Algorithms for Programming Contests - Week 04

Prof. Dr. Javier Esparza,
Vincent Fischer,
Jakob Schulz,
conpra@model.cit.tum.de

November 4, 2025

Graphs

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

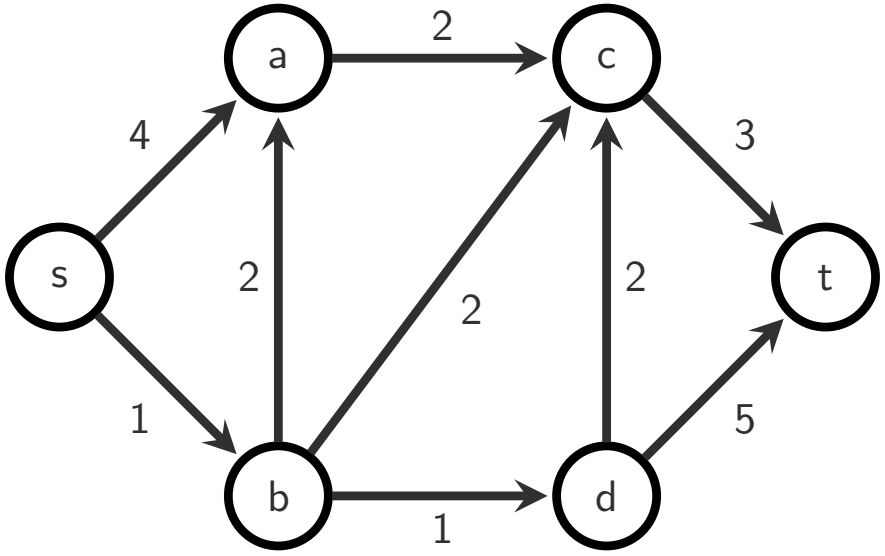
- V is a non-empty set of *vertices*,
- E is a set of edges,
- $c : E \to \mathbb{R}$ is the weight function.

A *simple 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.

Note: Length of a path can also mean the number of edges it traverses. Which one is meant has to be understood from context.

Shortest Path Problem - Classification



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

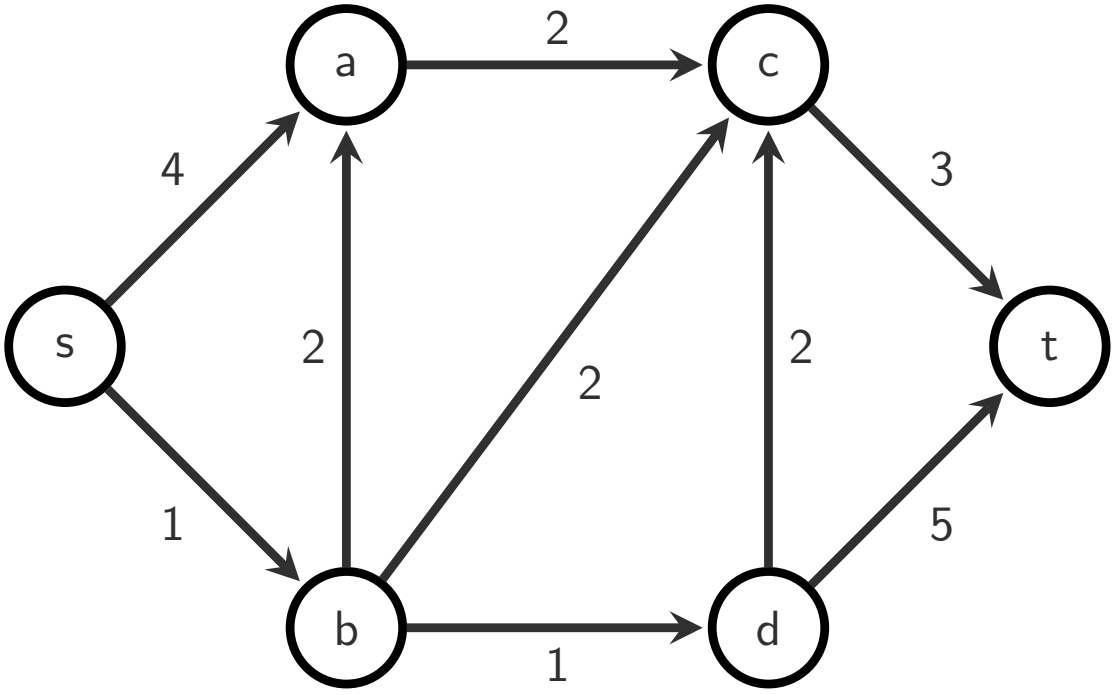
Shortest Path Problem - Applications

- transportation
- networking
- plant and facility layout
- ...

Dijkstra's Algorithm

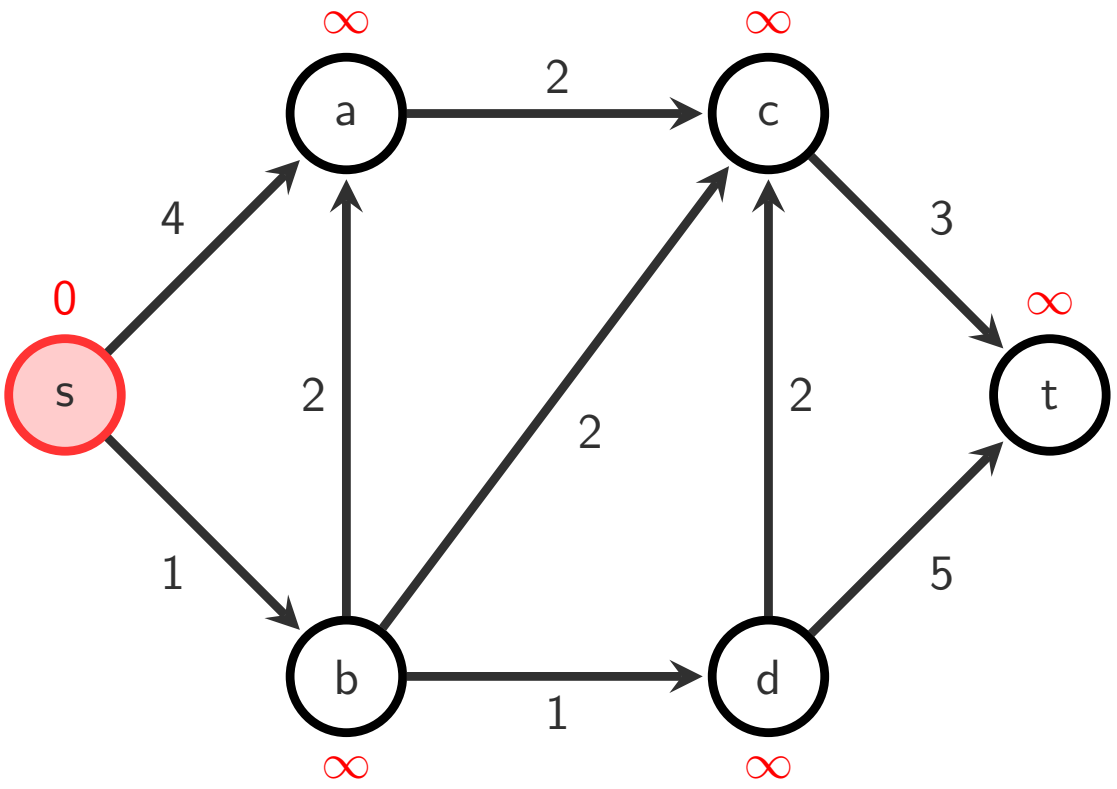
- Published by Edsger W. Dijkstra in 1959.
- Solves the SSSP for graphs with **non-negative** weights.
- Idea: Graph exploration similar to DFS/BFS, but instead of having a stack/queue as worklist, use priority queue.
- Priority given by distance to source state.
- Keep track of predecessors to construct shortest paths.

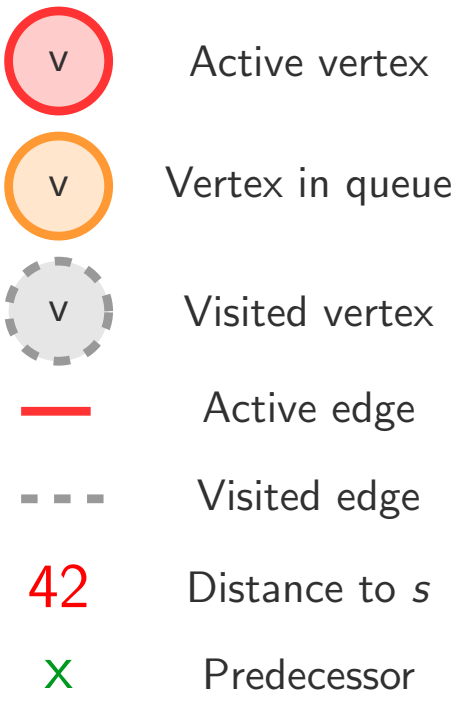
Dijkstra's Algorithm



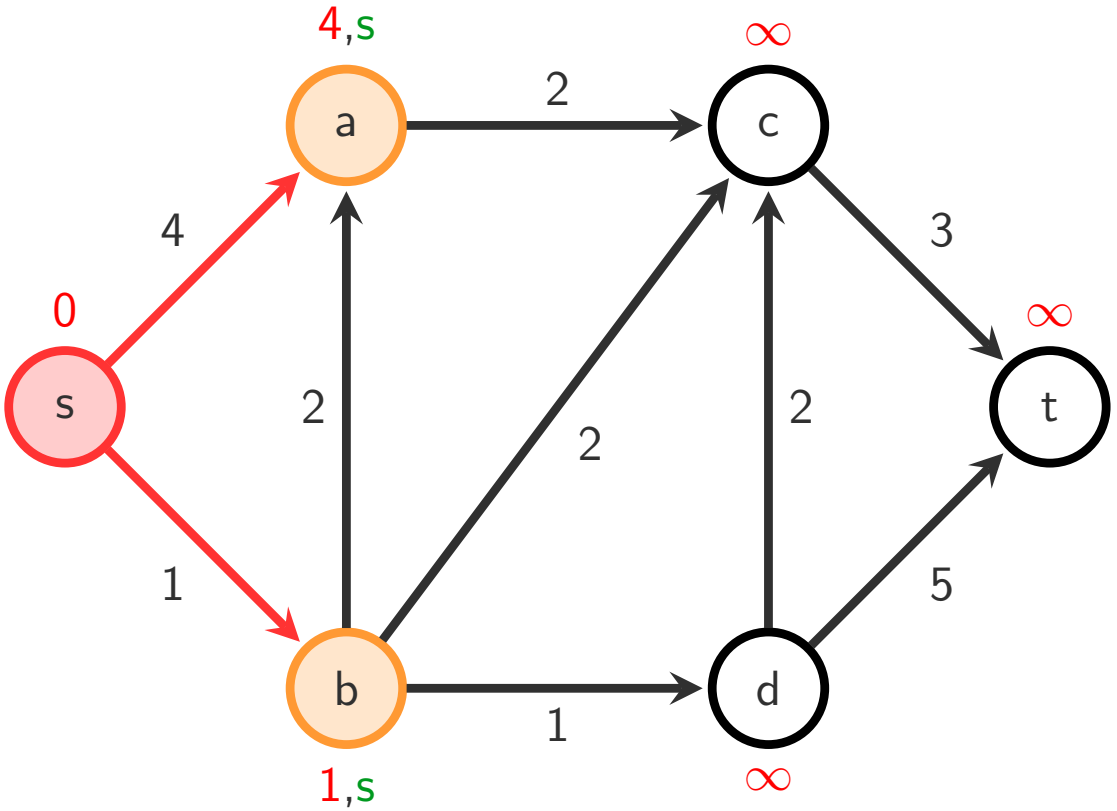
Dijkstra's Algorithm

Find the shortest path between s and t!

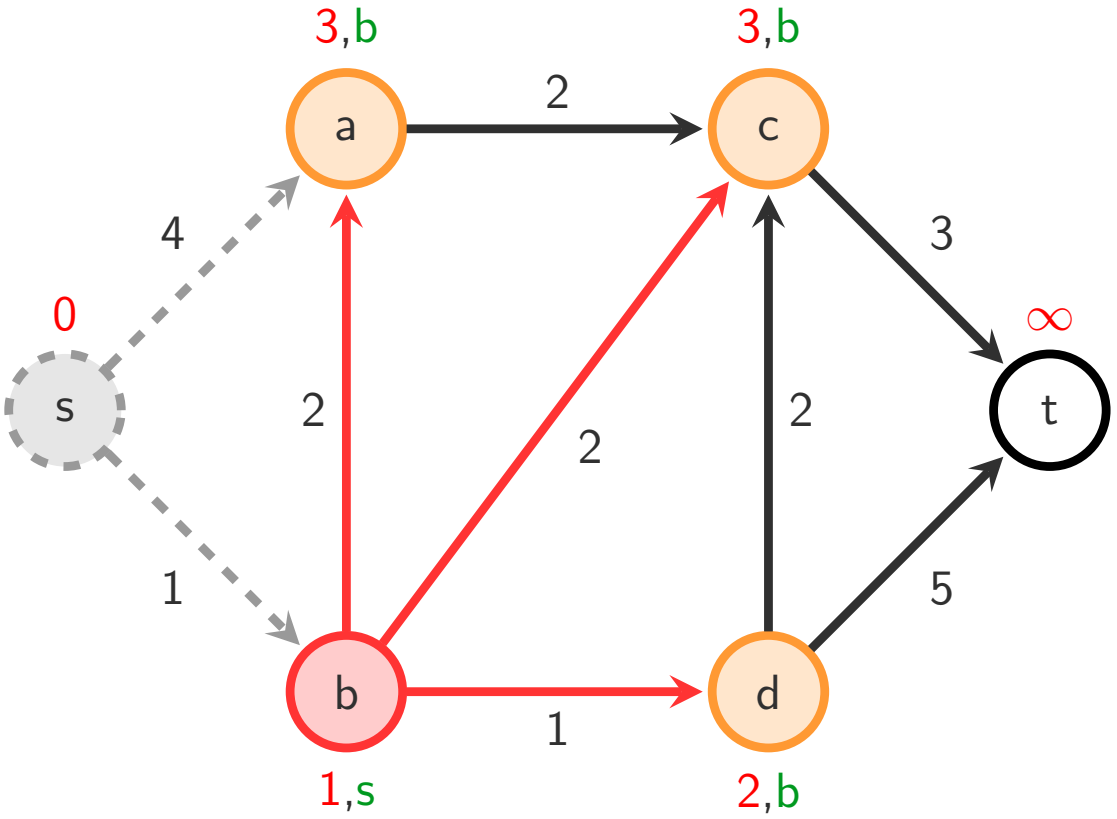


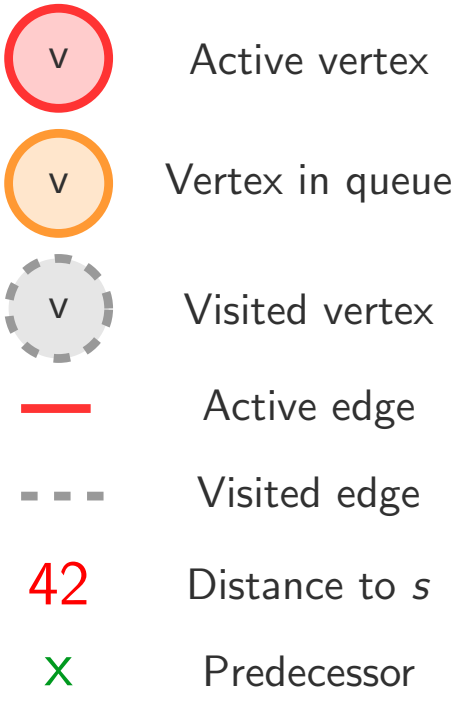


Dijkstra's Algorithm

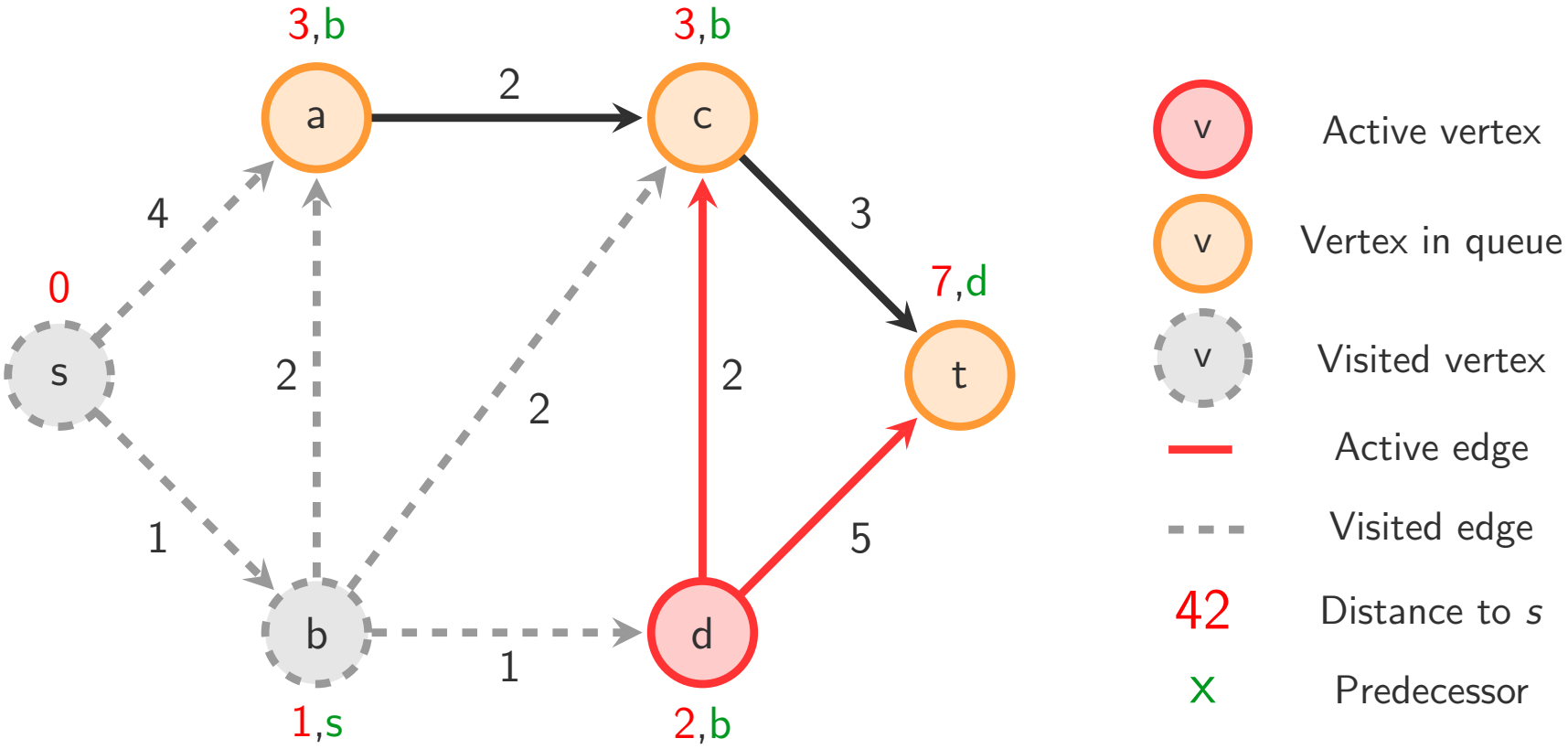


Dijkstra's Algorithm

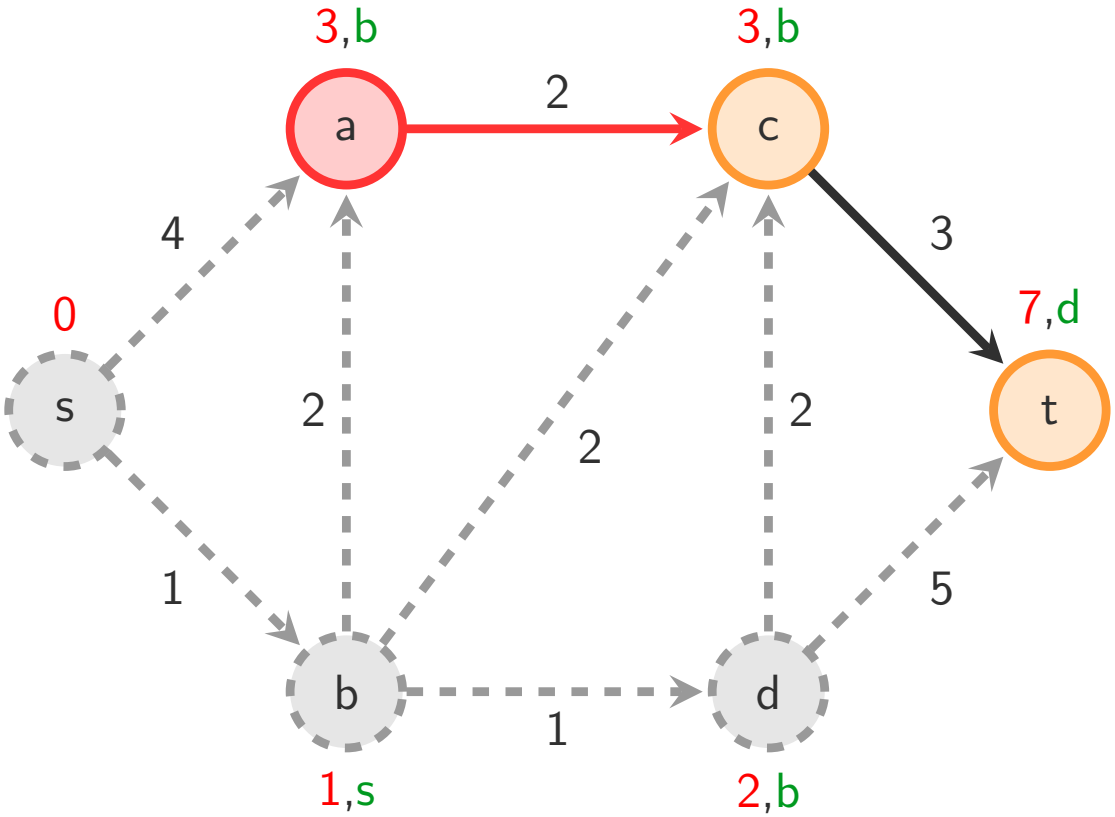


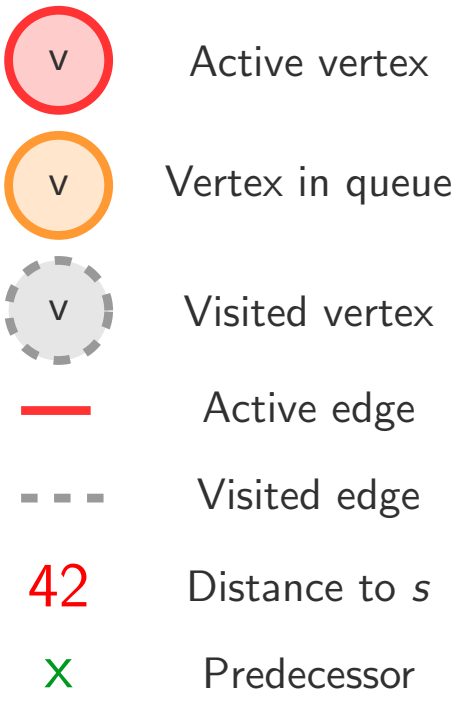


Dijkstra's Algorithm

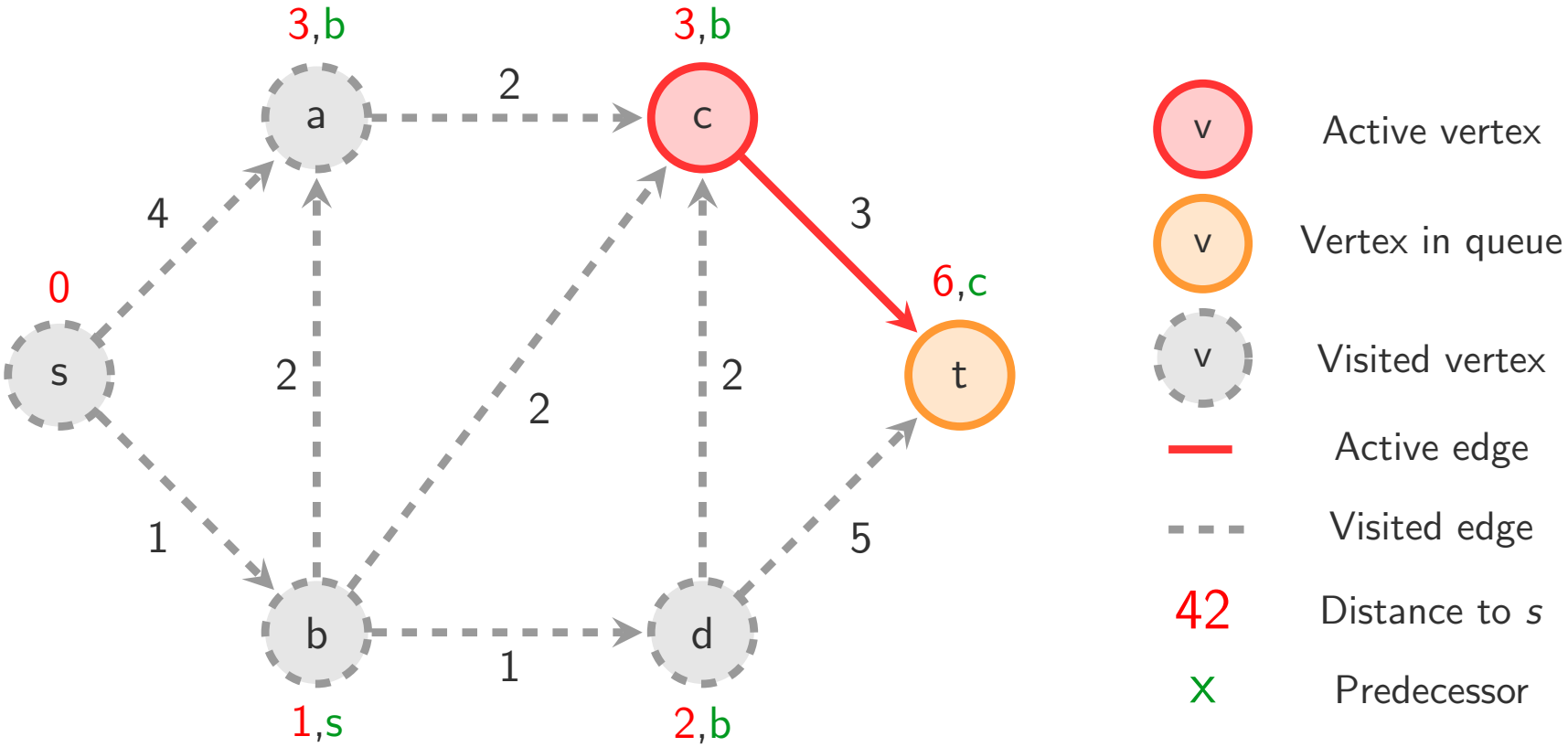


Dijkstra's Algorithm

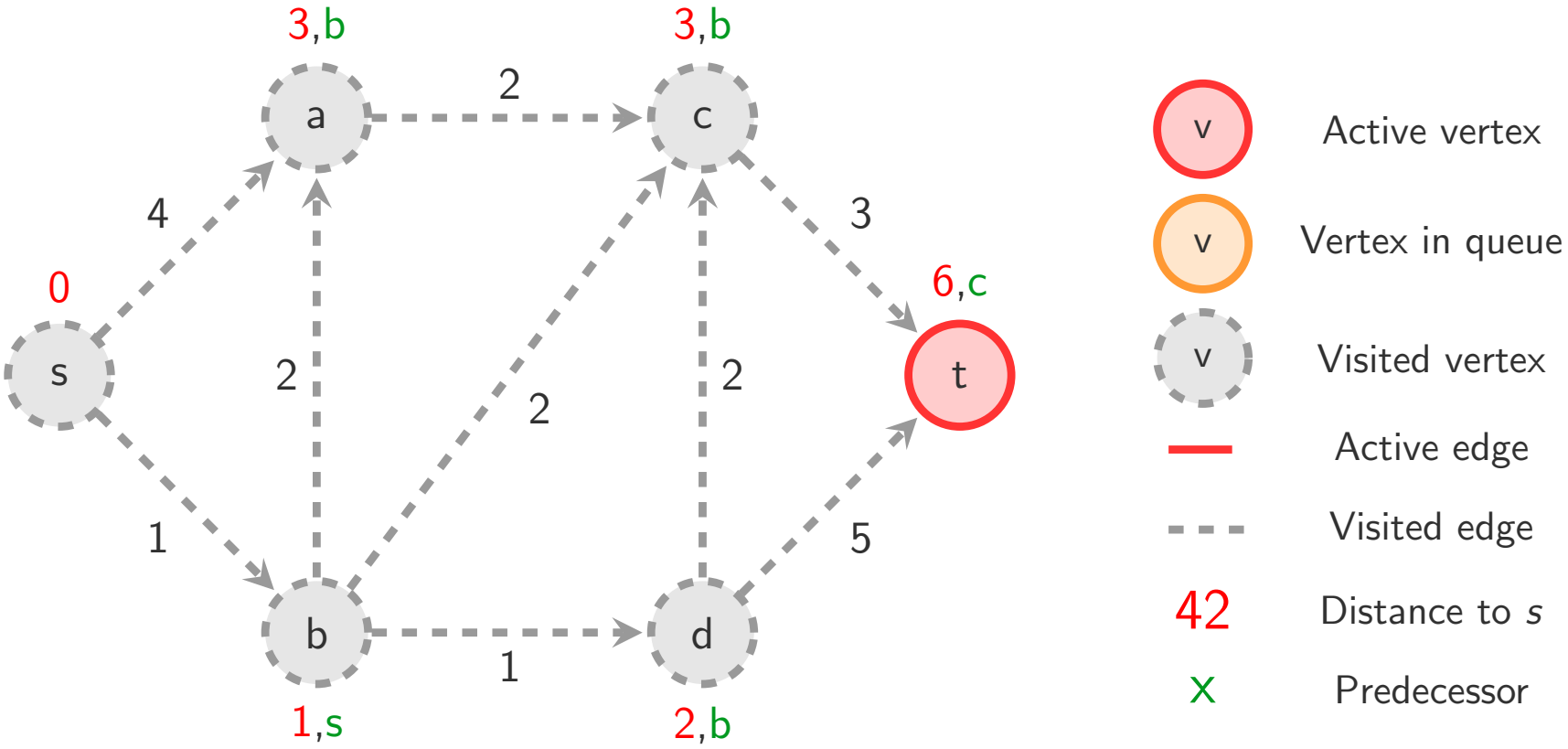




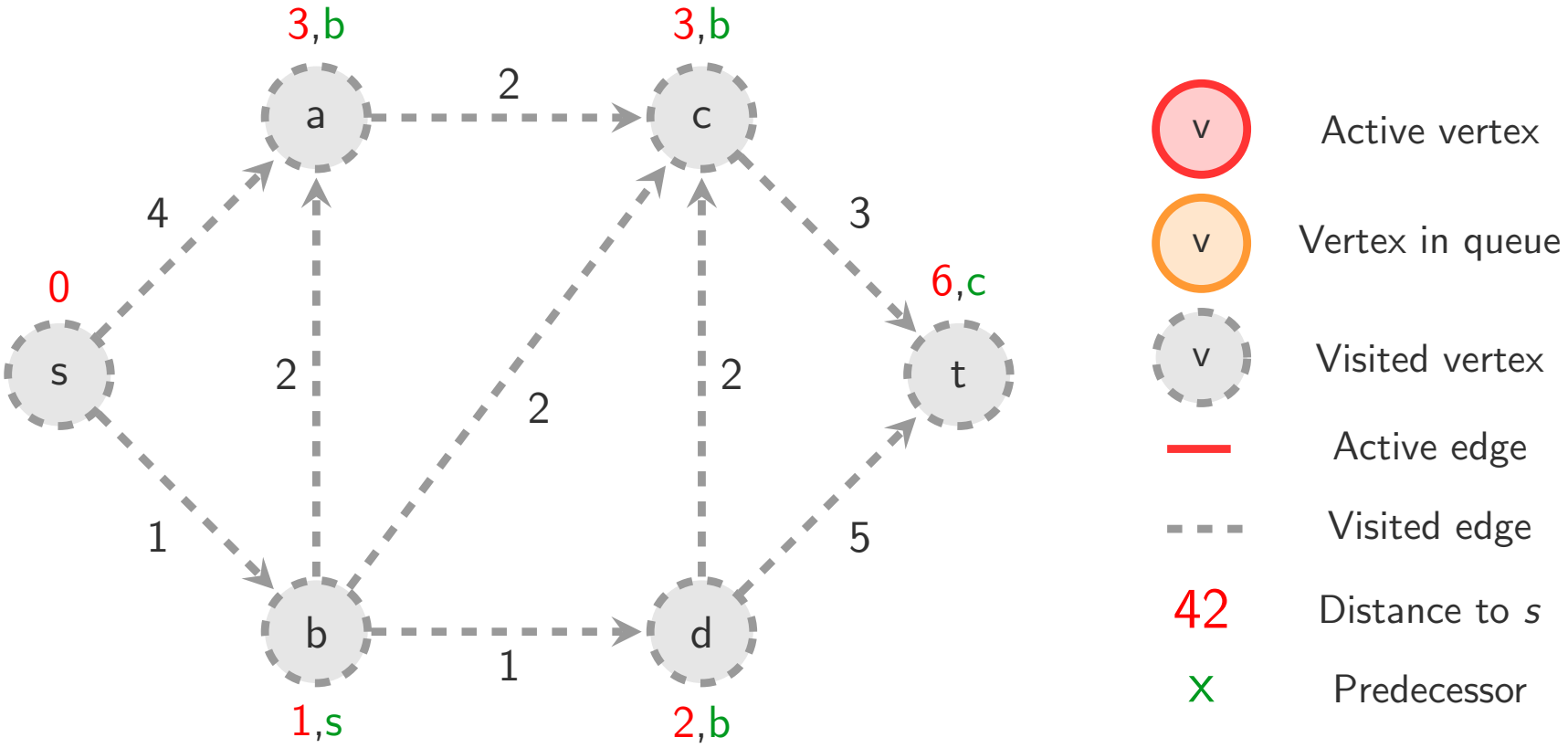
Dijkstra's Algorithm



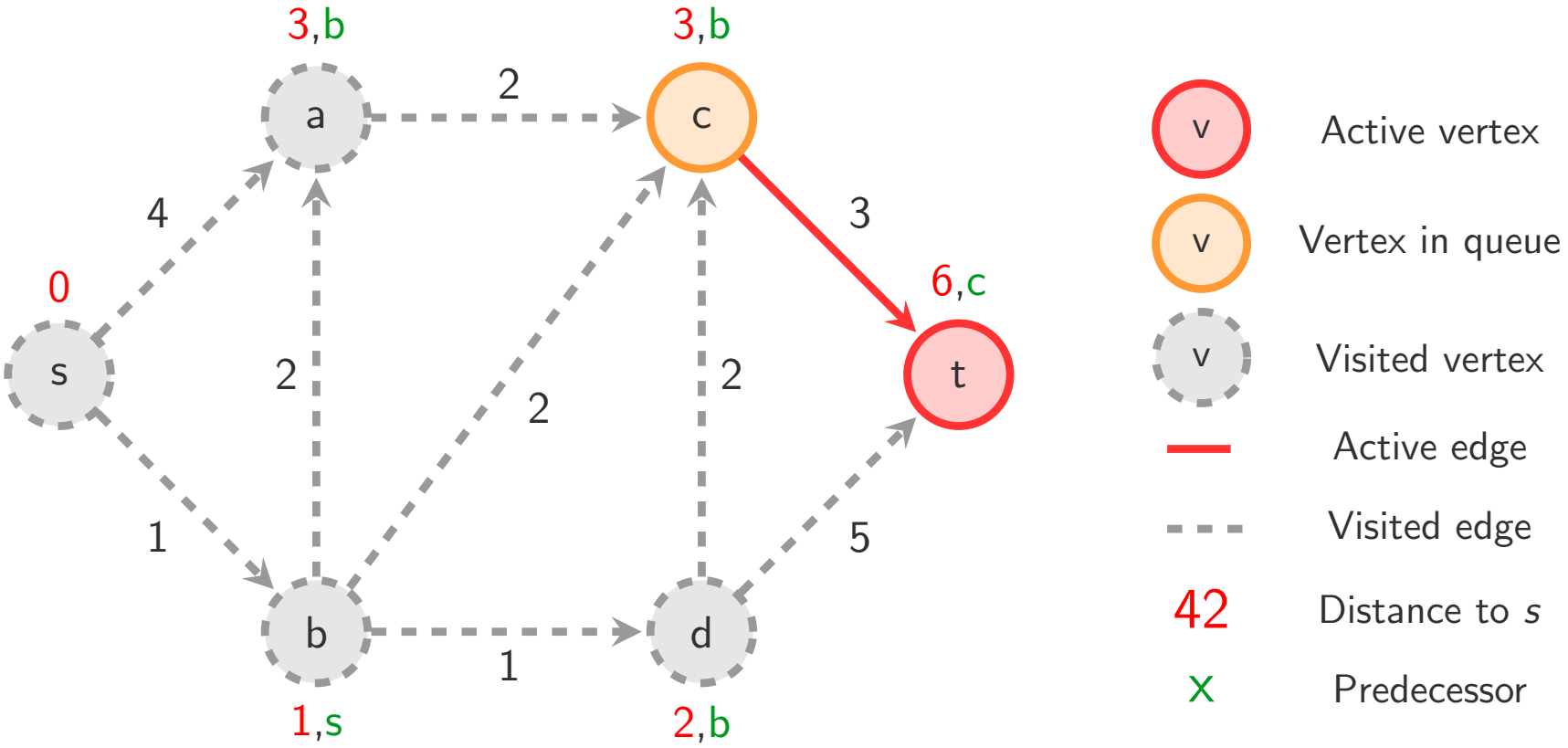
Dijkstra's Algorithm



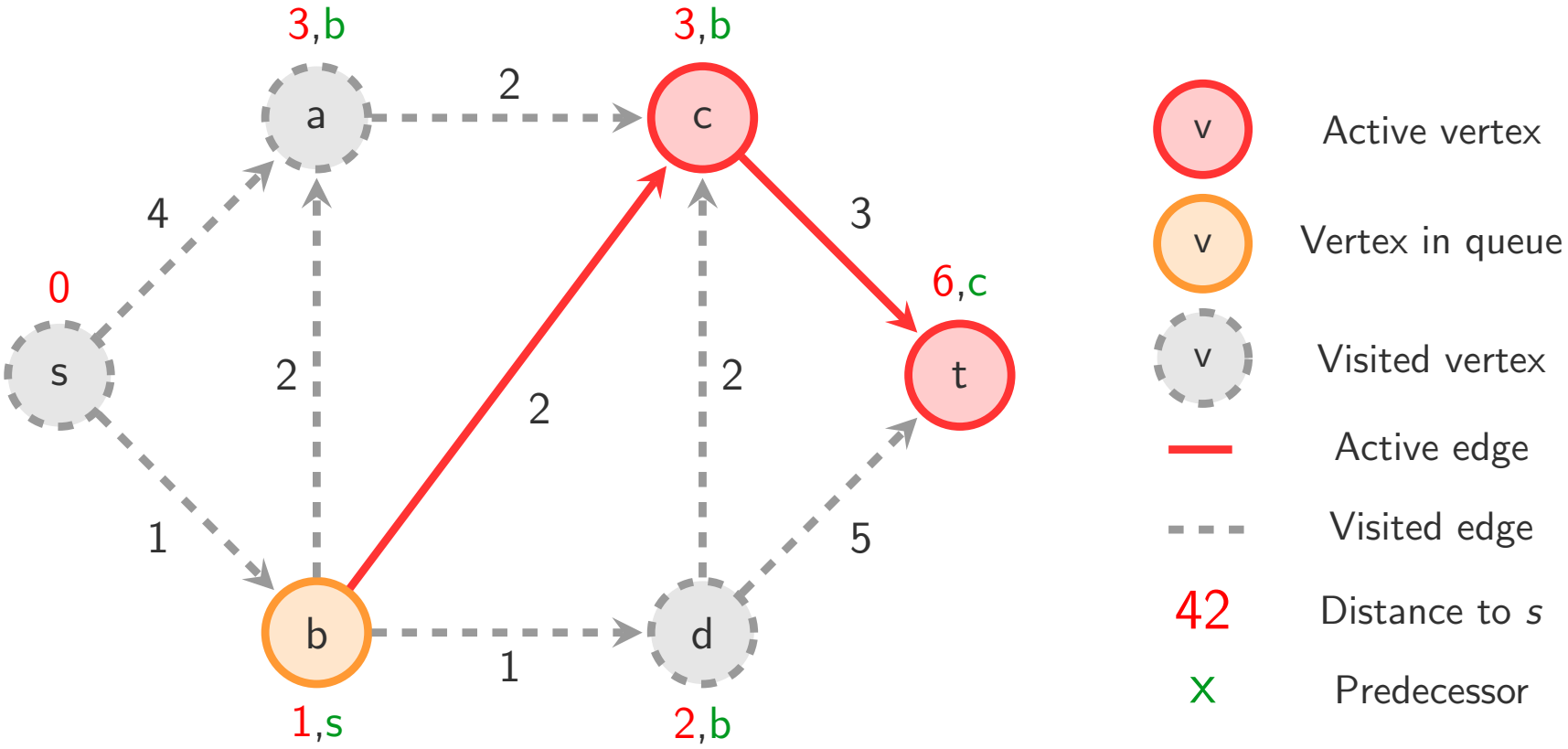
Dijkstra's Algorithm



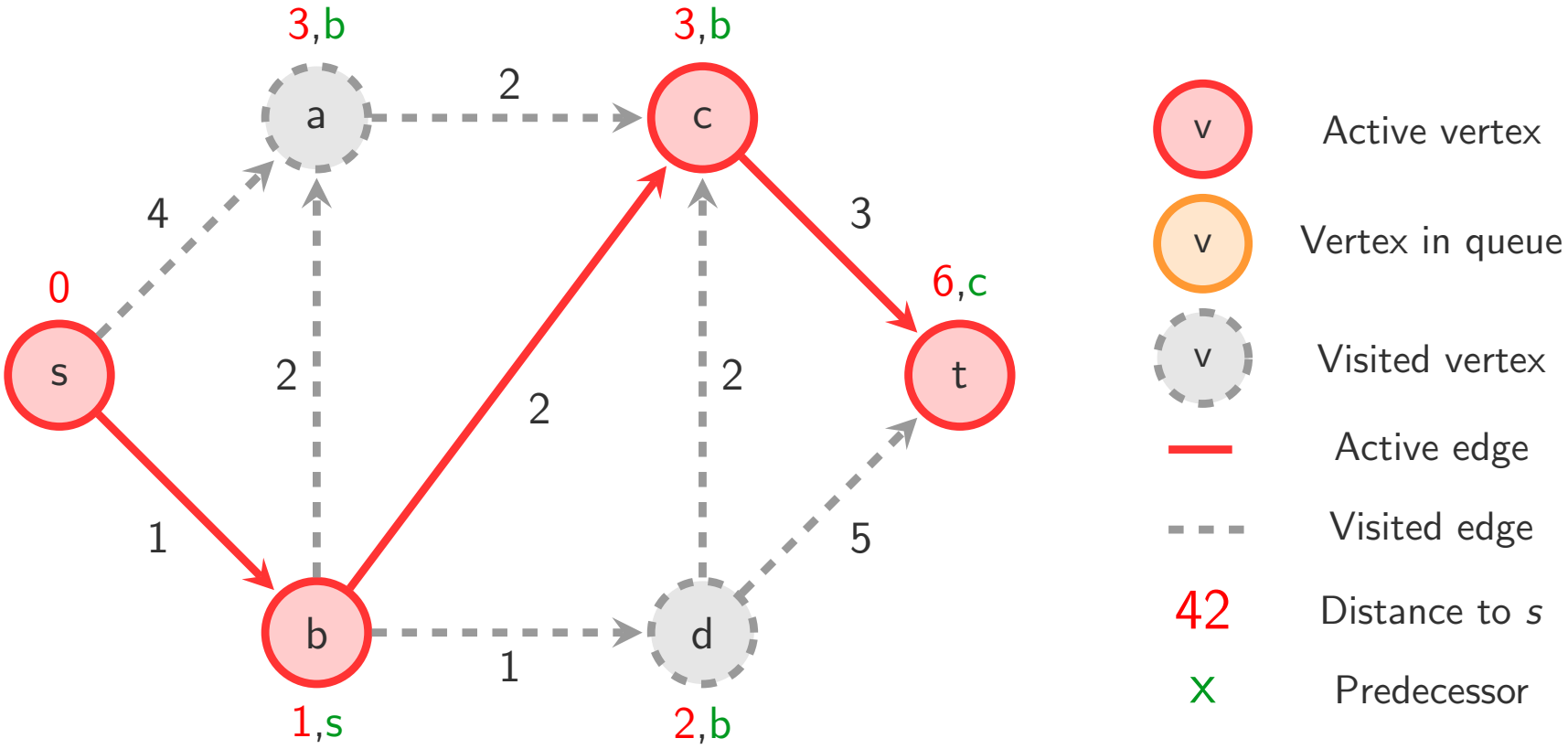
Shortest Path Tree



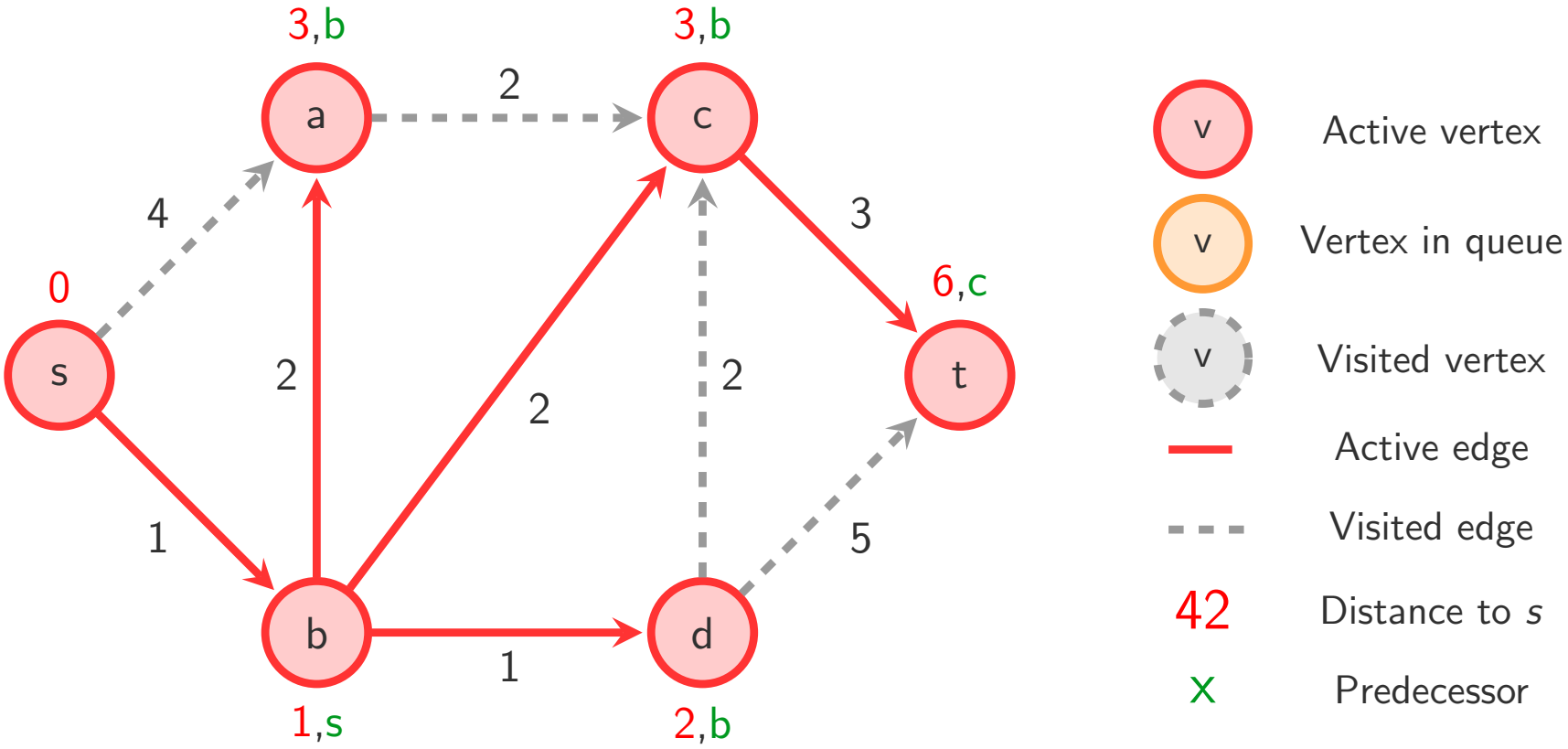
Shortest Path Tree



Shortest Path Tree



Shortest Path Tree



Algorithm 1 Dijkstra's Algorithm

Input: Graph $G = (V, E, c)$
  procedure DIJKSTRA(G, src)
    for each vertex $v \in V$ **do**
      dist[v] $\leftarrow \infty$, prev[v] $\leftarrow$ *null*, visited[v] $\leftarrow$ false
    end for
    dist[src] $\leftarrow 0$
    $PQ \leftarrow$ PriorityQueue over V
    for each vertex $v \in V$ **do**
      PQ.insert(v, dist[v])
    end for
    while PQ is not empty **do**
      $v \leftarrow PQ$.deleteMin()
      visited[v] $\leftarrow$ true
      for each neighbor w of v **do**
        if not visited[w] and dist[v] $+ c(v, w) <$ dist[w] **then**
          dist[w] $\leftarrow$ dist[v] $+ c(v, w)$
          PQ.decreaseKey(w, dist[w])
          prev[w] $\leftarrow v$
        end if
      end for
    end while
  end procedure

Analysis of Dijkstra's Algorithm

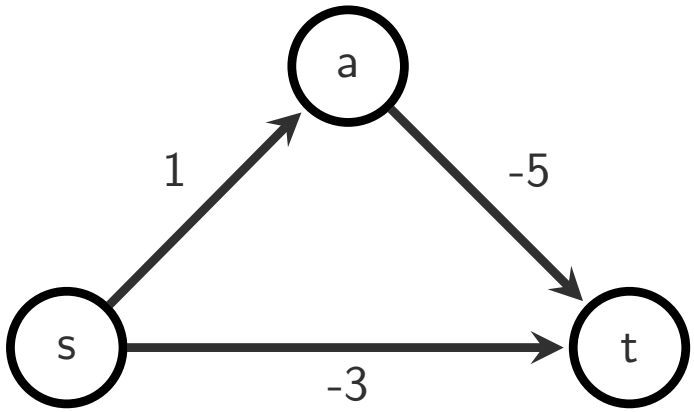
Running time

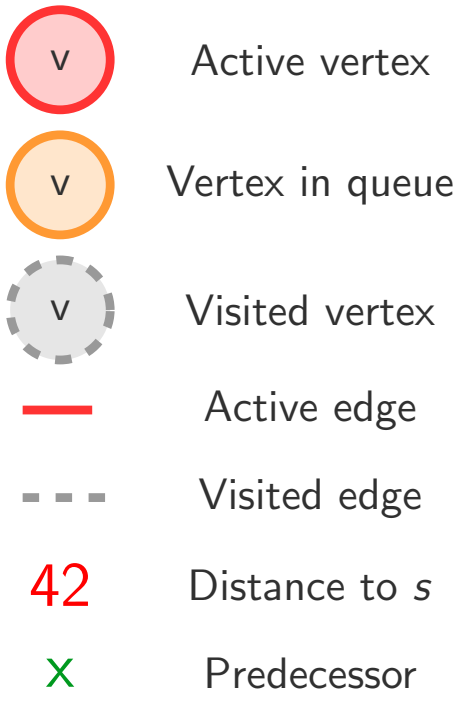
- With Fibonacci heap as priority queue:
- $|V|$ insert operations: $\mathcal{O}(|V|)$
- $|E|$ decreaseKey operations: $\mathcal{O}(|E|)$
- $|V|$ deleteMin operations: $\mathcal{O}(|V| \log |V|)$
- In total: $\mathcal{O}(|E| + |V| \log |V|)$

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

Limitations of Dijkstra's Algorithm

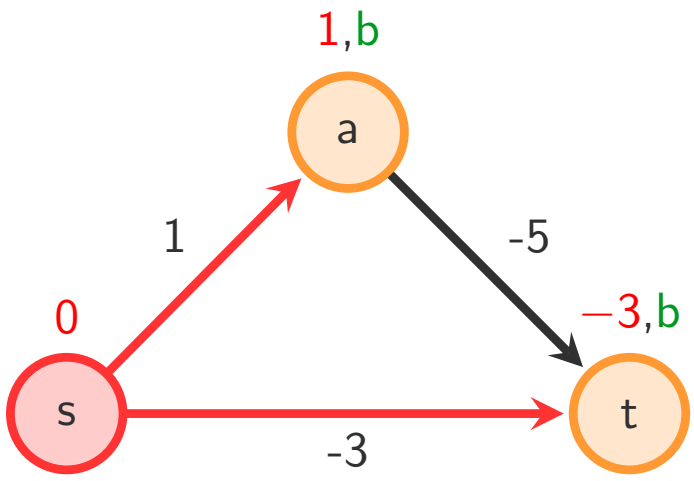
Dijkstra's Algorithm may not work for graphs with negative edge weights!

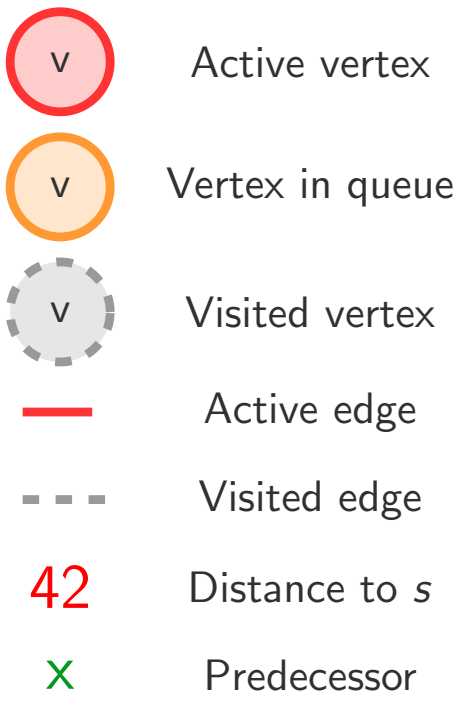




Limitations of Dijkstra's Algorithm

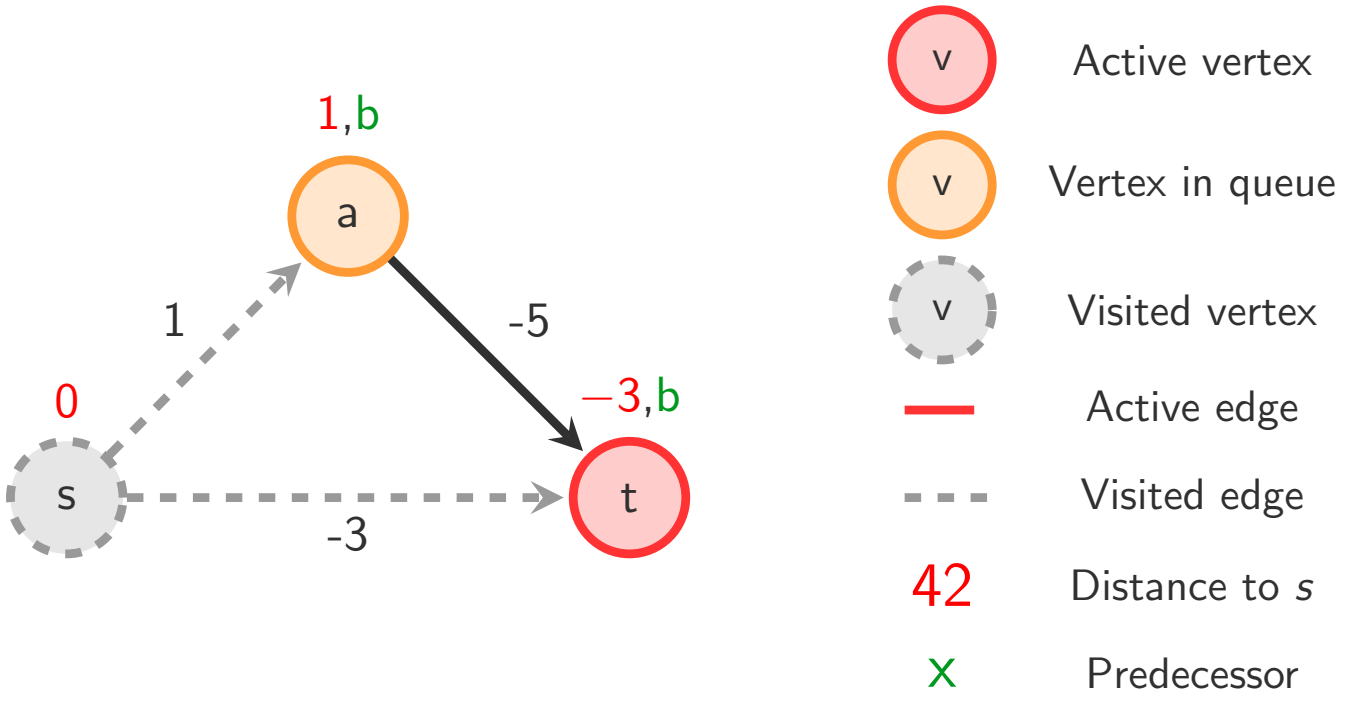
Dijkstra's Algorithm may not work for graphs with negative edge weights!





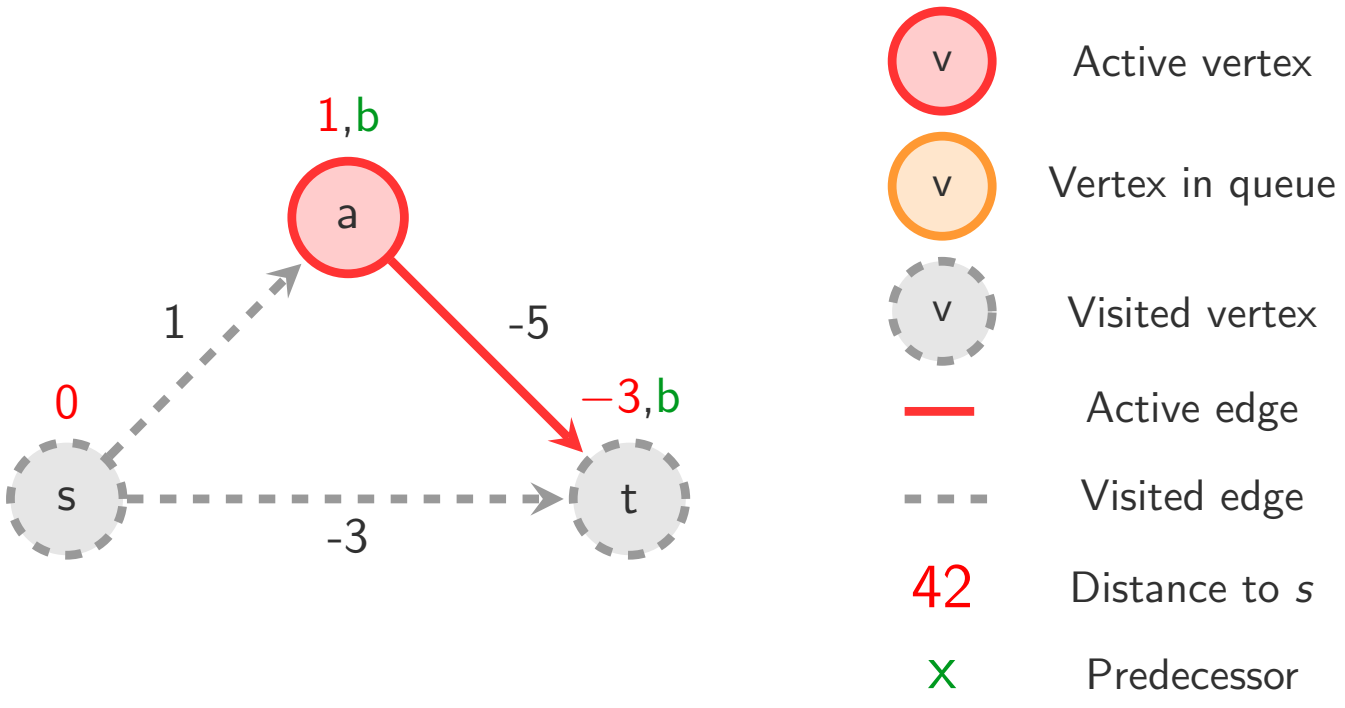
Limitations of Dijkstra's Algorithm

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



Limitations of Dijkstra's Algorithm

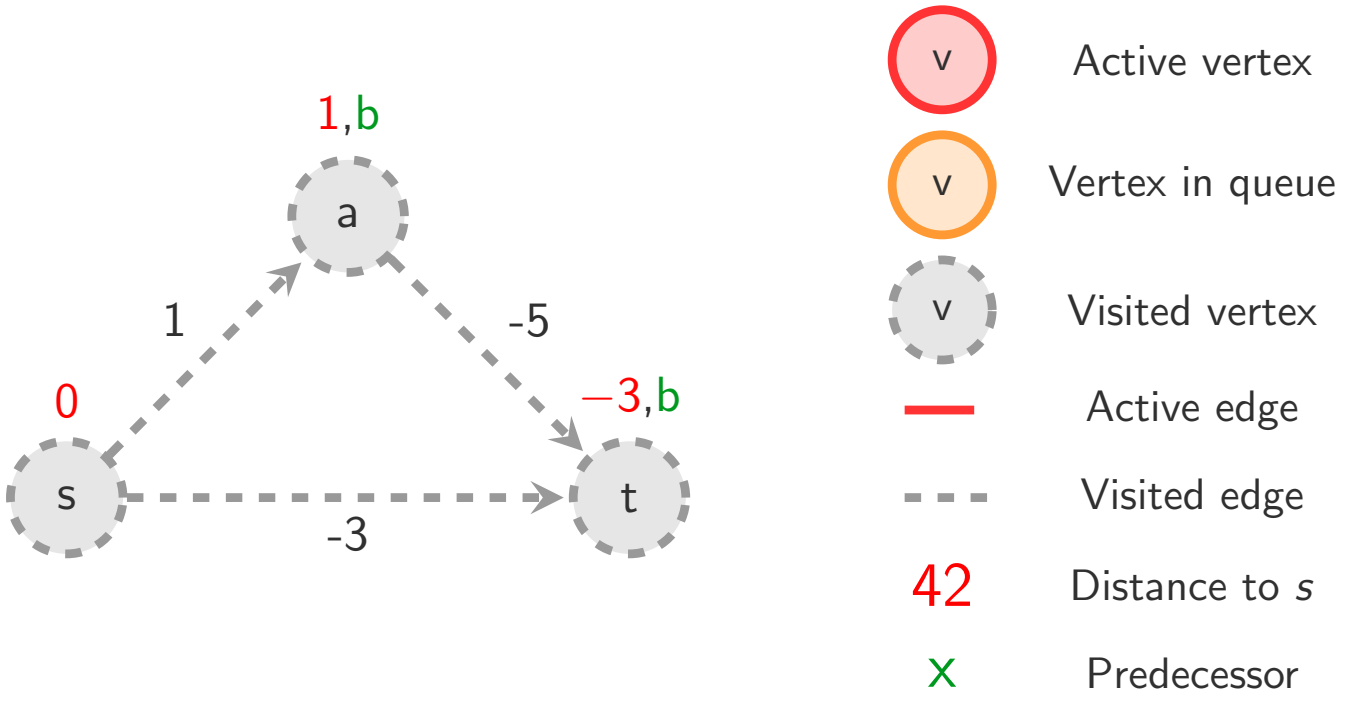
Dijkstra's Algorithm may not work for graphs with negative edge weights!



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.
- Basic Idea: "relax" all edges (i.e. update distances by considering each edge), repeat $|V| - 1$ times
- Some constant-factor optimizations possible

Bellman-Ford Algorithm

Algorithm 2 Bellman-Ford Algorithm (simplest version)

Input: Weighted 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
    for i = 1, 2, ..., |V| − 1 do
        for each (v, w) ∈ E do
            if dist[v] + c(v, w) < dist[w] then
                dist[w] ← dist[v] + c(v, w)
                prev[w] ← v
            end if
        end for
    end for
end procedure
```

Bellman-Ford Algorithm: Correctness

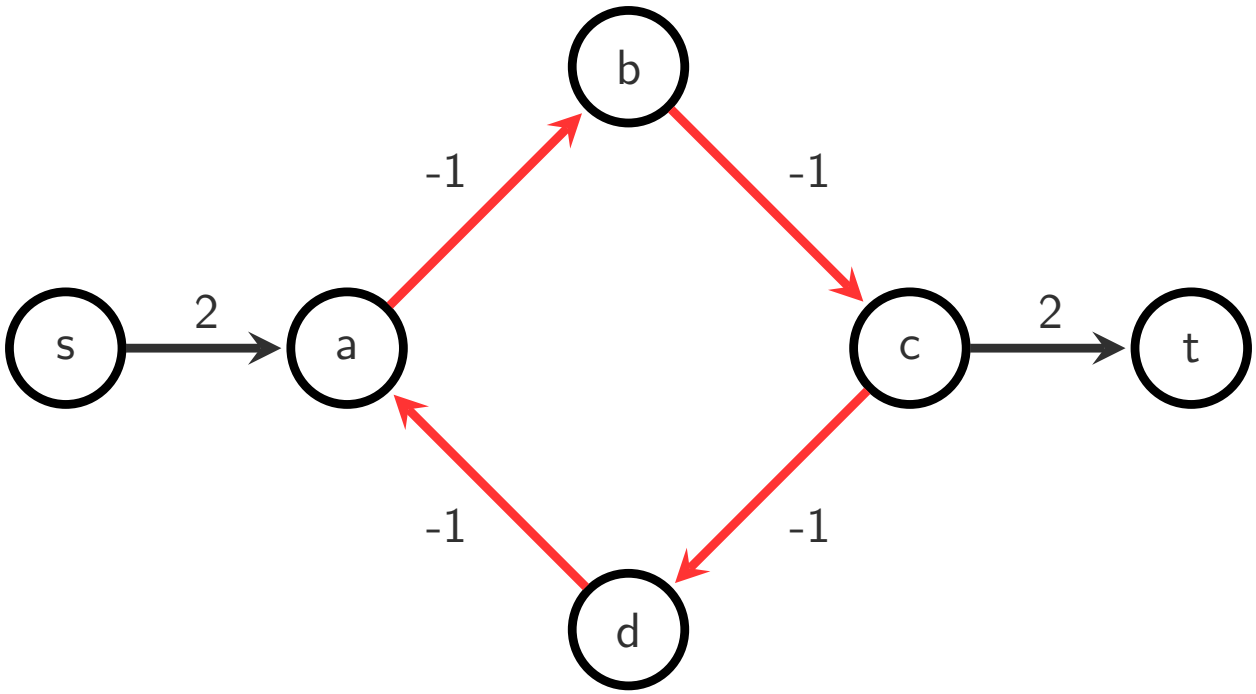
Lemma

Let $v \in V$ s.t. there exists a shortest path from src to v containing at most i edges. Then, after the i-th iteration of Bellman-Ford, dist[v] *is the length of this path.*

- *Note:* If there are no negative cycles, for every v there exists a shortest path from *src* to v containing at most $|V| - 1$ edges.
- But what if there are negative cycles?
- Idea: do another iteration. If there are still updates, there are negative cycles.

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.



Bellman-Ford Algorithm

Algorithm 3 Bellman-Ford Algorithm (simplest version with negative cycle detection)

Input: Weighted 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
    for i = 1, 2, ..., |V| − 1 do
        for each (v, w) ∈ E do
            if dist[v] + c(v, w) < dist[w] then
                dist[w] ← dist[v] + c(v, w)
                prev[w] ← v
            end if
        end for
    end for
    for each (v, w) ∈ E do
        if dist[v] + c(v, w) < dist[w] then
            return Negative Cycle detected
        end if
    end for
end procedure
```

Bellman-Ford Algorithm: Remarks

Negative Cycles:

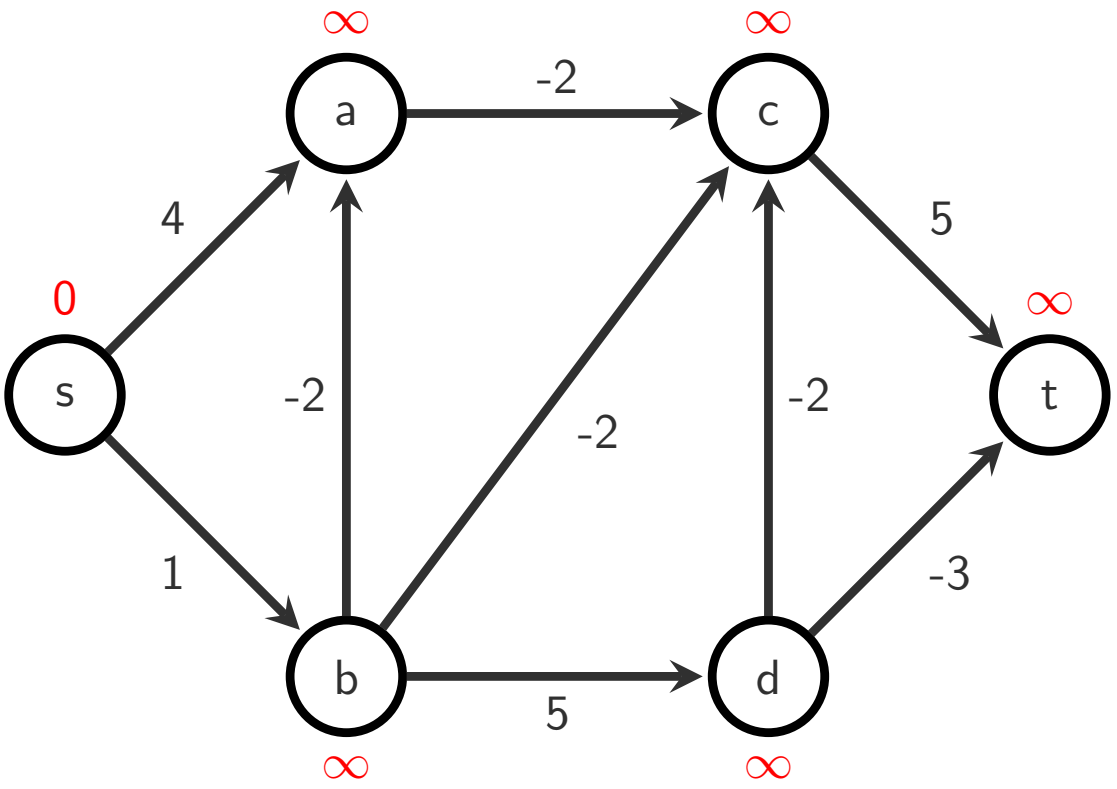
- If one wants to find which nodes are reachable through some negative cycle, don't return, but update dist[w] to $-\infty$. Then, propagate, e.g. by iterating another $|V| - 1$ times.
- Keeping paths to negative cycles also possible without changing complexity, but makes algorithm more complicated.

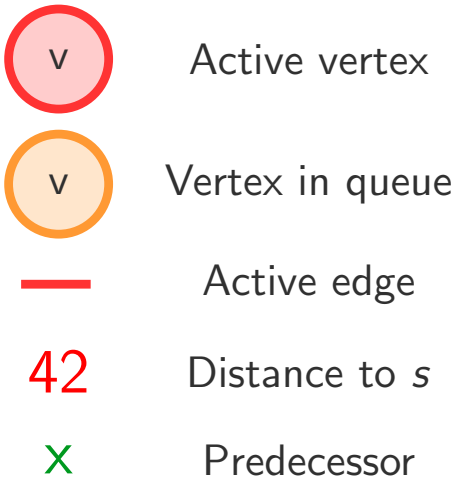
Constant-factor optimizations:

- Break loop once no edge relaxation changes anything
- Do not consider edges where the source vertex has not changed its distance since last iteration $\rightarrow$ use queue Q to store vertices whose distance has changed

Bellman-Ford Algorithm

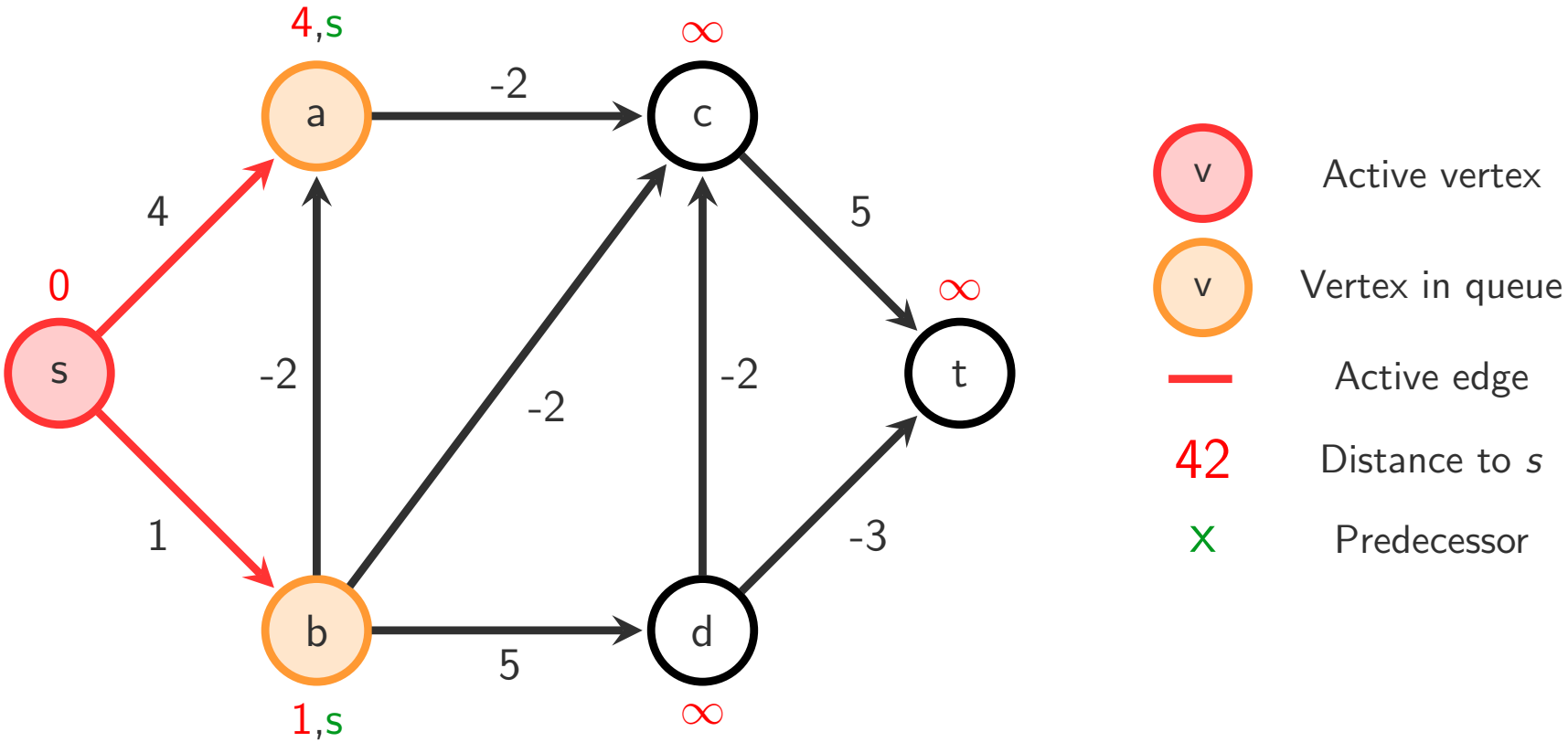
$Q = [s]$





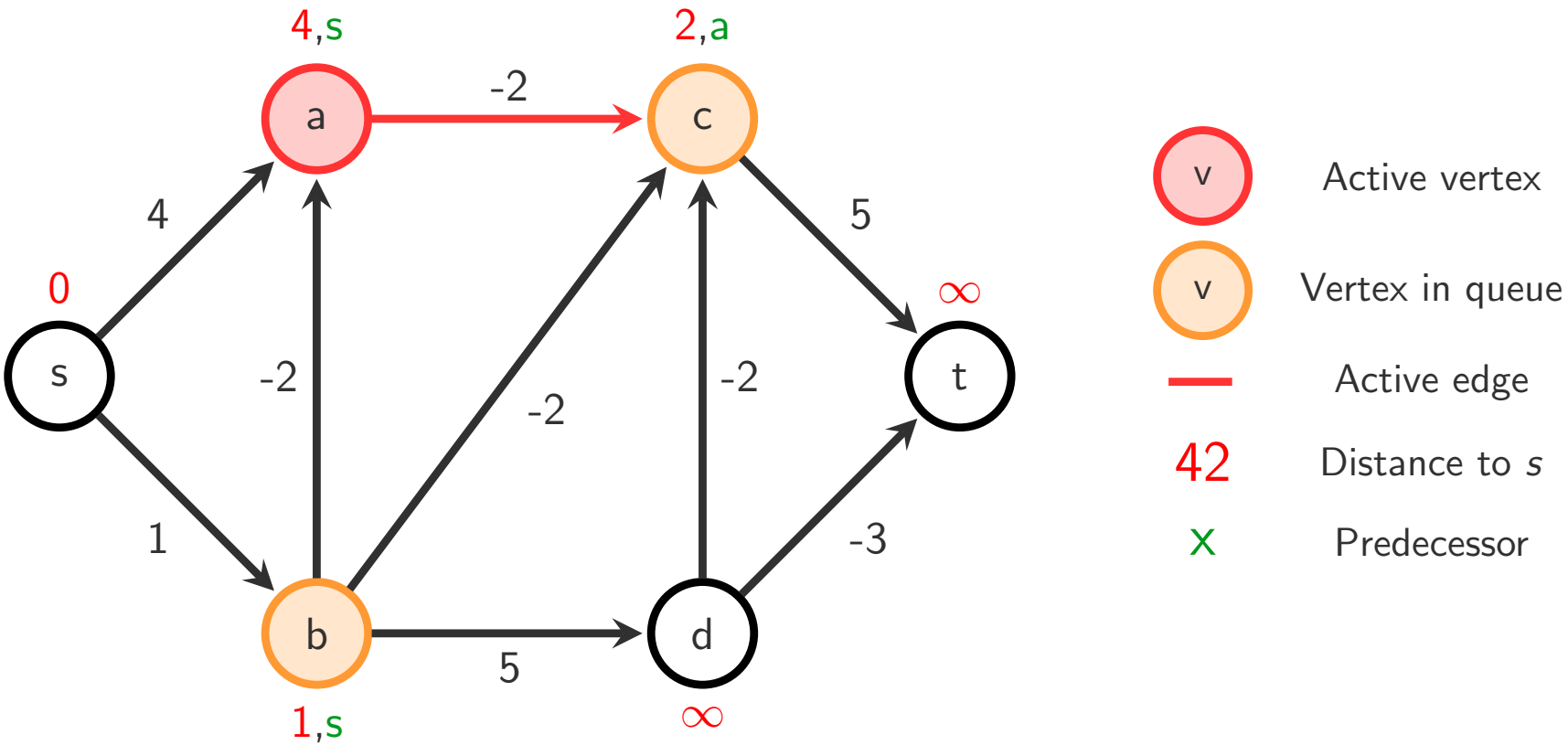
Bellman-Ford Algorithm

$Q = [a, b]$



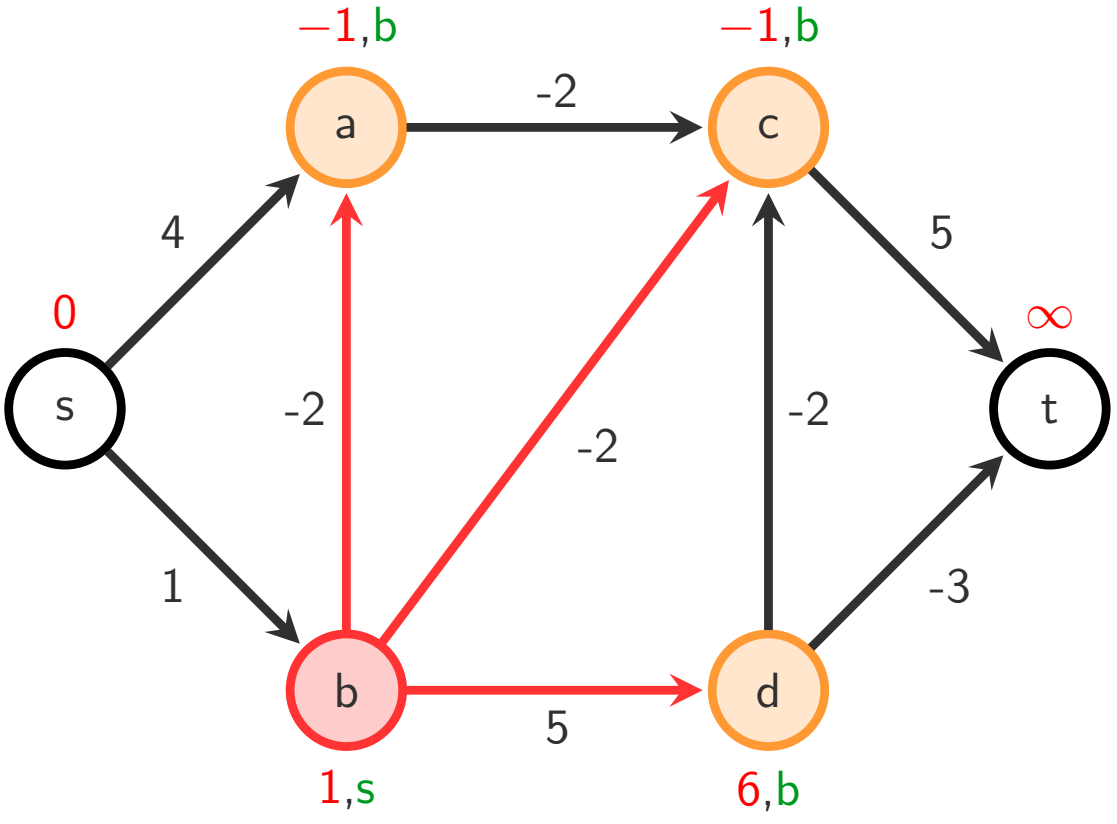
Bellman-Ford Algorithm

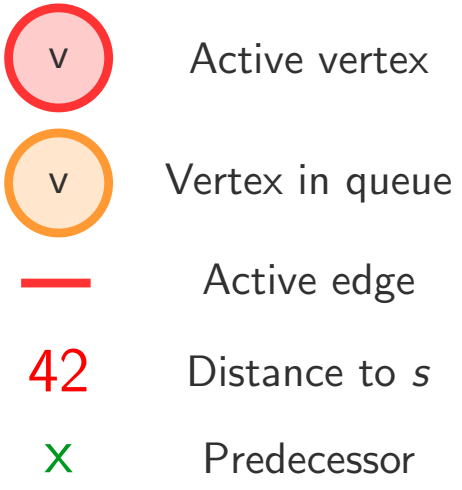
$Q = [b, c]$



Bellman-Ford Algorithm

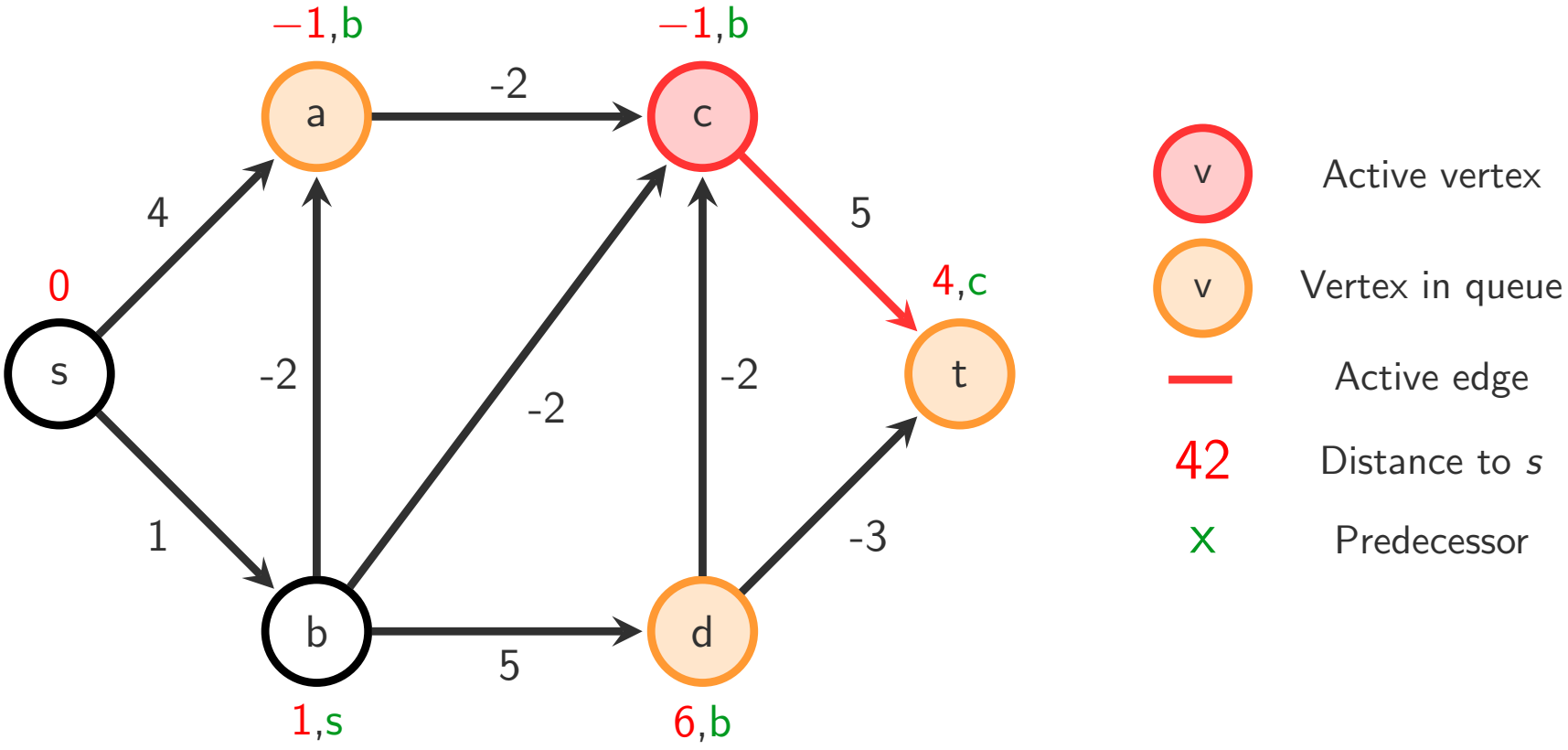
$Q = [c, a, d]$





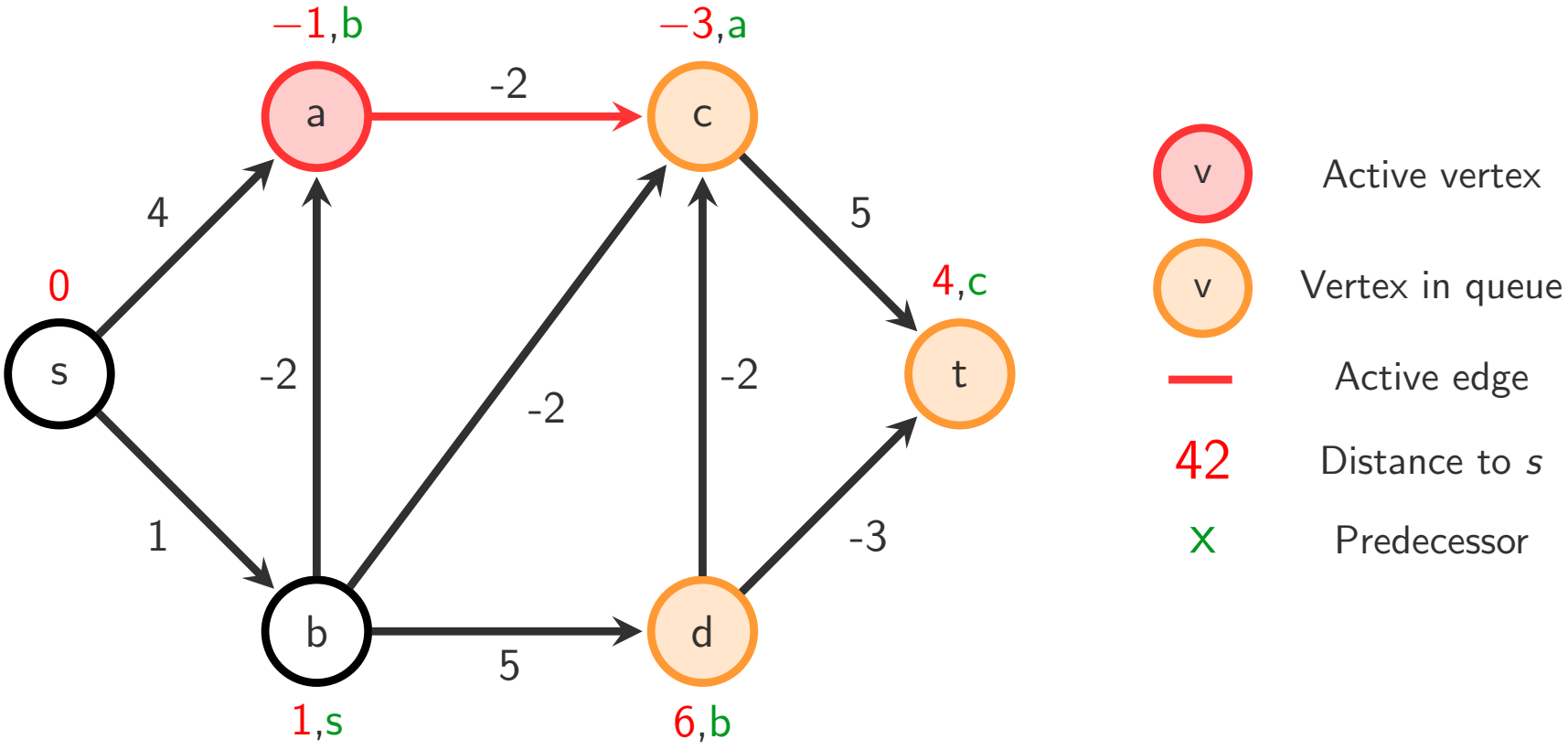
Bellman-Ford Algorithm

$Q = [a, d, t]$



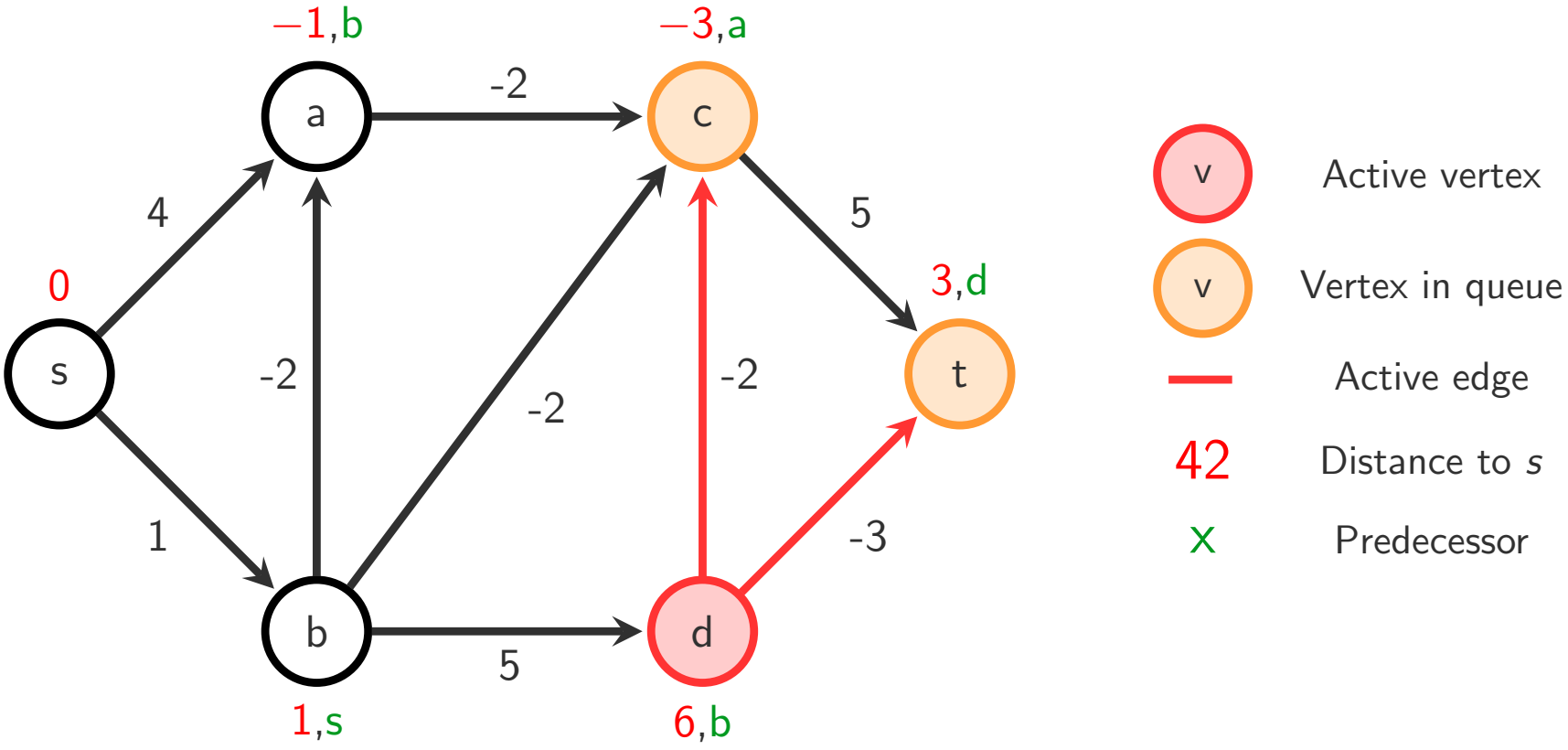
Bellman-Ford Algorithm

$Q = [d, t, c]$



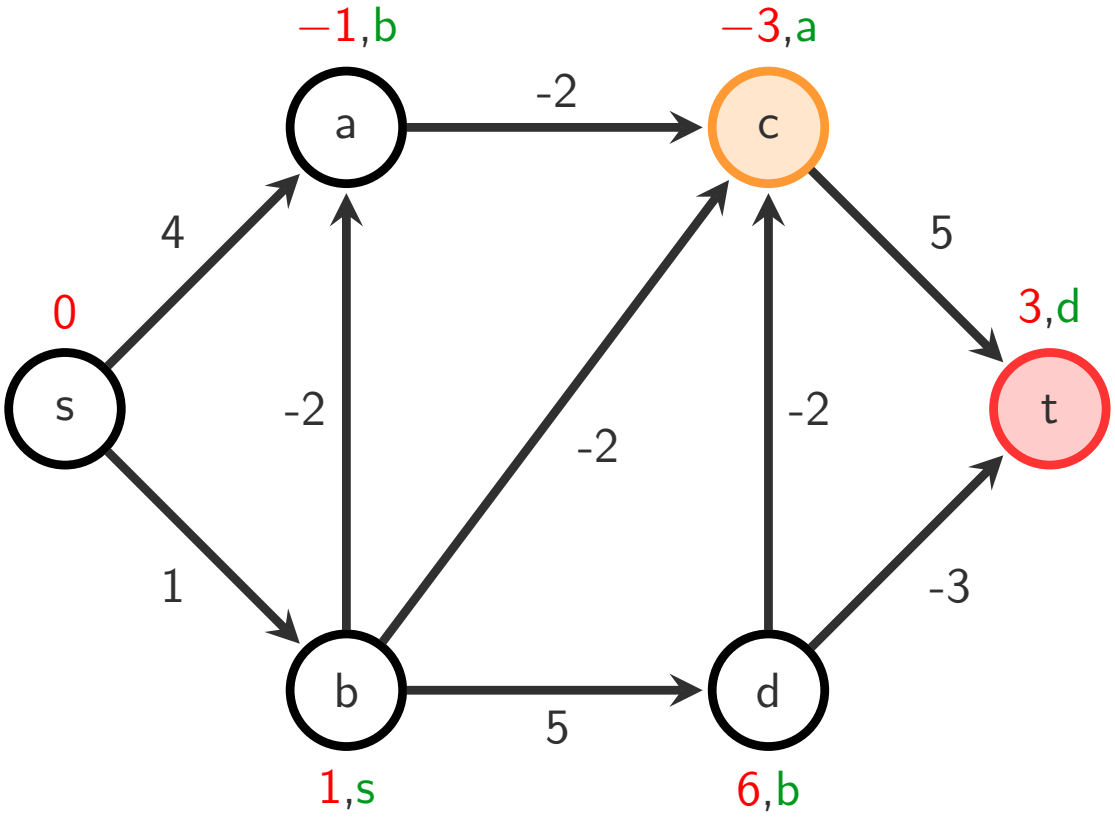
Bellman-Ford Algorithm

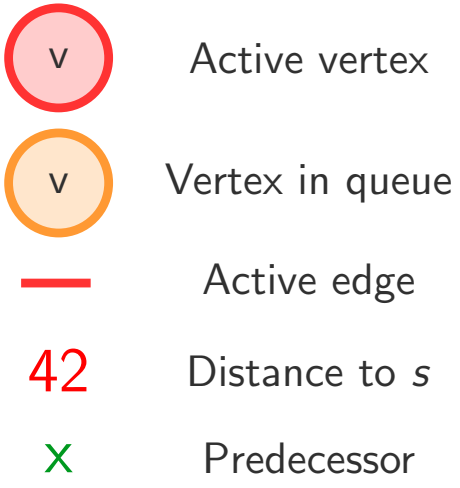
$Q = [t, c]$



Bellman-Ford Algorithm

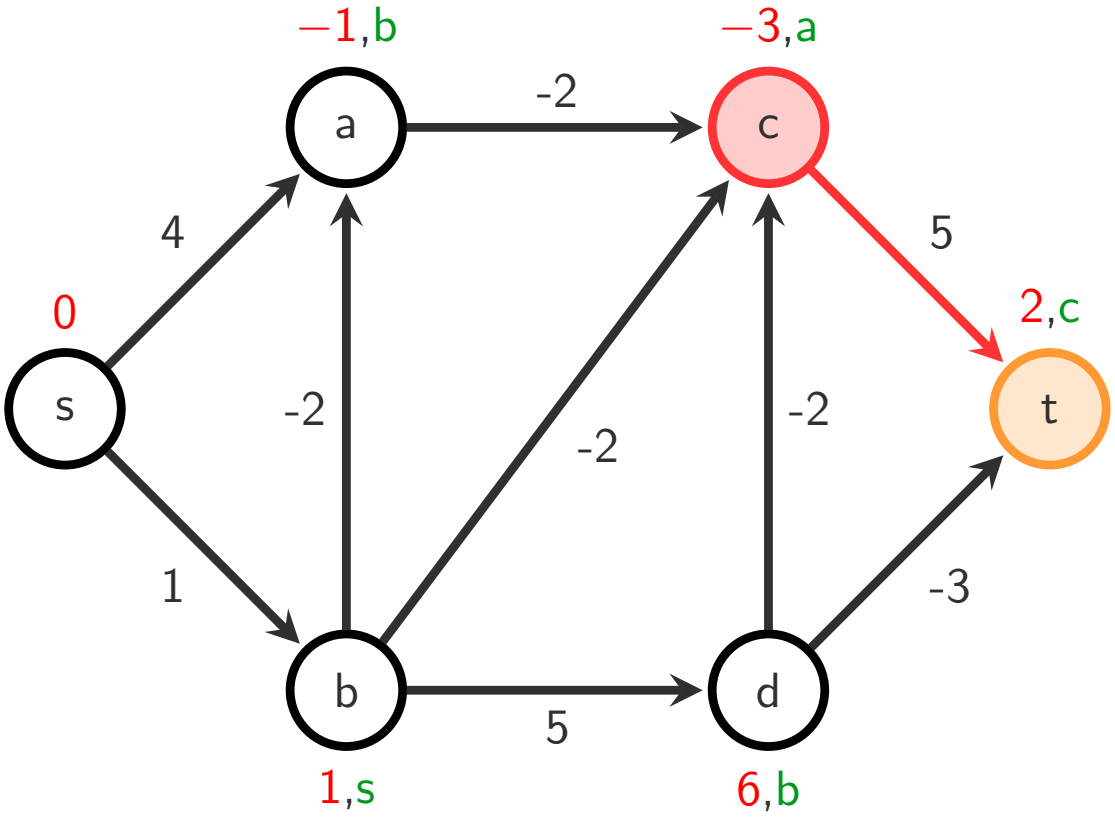
$Q = [c]$

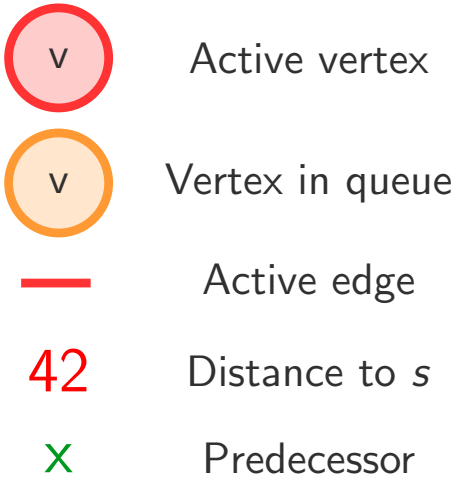




Bellman-Ford Algorithm

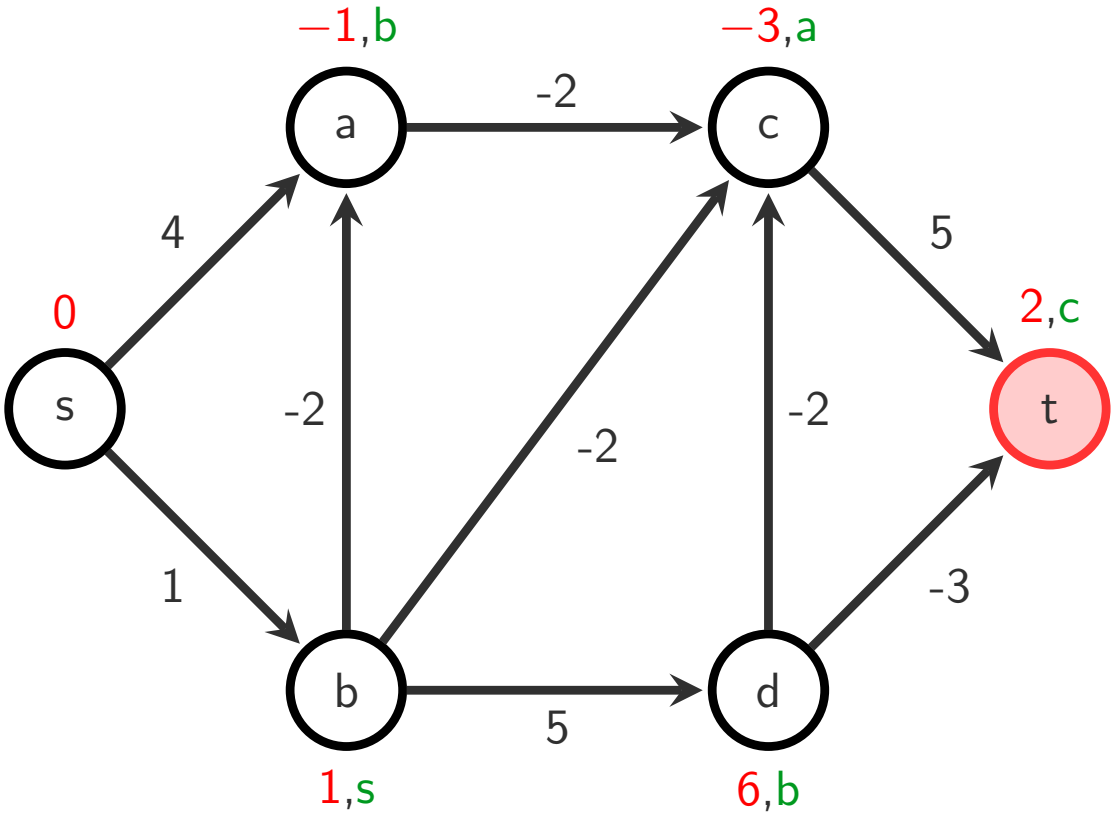
$Q = [t]$

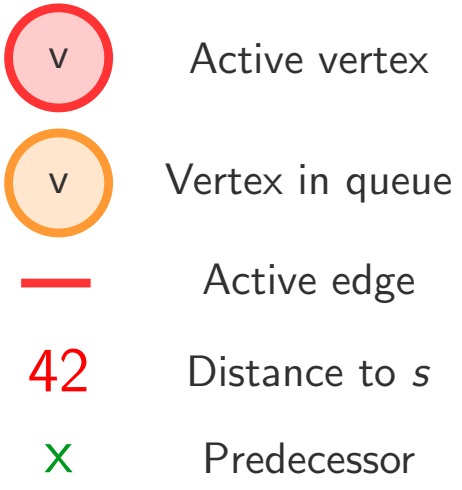




Bellman-Ford Algorithm

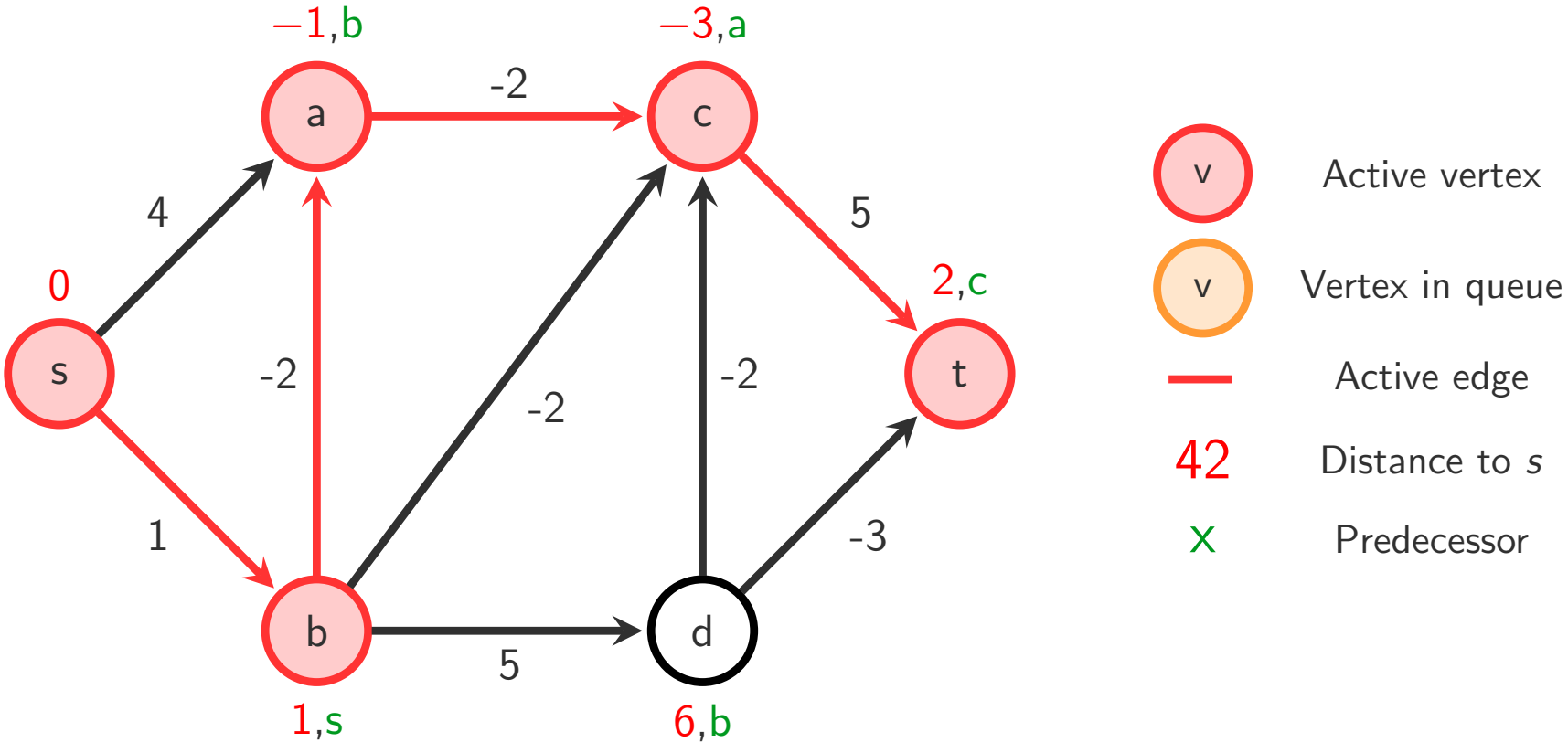
$Q = []$





Bellman-Ford Algorithm

$Q = []$



Algorithm 4 Bellman-Ford Algorithm (improved, no negative cycles)

Input: Weighted 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 Cycle Detection

- Idea: Process FIFO-Queue in phases.
- One phase = processing all nodes currently in the queue.
- After phase i, all shortest paths using at most i edges were detected.
- If there are nodes left in the queue after phase $|V|$, 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 $|V|$.

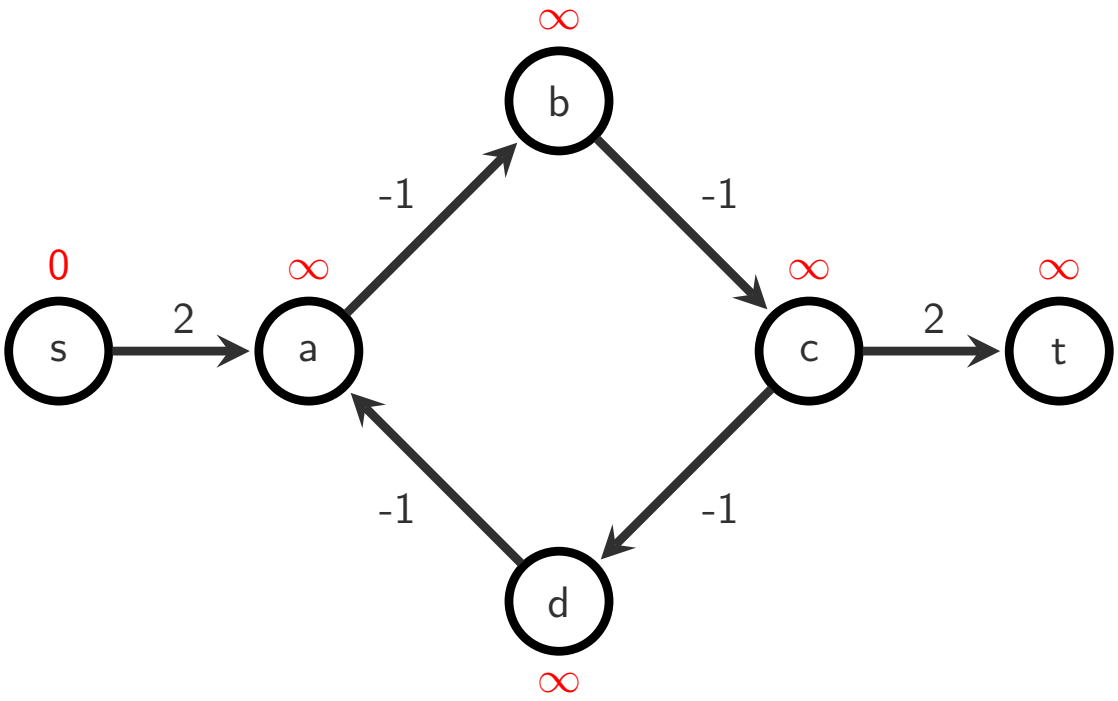
Algorithm 5 Bellman-Ford Algorithm (improved, negative cycle detection)

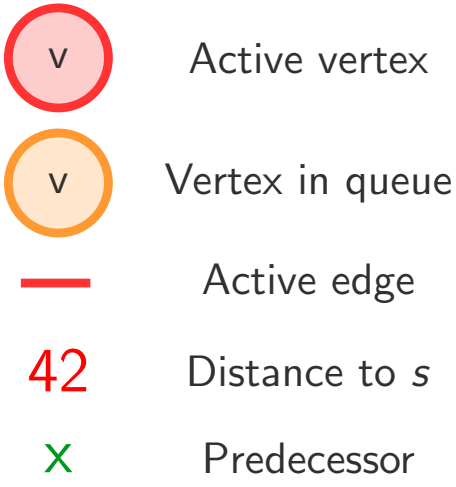
Input: Graph $G = (V, E, c)$

```
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, 2, ..., |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

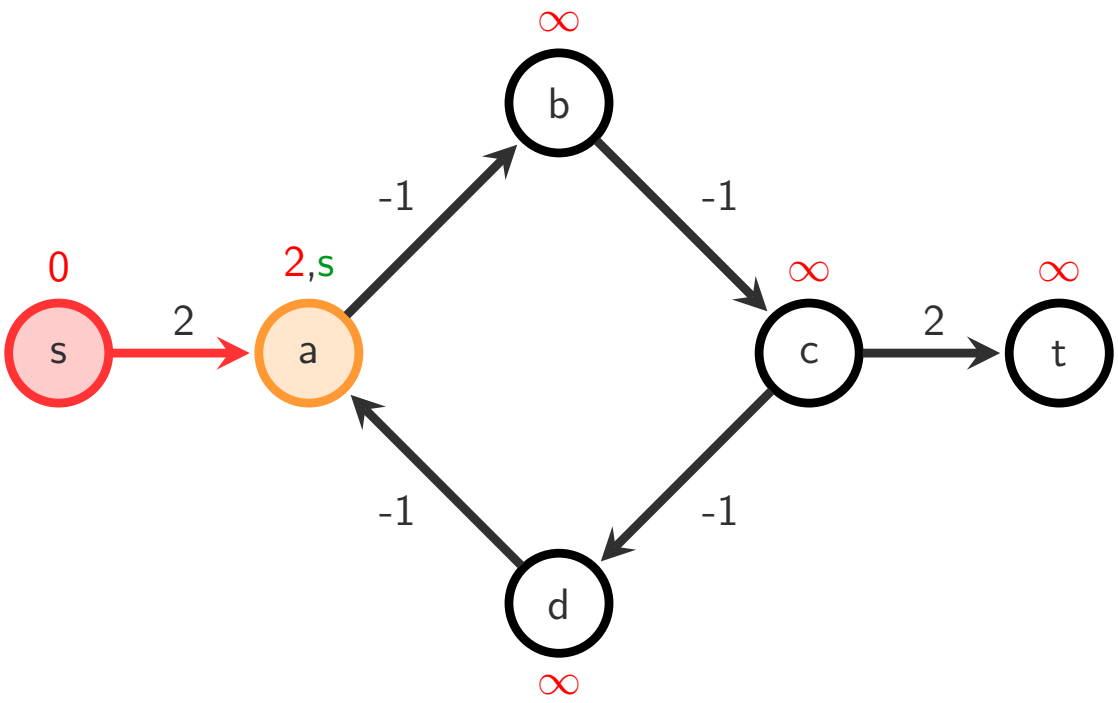
Initialization

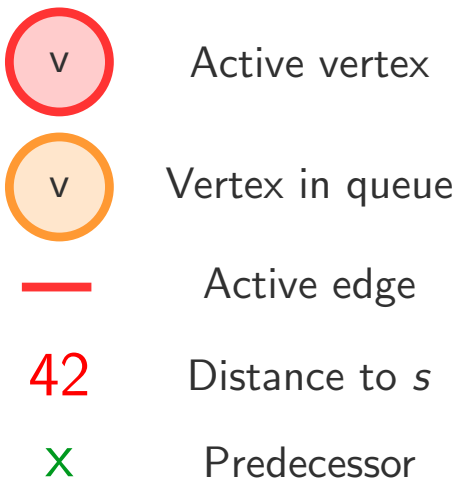




Bellman-Ford Algorithm with Negative Cycles

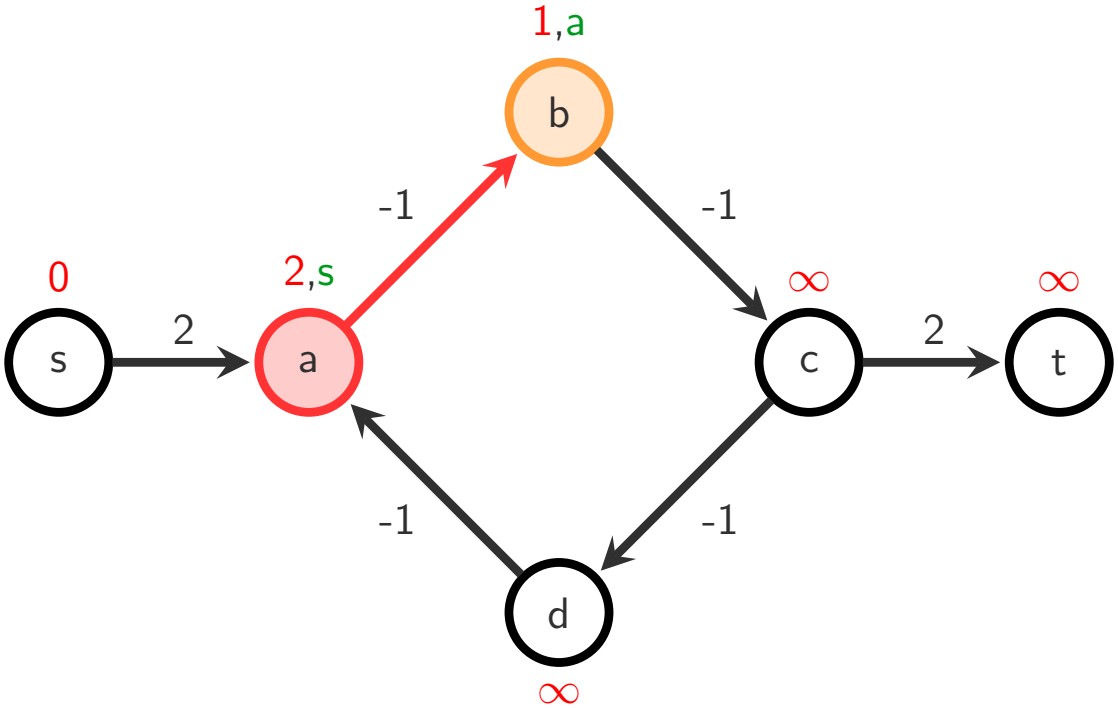
Phase 1: $Q = (s) \longrightarrow Q' = (a)$

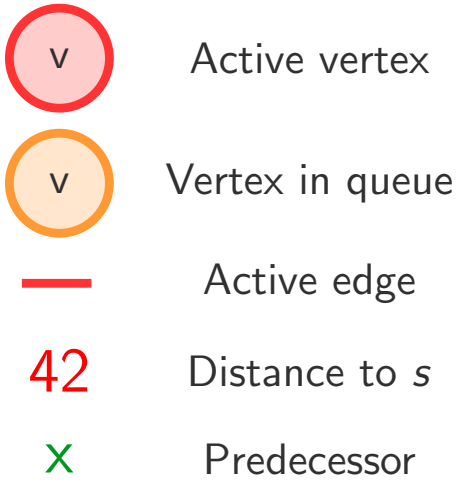




Bellman-Ford Algorithm with Negative Cycles

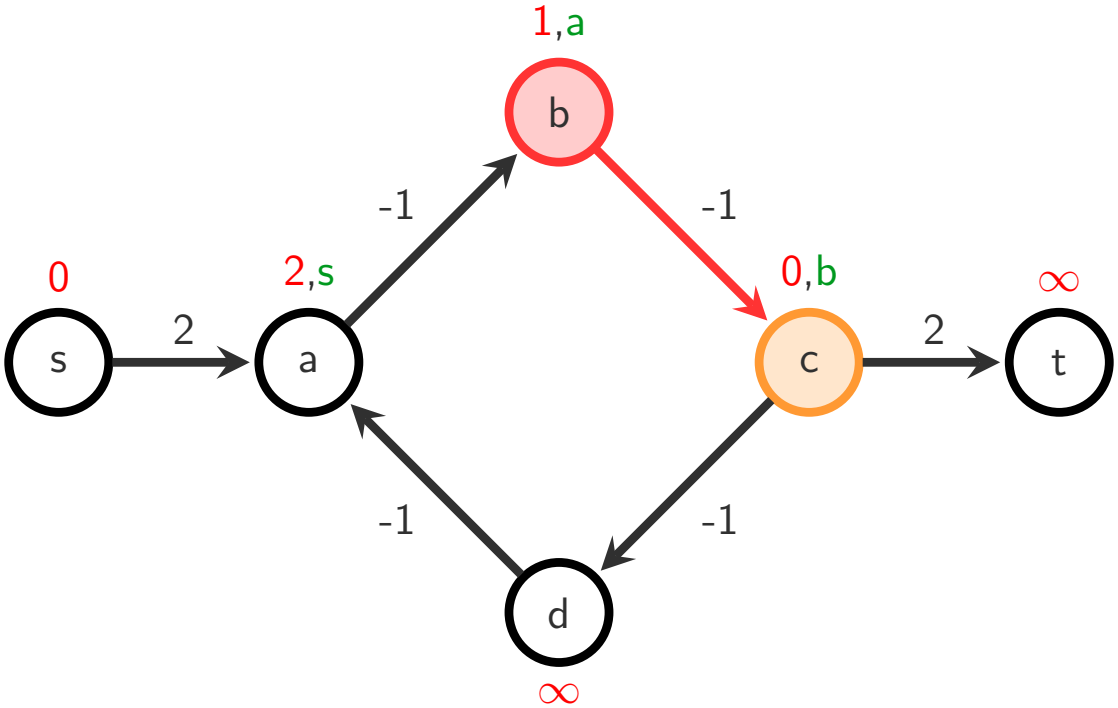
Phase 2: $Q = (a) \longrightarrow Q' = (b)$

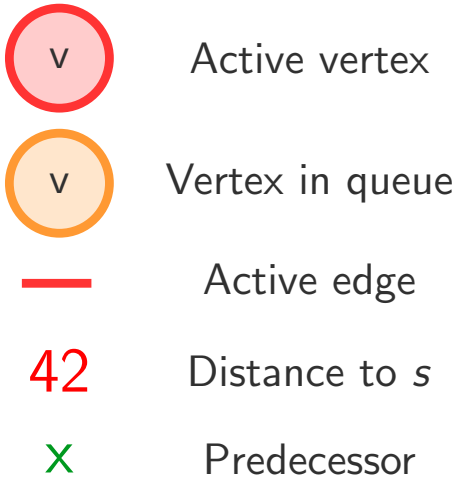




Bellman-Ford Algorithm with Negative Cycles

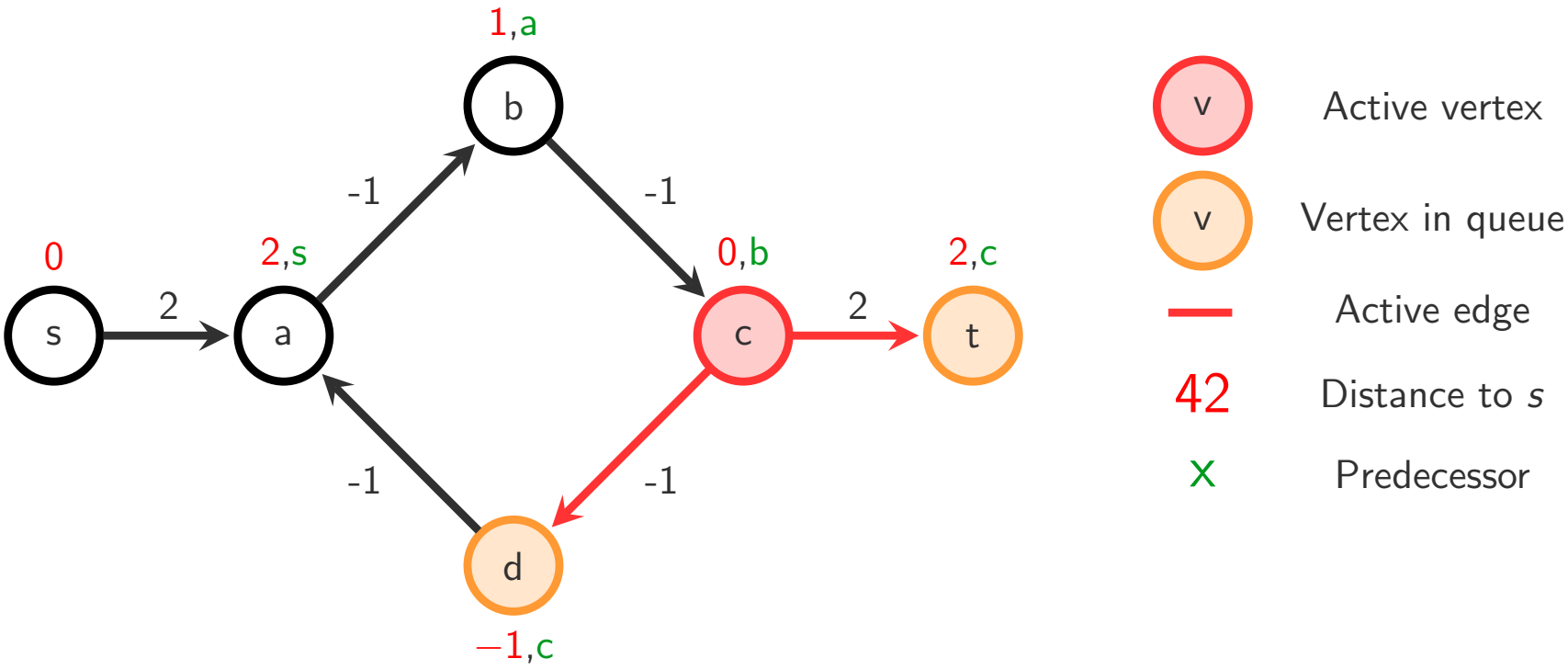
Phase 3: $Q = (b) \longrightarrow Q' = (c)$





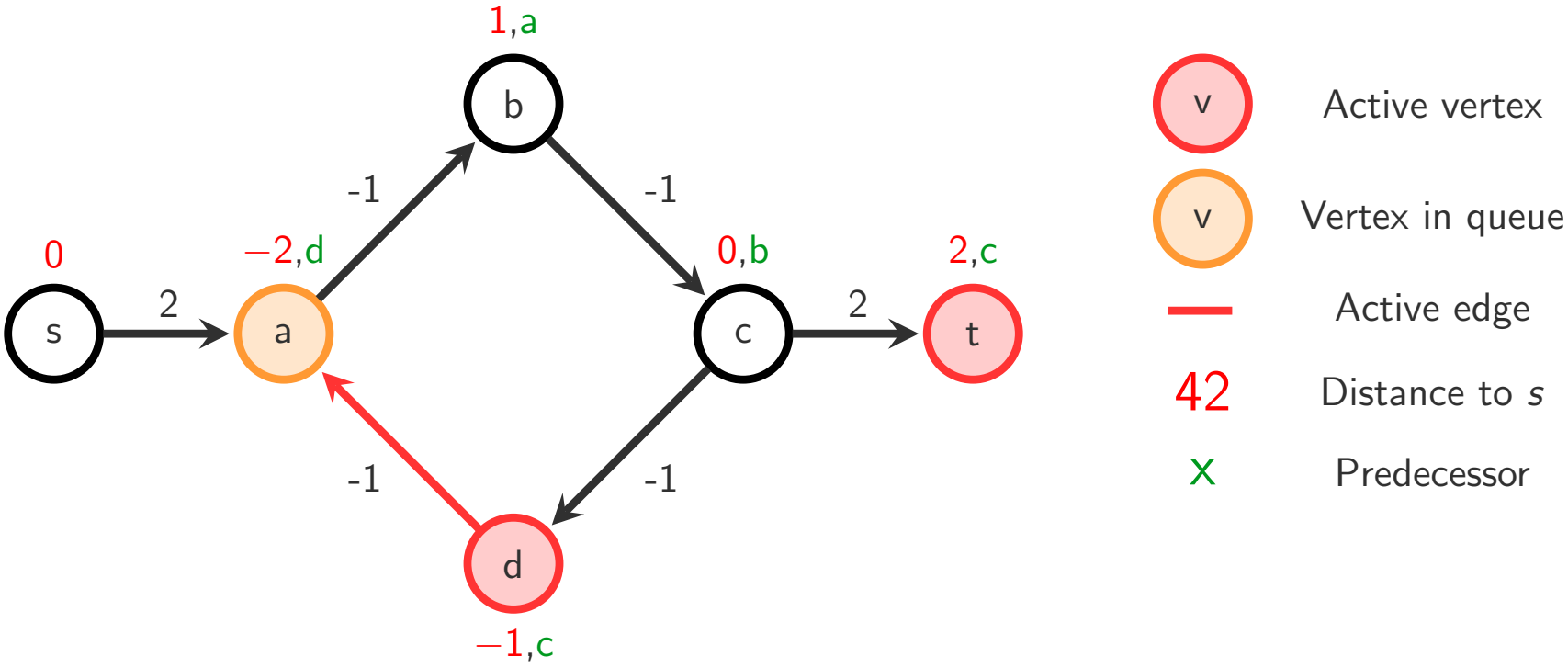
Bellman-Ford Algorithm with Negative Cycles

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



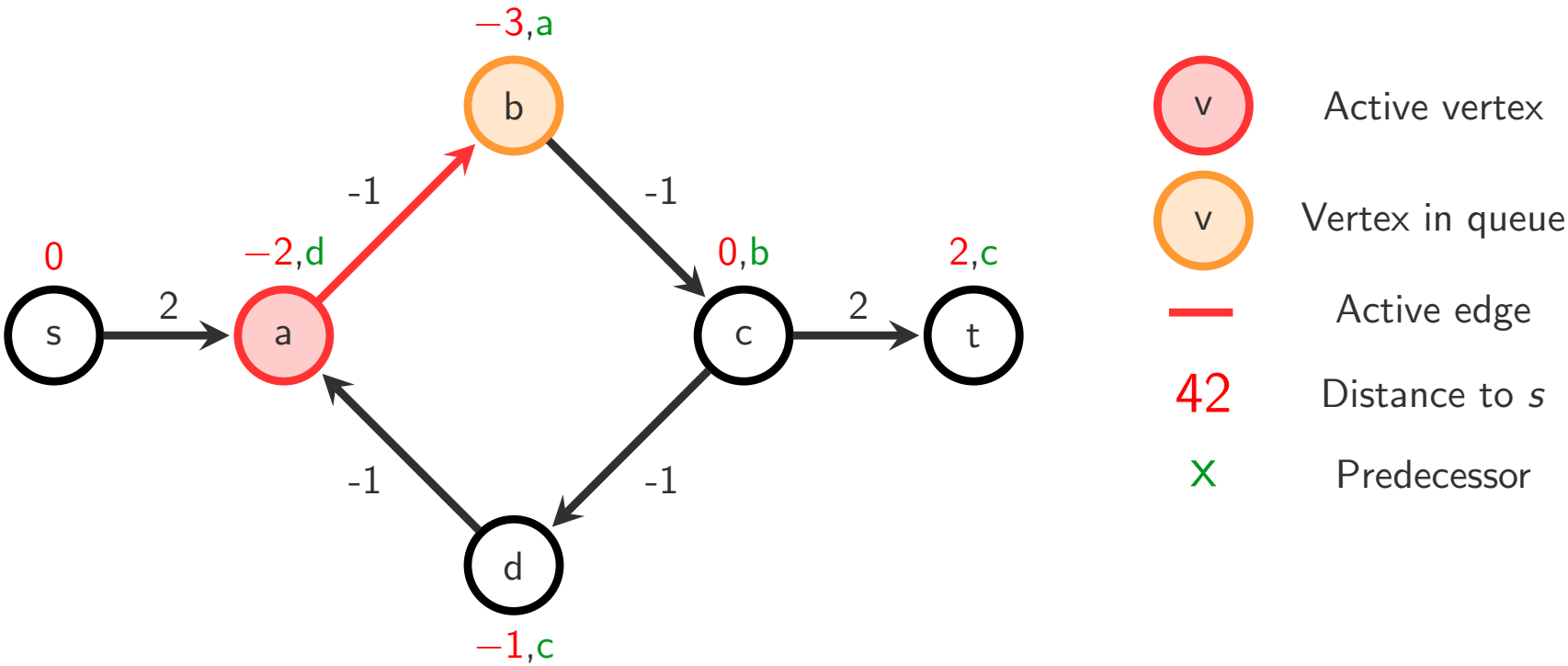
Bellman-Ford Algorithm with Negative Cycles

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



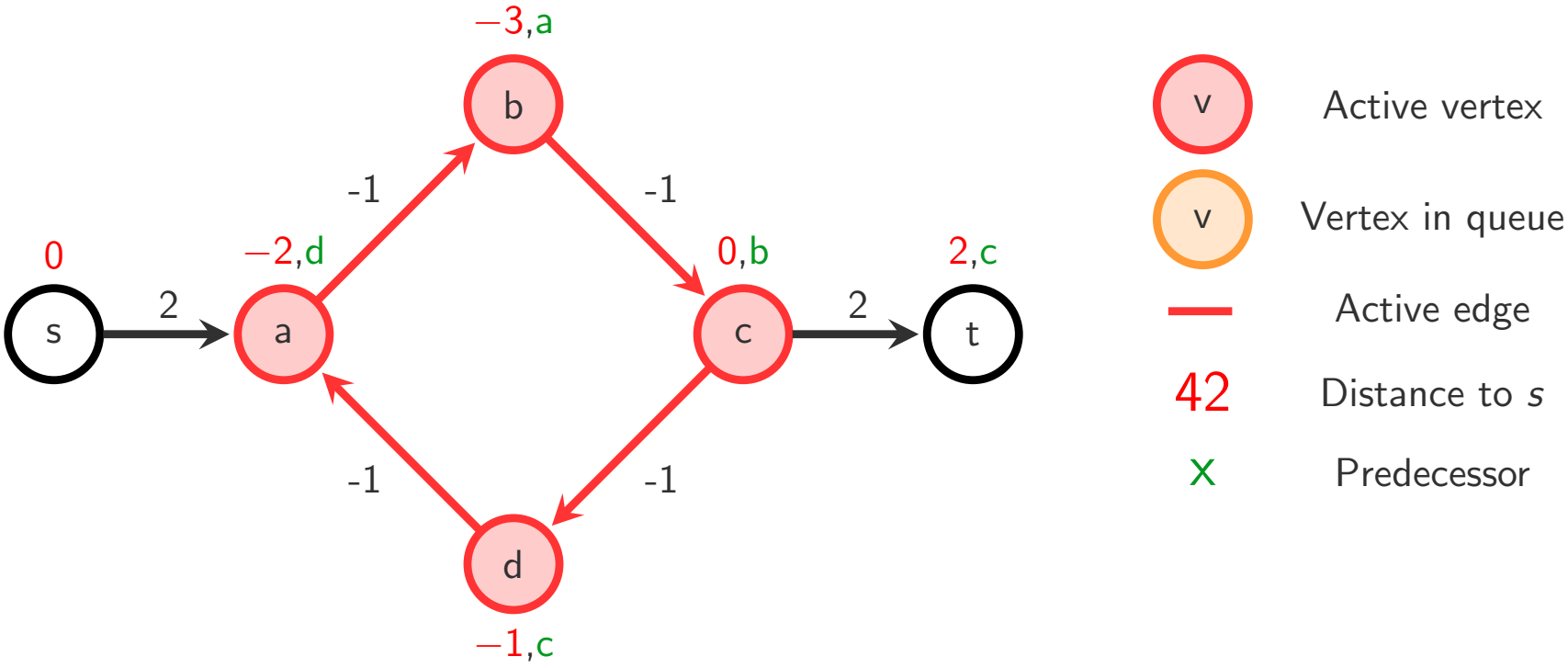
Bellman-Ford Algorithm with Negative Cycles

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

Simple version: Obviously $\mathcal{O}(|V||E|)$. Improved version:

- 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?

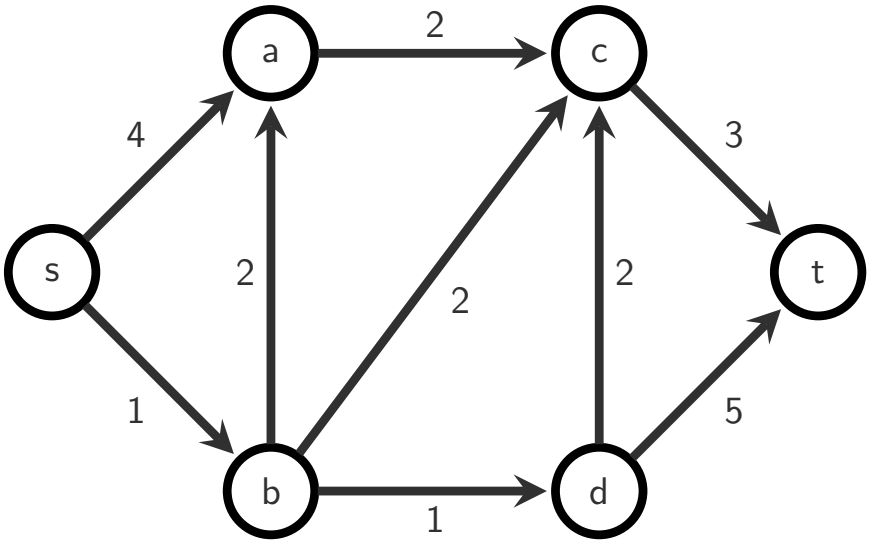
- **Naive approach:** Executing Dijkstra algorithm $|V|$ times
  - Running time: $\mathcal{O}(|V|\,|E| + |V|^2 \log |V|)$
  - Can neither handle negative edge weights nor negative cycles.

- **Floyd-Warshall Algorithm**:
  - Running time: $\mathcal{O}(|V|^3)$
  - 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

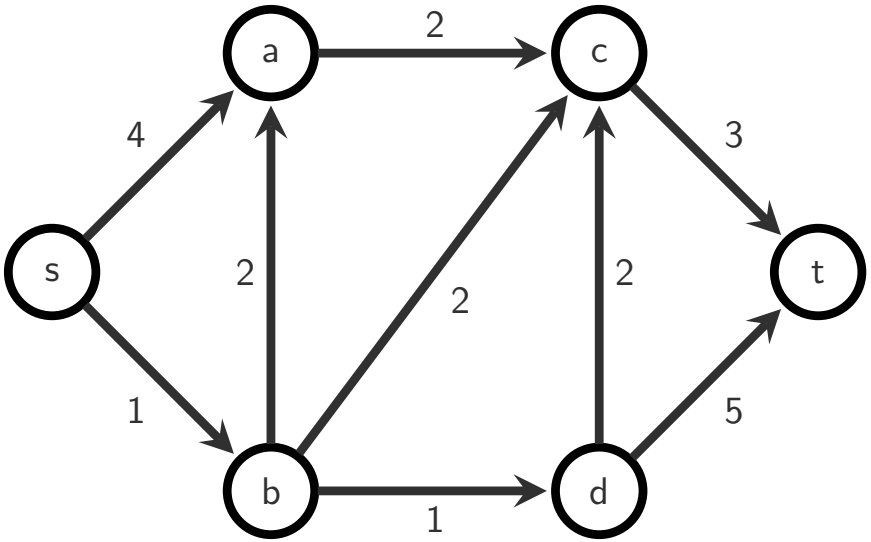
- Represent graph in distance matrix.
- Idea: iteratively try to shortcut over intermediate node



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

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 6 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

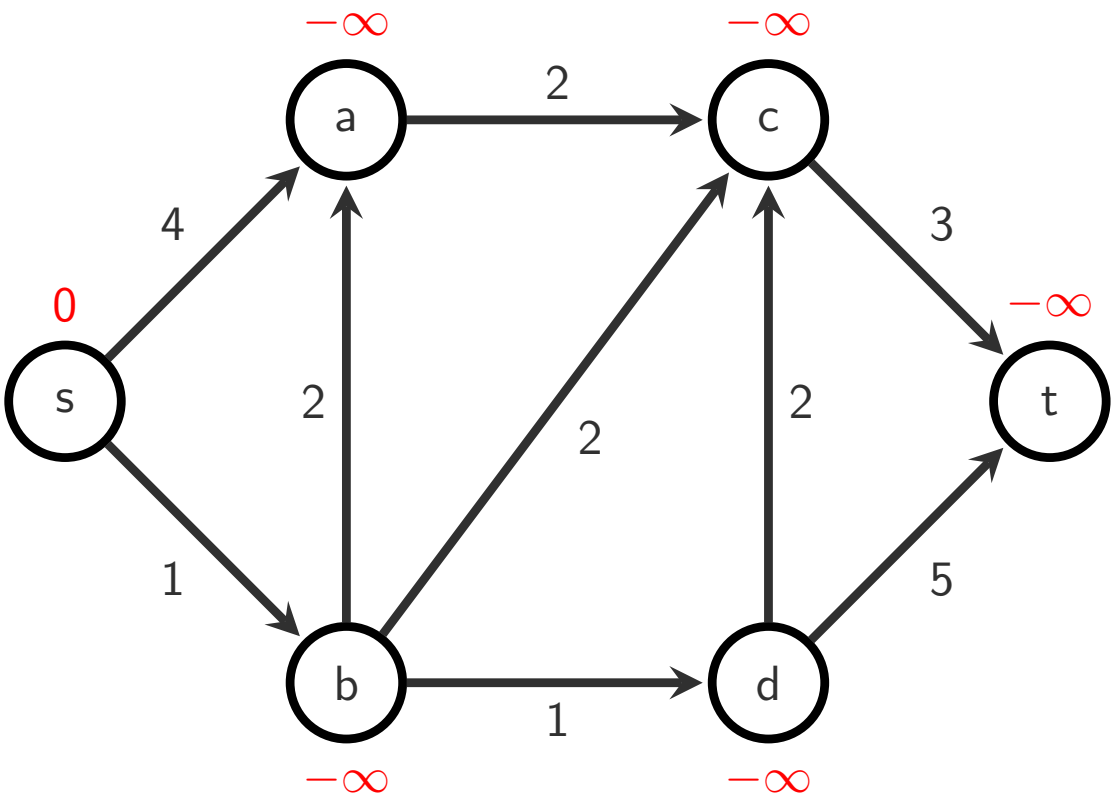
- **Longest Path Problem:** Find a simple path of maximum length between two nodes in a graph.
- NP-hard for general graphs.
- Polynomial time algorithms exist for directed acyclic graphs.
- Application in DAGs: Finding critical paths in scheduling problems.

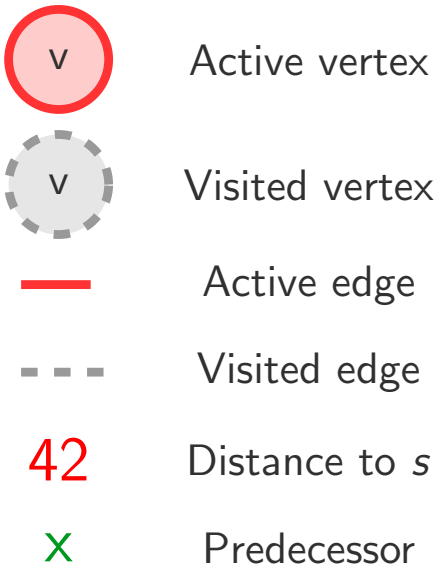
Longest Path Problem

- 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 order 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

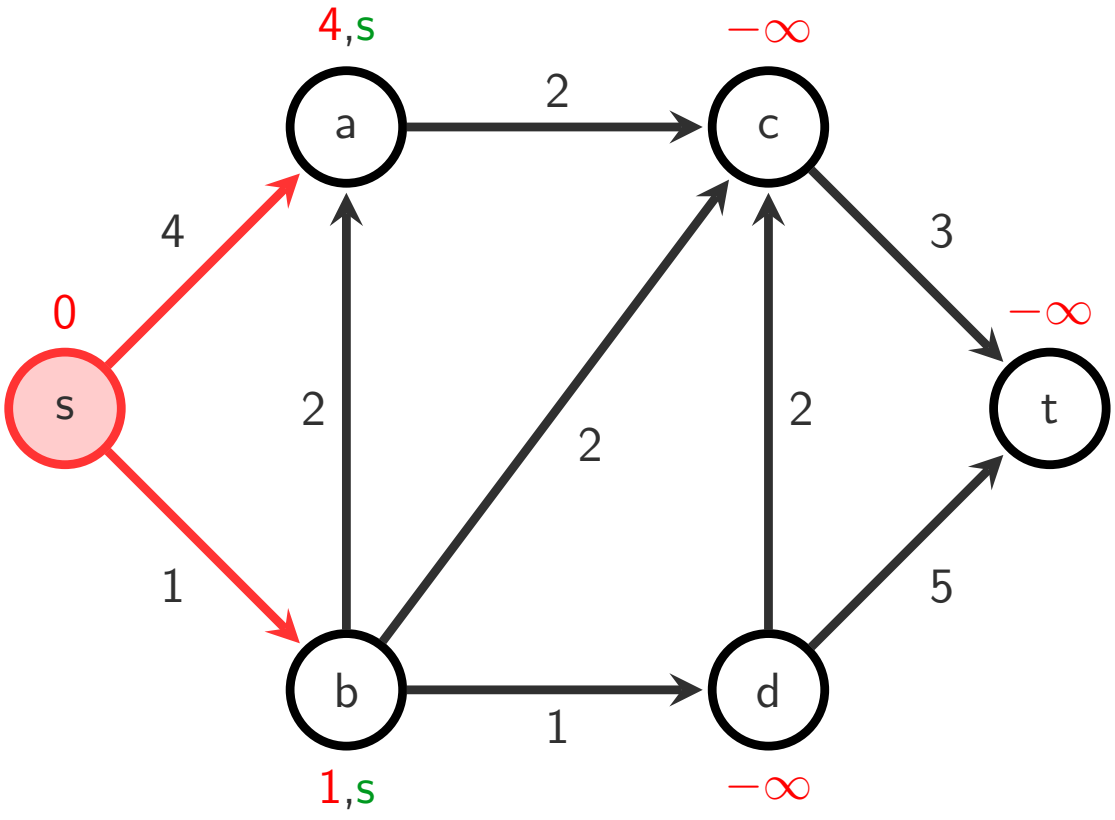
Topological order: s,b,a,d,c,t

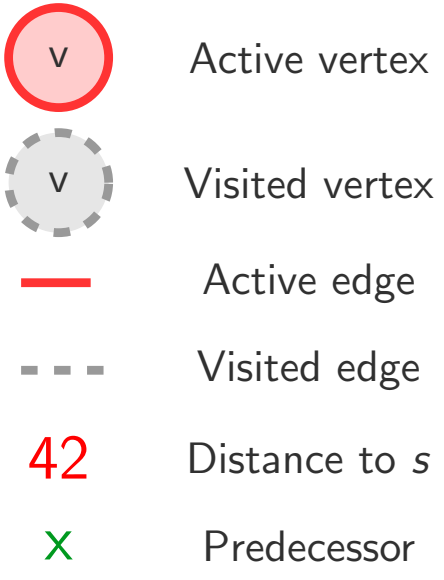




Longest Path in DAG

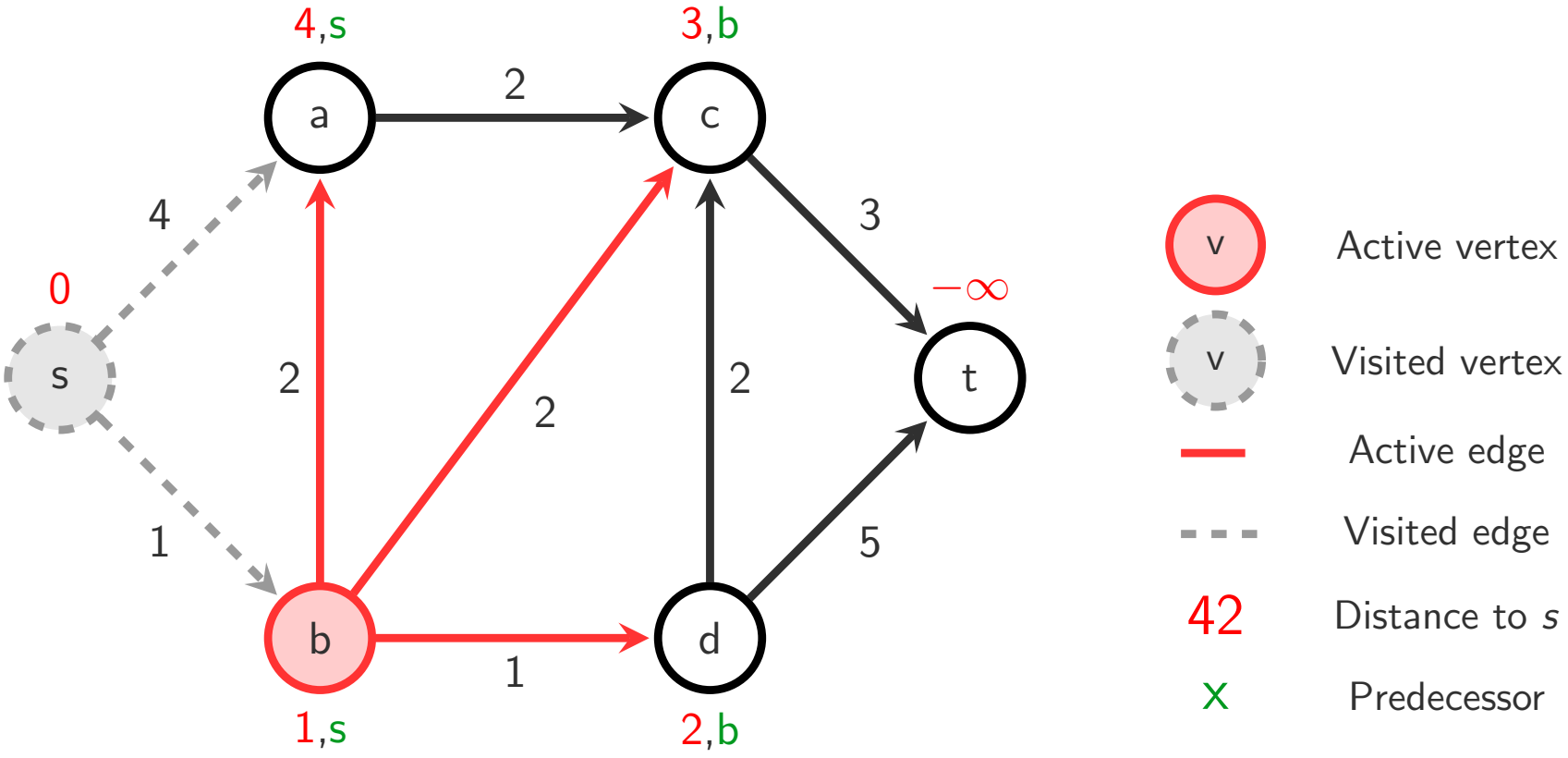
Topological order: s,b,a,d,c,t





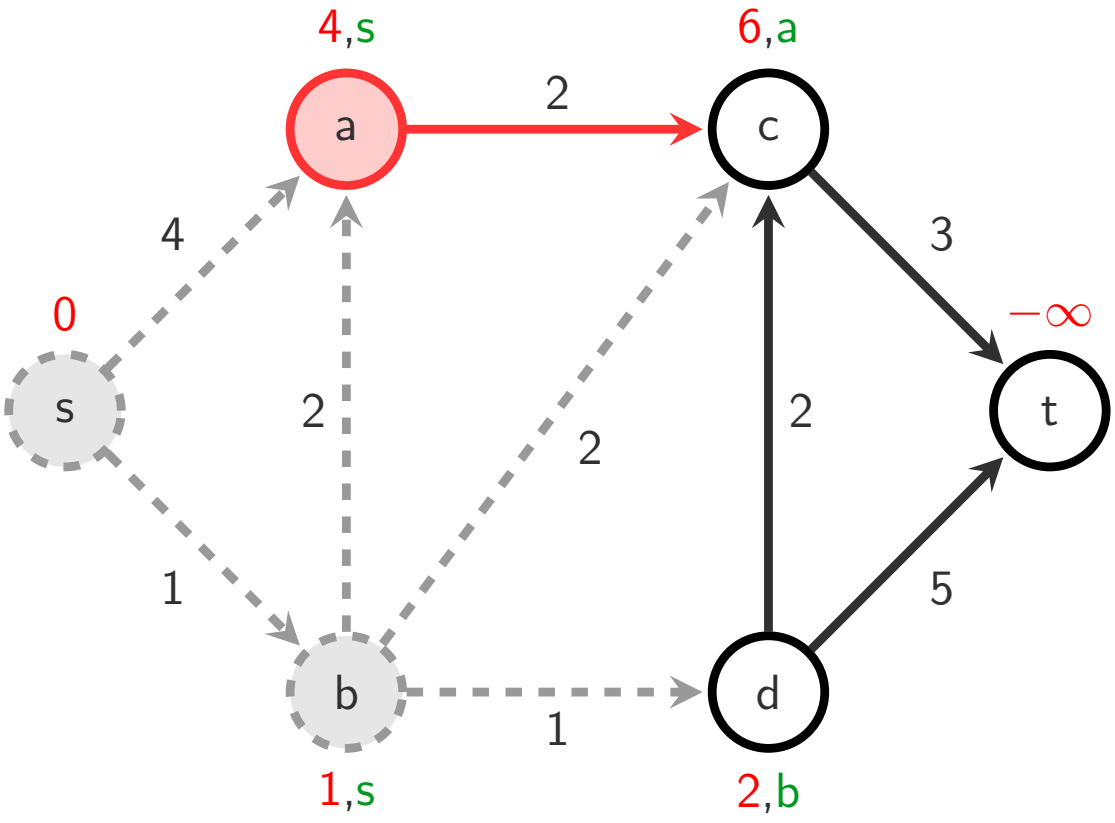
Longest Path in DAG

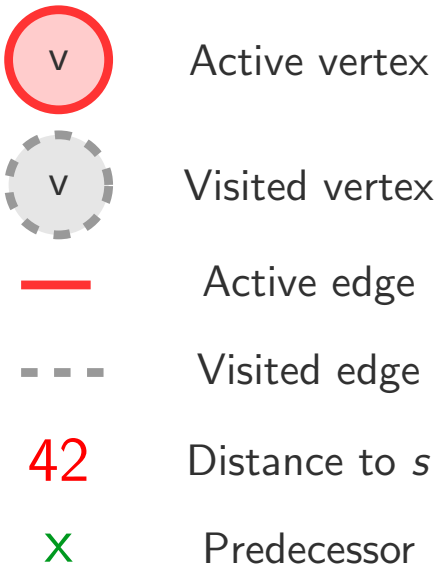
Topological order: s,b,a,d,c,t



Longest Path in DAG

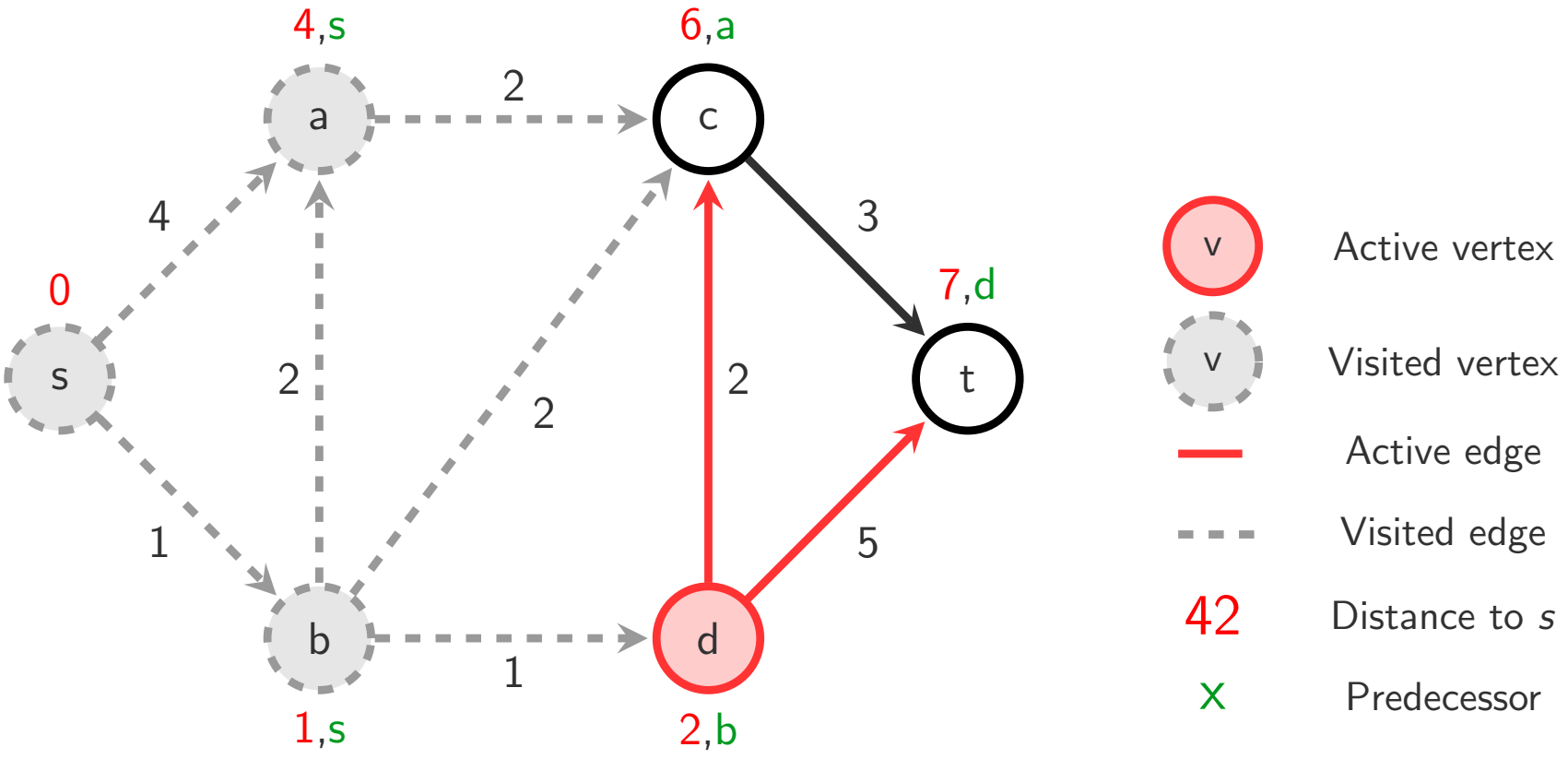
Topological order: s,b,a,d,c,t





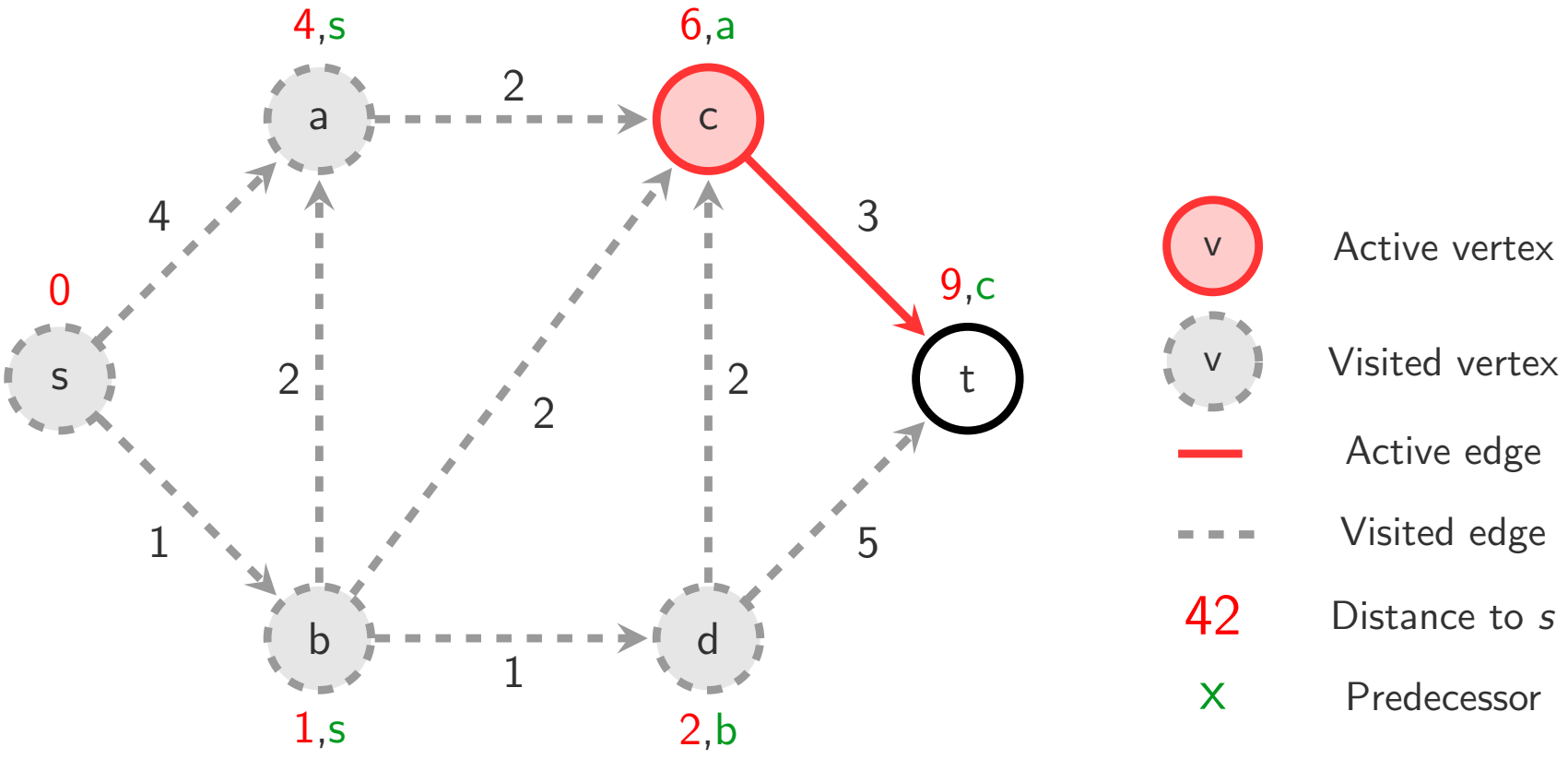
Longest Path in DAG

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



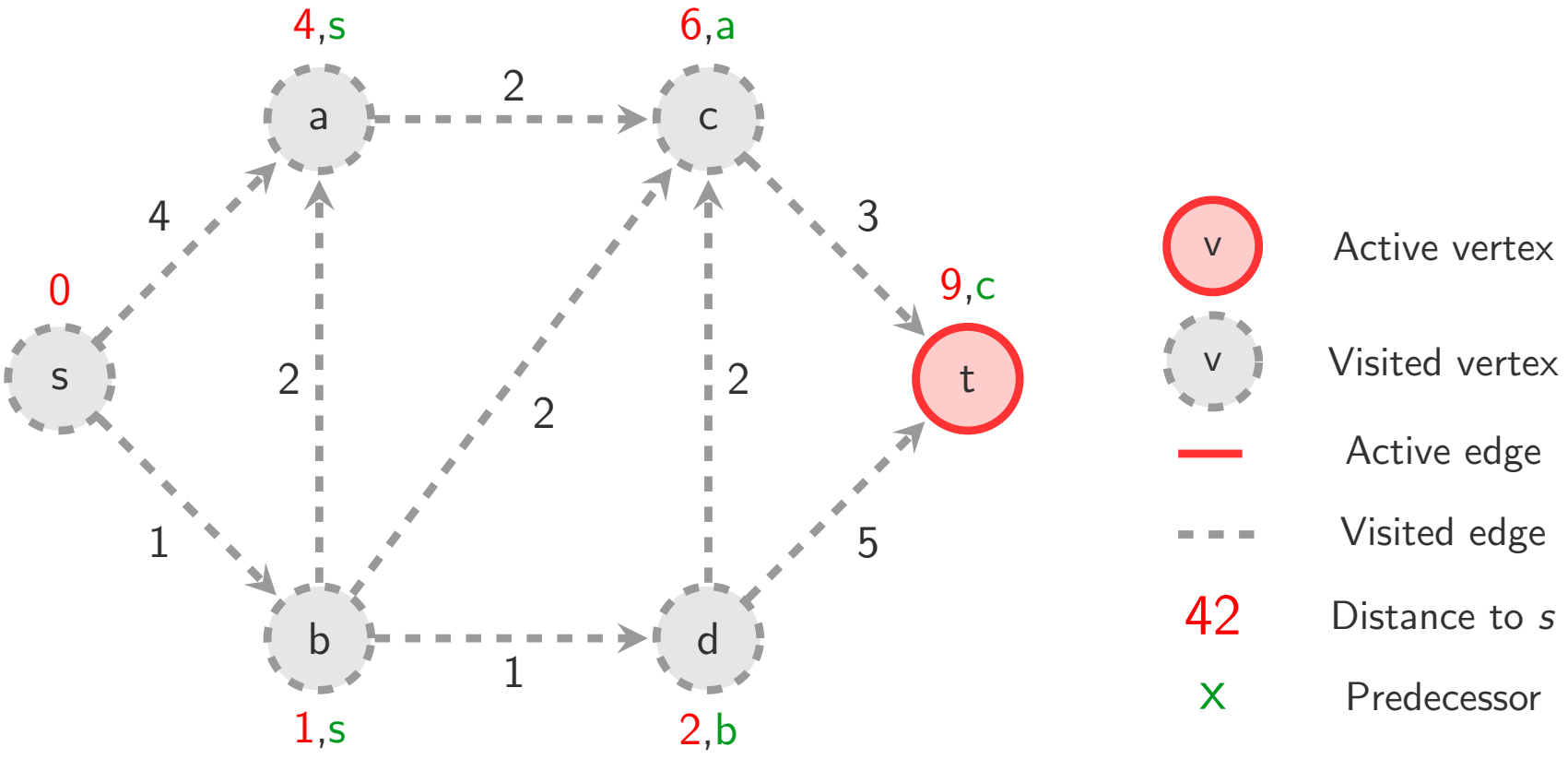
Longest Path in DAG

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