Algorithms for Programming Contests - Week 4

Tobias Meggendorfer, Philipp Meyer,
Christian Müller, Gregor Schwarz
conpra@in.tum.de

07.11.2018

Graphs

A *weighted graph* is a tuple $G = (V, E, c)$, where

- V is a non-empty set of *vertices*,
- E is a set of edges,
- $c : E \to \mathbb{R}$ is the weight function.

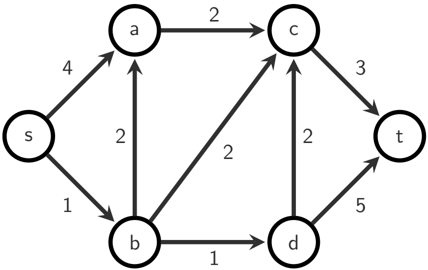
A *directed* graph is a graph with $E \subseteq V \times V = \{(u, v) \mid u, v \in V\}$.

An *undirected* graph is a graph with $E \subseteq \{\{u, v\} \mid u, v \in V\}$.

A *path* from v_1 to v_n is a sequence $p = v_1 v_2 \dots v_n$ such that $(v_i, v_{i+1}) \in E$ for all $i \in [1, n-1]$, and $v_i \neq v_j$ for all $i \neq j$.

The *length of a path* is the sum of its edge weights.

Shortest Path Problem - Classification



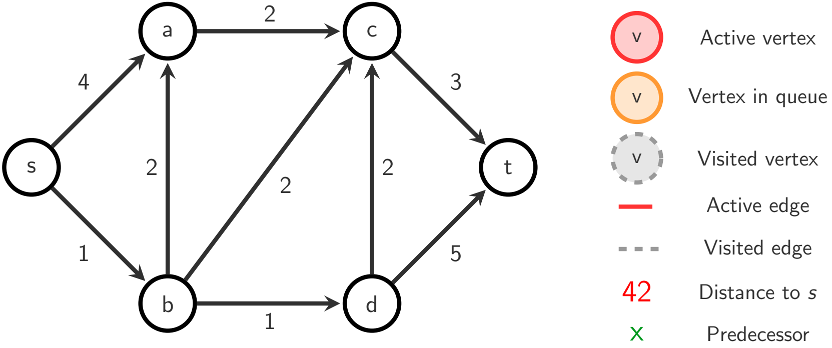
- **Single Pair Shortest Path (SPSP):**
  Find the shortest path between s and t.
- **Single Source Shortest Path (SSSP):**
  Find the shortest path between s and all the other nodes.
- **All Pairs Shortest Path (APSP):**
  Find the shortest path between any pair of nodes.

Shortest Path Problem - Applications

- transportation
- networking and telecommunication
- six degrees of separation
- plant and facility layout
- ...

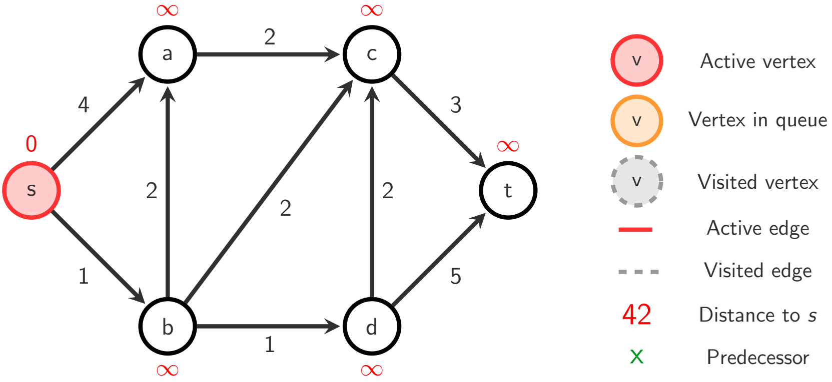
Dijkstra's Algorithm

- Published by Edsger W. Dijkstra in 1959
- Dijkstra's Algorithm solves the SSSP.

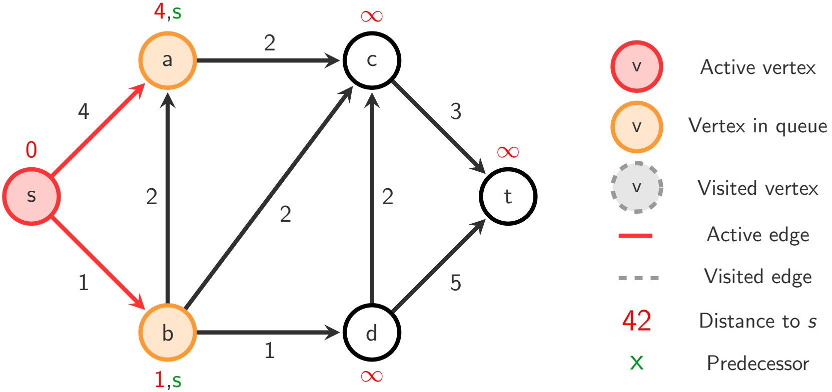


Dijkstra's Algorithm

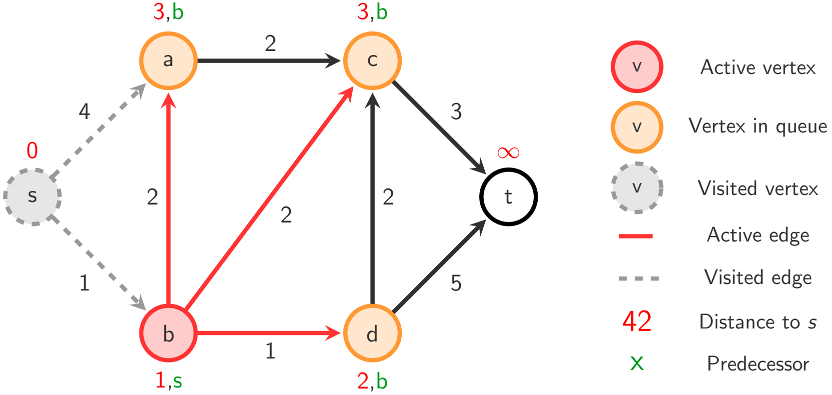
Find the shortest path between s and t!



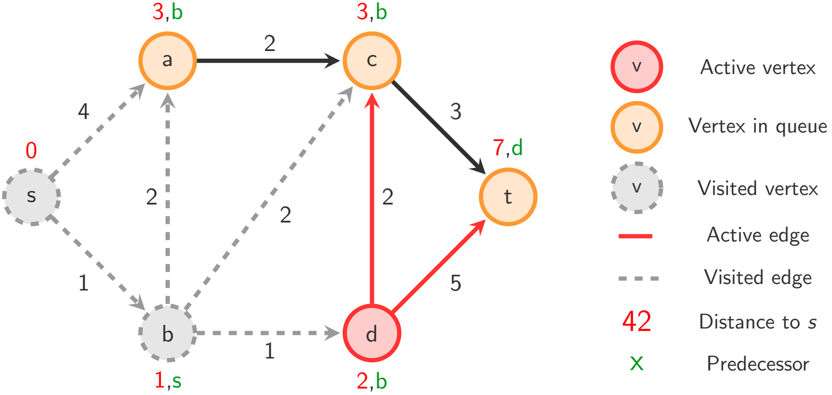
Dijkstra's Algorithm



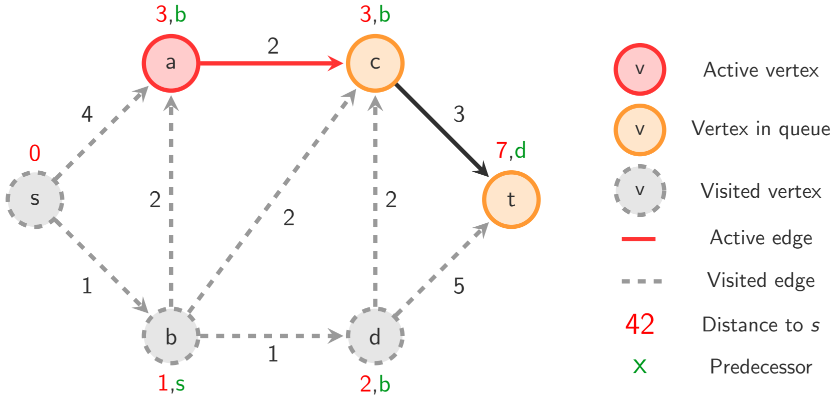
Dijkstra's Algorithm



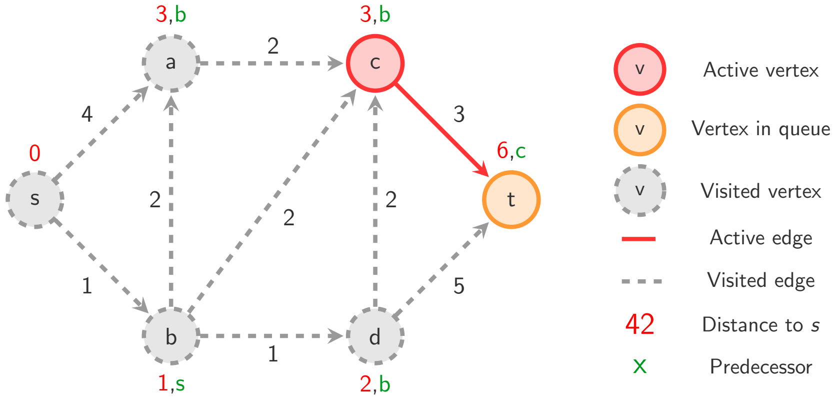
Dijkstra's Algorithm



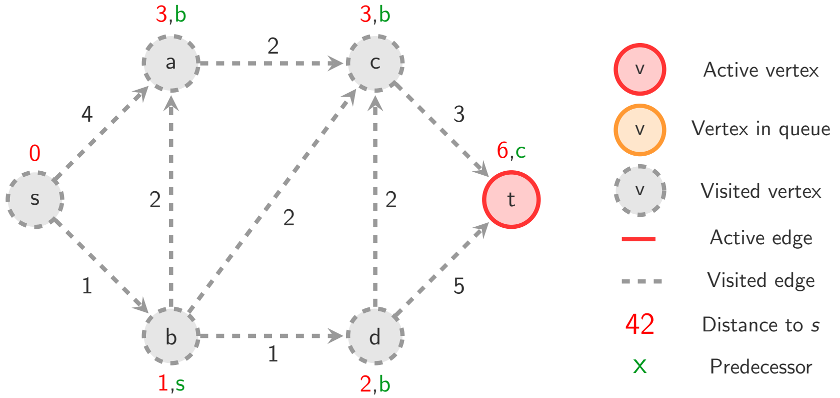
Dijkstra's Algorithm



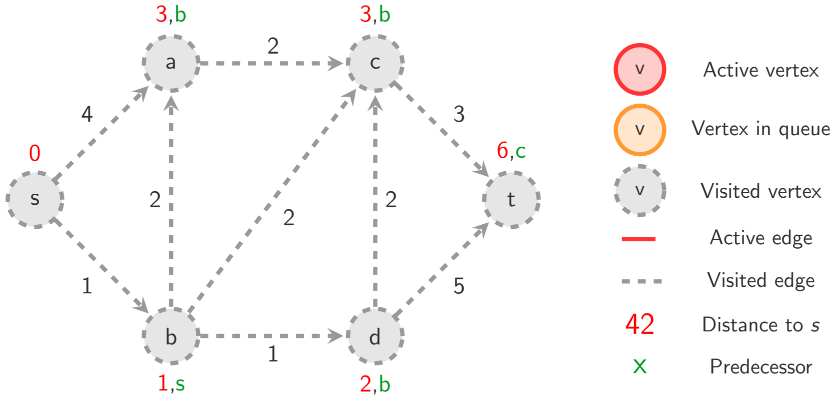
Dijkstra's Algorithm



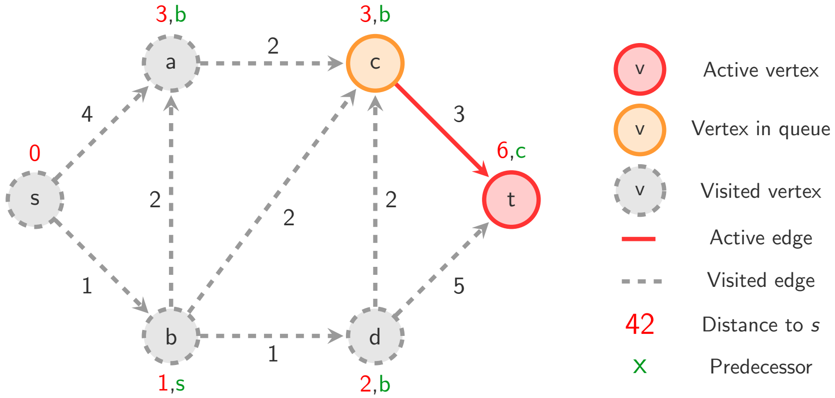
Dijkstra's Algorithm



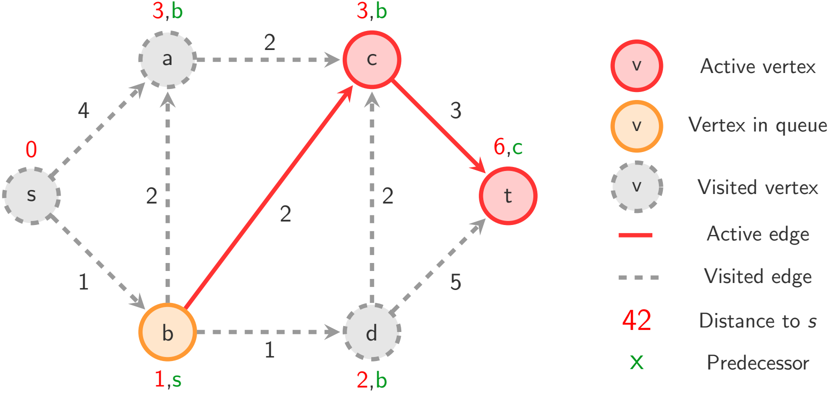
Dijkstra's Algorithm



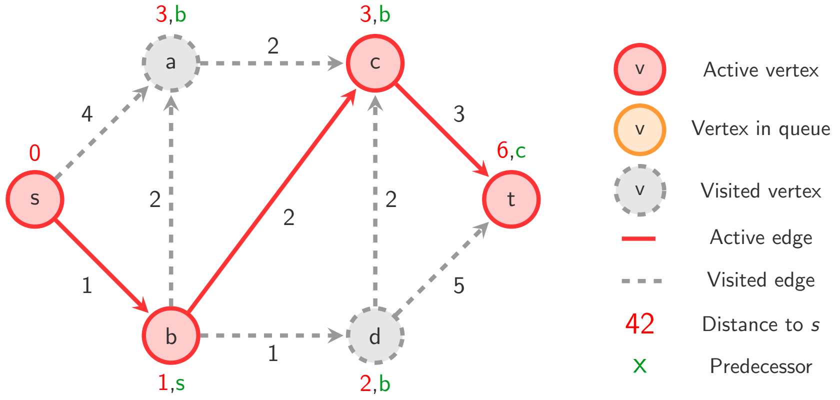
Shortest Path Tree



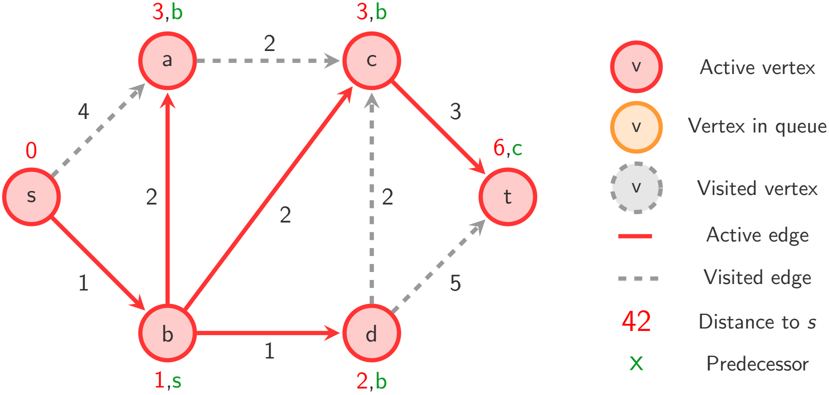
Shortest Path Tree



Shortest Path Tree



Shortest Path Tree



Algorithm 1 Dijkstra's Algorithm

Input: Graph $G = (V, E, c)$

```
procedure DIJKSTRA(G, src)
    for each vertex v ∈ V do
        dist[v] ← ∞, prev[v] ← null
    end for
    dist[src] ← 0
    PQ ← PriorityQueue over V
    for each vertex v ∈ V do
        PQ.insert(v, dist[v])
    end for
    while PQ is not empty do
        v ← PQ.deleteMin()
        for each neighbor w of v do
            if dist[v] + c(v, w) < dist[w] then
                dist[w] ← dist[v] + c(v, w)
                PQ.decreaseKey(w, dist[w])
                prev[w] ← v
            end if
        end for
    end while
end procedure
```

Analysis of Dijkstra's Algorithm

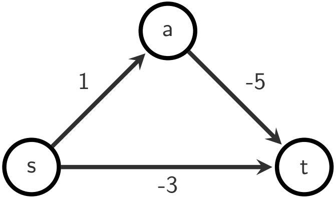
Running time

- With Fibonacci heap as priority queue:
- $|V|$ insert operations: $\mathcal{O}(|V|)$
- $|E|$ decreaseKey operations: $\mathcal{O}(|E|)$
- $|V|$ deleteMin operations: $\mathcal{O}(|V| \log |V|)$
- In total: $\mathcal{O}(|E| + |V| \log |V|)$

Note, that the running time is the same as for Prim's Algorithm.

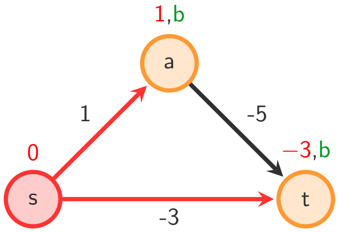
Limitations of Dijkstra's Algorithm

Dijkstra's Algorithm may not work for graphs with negative edge weights!



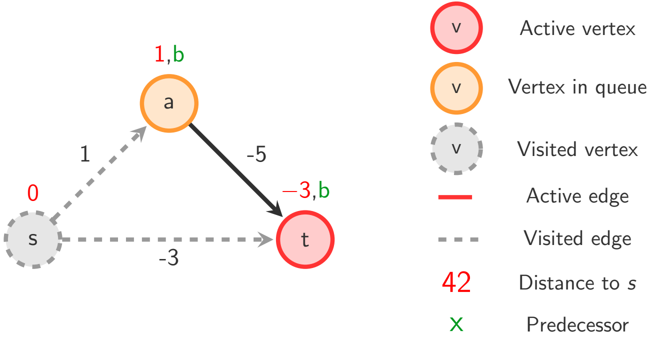
Limitations of Dijkstra's Algorithm

Dijkstra's Algorithm may not work for graphs with negative edge weights!



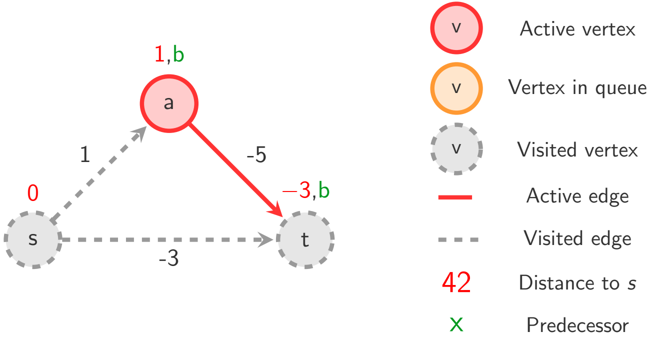
Limitations of Dijkstra's Algorithm

Dijkstra's Algorithm may not work for graphs with negative edge weights!



Limitations of Dijkstra's Algorithm

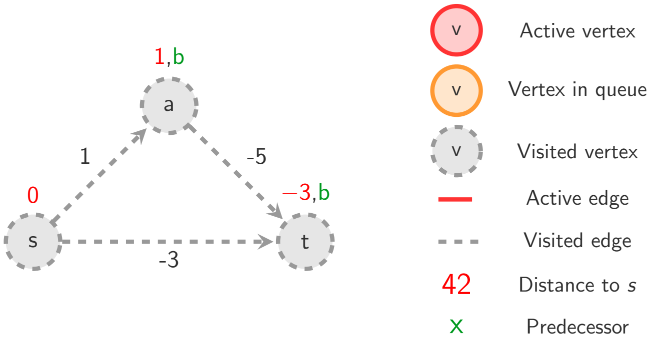
Dijkstra's Algorithm may not work for graphs with negative edge weights!



Vertex t is not updated because it was already visited.

Limitations of Dijkstra's Algorithm

Dijkstra's Algorithm may not work for graphs with negative edge weights!

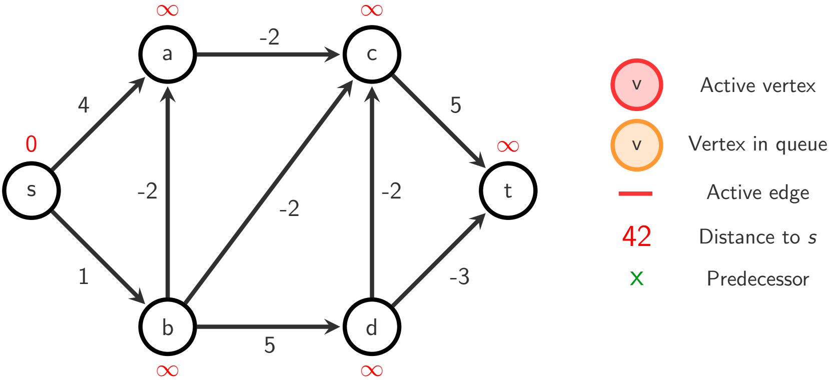


Bellman-Ford Algorithm

- Published by Richard Bellman and Lester Ford in 1958 and 1956 respectively.
- Solves SSSP even if the graph has negative edge weights.
- Idea: Start with shortest paths of length 1 and then successively construct all shortest paths of length 2, 3, ..., $|V| - 1$.

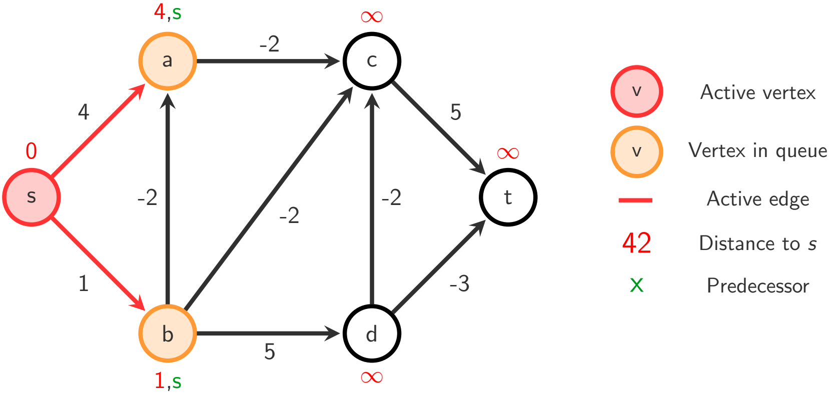
Bellman-Ford Algorithm

$Q = (s)$



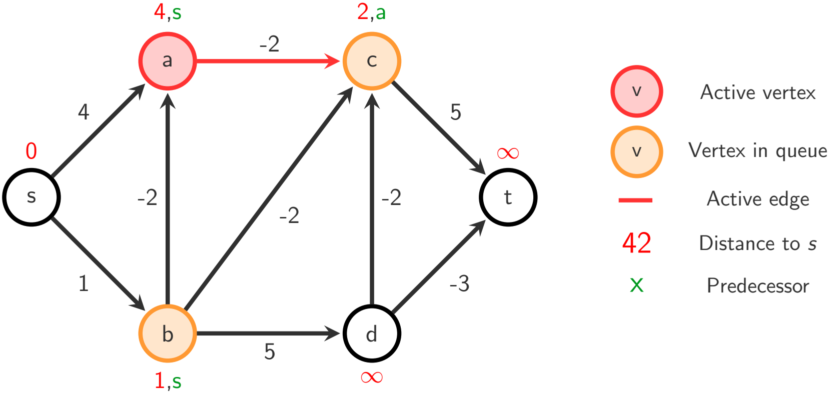
Bellman-Ford Algorithm

$Q = (a, b)$



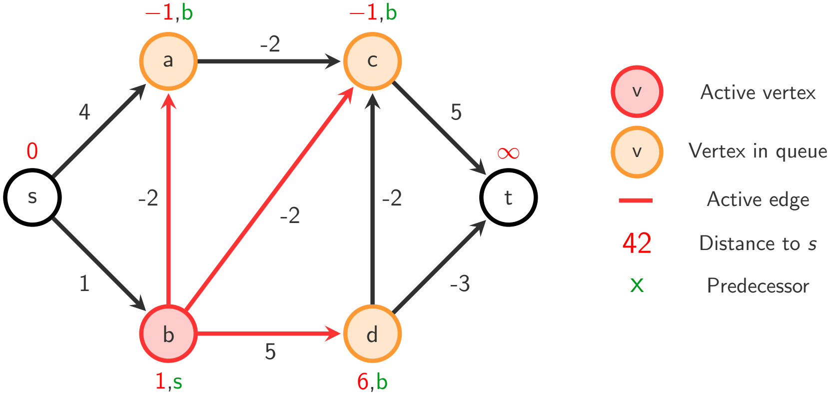
Bellman-Ford Algorithm

$Q = (b, c)$



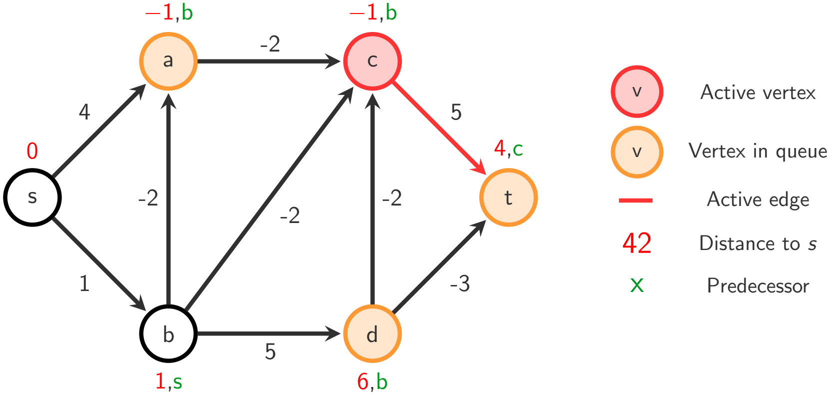
Bellman-Ford Algorithm

$Q = (c, a, d)$



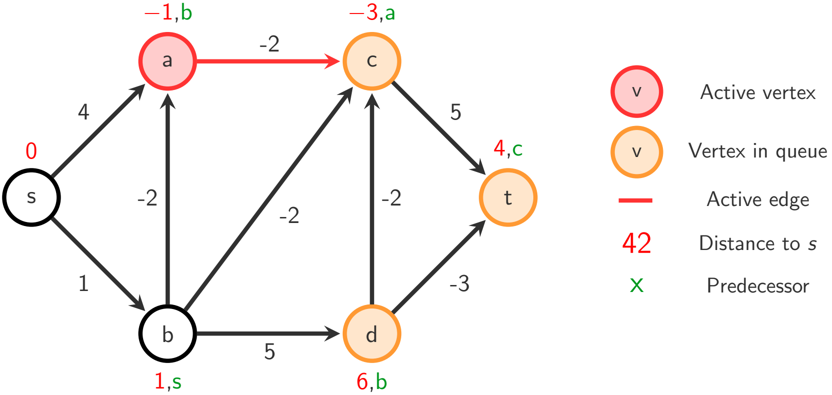
Bellman-Ford Algorithm

$Q = (a, d, t)$



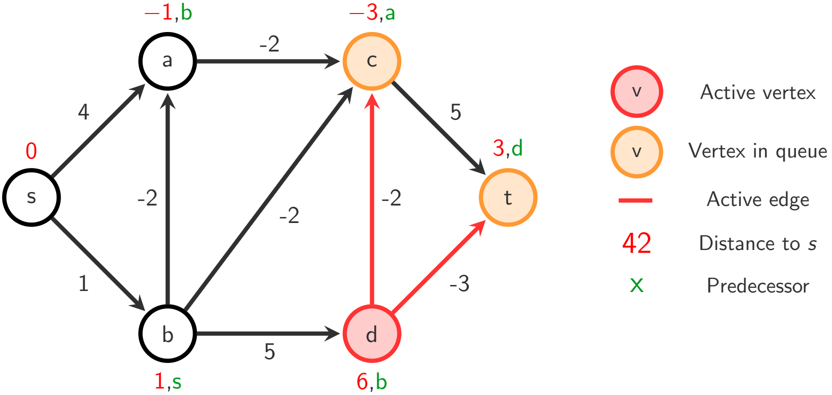
Bellman-Ford Algorithm

$Q = (d, t, c)$



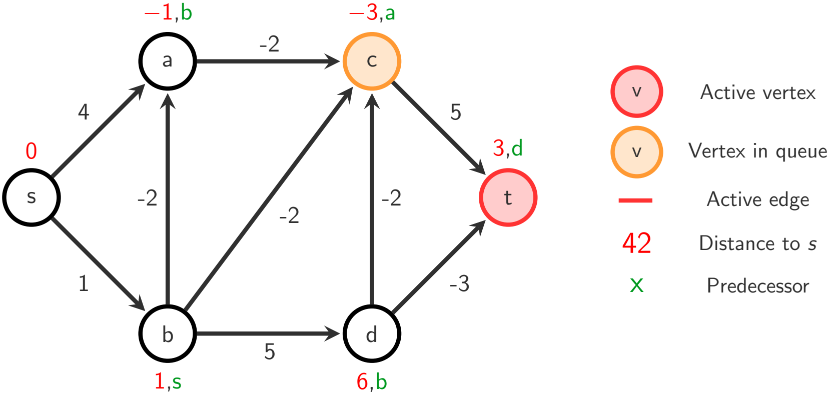
Bellman-Ford Algorithm

$Q = (t, c)$



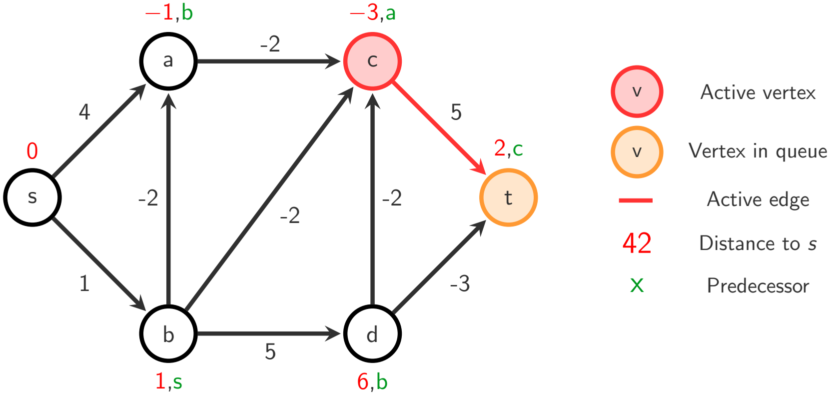
Bellman-Ford Algorithm

$Q = (c)$



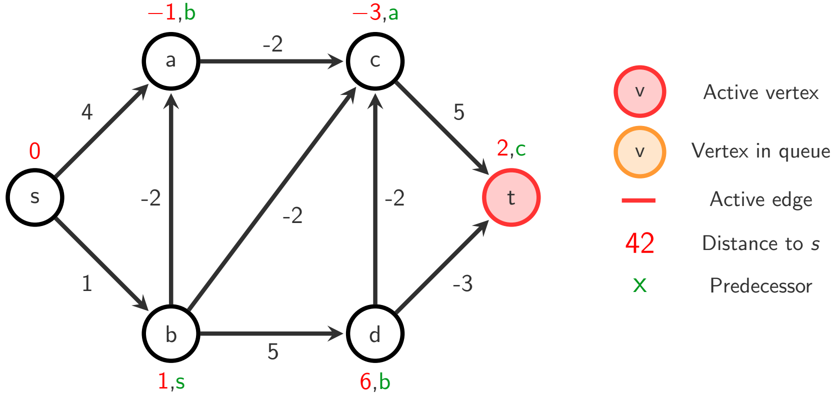
Bellman-Ford Algorithm

$Q = (t)$



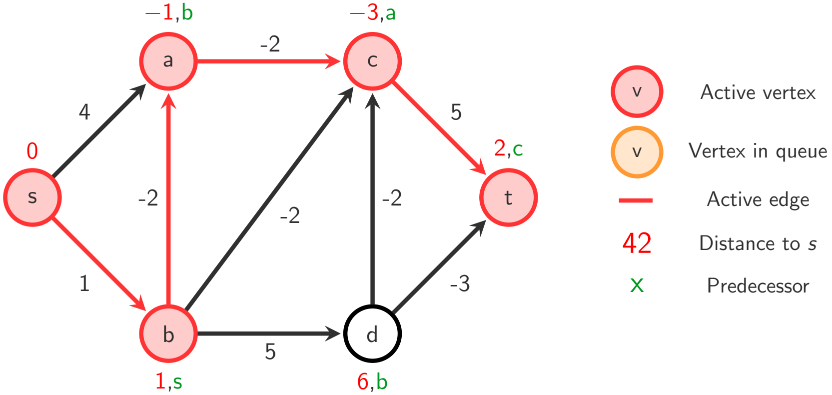
Bellman-Ford Algorithm

$Q = ()$



Bellman-Ford Algorithm

$Q = ()$

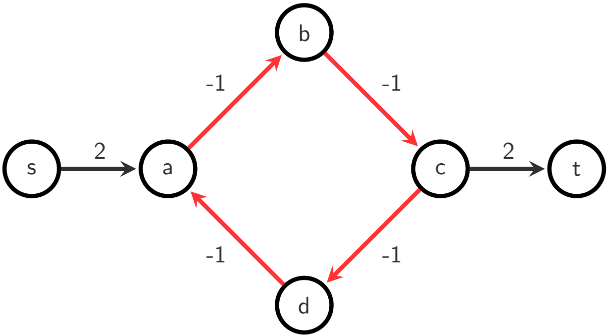


Algorithm 2 Bellman-Ford Algorithm (no negative cycles)

```
Input: Graph G = (V, E, c) with no negative cycles
  procedure BELLMAN-FORD(G, src)
    for each vertex v ∈ V do
      dist[v] ← ∞, prev[v] ← null
    end for
    dist[src] ← 0
    Q ← FIFO-Queue
    Q.insert(src)
    while Q is not empty do
      v ← Q.pop()
      for each neighbor w of v do
        if dist[v] + c(v, w) < dist[w] then
          dist[w] ← dist[v] + c(v, w)
          prev[w] ← v
          if w not in Q then
            Q.push(w)
          end if
        end if
      end for
    end while
  end procedure
```

Negative Cycles

- If there are negative cycles in the graph, the distance between s and t can become arbitrarily short.
- Detection of negative cycles becomes necessary.



Negative Cycle Detection

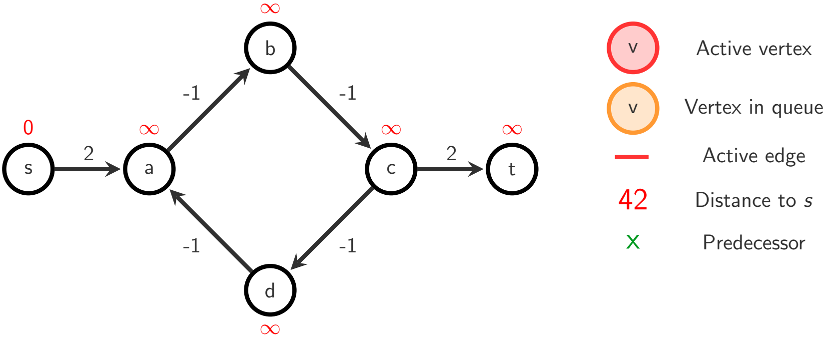
- Idea: Process FIFO-Queue in phases.
- One phase = processing all nodes currently in the queue.
- After phase i, all shortest paths of length i were detected.
- Longest shortest path contains at most $n-1$ edges if there is no negative cycle.
- If there are nodes left in the queue after phase n, then there is a negative cycle.
- Cycle can be constructed by recursively visiting the predecessors of a node that is left in the queue after phase n.

Algorithm 3 Bellman-Ford Algorithm (negative cycle detection)

```
Input: Graph G = (V, E, c)
  procedure BELLMAN-FORD(G, src)
      for each vertex v ∈ V do
          dist[v] ← ∞, prev[v] ← null
      end for
      dist[src] ← 0
      Q, Q′ ← FIFO-Queue
      Q.insert(src)
      for phase 1 to |V| do
          while Q is not empty do
              v ← Q.pop()
              for each neighbor w of v do
                  if dist[v] + c(v, w) < dist[w] then
                      dist[w] ← dist[v] + c(v, w)
                      prev[w] ← v
                      if w not in Q′ then
                          Q′.push(w)
                      end if
                  end if
              end for
          end while
          swap(Q, Q′)
      end for
      if Q is not empty then
          return there exists a negative cycle
      end if
  end procedure
```

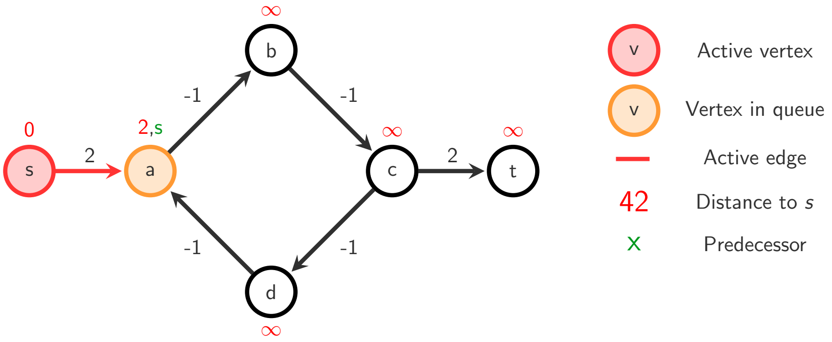
Bellman-Ford Algorithm with Negative Cycles

Initialization



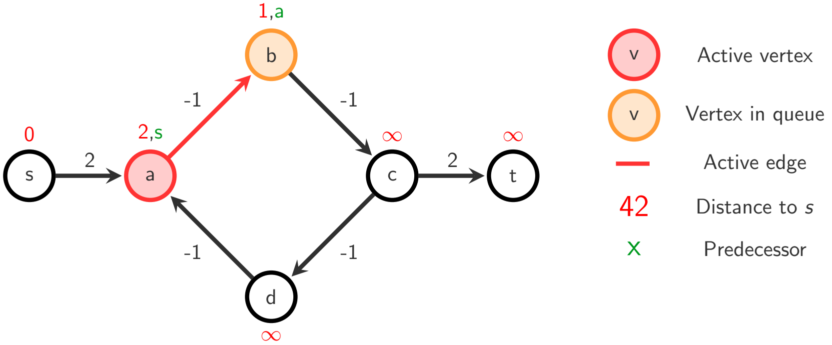
Bellman-Ford Algorithm with Negative Cycles

Phase 1: $Q = (s) \longrightarrow Q' = (a)$



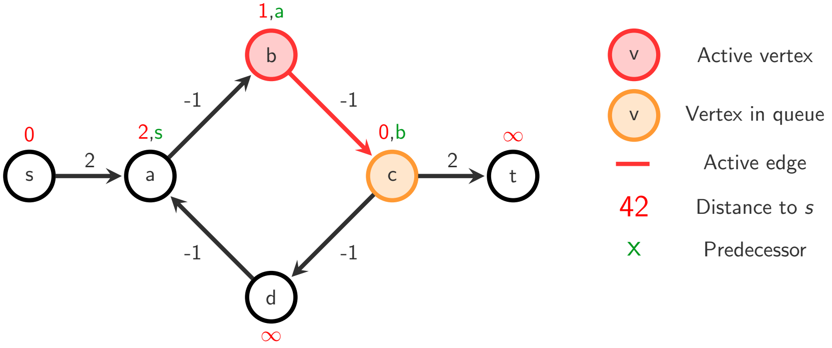
Bellman-Ford Algorithm with Negative Cycles

Phase 2: $Q = (a) \longrightarrow Q' = (b)$



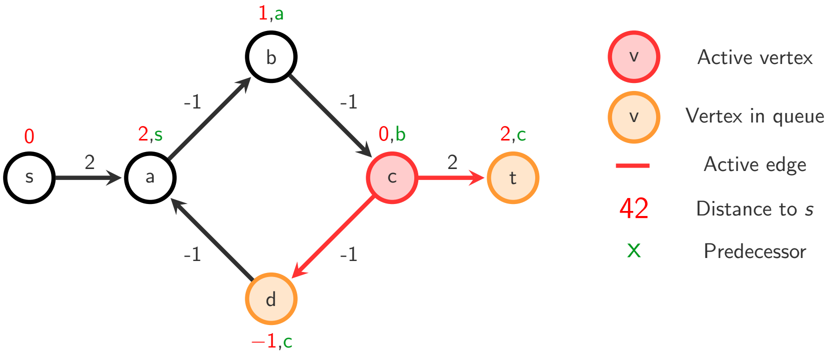
Bellman-Ford Algorithm with Negative Cycles

Phase 3: $Q = (b) \longrightarrow Q' = (c)$



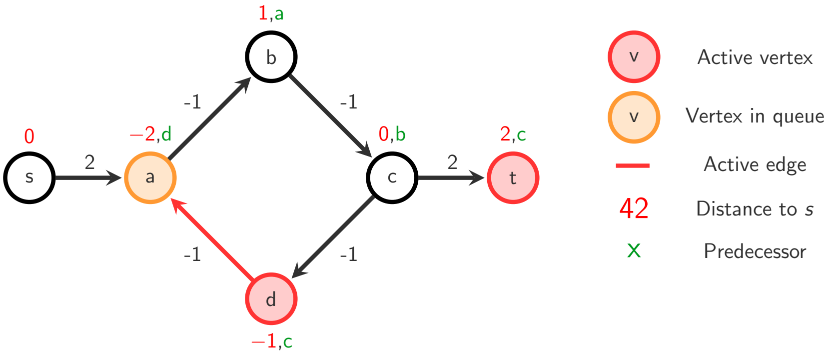
Bellman-Ford Algorithm with Negative Cycles

Phase 4: $Q = (c) \longrightarrow Q' = (d, t)$



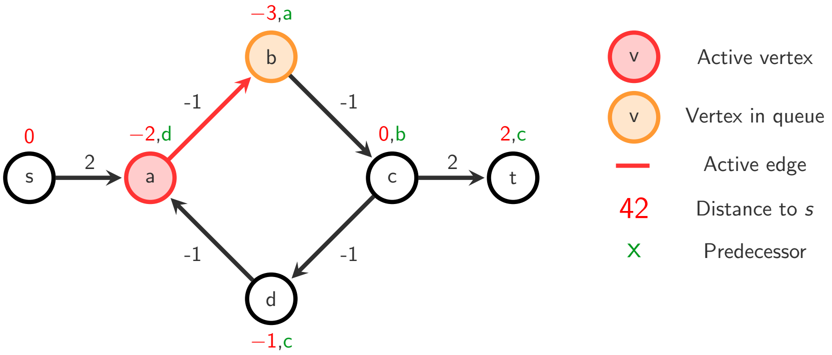
Bellman-Ford Algorithm with Negative Cycles

Phase 5: $Q = (d, t) \longrightarrow Q' = (a)$



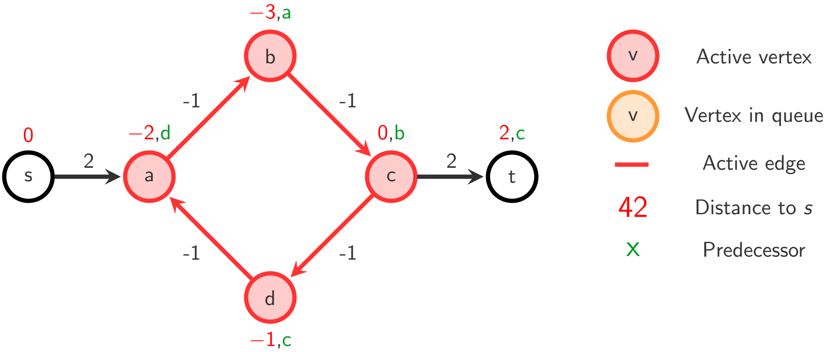
Bellman-Ford Algorithm with Negative Cycles

Phase 6: $Q = (a) \longrightarrow Q' = (b)$



Bellman-Ford Algorithm with Negative Cycles

After phase $6 = |V|$: $Q = (b)$
The queue is not empty $\rightarrow$ negative cycle $\rightarrow$ predecessor backtracking



Analysis of Bellman-Ford Algorithm

Running time

- At most $\mathcal{O}(|V|)$ phases.
- One phase takes at most $\mathcal{O}(|V| + |E|)$ operations.

  Pop all $|V|$ nodes, consider all $|E|$ edges, push all $|V|$ nodes.
- In total: $\mathcal{O}(|V||E|)$

Floyd-Warshall Algorithm

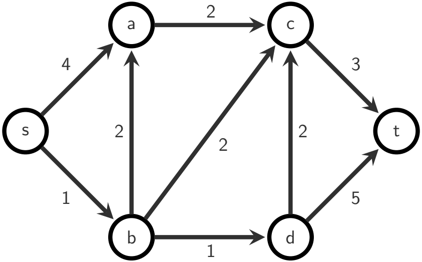
How to solve APSP?

- **Naive approach:** Executing Dijkstra algorithm $|V|$ times
  - Runtime: $\mathcal{O}(|V||E| + |V|^2 \log |V|)$
  - Can neither handle negative edge weights nor negative cycles.
- **Floyd-Warshall Algorithm**:
  - Runtime $\mathcal{O}(|V|^3)$
  - Can handle negative edge weights.
  - Negative cycle detection possible.
  - Easy to code.

$\Rightarrow$ Apply the naive approach if the graph is sparse!

Floyd-Warshall Algorithm

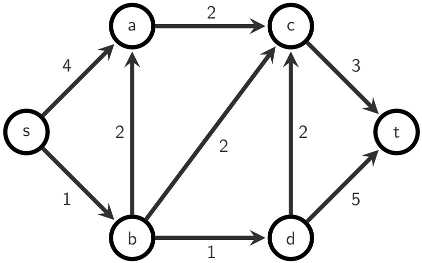
- Represent graph in distance matrix.
- Idea: successively add vertices as intermediate nodes for shortest paths.



$$\text{dist} = \begin{array}{c} \\ s \\ a \\ b \\ c \\ d \\ t \end{array} \begin{array}{c} \begin{array}{cccccc} s & a & b & c & d & t \end{array} \\ \begin{pmatrix} 0 & 4 & 1 & \infty & \infty & \infty \\ \infty & 0 & \infty & 2 & \infty & \infty \\ \infty & 2 & 0 & 2 & 1 & \infty \\ \infty & \infty & \infty & 0 & \infty & 3 \\ \infty & \infty & \infty & 2 & 0 & 5 \\ \infty & \infty & \infty & \infty & \infty & 0 \end{pmatrix} \end{array}$$

Floyd-Warshall Algorithm

- When considering a vertex k as intermediate node, there are two possibilities:
  - Shortest path between i and j does not go over k.
  - Shortest path between i and j uses k as intermediate node.
- Update: $\text{dist}[i][j] = \min\{\text{dist}[i][j], \text{dist}[i][k] + \text{dist}[k][j]\}$



$$\text{dist} = \begin{array}{c} \\ s \\ a \\ b \\ c \\ d \\ t \end{array} \begin{array}{c} \begin{array}{cccccc} s & a & b & c & d & t \end{array} \\ \begin{pmatrix} 0 & 4 & 1 & \infty & \infty & \infty \\ \infty & 0 & \infty & 2 & \infty & \infty \\ \infty & 2 & 0 & 2 & 1 & \infty \\ \infty & \infty & \infty & 0 & \infty & 3 \\ \infty & \infty & \infty & 2 & 0 & 5 \\ \infty & \infty & \infty & \infty & \infty & 0 \end{pmatrix} \end{array}$$

Algorithm 4 Floyd-Warshall Algorithm

Input: Graph $G = (V, E, c)$

```
procedure FLOYD-WARSHALL(G)
    dist[][] ← array of size |V| × |V| initialized to ∞
    for each vertex v ∈ V do
        dist[v][v] ← 0
    end for
    for each edge (u, v) ∈ E do
        dist[u][v] ← c(u, w)
    end for
    for each vertex k ∈ V do
        for each vertex i ∈ V do
            for each vertex j ∈ V do
                if dist[i][k] + dist[k][j] < dist[i][j] then
                    dist[i][j] ← dist[i][k] + dist[k][j]
                end if
            end for
        end for
    end for
end procedure
```

Analysis of Floyd-Warshall Algorithm

Running time

- Consider each of the $\mathcal{O}(|V|)$ vertices as intermediate node.
- Check if the shortest path between all $\mathcal{O}(|V|^2)$ vertex pairs becomes shorter by passing over intermediate node.
- In total: $\mathcal{O}(|V|^3)$

Floyd-Warshall Algorithm

- Order of loops matter: $k \rightarrow i \rightarrow j$
- Negative cycles exists $\Leftrightarrow$ negative entries on diagonal of matrix.
- Shortest path tree can be reconstructed by bookkeeping the update steps in another $|V| \times |V|$ matrix.
- Floyd-Warshall algorithm is an example of Dynamic Programming (discussed later in class).
- Other application: computation of transitive closure.

Longest Path Problem

- **Longest Path Problem:** Find a simple path of maximum length between two nodes in a graph.
- NP-hard for general graphs.
- Polynomial time algorithms exist for directed acyclic graphs.
- Application in DAGs: Finding critical paths in scheduling problems.

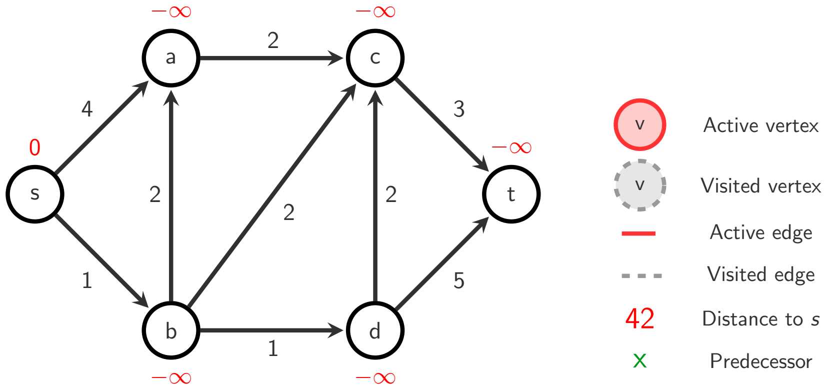
Longest Path Problem

- Approach 1:
  - Negate all edge weights in given DAG.
  - The shortest path in the modified graph is the longest path in the original graph.
  - Use Bellman-Ford to compute shortest path.
  - Complexity: $\mathcal{O}(|V||E|)$

- Approach 2:
  - Compute topological ordering of nodes in DAG.
  - Process nodes in topological order.
  - For each node v in the DAG check whether the distance to any of its successors can be increased by passing over v.
  - Complexity: $\mathcal{O}(|V| + |E|)$

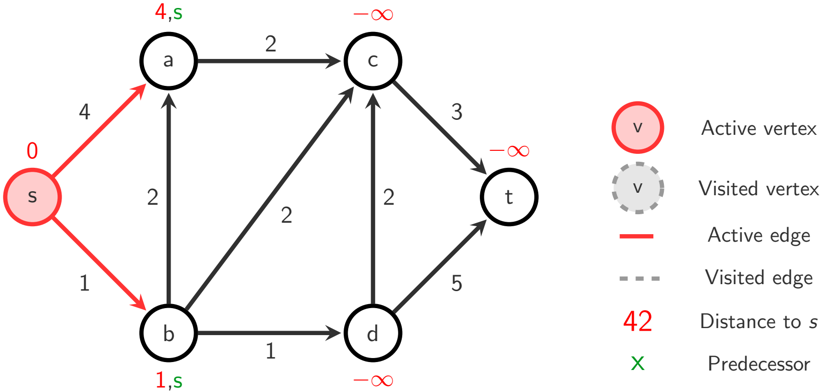
Longest Path in DAG

Topological order: s,b,a,d,c,t



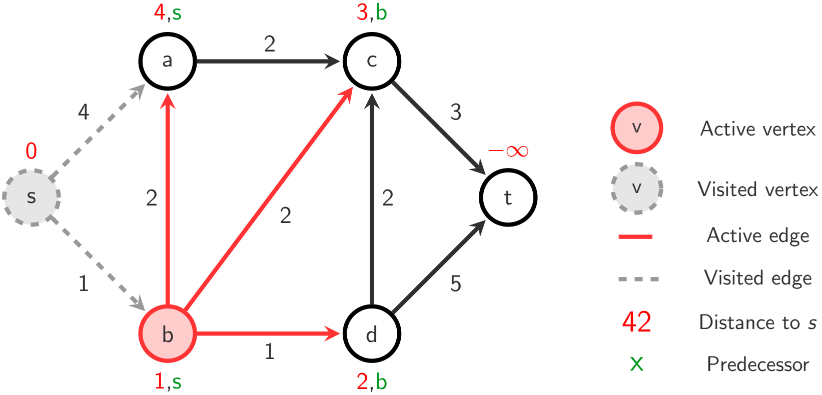
Longest Path in DAG

Topological order: s,b,a,d,c,t



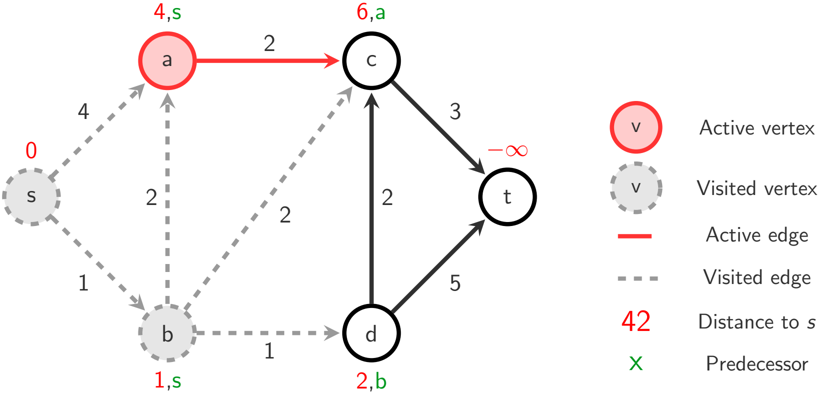
Longest Path in DAG

Topological order: s,b,a,d,c,t



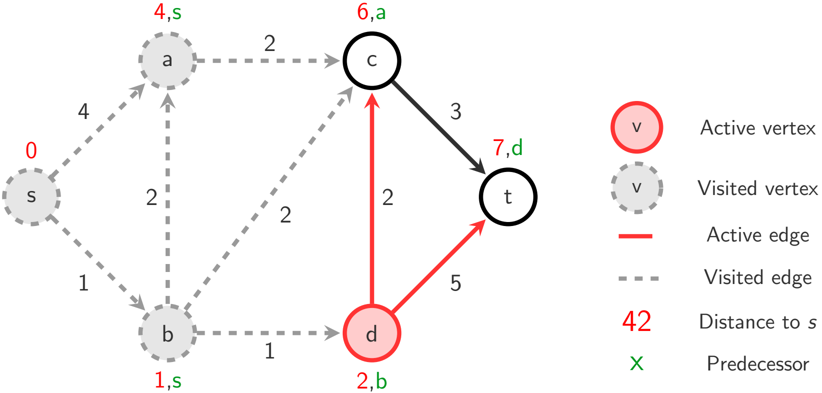
Longest Path in DAG

Topological order: s,b,a,d,c,t



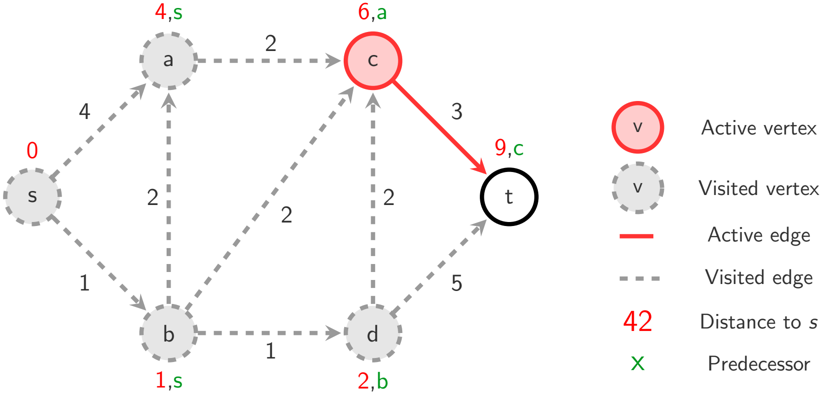
Longest Path in DAG

Topological order: s,b,a,d,c,t



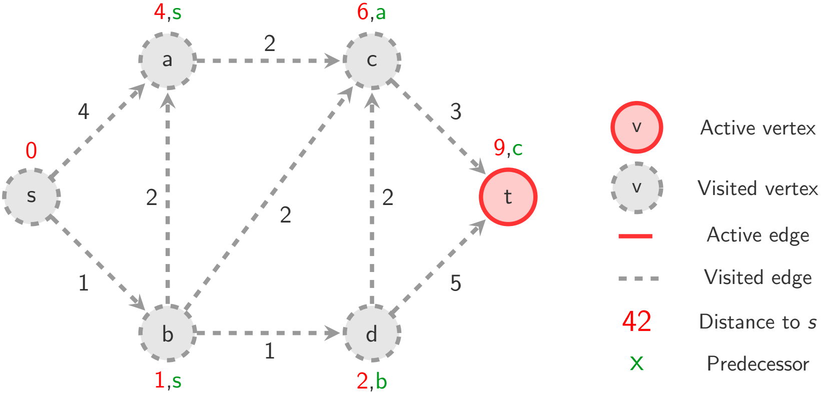
Longest Path in DAG

Topological order: s,b,a,d,c,t



Longest Path in DAG

Topological order: s,b,a,d,c,t



Longest Path in DAG

Topological order: s,b,a,d,c,t