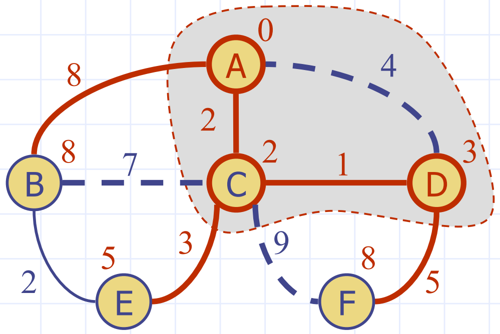
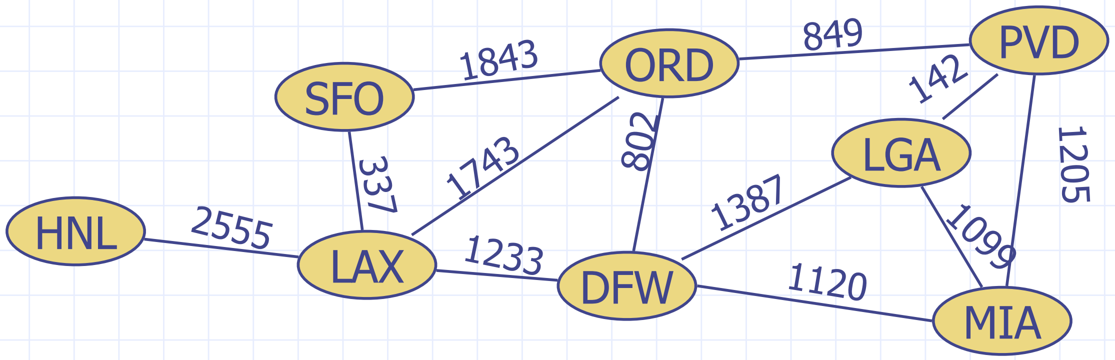
Shortest Paths



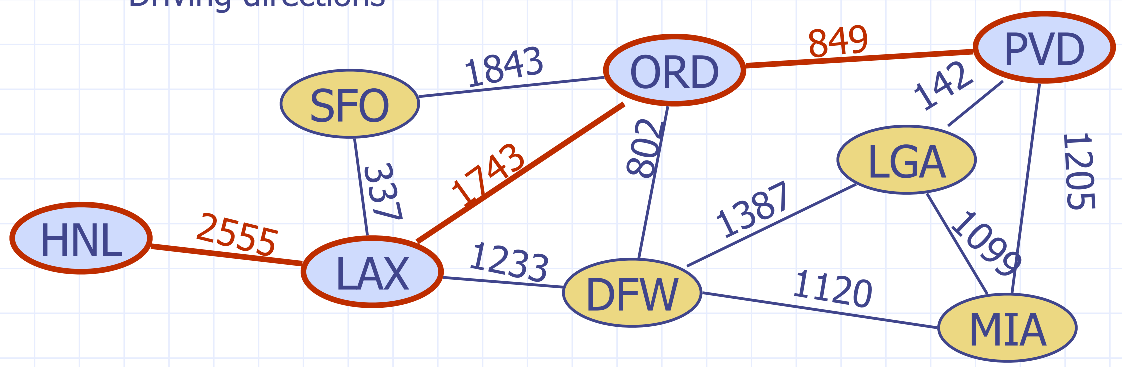
Weighted Graphs

- In a weighted graph, each edge has an associated numerical value, called the weight of the edge
- Edge weights may represent, distances, costs, etc.
- Example:
  - In a flight route graph, the weight of an edge represents the distance in miles between the endpoint airports



Shortest Paths

- Given a weighted graph and two vertices u and v, we want to find a path of minimum total weight between u and v.
  - Length of a path is the sum of the weights of its edges.
- Example:
  - Shortest path between Providence and Honolulu
- Applications
  - Internet packet routing
  - Flight reservations
  - Driving directions



Shortest Path Properties

Property 1:

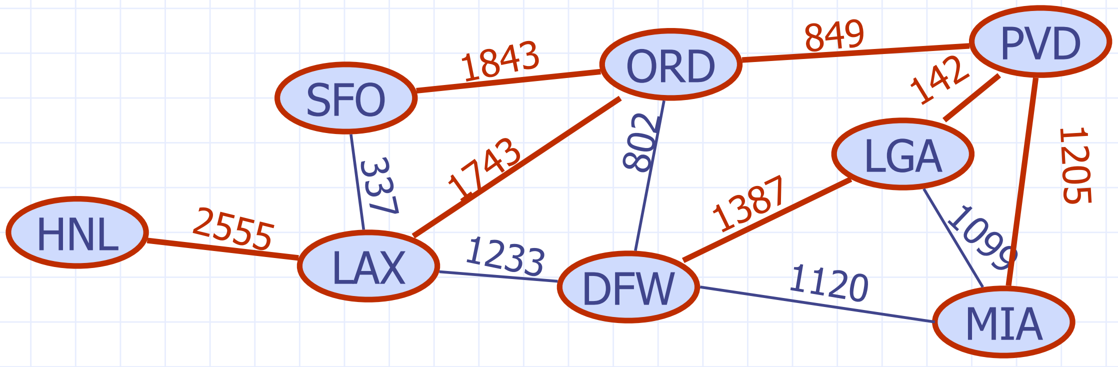
A subpath of a shortest path is itself a shortest path

Property 2:

There is a tree of shortest paths from a start vertex to all the other vertices

Example:

Tree of shortest paths from Providence



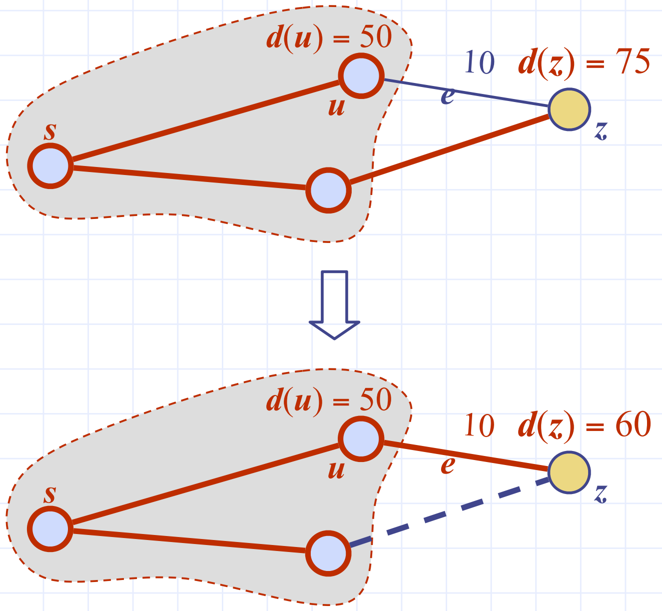
Dijkstra's Algorithm

- The distance of a vertex v from a vertex s is the length of a shortest path between s and v
- Dijkstra's algorithm computes the distances of all the vertices from a given start vertex s
- Assumptions:
  - the graph is connected
  - the edge weights are nonnegative
- We grow a "cloud" of vertices, beginning with s and eventually covering all the vertices
- We store with each vertex v a label $d(v)$ representing the distance of v from s in the subgraph consisting of the cloud and its adjacent vertices
- At each step
  - We add to the cloud the vertex u outside the cloud with the smallest distance label, $d(u)$
  - We update the labels of the vertices adjacent to u

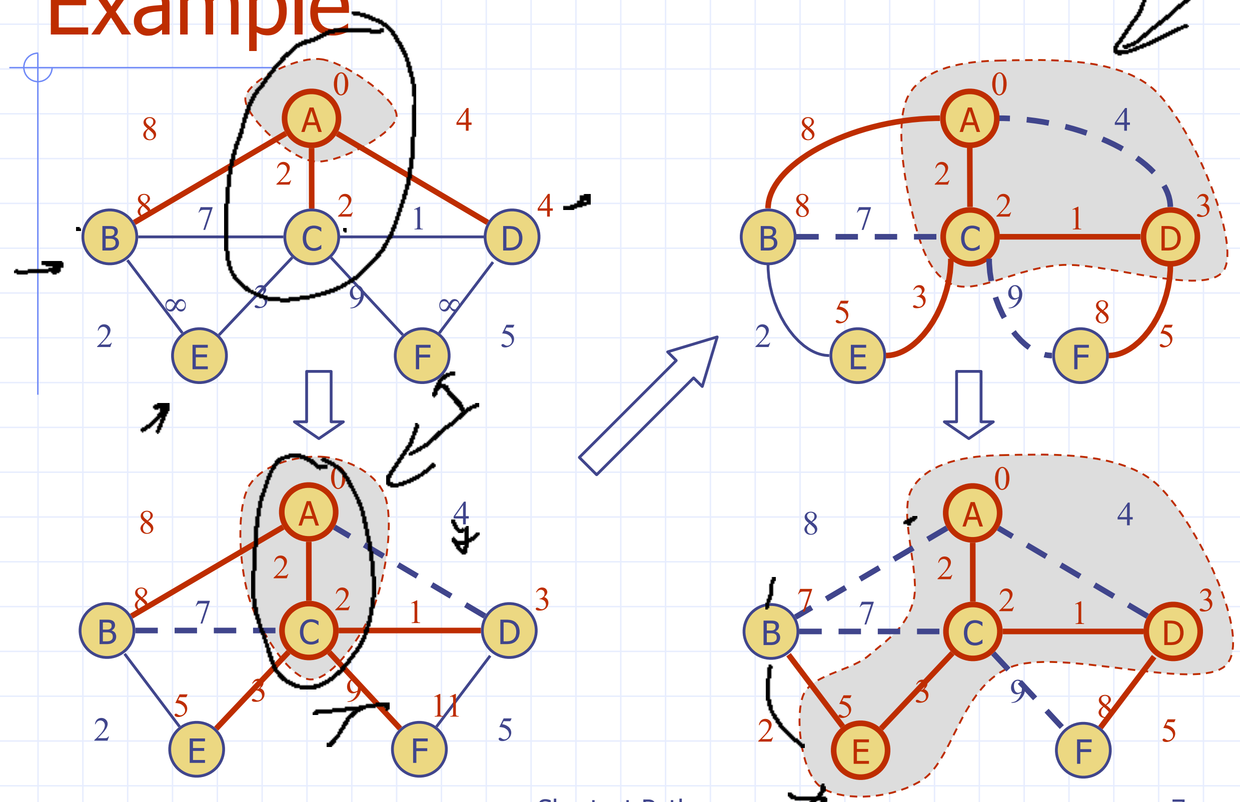
Edge Relaxation

- Consider an edge $e = (u,z)$ such that
  - u is the vertex most recently added to the cloud
  - z is not in the cloud
- The relaxation of edge e updates distance $d(z)$ as follows:

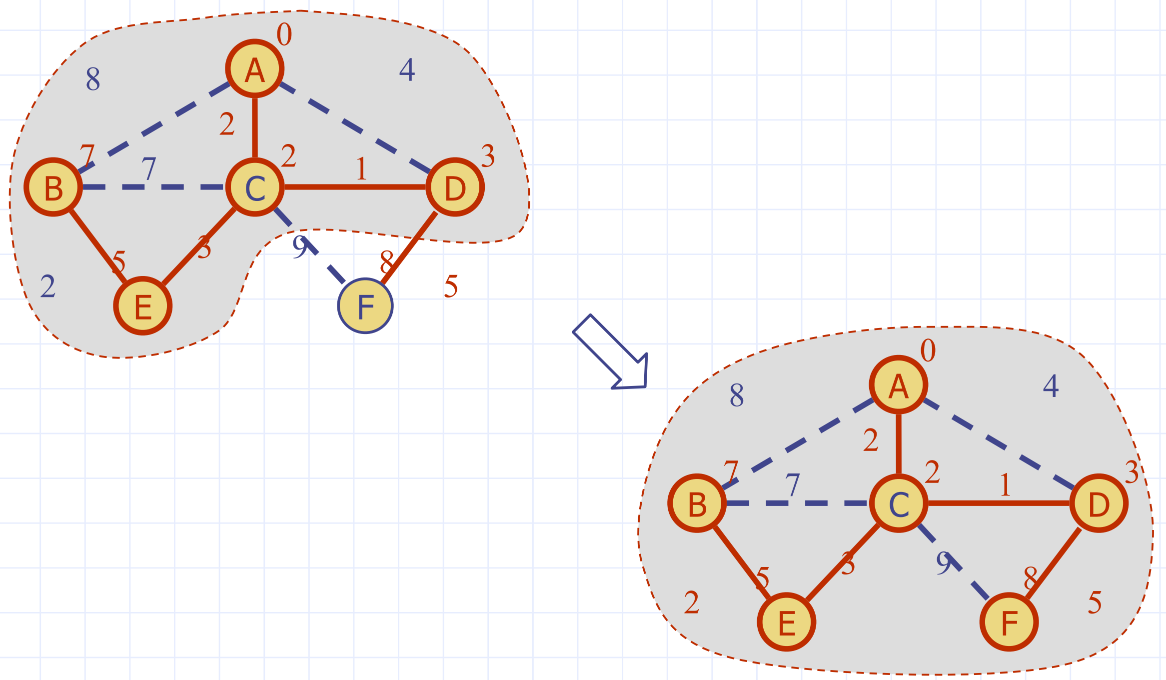
$$d(z) \leftarrow \min\{d(z), d(u) + weight(e)\}$$



Example

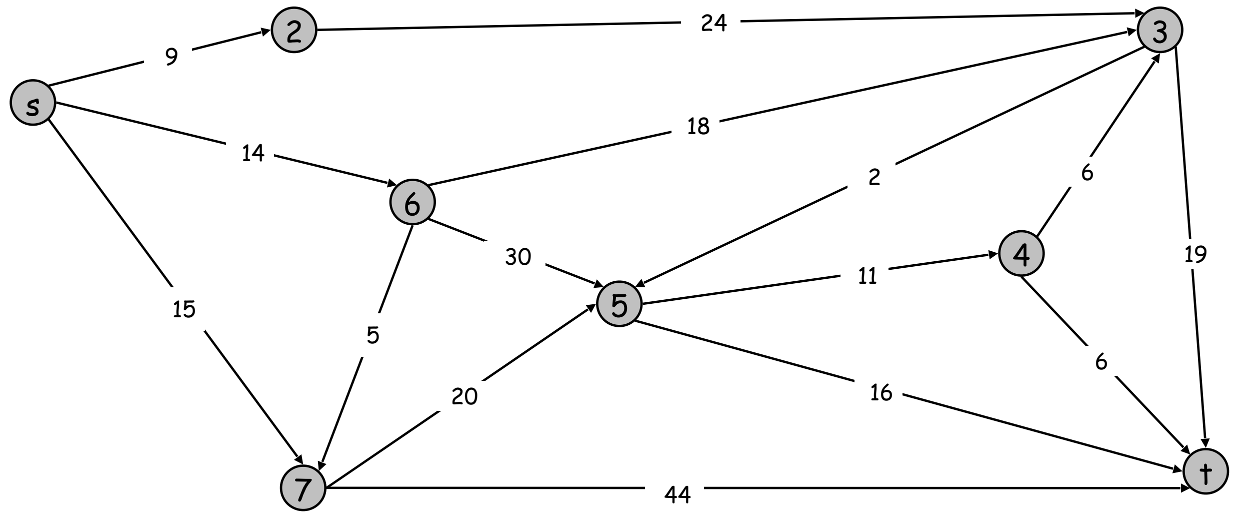


Example (cont.)



Dijkstra's Shortest Path Algorithm

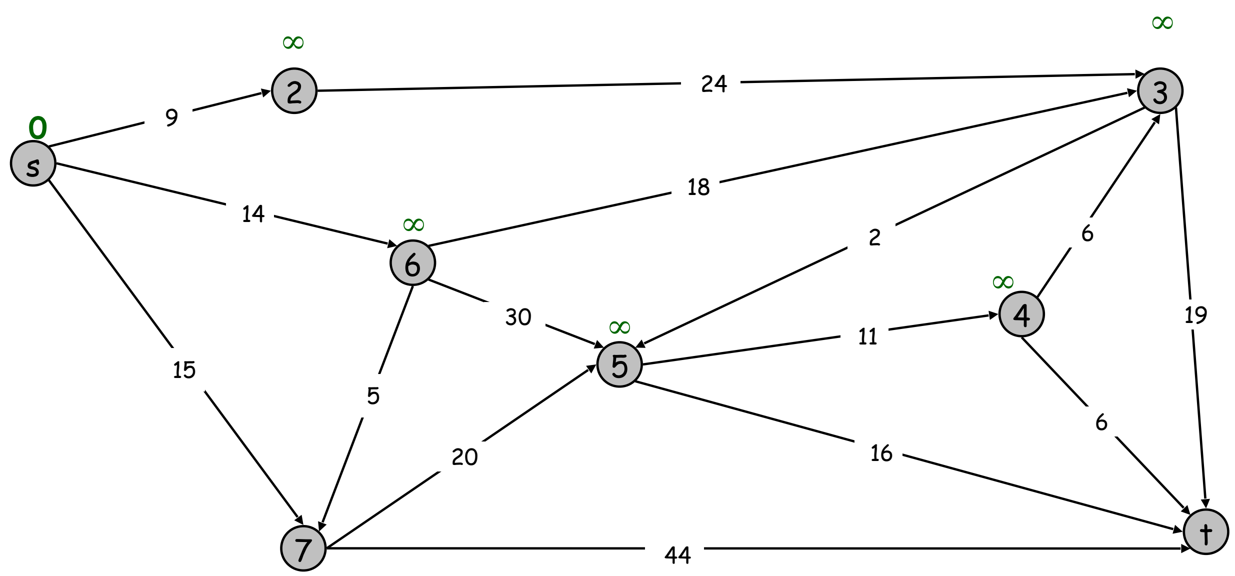
Find shortest path tree from s.



Dijkstra's Shortest Path Algorithm

S = { }

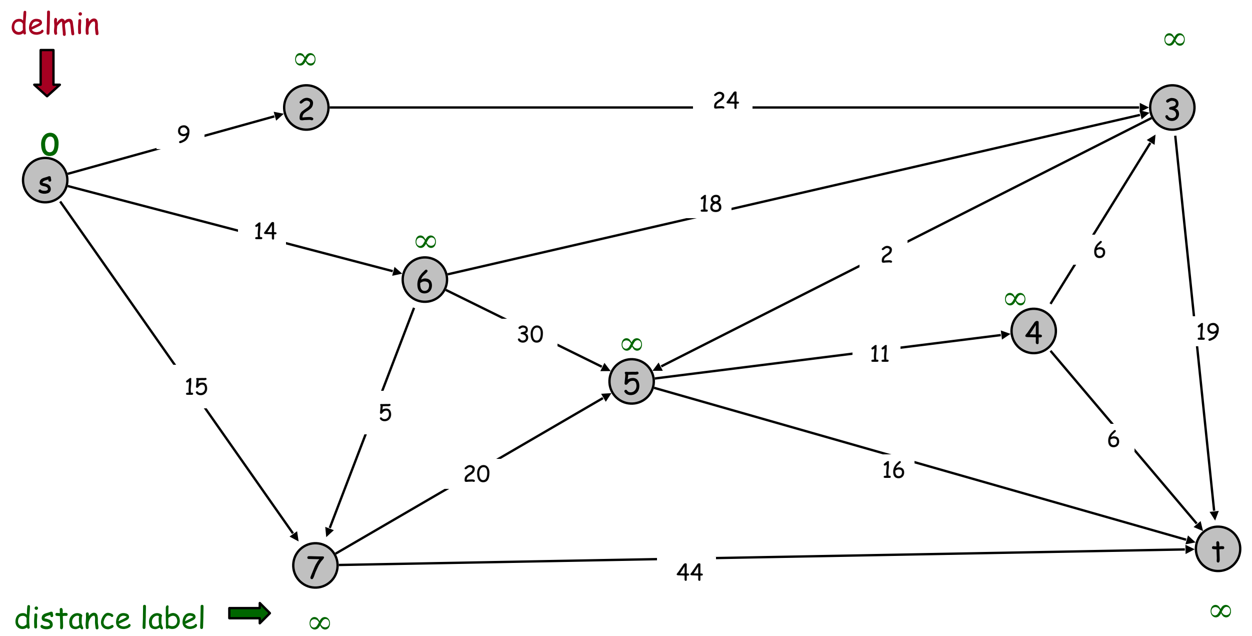
PQ = { s, 2, 3, 4, 5, 6, 7, t }



Dijkstra's Shortest Path Algorithm

S = { }

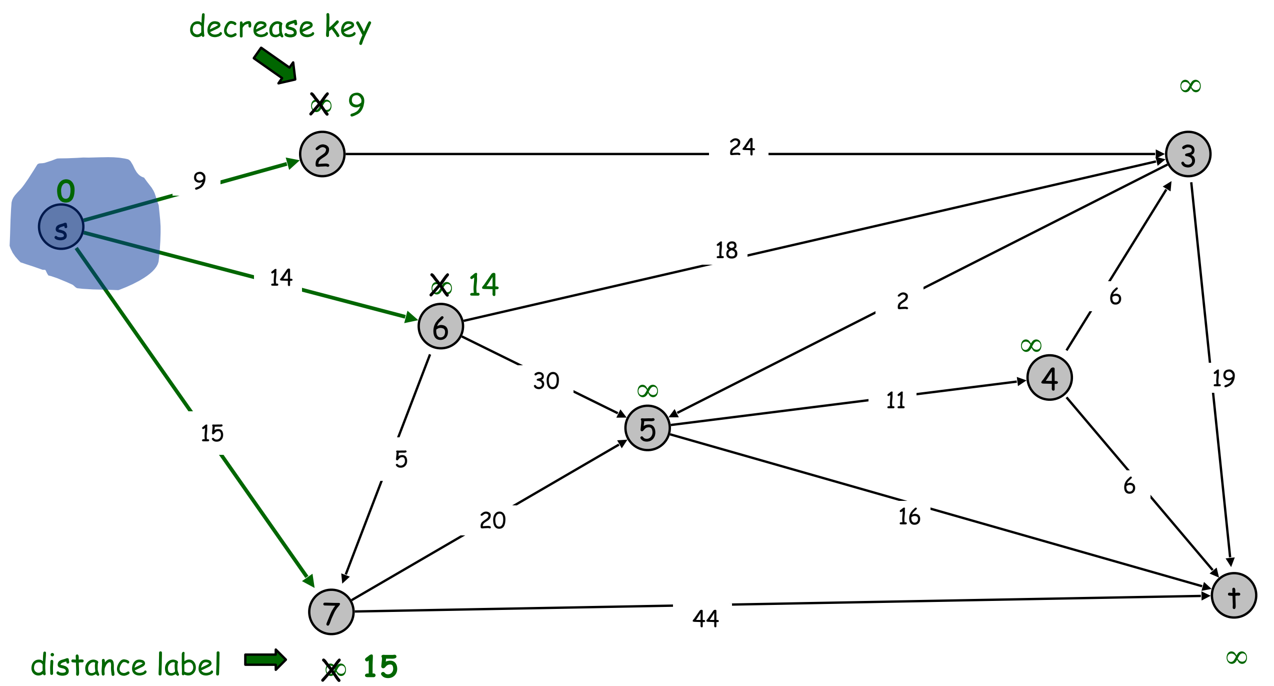
PQ = { s, 2, 3, 4, 5, 6, 7, t }



Dijkstra's Shortest Path Algorithm

S = { s }

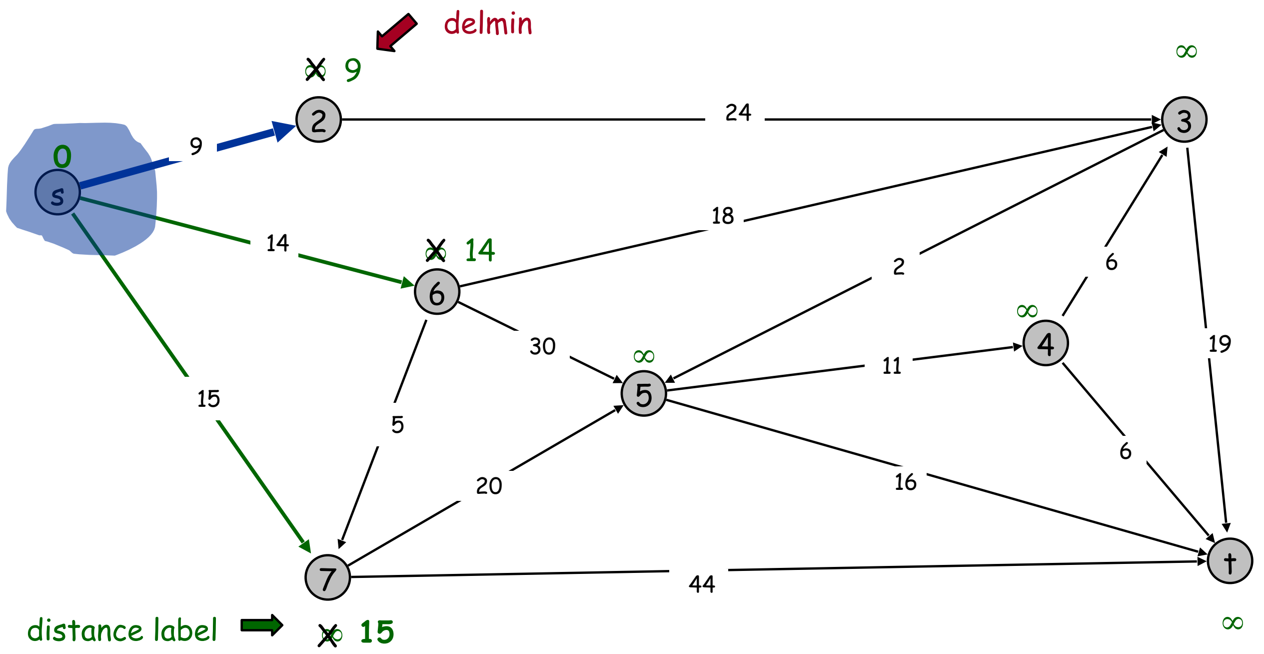
PQ = { 2, 3, 4, 5, 6, 7, t }



Dijkstra's Shortest Path Algorithm

S = { s }

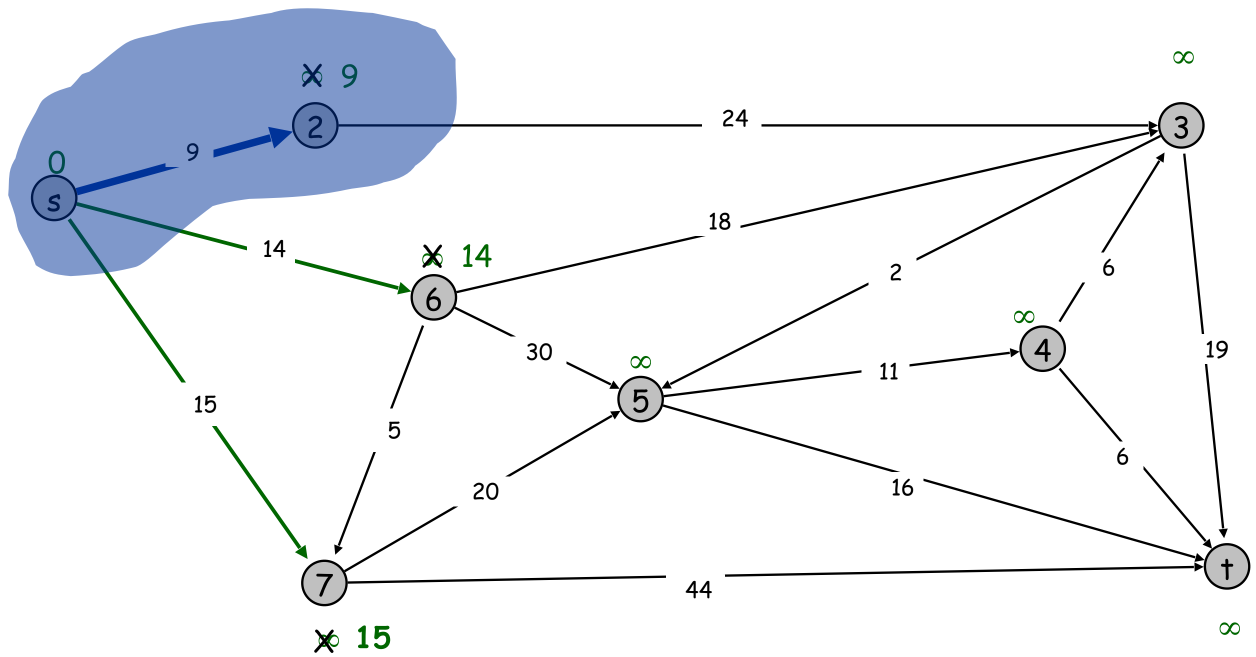
PQ = { 2, 3, 4, 5, 6, 7, t }



Dijkstra's Shortest Path Algorithm

S = { s, 2 }

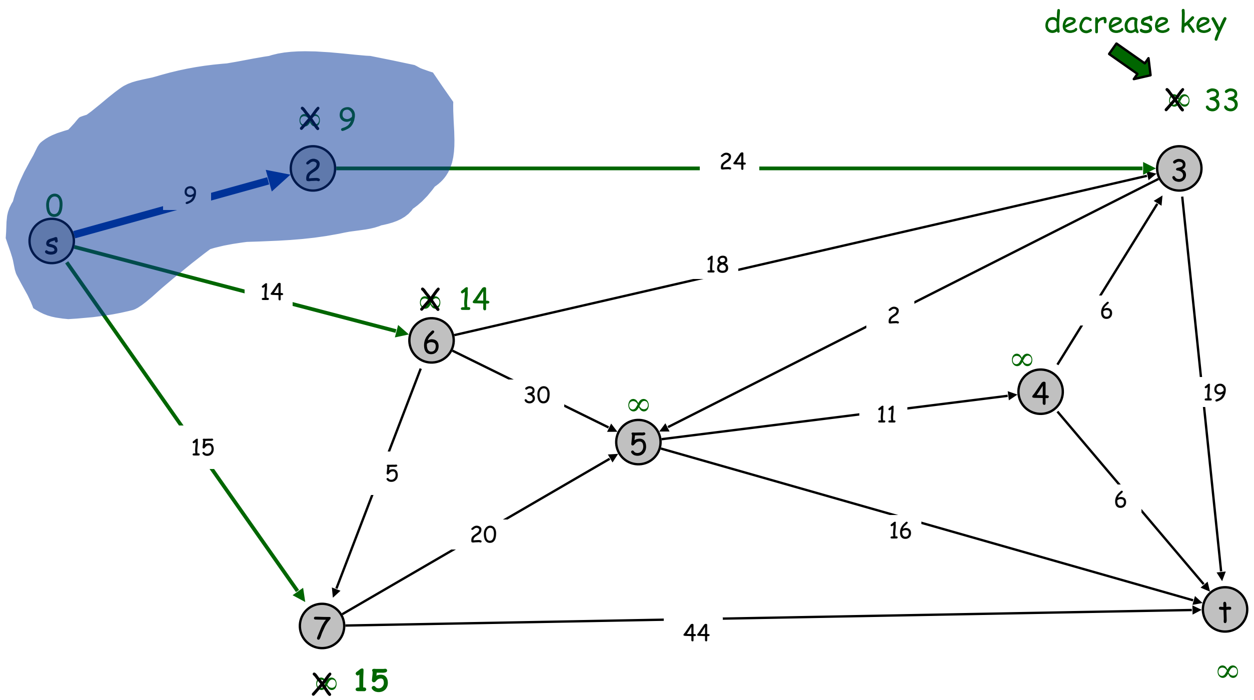
PQ = { 3, 4, 5, 6, 7, t }



Dijkstra's Shortest Path Algorithm

S = { s, 2 }

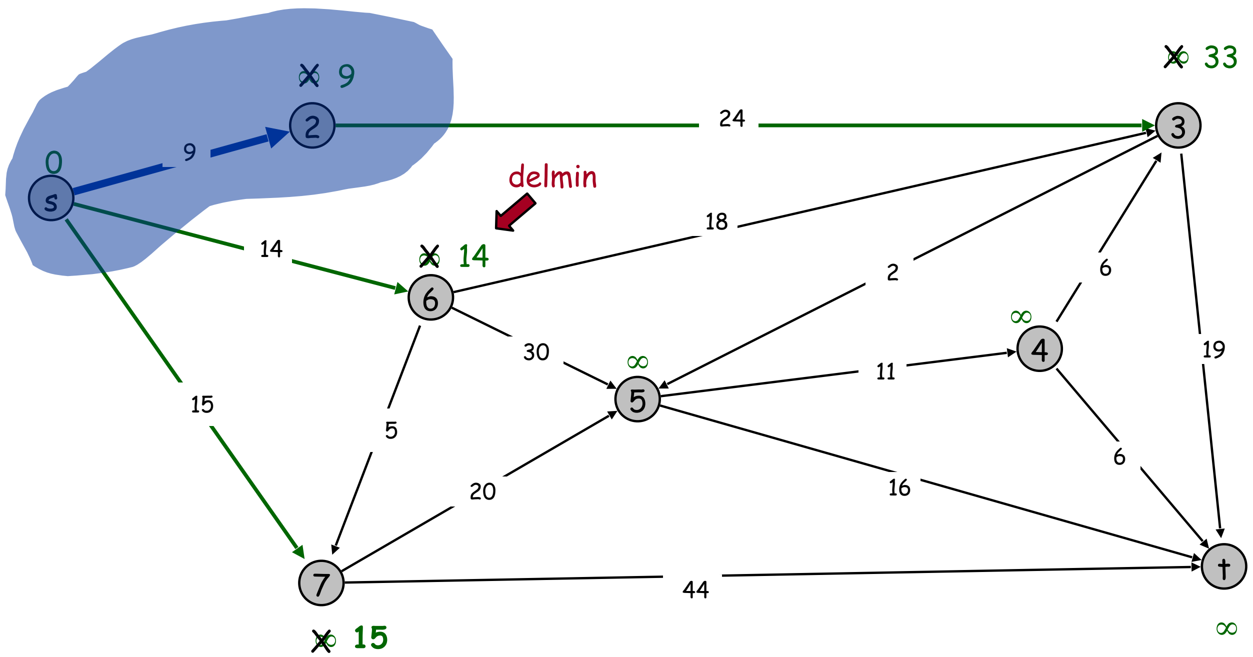
PQ = { 3, 4, 5, 6, 7, t }



Dijkstra's Shortest Path Algorithm

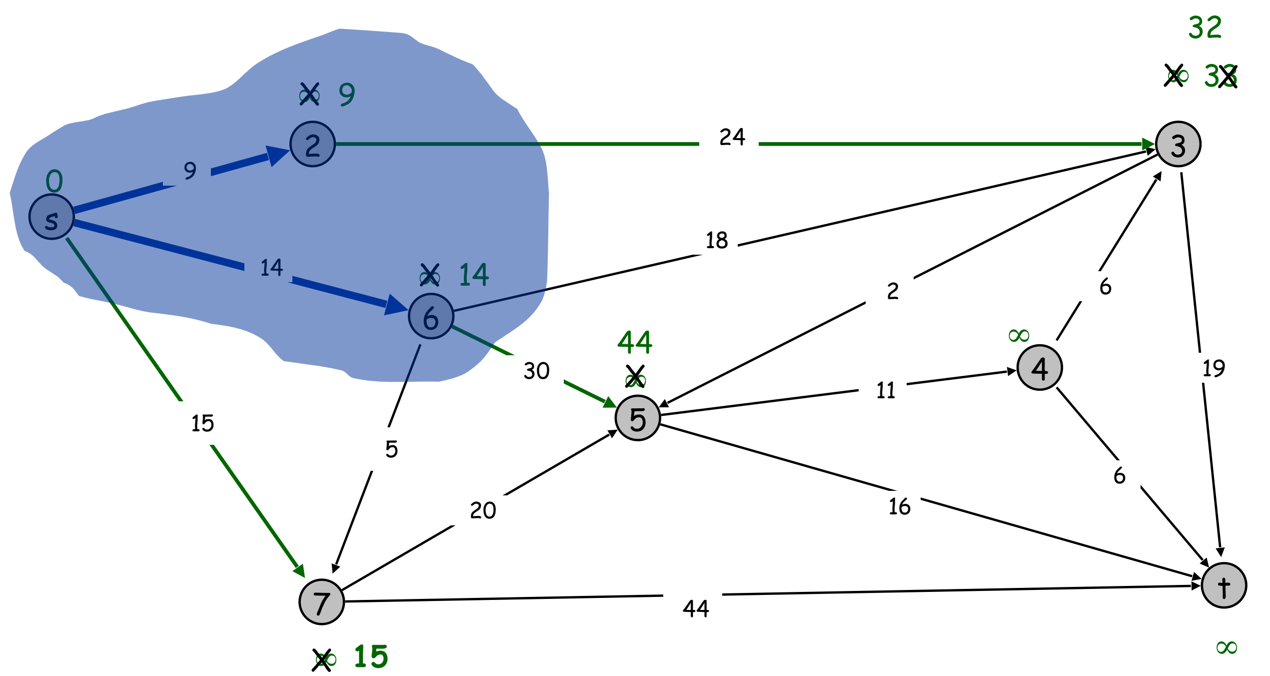
S = { s, 2 }

PQ = { 3, 4, 5, 6, 7, t }



Dijkstra's Shortest Path Algorithm

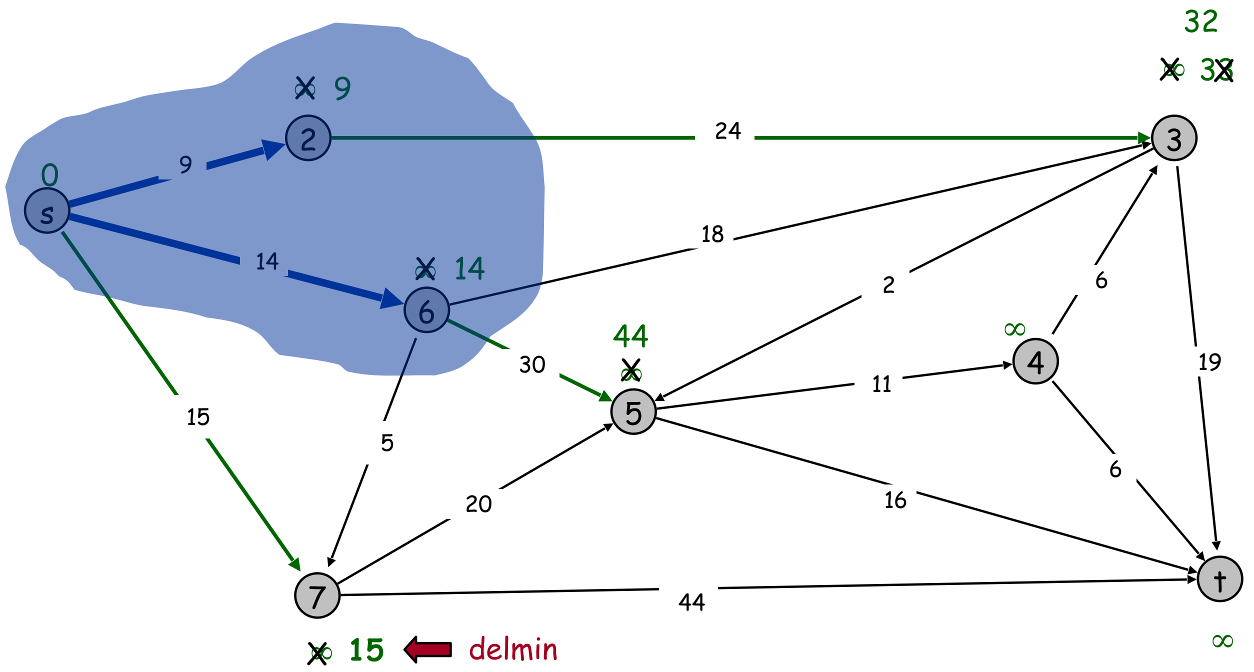
S = { s, 2, 6 }
PQ = { 3, 4, 5, 7, t }



Dijkstra's Shortest Path Algorithm

S = { s, 2, 6 }

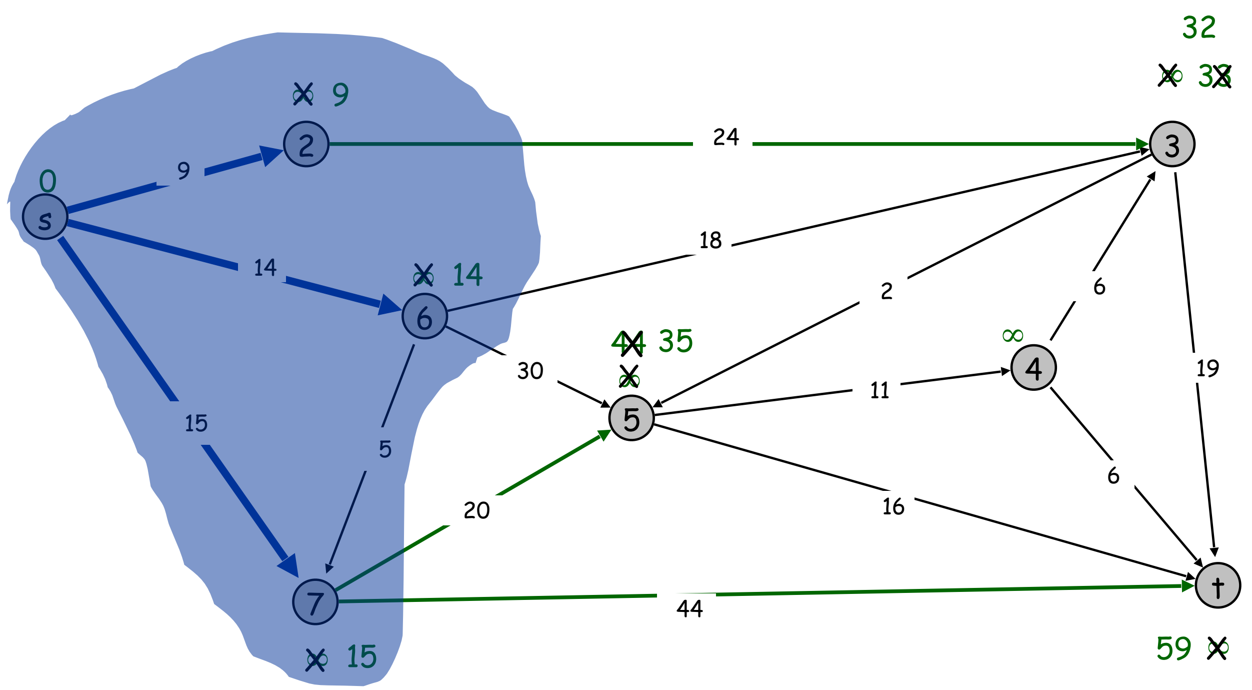
PQ = { 3, 4, 5, 7, t }



Dijkstra's Shortest Path Algorithm

S = { s, 2, 6, 7 }

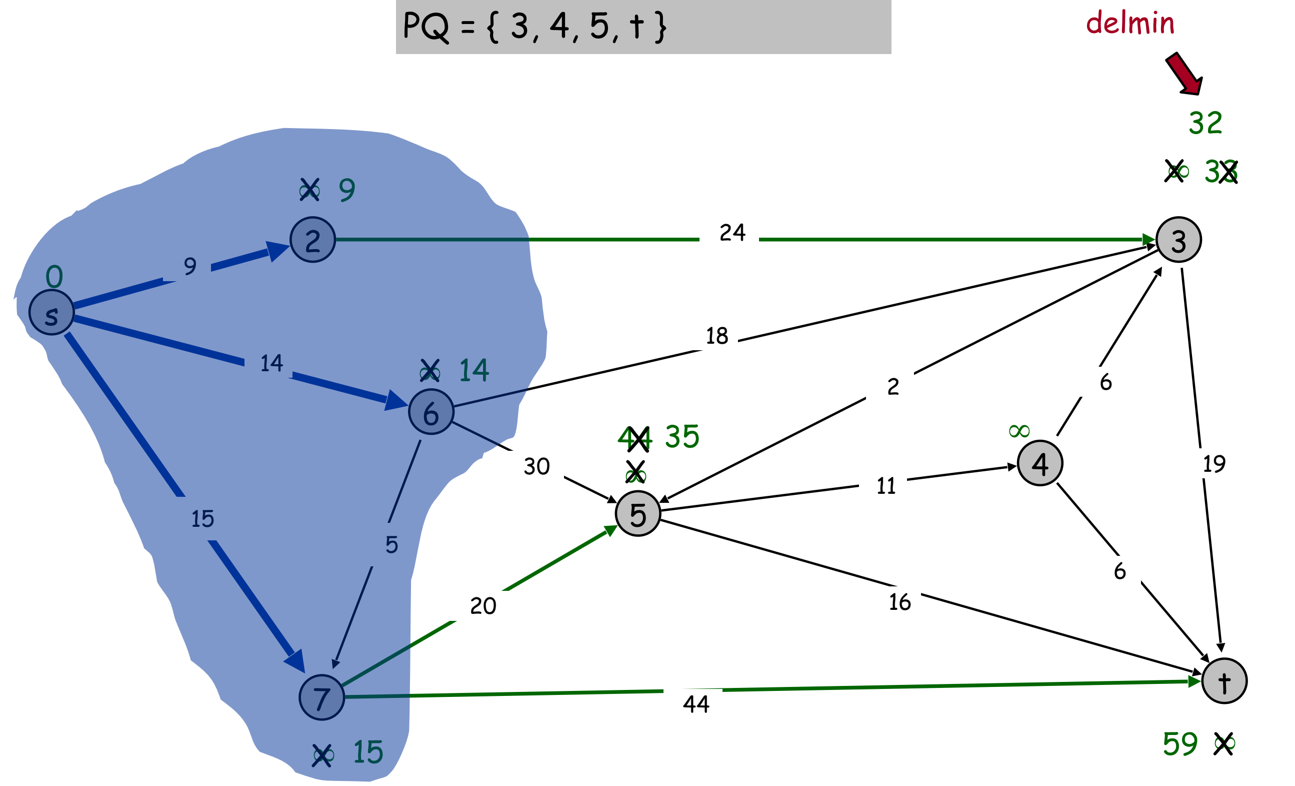
PQ = { 3, 4, 5, t }



Dijkstra's Shortest Path Algorithm

S = { s, 2, 6, 7 }

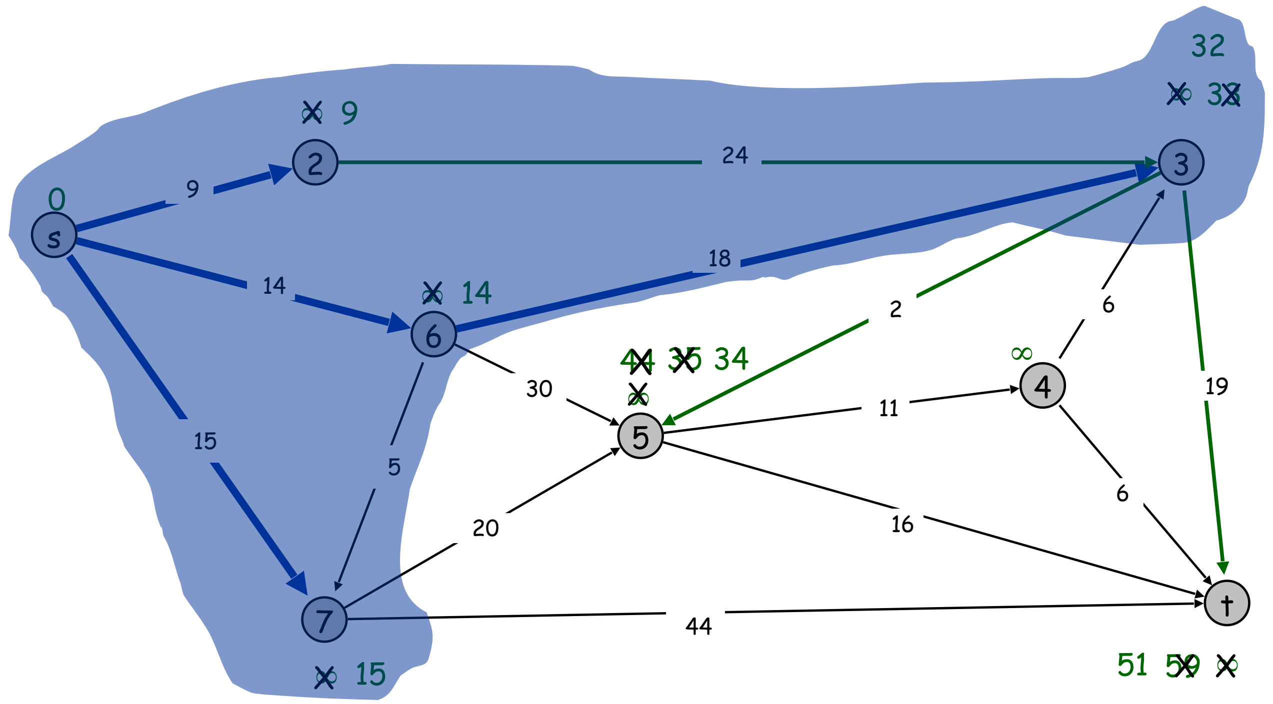
PQ = { 3, 4, 5, t }



Dijkstra's Shortest Path Algorithm

S = { s, 2, 3, 6, 7 }

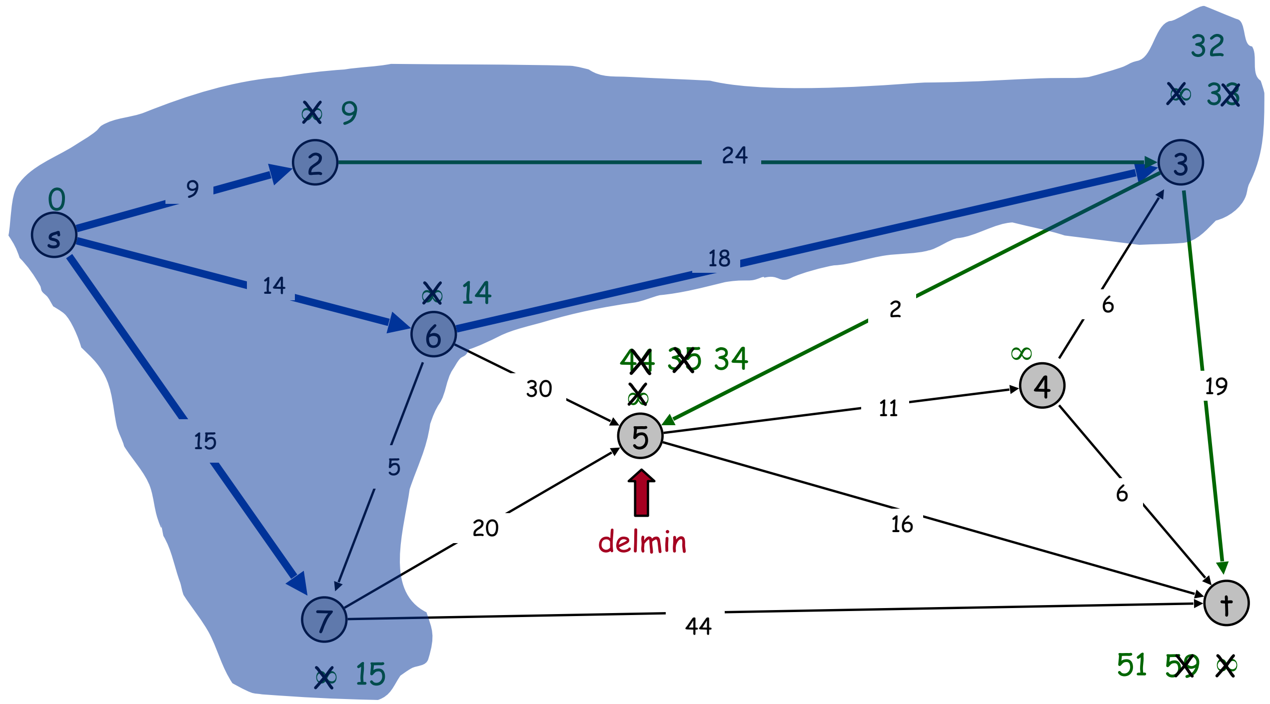
PQ = { 4, 5, t }



Dijkstra's Shortest Path Algorithm

S = { s, 2, 3, 6, 7 }

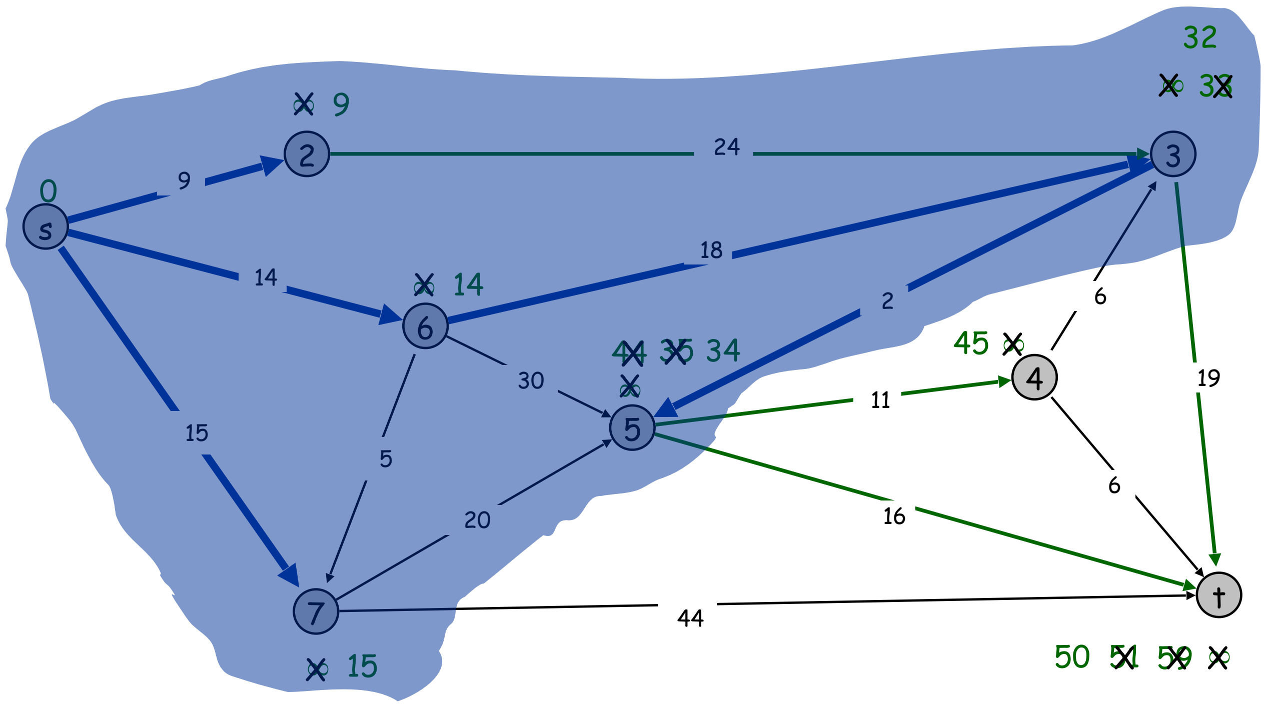
PQ = { 4, 5, t }



Dijkstra's Shortest Path Algorithm

S = { s, 2, 3, 5, 6, 7 }

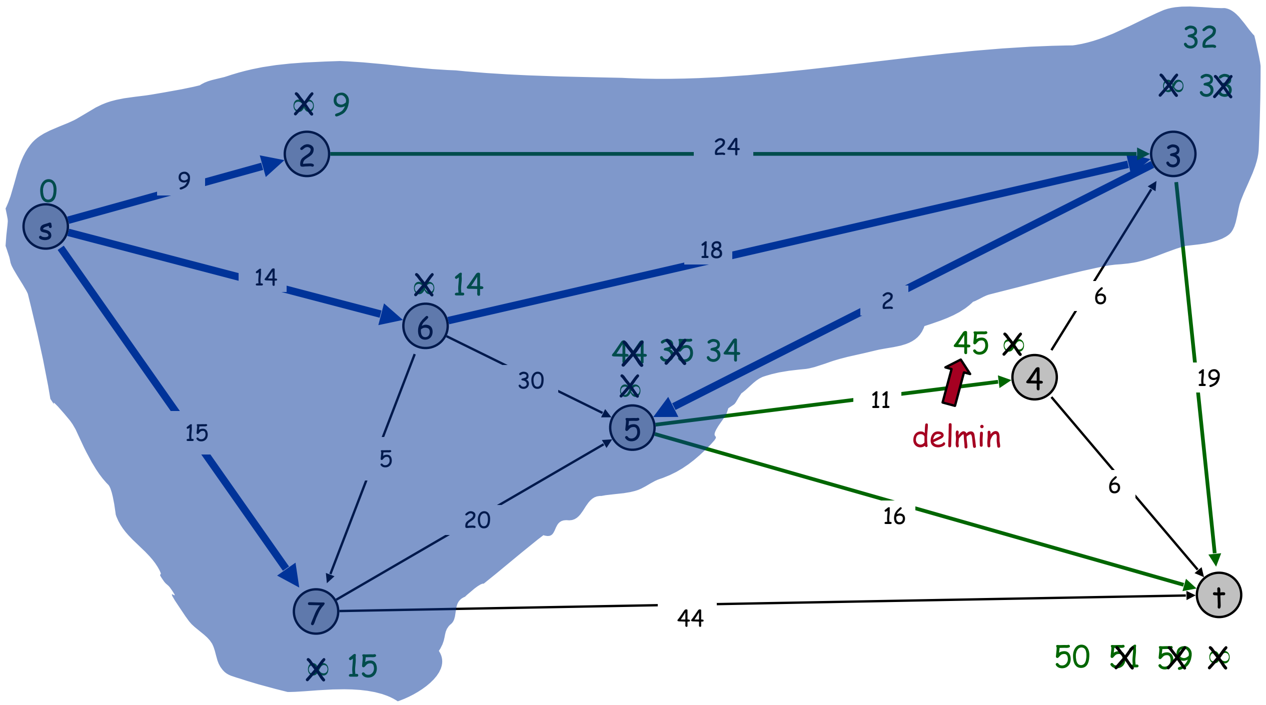
PQ = { 4, t }



Dijkstra's Shortest Path Algorithm

S = { s, 2, 3, 5, 6, 7 }

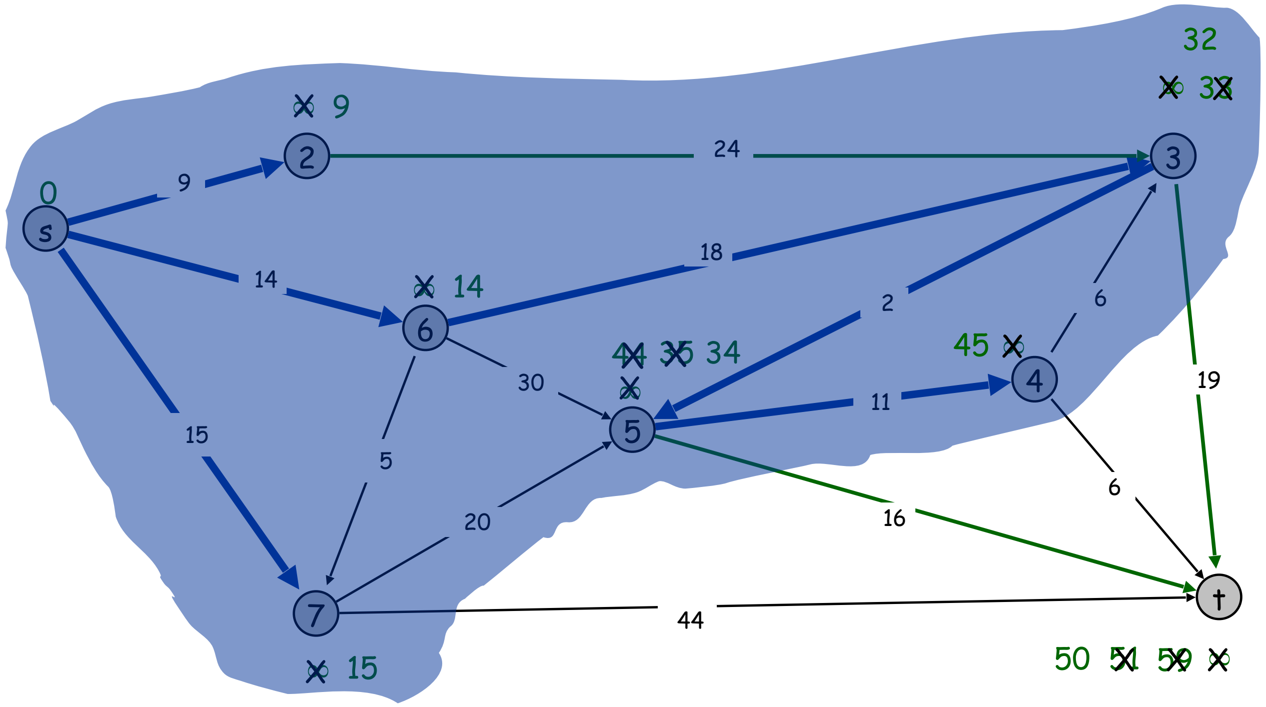
PQ = { 4, t }



Dijkstra's Shortest Path Algorithm

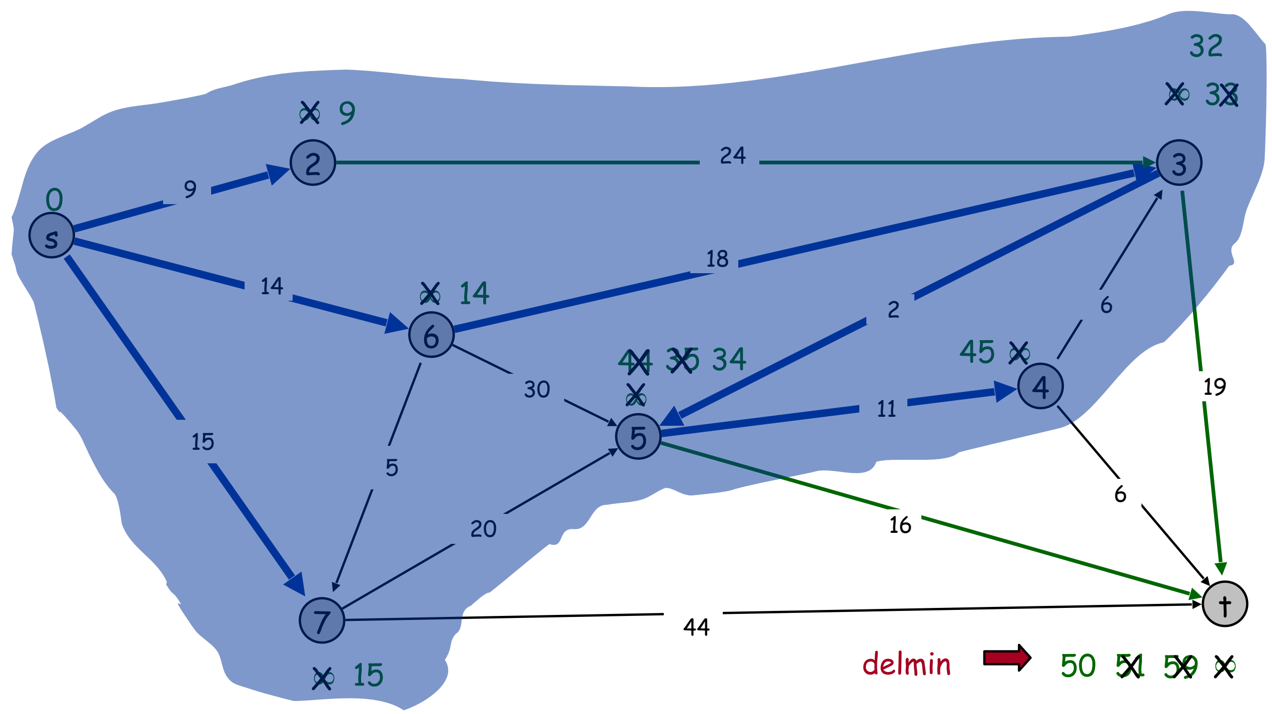
S = { s, 2, 3, 4, 5, 6, 7 }

PQ = { t }



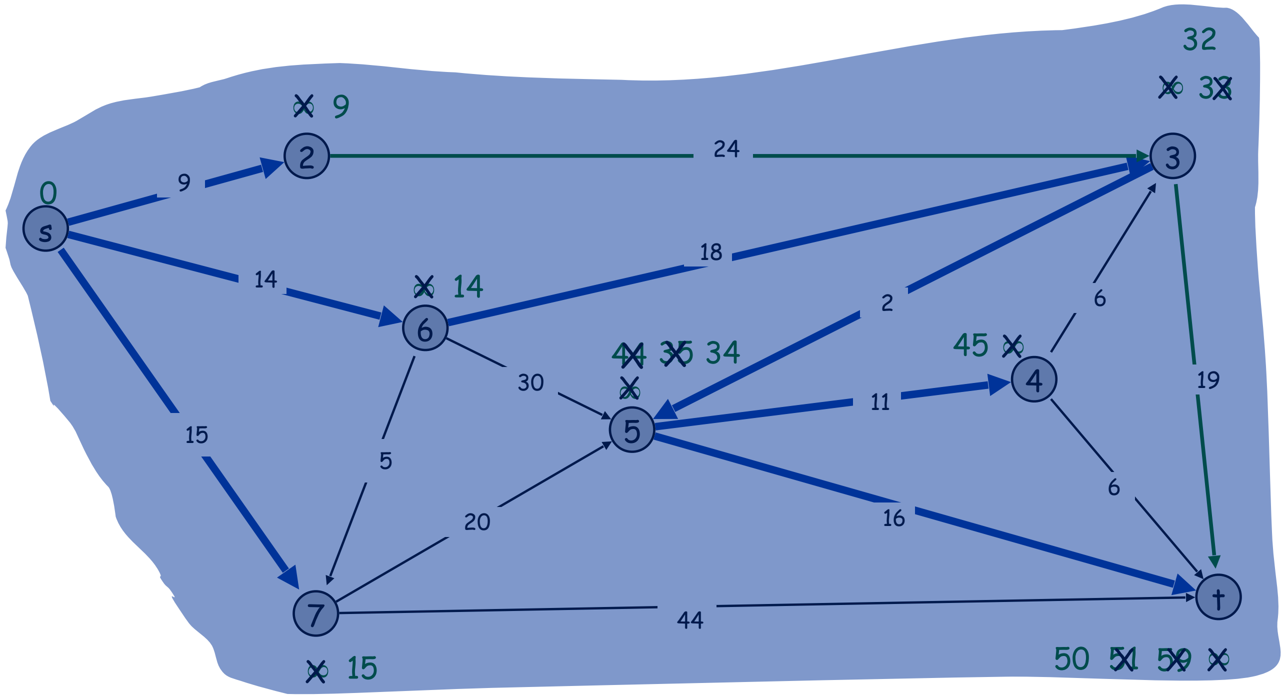
Dijkstra's Shortest Path Algorithm

S = { s, 2, 3, 4, 5, 6, 7 }
PQ = { t }



Dijkstra's Shortest Path Algorithm

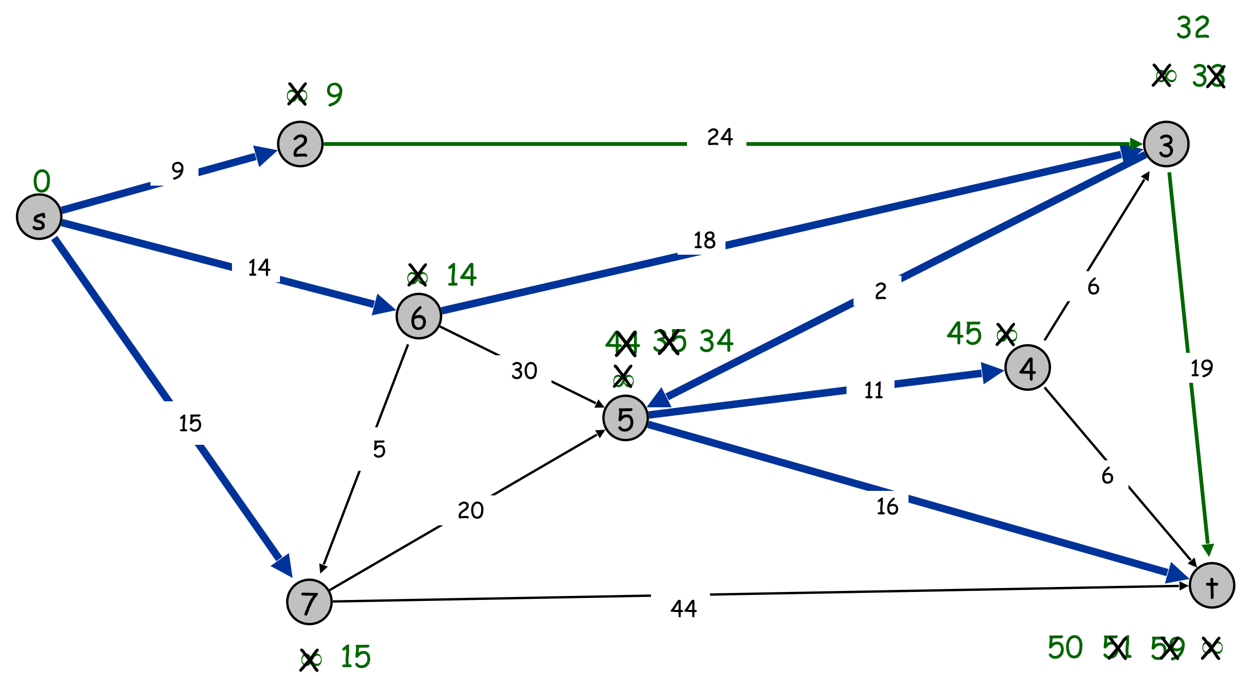
S = { s, 2, 3, 4, 5, 6, 7, t }
PQ = { }



Dijkstra's Shortest Path Algorithm

S = { s, 2, 3, 4, 5, 6, 7, t }

PQ = { }



Dijkstra's Algorithm

- A heap-based adaptable priority queue with location-aware entries stores the vertices outside the cloud
  - Key: distance
  - Value: vertex
  - Recall that method ***replaceKey***(***l***,***k***) changes the key of entry ***l***
- We store two labels with each vertex:
  - Distance
  - Entry in priority queue

Algorithm ***DijkstraDistances***(***G***, ***s***)
  Q ← new heap-based priority queue
  for all *v* ∈ ***G.vertices***()
    if *v* = *s*
      v.setDistance(0)
    else
      v.setDistance(∞)
    l ← ***Q.insert***(***v.getDistance***(), ***v***)
    v.setEntry(***l***)
  while ¬***Q.empty***()
    l ← ***Q.removeMin***()
    u ← ***l.getValue***()
    for all *e* ∈ ***u.incidentEdges***() { relax ***e*** }
      z ← ***e.opposite***(***u***)
      r ← ***u.getDistance***() + ***e.weight***()
      if *r* < ***z.getDistance***()
        z.setDistance(***r***)
        Q.replaceKey(***z.getEntry***(), ***r***)

Analysis of Dijkstra's Algorithm

- Graph operations
  - Method incidentEdges is called once for each vertex
- Label operations
  - We set/get the distance and locator labels of vertex z $O(\deg(z))$ times
  - Setting/getting a label takes $O(1)$ time
- Priority queue operations
  - Each vertex is inserted once into and removed once from the priority queue, where each insertion or removal takes $O(\log n)$ time
  - The key of a vertex in the priority queue is modified at most $\deg(w)$ times, where each key change takes $O(\log n)$ time
- Dijkstra's algorithm runs in $O((n + m) \log n)$ time provided the graph is represented by the adjacency list structure
  - Recall that $\sum_v \deg(v) = 2m$
- The running time can also be expressed as $O(m \log n)$ since the graph is connected

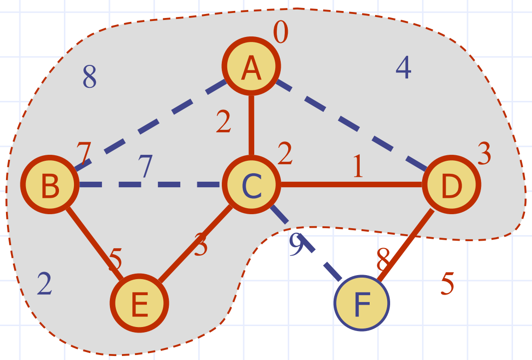
Shortest Paths Tree

- Using the template method pattern, we can extend Dijkstra's algorithm to return a tree of shortest paths from the start vertex to all other vertices
- We store with each vertex a third label:
  - parent edge in the shortest path tree
- In the edge relaxation step, we update the parent label

```
Algorithm DijkstraShortestPathsTree(G, s)

    …

  for all  v ∈ G.vertices()
    …
    v.setParent(∅)
    …

    for all  e ∈ u.incidentEdges()
      { relax edge e }
      z ← e.opposite(u)
      r ← u.getDistance() + e.weight()
      if  r < z.getDistance()
        z.setDistance(r)
        z.setParent(e)
        Q.replaceKey(z.getEntry(),r)
```

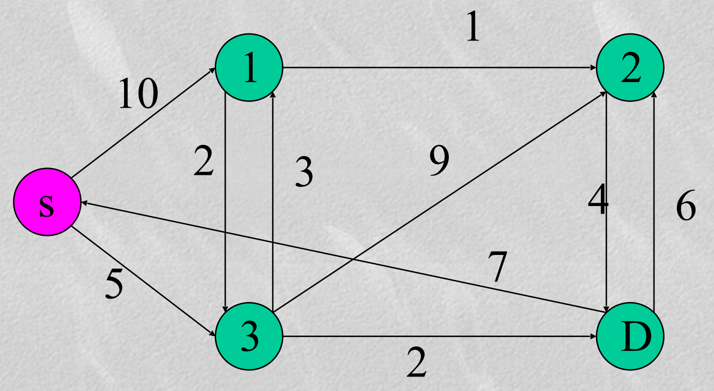
Why Dijkstra's Algorithm Works

- Dijkstra's algorithm is based on the greedy method. It adds vertices by increasing distance.
  - Suppose it didn't find all shortest distances. Let F be the first wrong vertex the algorithm processed.
  - When the previous node, D, on the true shortest path was considered, its distance was correct
  - But the edge (D,F) was relaxed at that time!
  - Thus, so long as d(F)≥d(D), F's distance cannot be wrong. That is, there is no wrong vertex



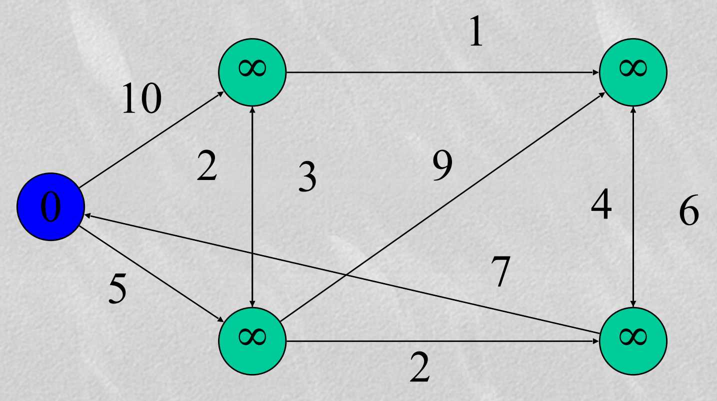
Graph Algorithms

Dijkstra's Algorithm - Example



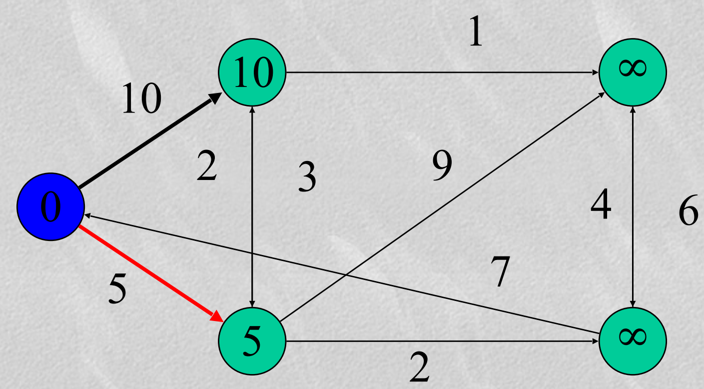
Graph Algorithms

Dijkstra's Algorithm - Example



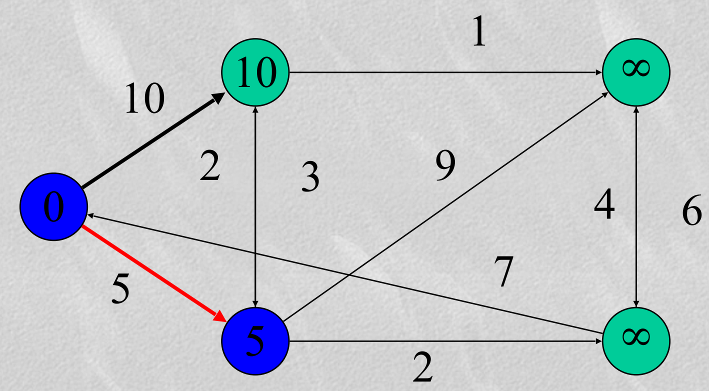
Graph Algorithms

Dijkstra's Algorithm - Example



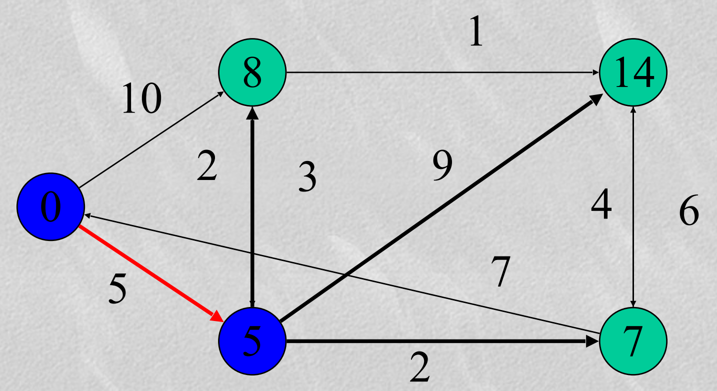
Graph Algorithms

Dijkstra's Algorithm - Example



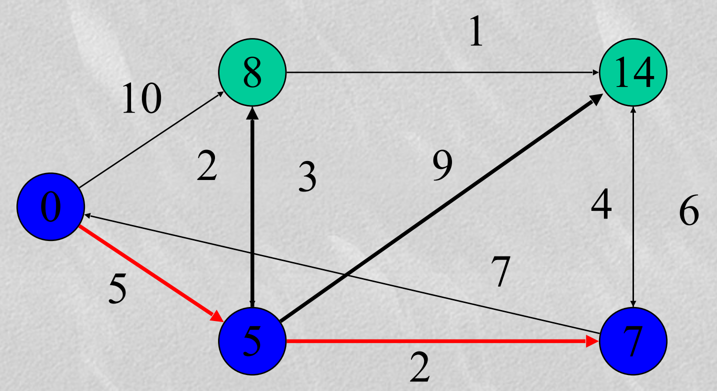
Graph Algorithms

Dijkstra's Algorithm - Example



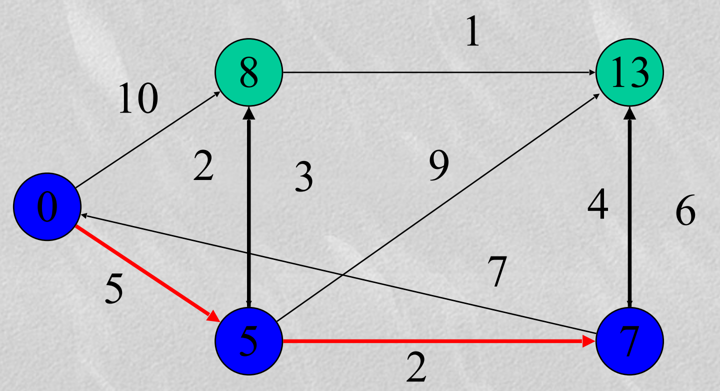
Graph Algorithms

Dijkstra's Algorithm - Example



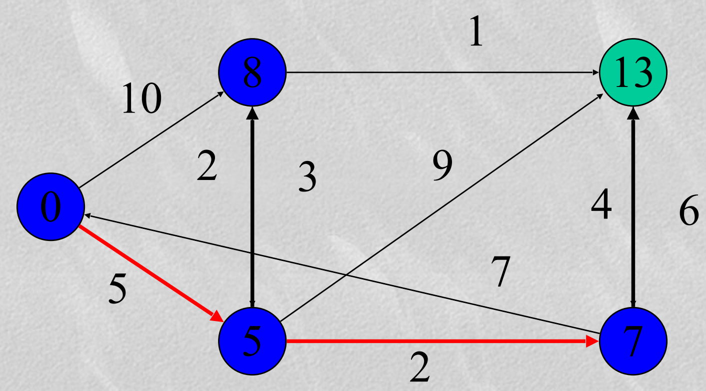
Graph Algorithms

Dijkstra's Algorithm - Example



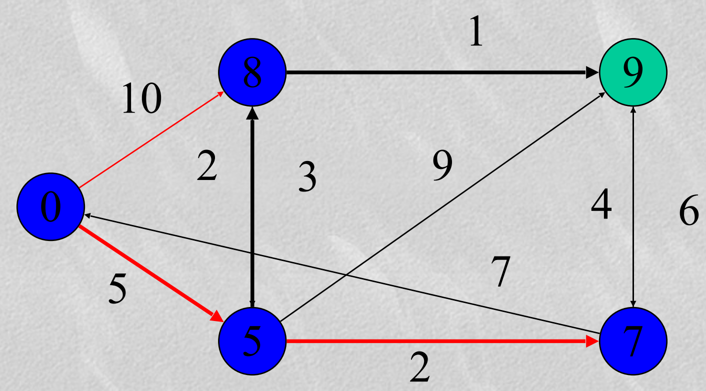
Graph Algorithms

Dijkstra's Algorithm - Example



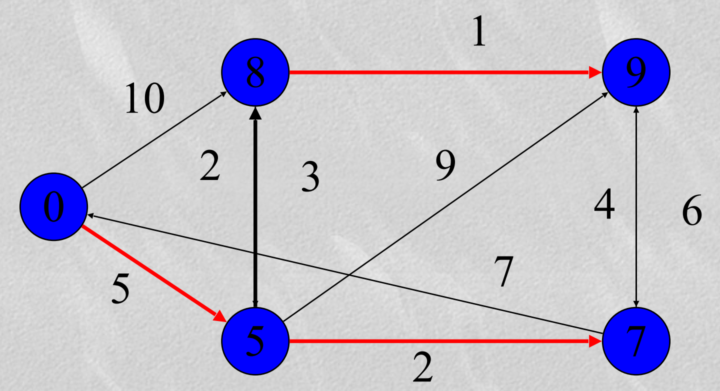
Graph Algorithms

Dijkstra's Algorithm - Example

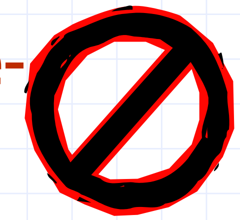


Graph Algorithms

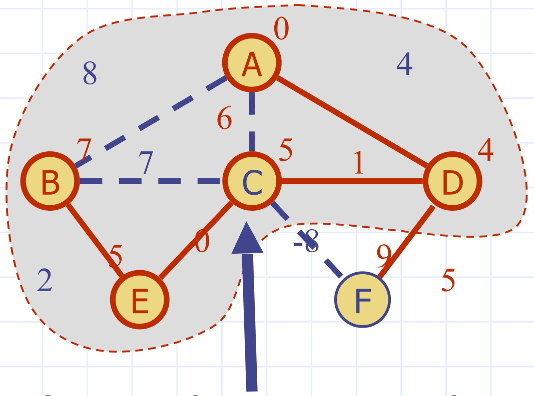
Dijkstra's Algorithm - Example



Why It Doesn't Work for Negative-Weight Edges



- Dijkstra's algorithm is based on the greedy method. It adds vertices by increasing distance.
  - If a node with a negative incident edge were to be added late to the cloud, it could mess up distances for vertices already in the cloud.



C's true distance is 1, but it is already in the cloud with d(C)=5!

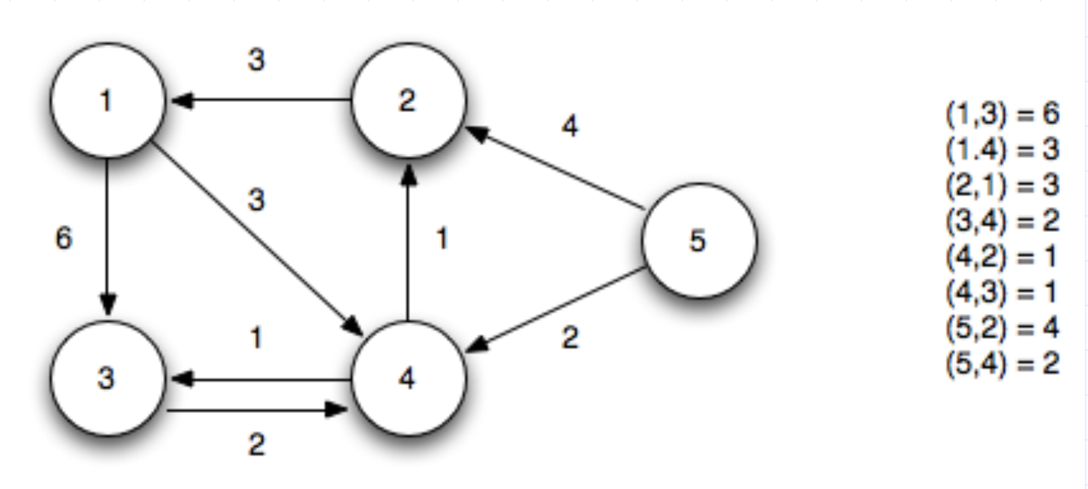
Bellman-Ford Algorithm

- Works even with negative-weight edges
- Must assume directed edges (for otherwise we would have negative-weight cycles)
- Iteration i finds all shortest paths that use i edges.
- Running time: O(nm).
- Can be extended to detect a negative-weight cycle if it exists
  - How?

Algorithm *BellmanFord*(G, s)
  for all $v \in G.vertices()$
    if $v = s$
      $v.setDistance(0)$
    else
      $v.setDistance(\infty)$
  for $i \leftarrow 1$ **to** $n - 1$ **do**
    for each $e \in G.edges()$
      { relax edge e }
      $u \leftarrow e.origin()$
      $z \leftarrow e.opposite(u)$
      $r \leftarrow u.getDistance() + e.weight()$
      if $r < z.getDistance()$
        $z.setDistance(r)$

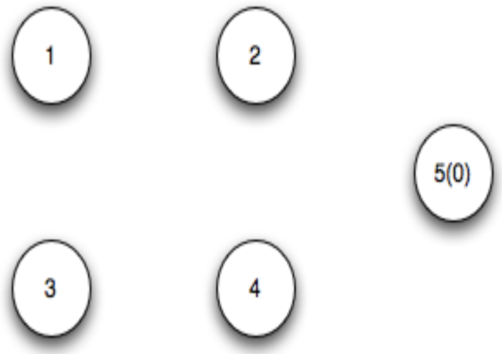
Bellman-Ford Algorithm

- start



Bellman-Ford Algorithm

- Using vertex 5 as the source (setting its distance to 0), we initialize all the other distances to ∞.

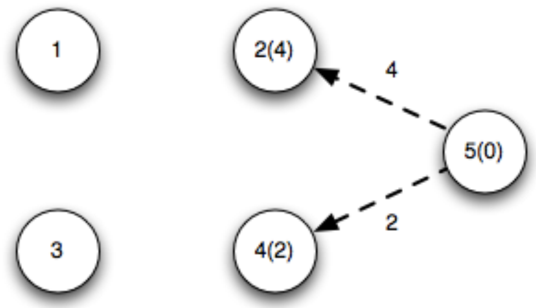


(1,3) = 6
(1,4) = 3
(2,1) = 3
(3,4) = 2
(4,2) = 1
(4,3) = 1
(5,2) = 4
(5,4) = 2

	1	2	3	4	5
d	∞	∞	∞	∞	**0**
π	/	/	/	/	/

Bellman-Ford Algorithm

- Iteration 1: Edges (u_5,u_2) and (u_5,u_4) relax updating the distances to 2 and 4



(1,3) = 6
(1,4) = 3
(2,1) = 3
(3,4) = 2
(4,2) = 1
(4,3) = 1
(5,2) = 4
(5,4) = 2

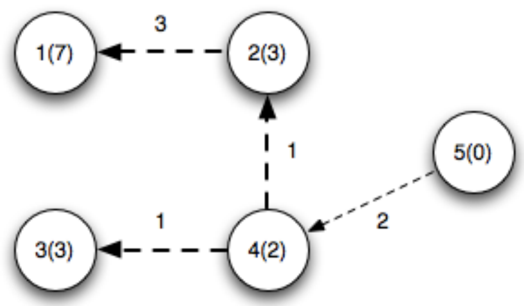
6
4 3 3

	1	2	3	4	5
d	~~∞~~	~~4~~	~~∞~~	**2**	0
π	~~/~~	~~5~~	~~/~~	**5**	/

2 4 4

Bellman-Ford Algorithm

- Iteration 2: Edges (u_2,u_1), (u_4,u_2) and (u_4,u_3) relax updating the distances to 1, 2, and 4 respectively. Note edge (u_4,u_2) finds a shorter path to vertex 2 by going through

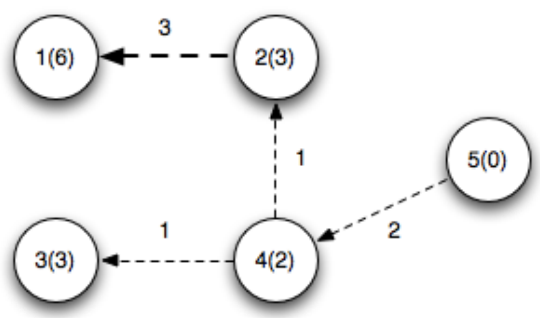


(1,3) = 6
(1.4) = 3
(2,1) = 3
(3,4) = 2
(4,2) = 1
(4,3) = 1
(5,2) = 4
(5,4) = 2

	1	2	3	4	5
d	**7**	**3**	**3**	2	0
π	**2**	**4**	**4**	5	/

Bellman-Ford Algorithm

- Iteration 3: Edge (u2,u1) relaxes (since a shorter path to vertex 2 was found in the previous iteration) updating the distance to 1

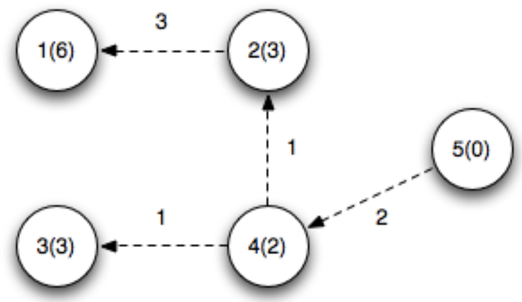


(1,3) = 6
(1,4) = 3
(2,1) = 3
(3,4) = 2
(4,2) = 1
(4,3) = 1
(5,2) = 4
(5,4) = 2

	1	2	3	4	5
d	**6**	3	3	2	0
π	**2**	4	4	5	/

Bellman-Ford Algorithm

- Iteration 4: No edges relax

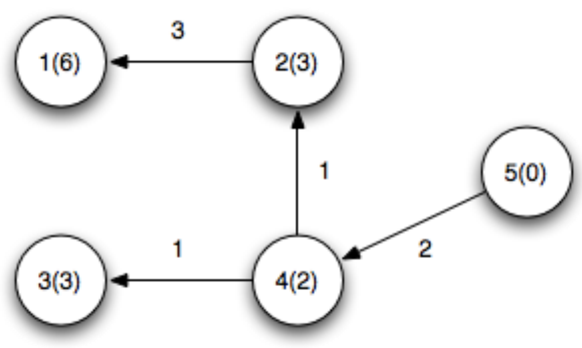


(1,3) = 6
(1,4) = 3
(2,1) = 3
(3,4) = 2
(4,2) = 1
(4,3) = 1
(5,2) = 4
(5,4) = 2

	1	2	3	4	5
d	6	3	3	2	0
π	2	4	4	5	/

Bellman-Ford Algorithm

- The final shortest paths from vertex 5 with corresponding distances is



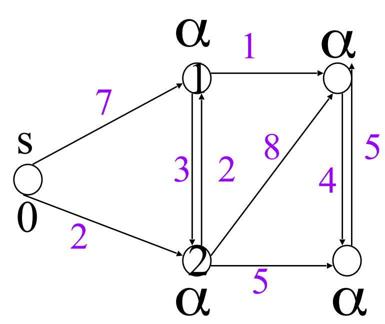
	1	2	3	4	5
d	6	3	3	2	0
π	2	4	4	5	/

Bellman-Ford Algorithm

- Negative cycle checks: We now check the relaxation condition one additional time for each edge. If any of the checks pass then there exists a negative weight cycle in the graph.
- v3.d > u1.d + w(1,3) $\Rightarrow$ 3 $\ngtr$ 6 + 6 = 12 ✓
- v4.d > u1.d + w(1,4) $\Rightarrow$ 2 $\ngtr$ 6 + 3 = 9 ✓
- v1.d > u2.d + w(2,1) $\Rightarrow$ 6 $\ngtr$ 3 + 3 = 6 ✓
- v4.d > u3.d + w(3,4) $\Rightarrow$ 2 $\ngtr$ 3 + 2 = 5 ✓
- v2.d > u4.d + w(4,2) $\Rightarrow$ 3 $\ngtr$ 2 + 1 = 3 ✓
- v3.d > u4.d + w(4,3) $\Rightarrow$ 3 $\ngtr$ 2 + 1 = 3 ✓

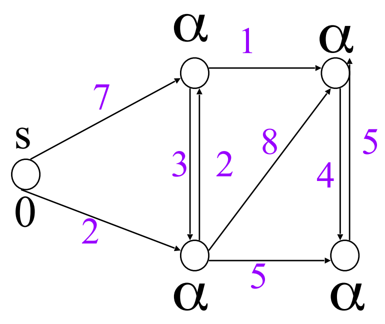
Bellman-Ford Algorithm Example

Bellman-Ford Algorithm Example

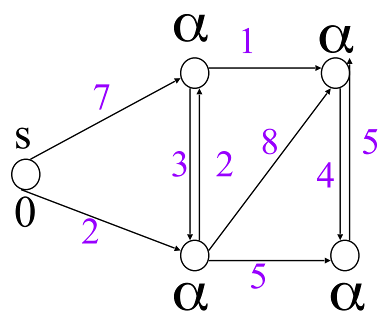


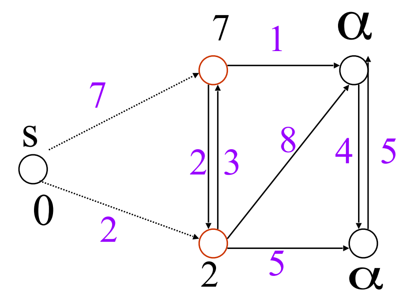
Bellman-Ford Algorithm Example

Bellman-Ford Algorithm Example



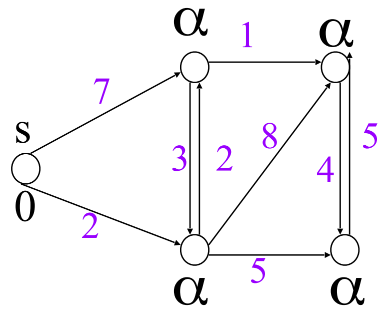
Bellman-Ford Algorithm Example



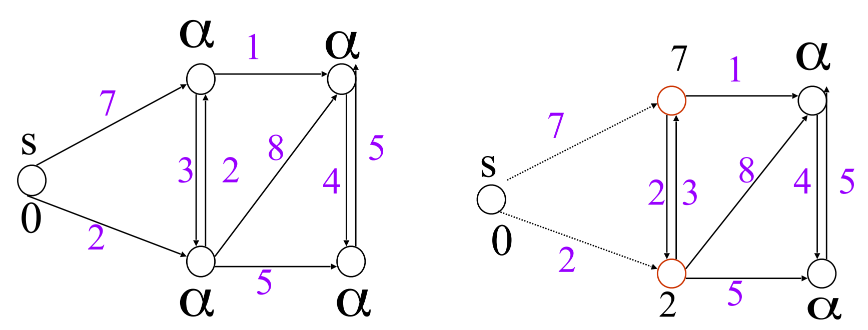


Bellman-Ford Algorithm Example

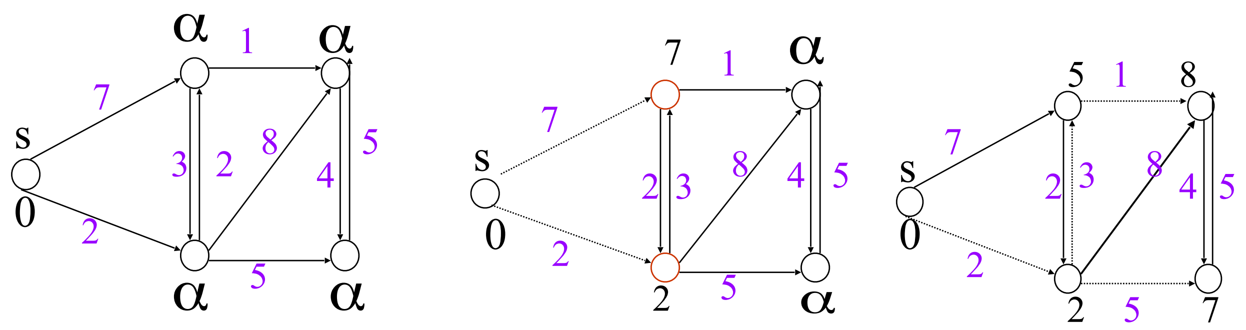
Bellman-Ford Algorithm Example



Bellman-Ford Algorithm Example

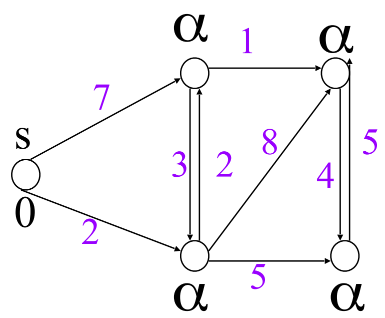


Bellman-Ford Algorithm Example

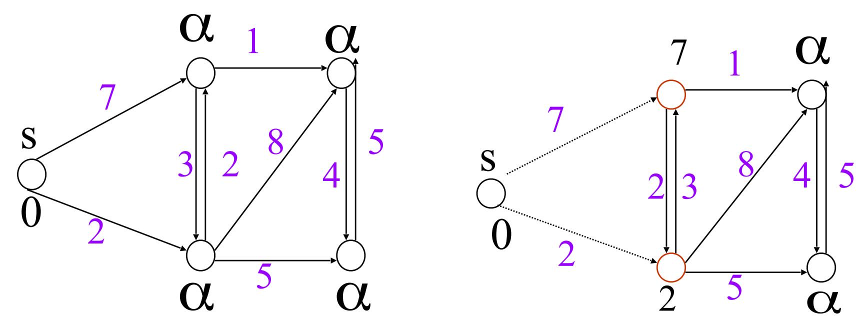


Bellman-Ford Algorithm Example

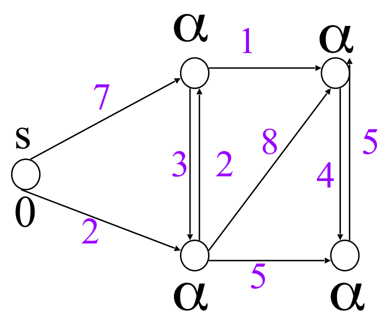
Bellman-Ford Algorithm Example

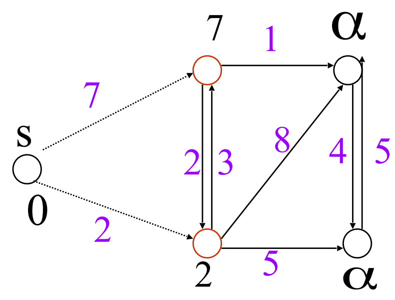


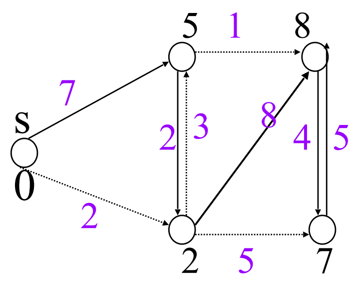
Bellman-Ford Algorithm Example



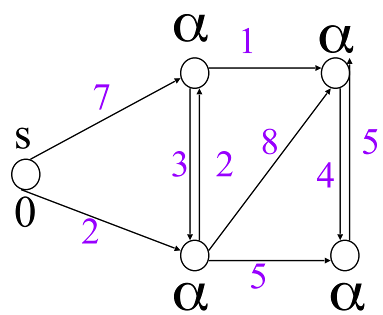
Bellman-Ford Algorithm Example

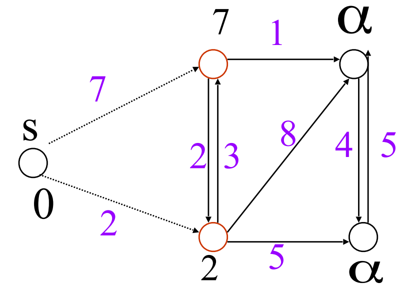


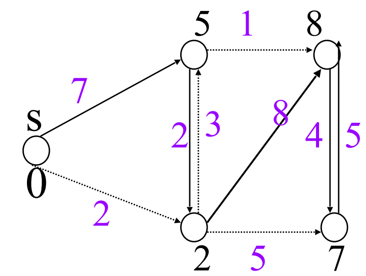


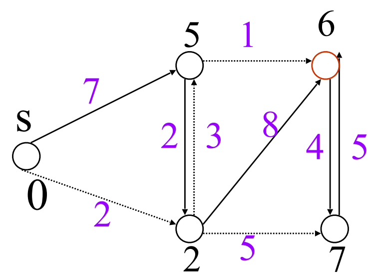


Bellman-Ford Algorithm Example



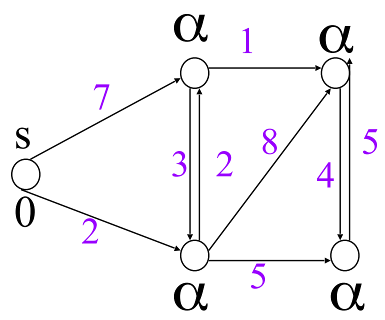




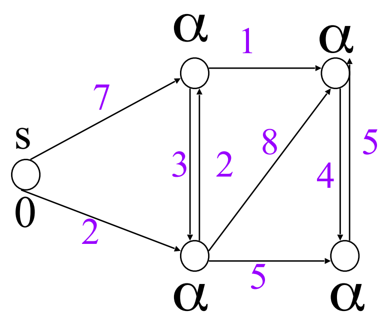


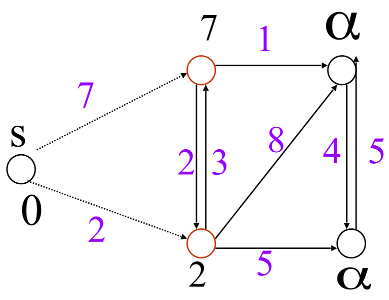
Bellman-Ford Algorithm Example

Bellman-Ford Algorithm Example

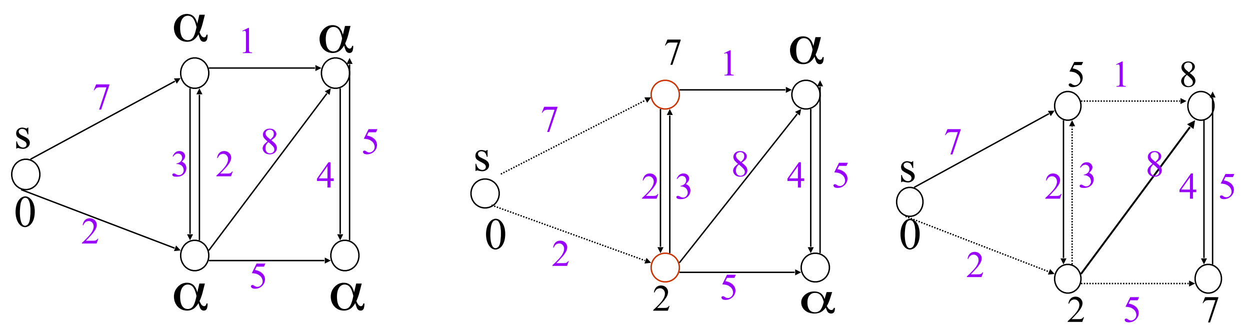


Bellman-Ford Algorithm Example

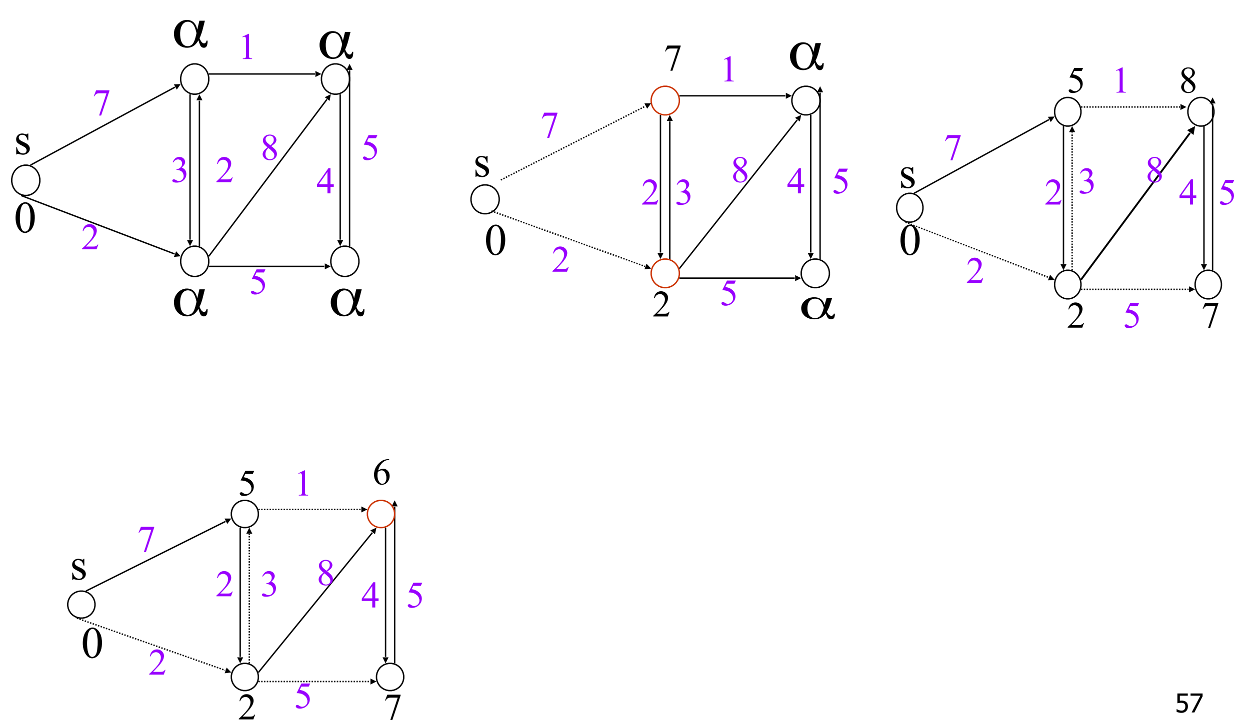




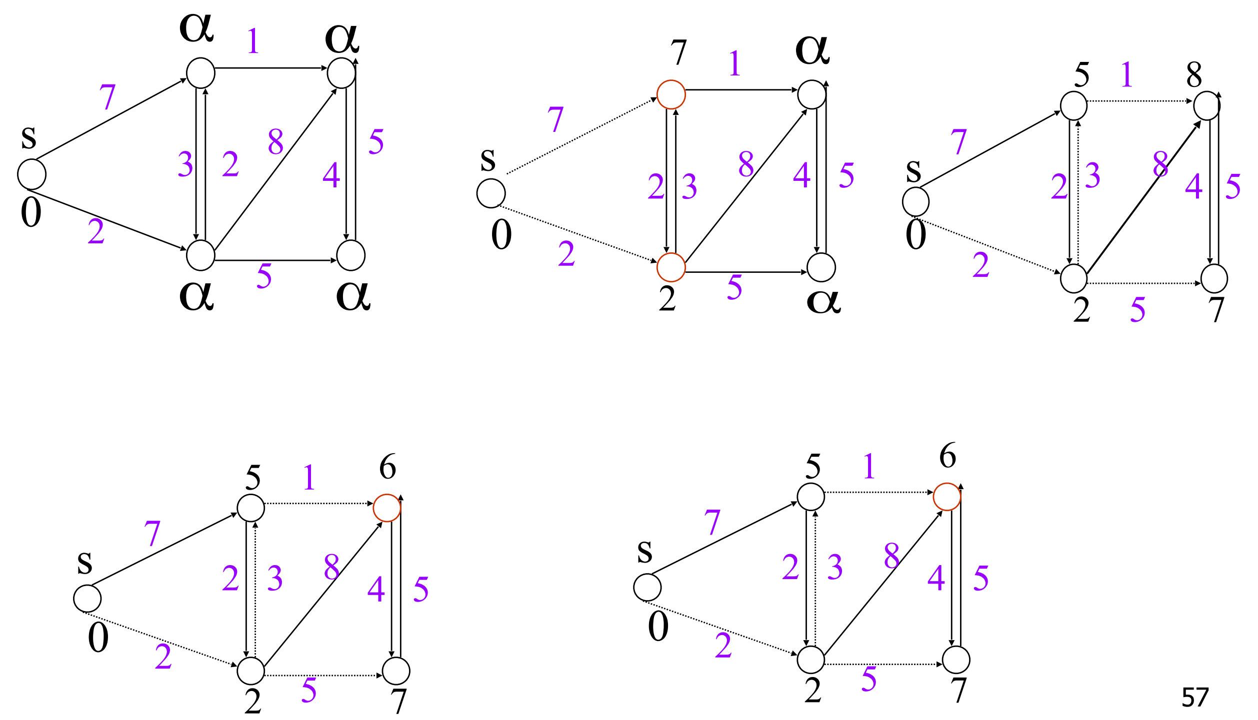
Bellman-Ford Algorithm Example



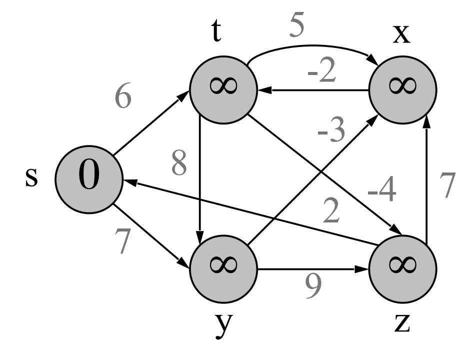
Bellman-Ford Algorithm Example



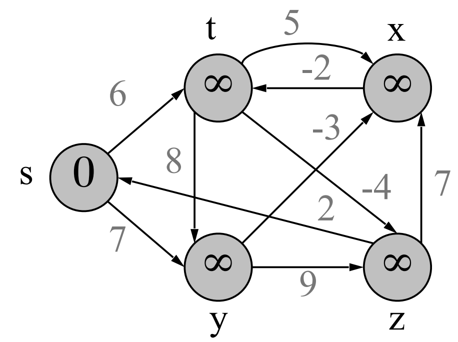
Bellman-Ford Algorithm Example



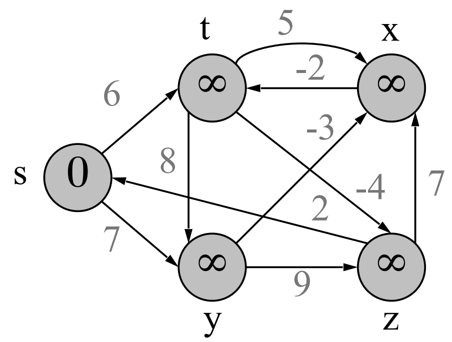
Bellman-Ford Example

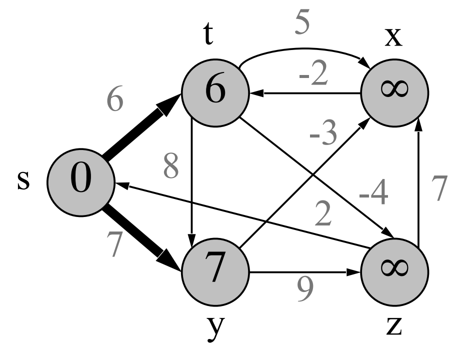


Bellman-Ford Example

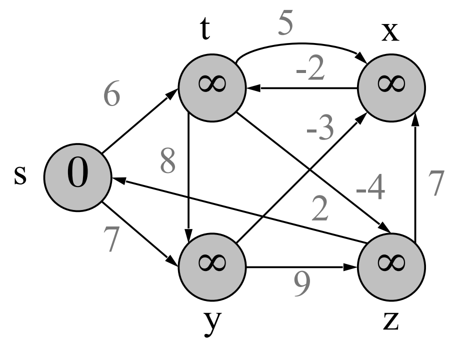


Bellman-Ford Example

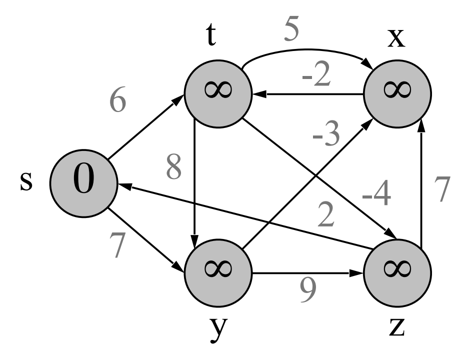


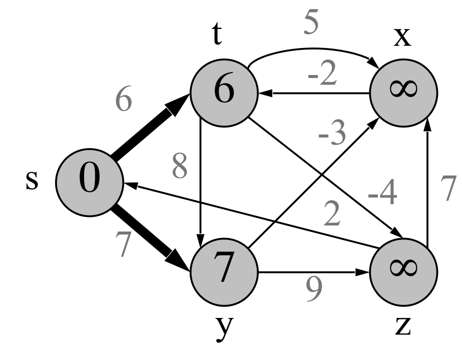


Bellman-Ford Example

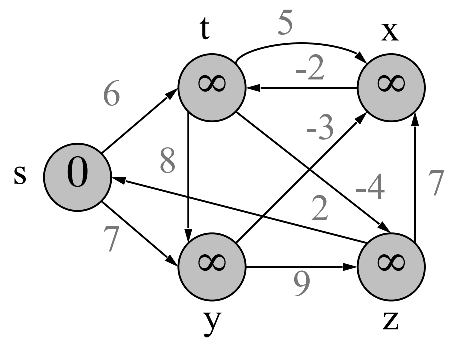


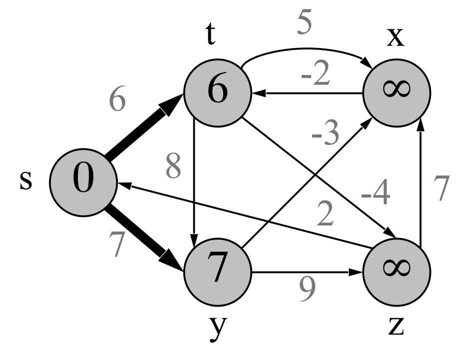
Bellman-Ford Example

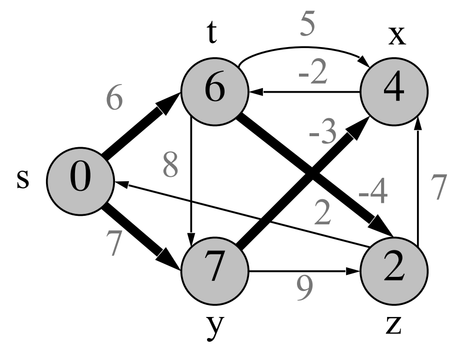




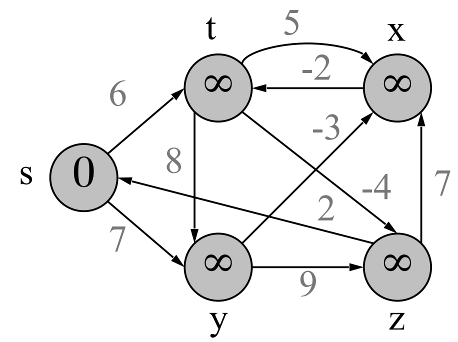
Bellman-Ford Example



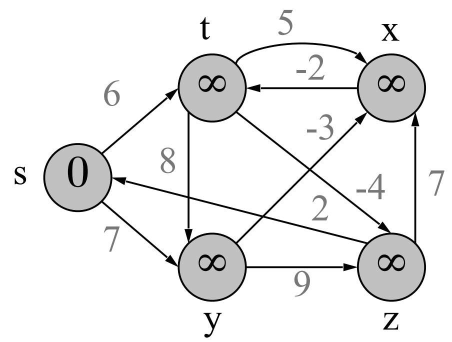


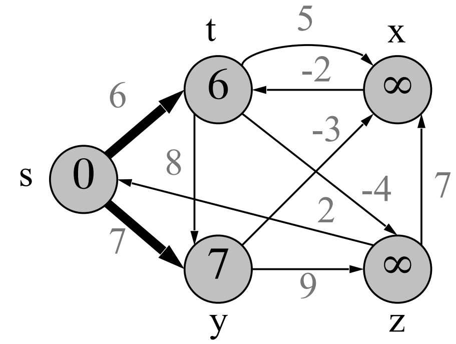


Bellman-Ford Example

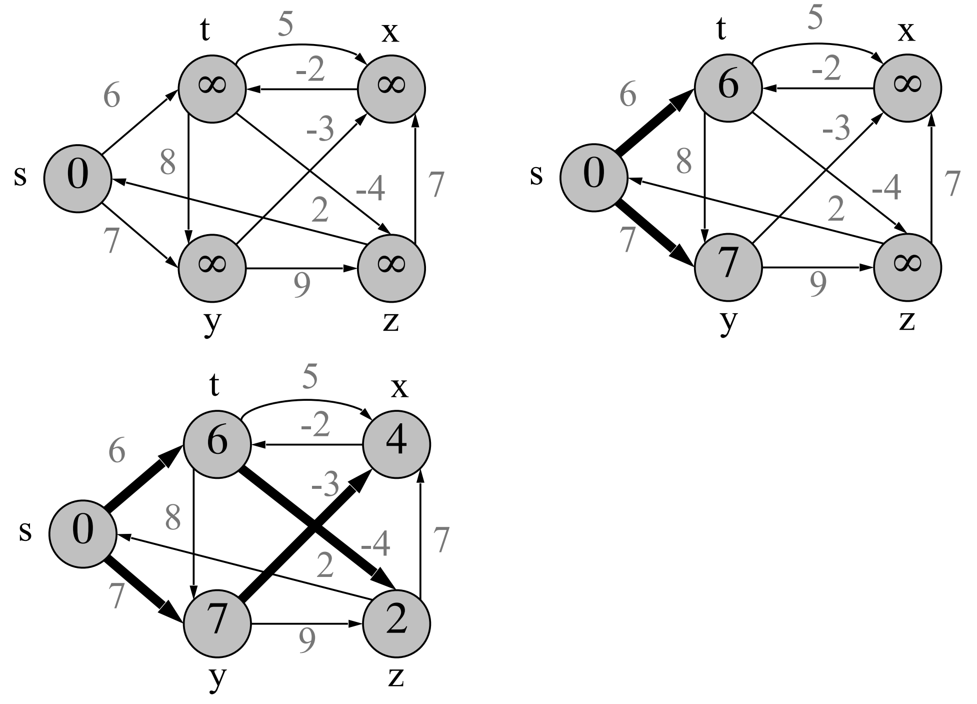


Bellman-Ford Example

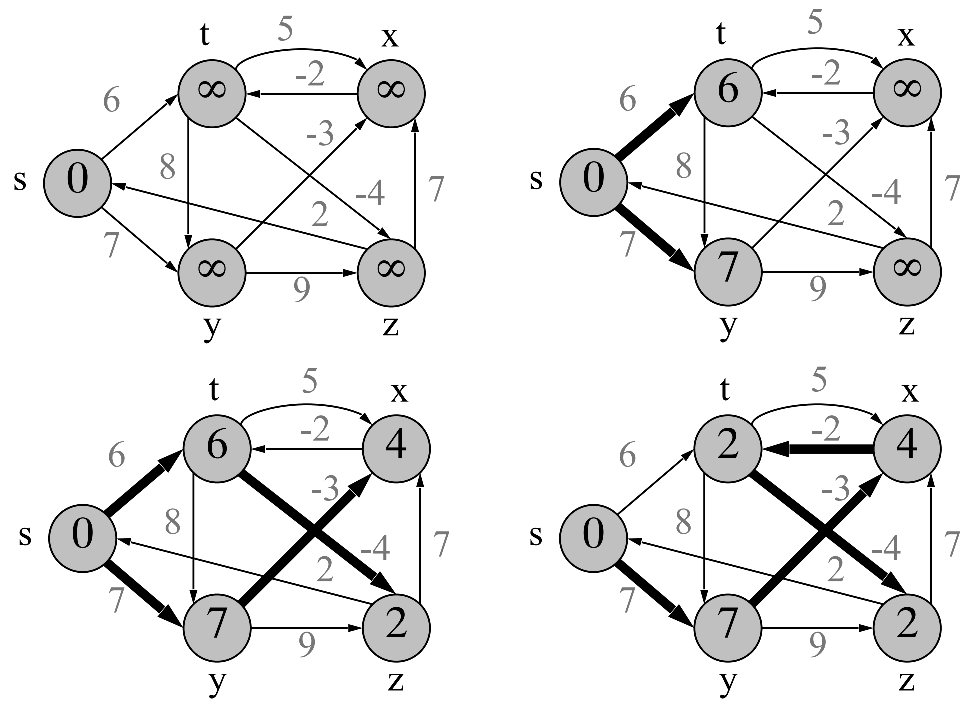




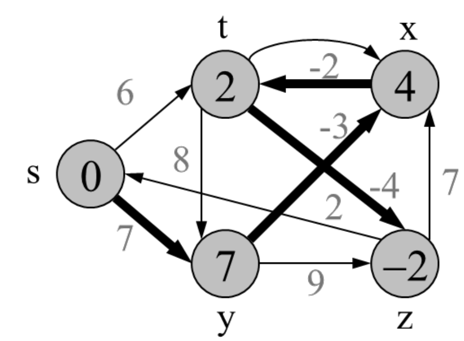
Bellman-Ford Example



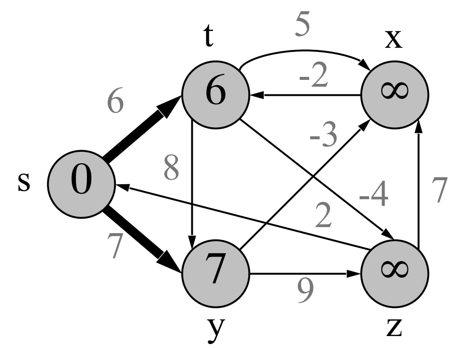
Bellman-Ford Example

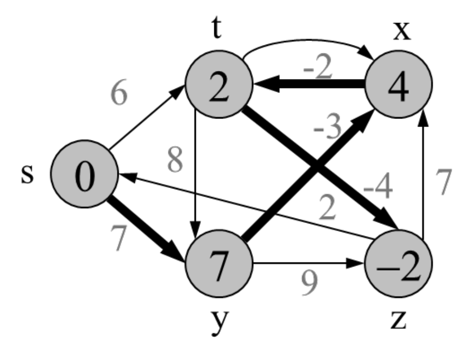


Bellman-Ford Example

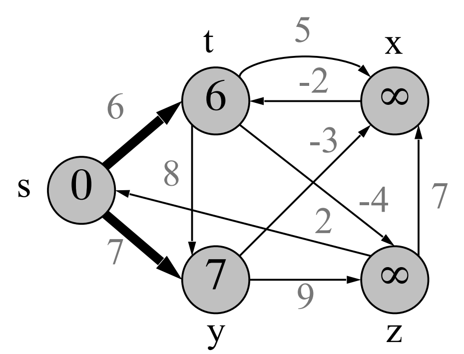


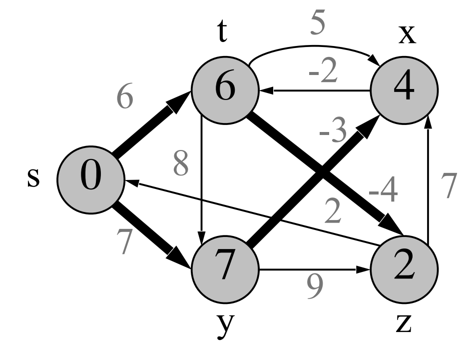
Bellman-Ford Example

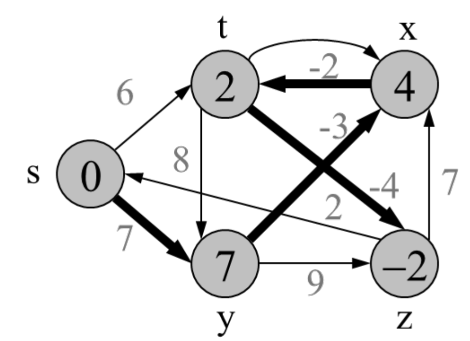




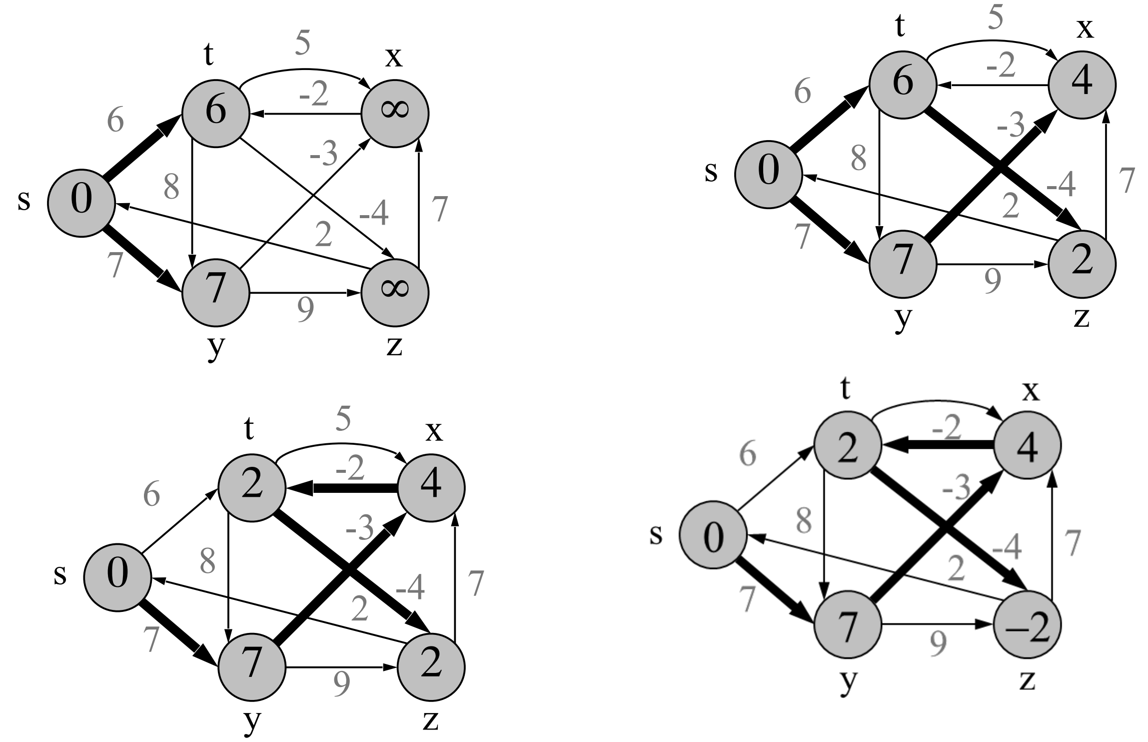
Bellman-Ford Example



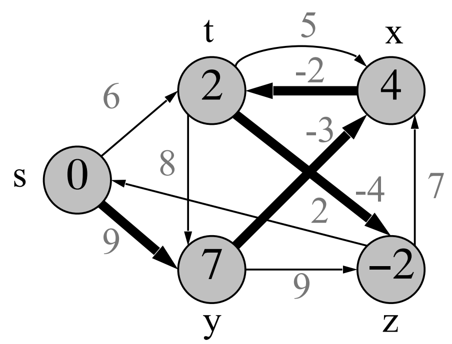




Bellman-Ford Example



Bellman-Ford Example



- Bellman-Ford running time:
  - (|V|-1)|E| + |E| = Θ(VE)

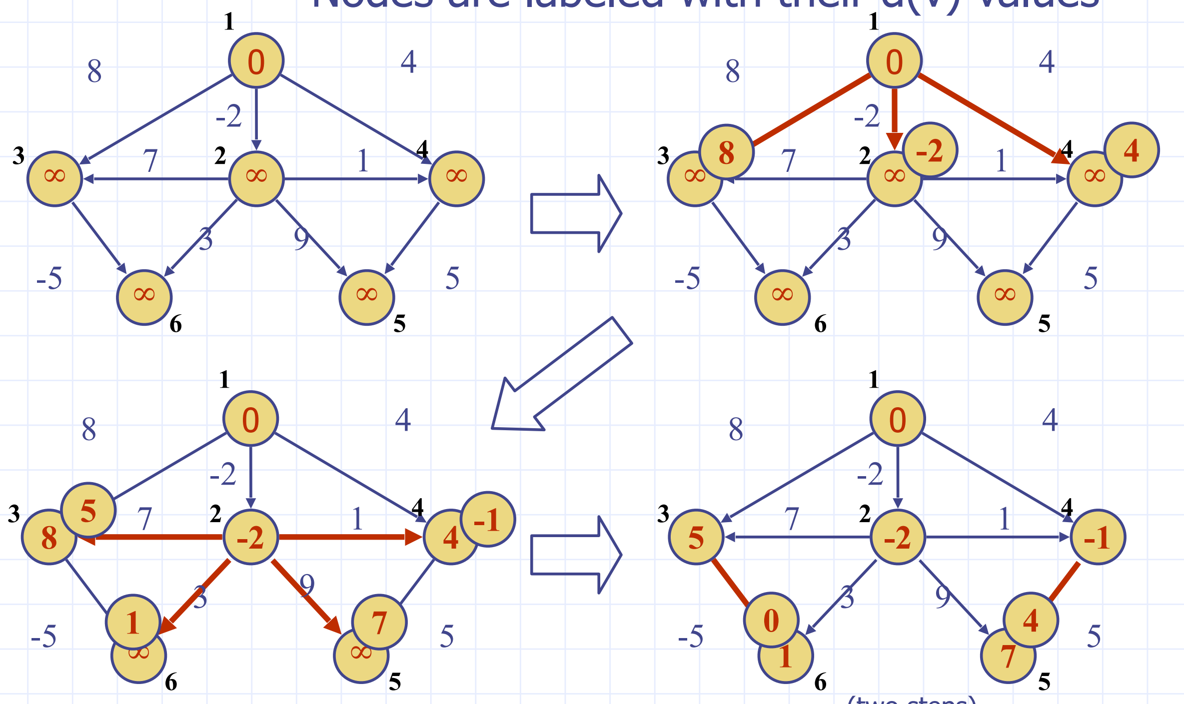
DAG-based Algorithm (not in book)

- Works even with negative-weight edges
- Uses topological order
- Doesn't use any fancy data structures
- Is much faster than Dijkstra's algorithm
- Running time: O(n+m).

```
Algorithm DagDistances(G, s)
  for all v ∈ G.vertices()
    if v = s
      v.setDistance(0)
    else
      v.setDistance(∞)
  { Perform a topological sort of the vertices }
  for u ← 1 to n do    {in topological order}
    for each e ∈ u.outEdges()
      { relax edge e }
      z ← e.opposite(u)
      r ← u.getDistance() + e.weight()
      if r < z.getDistance()
      z.setDistance(r)
```

DAG Example

Nodes are labeled with their d(v) values



(two steps)