Single Source Shortest Path (I)

The content of this presentation is a collection from various books and online resources, such as Introduction to Algorithms, by CLRS and the lecture notes of Dr. E Demaine and others. Thanks to all for their valuable contributions.

Shortest Path

- In a ***shortest-paths problem***, we are given a weighted, directed graph $G = (V, E)$, with weight function $w: E \rightarrow R$ mapping edges to real-valued weights.
- The ***weight w(p)*** of path $p =< v_0, v_1, \ldots, v_k >$ is the sum of the weights of its constituent edges:

$$w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

- We define the ***shortest-path weight*** $\delta(u, v)$ from u to v by

$$\delta(u, v) = \begin{cases} \min\{w(p) : u \overset{p}{\leadsto} v\} & \text{if there is a path from } u \text{ to } v \text{ ,} \\ \infty & \text{otherwise .} \end{cases}$$

- A ***shortest path*** from vertex u to vertex v is then defined as any path p with weight $w(p) = \delta(u, v)$.

Variants

- Can a single-source shortest-paths algorithm solve the following?
  a) Single-destination shortest-paths problem: Find a shortest path to a given ***destination*** vertex t from each vertex v.
  b) All-pairs shortest paths problem: Find a shortest path from u to v for every pair of vertices u and v.

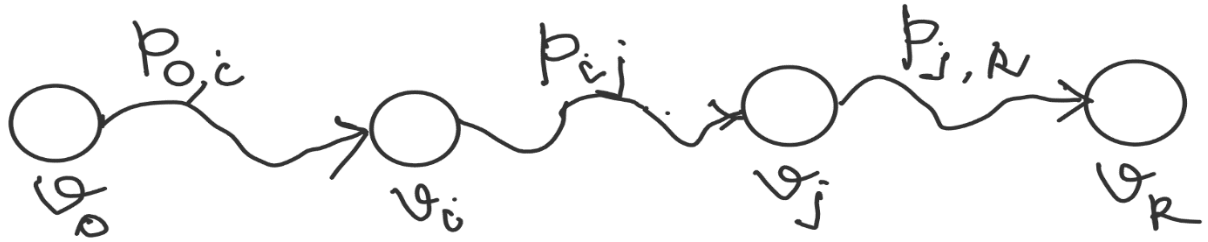
- Answer is Yes.
  - Problem (a) can be solved by reversing the direction of each edge in the graph, we can reduce this problem to a single-source problem.
  - For problem (b), although we can solve this problem by running a single source algorithm once from each vertex, we usually can solve it faster.

Optimal substructure of a shortest path

- **Lemma:** Given a weighted, directed graph $G = (V, E)$ with a weight function $w: E \rightarrow R$, let $p =< v_0, v_1, \ldots, v_k >$ be a shortest path from v_0 to v_k and, for any i and j such that $0 \leq i \leq j \leq p$, let $p_{ij} =< v_i, v_{i+1}, \ldots, v_j >$ be the shortest sub-path of p from vertex v_i to vertex v_j. Then, p_{ij} is a shortest path from v_i to v_j.

  - That is, *sub-paths of shortest paths are shortest paths*.

Optimal substructure of a shortest path



- **Proof:**
  - Decompose the path p, into three paths p_{0i}, p_{ij}, p_{jk}. That is, $v_0 \to v_i \to v_j \to v_k$.
  - Then we have that $w(p) = w(p_{0i}) + w(p_{ij}) + w(p_{jk})$.

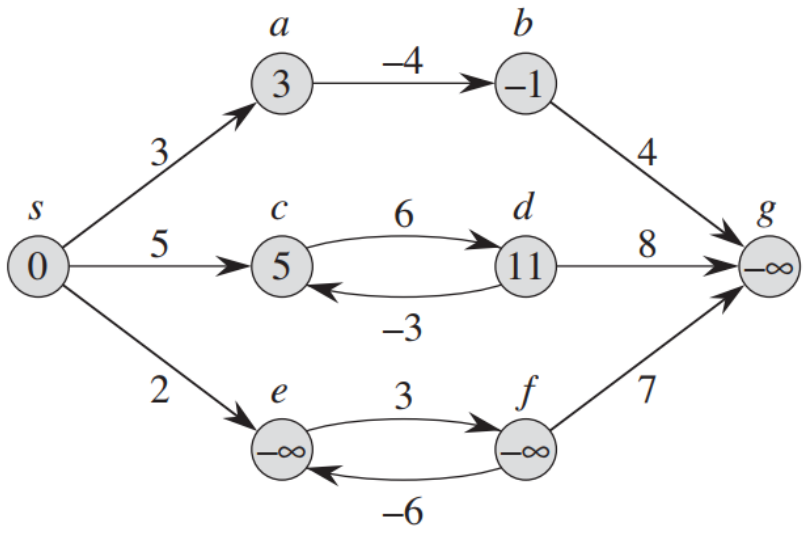
  - The proof is by contradiction.
  - So, we assume that if p_{ij} is not the shortest path between i and j, then there exists a SP, say p'_{ij}.
  - That is, $w(p'_{ij}) < w(p_{ij})$
  - Then, replacing p_{ij} with p'_{ij}, we have a new shortest path from v_0 to v_k, which is p_{0i}, p'_{ij}, p_{jk}, with weight:

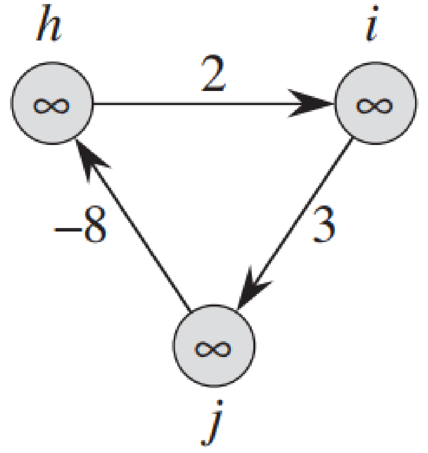
$$w(p') = w(p_{0i}) + w(p'_{ij}) + w(p_{jk}) < w(p)$$

This is a contradiction to the fact the p is the shortest path between v_0 to v_k. Thus, p'_{ij} can not be a shortest sub-path between i and j.

Negative-weight edges

- Some edges can have negative weights.
- If a graph has negative weights (for some edges) reachable from s, the shortest-path weight $\delta(s, v)$ remains well defined, even if it has negative value.
- However, when a graph has **negative-weight cycle** reachable from s, the shortest-path weights are not well defined.
- We can always find a path with lower weight by following the proposed "shortest" path and then traversing the negative-weight cycle. If there is a **negative-weight cycle** on some path from s to v, we define $\delta(s, v) = -\infty$.
- Dijkstra's algorithm assumes that all weights are nonnegative. Bellman-Ford algorithm allows negative weight edges.





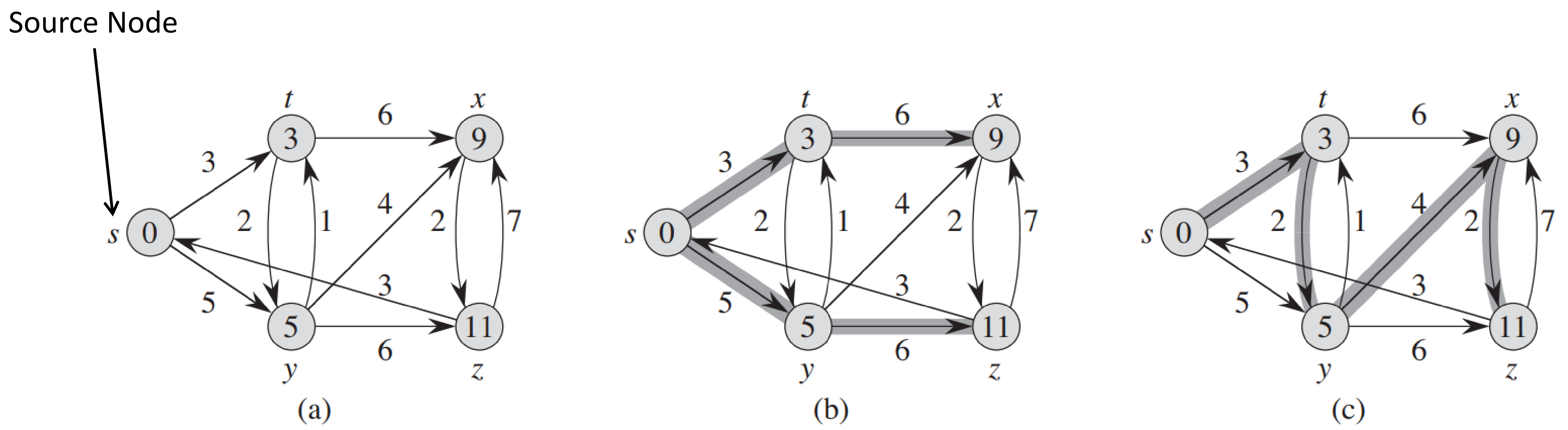
Can a shortest path contain a cycle?

- It cannot contain a negative weight cycle.
- Nor can it contain a positive weight cycle. Because we can get a shorter path by removing such a cycle.
- This implies it can only contain a 0-weight cycle.
    - But, we can remove a 0-weight cycle from any path and produce another path whose weight is same.
    - We can repeatedly remove all 0-weight cycles.
- Therefore, when we are finding shortest paths, we can assume that they have no cycles.
- Thus, in a graph with $|V|$ vertices, we restrict our attention to shortest paths of at most $|V| - 1$ edges.

Representing shortest paths

- We often wish to compute not only shortest-path weights, but the vertices on shortest paths as well.
- A shortest-paths tree rooted at s is a directed subgraph $G' = (V', E')$, where $V' \subseteq V$ and $E' \subseteq E$, such that:
  - V' is the set of vertices reachable from s in G,
  - G' forms a rooted tree with root s, and
  - For all $v \in V'$, the unique simple path from s to v in G' is a shortest path from s to v in G.

Multiple shortest-paths tree



Initialization of shortest path algorithms

- We initialize the shortest-path estimates and predecessors by the following, $O(V)$-time procedure:

```
INITIALIZE-SINGLE-SOURCE$(G, s)$
1  for each vertex $v \in G.V$
2      $v.d = \infty$
3      $v.\pi = \text{NIL}$
4  $s.d = 0$
```

- $v.d$: is the distance between (s, v)
- $v.\pi$: the parent of v

- After initialization, we have v. $\pi = NIL$ for all $v \in V$,
- $s.d = 0$, and $s.d = \infty$ for $v \in V - \{s\}$

Relaxation

- The algorithms in the following discussion use a technique, called ***Relaxation***.

- For each vertex v, we maintain an attribute $v.d$, which is an upper bound on the weight of a shortest path from source s to v.
  - We call $v.d$ a shortest-path estimate.

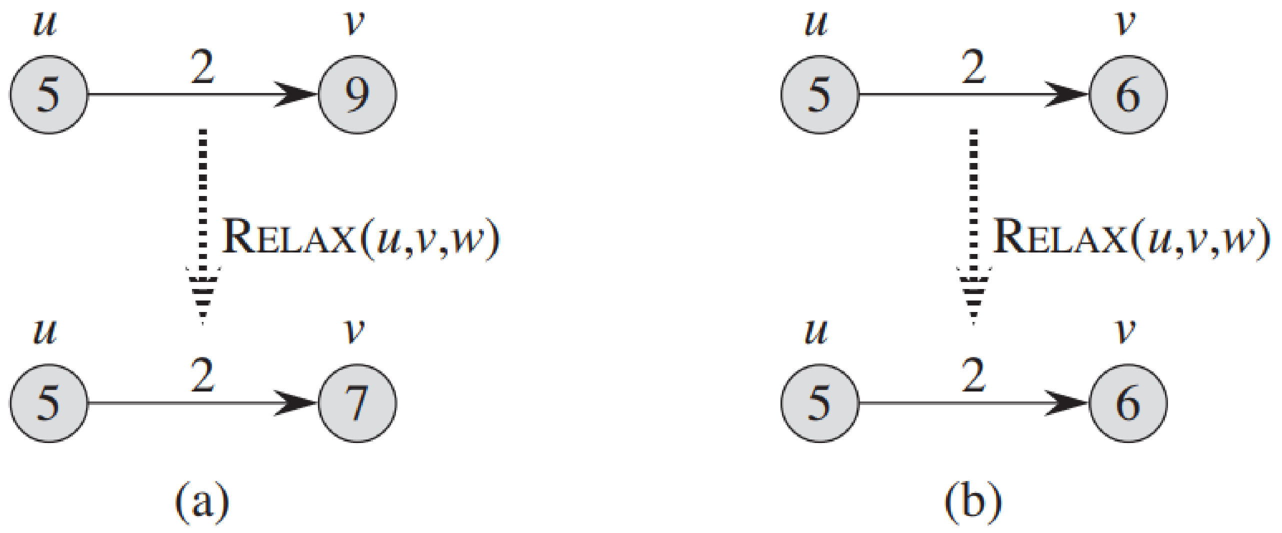
- The process of relaxing an edge consists of testing whether we can improve the shortest path to v found so far by going through u and if so, updating $v.d$ and $v.\pi$.

- A relaxation step may decrease the value of the shortest-path estimate $v.d$ and update v's predecessor attribute π.

- The following code performs a relaxation step on edge (u, v) in $O(1)$ time:

```
RELAX($u, v, w$)
if $v.d > u.d + w(u, v)$
    $v.d = u.d + w(u, v)$
    $v.\pi = u$
```

- Figure, shows two examples of relaxing an edge, one in which a shortest-path estimate decreases and one in which no estimate changes.

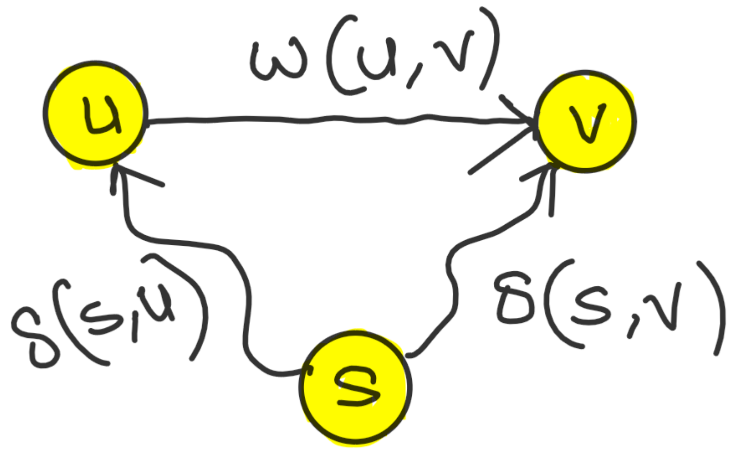


Properties of shortest paths and relaxation

- Triangle inequality: For any edge $(u, v) \in E$, we have $\delta(s, v) \leq \delta(s, u) + w(u, v)$.
- Upper-bound property: We always have $v.d \geq \delta(s, v)$ for all vertices $v \in V$, and once $v.d$ achieves the value $\delta(s, v)$, it never changes.
- No-path property: If there is no path from s to v, then we always have $v.d = \delta(s, v) = \infty$.
- Convergence property: If $s \rightsquigarrow u \rightarrow v$ is a shortest path in G for some $u, v \in V$, and if $u.d = \delta\ (s, u)$ at any time prior to relaxing edge (u, v) then $v.d = \delta(s, v)$ at all times afterward.
- Path-relaxation property: If $p =< v_0, v_1, \dots, v_k >$ is a shortest path from $s = v_0$ to v_k, and we relax the edges of p in the order $(v_0, v_1), (v_1, v_2), \dots, (v_{k-1}, v_k)$, then $v_k.d = \delta(s, v_k)$.
    - This property holds regardless of any other relaxation steps that occur, even if they are intermixed with relaxations of the edges of p.
- Predecessor-subgraph property: Once $v.d = \delta(s, v)$ for all $v \in\ V$, the predecessor subgraph is a shortest-paths tree rooted at s

Properties of shortest paths and relaxation

- Triangle inequality: For any edge $(u, v) \in E$, we have $\delta(s, v) \leq \delta(s, u) + w(u, v)$. WHY?
- Since there is an edge between (u, v), then the shortest path to v must either pass through u or some other node (say u')
  - In case, it pass through u then $\delta(s, v)$ must be equal to $\delta(s, u)$ + weight of (u, v) (by relaxation).
  - Else, if it passes through u', then the weight of the path to v (via u), i.e., $\delta(s, u) + \delta(u, v)$, should be more than $\delta(s, v)$.
  - Thus we have, $\delta(s, v) \leq \delta(s, u) + w(u, v)$



- Note: This does not imply that

$$w(s, v) \leq w(s, u) + w(u, v)$$

Homework

- **To Dos:** Go though the all properties of listed in slide-14, and argue, why they are TRUE!!!

- **Question:** What will have if we list the edges of a graph (in some sequence) and apply the relaxation function 3 times on that list. What can you say about the shortest path from some node (say s)?

Bellman-Ford algorithm vs Dijkstra’s algorithm

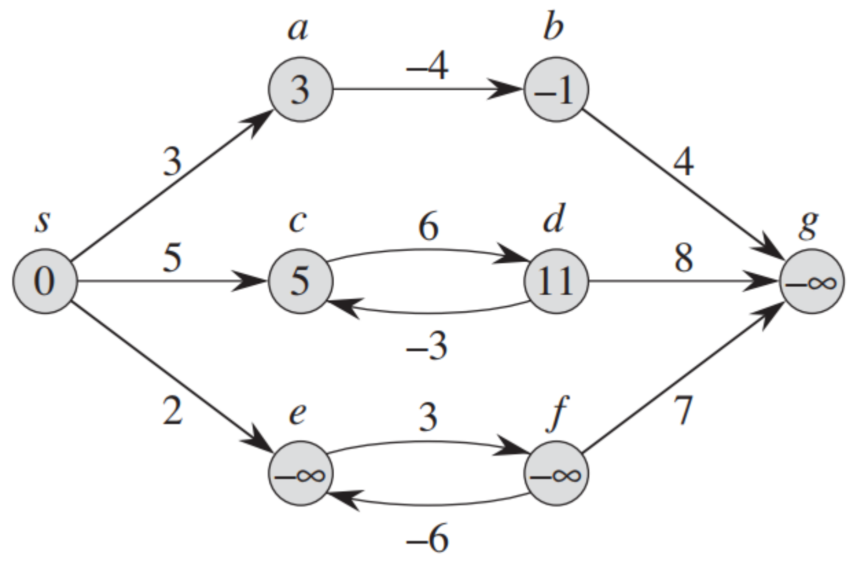
- Bellman-Ford algorithm (BFA)
  - A general solution to the single-source shortest-paths problem
  - Remarkably simple
  - Edges can have negative weights
  - Detects whether a negative-weight cycle is reachable from the source
- Dijkstra’s algorithm (DA)
  - Runs faster
  - Used when edge weights are non-negative $w(u, v) \geq 0, \forall\ (u, v) \in E$

Bellman-Ford algorithm

- Idea of Bellman-Ford algorithm
    - After relaxing all edges we have relaxed the first edge on any shortest path
    - Similarly, after relaxing all edges once more, we have relaxed the second edge.
    - etc.

Bellman-Ford algorithm

- Maintain $v.d$ for each vertex, which estimates the shortest path from s to v.
- The algorithm relaxes edges, progressively decreasing an estimate $v.d$ until it achieves the actual shortest-path weight $\delta(s, v)$.
- Finally, it returns a Boolean value indicating whether or not there is a negative-weight cycle that is reachable from the source.
- If such a negative-weight cycle exists, the algorithm indicates that no solution exists.
- For an undirected graph, replace each edge with two directed edges.



Bellman-Ford algorithm

- After initializing the d and π values of all vertices in line 1, the algorithm makes $|V| - 1$ passes over the edges of the graph.
- Each pass is one iteration of the **for** loop of lines 2–4 and consists of relaxing each edge of the graph once.
- Figure bellows shows an example.
- After making $|V| - 1$ passes, lines 5–8 check for a negative-weight cycle and return the appropriate Boolean value.

- It is easy to see that the Bellman-Ford algorithm runs in time $O(VE)$.

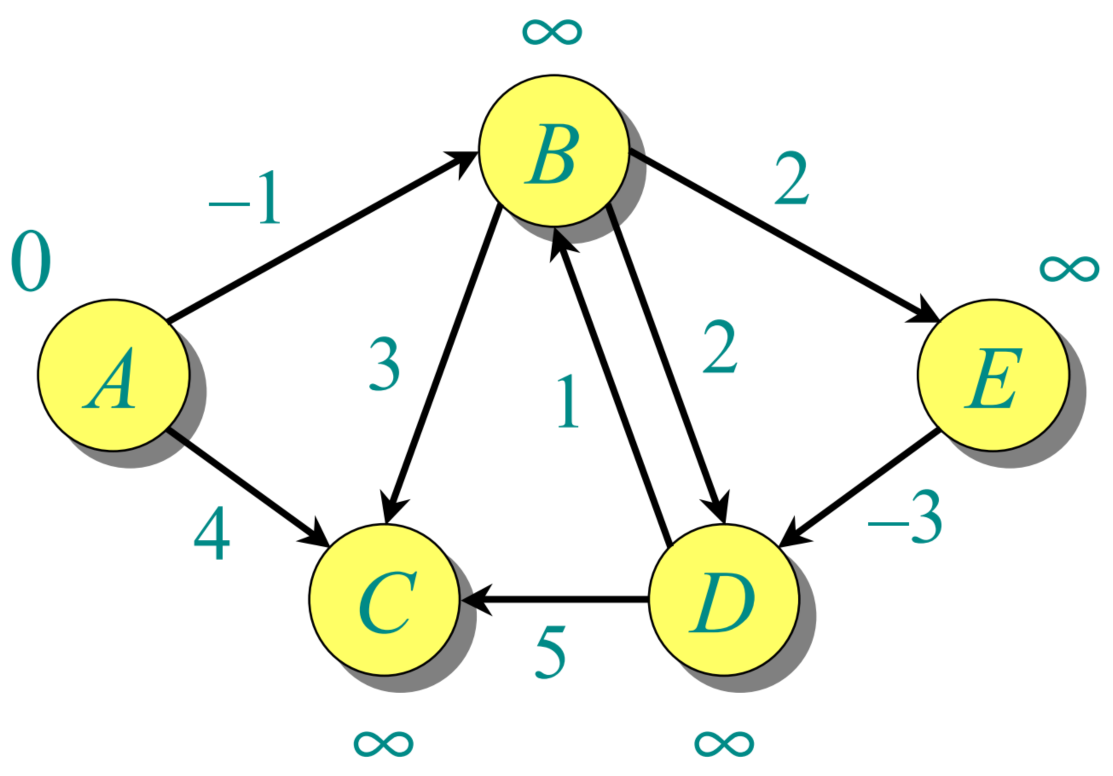
```
BELLMAN-FORD$(G, w, s)$
1  INITIALIZE-SINGLE-SOURCE$(G, s)$
2  for $i = 1$ to $|G.V| - 1$
3      for each edge $(u, v) \in G.E$
4          RELAX$(u, v, w)$
5  for each edge $(u, v) \in G.E$
6      if $v.d > u.d + w(u, v)$
7          return FALSE
8  return TRUE
```

```
INITIALIZE-SINGLE-SOURCE$(G, s)$
1  for each vertex $v \in G.V$
2      $v.d = \infty$
3      $v.\pi = $ NIL
4  $s.d = 0$
```

```
RELAX$(u, v, w)$
1  if $v.d > u.d + w(u, v)$
2      $v.d = u.d + w(u, v)$
3      $v.\pi = u$
```

Bellman-Ford algorithm (Example-1)

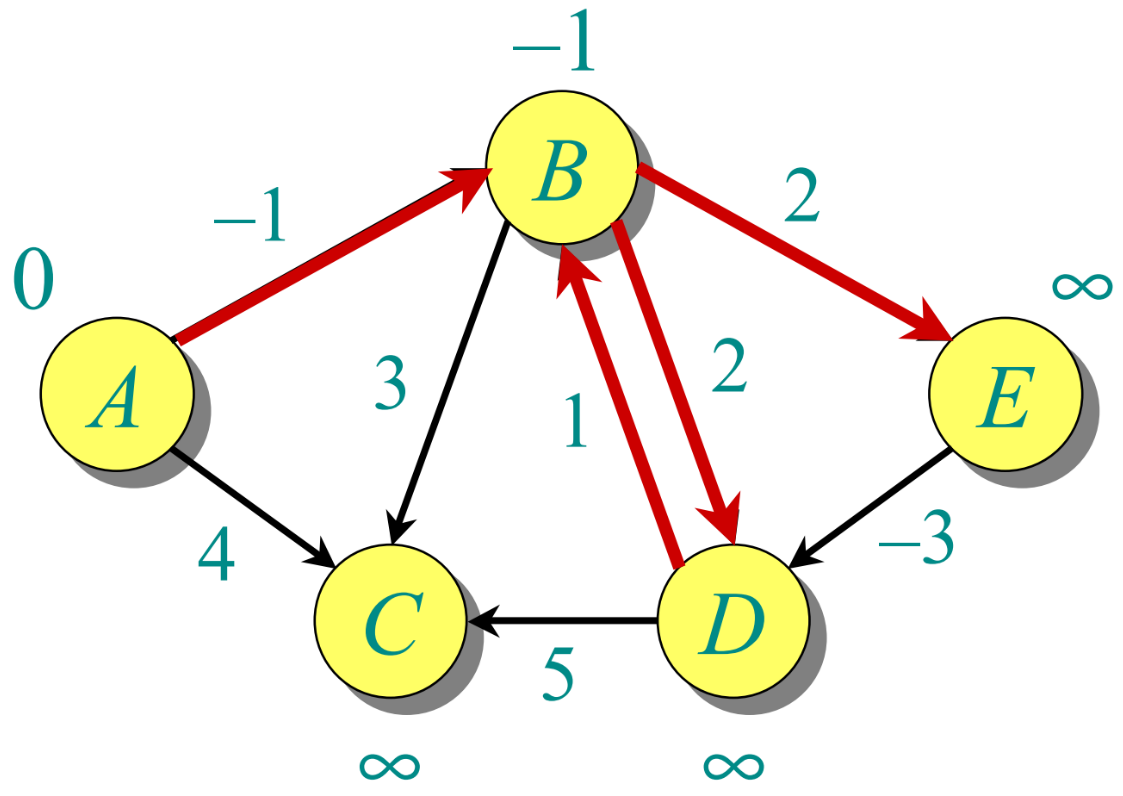
Order of edges: *(B,E), (D,B), (B,D), (A,B), (A,C), (D,C), (B,C), (E,D)*



A	*B*	*C*	*D*	*E*
0	∞	∞	∞	∞

Bellman-Ford algorithm (Example-1)

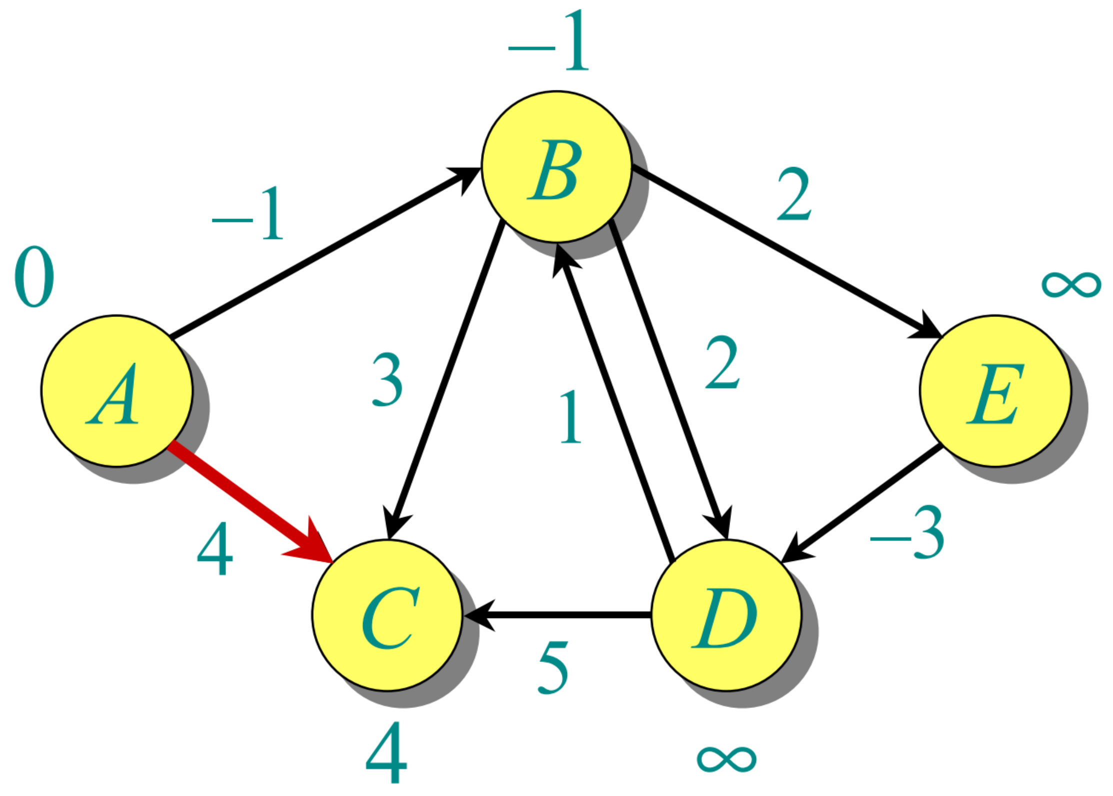
Order of edges: *(B,E), (D,B), (B,D), (A,B), (A,C), (D,C), (B,C), (E,D)*



A	B	C	D	E
0	∞	∞	∞	∞
0	–1	∞	∞	∞

Bellman-Ford algorithm (Example-1)

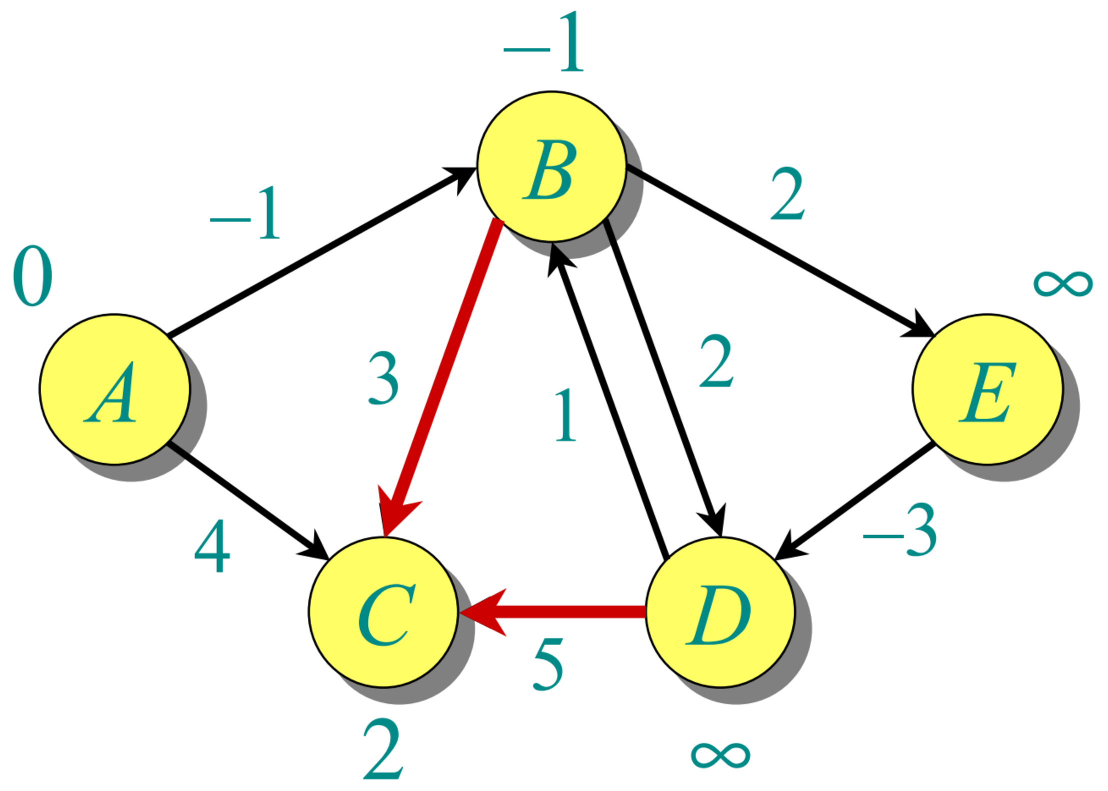
Order of edges: *(B,E), (D,B), (B,D), (A,B), (A,C), (D,C), (B,C), (E,D)*



A	*B*	*C*	*D*	*E*
0	∞	∞	∞	∞
0	–1	∞	∞	∞
0	–1	4	∞	∞

Bellman-Ford algorithm (Example-1)

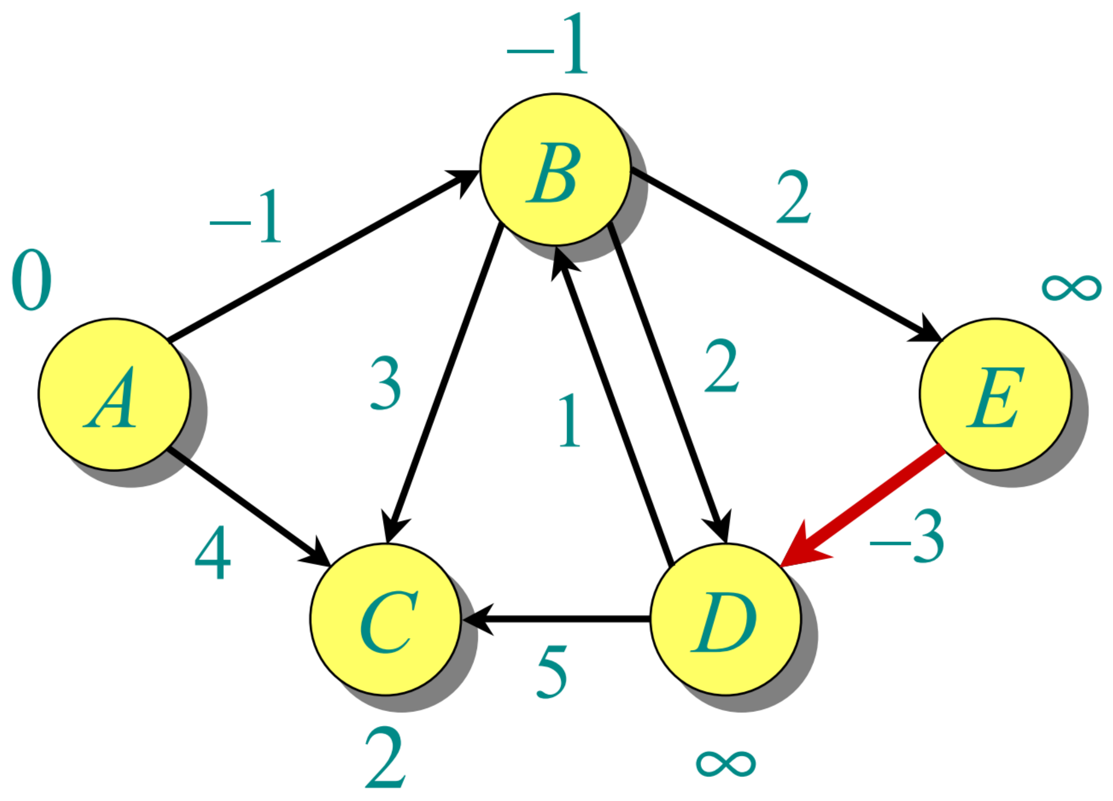
Order of edges: *(B,E), (D,B), (B,D), (A,B), (A,C), (D,C), (B,C), (E,D)*



A	*B*	*C*	*D*	*E*
0	∞	∞	∞	∞
0	–1	∞	∞	∞
0	–1	4	∞	∞
0	–1	2	∞	∞

Bellman-Ford algorithm (Example-1)

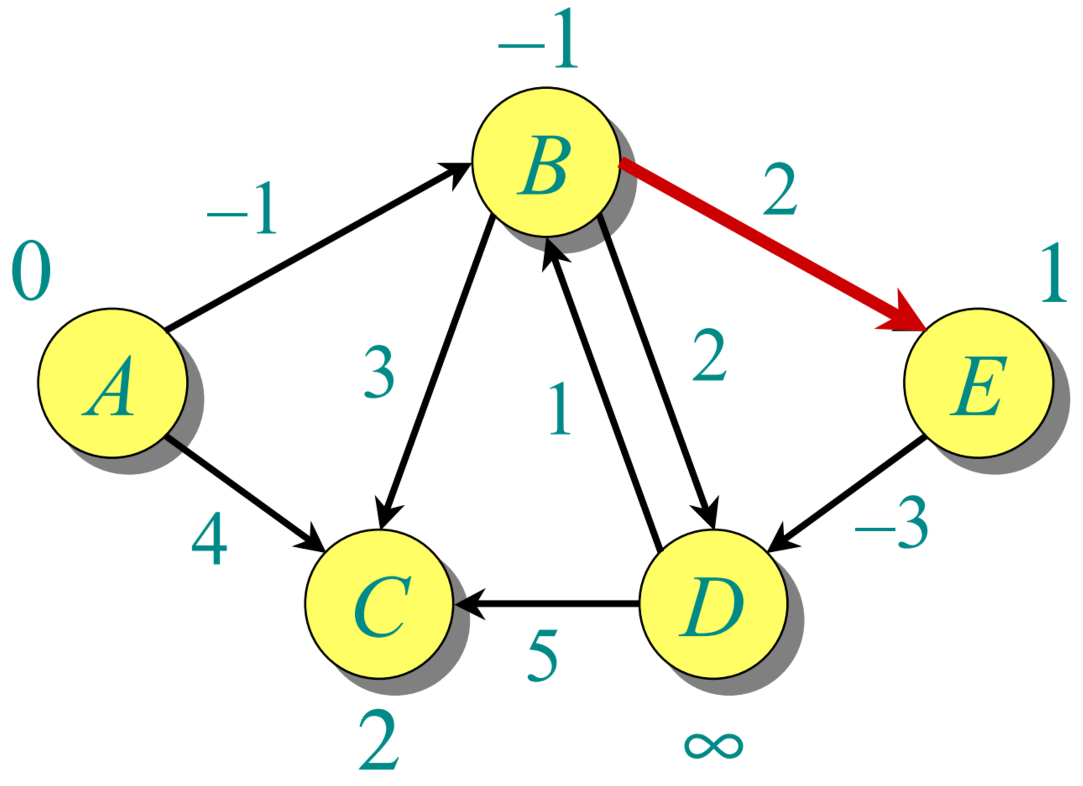
Order of edges: *(B,E), (D,B), (B,D), (A,B), (A,C), (D,C), (B,C), (E,D)*



A	*B*	*C*	*D*	*E*
0	∞	∞	∞	∞
0	–1	∞	∞	∞
0	–1	4	∞	∞
0	–1	2	∞	∞

Bellman-Ford algorithm (Example-1)

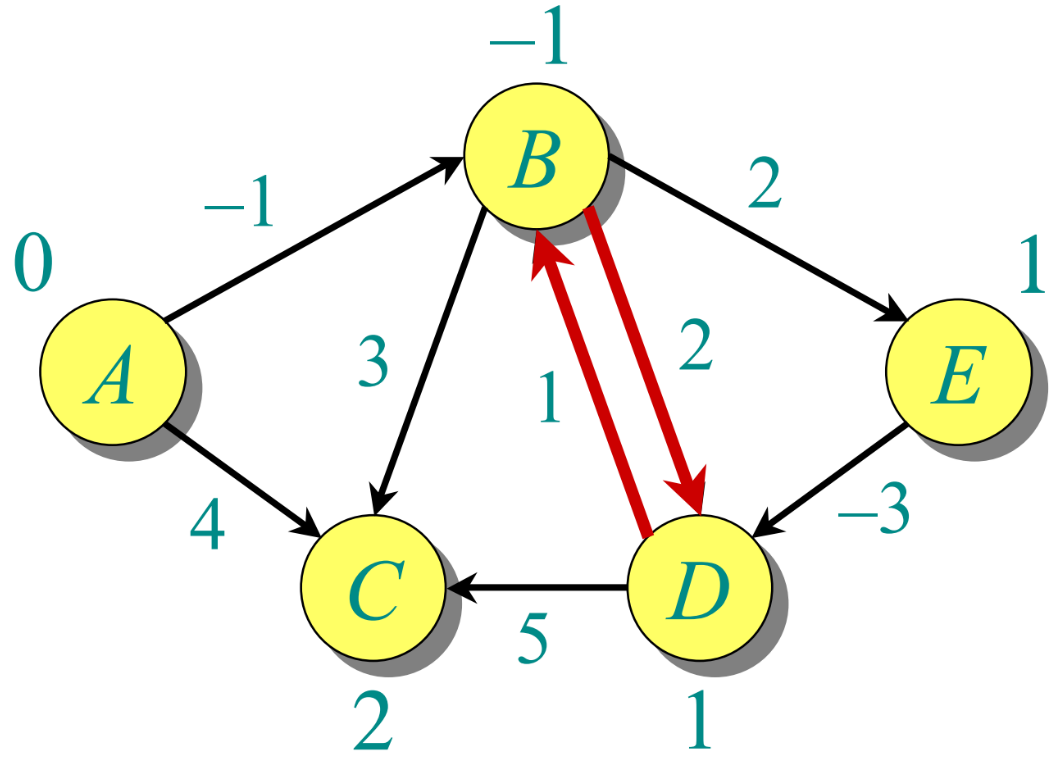
Order of edges: *(B,E), (D,B), (B,D), (A,B), (A,C), (D,C), (B,C), (E,D)*



A	*B*	*C*	*D*	*E*
0	∞	∞	∞	∞
0	–1	∞	∞	∞
0	–1	4	∞	∞
0	–1	2	∞	∞
0	–1	2	∞	1

Bellman-Ford algorithm (Example-1)

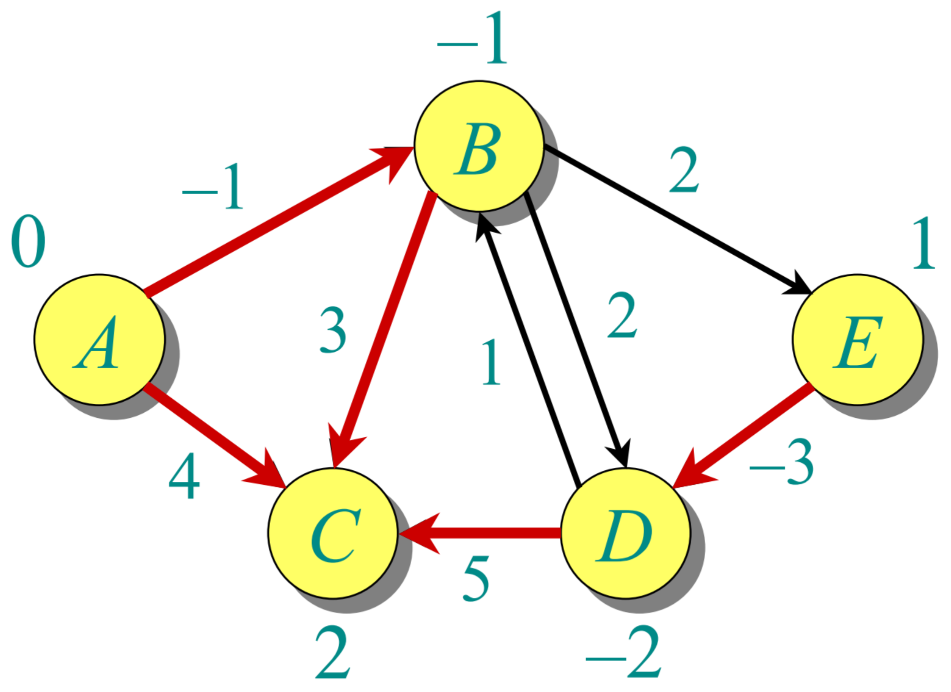
Order of edges: *(B,E), (D,B), (B,D), (A,B), (A,C), (D,C), (B,C), (E,D)*



A	*B*	*C*	*D*	*E*
0	∞	∞	∞	∞
0	–1	∞	∞	∞
0	–1	4	∞	∞
0	–1	2	∞	∞
0	–1	2	∞	1
0	–1	2	1	1

Bellman-Ford algorithm (Example-1)

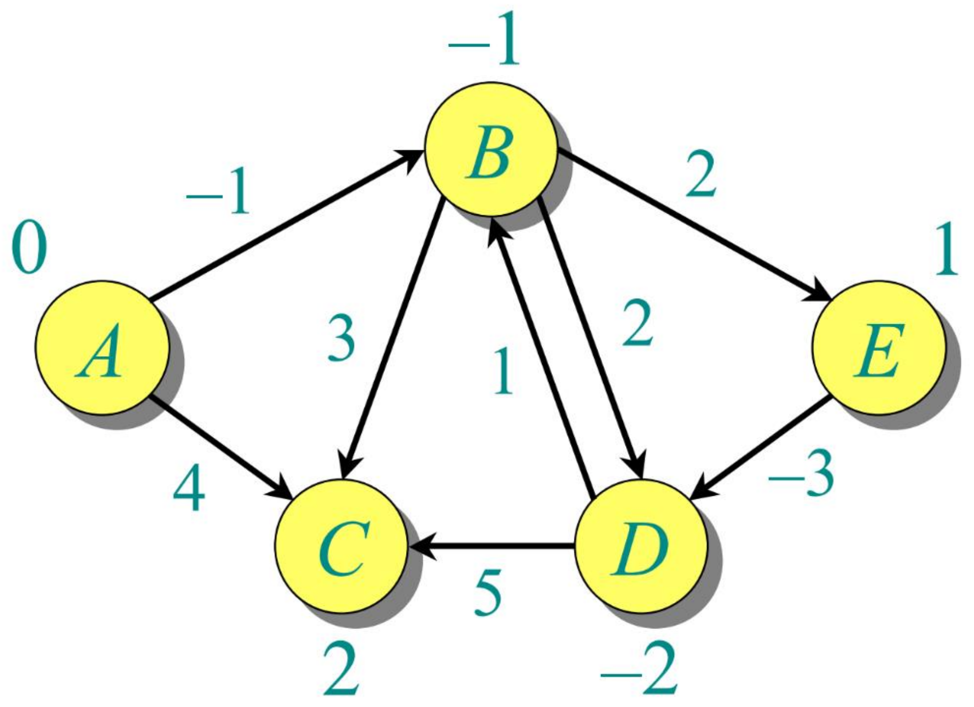
Order of edges: *(B,E), (D,B), (B,D), (A,B), (A,C), (D,C), (B,C), (E,D)*



A	*B*	*C*	*D*	*E*
0	∞	∞	∞	∞
0	−1	∞	∞	∞
0	−1	4	∞	∞
0	−1	2	∞	∞
0	−1	2	∞	1
0	−1	2	1	1
0	−1	2	−2	1

Bellman-Ford algorithm (Example-1)

Order of edges: *(B,E), (D,B), (B,D), (A,B), (A,C), (D,C), (B,C), (E,D)*



A	*B*	*C*	*D*	*E*
0	∞	∞	∞	∞
0	–1	∞	∞	∞
0	–1	4	∞	∞
0	–1	2	∞	∞
0	–1	2	∞	1
0	–1	2	1	1
0	–1	2	–2	1

1st iteration

2nd iteration.

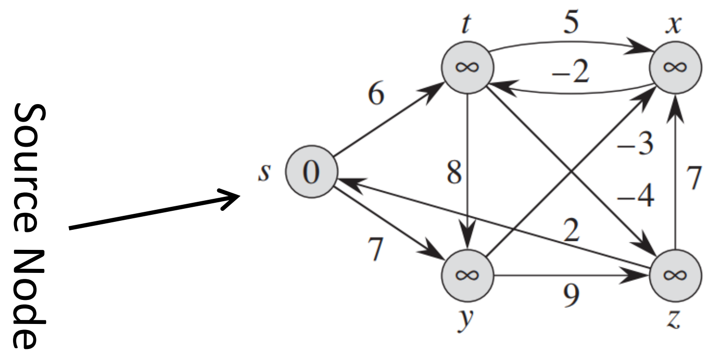
Should we stop here… or is it possible to get a better shortest path tree for the given graph???

Note: Values decrease monotonically.

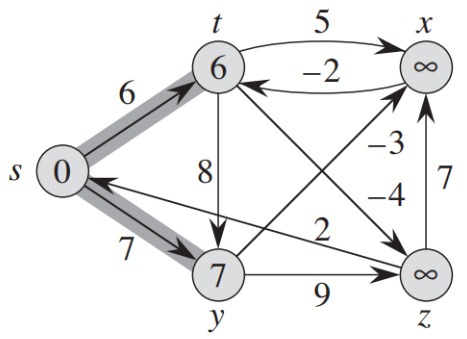
Bellman-Ford algorithm (some modification can be done)

- One can stop the algorithm if an iteration does not modify distance estimates.
- This is beneficial if shortest paths are likely to be less than $|V| - 1$.
- One can detect negative weight cycles by checking whether distance estimates can be reduced after |V|-1 iterations.

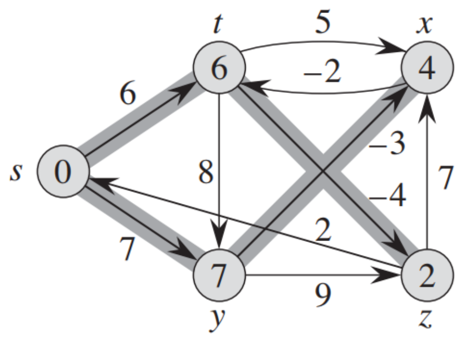
Bellman-Ford algorithm (Example-2)



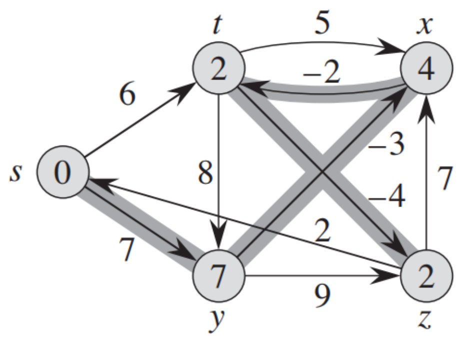
(a)



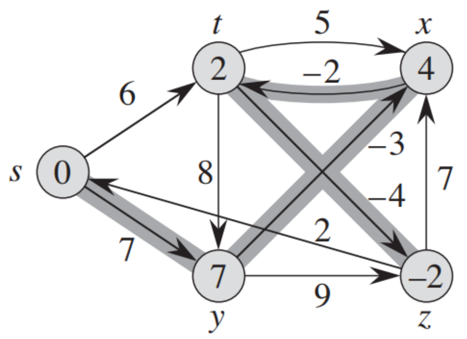
(b)



(c)



(d)



(e)

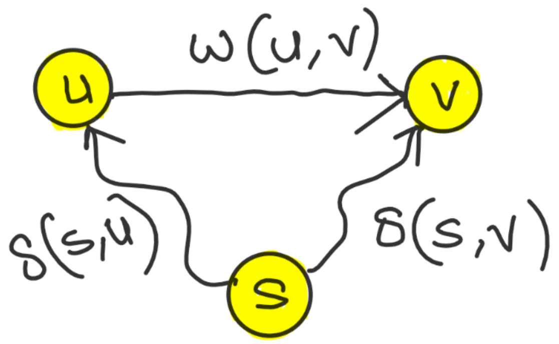
```
RELAX$(u, v, w)$
1  if $v.d > u.d + w(u, v)$
2      $v.d = u.d + w(u, v)$
3      $v.\pi = u$
```

In the example, each pass relaxes the edges in the following order $(t,x),(t,y),(t,z),(x,t),(y,x),(y,z),(z,x),(z,s),(s,t),(s,y)$.

Bellman-Ford algorithm

- **Lemma-1:** The relaxation algorithm maintains the invariant that $v.d \geq \delta(s, v)$, for all $v \in V$.
  - That is the relaxation step $v.d$ never under-estimates the actual shortest path distance from s to v.

- **Proof:** By induction on the number of steps.
  - Consider that we call the relax function: $\mathrm{relax}(u, v, w)$
  - By induction hypothesis, we assume that the above statement is true for u, that is $u.d \geq \delta(s, u)$.
  - By triangular inequality, we have:
    $$\delta(s, v) \leq \delta(s, u) + \delta(u, v) \leq \delta(s, u) + w(u, v)$$
  - So, setting $v.d = u.d + w(u, v)$, is safe for relaxation.



Bellman-Ford Algorithm

- **Lemma-2:** If a graph has no negative-weight cycle, then for any vertex v with shortest path from s consists of k edges, Bellman-Ford sets $d(v)$ to the correct value after the k-th iteration of the for-loop (line-2-4). (For any ordering of edges)
  - Thus, after $|V| - 1$ steps, the algorithm will correctly find the shortest path to all vertices.

- Proof: By induction on i-th iteration.
  - Notice that, we start with $s.d = 0$, (where s is the source).
    - Thus, the initialization step, correctly sets the distance of s.

  - Moreover, in the 1st iteration, the algorithm correctly sets the $v_i.d$ of the 1-hop neighbors (say v_i) of s, that has the shortest path length of 1.
  - That is, edge weight $w(s, v_i)$ is the least of all possible paths from s to v_i.
$$v_i.d \geq s.d + w(s, v_i) = w(s, v_i)$$
  - So, the statement is true for the base case.

Bellman-Ford Algorithm

- Proof (Continues):
  - Consider iteration k, and suppose that the theorem is true for all $k' < k$.
  - Let $u \in V$ be an arbitrary node for which there exists a path from s to u with at most k edges.
  - Let w be the node immediately before u on this path.
  - So the shortest path with at most k edges from s to u consists of a shortest path from s to w with at most $k - 1$ edges, and then the edge (w, u).
  - By the induction hypothesis, before k-th iteration starts, the node w knows its distance from s along this path.
  - So when we relax the edge (w, u), we get a distance estimate for u which exactly corresponds to this path.
  - Since this is the shortest possible path of this form, this is the estimate which we will end up with at the end of the iteration.

Single Source Shortest Path (II)

Single-source shortest paths (SSSPs) in Directed Acyclic Graphs (DAG)

- We can compute shortest paths from a single source in $\Theta(V + E)$ time for a weighted DAG (directed acyclic graph).
- Shortest paths are always defined in a DAG, since no negative-weight cycles exist - negative-weight edges can be present.

- At first topologically sort the DAG to impose a linear ordering on the vertices.
    - If the dag contains a path from vertex u to v, then u precedes v in the topological sort.

- Shortest path can be obtained by making just one pass over the vertices in the topologically sorted order.

- Note: Bellman-Ford works on directed cyclic graphs as well, this algorithm is only for directed acyclic graphs (DAG).

SSSPs in DAGs

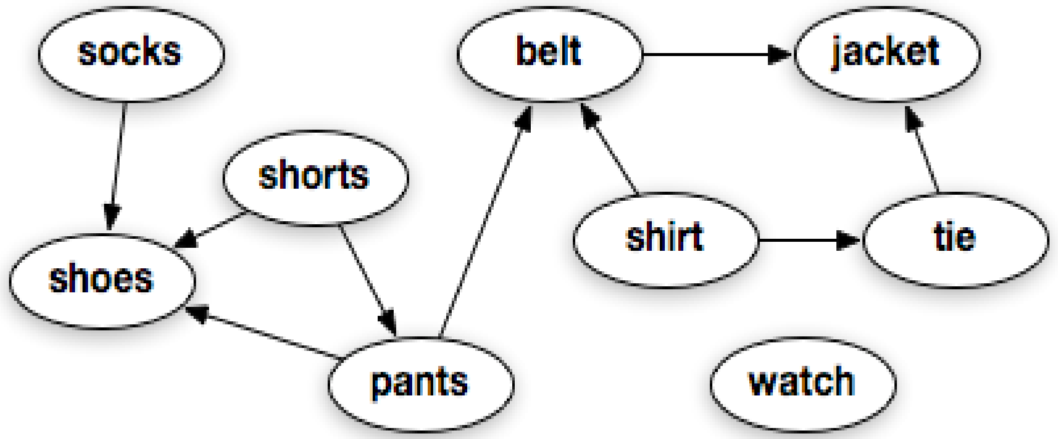
- The algorithm starts by topologically sorting the DAG to impose a linear ordering on the vertices.
- If the DAG contains a path from vertex u to vertex v, then u precedes v in the topological sort.
- We make just one pass over the vertices in the topologically sorted order.
- As we process each vertex, we relax each edge that leaves the vertex.

Topological Sort

- Topological Sort
  - We can use depth-first search to perform a topological sort of a directed acyclic graph (DAG).
  - A ***topological sort*** of a DAG $G(V, E)$ is a linear ordering of all its vertices such that if G contains an edge (u, v), then u appears before v in the ordering.
  - We can view a topological sort of a graph as an ordering of its vertices along a horizontal line so that all directed edges go from left to right.
  - DAG is used to show the precedence among various events.
  - See the following example.

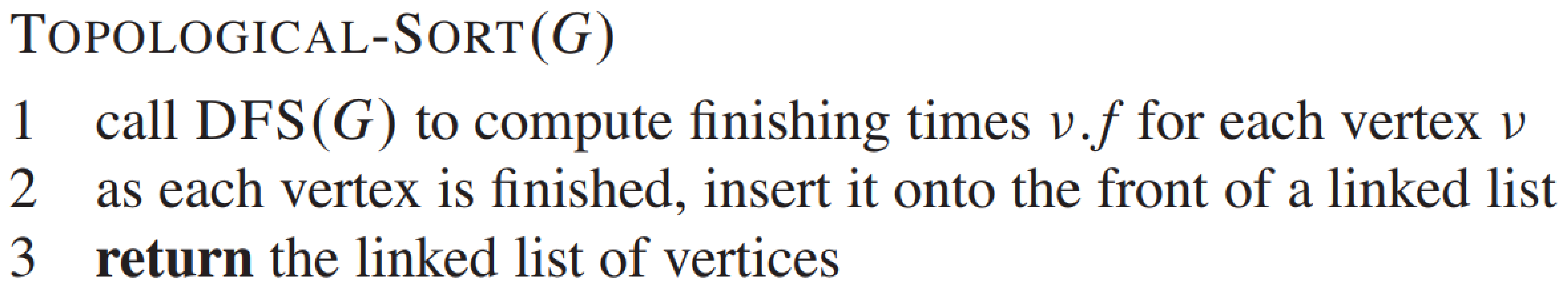
Topological Sort (Example)

- Absent-minded Dr. Wallach (like professor Budimlić before him) has a problem when getting ready to go to work in the morning, he sometimes dresses out of order.
- For example, he might put his shoes on before putting the socks on, so he’ll have to take the shoes off, put the socks on and than the shoes back on.
- There’s also a shirt, tie, belt, shorts, pants, watch and jacket that have to be put on in a certain order.
- The order between different parts of clothing forms a graph: shorts before pants means there’s an edge between shorts and pants, and so on.



Algorithm for Topological Sort

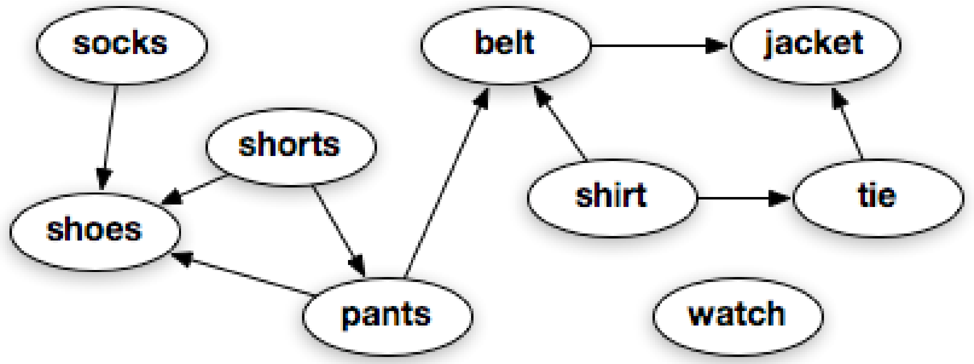
- Algorithm for Topological Sort



```
TOPOLOGICAL-SORT($G$)
1  call DFS($G$) to compute finishing times $v.f$ for each vertex $v$
2  as each vertex is finished, insert it onto the front of a linked list
3  return the linked list of vertices
```

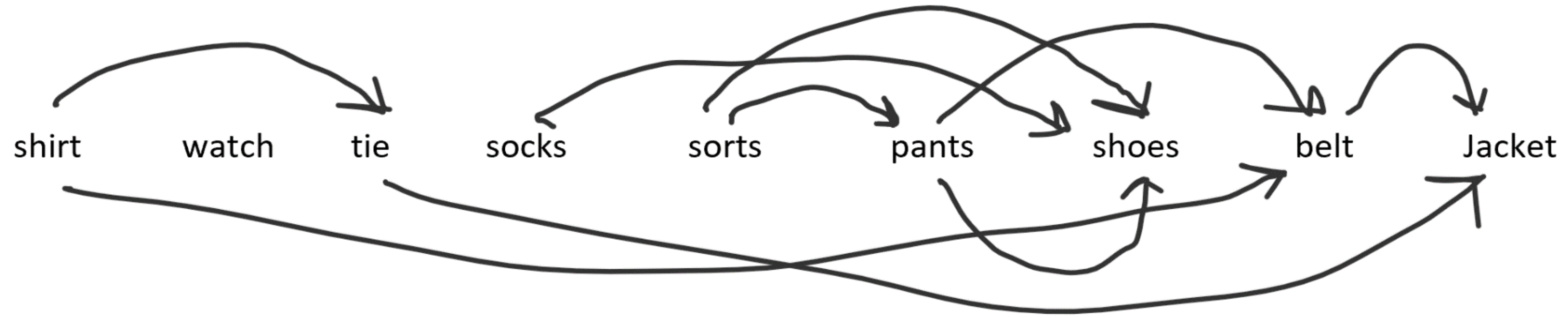
- We can perform a topological sort in time $\Theta(V + E)$ since depth-first search takes $\Theta(V + E)$ time and it takes $O(1)$ time to insert each of the $|V|$ vertices onto the front of the linked list.

Topological Sort



- Example (Continues):

- Let’s do a DFS of the graph above.
- Say, we have the following start/finish times. Shorts: (1/10), pants(2/9), belt(3/6), jacket(4/5), shoes (7/8), socks (11/12), tie(13/14), watch(15/16), shirt(17/18)
- Then, if we arrange the nodes in the decreasing order of finish time.
- This is an order that is sure to get Dr. Wallach dressed without any time-consuming backtracking
- This is called topological sort.
- The topologically sorted DAG as an ordering of vertices along a horizontal line such that all directed edges go from left to right.



SSSPs in DAGs

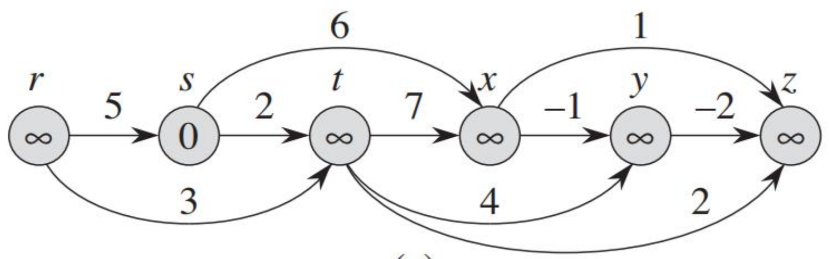
- Now, back to our SSSP problem for the DAGs.
- Following is the algorithm

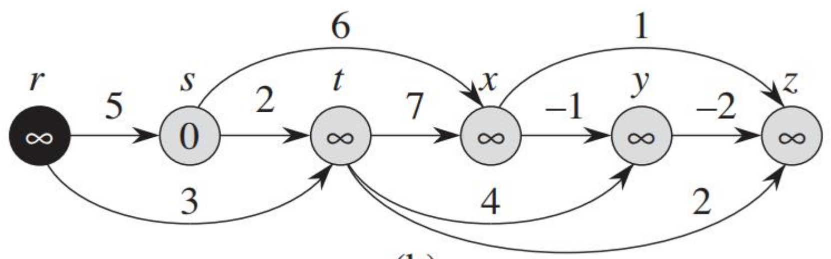
```
DAG-SHORTEST-PATHS($G, w, s$)
1  topologically sort the vertices of $G$
2  INITIALIZE-SINGLE-SOURCE($G, s$)
3  for each vertex $u$, taken in topologically sorted order
4      for each vertex $v \in G.Adj[u]$
5          RELAX($u, v, w$)
```

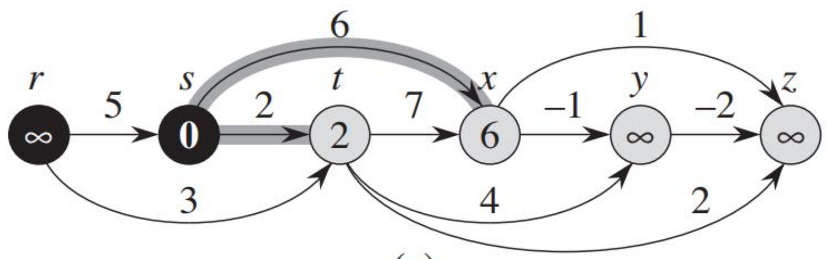
- The topological sort of line-1 takes, $\Theta(V + E)$ time
- The call of INITIALIZE-SINGLE-SOURCE in line-2 takes, $\Theta(V)$ time.
- The **for** loop of lines 3–5 makes one iteration per vertex.
- The **for** loop of lines 4–5 relaxes each edge exactly once.
- Because each iteration of the inner **for** loop takes, $\Theta(1)$ time, the total running time is $\Theta(V + E)$, which is linear in the size of an adjacency-list representation of the graph.

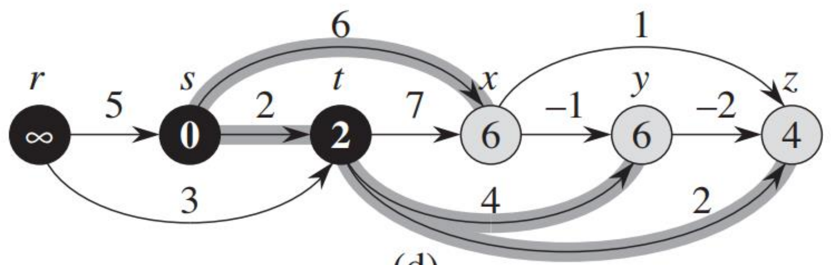
SSSPs in DAGs

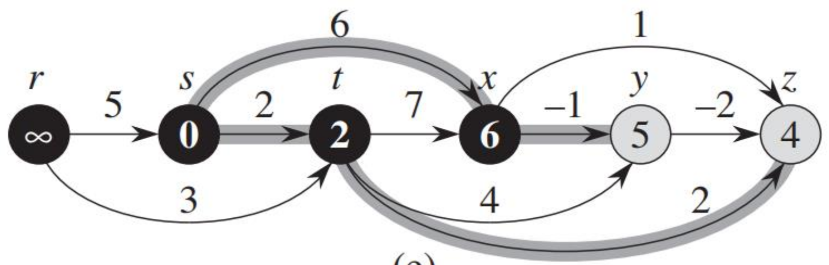
- In a topologically sorted list of vertices, all edges will go from left to right.
- Once all outgoing edges at a node u have been relaxed, u will never be revisited.
- Since we process the nodes in topologically-sorted order, the nodes at 1 hop from s will be finished before those at 2 hops, etc.
- By the path-relaxation property, all shortest paths will be found.

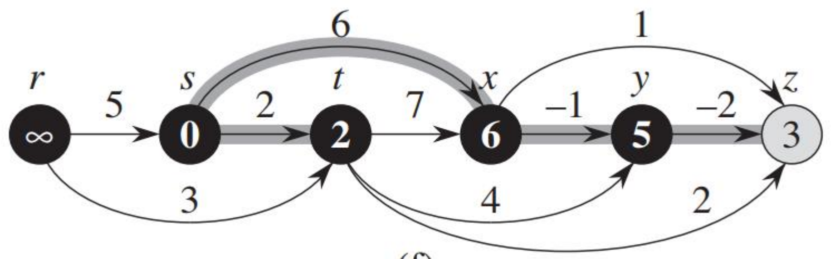

6
1
r
s
t
x
y
z
5
2
7
–1
–2
∞
0
∞
∞
∞
∞
3
4
2
(a)

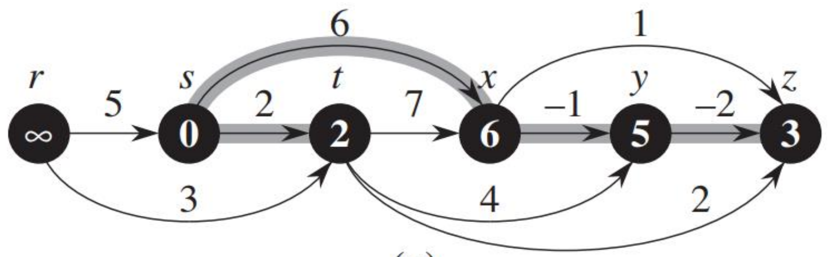

6
1
r
s
t
x
y
z
5
2
7
–1
–2
∞
0
∞
∞
∞
∞
3
4
2
(b)


6
1
r
s
t
x
y
z
5
2
7
–1
–2
∞
0
2
6
∞
∞
3
4
2
(c)


6
1
r
s
t
x
y
z
5
2
7
–1
–2
∞
0
2
6
6
4
3
4
2
(d)


6
1
r
s
t
x
y
z
5
2
7
–1
–2
∞
0
2
6
5
4
3
4
2
(e)


6
1
r
s
t
x
y
z
5
2
7
–1
–2
∞
0
2
6
5
3
3
4
2
(f)


6
1
r
s
t
x
y
z
5
2
7
–1
–2
∞
0
2
6
5
3
3
4
2
(g)

Shortest path for DAG - An example application

- Determining critical paths in a PERT chart.
    - A **PERT chart** is a project management tool used to schedule, organize, and coordinate tasks within a project.
- Edges represent jobs to be performed, and edge weights represent the times required to perform particular jobs.
- A path through this directed acyclic graph (DAG) represents a sequence of jobs that must be performed in a particular order.
- A critical path is a longest path through the DAG, corresponding to the longest time to perform any sequence of jobs.
- The weight of a critical path provides a lower bound on the total time to perform all the jobs.

Dijkstra’s algorithm

- Edsger Wybe Dijkstra.
- May 11, 1930 – August 6, 2002
- Dutch computer scientist from Netherlands
- Received the 1972 A. M. Turing Award, widely considered the most prestigious award in computer science
- Known for his many essays on programming

Dijkstra’s algorithm

- Applications of Dijkstra’s Algorithm :
  - Dijkstra’s algorithm is applied to automatically find directions between physical locations.
  - In a networking or telecommunication applications, Dijkstra’s algorithm has been used for solving the min-delay path problem (which is the shortest path problem).
    - For example in data network routing, the goal is to find the path for data packets to go through a switching network with minimal delay.
  - It is also used for solving a variety of shortest path problems arising in plant and facility layout, robotics, transportation, and VLSI design

Dijkstra’s algorithm

- How will you go from your home to the nearest railway station?
- This is a Single-Source Shortest Path Problem in a Graph G.
    - Both directed and undirected graphs
    - All edges must have nonnegative weights
    - Graph must be connected.

- Dijkstra’s algorithm solves the single-source shortest-paths problem on a weighted, directed graph G = (V, E) for the case in which all edge weights are nonnegative.
    - Therefore, we assume that $w(u, v) \geq 0$ for each edge $(u, v) \in E$.

Dijkstra’s algorithm

- Just like the Prim’s algorithm: the algorithm partition the nodes into two sets, S and $V - S$.
  - The set S maintains the vertices whose final shortest-path weights from the source (s) have already been determined.
  - The algorithm repeatedly selects the vertex $u \in V - S$ with minimum shortest path estimate, adds u to S, and relaxes all edges leaving u.

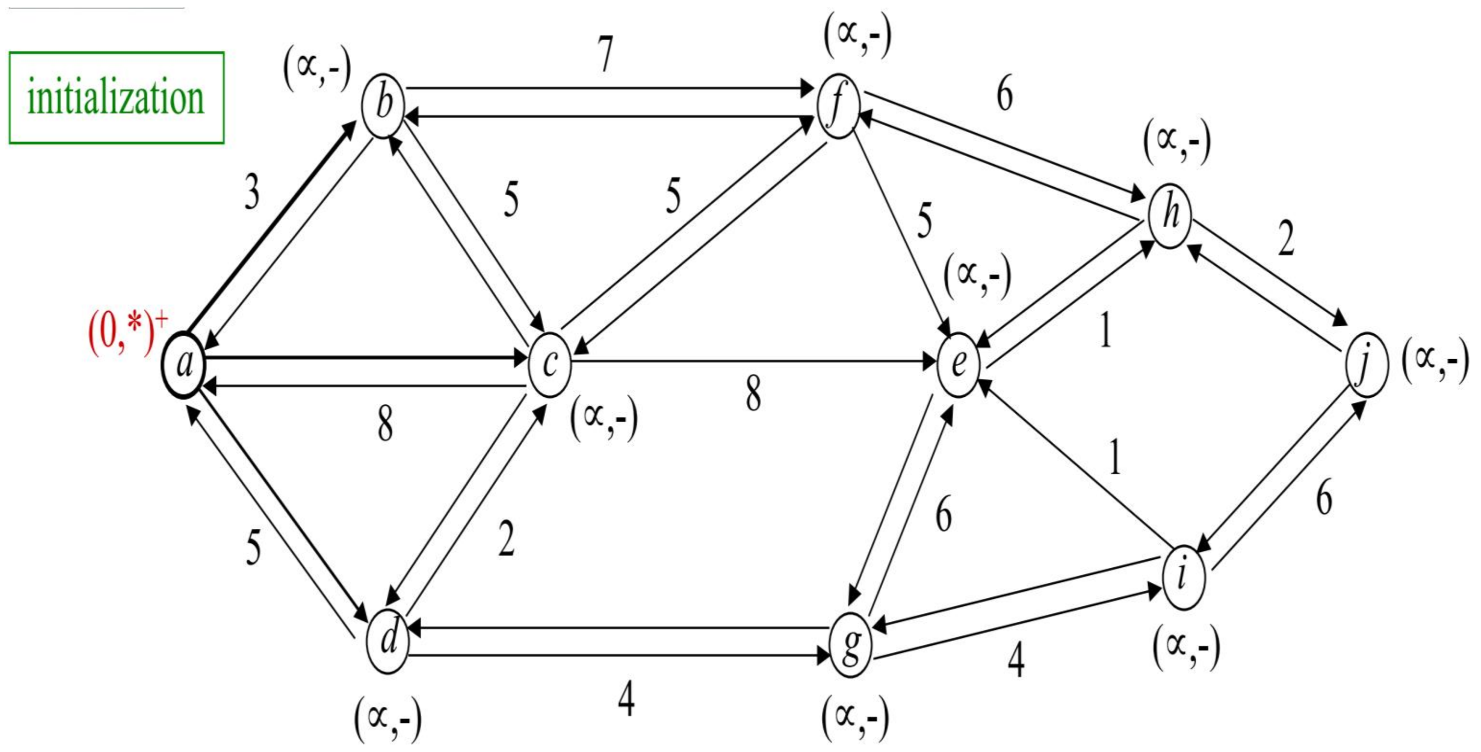
- It uses a min-priority queue Q of vertices, keyed by their d values.
- It runs like the breadth-first algorithm (explores breadth first).

Dijkstra’s algorithm - Algorithm

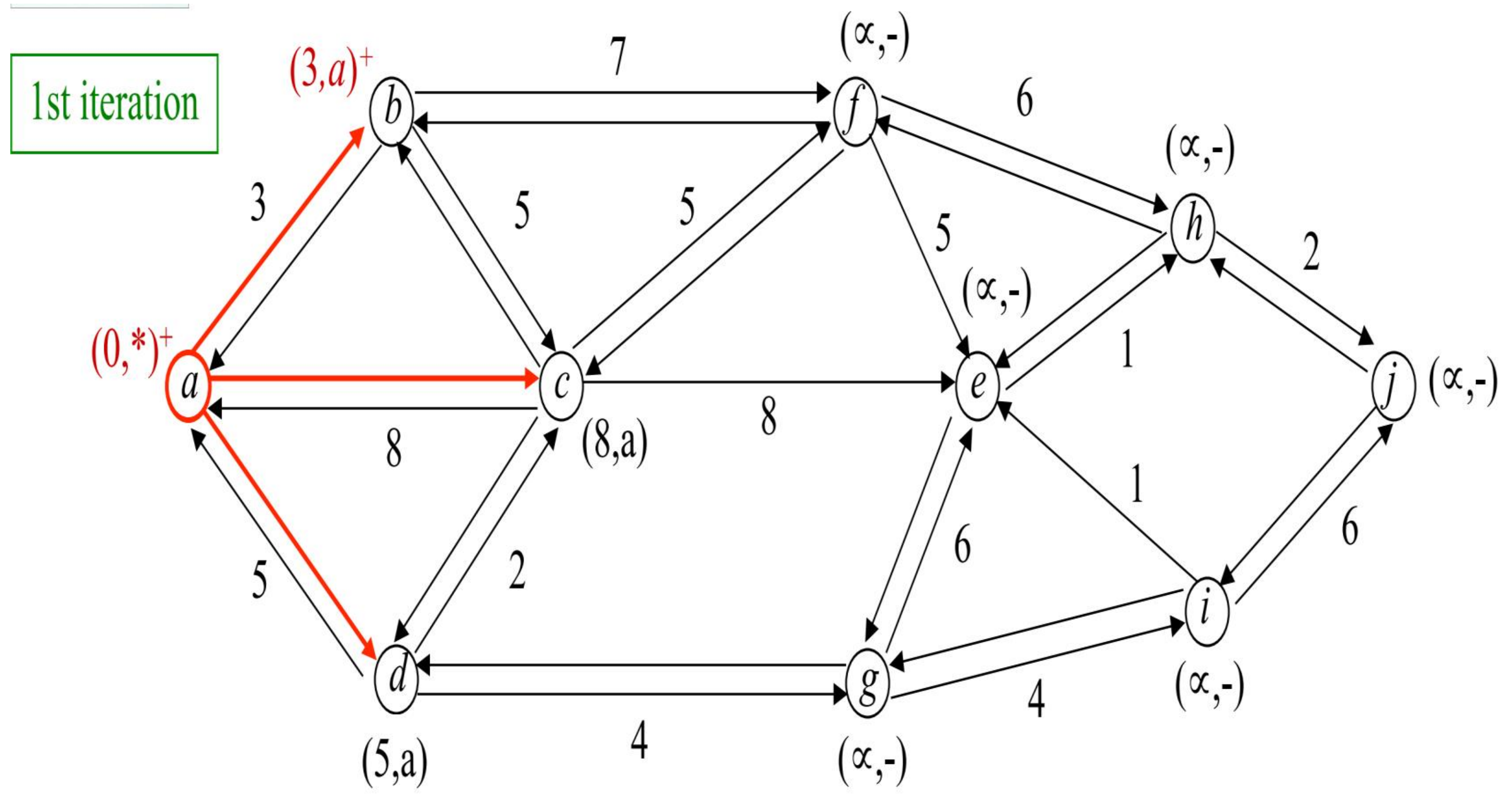
```
DIJKSTRA($G, w, s$)
1  INITIALIZE-SINGLE-SOURCE($G, s$)
2  $S = \emptyset$
3  $Q = G.V$
4  while $Q \neq \emptyset$
5      $u$ = EXTRACT-MIN($Q$)
6      $S = S \cup \{u\}$
7      for each vertex $v \in G.Adj[u]$
8          RELAX($u, v, w$)
```

- Line 1 initializes the d and π values.
- Line 2 initializes the set S to the empty set
- Line 3 initializes the min-priority queue Q to contain all the vertices in V.
- Each time through the **while** loop of lines 4–8, line 5 extracts a vertex u from $Q = V - S$ and line 6 adds it to set S.
- Lines 7–8 relax each edge (u, v) leaving u, thus updating the estimate $v.d$ and the predecessor $v.\pi$ if we can improve the shortest path to v found so far by going through u.
- The **while** loop of lines 4–8 iterates exactly $|V|$ times.

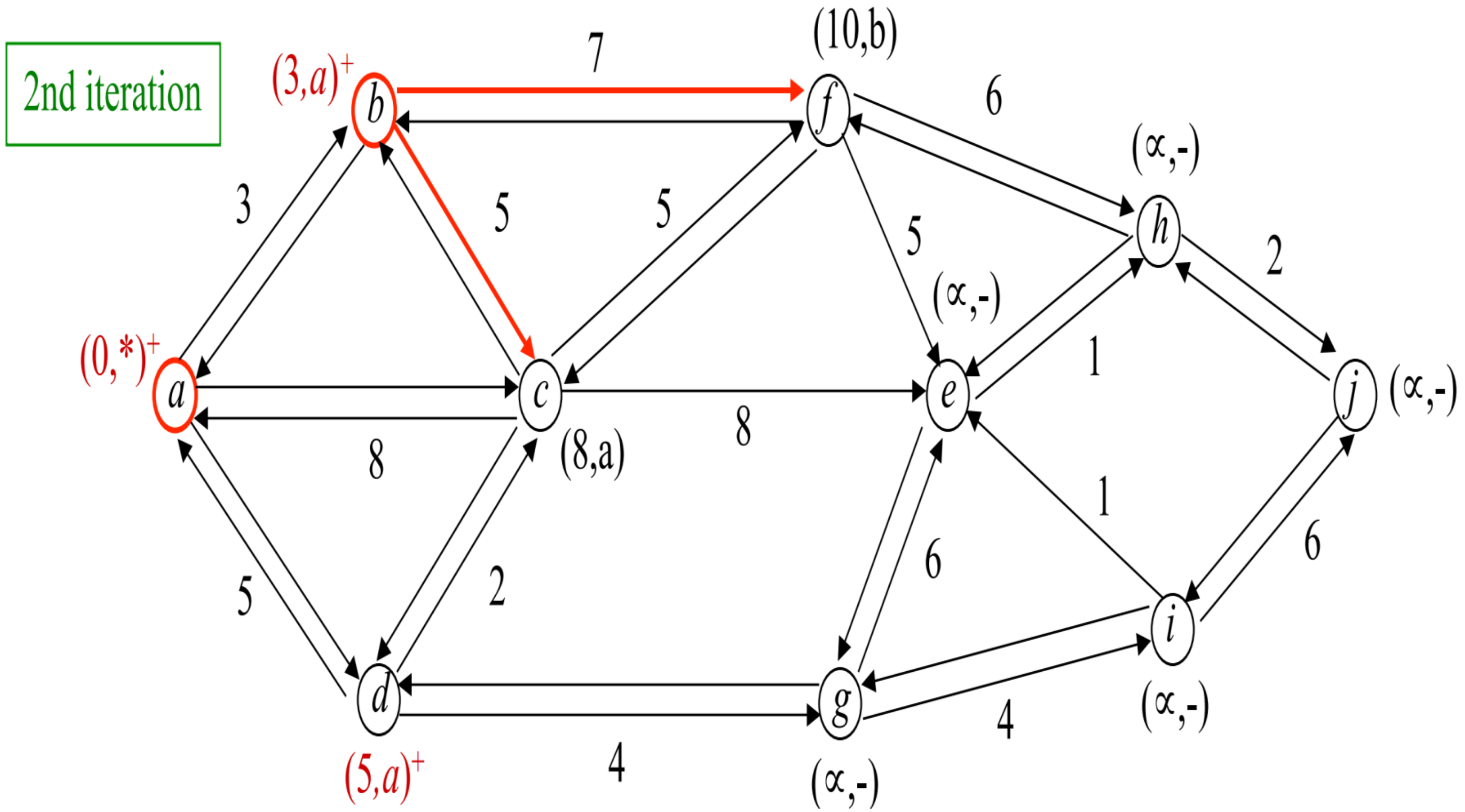
Dijkstra’s algorithm – Example (1)



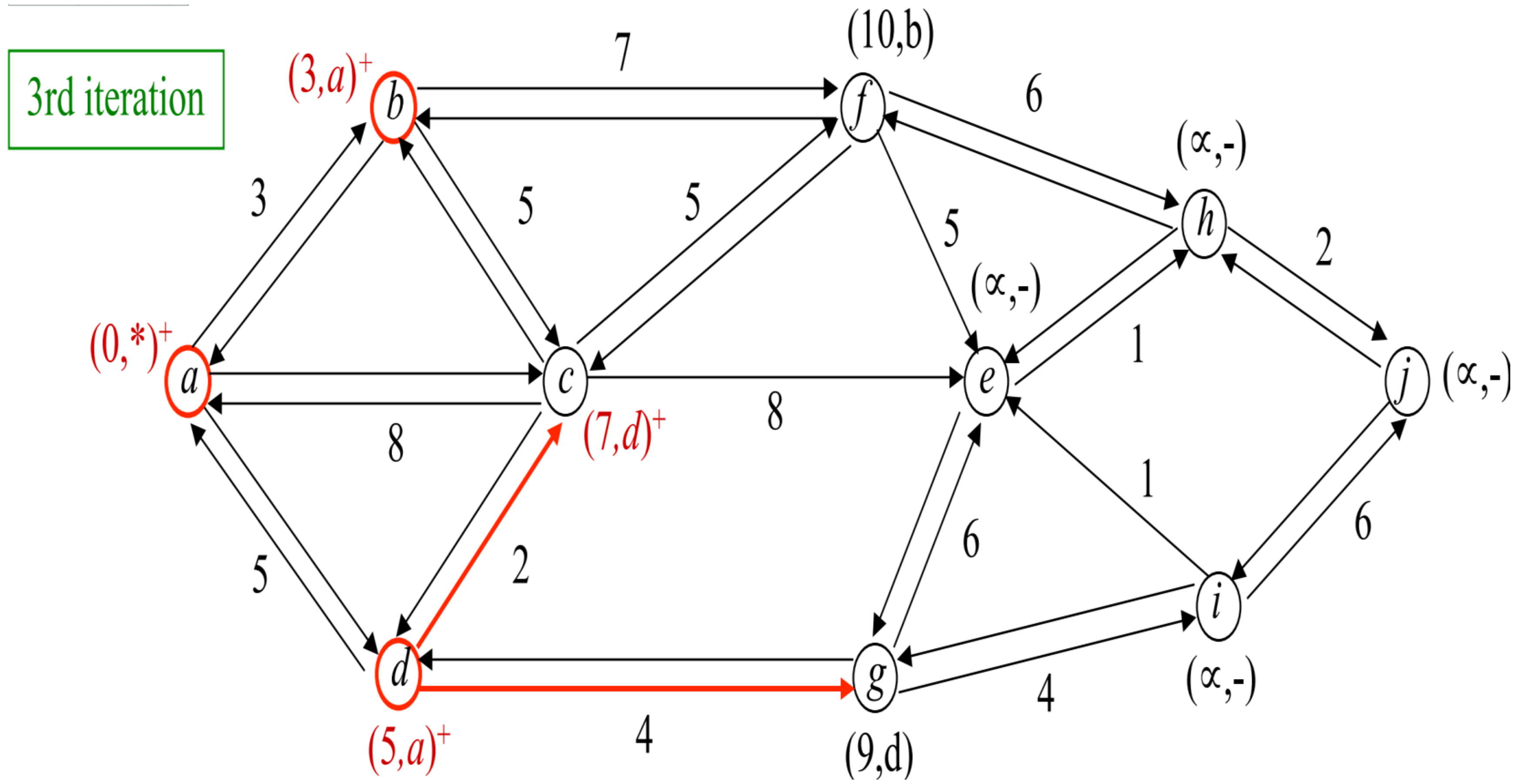
Dijkstra’s algorithm – Example (1)



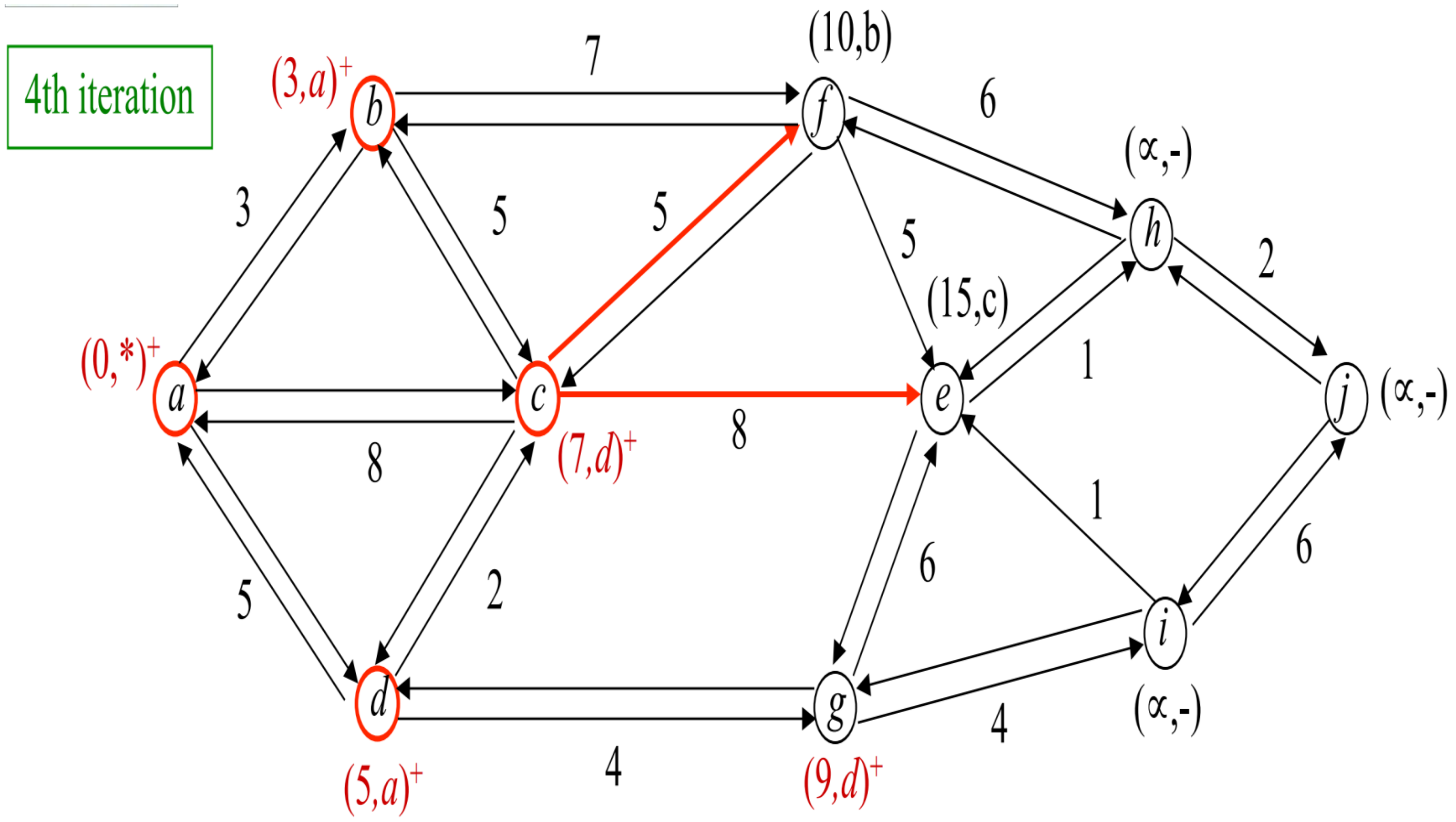
Dijkstra’s algorithm – Example (1)



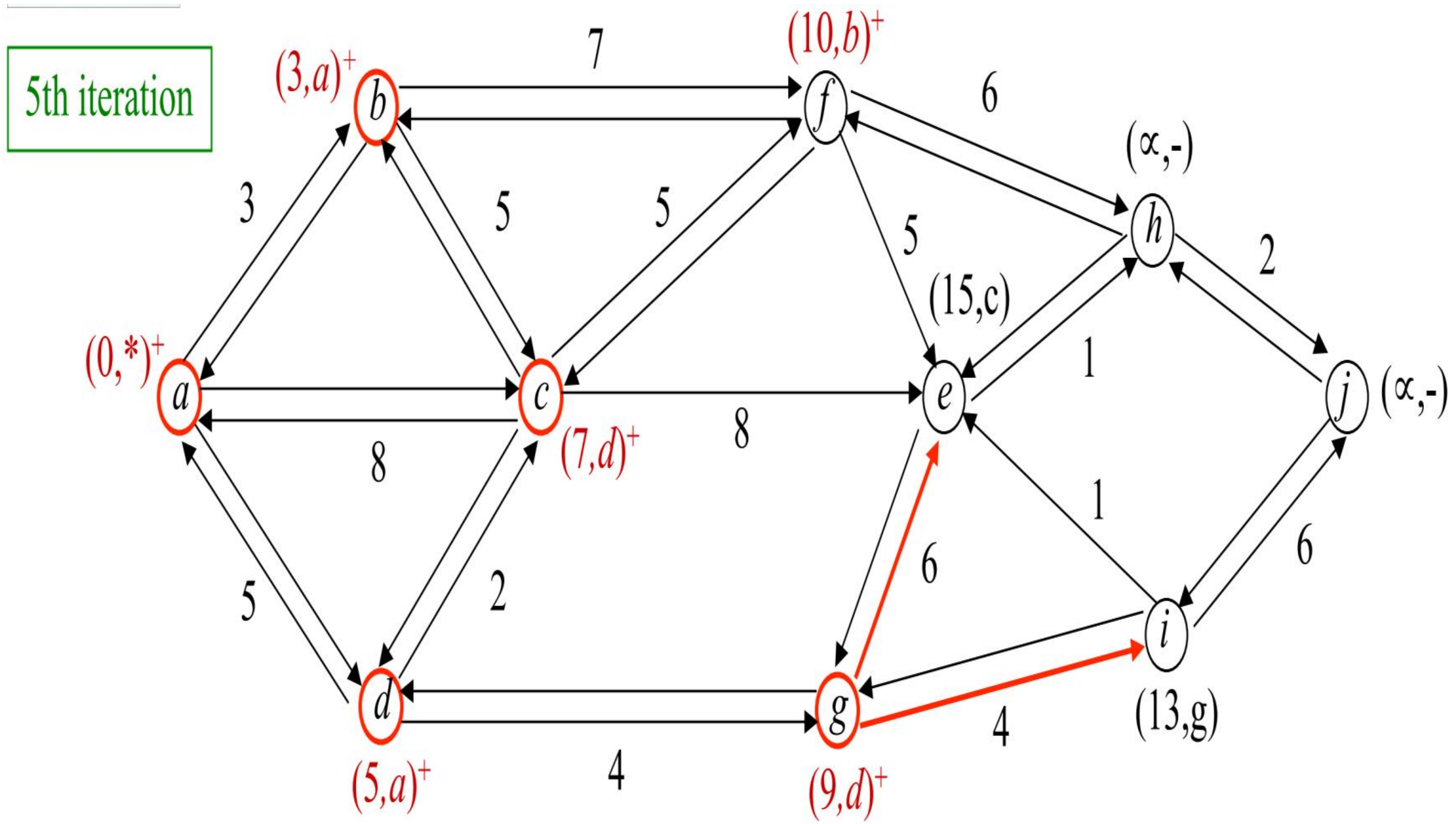
Dijkstra's algorithm – Example (1)



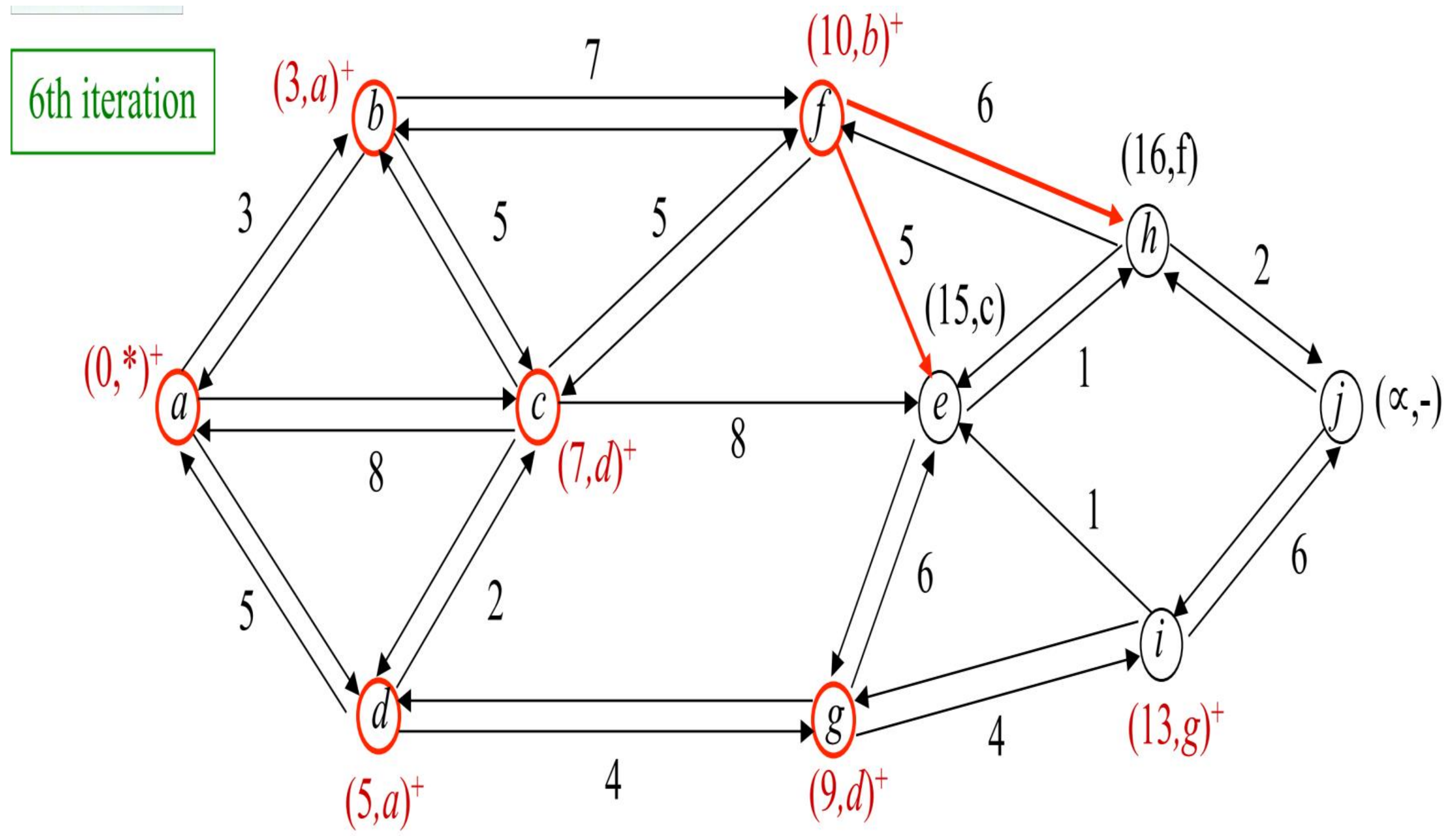
Dijkstra’s algorithm – Example (1)



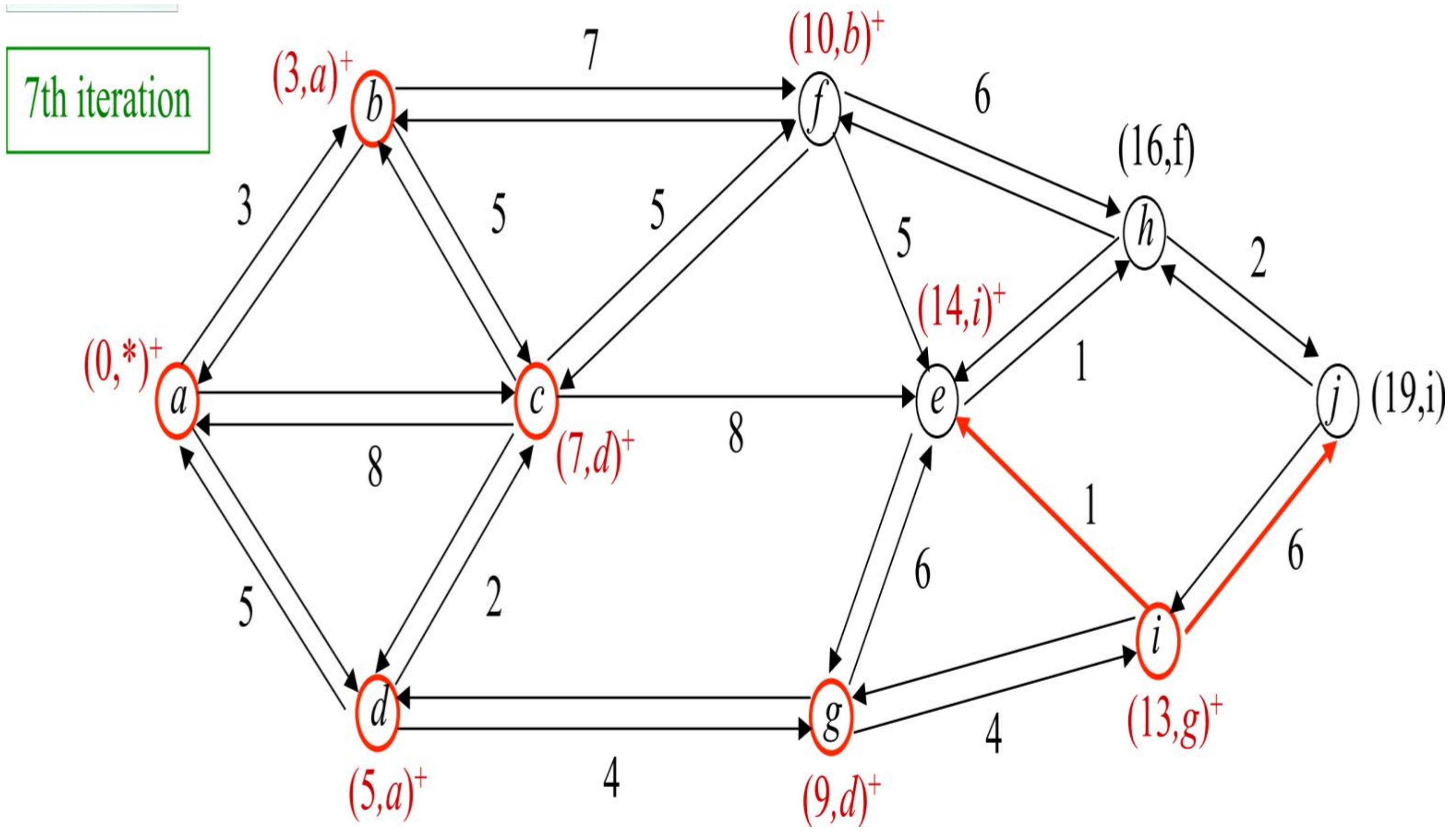
Dijkstra’s algorithm – Example (1)



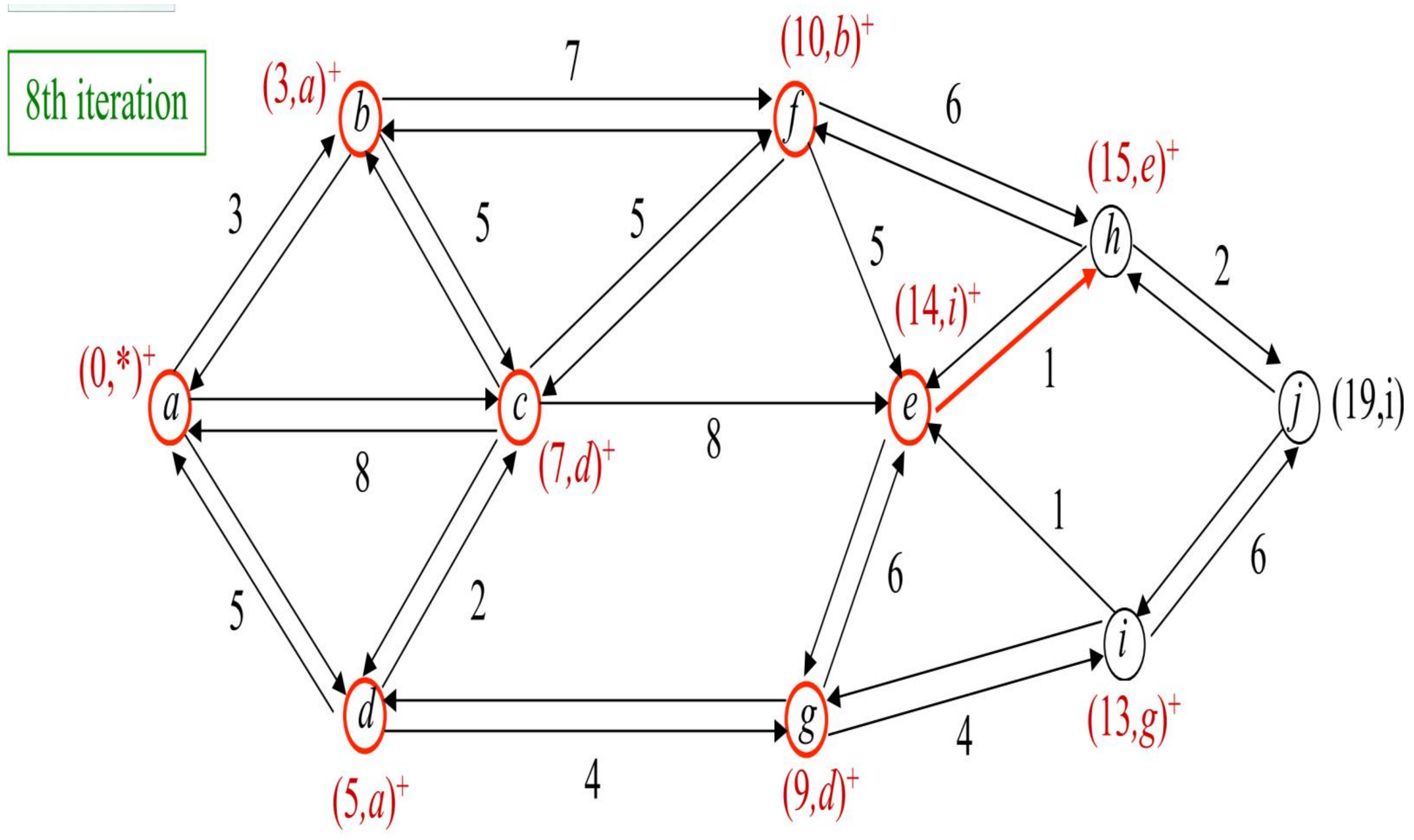
Dijkstra’s algorithm – Example (1)



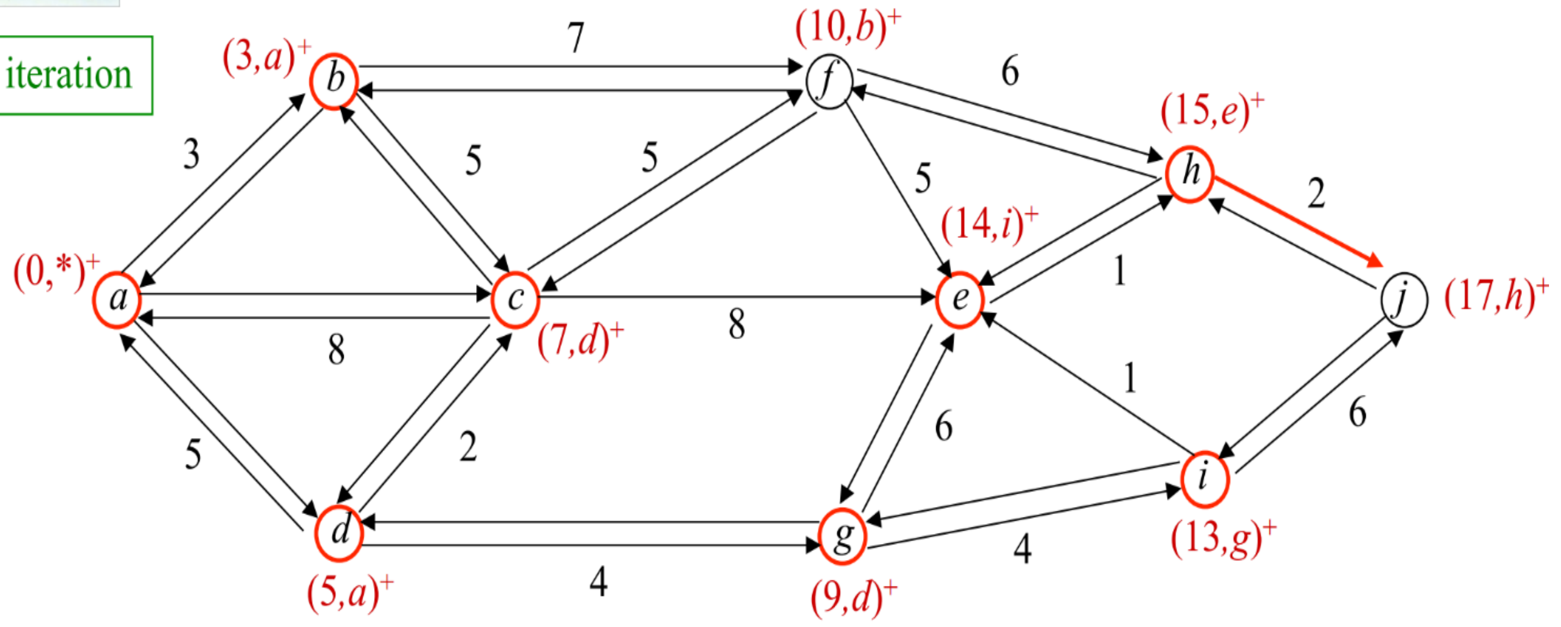
Dijkstra’s algorithm – Example (1)



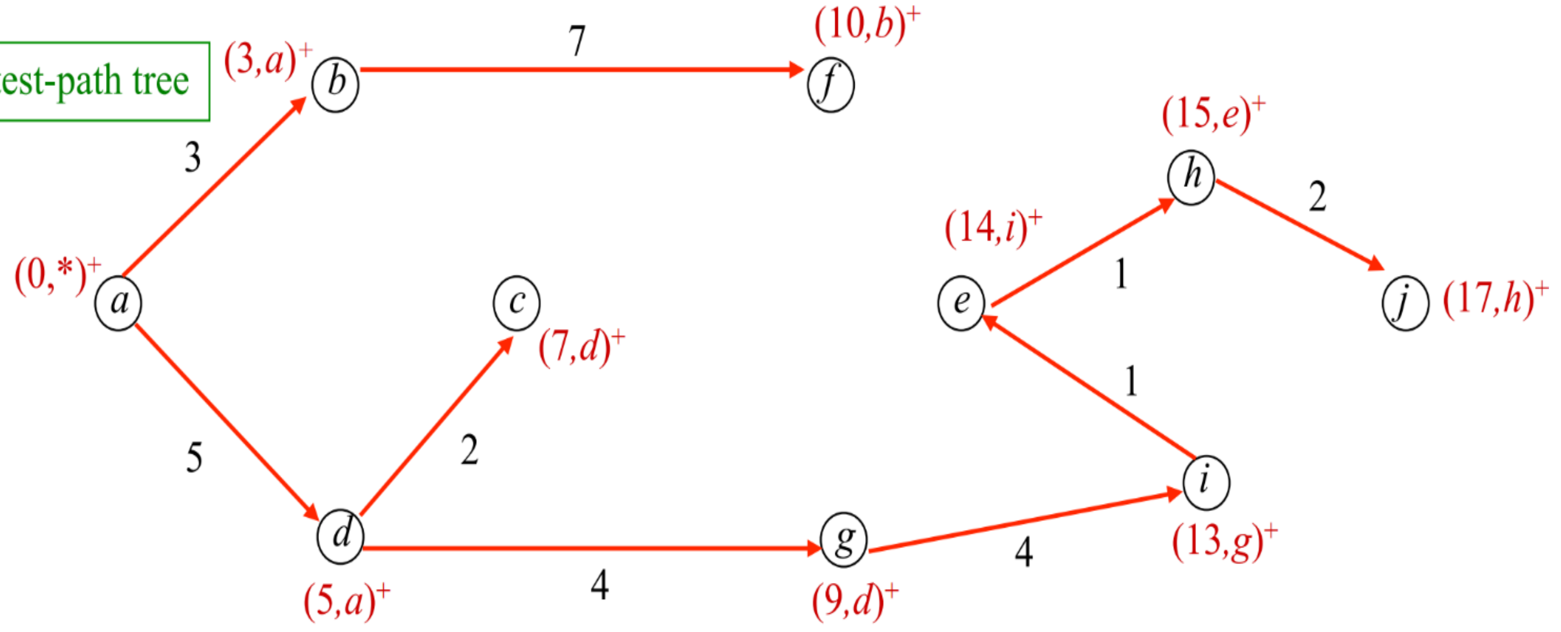
Dijkstra’s algorithm – Example (1)



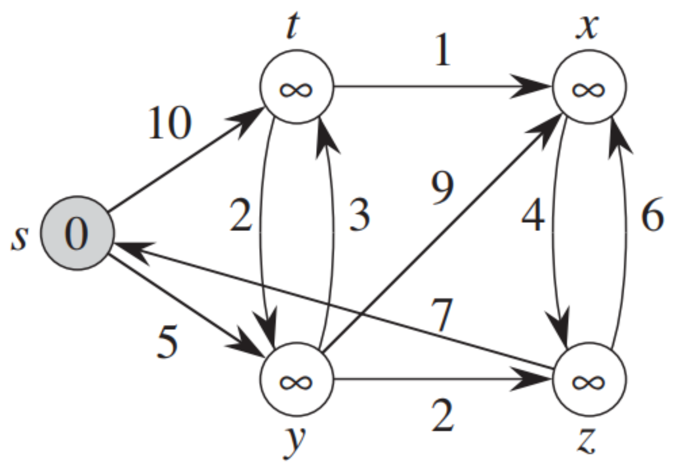
9th iteration



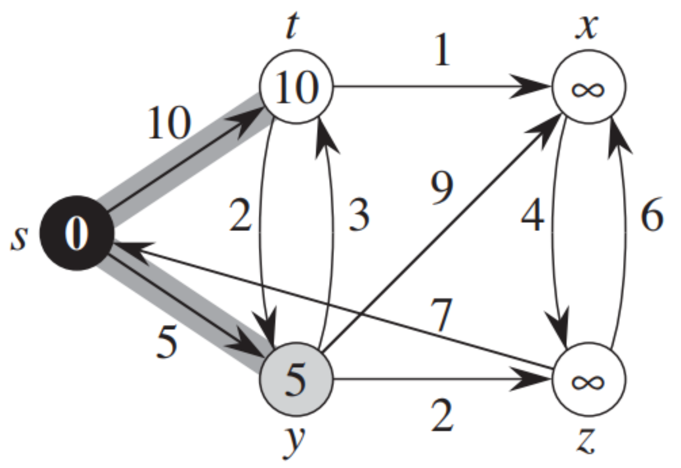
Shortest-path tree



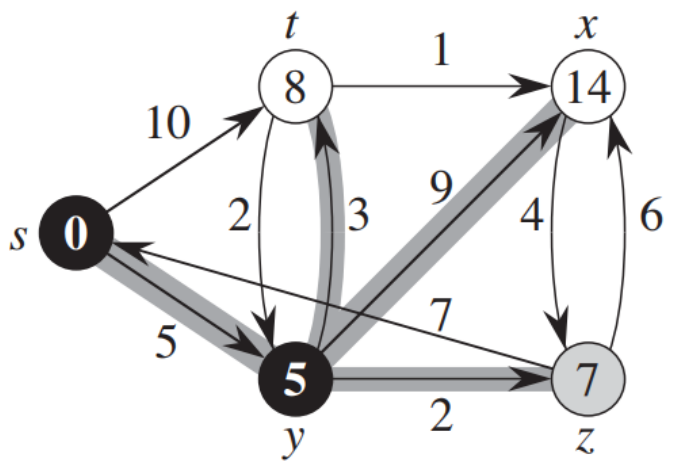
Dijkstra’s algorithm – Example (2)



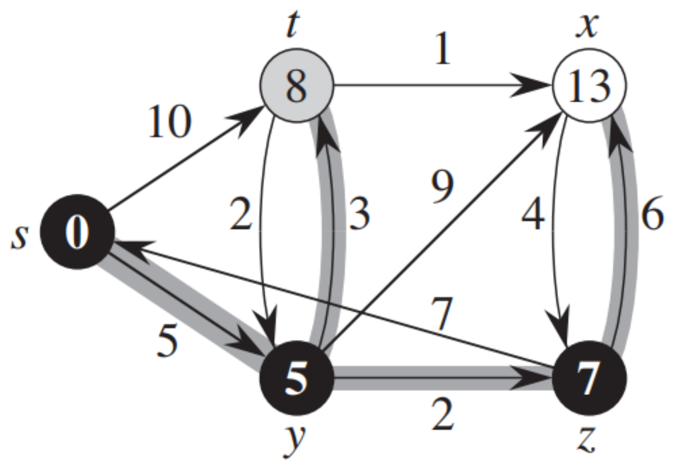
(a)



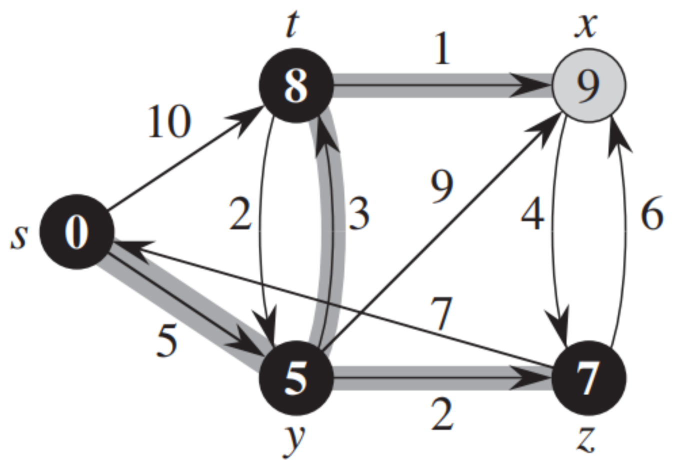
(b)



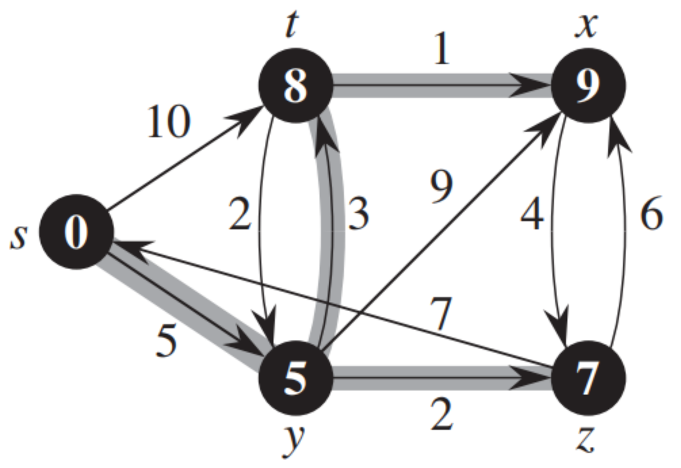
(c)



(d)



(e)



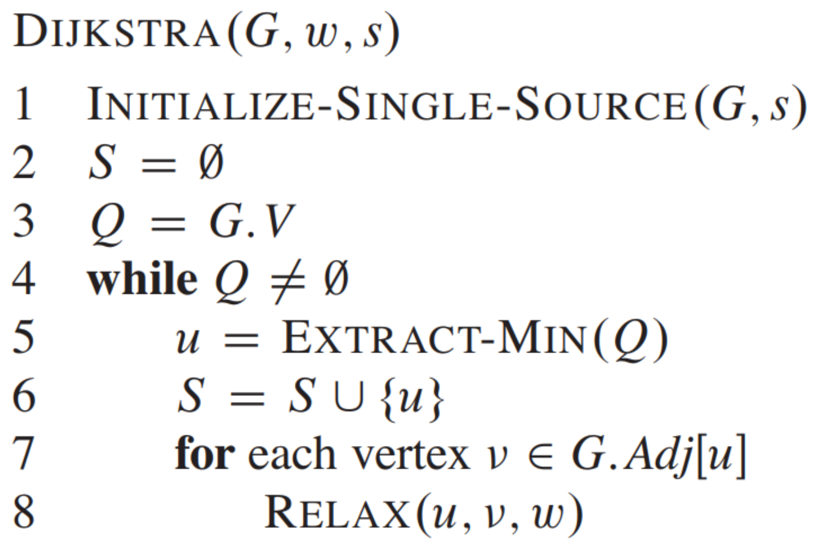
(f)

Running time of Dijkstra’s algorithm

- O(V . time for EXTRACT-MIN) = $O(V \lg V)$, if priority queue implemented using Fibonacci heap.

- Thus, the total running time = $O(V \lg V + E)$

- The running time of Dijkstra’s algorithm is lower than that of the Bellman-Ford algorithm



```
DIJKSTRA$(G, w, s)$
INITIALIZE-SINGLE-SOURCE$(G, s)$
$S = \emptyset$
$Q = G.V$
while $Q \neq \emptyset$
    $u$ = EXTRACT-MIN$(Q)$
    $S = S \cup \{u\}$
    for each vertex $v \in G.Adj[u]$
        RELAX$(u, v, w)$
```

Selection in Dijkstra's Algorithm

- The question that we can ask is $\rightarrow$ what is the best order in which to process vertices, so that the estimates are guaranteed to converge to the true distances.
  - That is, how does the algorithm select which vertex among the vertices of $V - S$ to process next.

- Answer:
  - We use a greedy algorithm.
  - For each vertex u in $(V - S)$, we have computed a distance estimate d[u].
  - The next vertex processed is always a vertex $u \in (V - S)$ for which $d[u]$ is minimum, that is, we take the unprocessed vertex that is closest (by our estimate)

- **Note:** As the Dijkstra's algorithm always chooses the "lightest" or "closest" vertex in $V - S$ to add to set S, we say that it uses a greedy strategy.

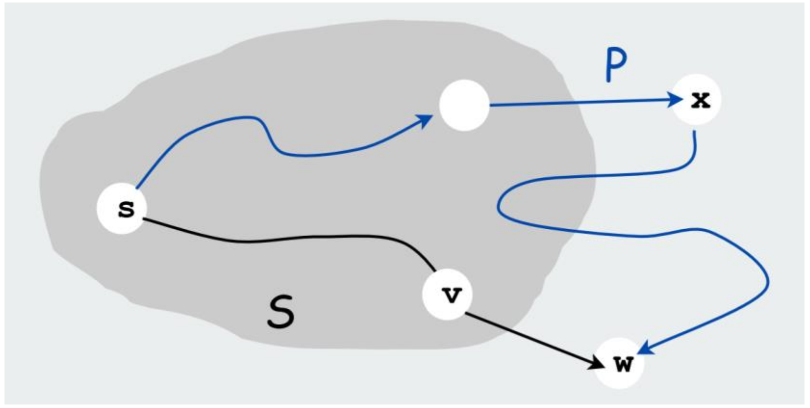
Dijkstra’s algorithm

- Greedy algorithms always make choices that currently seem the best
    - Short-sighted – no consideration of long-term or global issues
    - Locally optimal does not always mean globally optimal
- In Dijkstra’s case: we choose the least cost node, but what if there is another path through other vertices that is cheaper?
    - Is this possible?
- We need to prove that this never happens… provided if all edge weights are positive

Dijkstra's algorithm

- **Claim:** At the start of each iteration of the **while** loop (of lines 4–8, in the algo.), $v.d = \delta(s, v)$ for each vertex $v \in S$. That is, we know the correct shortest paths of all nodes in the gray cloud, shown in the figure.

- Assume that in the next iteration, let w be vertex that the Dijkstra's algorithm will add to S.
  - Then, we need to prove that, when we add w to S $\rightarrow$ We get the shortest path to node w, i.e., $w.d = \delta(s, w)$.
  - Or, there can not be any other shortest path to w.

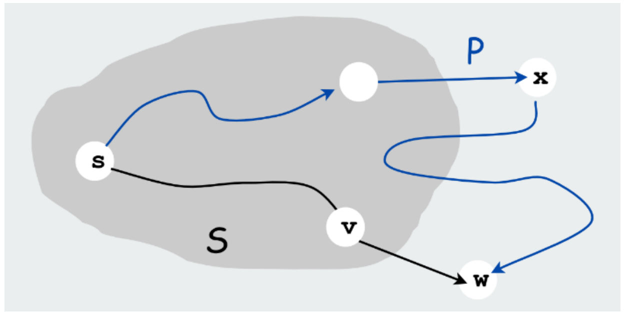


- Proof (An intuitive sketch):
  - Let w be next vertex added to S.
  - Let P' be the $s \rightarrow w$ path through v.
  - Consider any other path P from $s \rightarrow w$, and let x be first node on path outside S.
  - For P to be the shortest path (to node w), the weight of path P must be less than or equal to the weight of the path P'.
    - That is, path P must be at least as long P'.
  - However, our claim is that, P is already longer than P' as soon as it reaches x. **(WHY?)**
    - Because, …

Dijkstra’s algorithm

- Lemma: When a vertex w is added to S (i.e., de-queued from the queue), $d[w] = \delta(s, w)$.
- Proof is by induction on the number of nodes in set S
    - Base case: Initial cloud is just the source with shortest path 0
    - Inductive hypothesis: Assume that the shortest paths to all k-1 vertices in the set S is known.
    - Inductive step: choose the least cost node w from the set $V - S$.
        - Show that $d[w] = \delta(s, w)$. That is, the algorithm yields the shortest path to w (as discussed in the previous slide).

Dijkstra’s algorithm (Discussion)

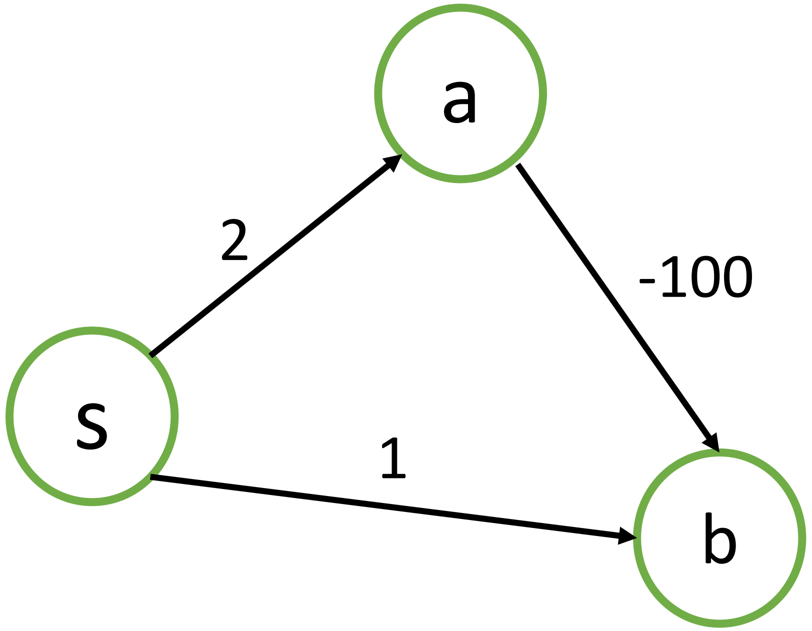


- Questions: Will the Dijkstra’s algorithm work if we have negative edge weights (not considering –ve cycles) in the graph? (WHY or Why NOT! ☺)
  - Notice, Dijkstra’s algorithm, we are not revisiting any vertex again.
  - That is, once the algorithm relaxes all edges, say (u, v), coming out from node u, it will not visit the vertex u again.
    - Then… what is the difference between Dijkstra’s algorithm and Bellman-Ford algorithm? We will get back to this, later.

  - Now, assume that in this process, say in i^{th} iteration, we get a shortest path P' to w (as discussed in the previous slides).
  - Then, is it possible that (if we allow –ve edge weights), then can find a better path to w.
  - Consider the next example.

Dijkstra’s algorithm (Discussion)

- Try to run the Dijkstra’s algorithm for the graph, and argue why it fails?



Dijkstra’s algorithm (Discussion)

- Questions: What are the difference (algorithmically) between the Dijkstra’s algorithm and Bellman-Ford algorithm?