

Summary

- ▶ Definitions
- ▶ Floyd-Warshall algorithm
- ▶ Bellman-Ford-Moore algorithm
- ▶ Dijkstra algorithm

Definition: weight of a path

- ▶ Consider a directed, weighted graph $G=(V,E)$, with weight function $w: E \rightarrow \mathbf{R}$
 - ▶ This is the general case: undirected or un-weighted are automatically included
- ▶ The weight $w(p)$ of a path p is the sum of the weights of the edges composing the path

$$w(p) = \sum_{(u,v) \in p} w(u, v)$$

Definition: shortest path

- ▶ The shortest path between vertex u and vertex v is defined as the minimum-weight path between u and v , if the path exists.
- ▶ The weight of the shortest path is represented as $\delta(u,v)$
- ▶ If v is not reachable from u , then $\delta(u,v)=\infty$

Finding shortest paths

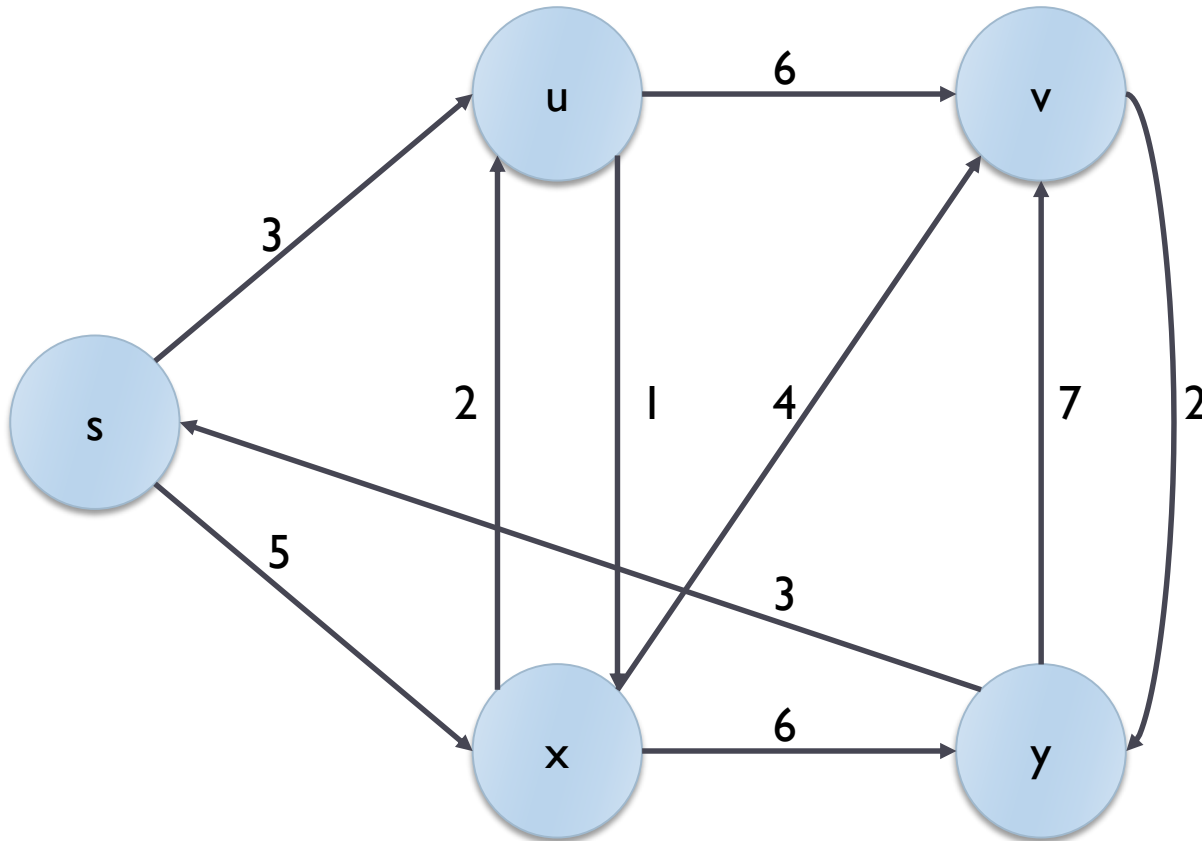
- ▶ **Single-source shortest path (SS-SP)**
 - ▶ Given u and v , find the shortest path between u and v
 - ▶ Given u , find the shortest path between u and any other vertex
- ▶ **All-pairs shortest path (AP-SP)**
 - ▶ Given a graph, find the shortest path between any pair of vertices

What to find?

- ▶ Depending on the problem, you might want:
 - ▶ The **value** of the shortest path weight
 - ▶ Just a real number
 - ▶ The **actual path** having such minimum weight
 - ▶ For simple graphs, a sequence of vertices. For multigraphs, a sequence of edges

Example

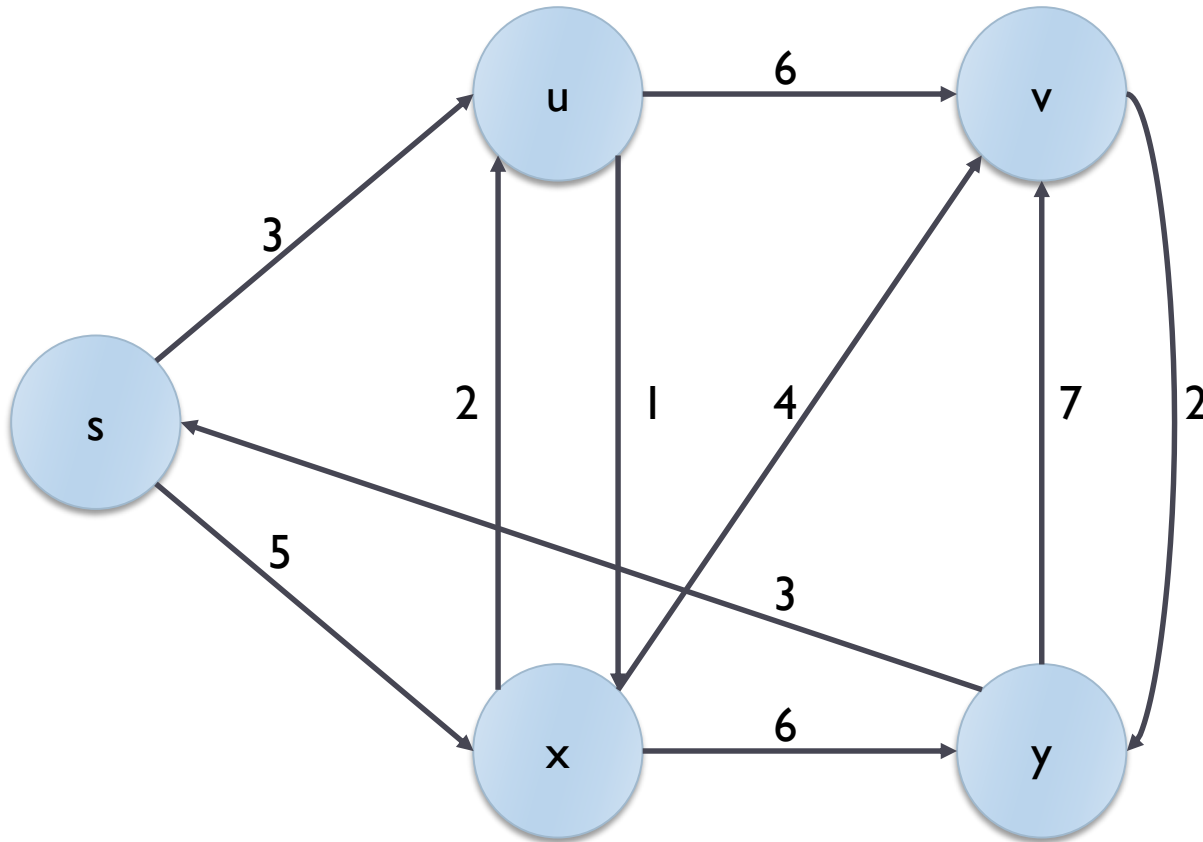
What is the shortest path between s and v ?



Representing shortest paths

- ▶ To store all shortest paths from a single source u , we may add
 - ▶ For each vertex v , the weight of the shortest path $\delta(u,v)$
 - ▶ For each vertex v , the “preceding” vertex $\pi(v)$ that allows to reach v in the shortest path
 - ▶ For multigraphs, we need the preceding edge
- ▶ **Example:**
 - ▶ Source vertex: u
 - ▶ For any vertex v :
 - ▶ `double v.weight ;`
 - ▶ `Vertex v.preceding ;`

Example



π

Vertex	Previous
s	NULL
u	s
x	u
v	x
y	v

δ

Vertex	Weight
s	0
u	3
x	4
v	8
y	10

Lemma

- ▶ The “previous” vertex in an intermediate node of a minimum path does not depend on the final destination
- ▶ **Example:**
 - ▶ Let p_1 = shortest path between u and v_1
 - ▶ Let p_2 = shortest path between u and v_2
 - ▶ Consider a vertex $w \in p_1 \cap p_2$
 - ▶ The value of $\pi(w)$ may be chosen in a single way and still guarantee that both p_1 and p_2 are shortest

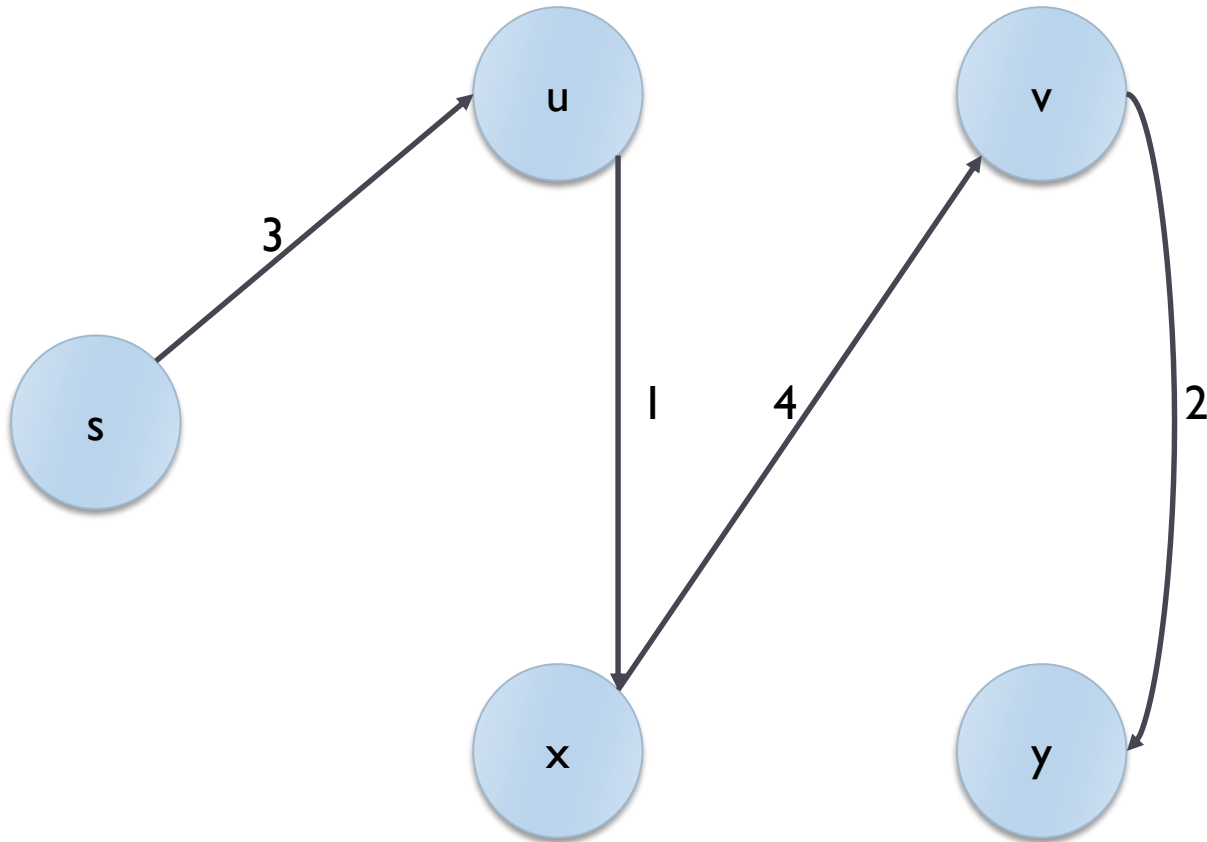
Shortest path graph

- ▶ Consider a source node u
- ▶ Compute all shortest paths from u
- ▶ Consider the relation $E\pi = \{ (v.\text{preceding}, v) \}$
- ▶ $E\pi \subseteq E$
- ▶ $V\pi = \{ v \in V : v \text{ reachable from } u \}$
- ▶ $G\pi = G(V\pi, E\pi)$ is a subgraph of $G(V, E)$
- ▶ $G\pi$: the predecessor-subgraph

Shortest path tree

- ▶ G_π is a tree (due to the Lemma) rooted in u
- ▶ In G_π , the (unique) paths starting from u are always shortest paths
- ▶ G_π is not unique, but all possible G_π are equivalent (same weight for every shortest path)

Example



π

Vertex	Previous
s	NULL
u	s
x	u
v	x
y	v

δ

Vertex	Weight
s	0
u	3
x	4
v	8
y	10

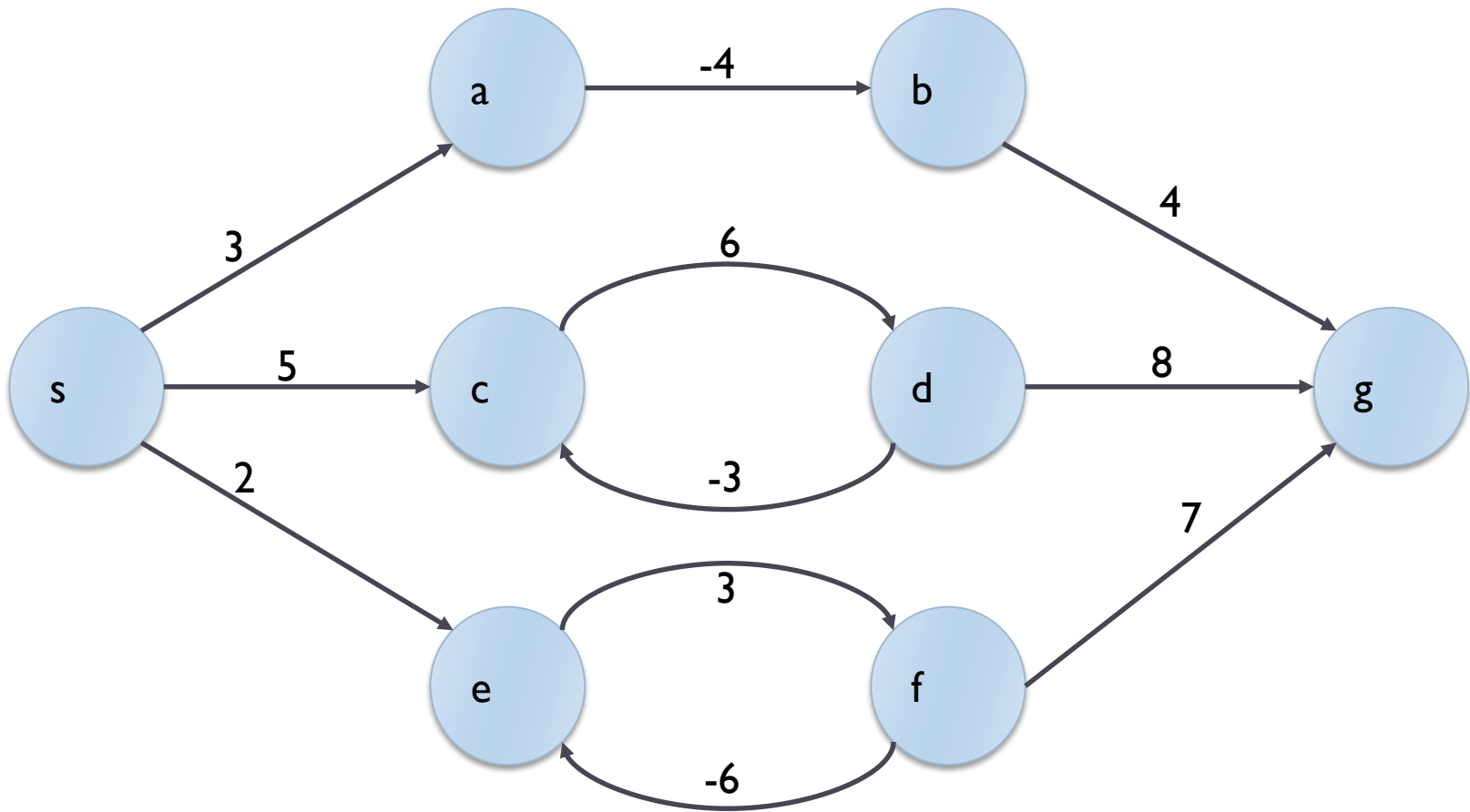
Special case

- ▶ If G is an un-weighted graph, then the shortest paths may be computed just with a breadth-first visit

Negative-weight cycles

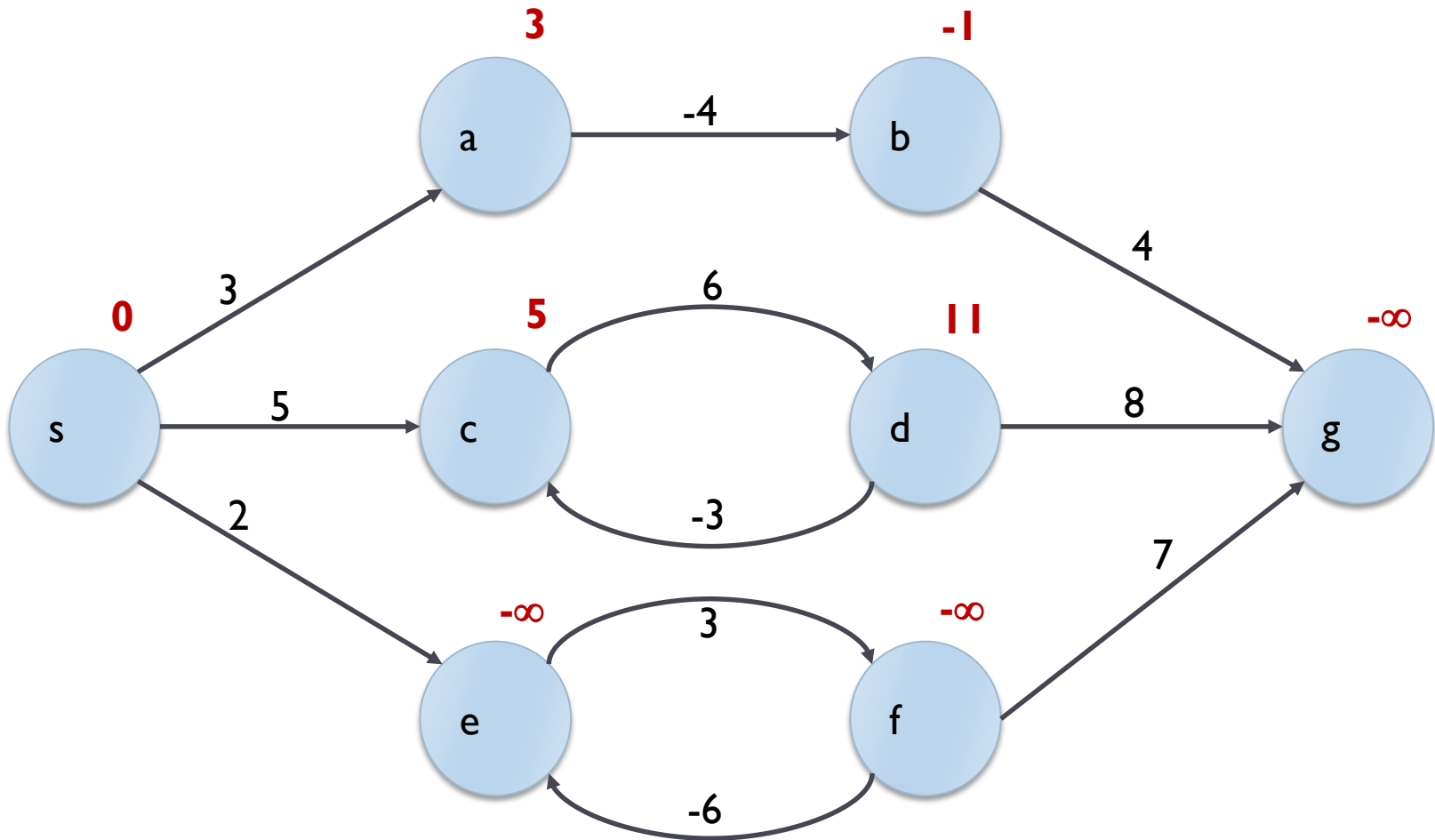
- ▶ Minimum paths cannot be defined if there are negative-weight cycles in the graph
- ▶ In this case, the minimum path does not exist, because you may always decrease the path weight by going once more through the loop.
- ▶ Conventionally, in these case we say that the path weight is $-\infty$.

Example



Example

Minimum-weight paths from source vertex s



Lemma

- ▶ Consider an ordered weighted graph $G=(V,E)$, with weight function $w: E \rightarrow \mathbf{R}$.
- ▶ Let $p = \langle v_1, v_2, \dots, v_k \rangle$ a shortest path from vertex v_1 to vertex v_k .
- ▶ For all i, j such that $1 \leq i \leq j \leq k$, let $p_{ij} = \langle v_i, v_{i+1}, \dots, v_j \rangle$ be the sub-path of p , from vertex v_i to vertex v_j .
- ▶ Therefore, p_{ij} is a shortest path from v_i to v_j .

Corollary

- ▶ Let p be a shortest path from s to v
- ▶ Consider the vertex u , such that (u,v) is the last edge in the shortest path
- ▶ We may decompose p (from s to v) into:
 - ▶ A sub-path from s to u
 - ▶ The final edge (u,v)
- ▶ Therefore
 - ▶ $\delta(s,v) = \delta(s,u) + w(u,v)$

Lemma

- ▶ If we chose arbitrarily the vertex u' , then for all edges $(u',v) \in E$ we may say that
 - ▶ $\delta(s,v) \leq \delta(s,u') + w(u',v)$

Relaxation

- ▶ Most shortest-path algorithms are based on the relaxation technique
- ▶ It consists of
 - ▶ Vector $d[u]$ represents $\delta(s,u)$
 - ▶ Keeping track of an updated estimate $d[u]$ of the shortest path towards each node u
 - ▶ Relaxing (i.e., updating) $d[v]$ (and therefore the predecessor $\pi[v]$) whenever we discover that node v is more conveniently reached by traversing edge (u,v)

Initial state

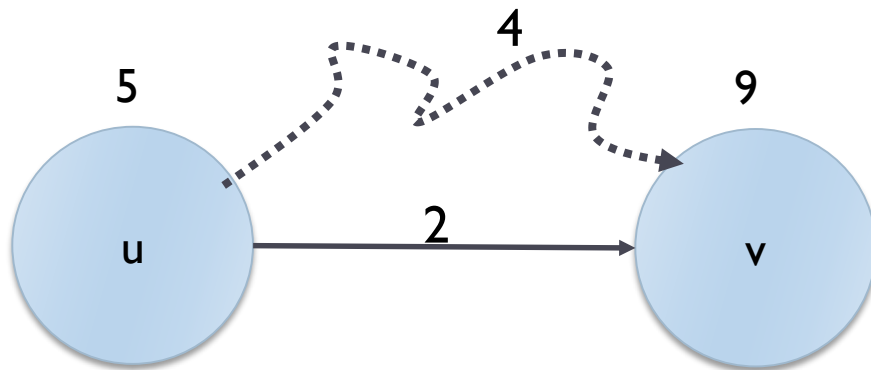
► **Initialize-Single-Source($G(V,E), s$)**

1. **for** all vertices $v \in V$
2. **do**
 1. $d[v] \leftarrow \infty$
 2. $\pi[v] \leftarrow \text{NIL}$
3. $d[s] \leftarrow 0$

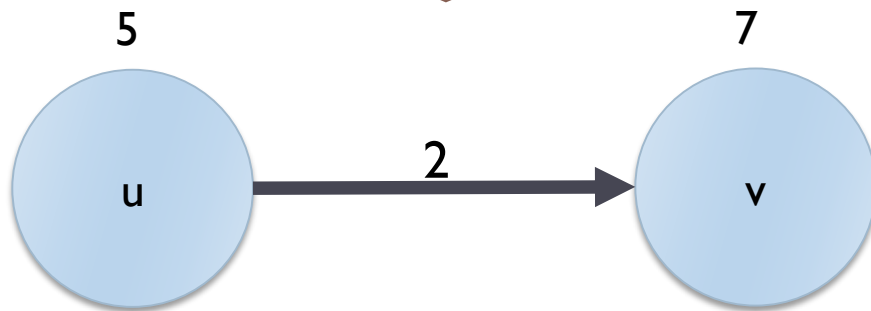
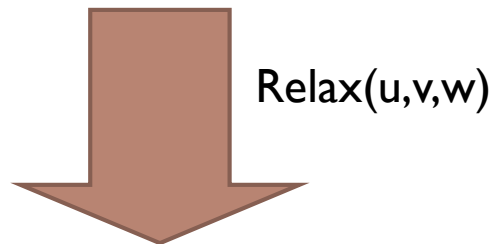
Relaxation

- ▶ We consider an edge (u,v) with weight w
- ▶ Relax(u, v, w)
 1. **if** $d[v] > d[u] + w(u,v)$
 2. **then**
 1. $d[v] \leftarrow d[u] + w(u,v)$
 2. $\pi[v] \leftarrow u$

Example 1

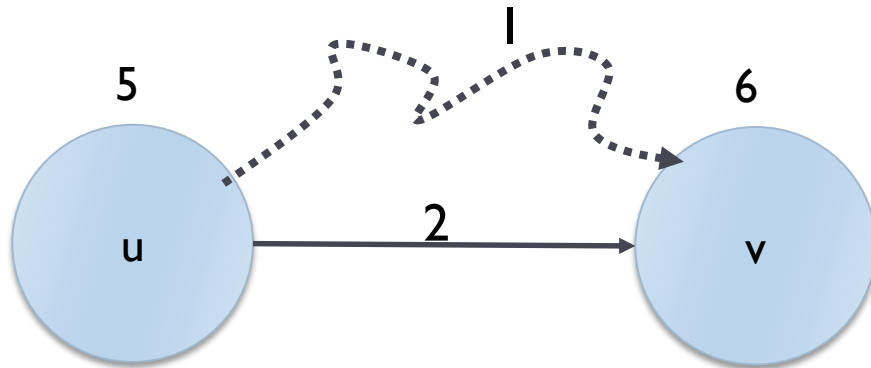


Before:
Shortest known path to v weights 9, does not contain (u,v)

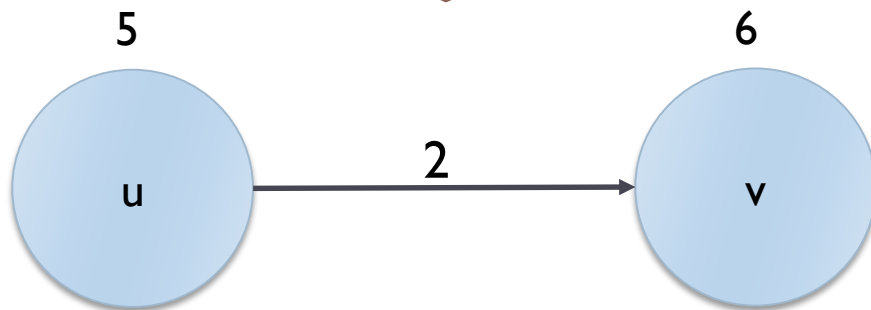
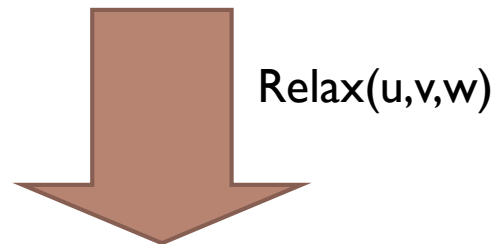


After:
Shortest path to v weights 7, the path includes (u,v)

Example 2



Before:
Shortest path to v
weights 6, does not
contain (u,v)



After:
No relaxation possible,
shortest path unchanged

Lemma

- ▶ Consider an ordered weighted graph $G=(V,E)$, with weight function $w: E \rightarrow \mathbf{R}$.
- ▶ Let (u,v) be an edge in G .
- ▶ After relaxation of (u,v) we may write that:
 - ▶ $d[v] \leq d[u] + w(u,v)$

Lemma

- ▶ Consider an ordered weighted graph $G=(V,E)$, with weight function $w: E \rightarrow \mathbf{R}$ and source vertex $s \in V$. Assume that G has no negative-weight cycles reachable from s .
- ▶ Therefore
 - ▶ After calling Initialize-Single-Source(G,s), the predecessor subgraph G_π is a rooted tree, with s as the root.
 - ▶ Any relaxation we may apply to the graph does not invalidate this property.

Lemma

- ▶ Given the previous definitions.
- ▶ Apply any possible sequence of relaxation operations
- ▶ Therefore, for each vertex v
 - ▶ $d[v] \geq \delta(s,v)$
- ▶ Additionally, if $d[v] = \delta(s,v)$, then the value of $d[v]$ will not change anymore due to relaxation operations.

Shortest path algorithms

- ▶ Various algorithms
- ▶ Differ according to one-source or all-sources requirement
- ▶ Adopt repeated relaxation operations
- ▶ Vary in the order of relaxation operations they perform
- ▶ May be applicable (or not) to graph with negative edges (but no negative cycles)







Floyd-Warshall algorithm

Graphs: Finding shortest paths



Floyd-Warshall algorithm

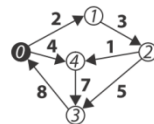
- ▶ Computes the all-source shortest path (AP-SP)
- ▶ $dist[i][j]$ is an n -by- n matrix that contains the length of a shortest path from v_i to v_j .
- ▶ if $dist[u][v]$ is ∞ , there is no path from u to v
- ▶ $pred[s][j]$ is used to reconstruct an actual shortest path: stores the predecessor vertex for reaching v_j starting from source v_s

FLOYD-WARSHALL			 Weighted Directed Graph	 Overflow
Best	Average	Worst		
$O(V^3)$	$O(V^3)$	$O(V^3)$	 Dynamic Programming	 2D Array

allPairsShortestPath (G)

1. **foreach** $u \in V$ **do**
2. **foreach** $v \in V$ **do**
3. $dist[u][v] = \infty$
4. $pred[u][v] = -1$
5. $dist[u][u] = 0$
6. **foreach** neighbor v of u **do**
7. $dist[u][v] = \text{weight of edge } (u,v)$
8. $pred[u][v] = u$
9. **foreach** $t \in V$ **do**
10. **foreach** $u \in V$ **do**
11. **foreach** $v \in V$ **do**
12. $newLen = dist[u][t] + dist[t][v]$
13. **if** ($newLen < dist[u][v]$) **then**
14. $dist[u][v] = newLen$
15. $pred[u][v] = pred[t][v]$
16. **end**
17. **end**

Initialize $dist[][]$ matrix with existing edges

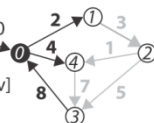


	0	1	2	3	4
0	0	2	∞	∞	4
1	∞	0	3	∞	∞
2	∞	∞	0	5	1
3	8	∞	∞	0	∞
4	∞	∞	∞	7	0

dist[u][v]

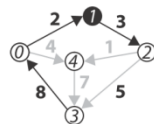
For each vertex $t \in V$, reduce paths between each pair of (u,v) vertices through t when possible

$t=1$




	0	1	2	3	4
0	0	2	∞	∞	4
1	∞	0	3	∞	∞
2	∞	∞	0	5	1
3	8	10	0	0	12
4	∞	∞	∞	7	0

$t=2$



	0	1	2	3	4
0	0	2	5	10	4
1	∞	0	3	∞	∞
2	∞	∞	0	5	1
3	8	10	13	0	12
4	∞	∞	∞	7	0

$t=3$



	0	1	2	3	4
0	0	2	5	10	4
1	16	0	3	8	4
2	13	15	0	5	1
3	8	10	13	0	12
4	15	17	20	7	0

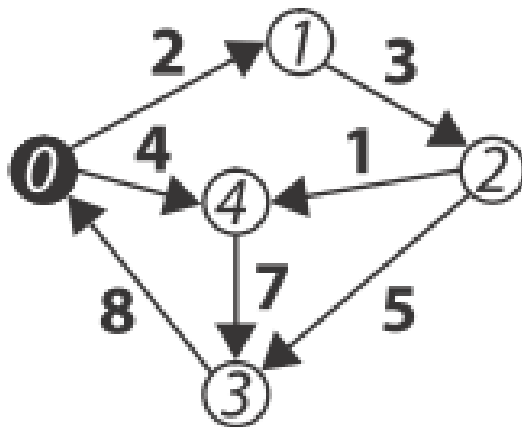
This is the final result since processing vertex 4 has no impact

Floyd-Warshall: initialization

allPairsShortestPath (G)

1. **foreach** $u \in V$ **do**
2. **foreach** $v \in V$ **do** Ⓢ
3. $\text{dist}[u][v] = \infty$
4. $\text{pred}[u][v] = -1$
5. $\text{dist}[u][u] = 0$
6. **foreach** neighbor v of u **do**
7. $\text{dist}[u][v] = \text{weight of edge } (u,v)$
8. $\text{pred}[u][v] = u$

Example, after initialization




	0	1	2	3	4
0	0	2	∞	∞	4
1	∞	0	3	∞	∞
2	∞	∞	0	5	1
3	8	∞	∞	0	∞
4	∞	∞	∞	7	0

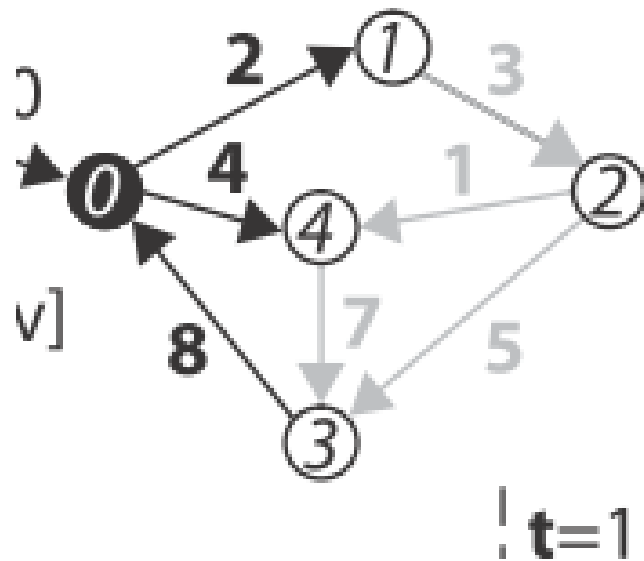
dist[u][v]

Floyd-Warshall: relaxation

```
9.  foreach  $t \in V$  do
10.   foreach  $u \in V$  do
11.    foreach  $v \in V$  do
12.     newLen = dist[u][t] + dist[t][v]
13.     if (newLen < dist[u][v]) then
14.       dist[u][v] = newLen
15.       pred[u][v] = pred[t][v]
```

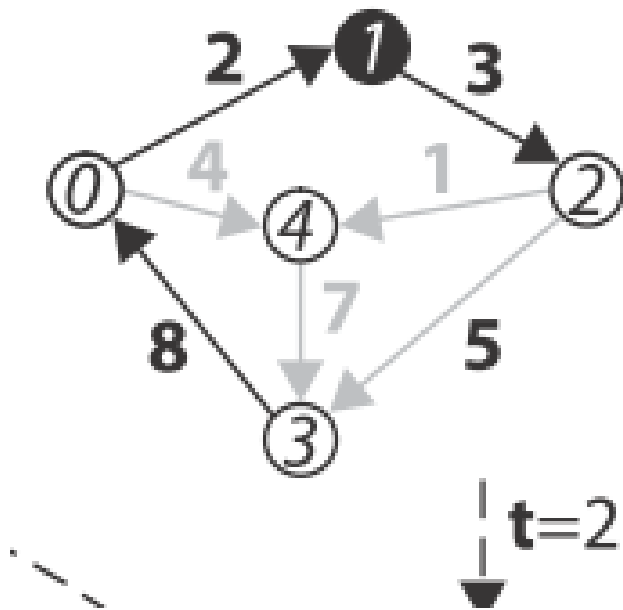


Example, after step $t=0$



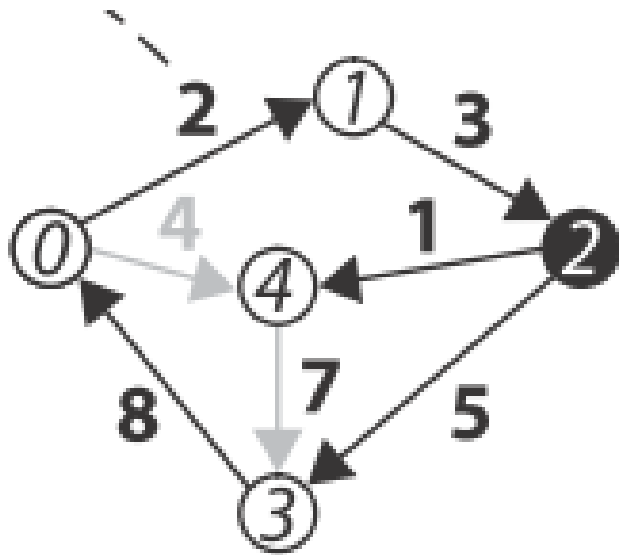
	0	1	2	3	4
0	0	2	∞	∞	4
1	∞	0	3	∞	∞
2	∞	∞	0	5	1
3	8	10	∞	0	12
4	∞	∞	∞	7	0

Example, after step $t=1$



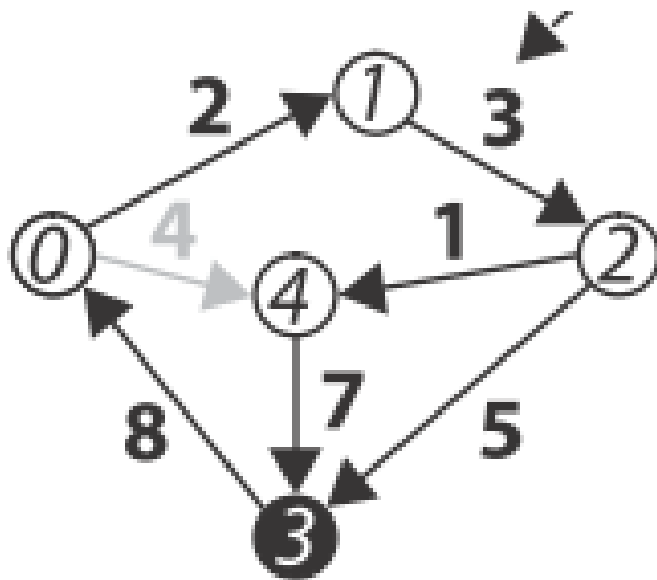
	0	1	2	3	4
0	0	2	5	∞	4
1	∞	0	3	∞	∞
2	∞	∞	0	5	1
3	8	10	13	0	12
4	∞	∞	∞	7	0

Example, after step $t=2$



	0	1	2	3	4
0	0	2	5	10	4
1	∞	0	3	8	4
2	∞	∞	0	5	1
3	8	10	13	0	12
4	∞	∞	∞	7	0

Example, after step $t=3$



	0	1	2	3	4
0	0	2	5	10	4
1	16	0	3	8	4
2	13	15	0	5	1
3	8	10	13	0	12
4	15	17	20	7	0

Complexity

- ▶ The Floyd-Warshall is basically executing 3 nested loops, each iterating over all vertices in the graph
- ▶ Complexity: $O(V^3)$

Implementation

org.jgrapht.alg

Class FloydWarshallShortestPaths<V,E>

java.lang.Object

└ org.jgrapht.alg.FloydWarshallShortestPaths<V,E>

```
public class FloydWarshallShortestPaths<V,E>  
    extends java.lang.Object
```

The [Floyd-Warshall algorithm](#) finds all shortest paths (all n^2 of them) in $O(n^3)$ time. This also works out the graph diameter during the process.

Author:

Tom Larkworthy, Soren Davidsen

Constructor Summary

[FloydWarshallShortestPaths](#)([Graph](#)<V,E> graph)

Method Summary

double	getDiameter ()
Graph <V,E>	getGraph ()
GraphPath <V,E>	getShortestPath (V a, V b) Get the shortest path between two vertices.
java.util.List< GraphPath <V,E>>	getShortestPaths (V v) Get shortest paths from a vertex to all other vertices in the graph.
int	getShortestPathsCount ()
double	shortestDistance (V a, V b) Get the length of a shortest path.

Bellman-Ford-Moore Algorithm

- ▶ Solution to the single-source shortest path (SS-SP) problem in graph theory
- ▶ Based on relaxation (for every vertex, relax all possible edges)
- ▶ Does not work in presence of negative cycles
 - ▶ but it is able to detect the problem
- ▶ $O(V \cdot E)$

Bellman-Ford-Moore Algorithm

```
dist[s] ← 0          (distance to source vertex is zero)
for all v ∈ V - {s}
  do dist[v] ← ∞    (set all other distances to infinity)
for i ← 0 to |V|
  for all (u, v) ∈ E
    do if dist[v] > dist[u] + w(u, v)    (if new shortest path found)
       then d[v] ← d[u] + w(u, v)      (set new value of shortest path)
                                           (if desired, add traceback code)

for all (u, v) ∈ E    (sanity check)
  do if dist[v] > dist[u] + w(u, v)
     then PANIC!
```


Dijkstra's algorithm

- ▶ Solution to the single-source shortest path (SS-SP) problem in graph theory
- ▶ Works on both directed and undirected graphs
- ▶ All edges must have nonnegative weights
 - ▶ the algorithm would miserably fail
- ▶ Greedy
 - ... but guarantees the optimum!

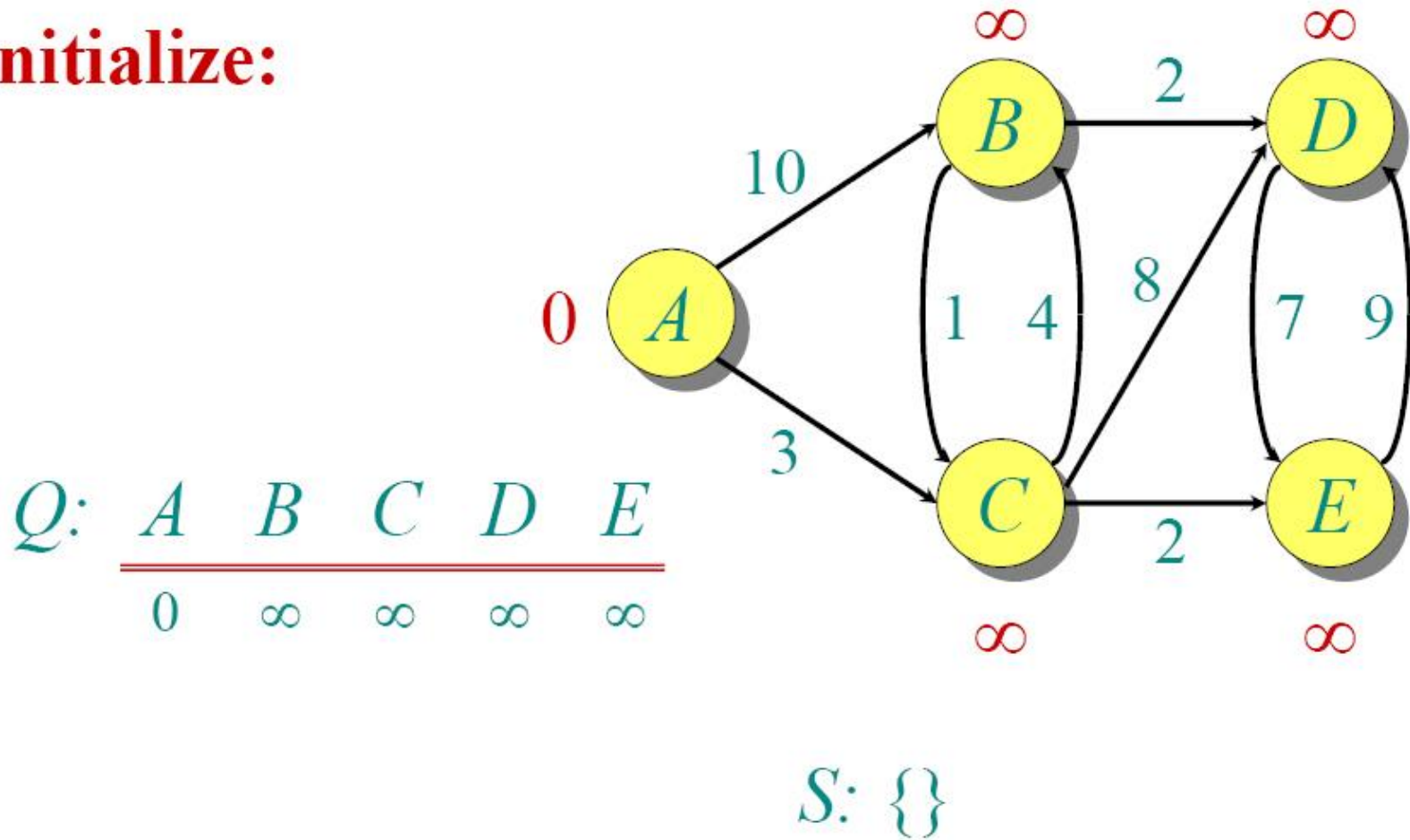


Dijkstra's algorithm

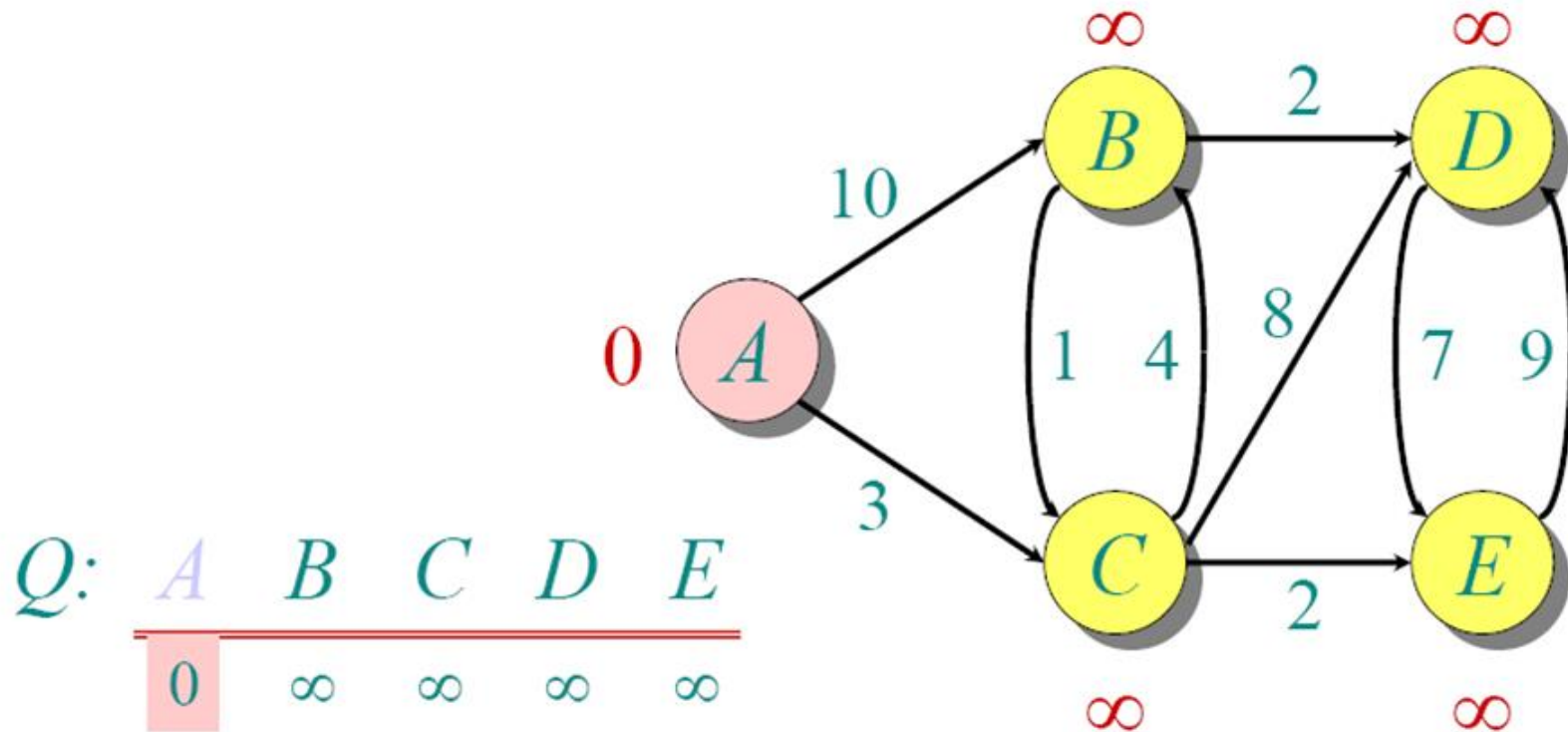
dist[s] ← 0 (distance to source vertex is zero)
for all $v \in V - \{s\}$
 do **dist[v] ← ∞** (set all other distances to infinity)
S ← ∅ (S, the set of visited vertices is initially empty)
Q ← V (Q, the queue initially contains all vertices)
while **Q** ≠ ∅ (while the queue is not empty)
do **u** ← mindistance(Q, dist) (select $e \in Q$ with the min. distance)
 S ← S ∪ {u} (add u to list of visited vertices)
 for all $v \in \text{neighbors}[u]$
 do **if** **dist[v] > dist[u] + w(u, v)** (if new shortest path found)
 then **d[v] ← d[u] + w(u, v)** (set new value of shortest path)
 (if desired, add traceback code)

Dijkstra Animated Example

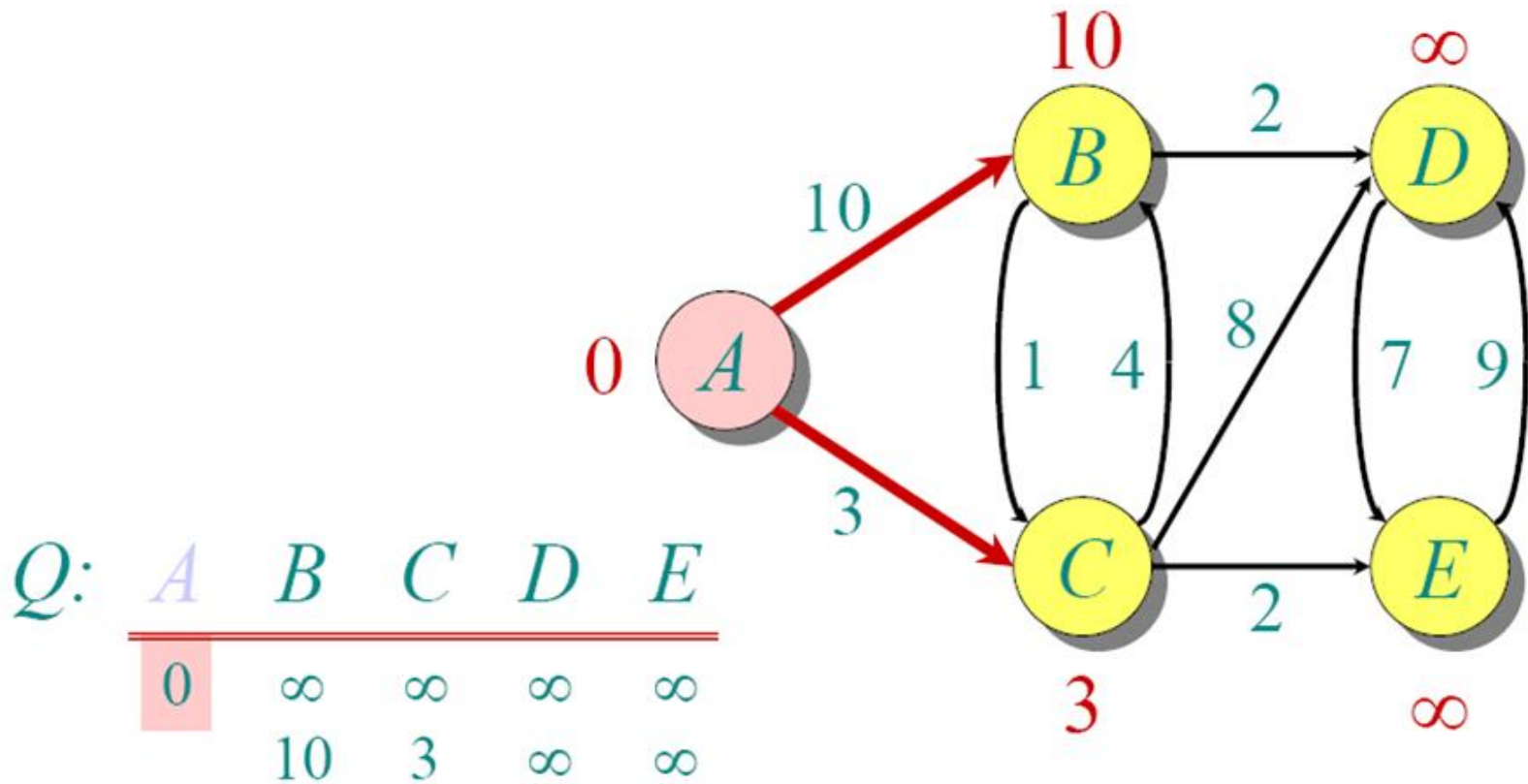
Initialize:



Dijkstra Animated Example

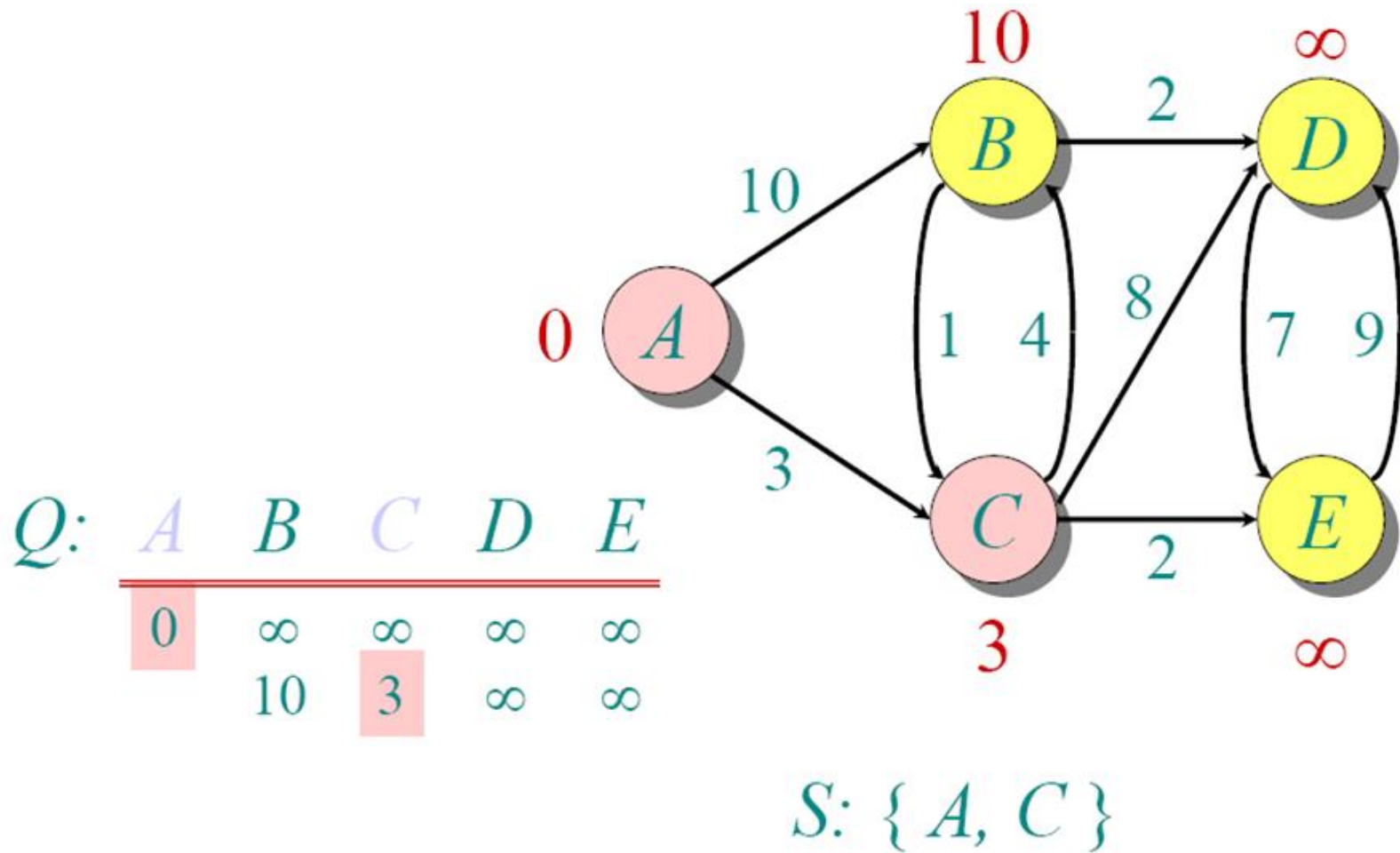


Dijkstra Animated Example

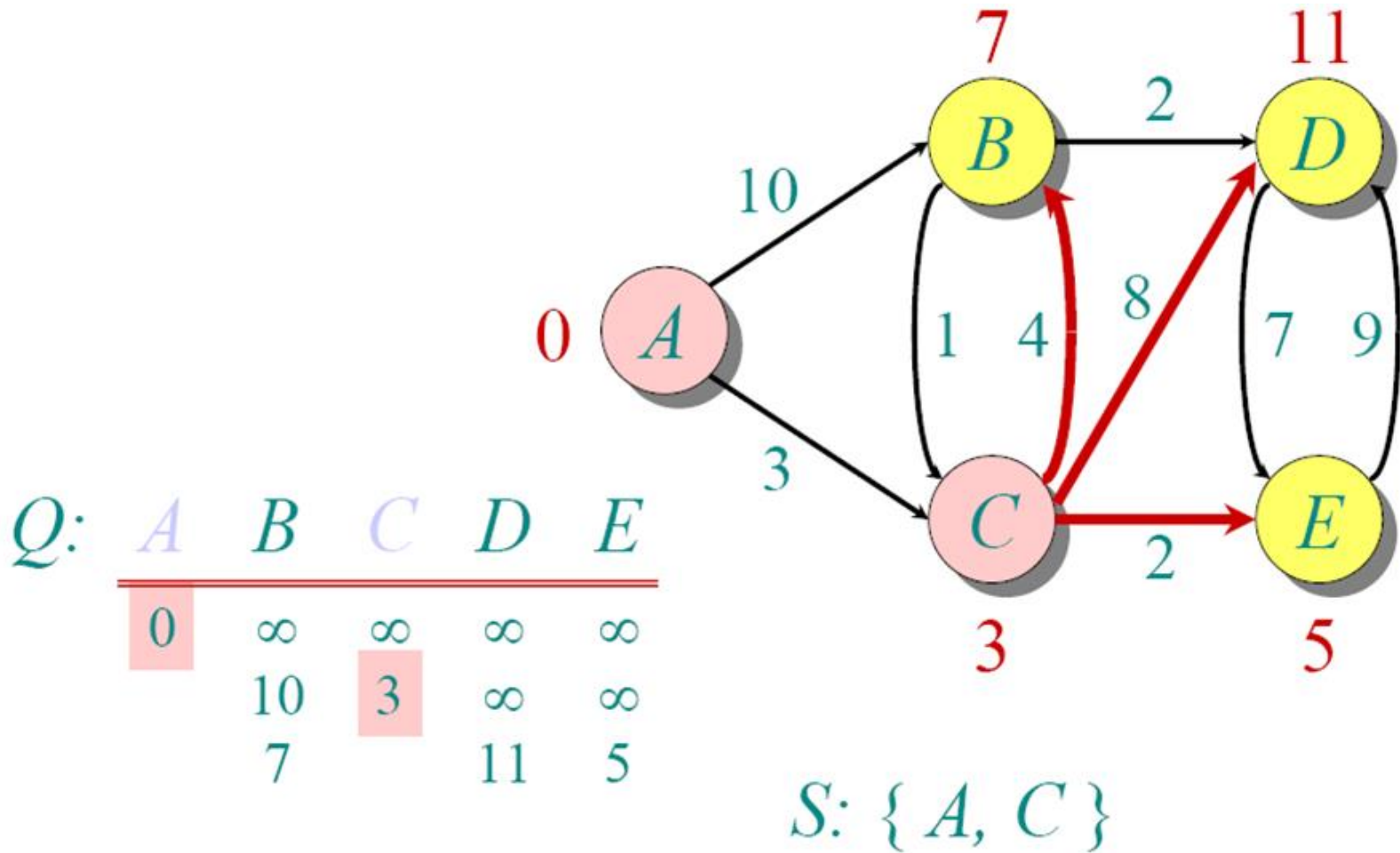


S: {A}

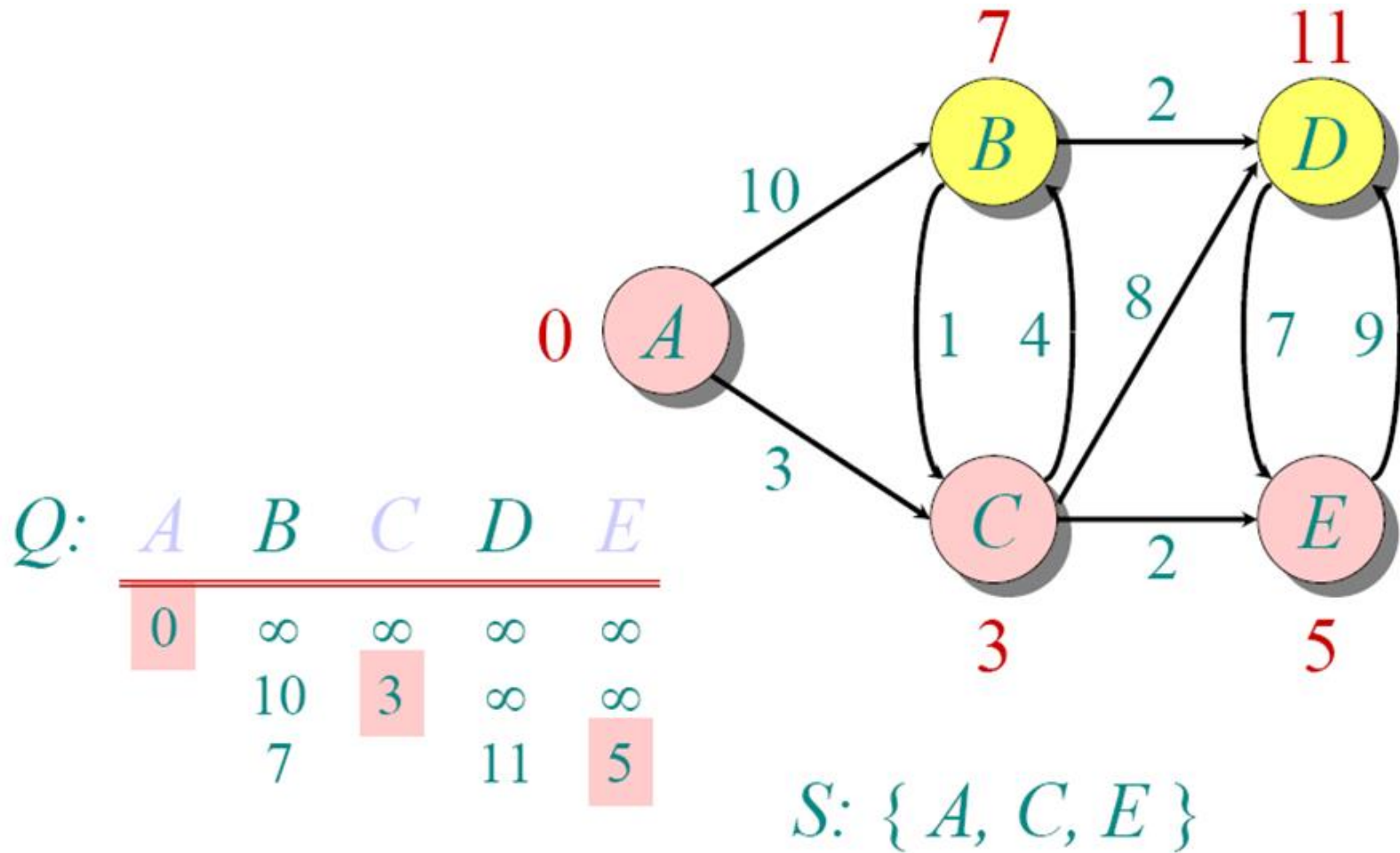
Dijkstra Animated Example



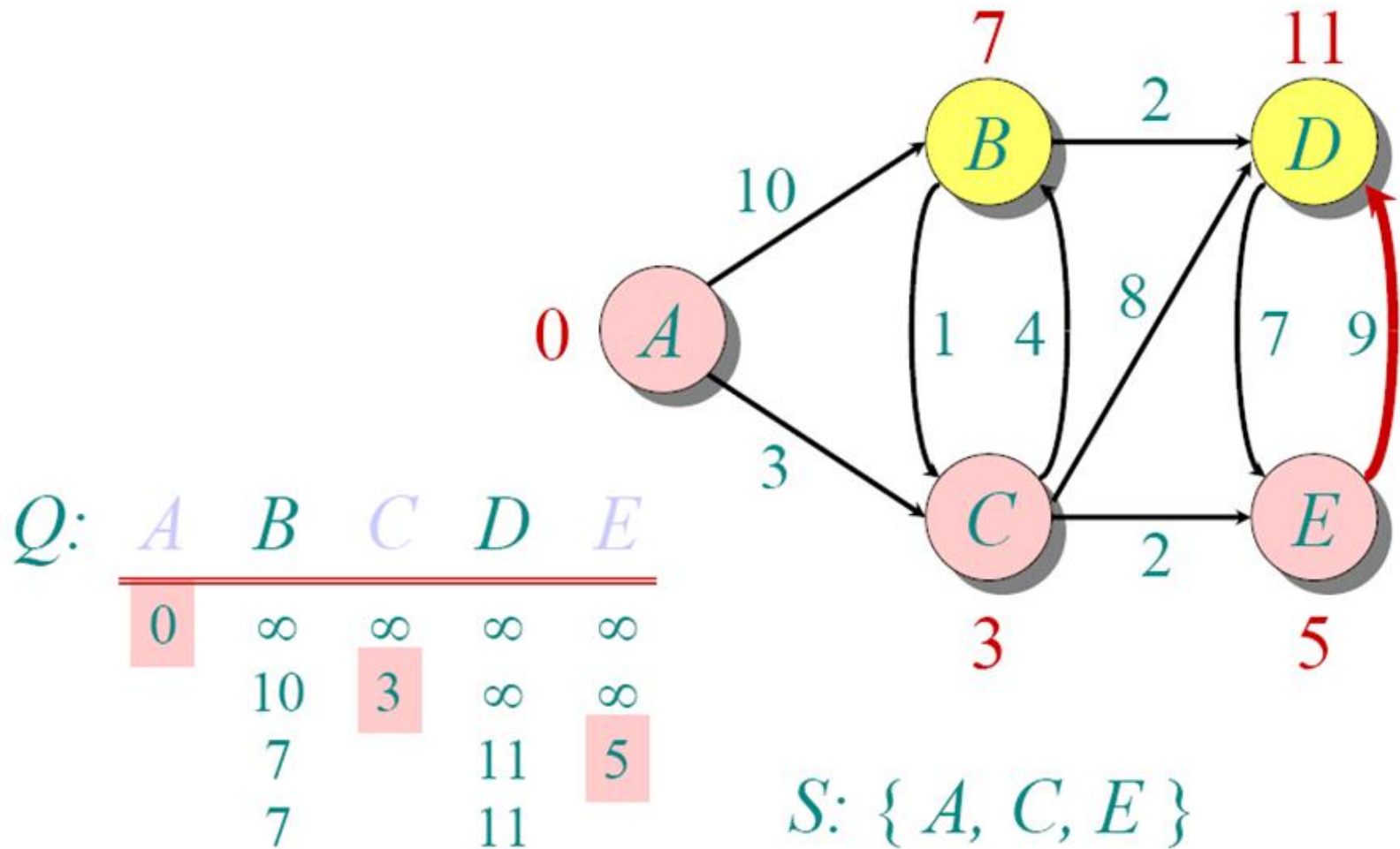
Dijkstra Animated Example



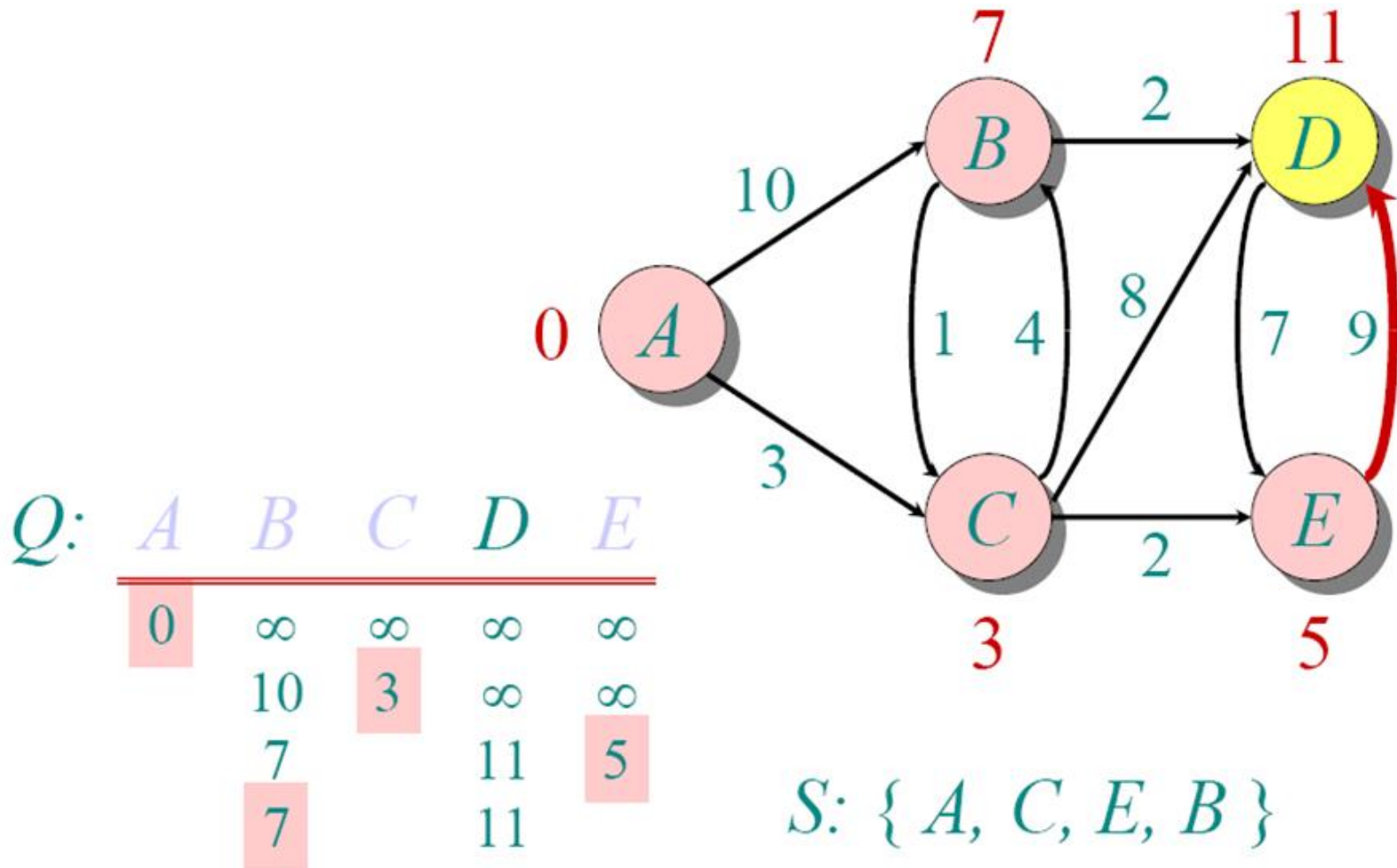
Dijkstra Animated Example



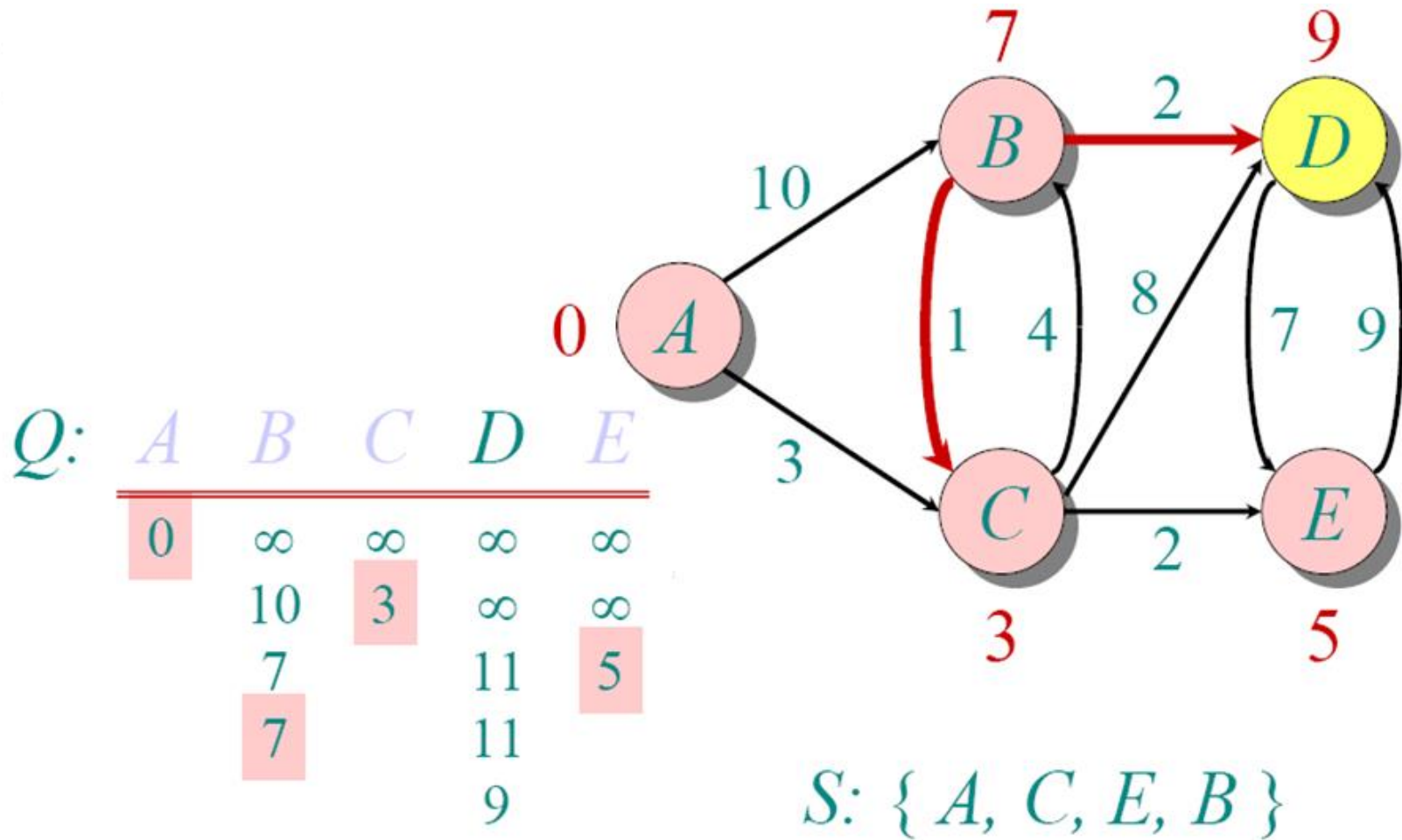
Dijkstra Animated Example



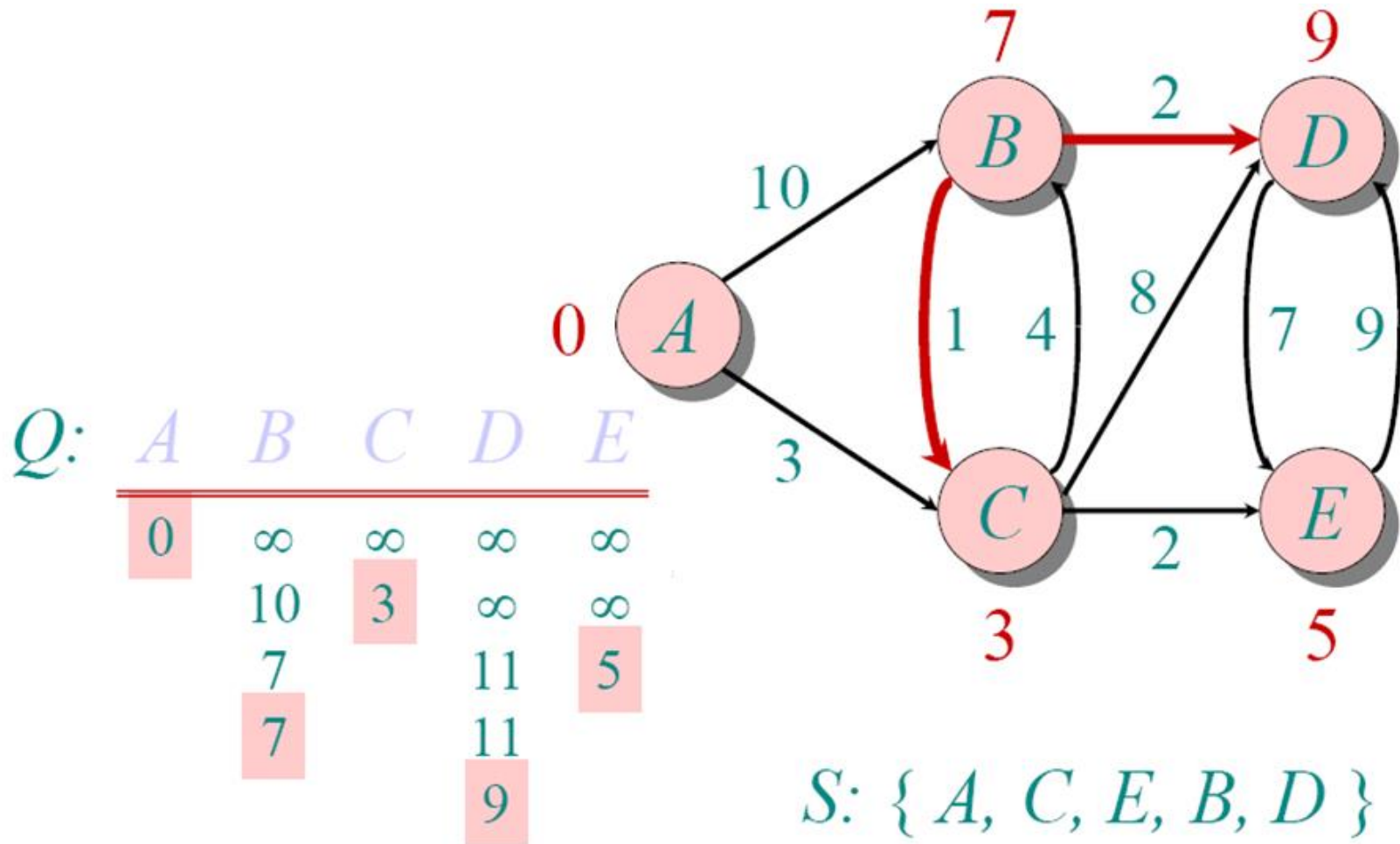
Dijkstra Animated Example



Dijkstra Animated Example



Dijkstra Animated Example



Why it works

- ▶ A formal proof would take longer than this presentation, but we can understand how the argument works intuitively
 - ▶ Think of Dijkstra's algorithm as a water-filling algorithm
 - ▶ Remember that all edge's weights are positive

Dijkstra efficiency

- ▶ The simplest implementation is:

$$O(E + V^2)$$

- ▶ But it can be implemented more efficiently:

$$O(E + V \cdot \log V)$$



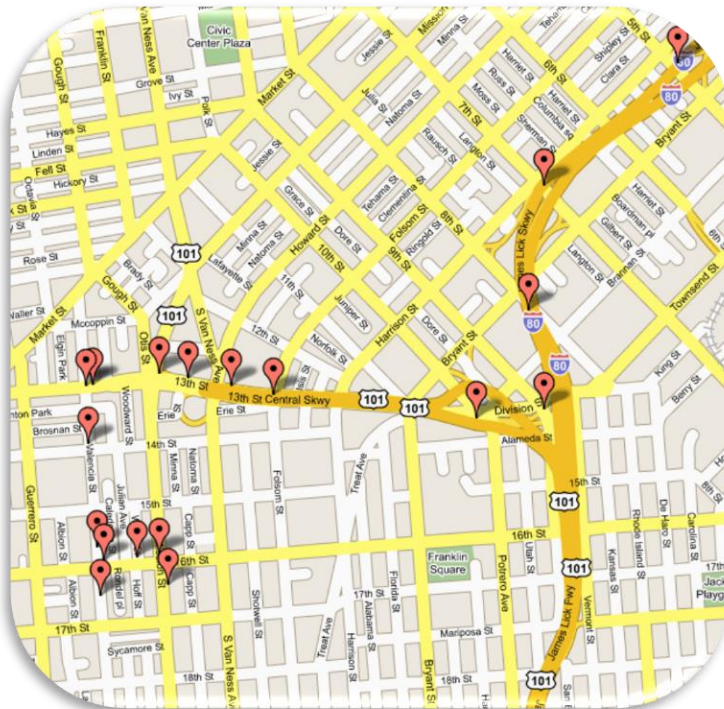
Floyd–Warshall: $O(V^3)$
Bellman-Ford-Moore : $O(V \cdot E)$

Applications

- ▶ Dijkstra's algorithm calculates the shortest path to every vertex from vertex s (SS-SP)
- ▶ It is about as computationally expensive to calculate the shortest path from vertex u to every vertex using Dijkstra's as it is to calculate the shortest path to some particular vertex t
- ▶ Therefore, anytime we want to know the optimal path to some other vertex t from a determined origin s , we can use Dijkstra's algorithm (and stop as soon t exit from Q)

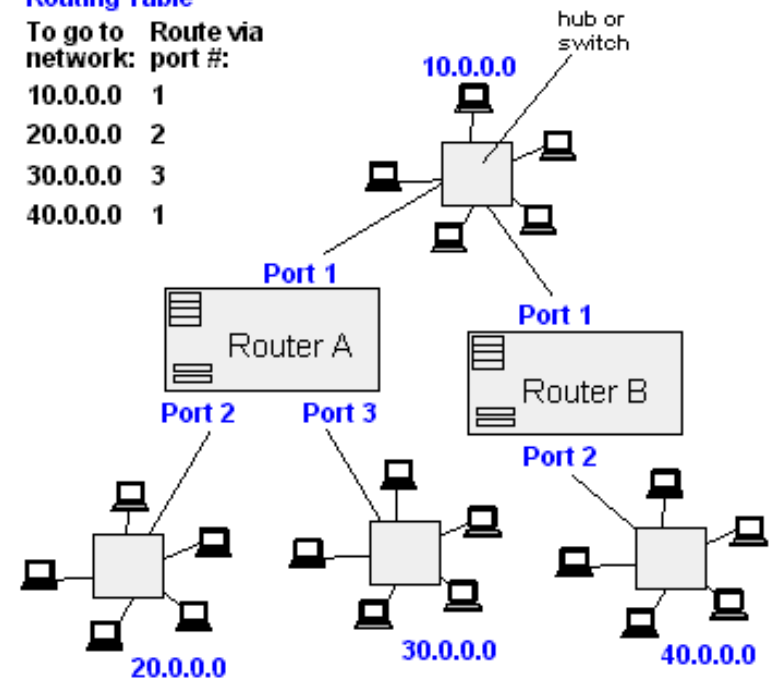
Applications

- ▶ Traffic Information Systems are most prominent use
- ▶ Mapping (Map Quest, Google Maps)
- ▶ Routing Systems



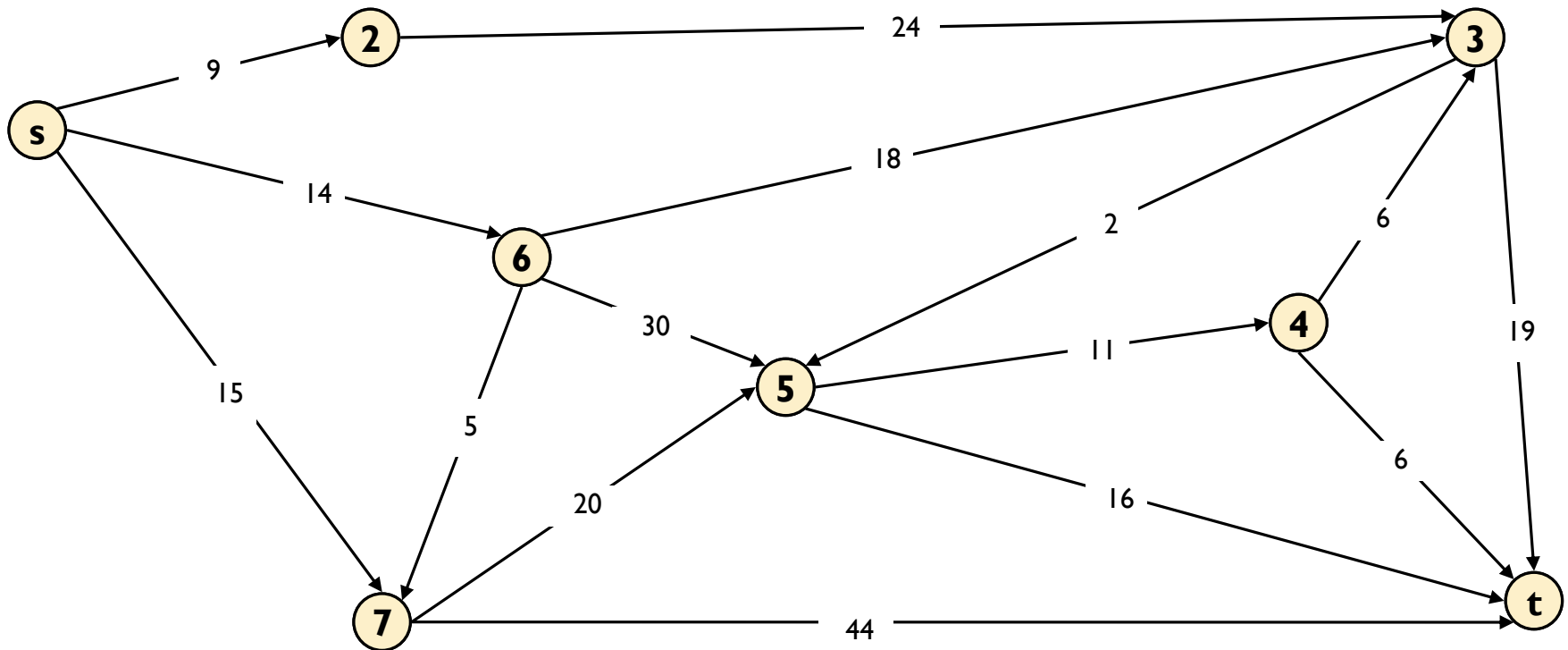
Router A Routing Table

To go to network:	Route via port #:
10.0.0.0	1
20.0.0.0	2
30.0.0.0	3
40.0.0.0	1



Dijkstra's Shortest Path Algorithm

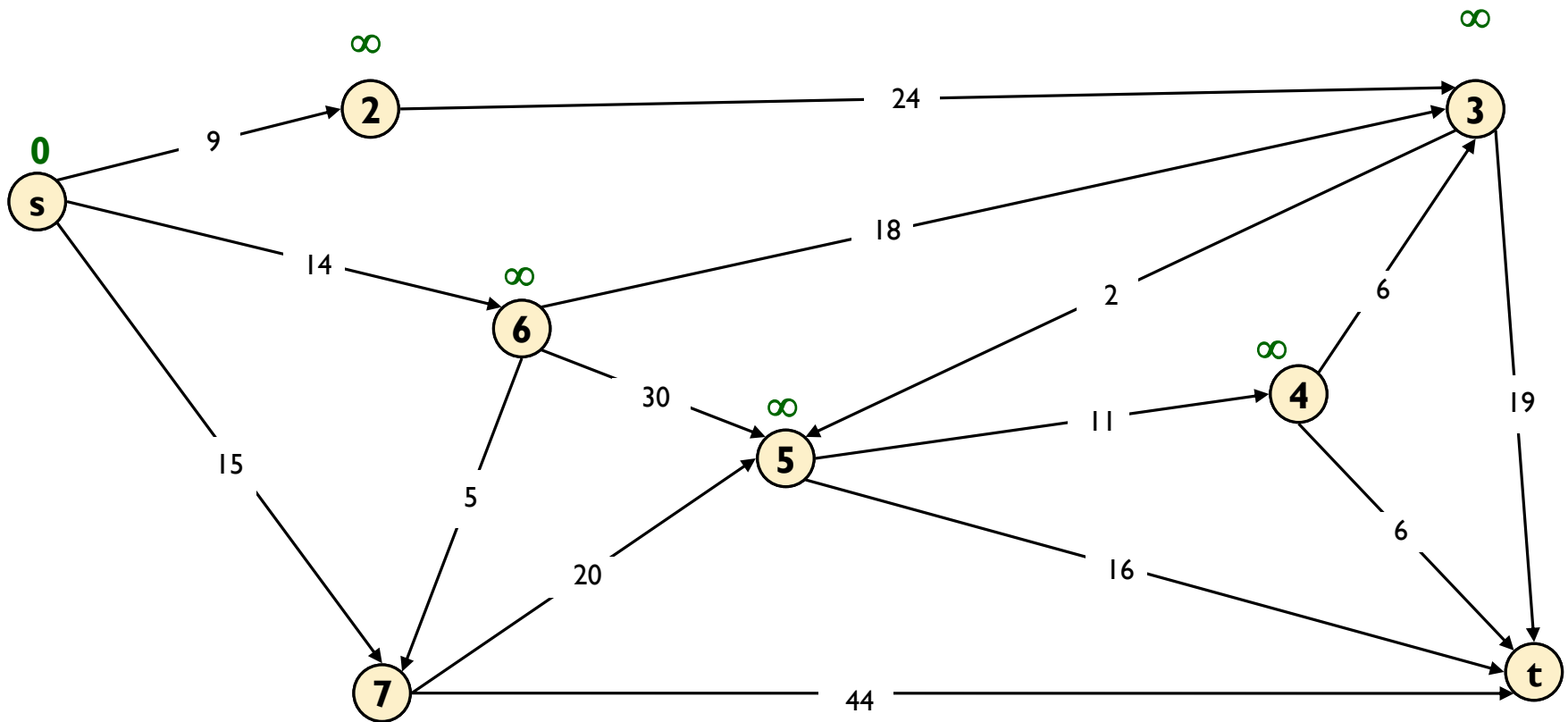
- ▶ Find shortest path from **s** to **t**



Dijkstra's Shortest Path Algorithm

$S = \{ \}$

$Q = \{ s, 2, 3, 4, 5, 6, 7, t \}$

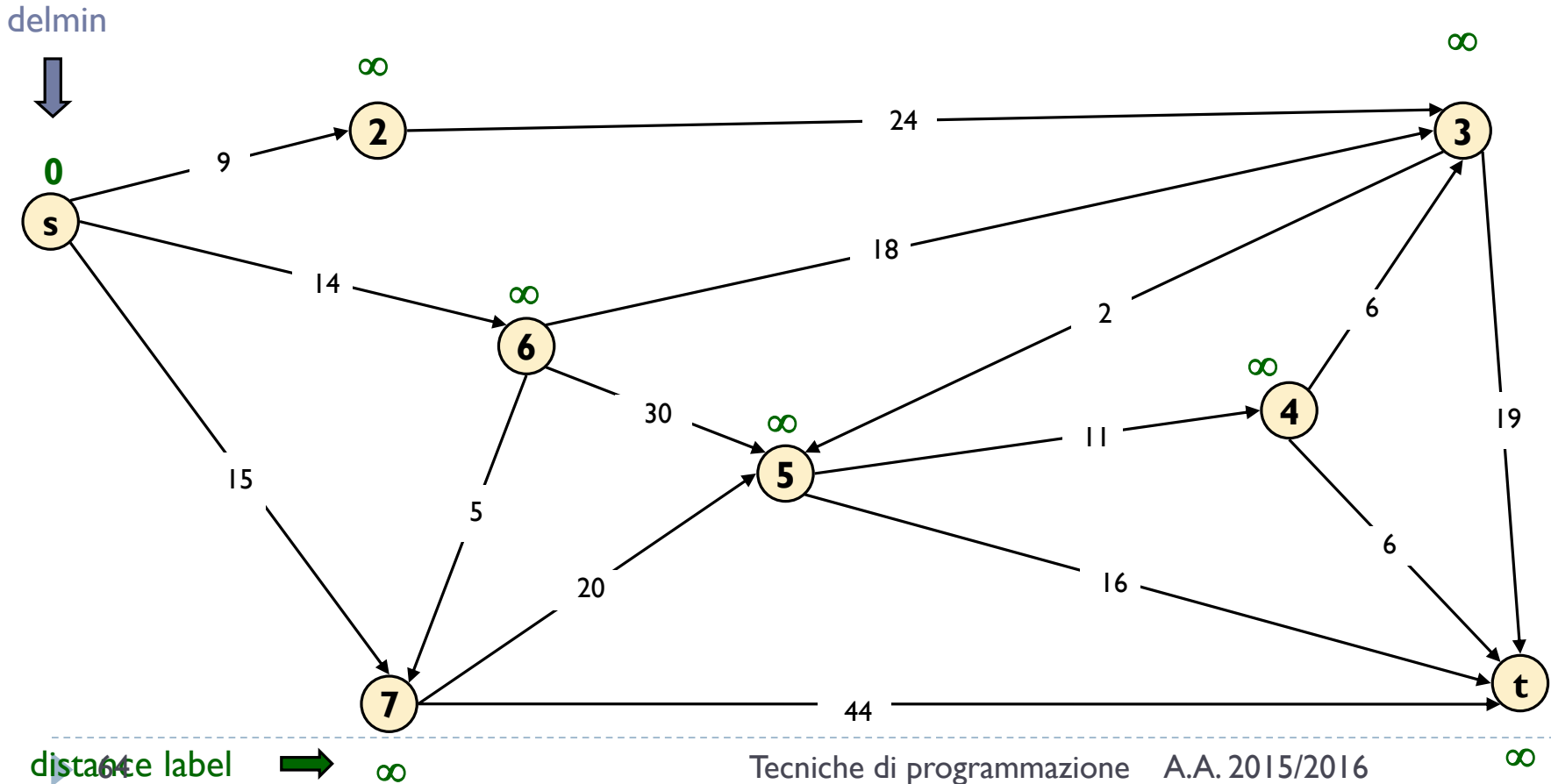


distance label $\rightarrow \infty$

Dijkstra's Shortest Path Algorithm

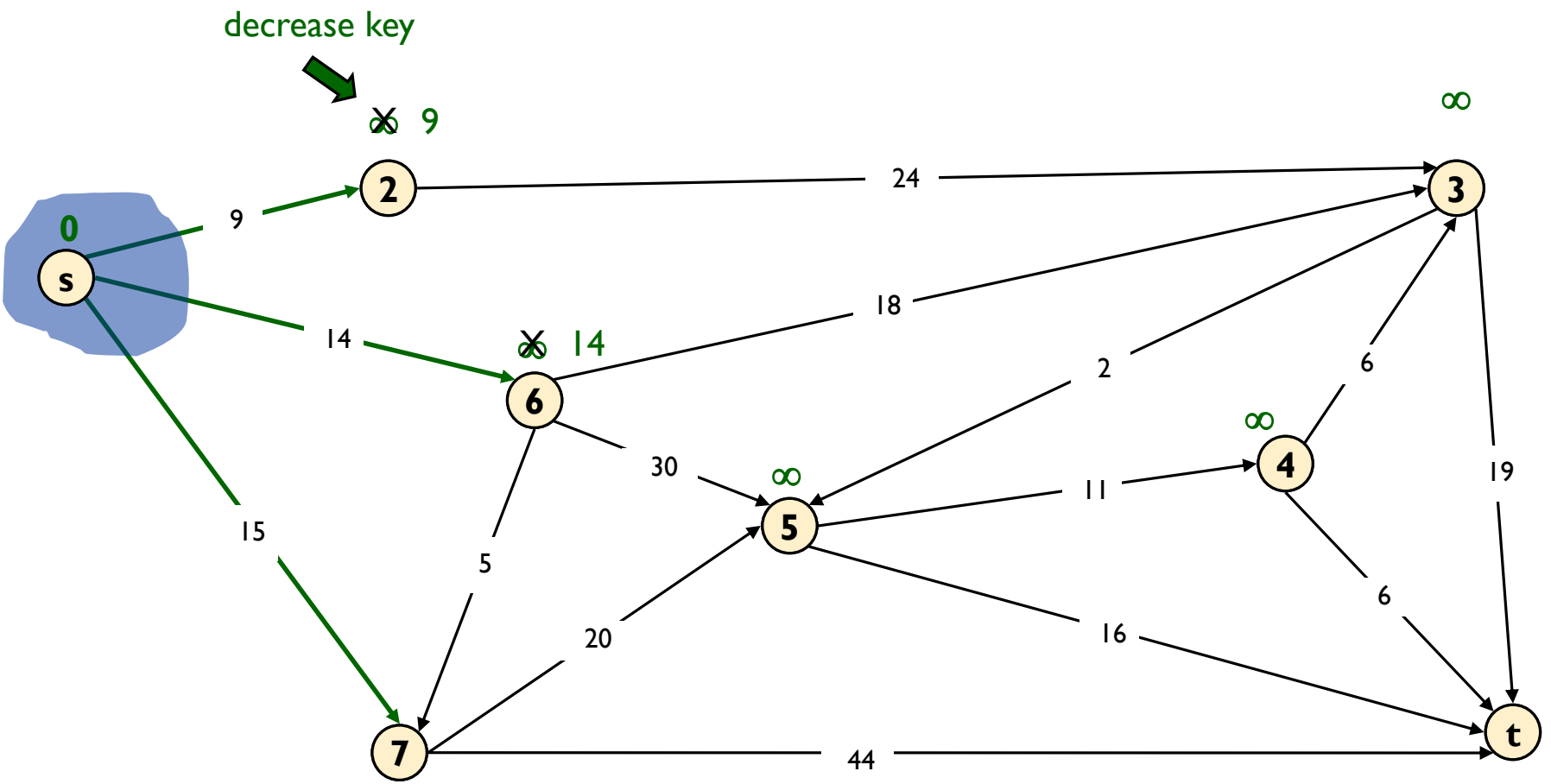
$S = \{ \}$

$Q = \{ s, 2, 3, 4, 5, 6, 7, t \}$



Dijkstra's Shortest Path Algorithm

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



decrease key



~~∞~~ 9

~~∞~~ 14

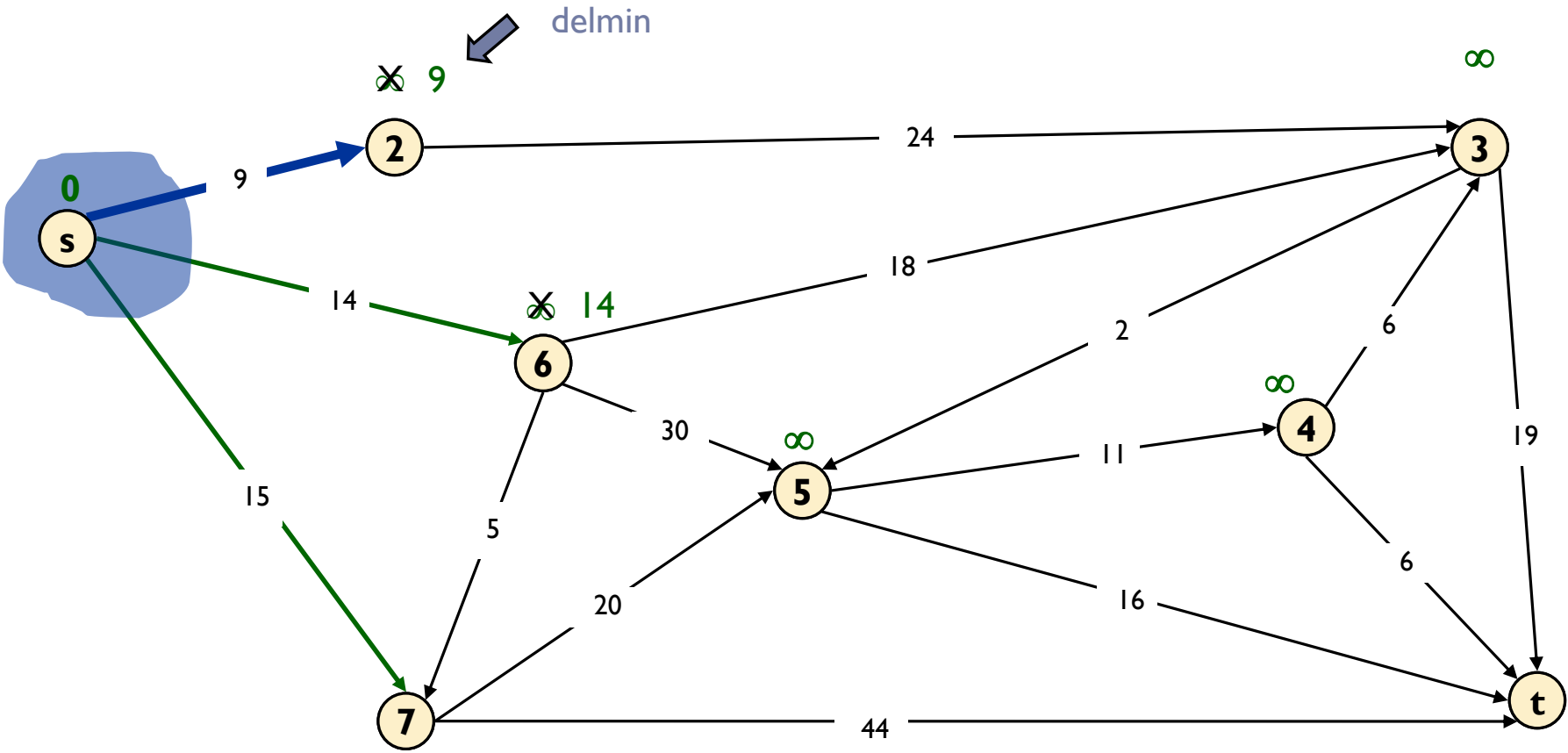
~~∞~~ 15

distance label



Dijkstra's Shortest Path Algorithm

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



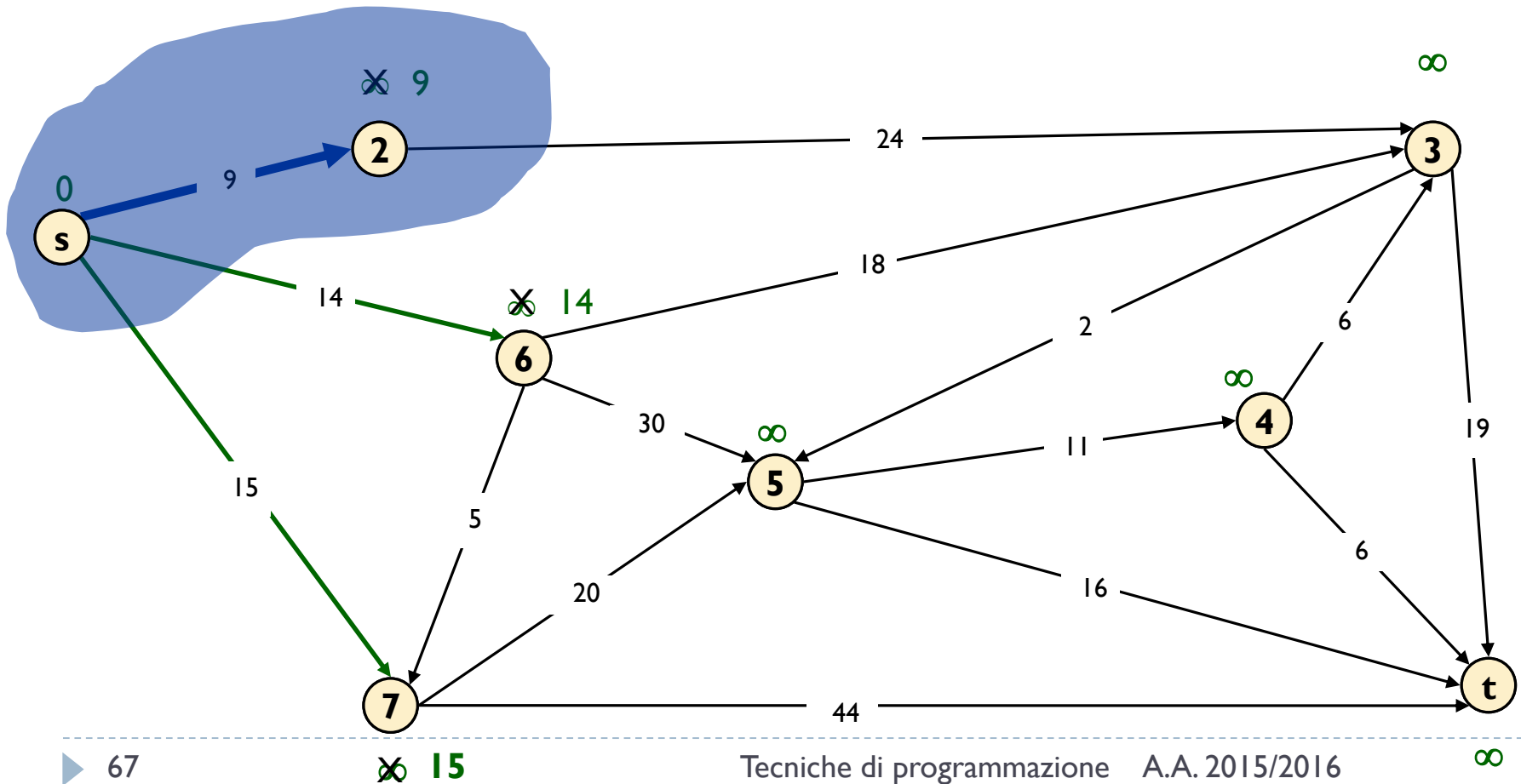
delmin

distance label → ~~15~~

Dijkstra's Shortest Path Algorithm

$S = \{s, 2\}$

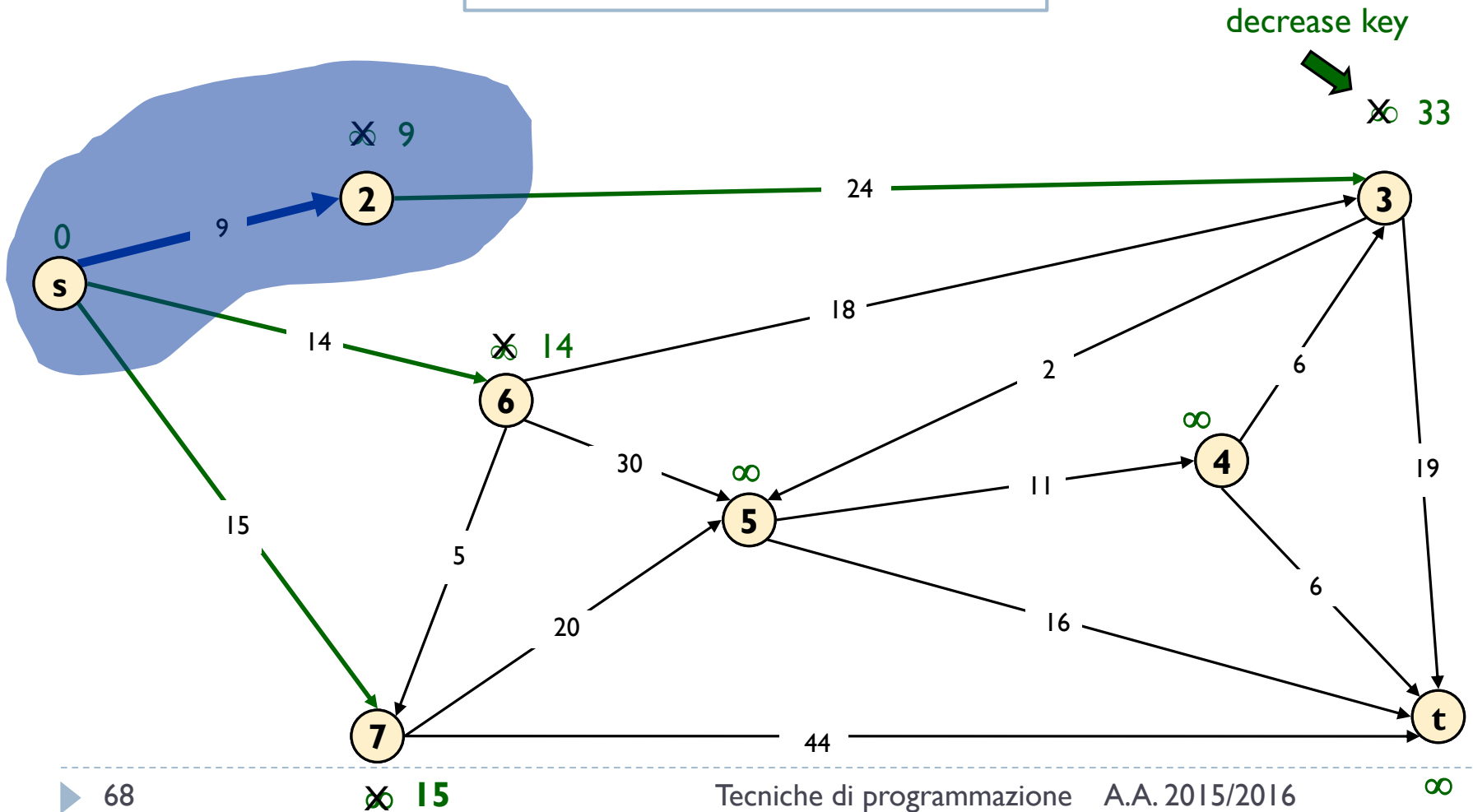
$Q = \{3, 4, 5, 6, 7, t\}$



Dijkstra's Shortest Path Algorithm

$S = \{s, 2\}$

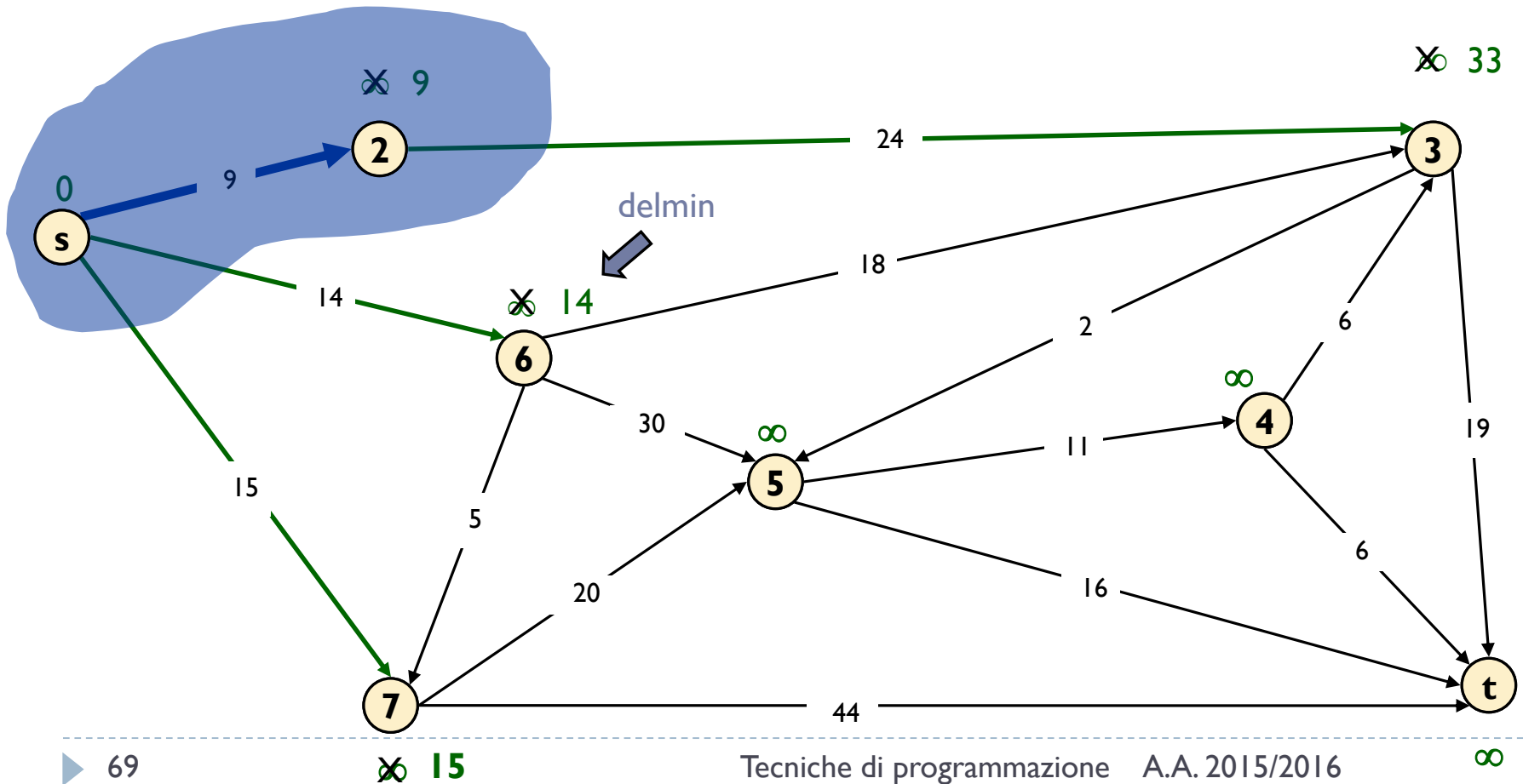
$Q = \{3, 4, 5, 6, 7, t\}$



Dijkstra's Shortest Path Algorithm

$S = \{s, 2\}$

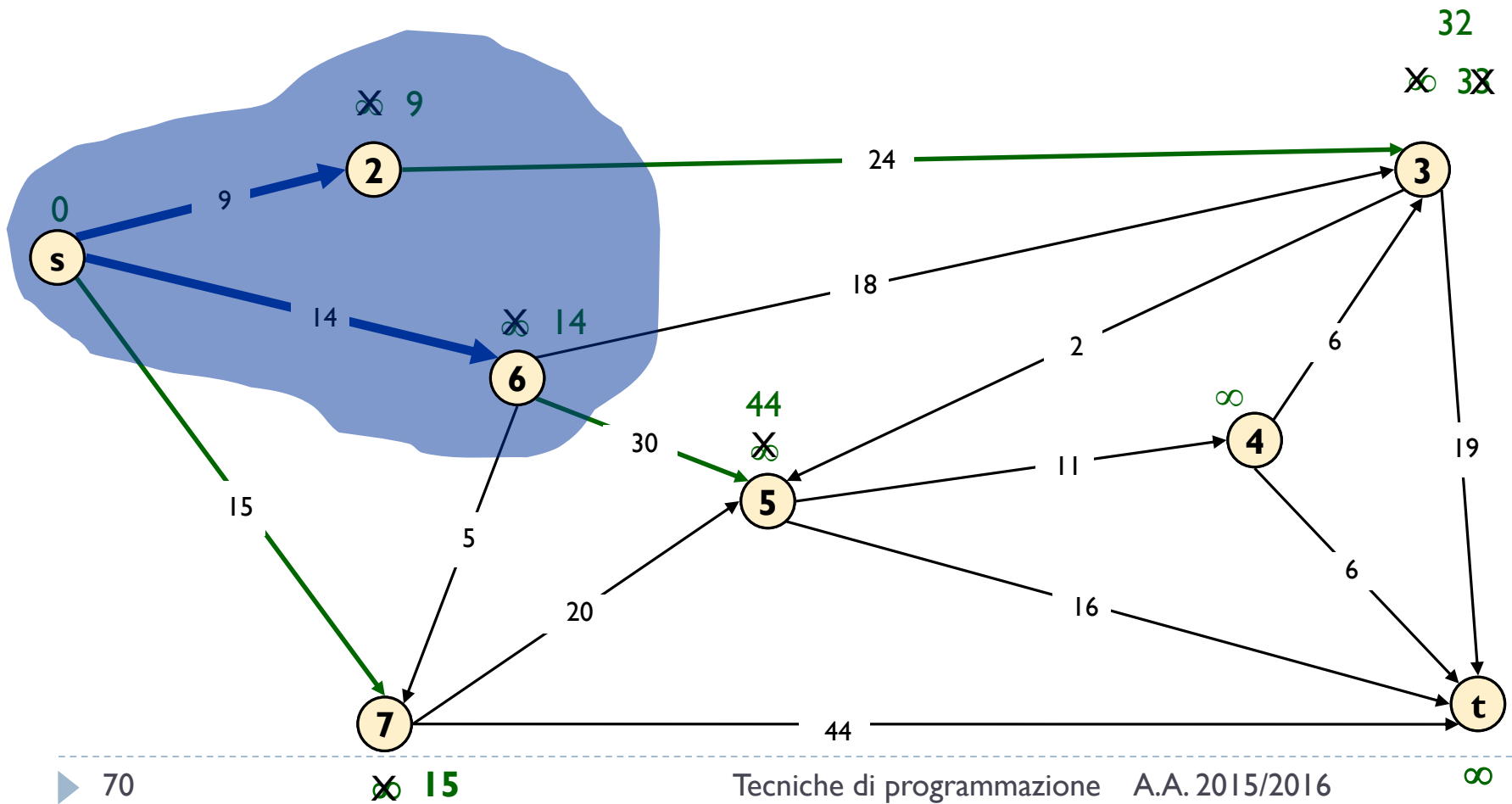
$Q = \{3, 4, 5, 6, 7, t\}$



Dijkstra's Shortest Path Algorithm

$S = \{s, 2, 6\}$

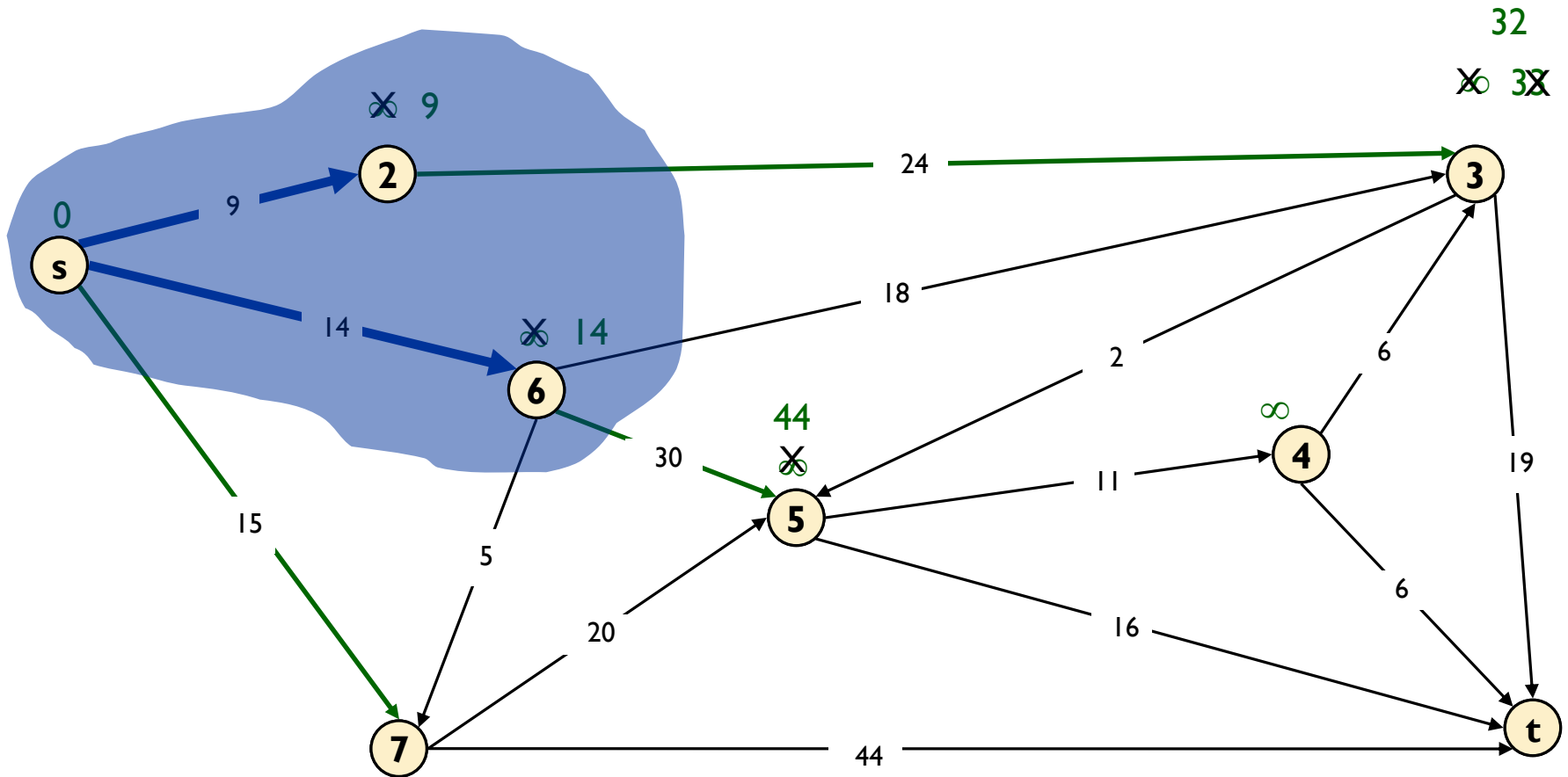
$Q = \{3, 4, 5, 7, t\}$



Dijkstra's Shortest Path Algorithm

$S = \{s, 2, 6\}$

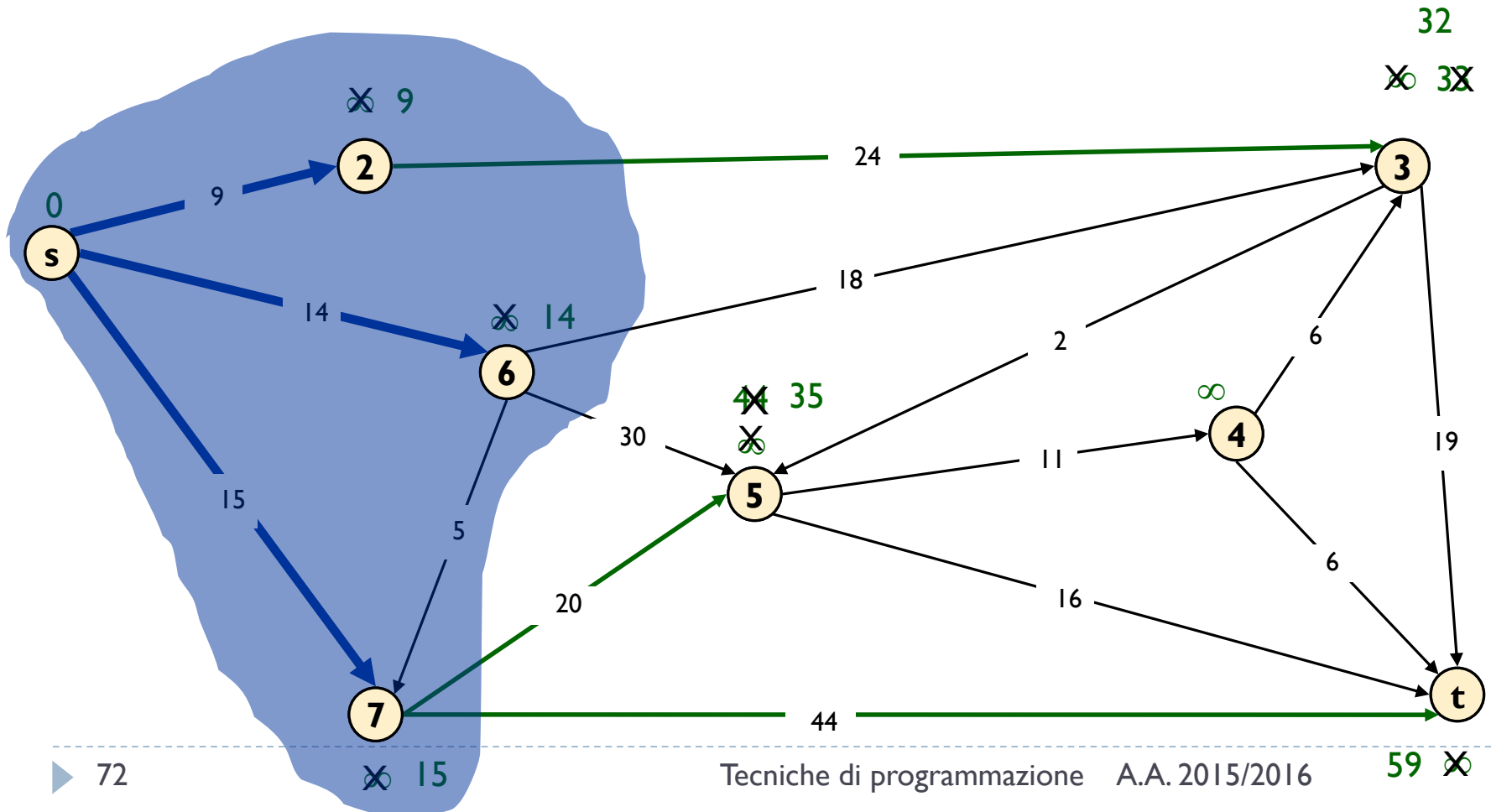
$Q = \{3, 4, 5, 7, t\}$



Dijkstra's Shortest Path Algorithm

$S = \{s, 2, 6, 7\}$

$Q = \{3, 4, 5, t\}$

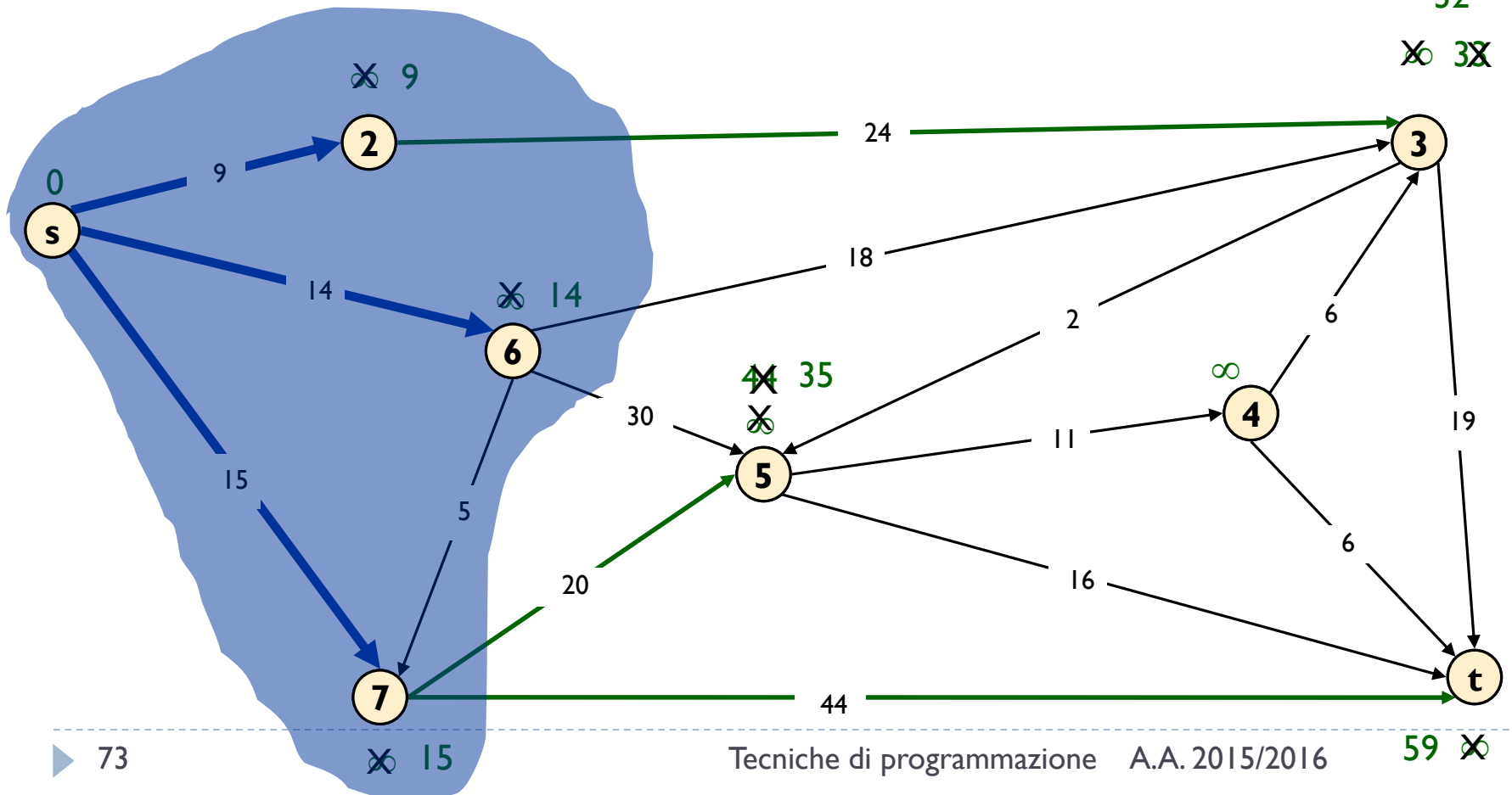


Dijkstra's Shortest Path Algorithm

$S = \{s, 2, 6, 7\}$

$Q = \{3, 4, 5, t\}$

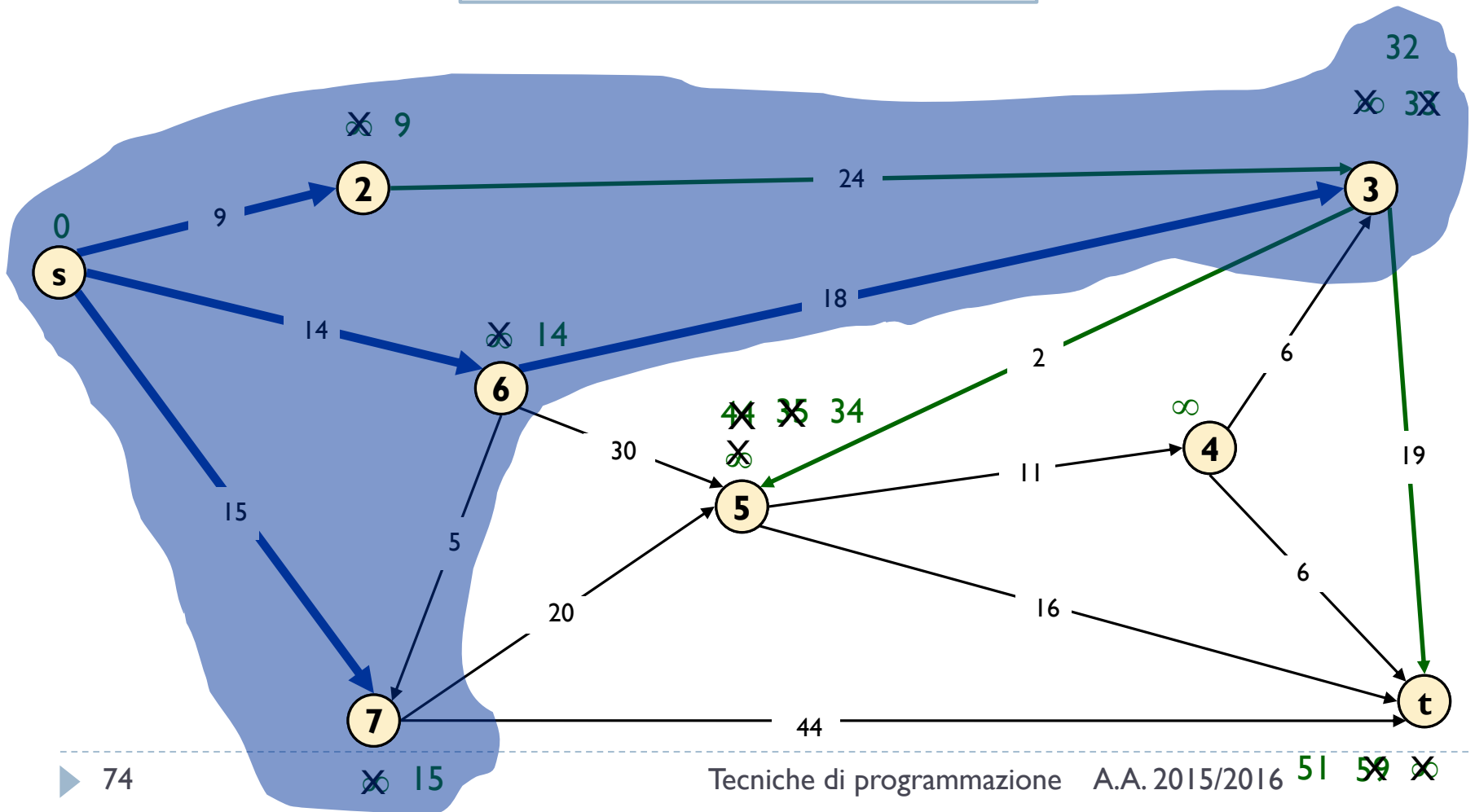
delmin



Dijkstra's Shortest Path Algorithm

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

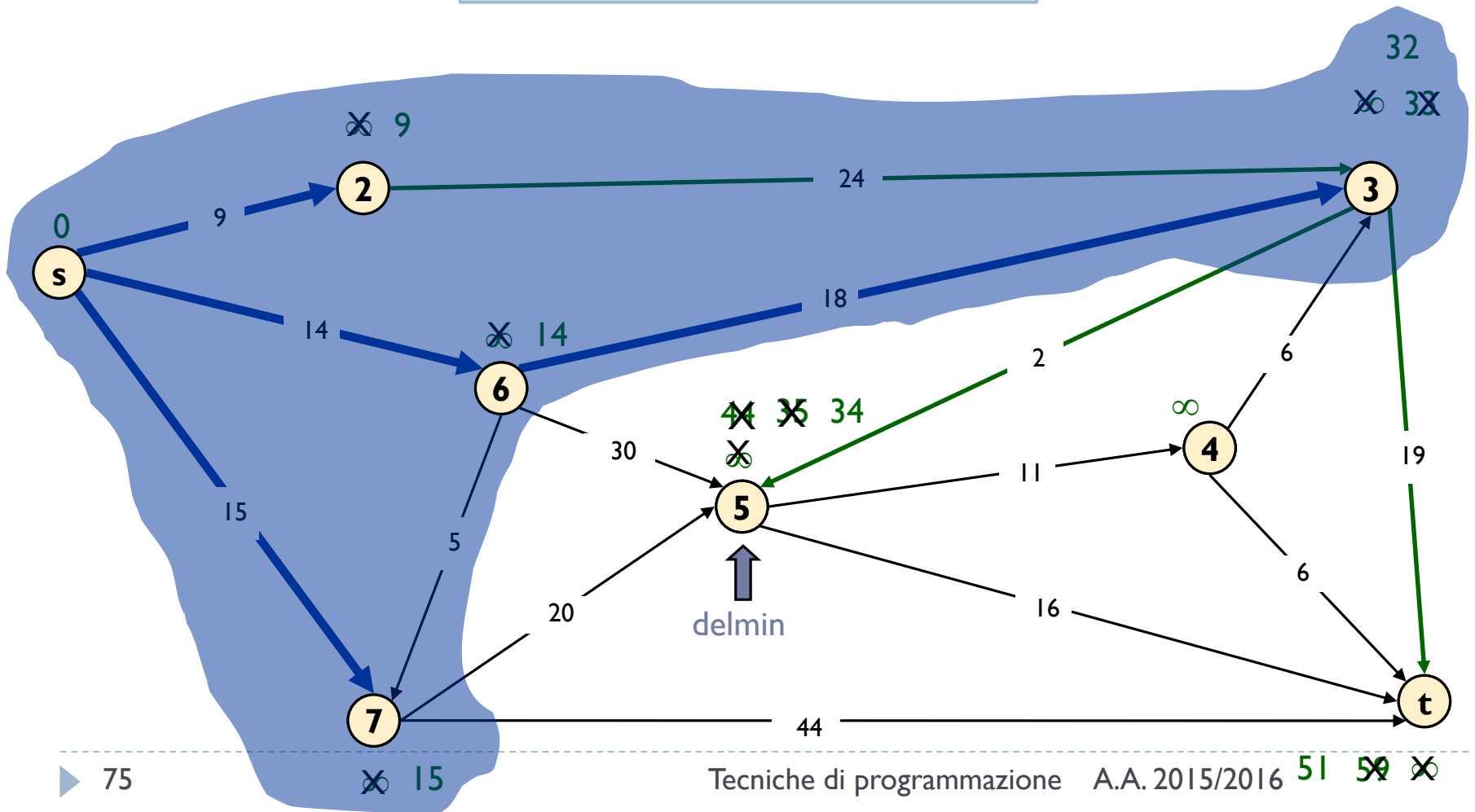
$Q = \{4, 5, t\}$



Dijkstra's Shortest Path Algorithm

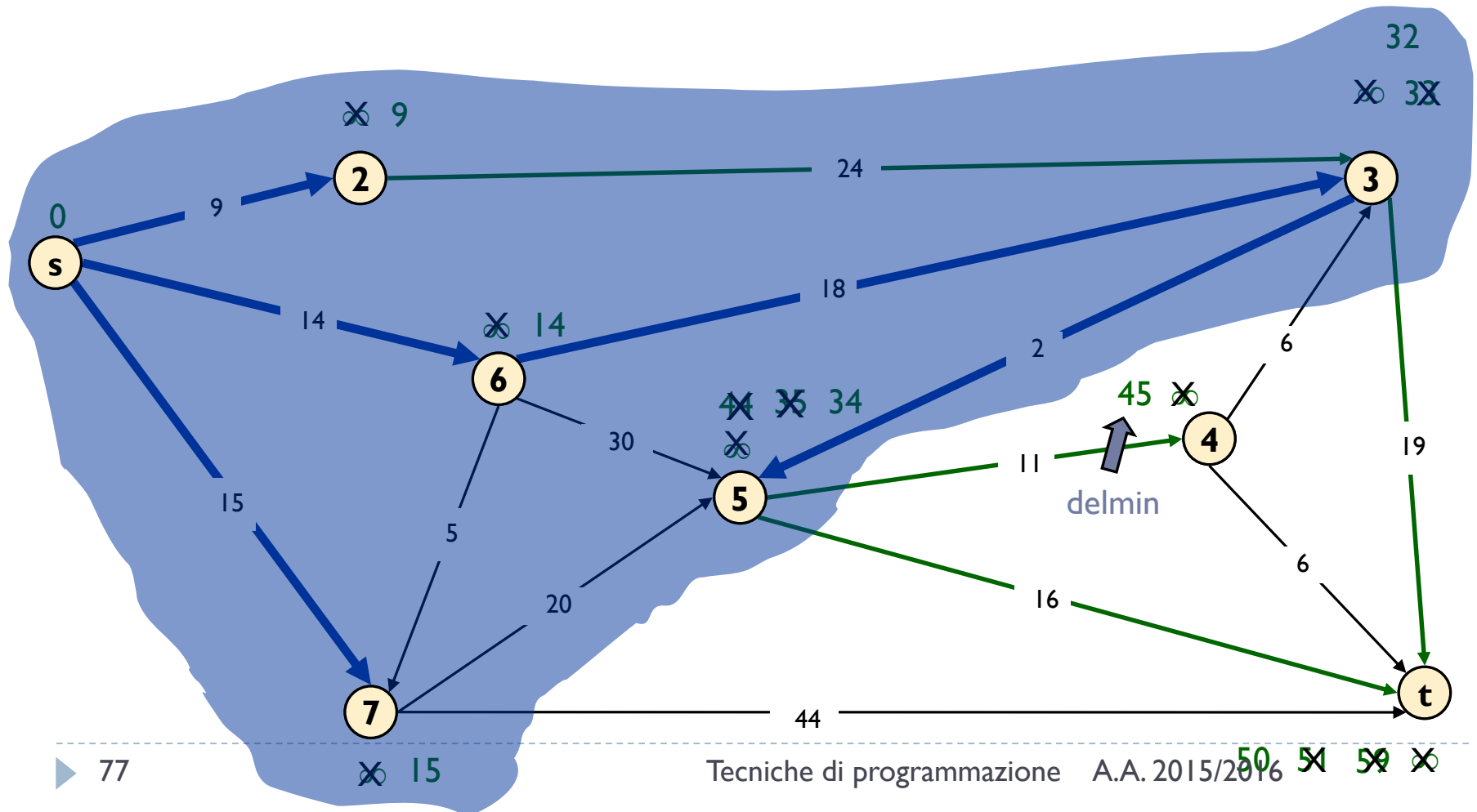
$S = \{s, 2, 3, 6, 7\}$

$Q = \{4, 5, t\}$



Dijkstra's Shortest Path Algorithm

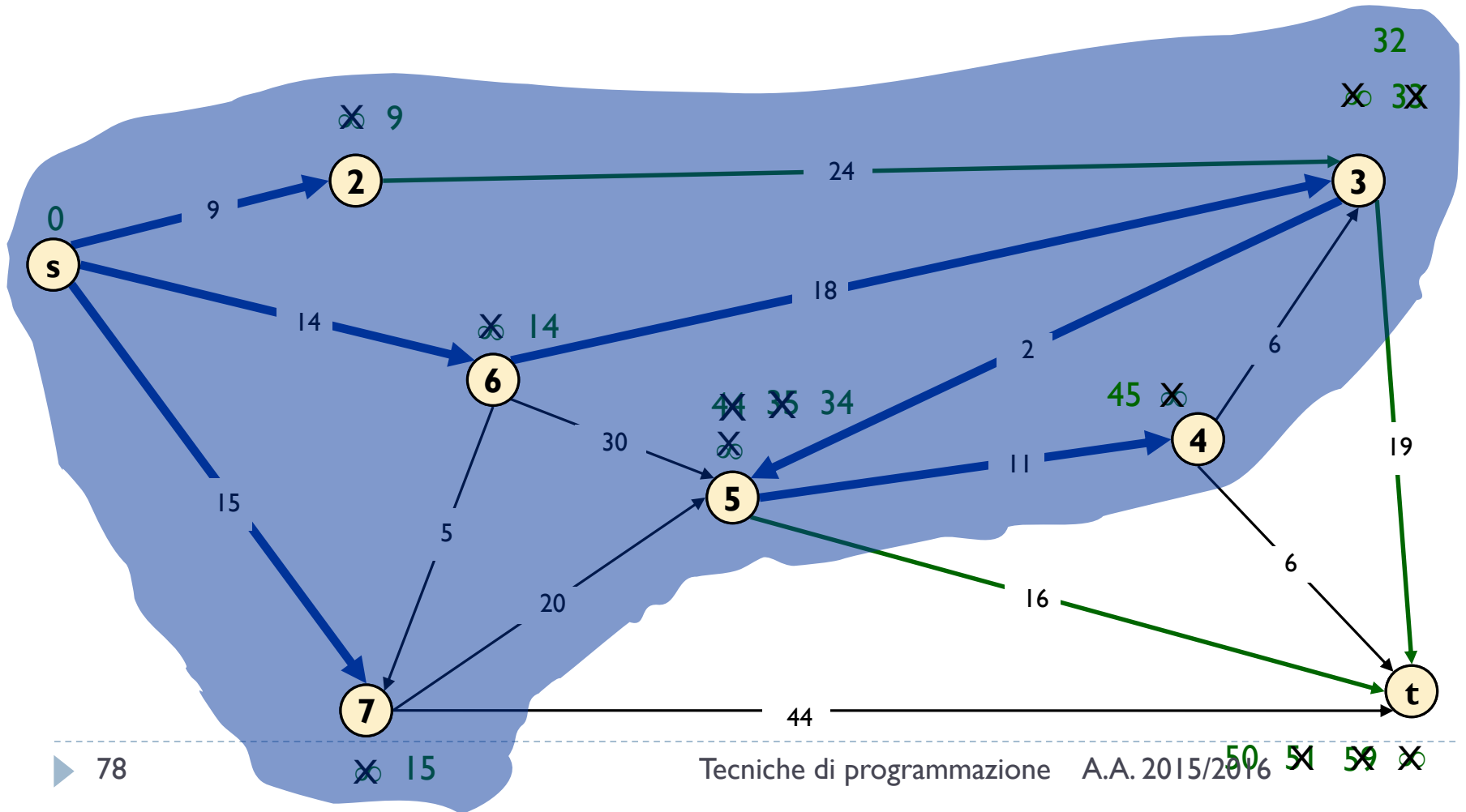
$S = \{s, 2, 3, 5, 6, 7\}$
 $Q = \{4, t\}$



Dijkstra's Shortest Path Algorithm

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

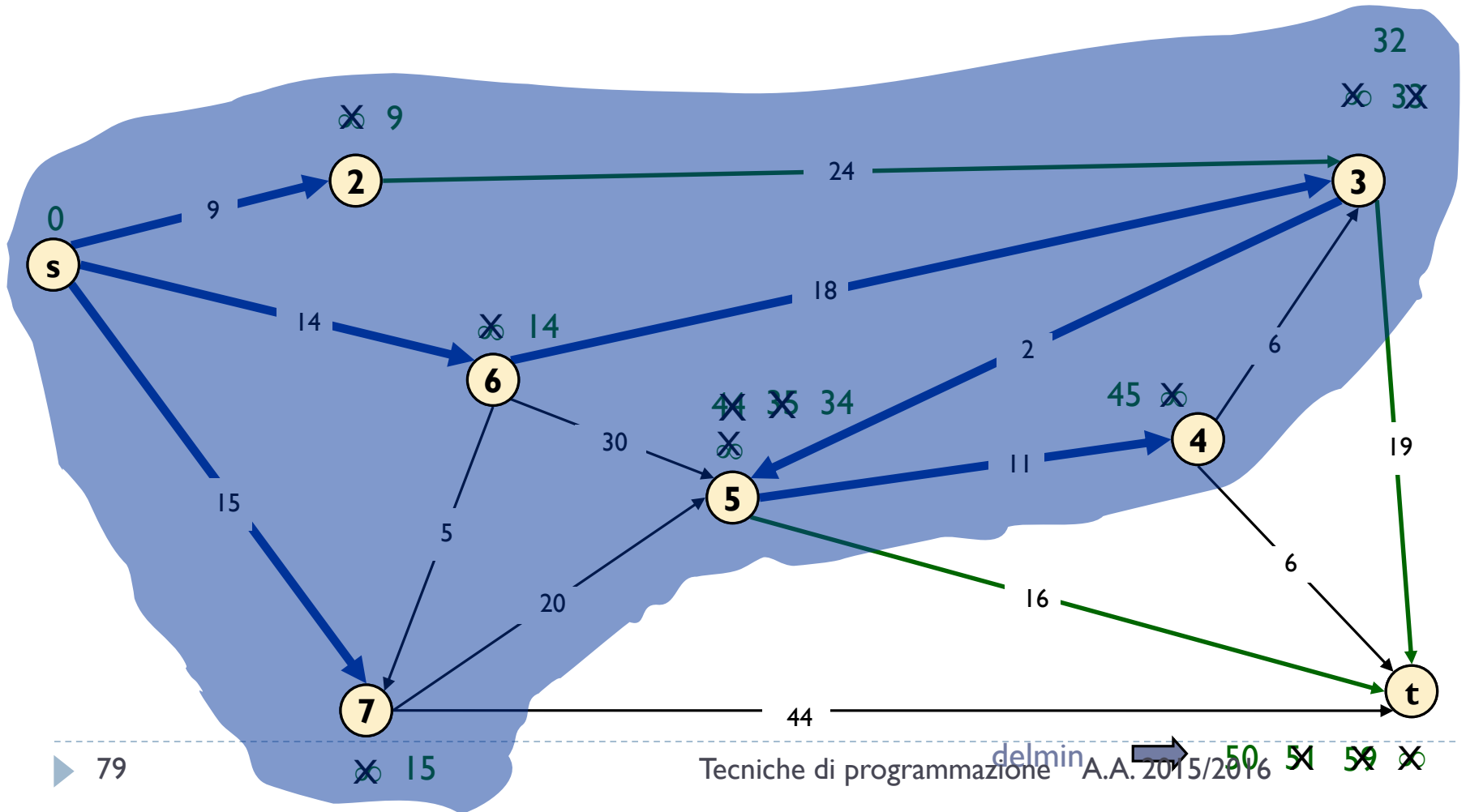
$Q = \{t\}$



Dijkstra's Shortest Path Algorithm

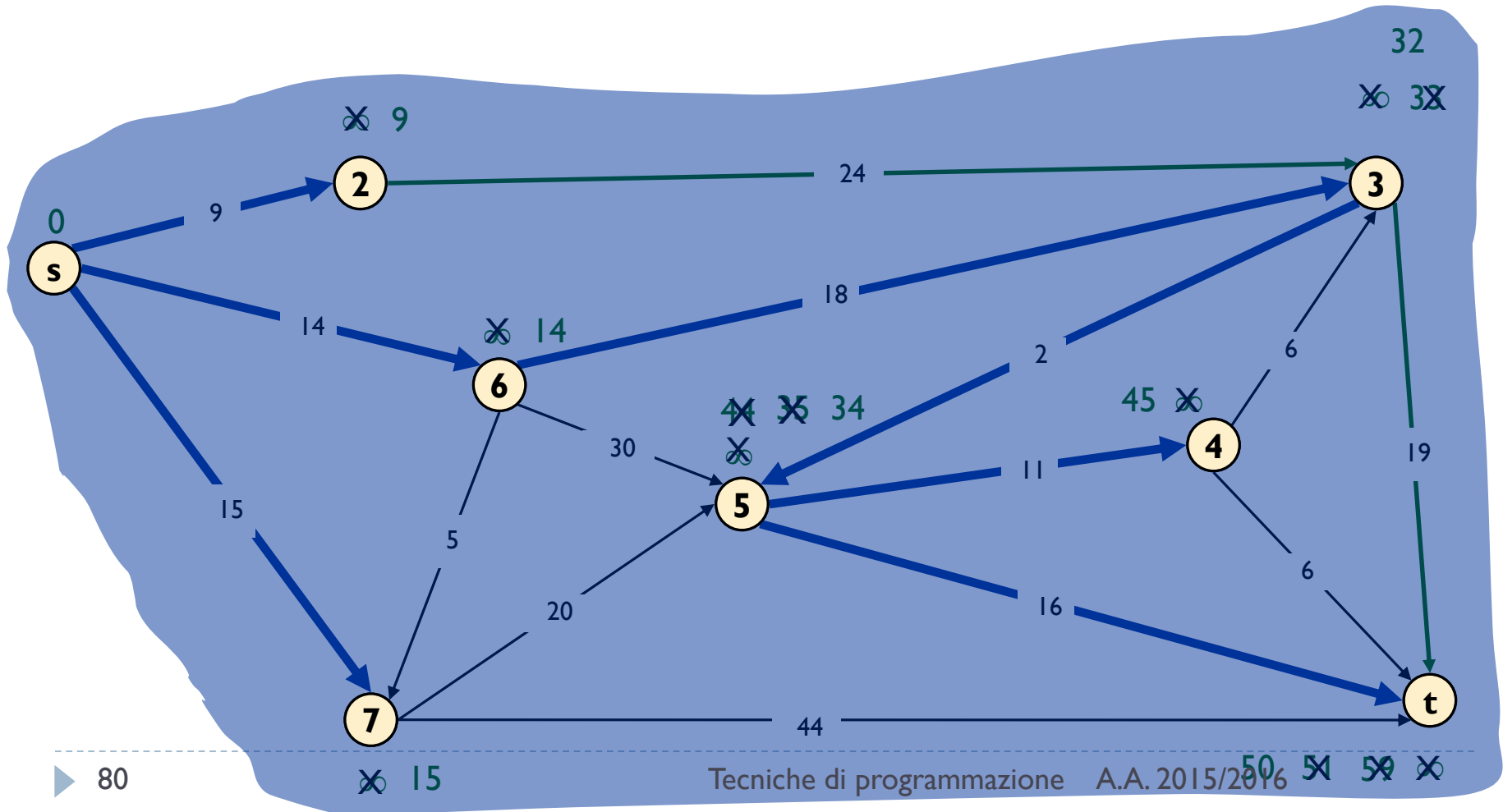
$S = \{s, 2, 3, 4, 5, 6, 7\}$

$Q = \{t\}$



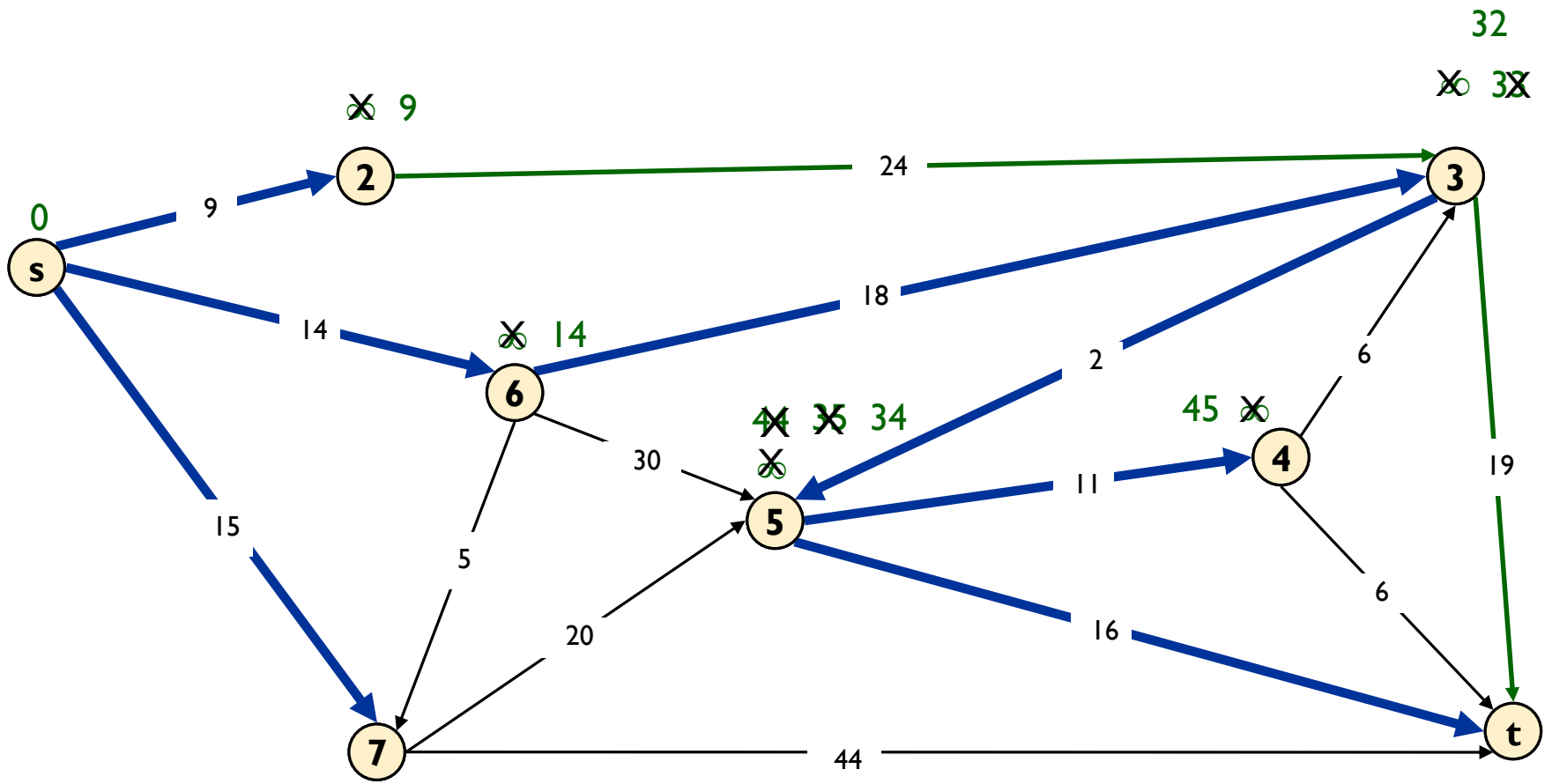
Dijkstra's Shortest Path Algorithm

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



Dijkstra's Shortest Path Algorithm

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



Shortest Paths wrap-up

Algorithm	Problem	Efficiency	Limitation
Floyd-Warshall	AP	$O(V^3)$	No negative cycles
Bellman-Ford	SS	$O(V \cdot E)$	No negative cycles
Repeated Bellman-Ford	AP	$O(V^2 \cdot E)$	No negative cycles
Dijkstra	SS	$O(E + V \cdot \log V)$	No negative edges
Repeated Dijkstra	AP	$O(V \cdot E + V^2 \cdot \log V)$	No negative edges
Breadth-First visit	SS	$O(V + E)$	Unweighted graph



JGraphT



```
public class FloydWarshallShortestPaths<V,E>
public class BellmanFordShortestPath<V,E>
public class DijkstraShortestPath<V,E>
```

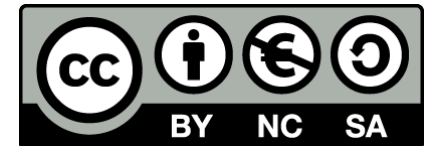
```
// APSP
List<GraphPath<V,E>>  getShortestPaths (V v)
GraphPath<V,E>      getShortestPath (V a, V b)





// SSSP
GraphPath<V,E>      getPath ()
```

Resources

- ▶ Algorithms in a Nutshell, G. Heineman, G. Pollice, S. Selkow, O'Reilly, ISBN 978-0-596-51624-6, Chapter 6
<http://shop.oreilly.com/product/9780596516246.do>
- ▶ http://en.wikipedia.org/wiki/Floyd%E2%80%93Warshall_algorithm

Licenza d'uso



- ▶ Queste diapositive sono distribuite con licenza Creative Commons “Attribuzione - Non commerciale - Condividi allo stesso modo (CC BY-NC-SA)”
- ▶ Sei libero:
 - ▶ di riprodurre, distribuire, comunicare al pubblico, esporre in pubblico, rappresentare, eseguire e recitare quest'opera 
 - ▶ di modificare quest'opera 
- ▶ Alle seguenti condizioni:
 - ▶ Attribuzione — Devi attribuire la paternità dell'opera agli autori originali e in modo tale da non suggerire che essi avallino te o il modo in cui tu usi l'opera. 
 - ▶ Non commerciale — Non puoi usare quest'opera per fini commerciali. 
 - ▶ Condividi allo stesso modo — Se alteri o trasformi quest'opera, o se la usi per crearne un'altra, puoi distribuire l'opera risultante solo con una licenza identica o equivalente a questa.
- ▶ <http://creativecommons.org/licenses/by-nc-sa/3.0/>

