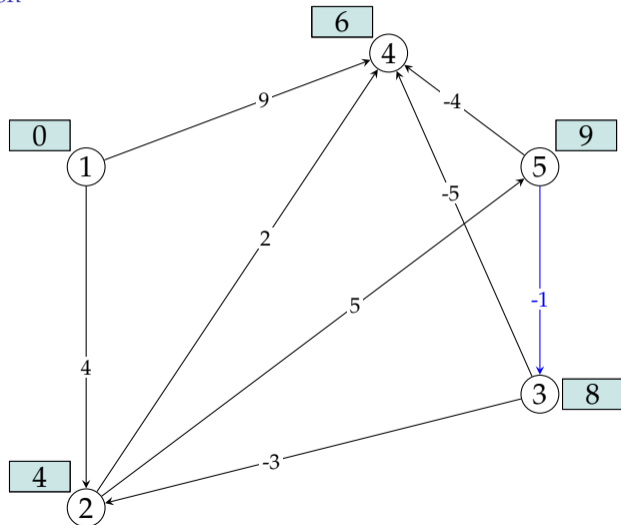


*edges*

1 → 2
1 → 4
2 → 4
2 → 5
3 → 2
3 → 4
5 → 3
5 → 4

# Bellman-Ford algorithm

## demonstration



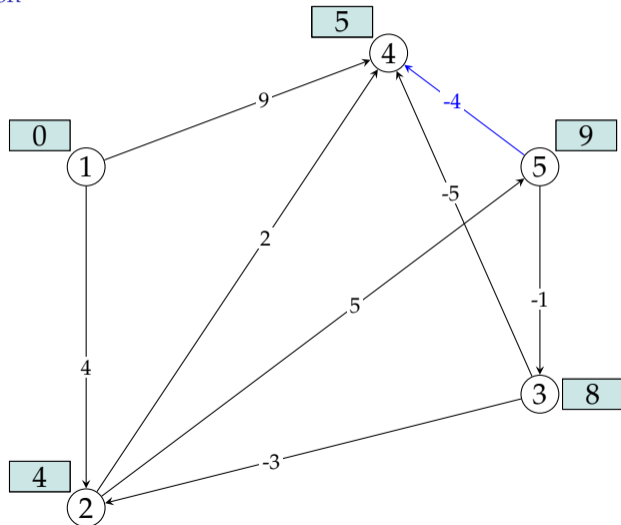
*edges*

1 → 2
1 → 4
2 → 4
2 → 5
3 → 2
3 → 4
5 → 3
5 → 4



# Bellman-Ford algorithm

## demonstration

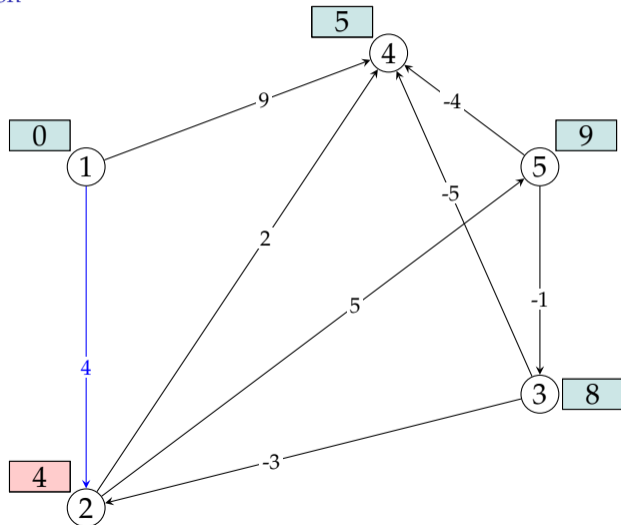


*edges*

1 → 2
1 → 4
2 → 4
2 → 5
3 → 2
3 → 4
5 → 3
5 → 4

# Bellman-Ford algorithm

## demonstration

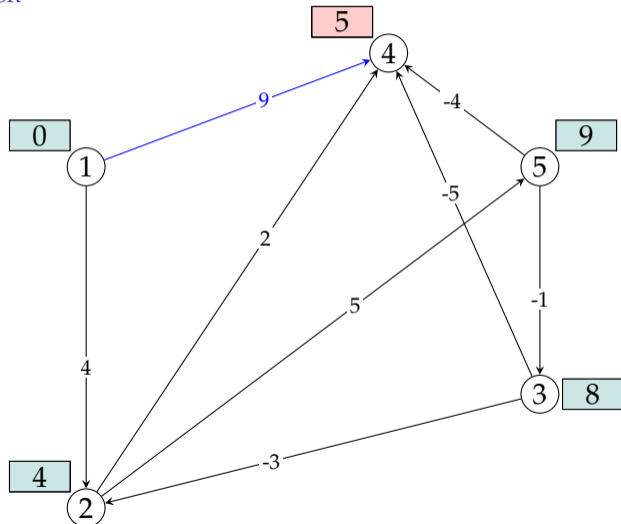


*edges*

1 → 2
1 → 4
2 → 4
2 → 5
3 → 2
3 → 4
5 → 3
5 → 4

# Bellman-Ford algorithm

demonstration

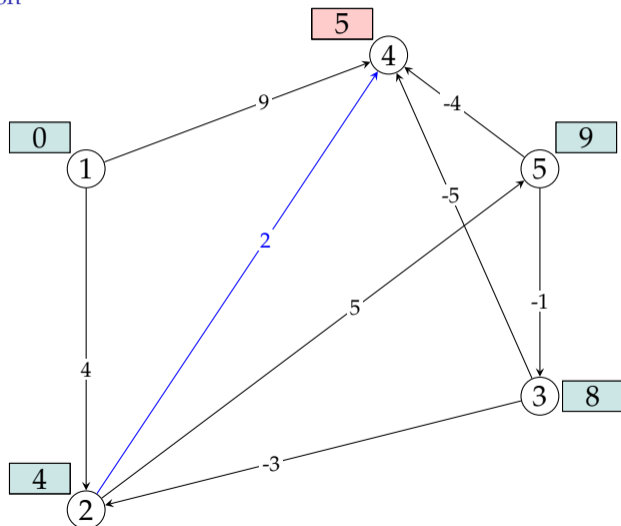


*edges*

1 → 2
1 → 4
2 → 4
2 → 5
3 → 2
3 → 4
5 → 3
5 → 4

# Bellman-Ford algorithm

## demonstration

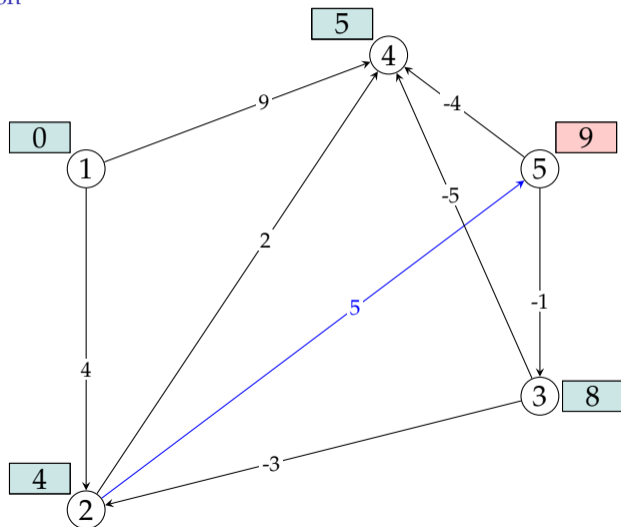


*edges*

1 → 2
1 → 4
2 → 4
2 → 5
3 → 2
3 → 4
5 → 3
5 → 4

# Bellman-Ford algorithm

## demonstration

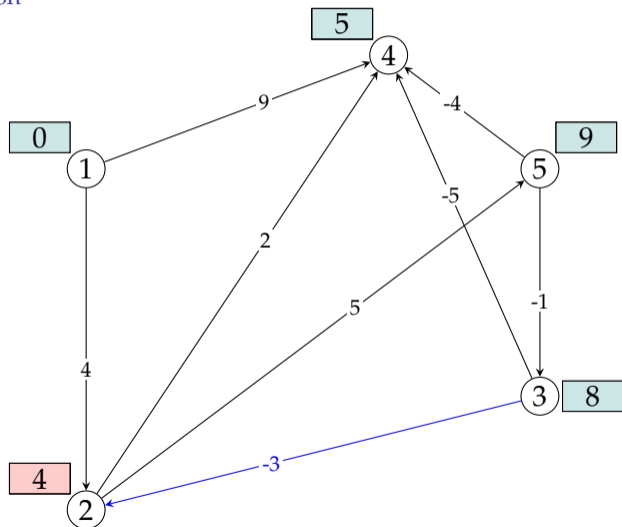


*edges*

1 → 2
1 → 4
2 → 4
2 → 5
3 → 2
3 → 4
5 → 3
5 → 4

# Bellman-Ford algorithm

## demonstration

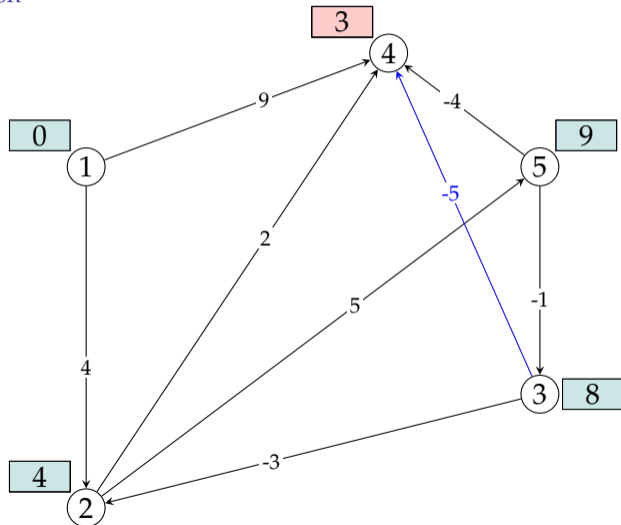


*edges*

1 → 2
1 → 4
2 → 4
2 → 5
3 → 2
3 → 4
5 → 3
5 → 4

# Bellman-Ford algorithm

## demonstration

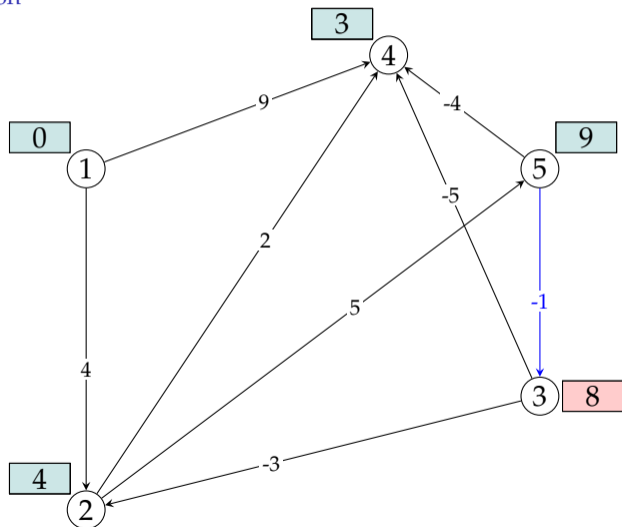


*edges*

1 → 2
1 → 4
2 → 4
2 → 5
3 → 2
3 → 4
5 → 3
5 → 4

# Bellman-Ford algorithm

## demonstration



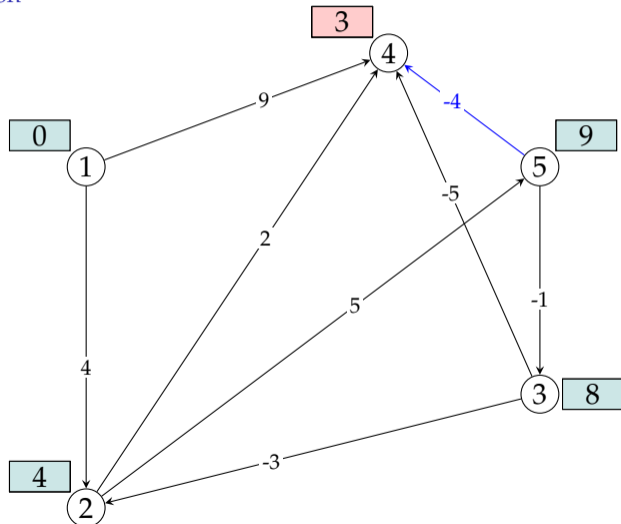
*edges*

1 → 2
1 → 4
2 → 4
2 → 5
3 → 2
3 → 4
5 → 3
5 → 4



# Bellman-Ford algorithm

demonstration

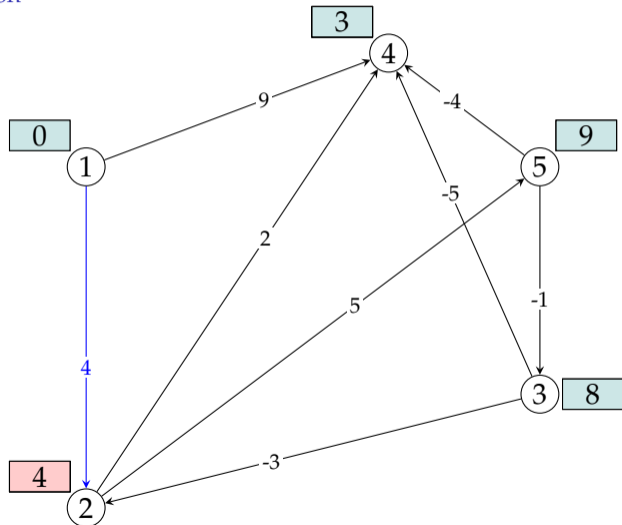


*edges*

1 → 2
1 → 4
2 → 4
2 → 5
3 → 2
3 → 4
5 → 3
5 → 4

# Bellman-Ford algorithm

## demonstration

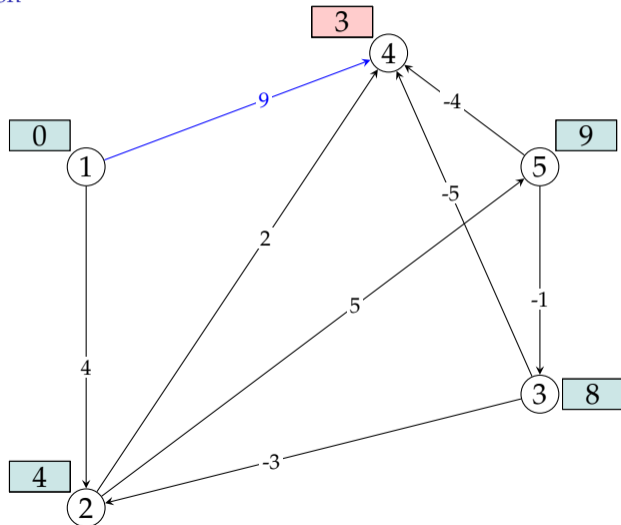


*edges*

1 → 2
1 → 4
2 → 4
2 → 5
3 → 2
3 → 4
5 → 3
5 → 4

# Bellman-Ford algorithm

## demonstration

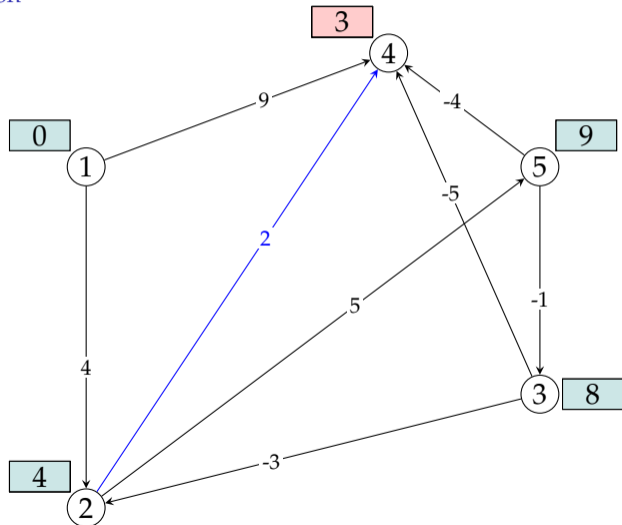


*edges*

1 → 2
1 → 4
2 → 4
2 → 5
3 → 2
3 → 4
5 → 3
5 → 4

# Bellman-Ford algorithm

## demonstration

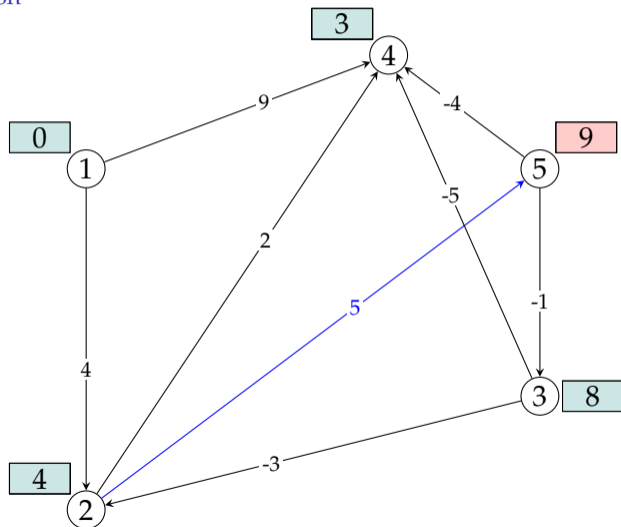


*edges*

1 → 2
1 → 4
2 → 4
2 → 5
3 → 2
3 → 4
5 → 3
5 → 4

# Bellman-Ford algorithm

## demonstration

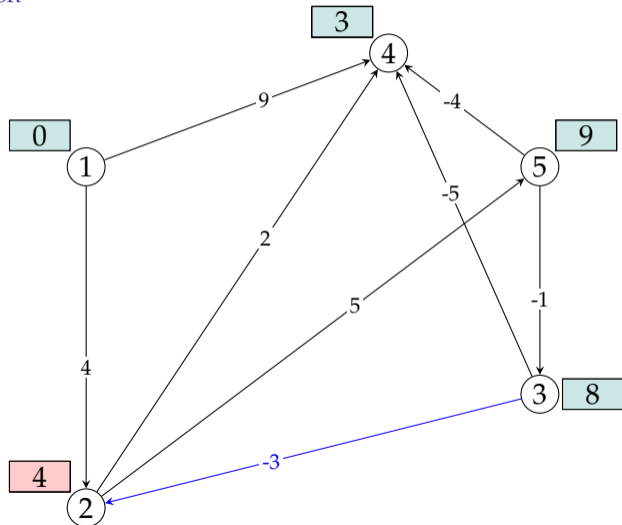


*edges*

1 → 2
1 → 4
2 → 4
2 → 5
3 → 2
3 → 4
5 → 3
5 → 4

# Bellman-Ford algorithm

## demonstration

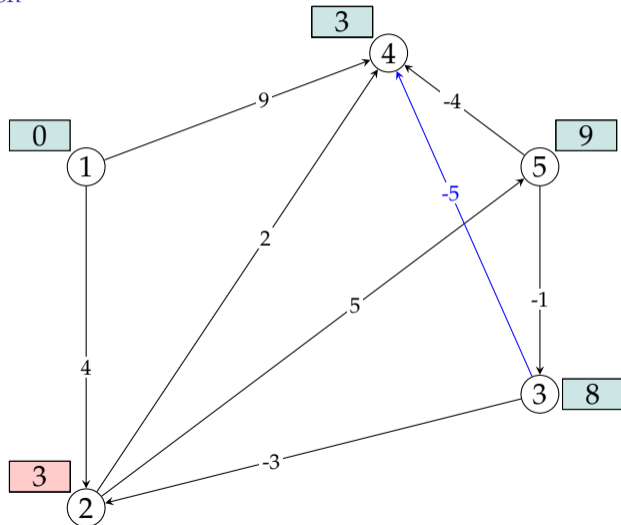


*edges*

1 → 2
1 → 4
2 → 4
2 → 5
3 → 2
3 → 4
5 → 3
5 → 4

# Bellman-Ford algorithm

## demonstration

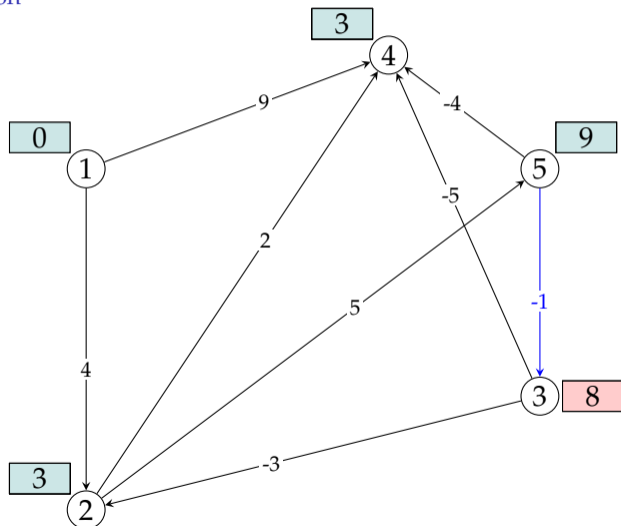


*edges*

1 → 2
1 → 4
2 → 4
2 → 5
3 → 2
3 → 4
5 → 3
5 → 4

# Bellman-Ford algorithm

## demonstration



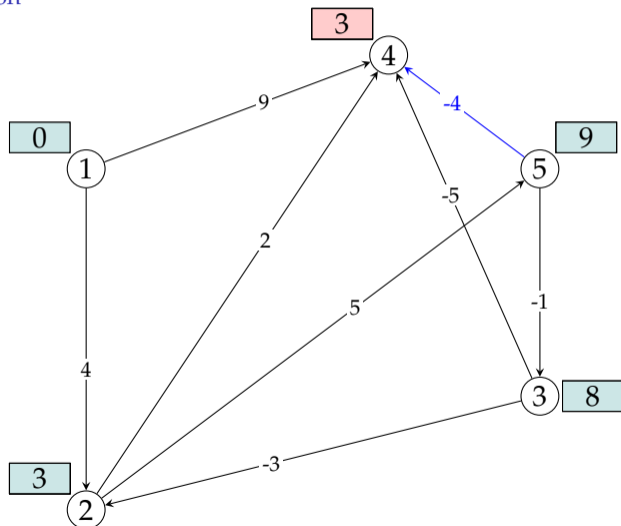
*edges*

1 → 2
1 → 4
2 → 4
2 → 5
3 → 2
3 → 4
5 → 3
5 → 4



# Bellman-Ford algorithm

## demonstration



*edges*

1 → 2
1 → 4
2 → 4
2 → 5
3 → 2
3 → 4
5 → 3
5 → 4

# Summary

- Shortest path algorithms are one of the most applied graph algorithms
- We revised three algorithms
  - Dijkstra's: non-negative weights, general algorithm
  - For DAGs: unrestricted weights, following topological order
  - Bellman-Ford: no negative cycles, digraphs
- Reading: Goodrich, Tamassia, and Goldwasser (2013, chapter 14)

Next:

- Maps and hashing
- Reading: Goodrich, Tamassia, and Goldwasser (2013, chapter 10)

# Acknowledgments, credits, references



Goodrich, Michael T., Roberto Tamassia, and Michael H. Goldwasser (2013).  
*Data Structures and Algorithms in Python*. John Wiley & Sons, Incorporated. ISBN:  
9781118476734.







