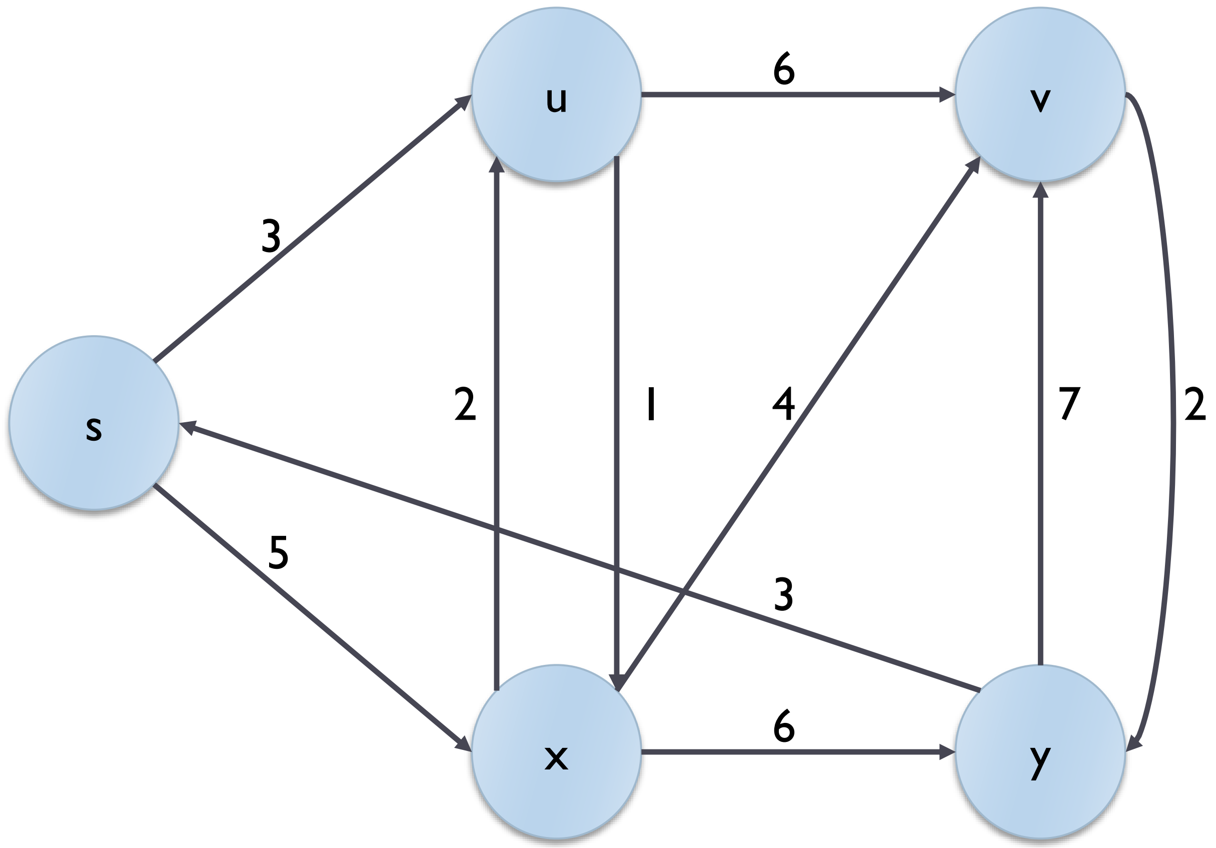
Graphs: Finding shortest paths

Tecniche di Programmazione – A.A. 2018/2019

Example

What is the shortest path between s and v ?



Summary

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

Definitions

Graphs: Finding shortest paths

Definition: weight of a path

- Consider a directed, weighted graph G=(V,E), with weight function w: E→***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 mininum-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

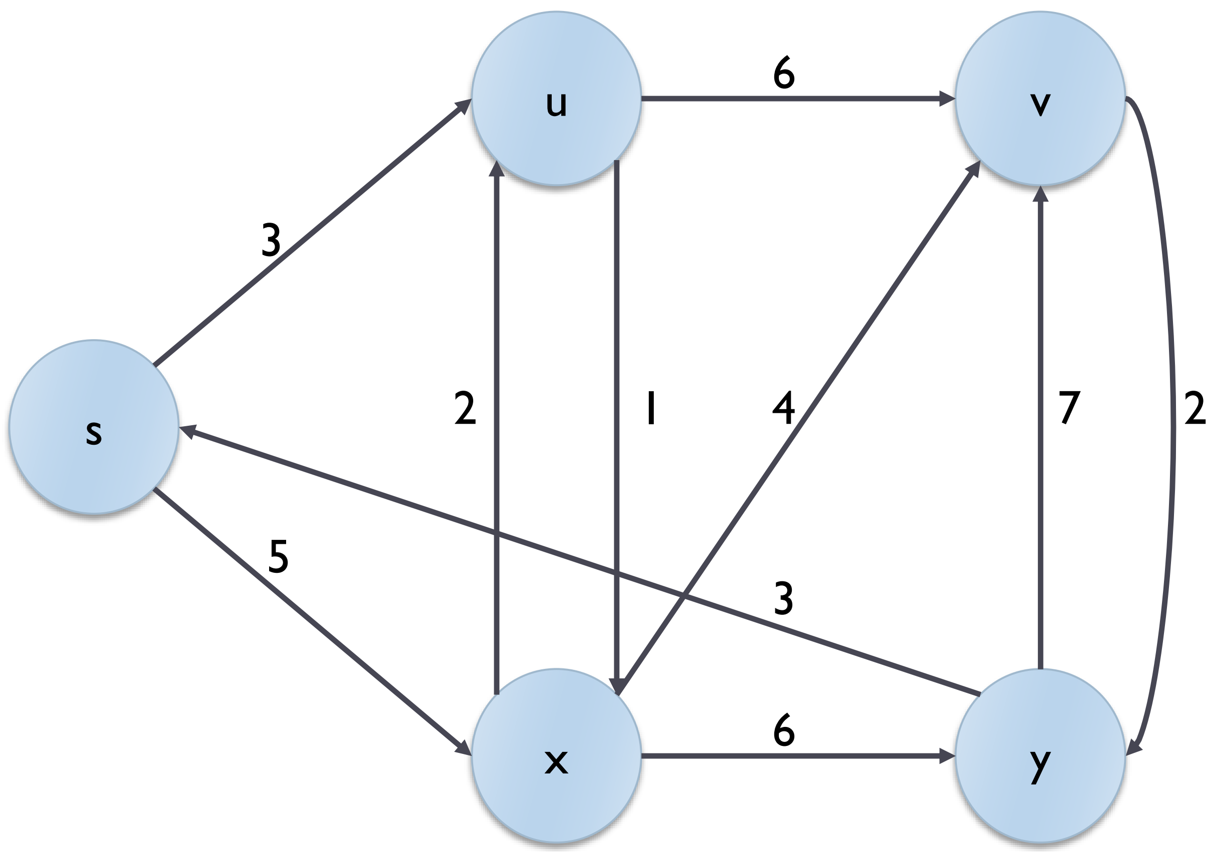
- Single-source shortest path (SS-SP)
  - Given u and v, find the shortest path between u and v
  - Given u, find the shortest path between u and any other vertex
- All-pairs shortest path (AP-SP)
  - Given a graph, find the shortest path between any pair of vertices

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

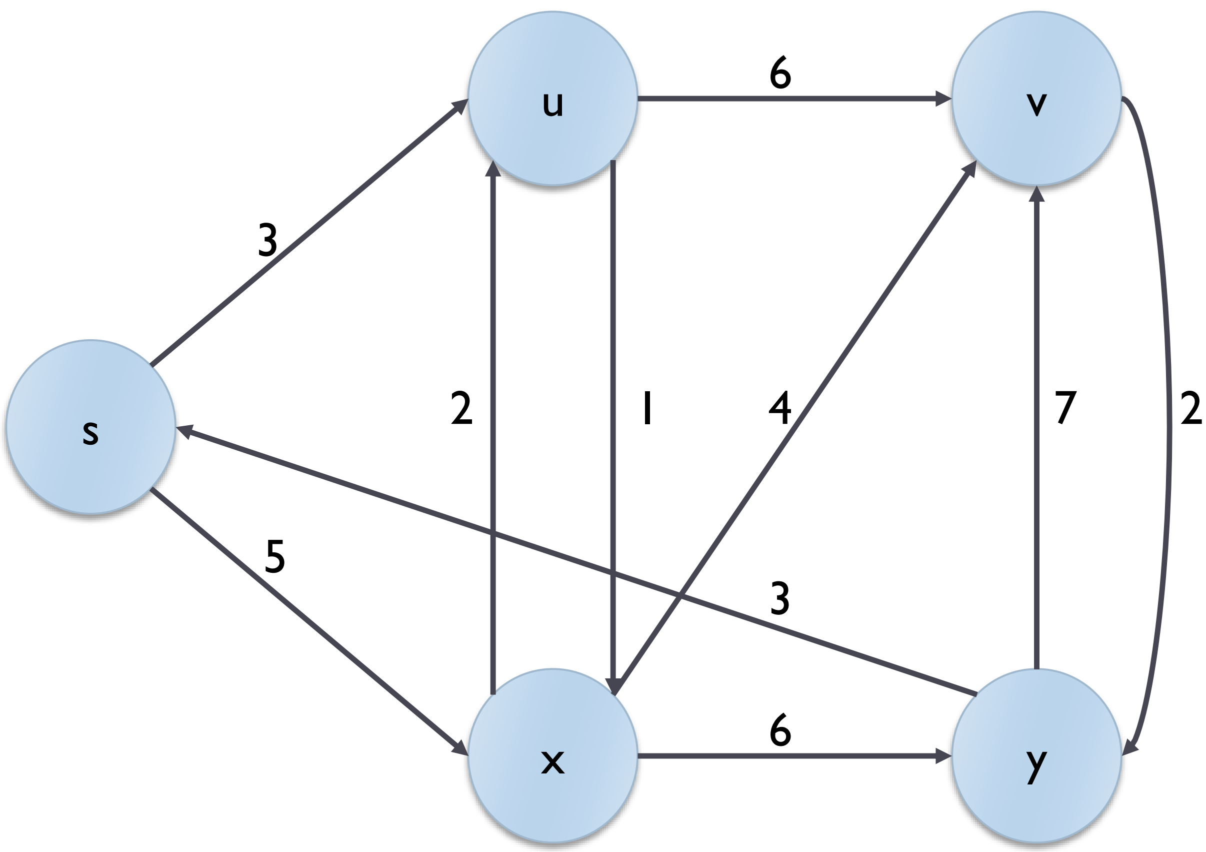
What is the shortest path between s and v ?



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

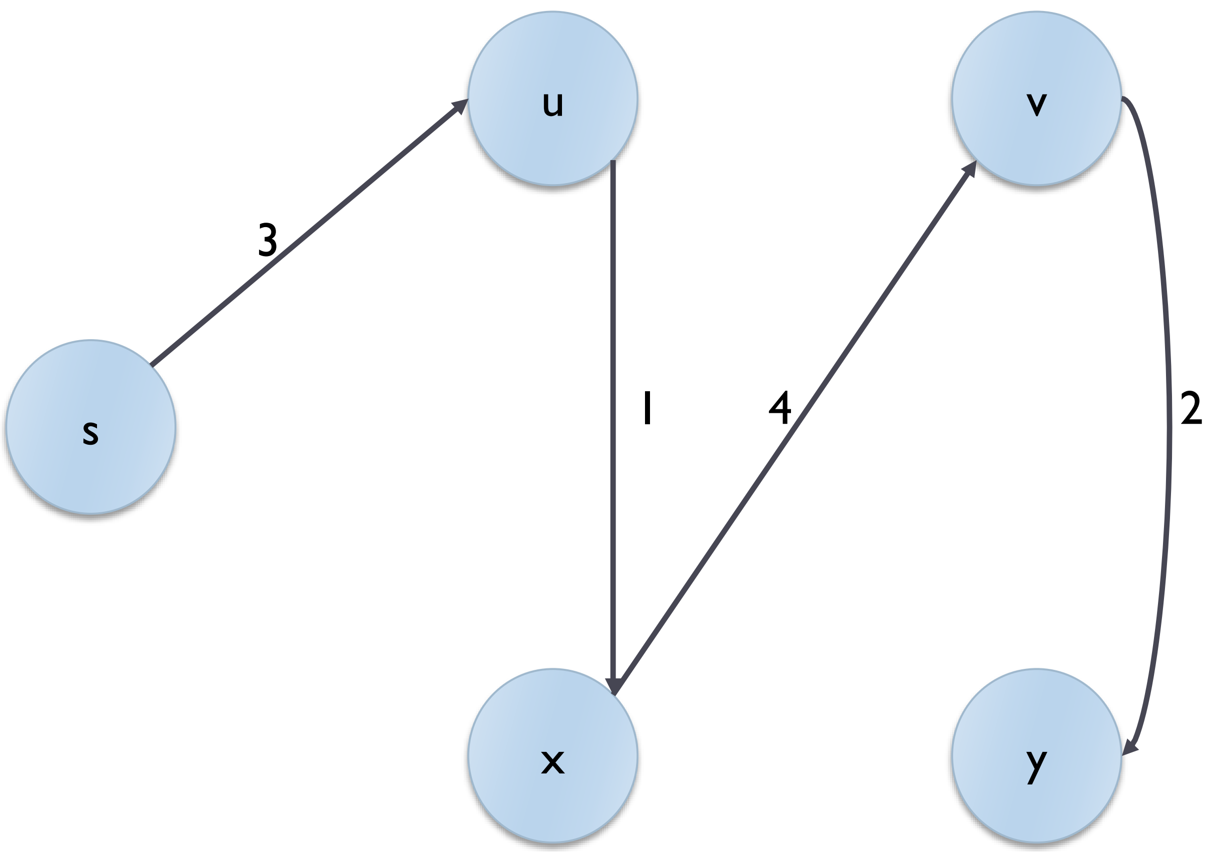
Shortest path graph

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

Shortest path tree

- $G\pi$ is a tree (due to the Lemma) rooted in u
- In $G\pi$, the (unique) paths starting from u are always shortest paths
- $G\pi$ is not unique, but all possible $G\pi$ 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

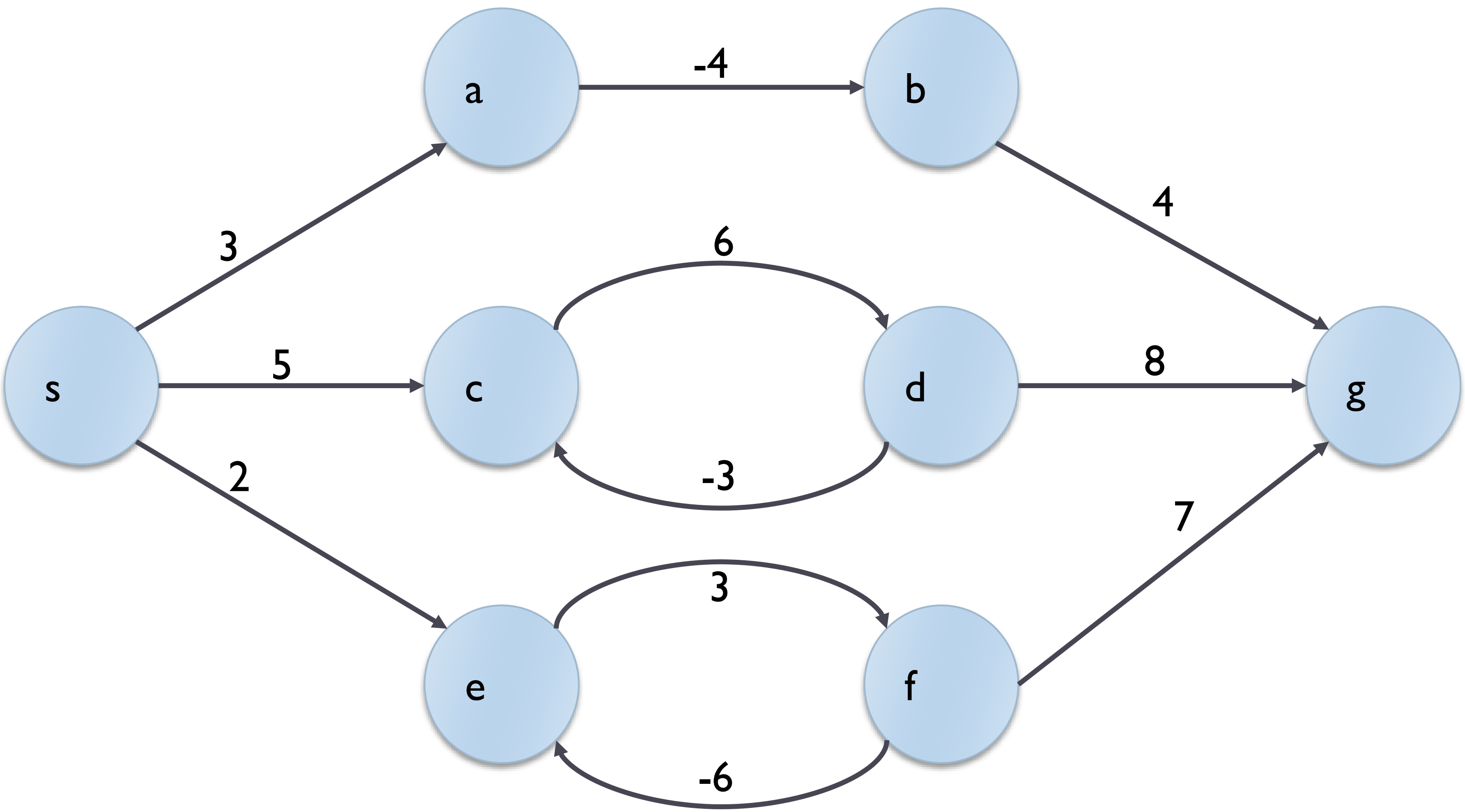
Special case

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

Negative-weight cycles

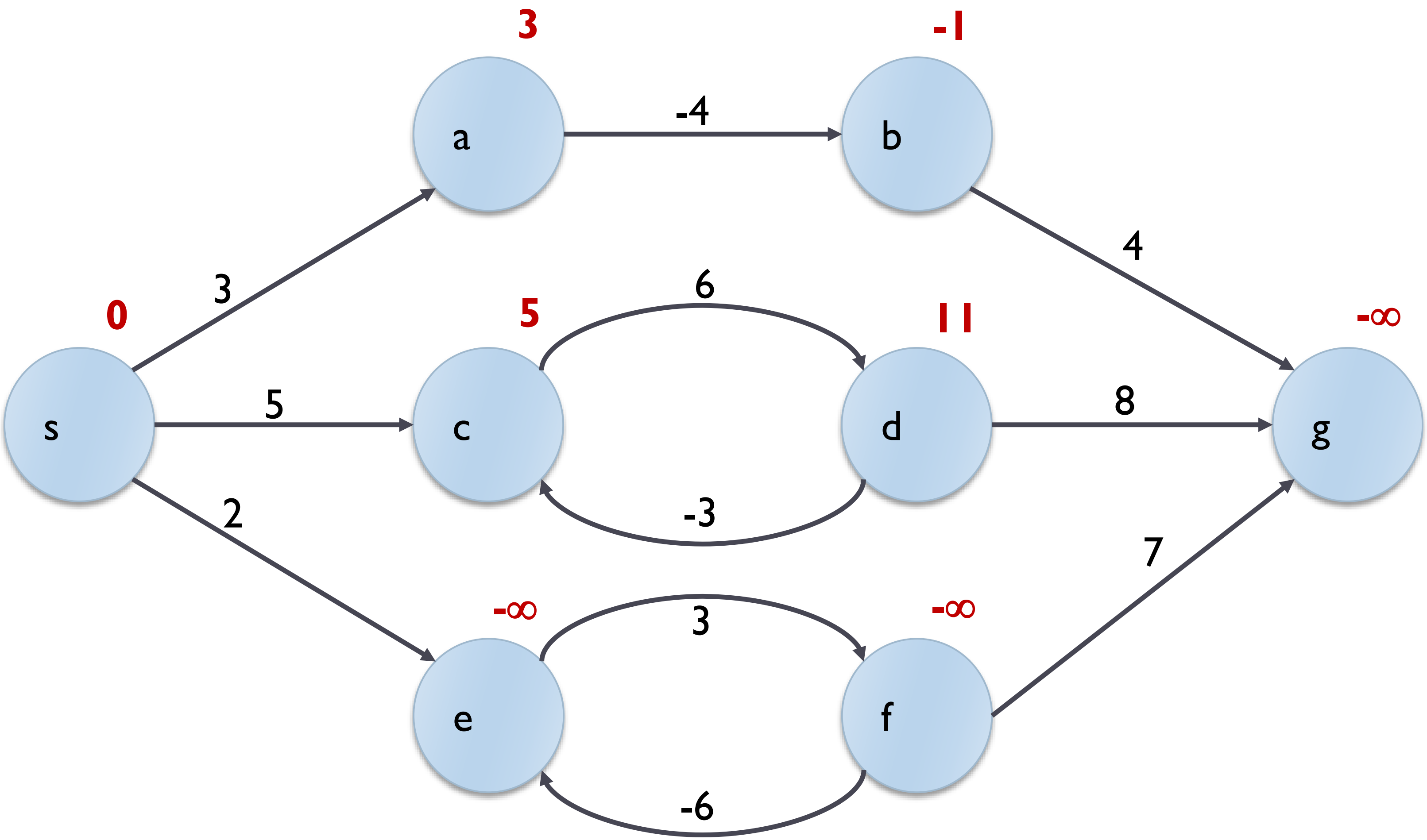
- Minimum paths cannot be defined if there are negative-weight cycles in the graph
- In this case, the minimum path does not exist, because you may always decrease the path weight by going once more through the loop.
- Conventionally, in these case we say that the path weight is $-\infty$.

Example



Example

Minimum-weight paths from source vertex s



Lemma

- Consider an ordered weighted graph G=(V,E), with weight function w: $E \rightarrow \boldsymbol{R}$.
- Let p=<v_1, v_2, …, v_k> 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}=<v_i, v_{i+1}, …, v_j> 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 arbitrarily chose 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 π[v]) whenever we discover that node v is more conveniently reached by traversing edge (u,v)

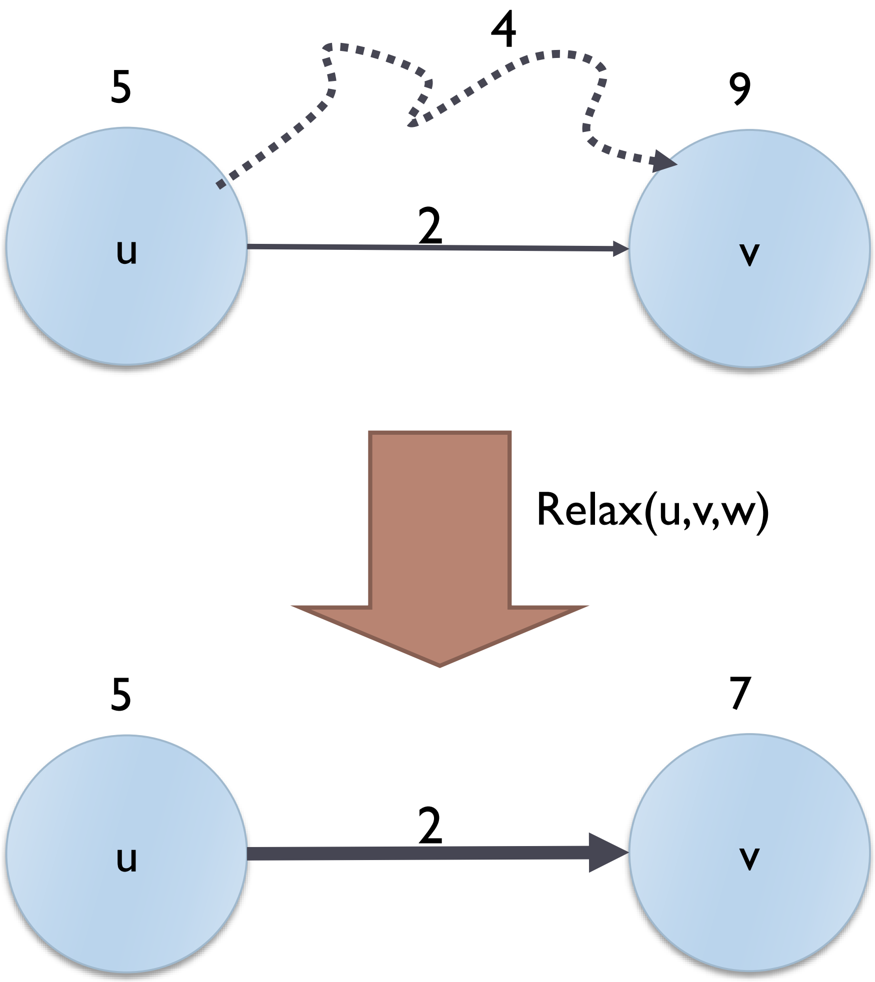
Initial state

- Initialize-Single-Source(G(V,E), s)
  1. **for** all vertices v $\in$ V
  2. **do**
     1. d[v]$\leftarrow\infty$
     2. π[v]$\leftarrow$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] ← d[u]+w(u,v)
     2. π[v] ← u

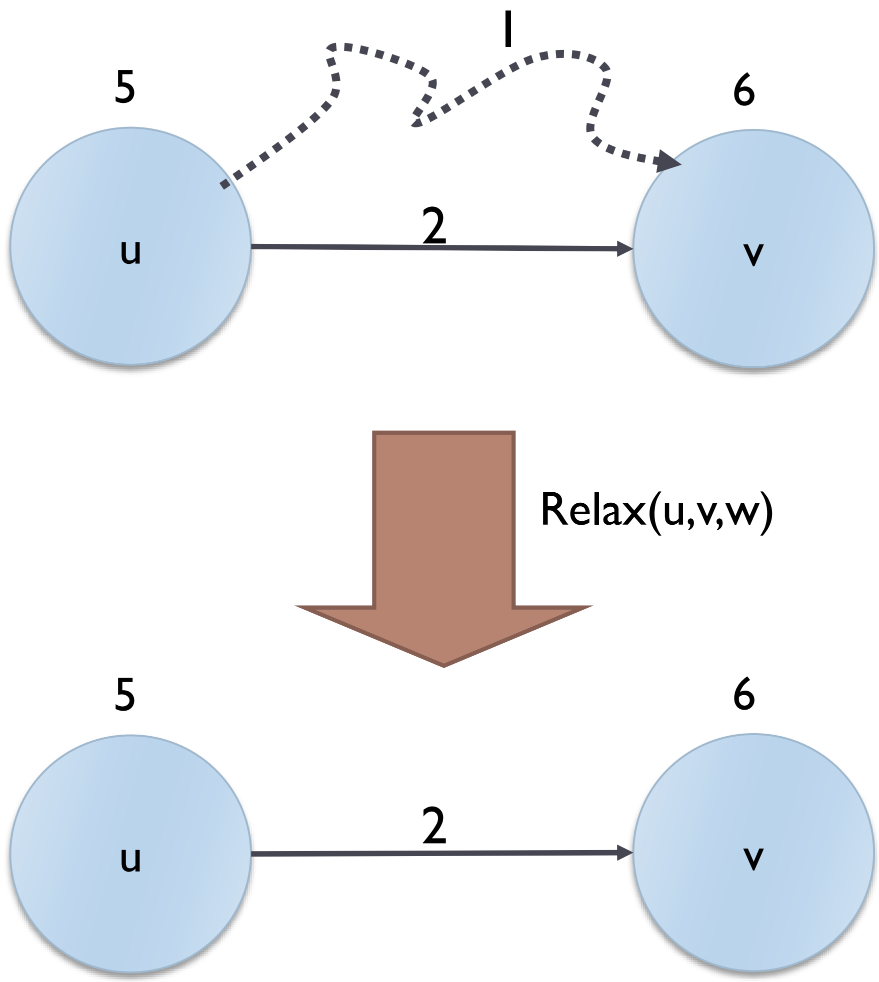
Example 1



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→***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→***R*** and source vertex s∈V. Assume that G has no negative-weight cycles reachable from s.

- Therefore
  - After calling Initialize-Single-Source(G,s), the predecessor subgraph Gπ 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)

Implementations

Package `org.jgrapht.alg.shortestpath`

Class Summary

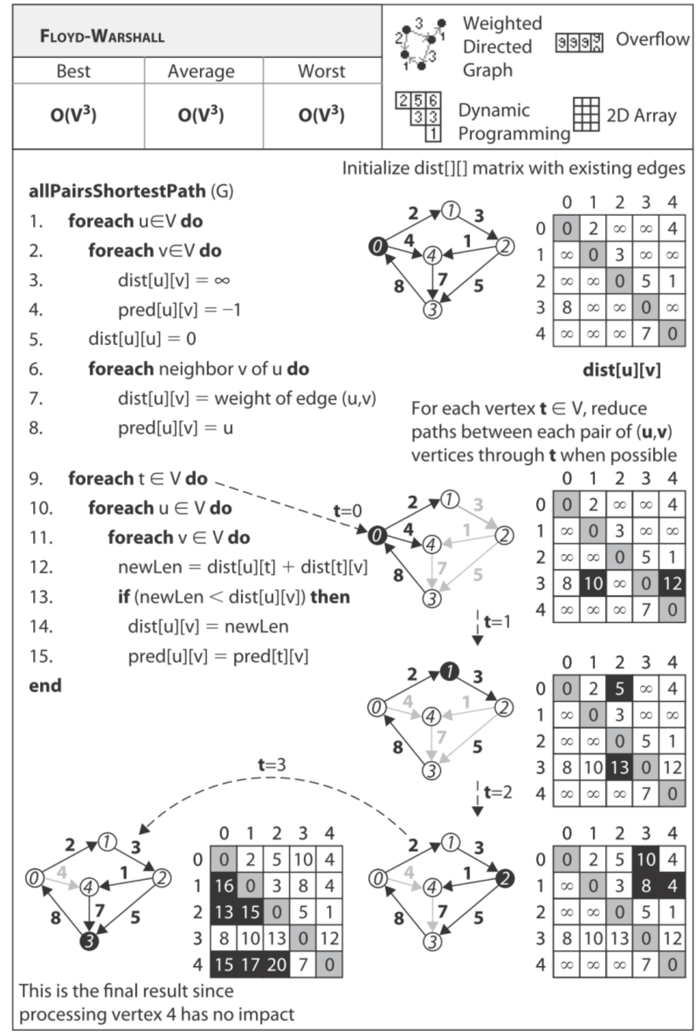
Class	Description
AllDirectedPaths<V,E>	A Dijkstra-like algorithm to find all paths between two sets of nodes in a directed graph, with options to search only simple paths and to limit the path length.
ALTAdmissibleHeuristic<V,E>	An admissible heuristic for the A* algorithm using a set of landmarks and the triangle inequality.
AStarShortestPath<V,E>	A* shortest path.
BellmanFordShortestPath<V,E>	The Bellman-Ford algorithm.
BhandariKDisjointShortestPaths<V,E>	An implementation of Bhandari algorithm for finding K edge-*disjoint* shortest paths.
BidirectionalDijkstraShortestPath<V,E>	A bidirectional version of Dijkstra's algorithm.
DijkstraShortestPath<V,E>	An implementation of **Dijkstra's shortest path algorithm** using a Fibonacci heap.
FloydWarshallShortestPaths<V,E>	The Floyd-Warshall algorithm.
GraphMeasurer<V,E>	Algorithm class which computes a number of distance related metrics.
JohnsonShortestPaths<V,E>	Johnson's all pairs shortest paths algorithm.
KShortestSimplePaths<V,E>	The algorithm determines the k shortest simple paths in increasing order of weight.
ListMultiObjectiveSingleSourcePathsImpl<V,E>	An implementation of `MultiObjectiveShortestPathAlgorithm.MultiObjectiveSingleSourcePaths` which stores one list of paths per vertex.
ListSingleSourcePathsImpl<V,E>	An implementation of `ShortestPathAlgorithm.SingleSourcePaths` which stores one path per vertex.
MartinShortestPath<V,E>	Martin's algorithm for the multi-objective shortest paths problem.
SuurballeKDisjointShortestPaths<V,E>	An implementation of Suurballe algorithm for finding K edge-*disjoint* shortest paths.
TreeMeasurer<V,E>	Algorithm class which computes a number of distance related metrics for trees.
TreeSingleSourcePathsImpl<V,E>	An implementation of `ShortestPathAlgorithm.SingleSourcePaths` which uses linear space.

Floyd-Warshall algorithm

Graphs: Finding shortest paths

Floyd-Warshall algorithm

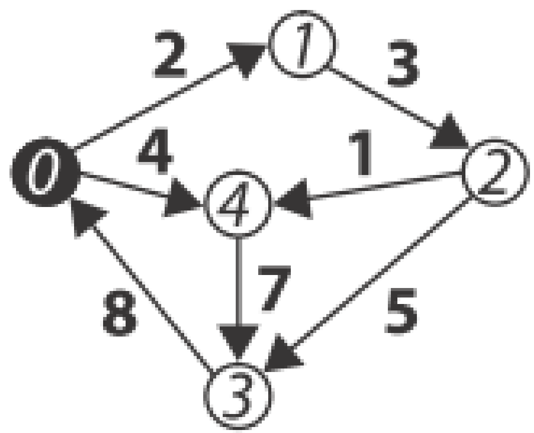
- Computes the all-source shortest path (AP-SP)
- dist[i][j] is an n-by-n matrix that contains the length of a shortest path from vi to vj.
- if dist[u][v] is ∞, there is no path from u to v
- pred[s][j] is used to reconstruct an actual shortest path: stores the predecessor vertex for reaching vj starting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



Floyd-Warshall: initialization

```
allPairsShortestPath (G)
    foreach u∈V do
        foreach v∈V do
            dist[u][v] = ∞
            pred[u][v] = −1
        dist[u][u] = 0
        foreach neighbor v of u do
            dist[u][v] = weight of edge (u,v)
            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

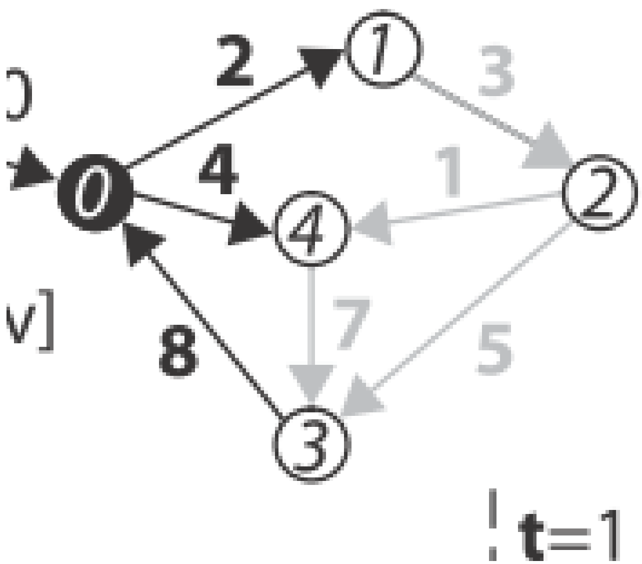
dist[u][v]

Floyd-Warshall: relaxation

```
foreach t ∈ V do
    foreach u ∈ V do
        foreach v ∈ V do
            newLen = dist[u][t] + dist[t][v]
            if (newLen < dist[u][v]) then
                dist[u][v] = newLen
                pred[u][v] = pred[t][v]
```

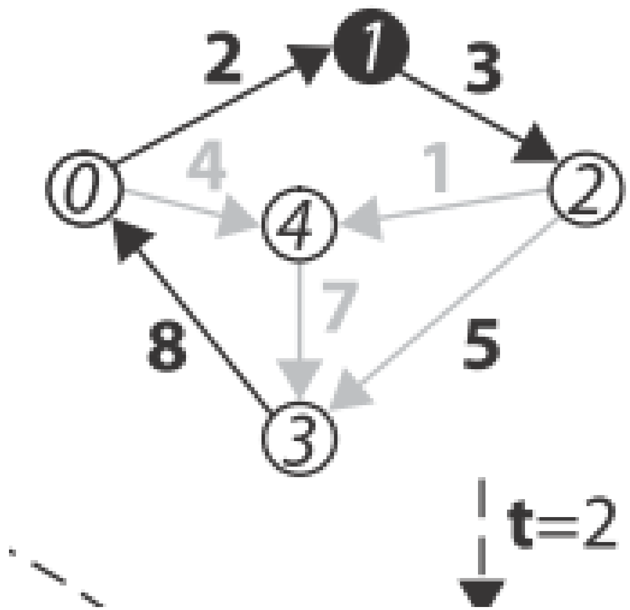
t=0

Example, after step t=0



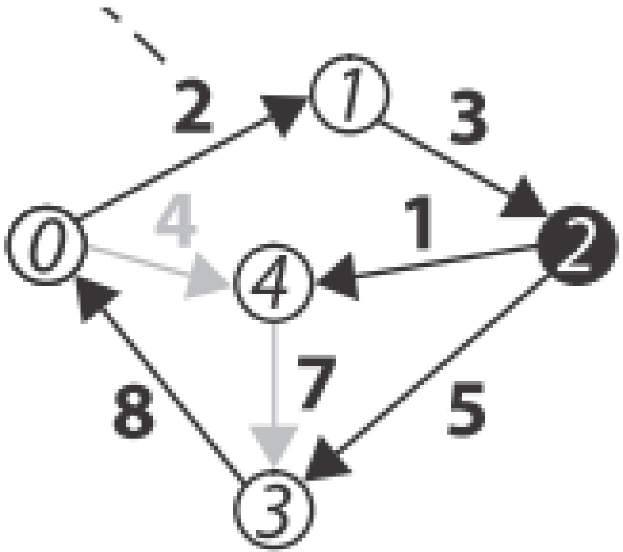
	0	1	2	3	4
0	0	2	∞	∞	4
1	∞	0	3	∞	∞
2	∞	∞	0	5	1
3	8	10	∞	0	12
4	∞	∞	∞	7	0

Example, after step t=1



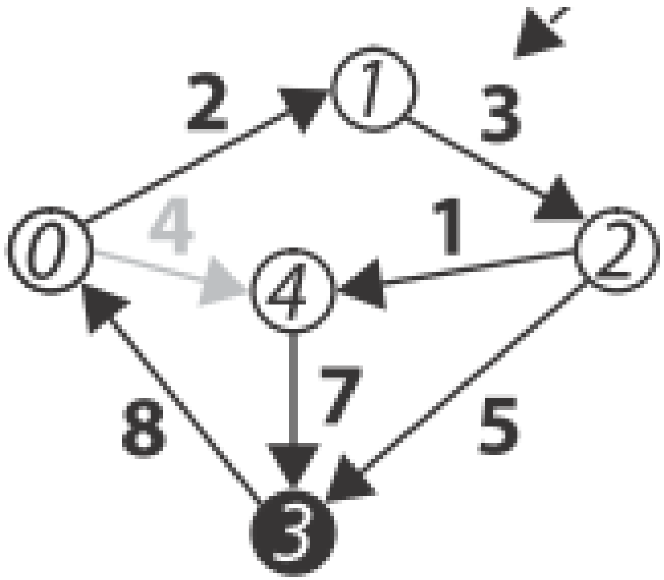
	0	1	2	3	4
0	0	2	5	∞	4
1	∞	0	3	∞	∞
2	∞	∞	0	5	1
3	8	10	13	0	12
4	∞	∞	∞	7	0

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

OVERVIEW PACKAGE **CLASS** USE TREE DEPRECATED INDEX HELP

PREV CLASS **NEXT CLASS** FRAMES NO FRAMES

SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

org.jgrapht.alg.shortestpath

Class FloydWarshallShortestPaths<V,E>

java.lang.Object
  org.jgrapht.alg.shortestpath.FloydWarshallShortestPaths<V,E>

Type Parameters:

`V - the graph vertex type`

`E - the graph edge type`

All Implemented Interfaces:

`ShortestPathAlgorithm<V,E>`

```
public class FloydWarshallShortestPaths<V,E>
extends Object
```

The Floyd-Warshall algorithm.

The Floyd-Warshall algorithm finds all shortest paths (all n^2 of them) in $O(n^3)$ time. Note that during construction time, no computations are performed! All computations are performed the first time one of the member methods of this class is invoked. The results are stored, so all subsequent calls to the same method are computationally efficient.

Author:

`Tom Larkworthy, Soren Davidsen (soren@tanesha.net), Joris Kinable, Dimitrios Michail`

Nested Class Summary

Nested classes/interfaces inherited from interface org.jgrapht.alg.interfaces.ShortestPathAlgorithm
`ShortestPathAlgorithm.SingleSourcePaths<V,E>`

Bellman-Ford-Moore Algorithm

Graphs: Finding shortest paths

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!
```

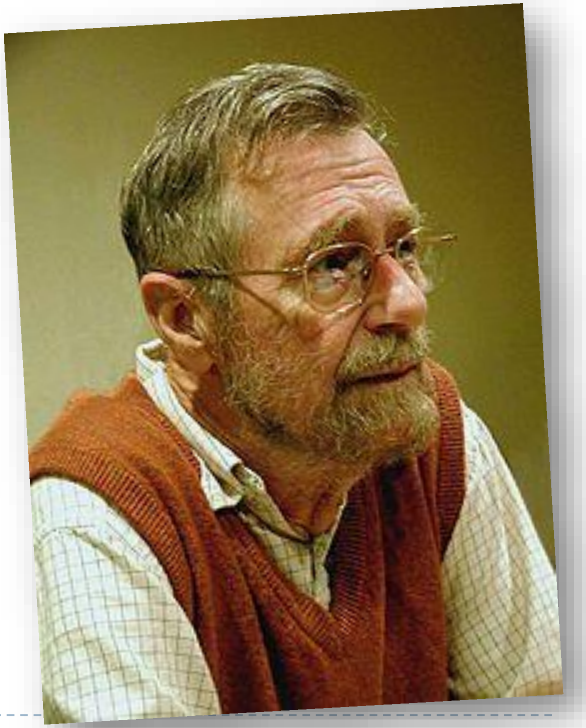
Dijkstra’s Algorithm

Graphs: Finding shortest paths

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!

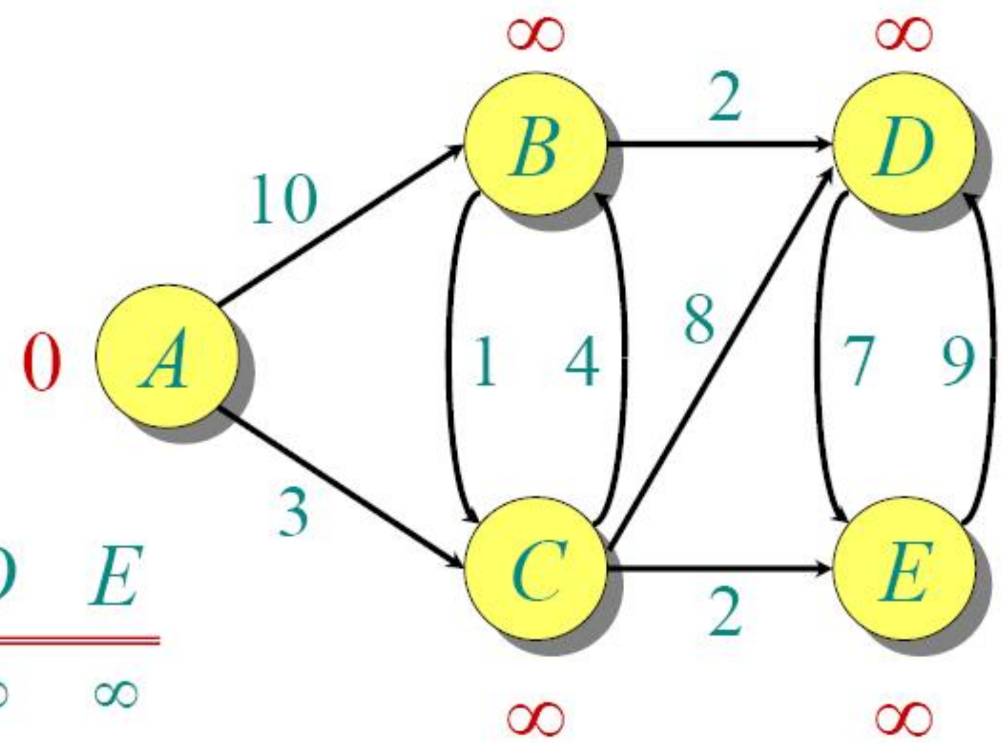


Dijkstra's algorithm

```
dist[s] ←0                          (distance to source vertex is zero)
for  all v ∈ 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 ∈ Q with the min. distance)
    S←S∪{u}                         (add u to list of visited vertices)
     for all v ∈ 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

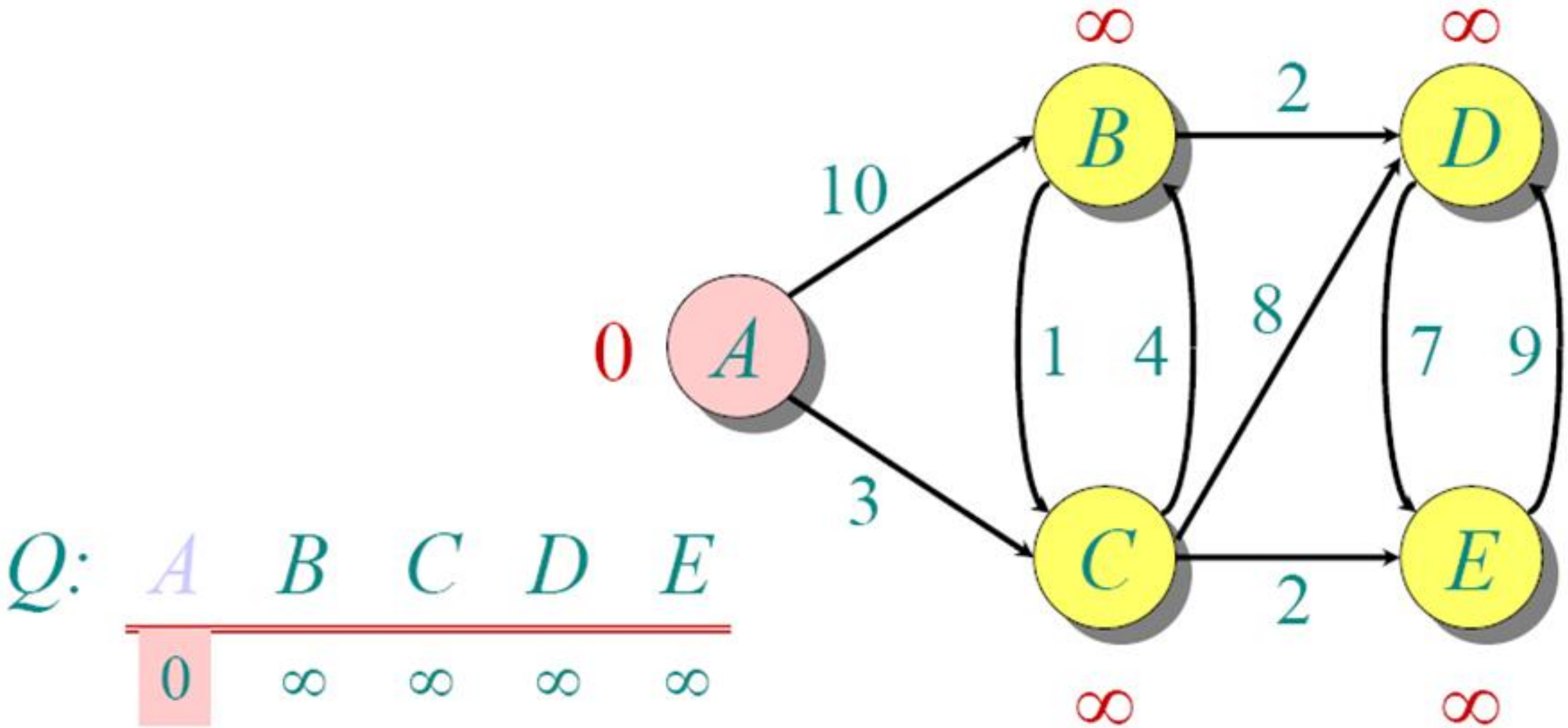
Initialize:



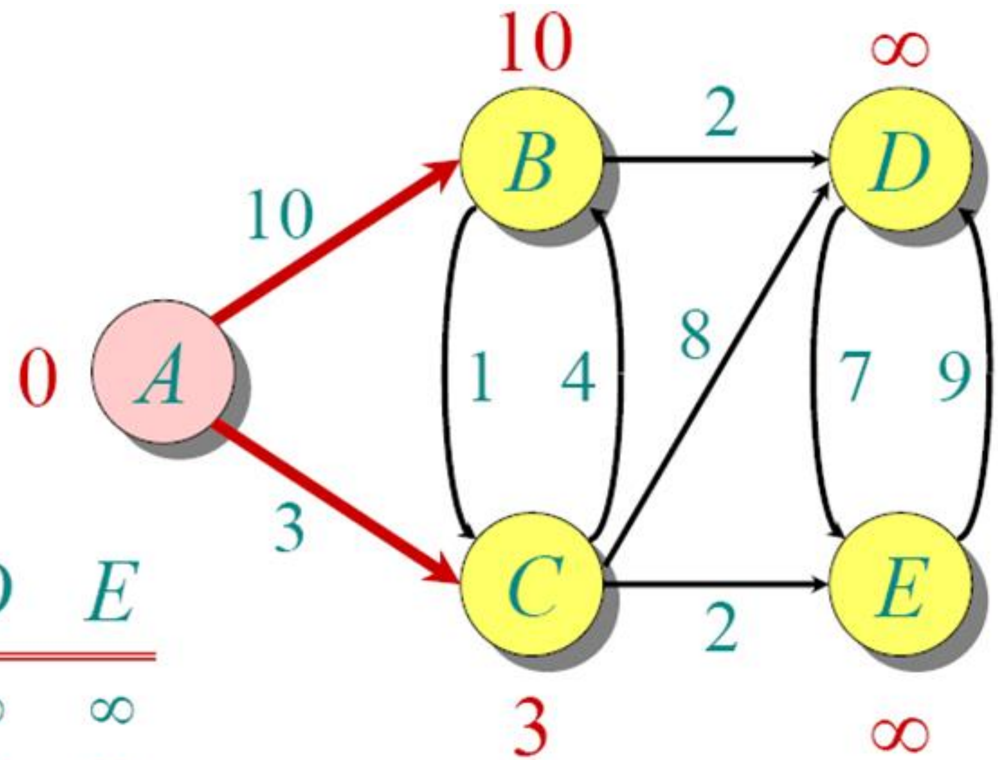
Q:	A	B	C	D	E
	0	∞	∞	∞	∞

S: {}

Dijkstra Animated Example



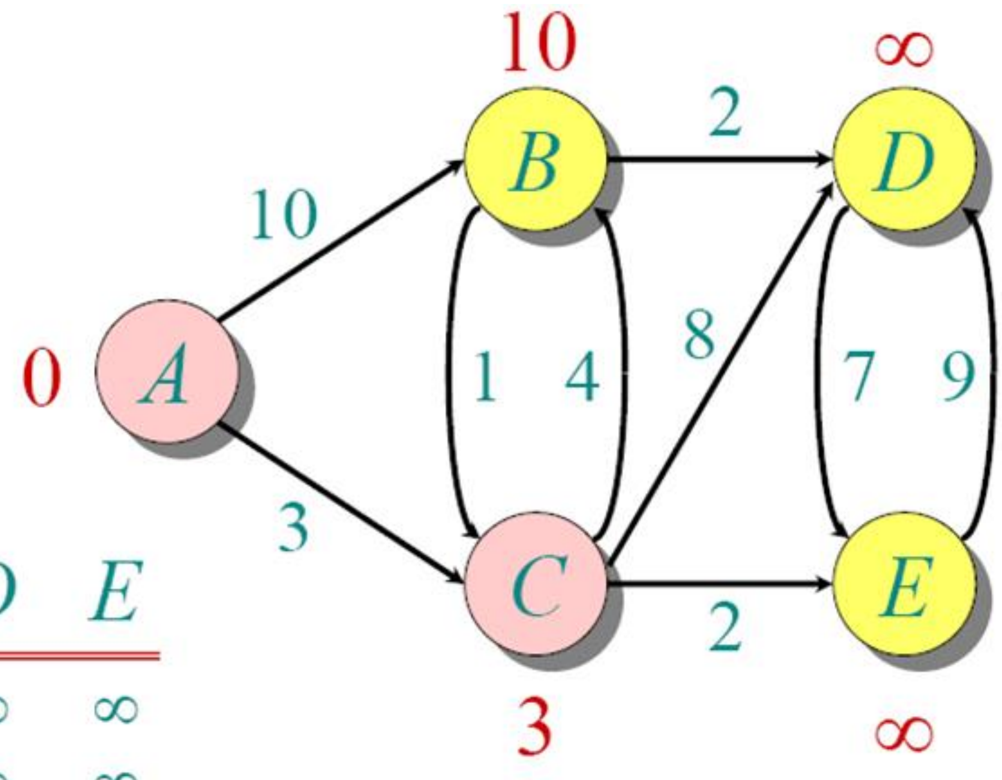
Dijkstra Animated Example



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞

S: { A }

Dijkstra Animated Example

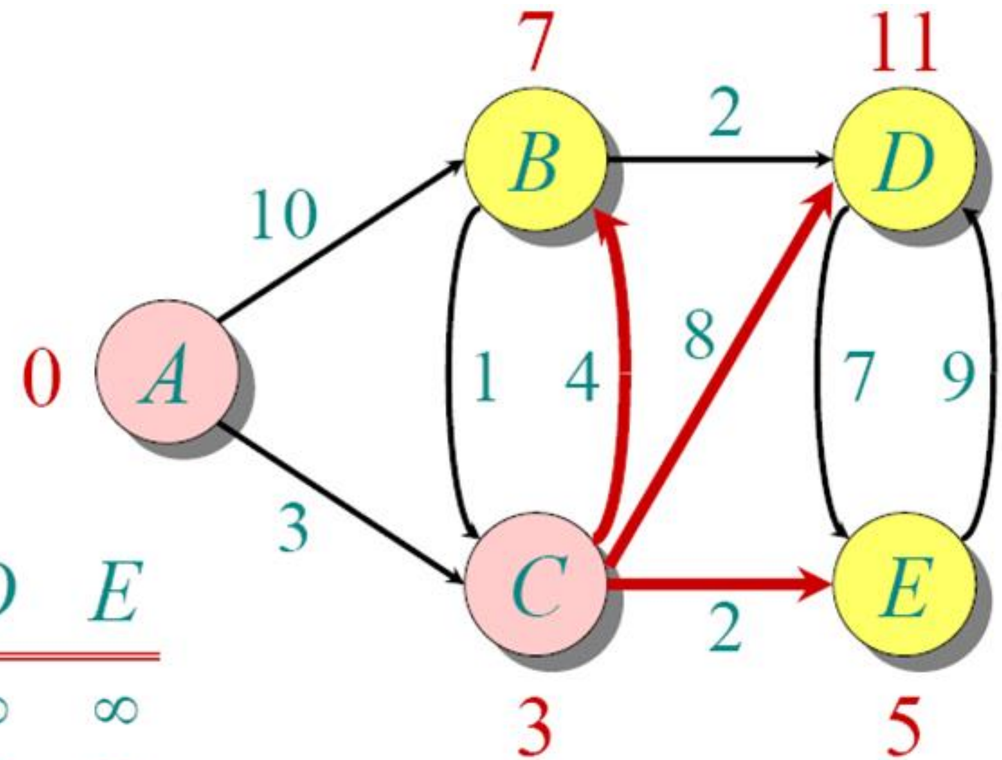


Q:

A	B	C	D	E
0	∞	∞	∞	∞
	10	3	∞	∞

S: { A, C }

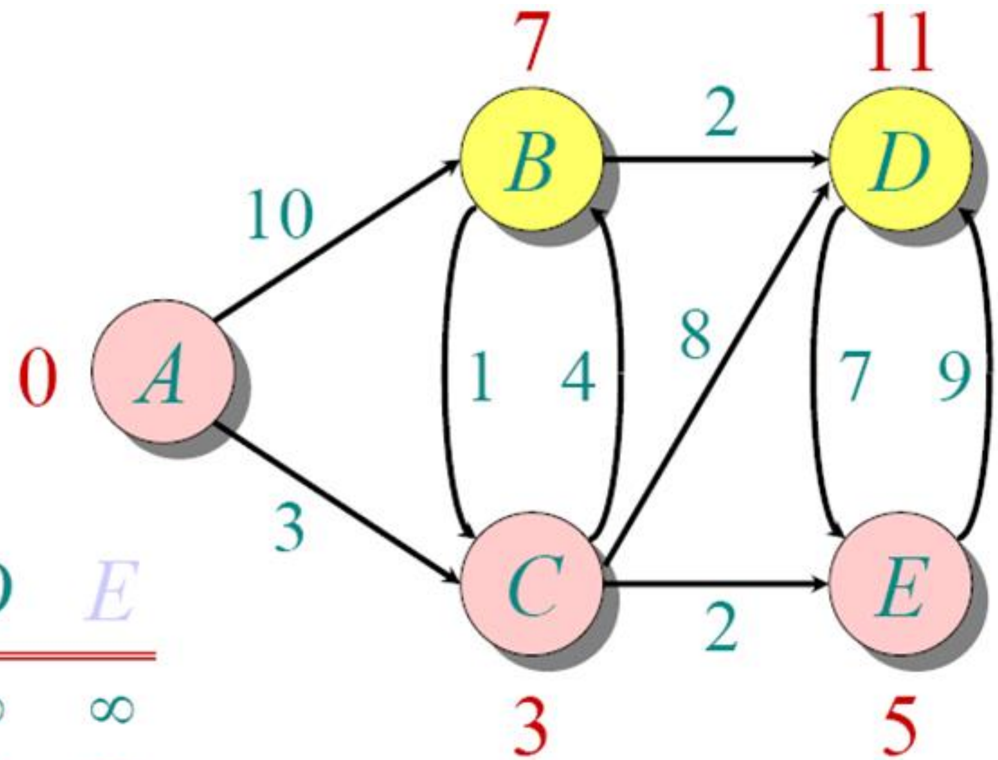
Dijkstra Animated Example



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5

S: { A, C }

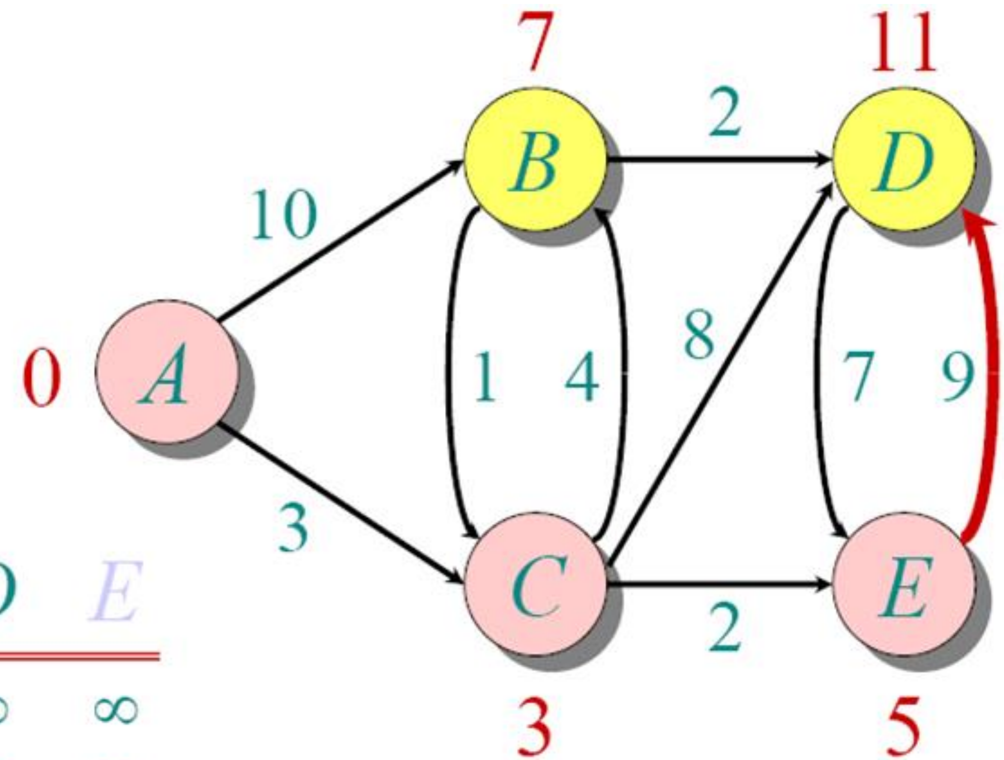
Dijkstra Animated Example



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5

S: { A, C, E }

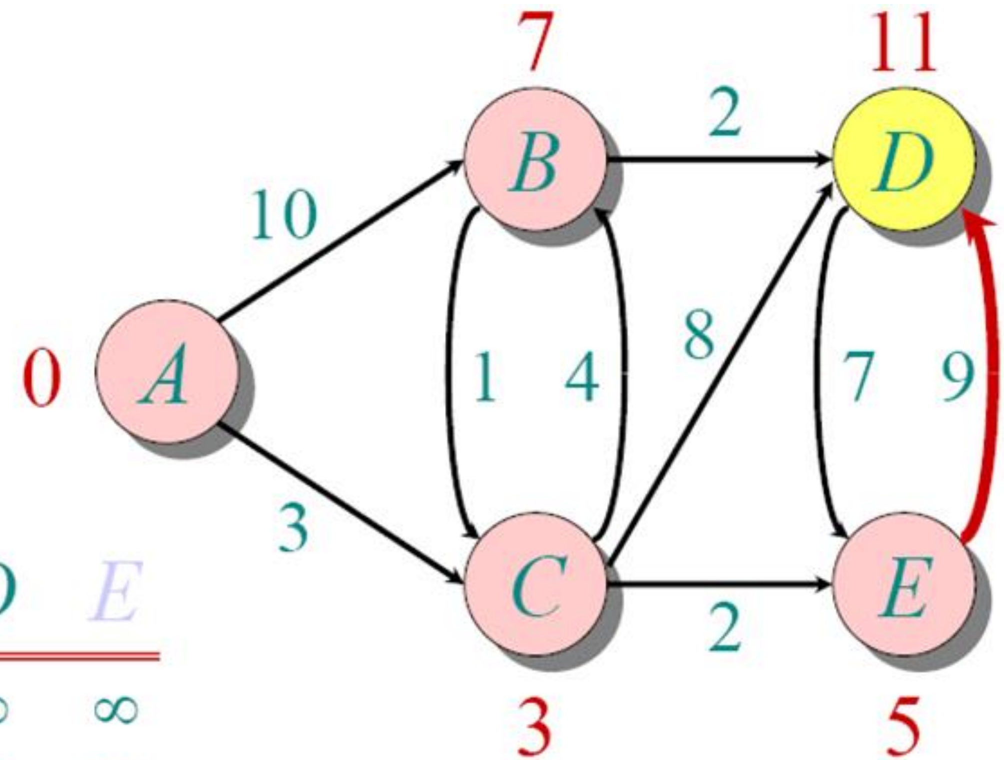
Dijkstra Animated Example



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5
		7		11	

S: { A, C, E }

Dijkstra Animated Example

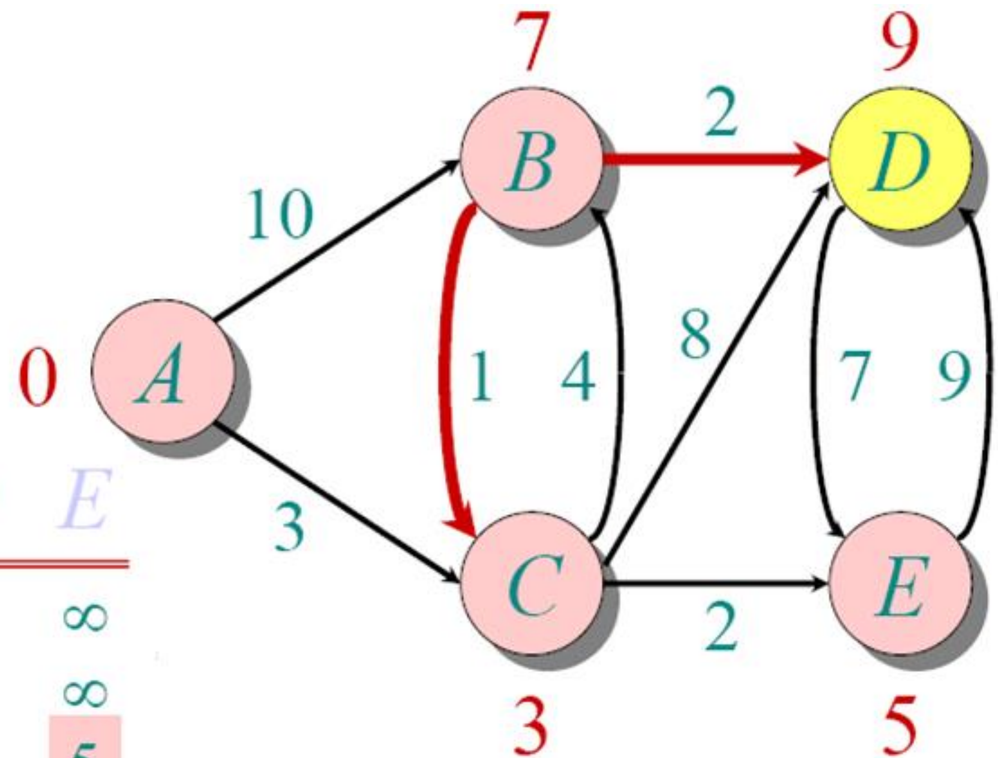


Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5
		7		11	

S: { A, C, E, B }

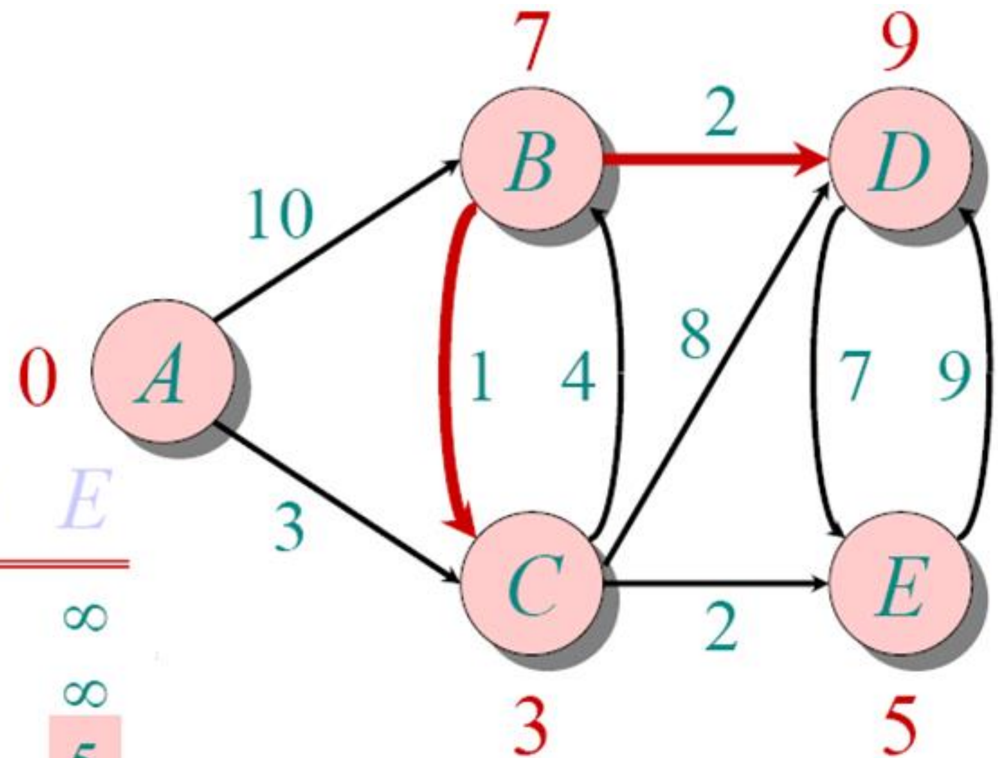
Dijkstra Animated Example

Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5
		7		11	
				9	



S: { A, C, E, B }

Dijkstra Animated Example



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5
		7		11	
				9	

S: { A, C, E, B, D }

Why it works

- A formal proof would take longer than this presentation, but we can understand how the argument works intuitively
  - Think of Djikstra’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 efficently:

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

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**)

Implementation

OVERVIEW PACKAGE CLASS USE TREE DEPRECATED INDEX HELP

PREV CLASS NEXT CLASS FRAMES NO FRAMES

SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

org.jgrapht.alg.shortestpath

Class DijkstraShortestPath<V,E>

java.lang.Object
    org.jgrapht.alg.shortestpath.DijkstraShortestPath<V,E>

Type Parameters:

V - the graph vertex type

E - the graph edge type

All Implemented Interfaces:

ShortestPathAlgorithm<V,E>

```
public final class DijkstraShortestPath<V,E>
extends Object
```

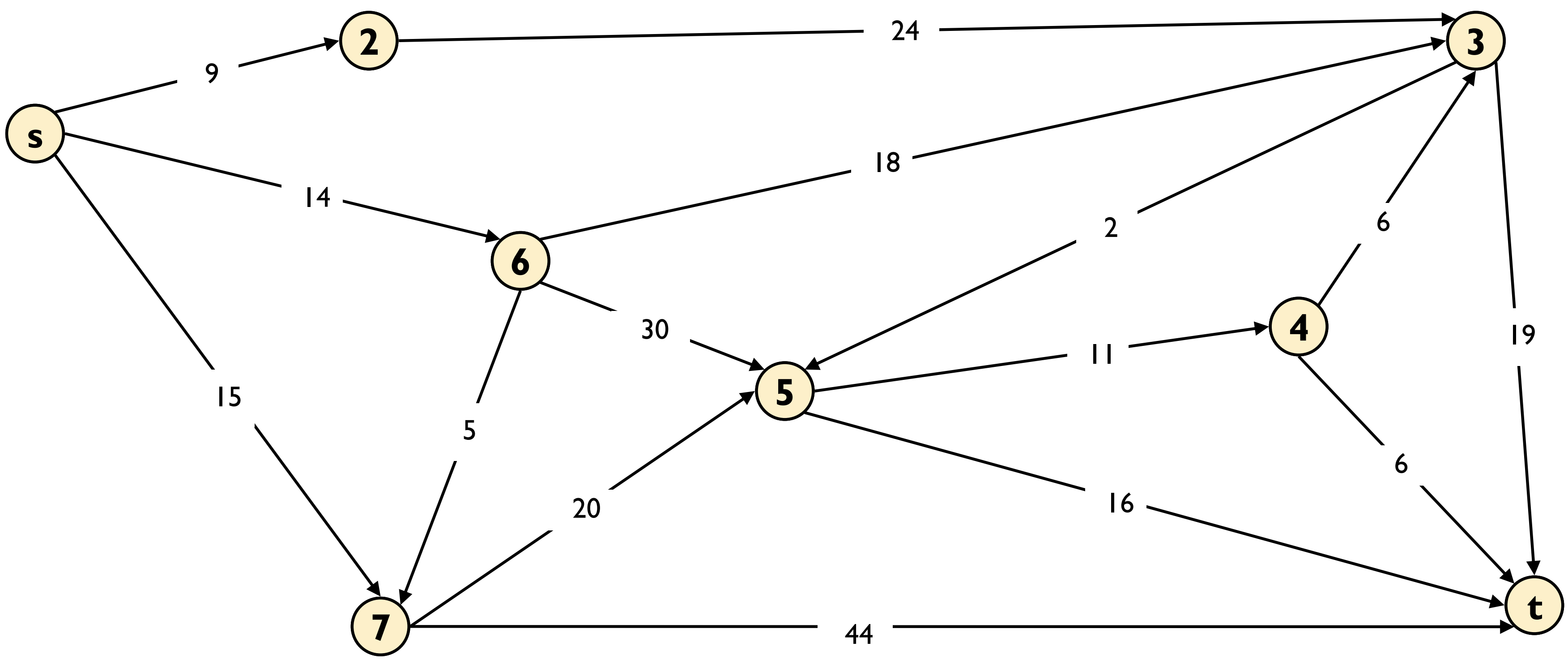
An implementation of Dijkstra's shortest path algorithm using a Fibonacci heap.

Author:

John V. Sichi

Dijkstra's Shortest Path Algorithm

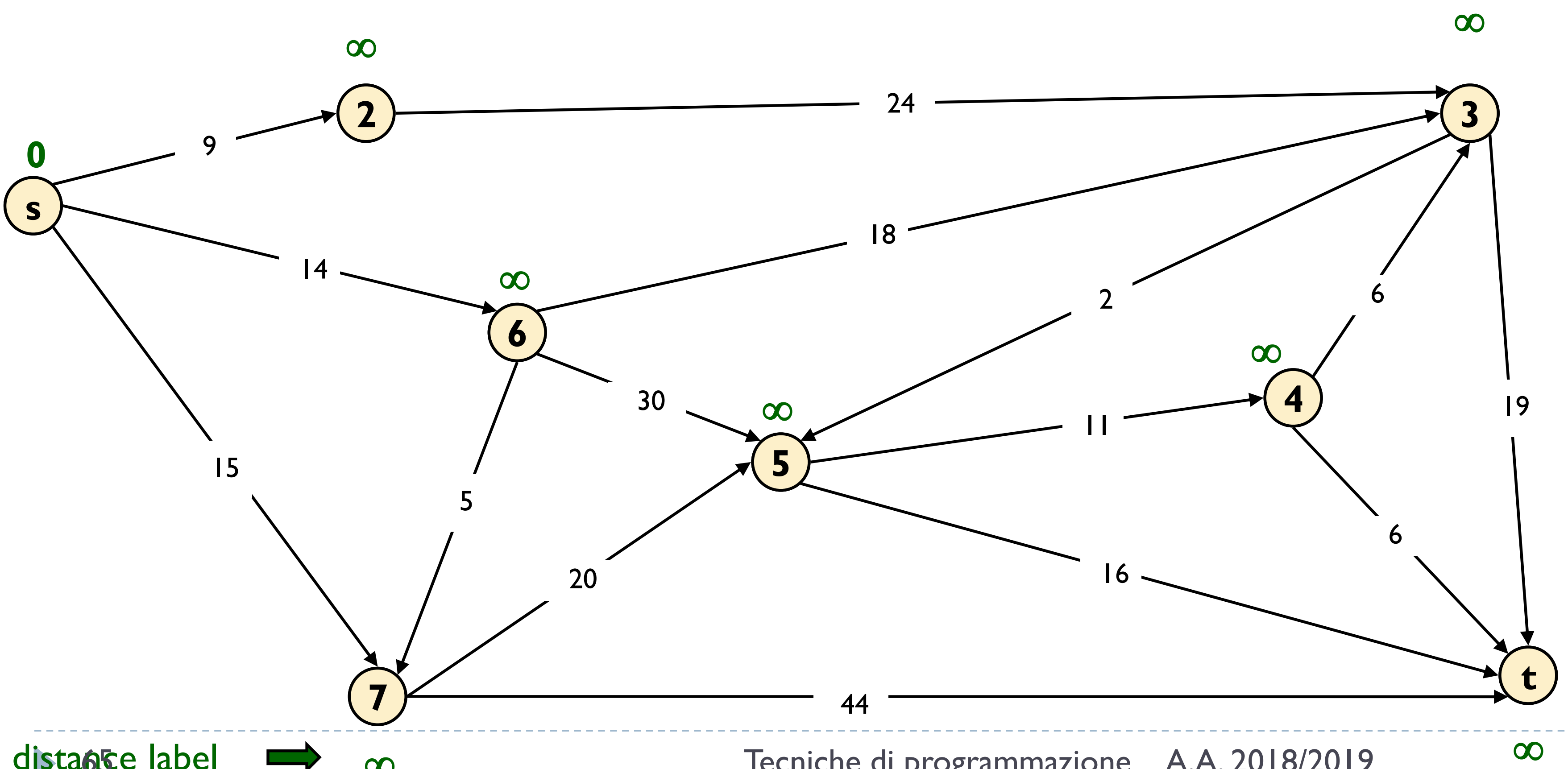
- Find shortest path from **s** to **t**



Dijkstra's Shortest Path Algorithm

S = { }

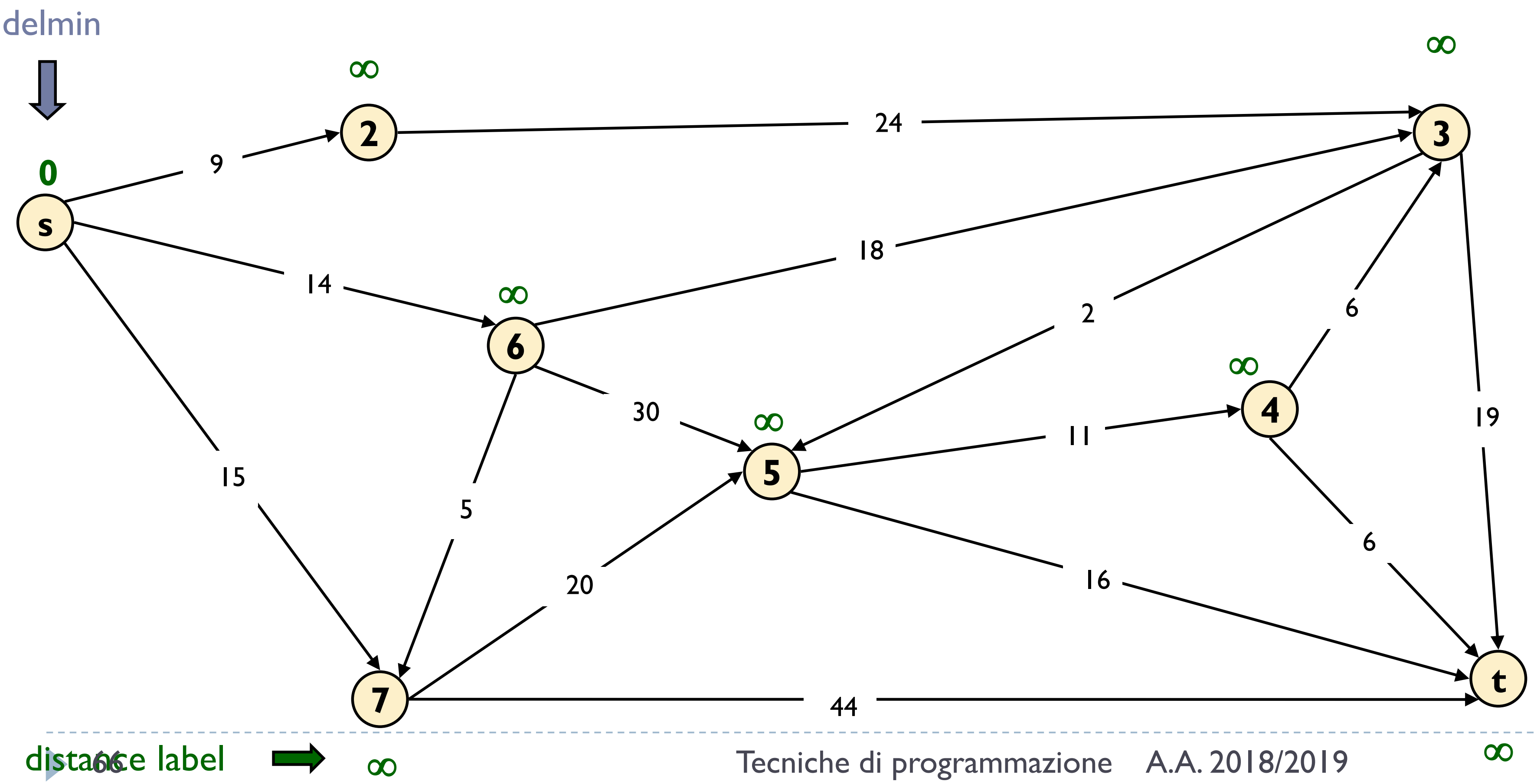
Q = { s, 2, 3, 4, 5, 6, 7, t }



Dijkstra's Shortest Path Algorithm

S = { }

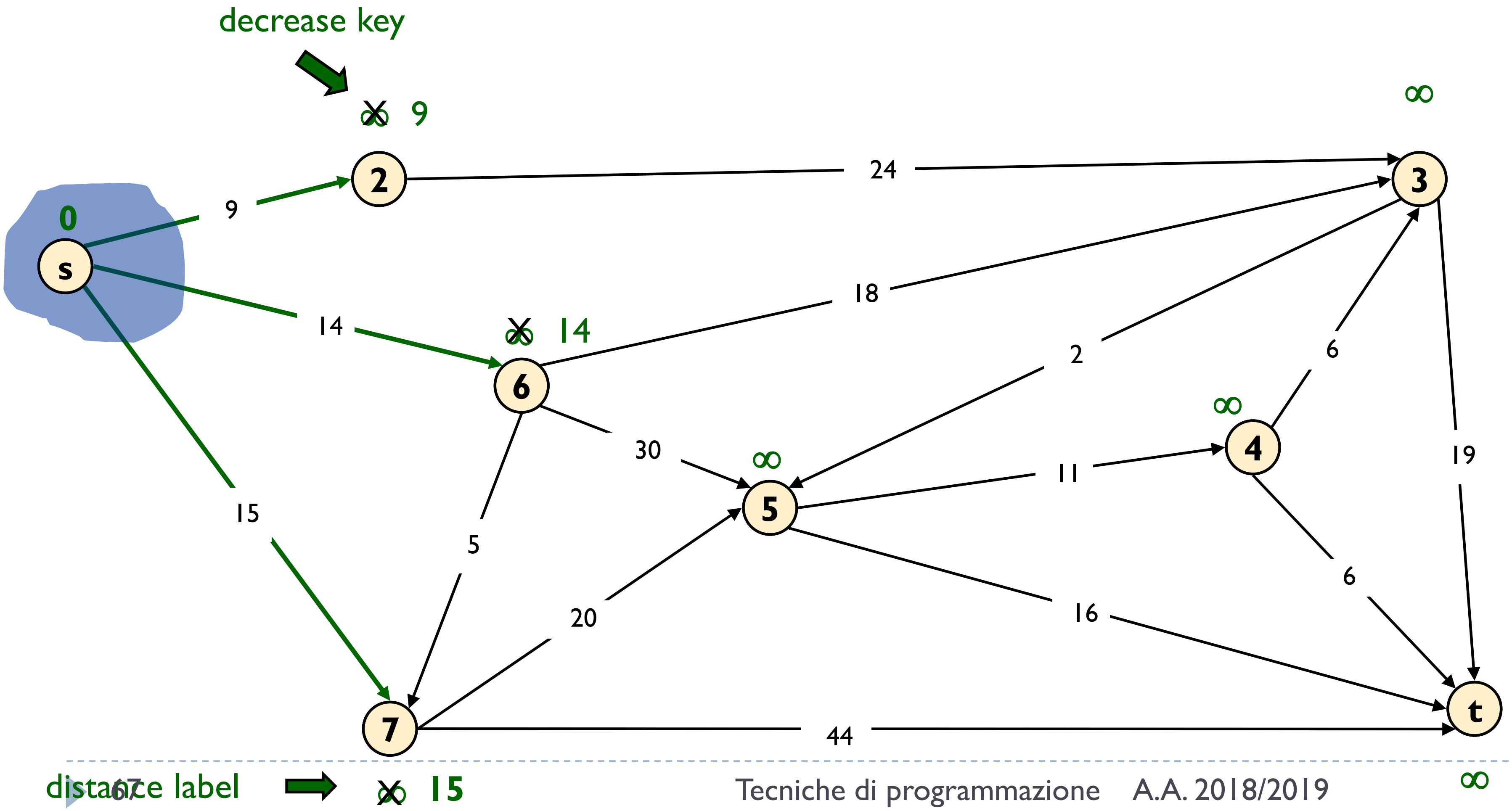
Q = { s, 2, 3, 4, 5, 6, 7, t }



Dijkstra's Shortest Path Algorithm

S = { s }

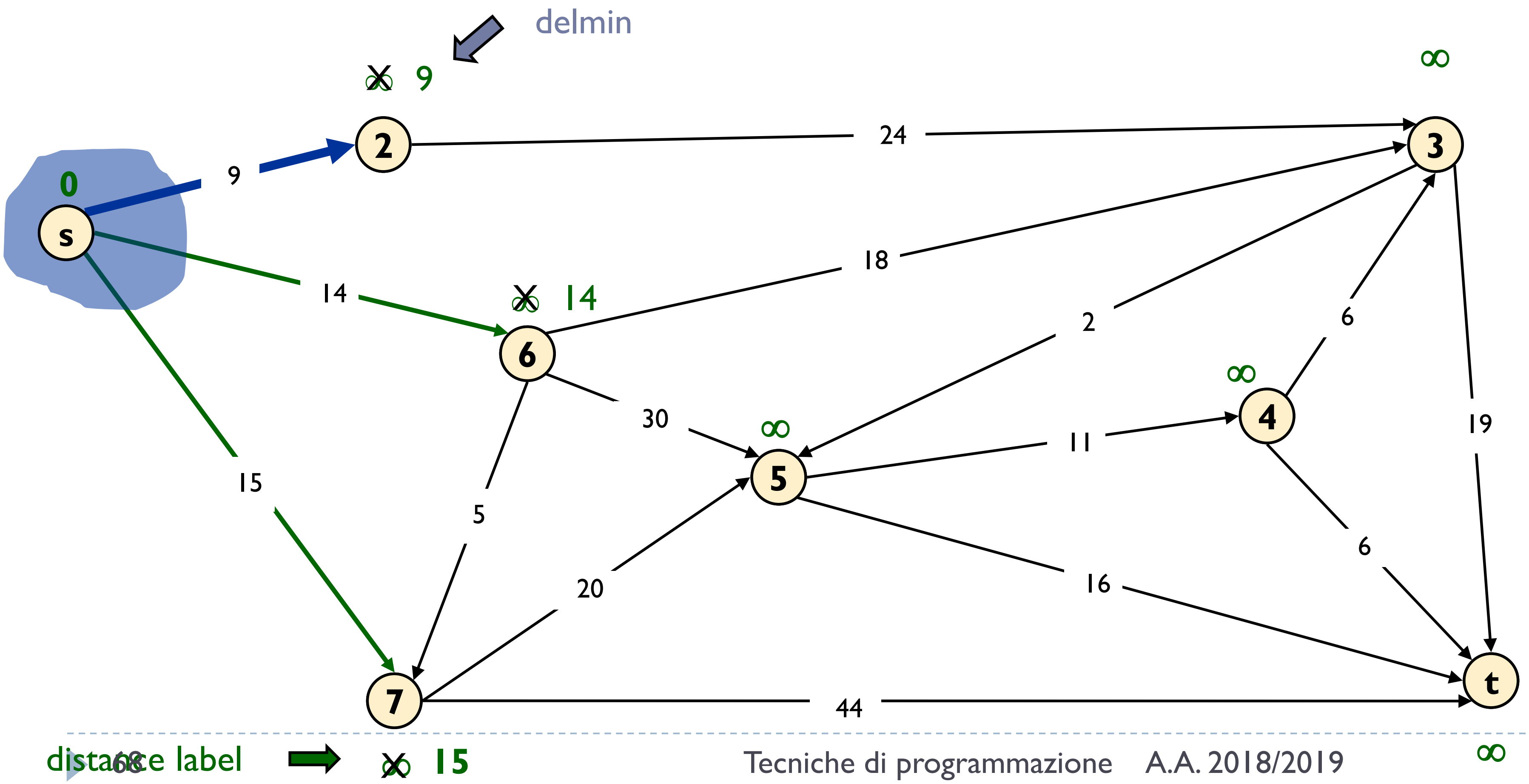
Q = { 2, 3, 4, 5, 6, 7, t }



Dijkstra's Shortest Path Algorithm

S = { s }

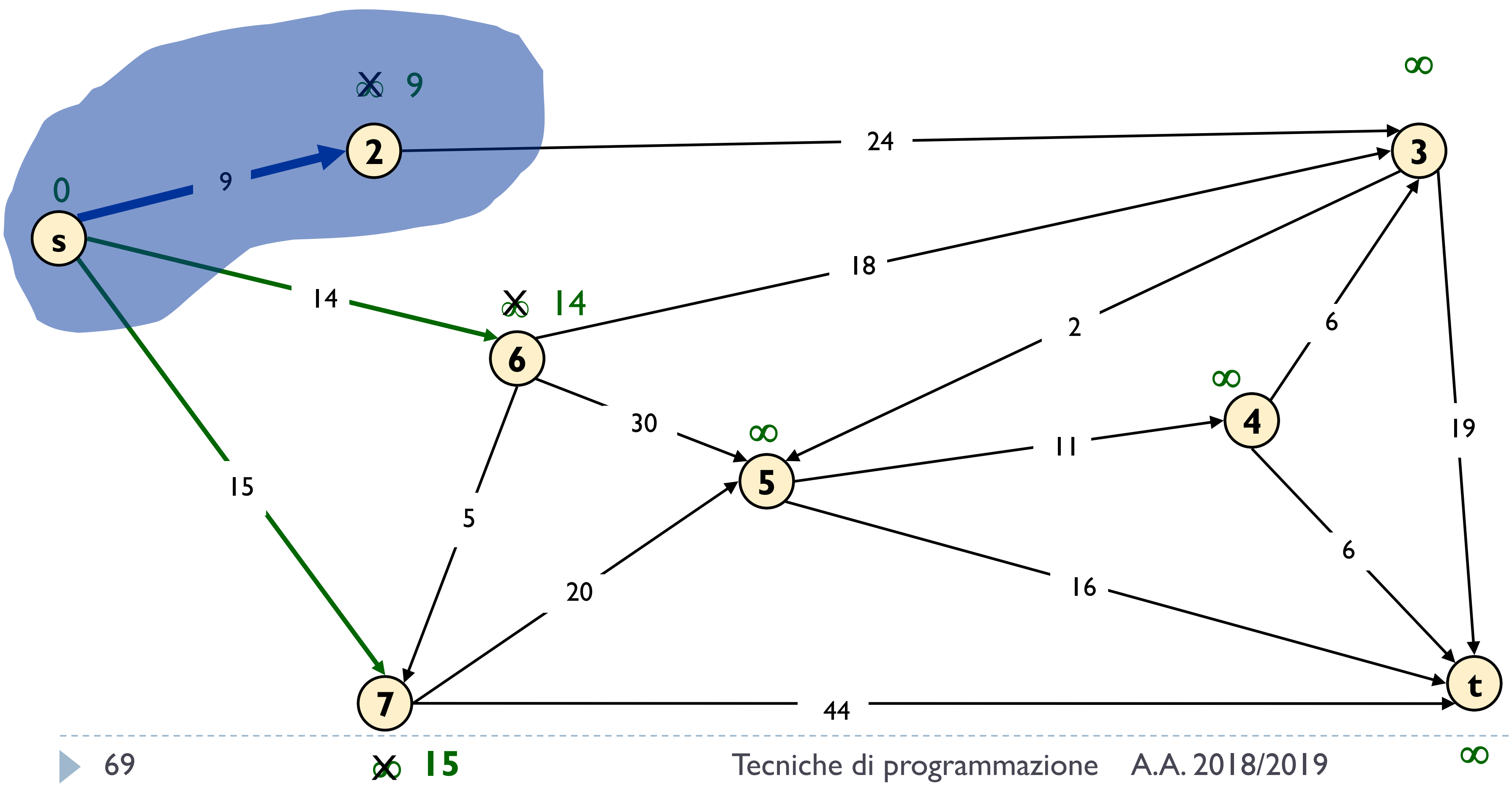
Q = { 2, 3, 4, 5, 6, 7, t }



Dijkstra's Shortest Path Algorithm

S = { s, 2 }

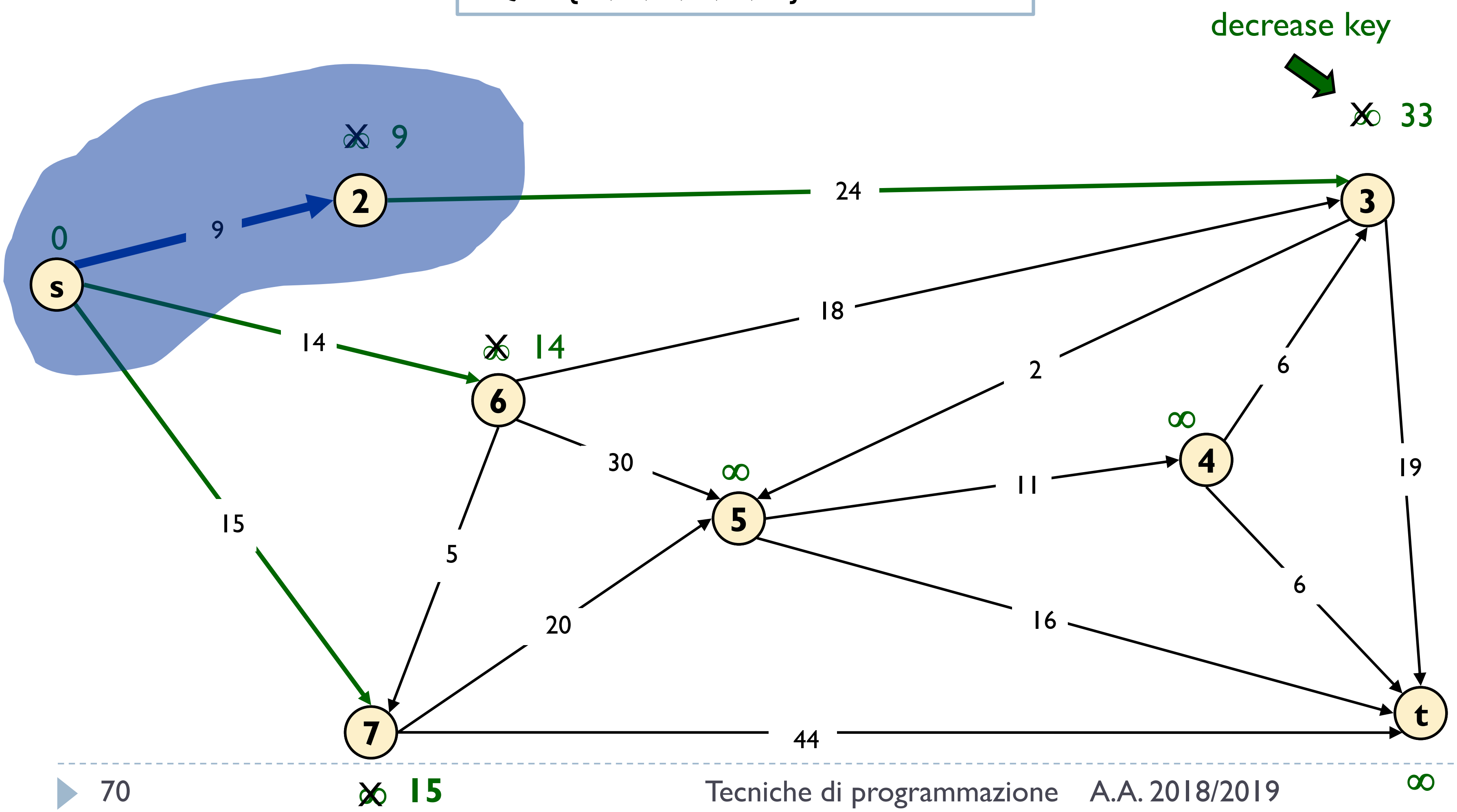
Q = { 3, 4, 5, 6, 7, t }



Dijkstra's Shortest Path Algorithm

S = { s, 2 }

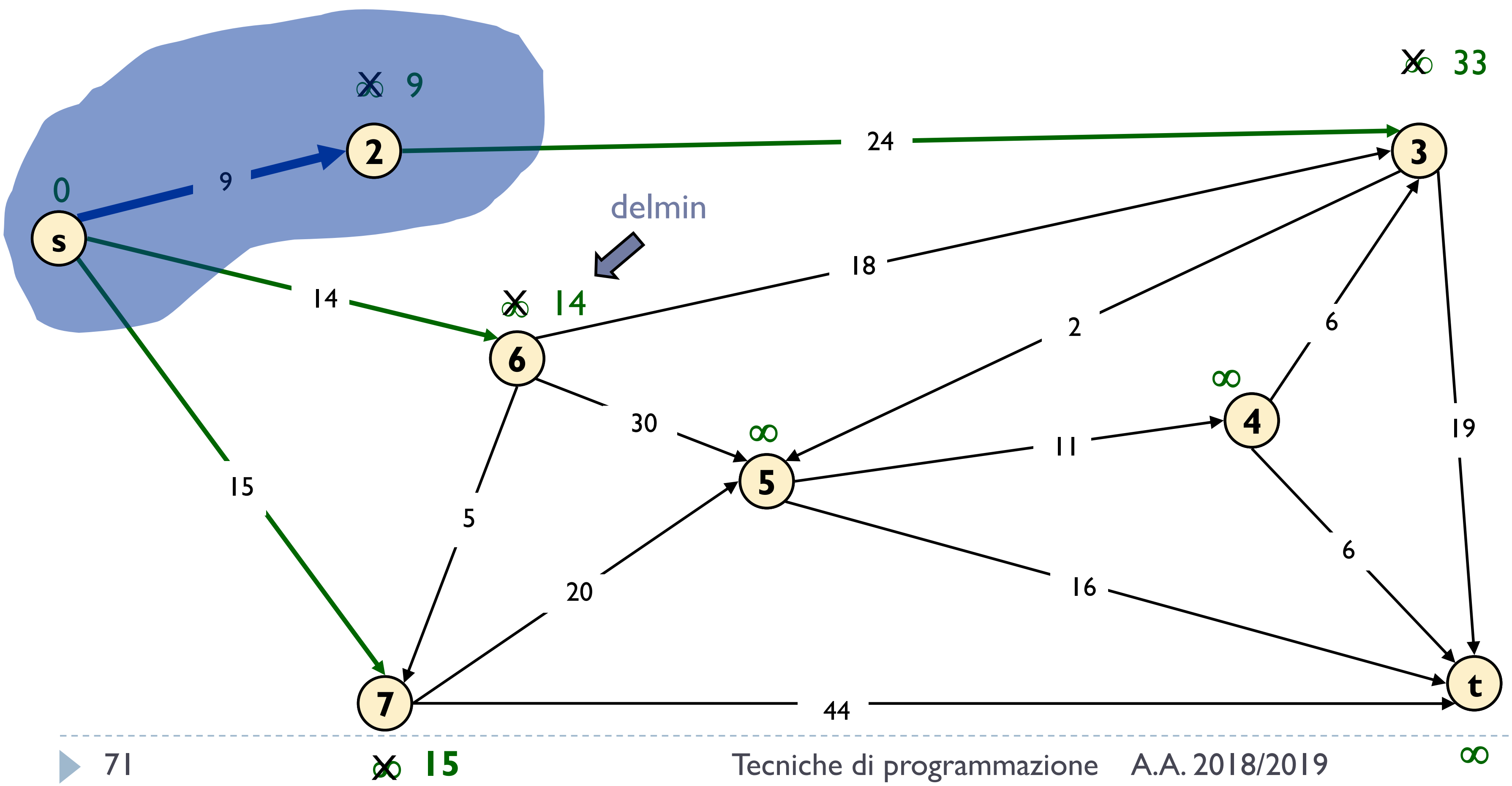
Q = { 3, 4, 5, 6, 7, t }



Dijkstra's Shortest Path Algorithm

S = { s, 2 }

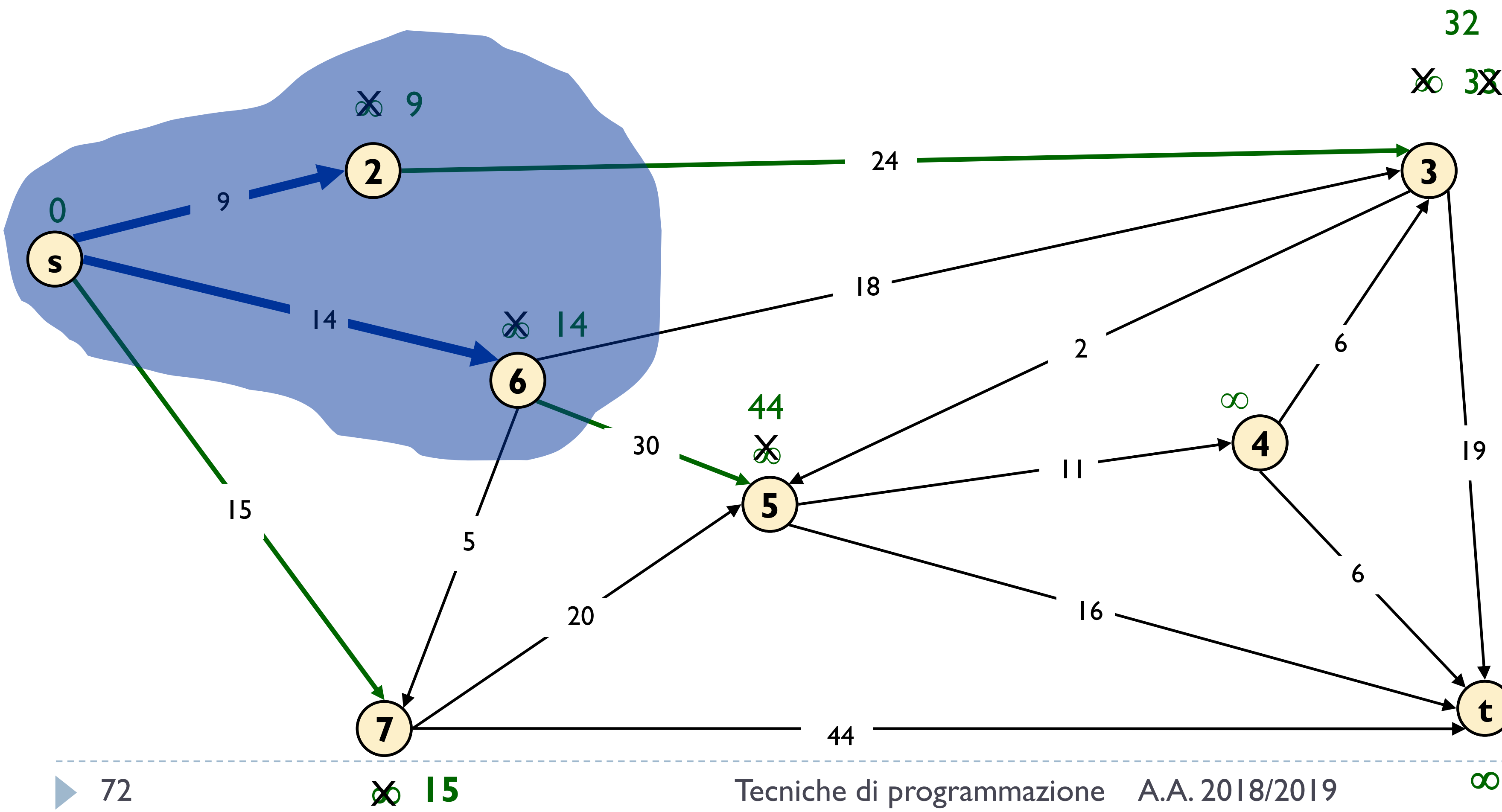
Q = { 3, 4, 5, 6, 7, t }



Dijkstra's Shortest Path Algorithm

S = { s, 2, 6 }

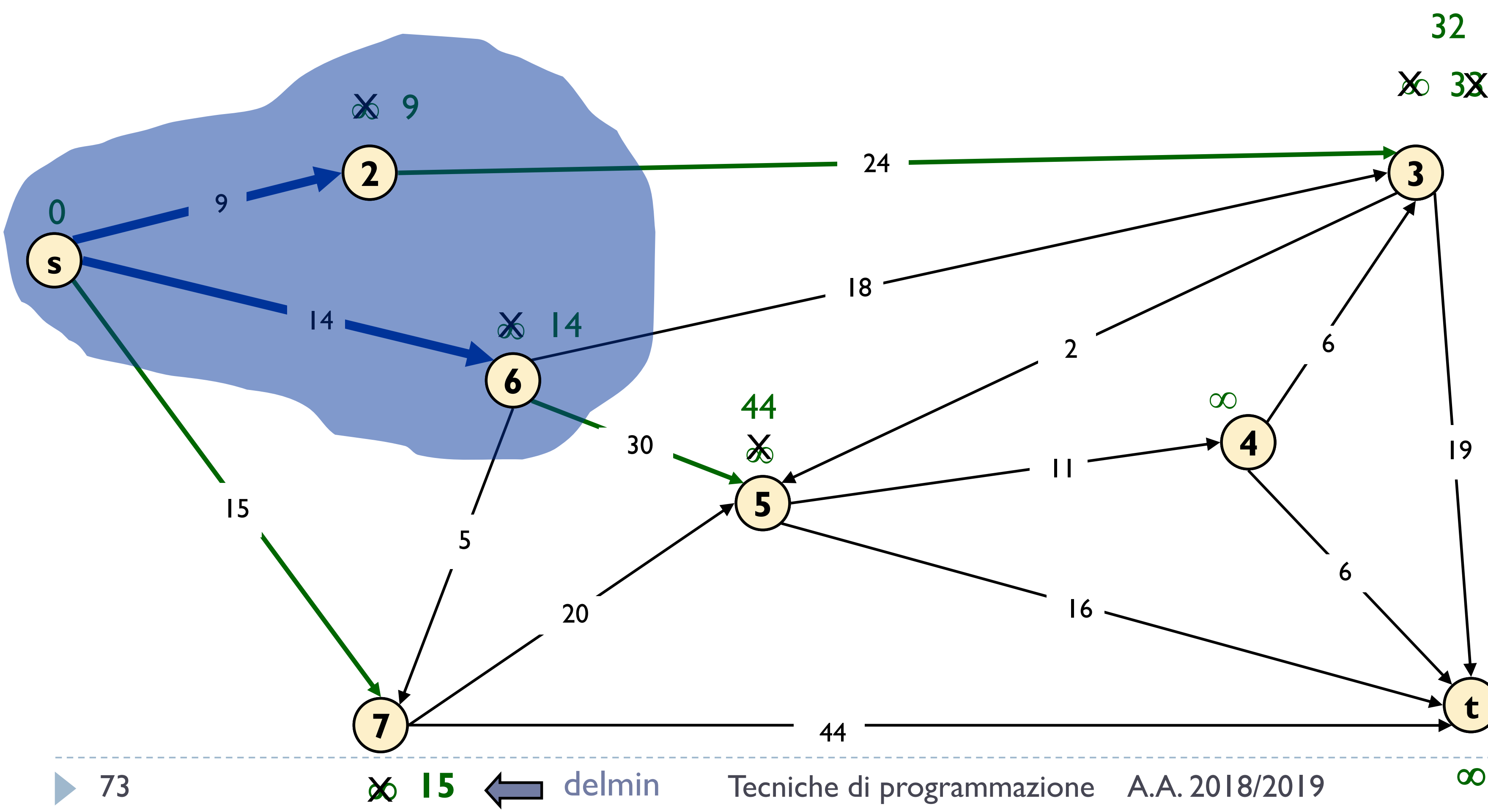
Q = { 3, 4, 5, 7, t }



Dijkstra's Shortest Path Algorithm

S = { s, 2, 6 }

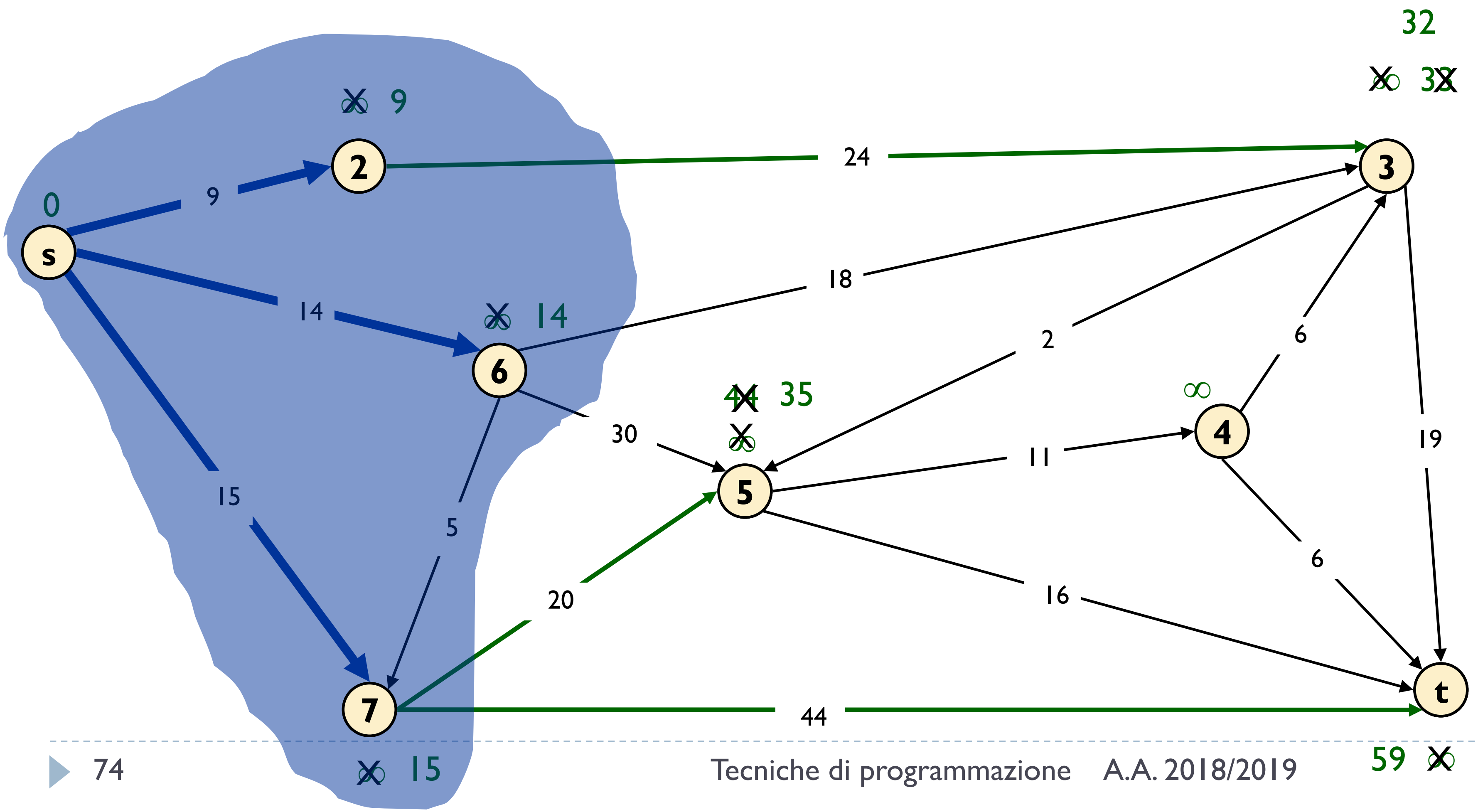
Q = { 3, 4, 5, 7, t }



Dijkstra's Shortest Path Algorithm

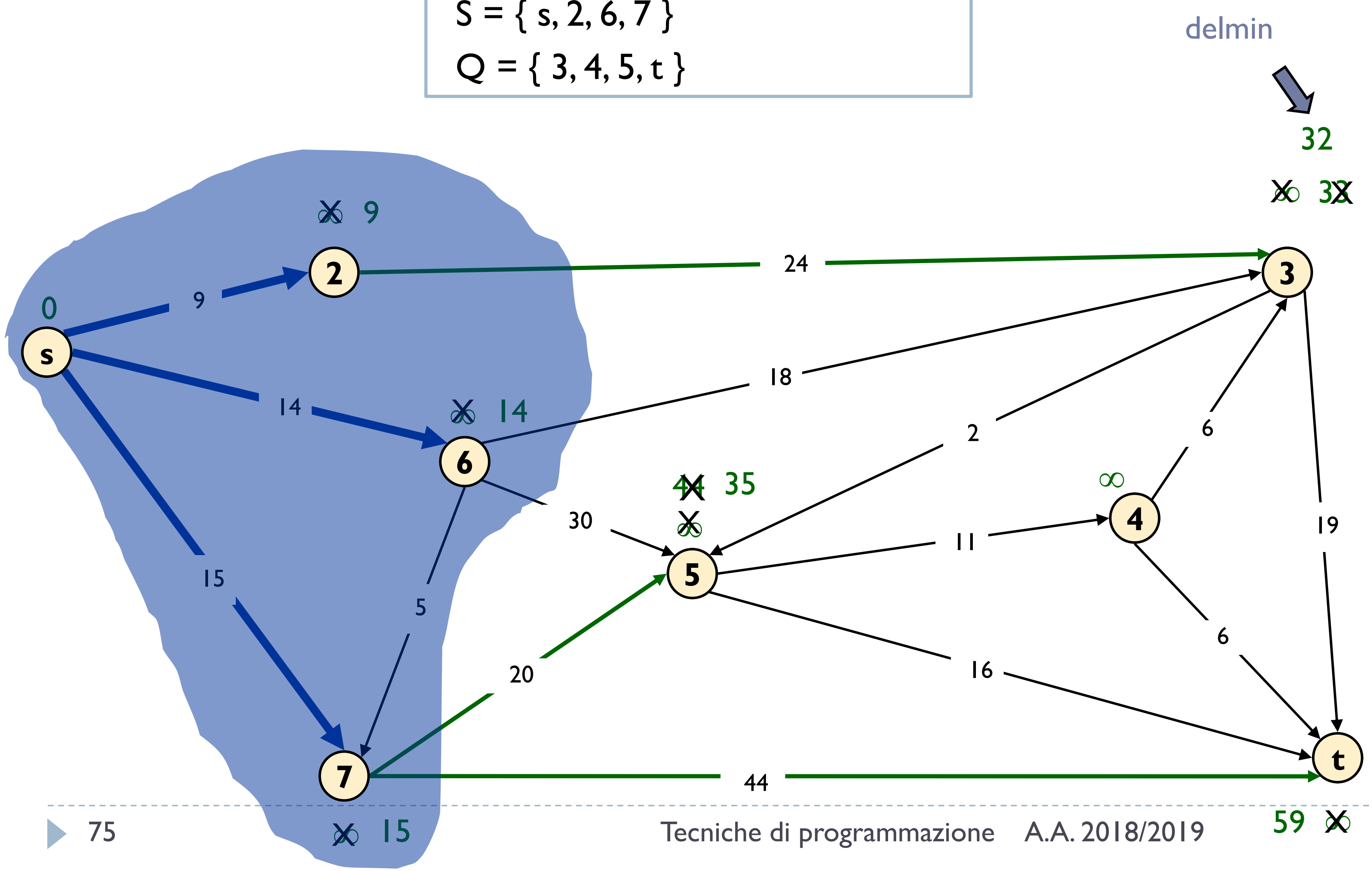
S = { s, 2, 6, 7 }

Q = { 3, 4, 5, t }



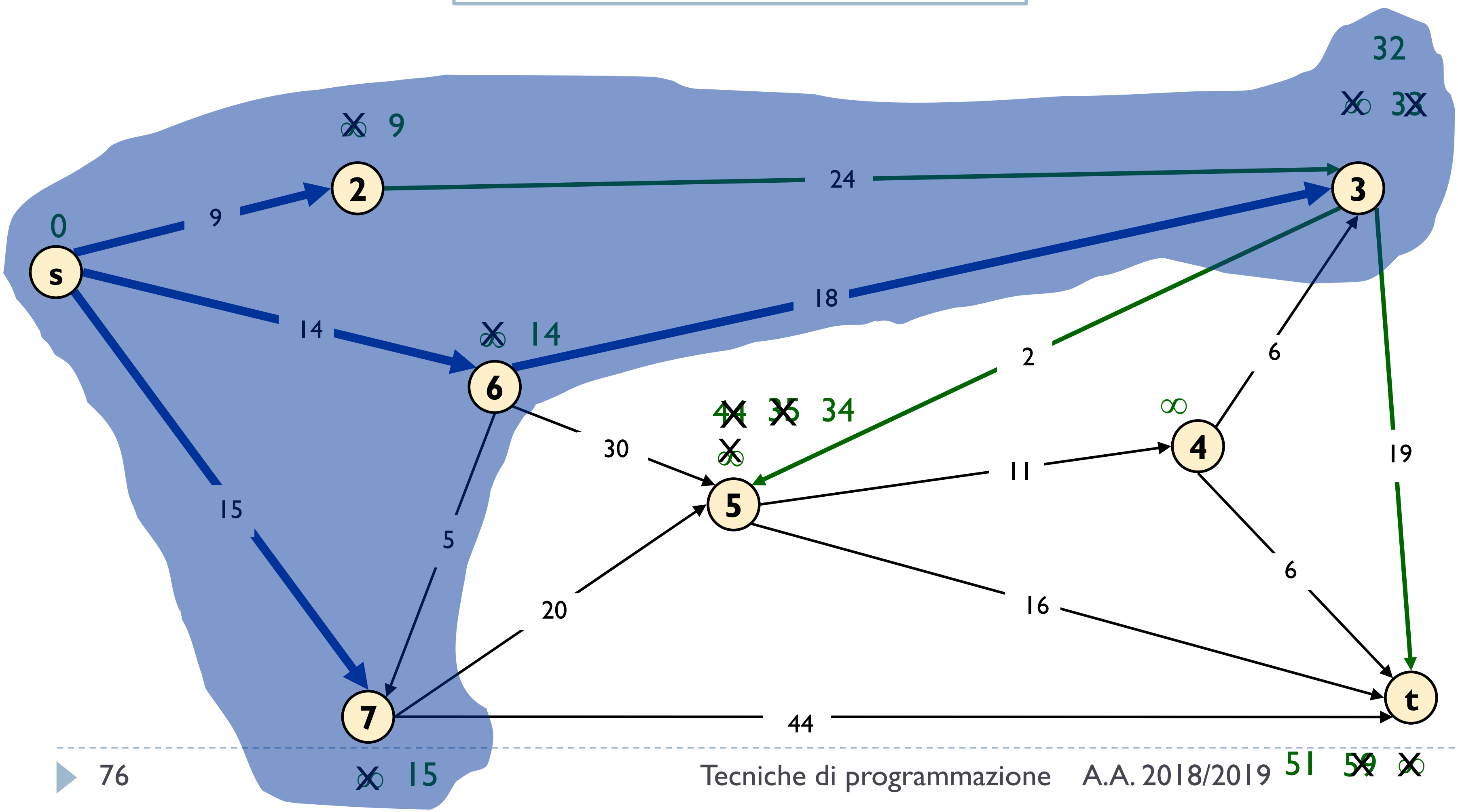
Dijkstra's Shortest Path Algorithm

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



Dijkstra's Shortest Path Algorithm

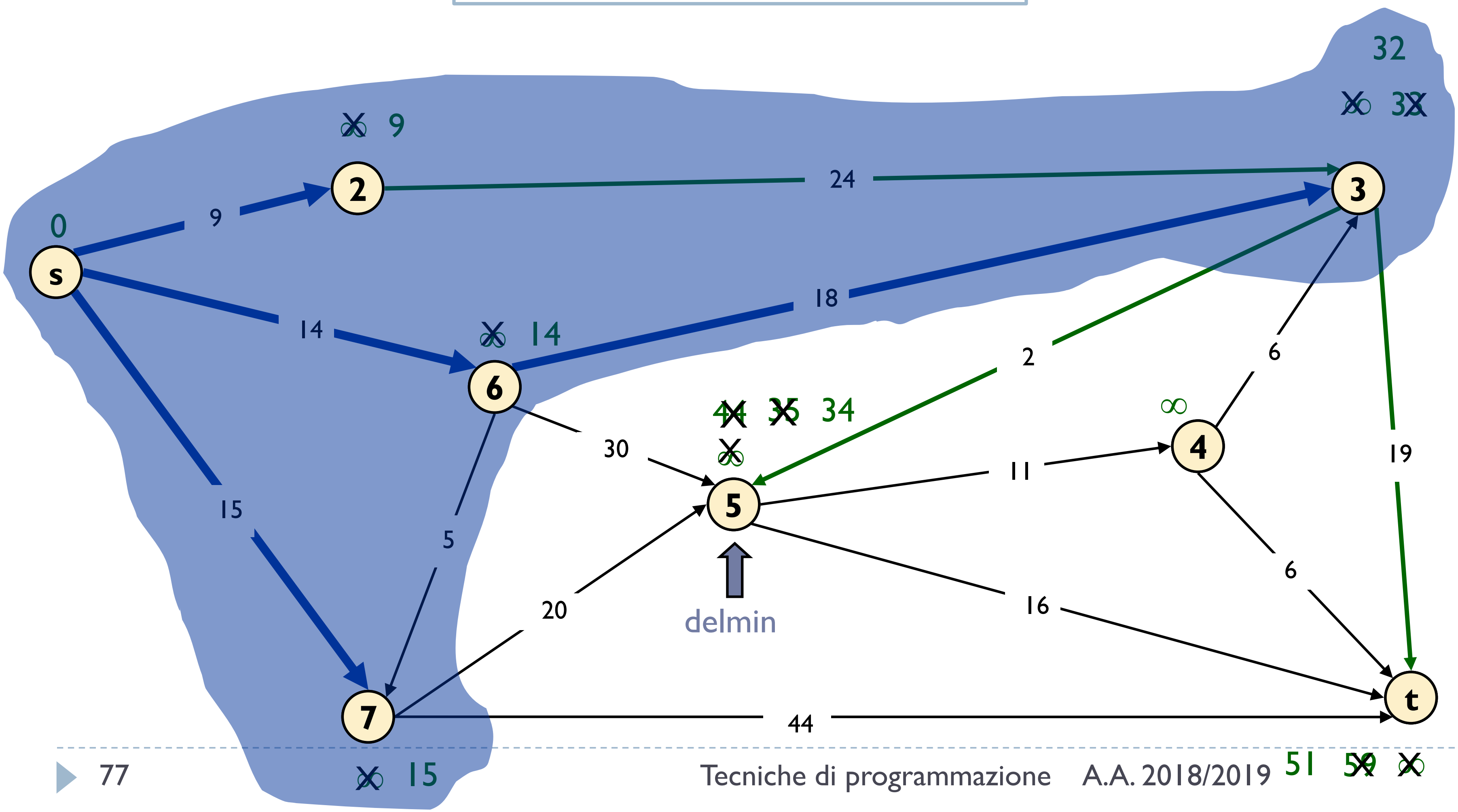
S = { s, 2, 3, 6, 7 }
Q = { 4, 5, t }



Dijkstra's Shortest Path Algorithm

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

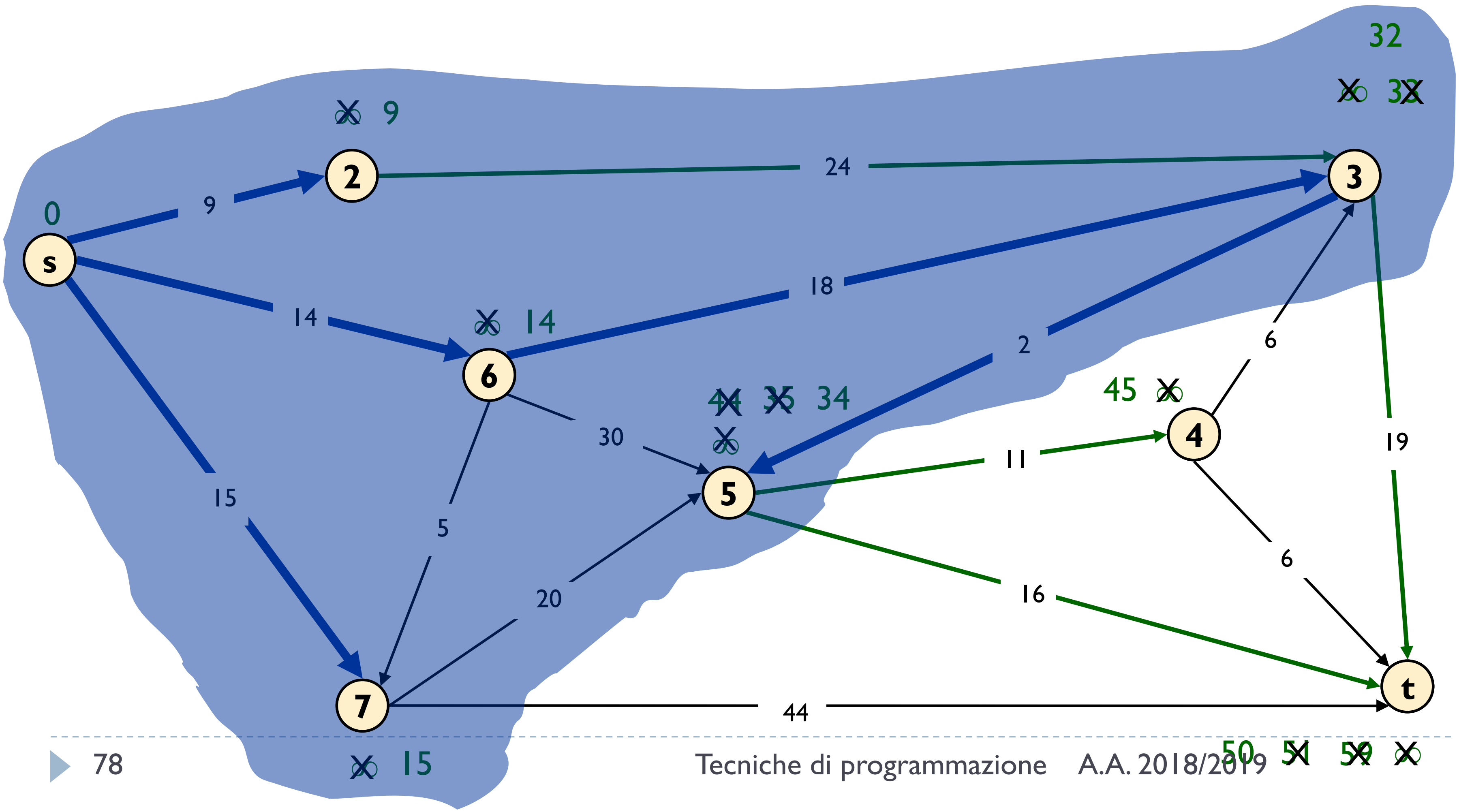
Q = { 4, 5, t }



Dijkstra's Shortest Path Algorithm

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

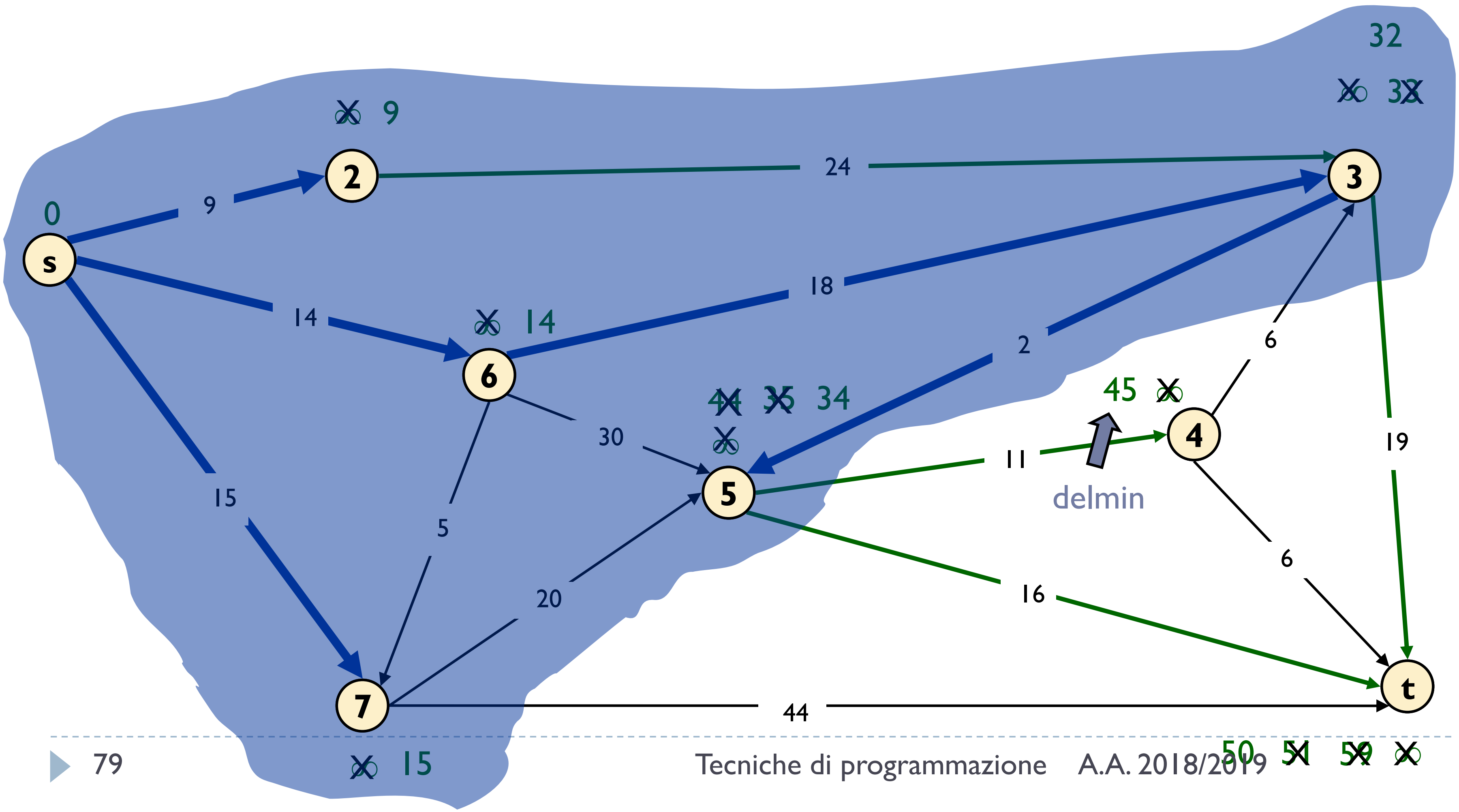
Q = { 4, t }



Dijkstra's Shortest Path Algorithm

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

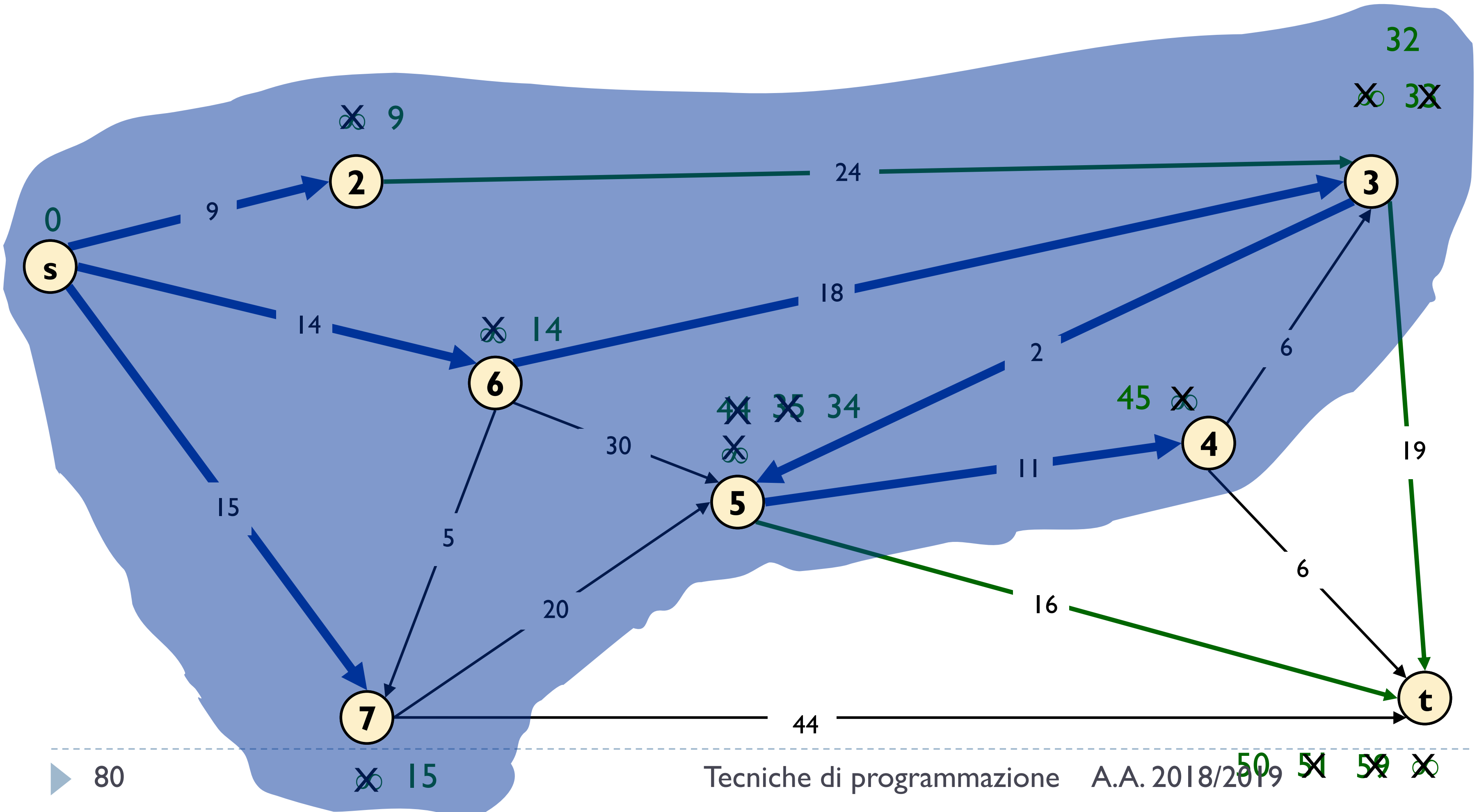
Q = { 4, t }



Dijkstra's Shortest Path Algorithm

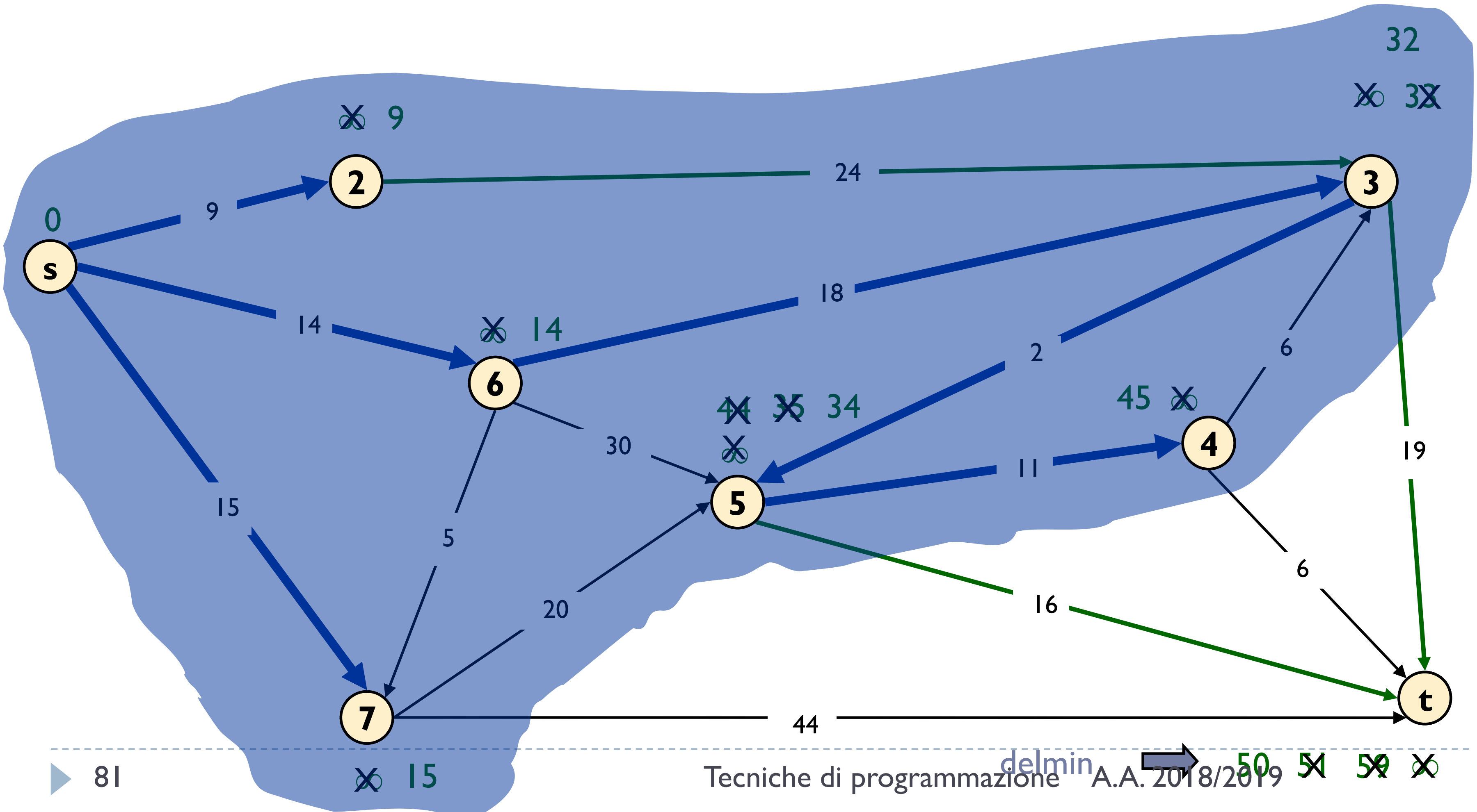
S = { s, 2, 3, 4, 5, 6, 7 }

Q = { t }



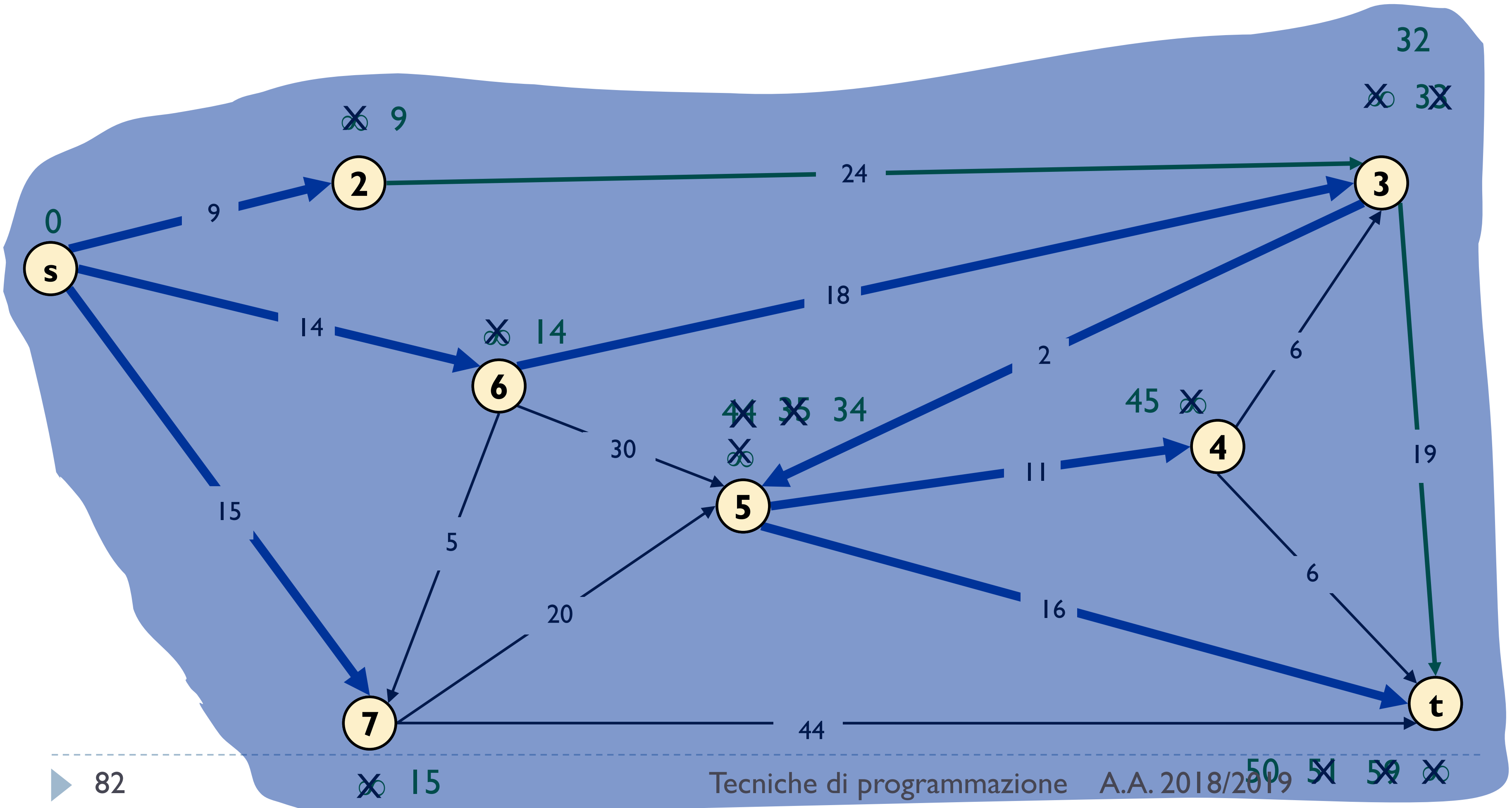
Dijkstra's Shortest Path Algorithm

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



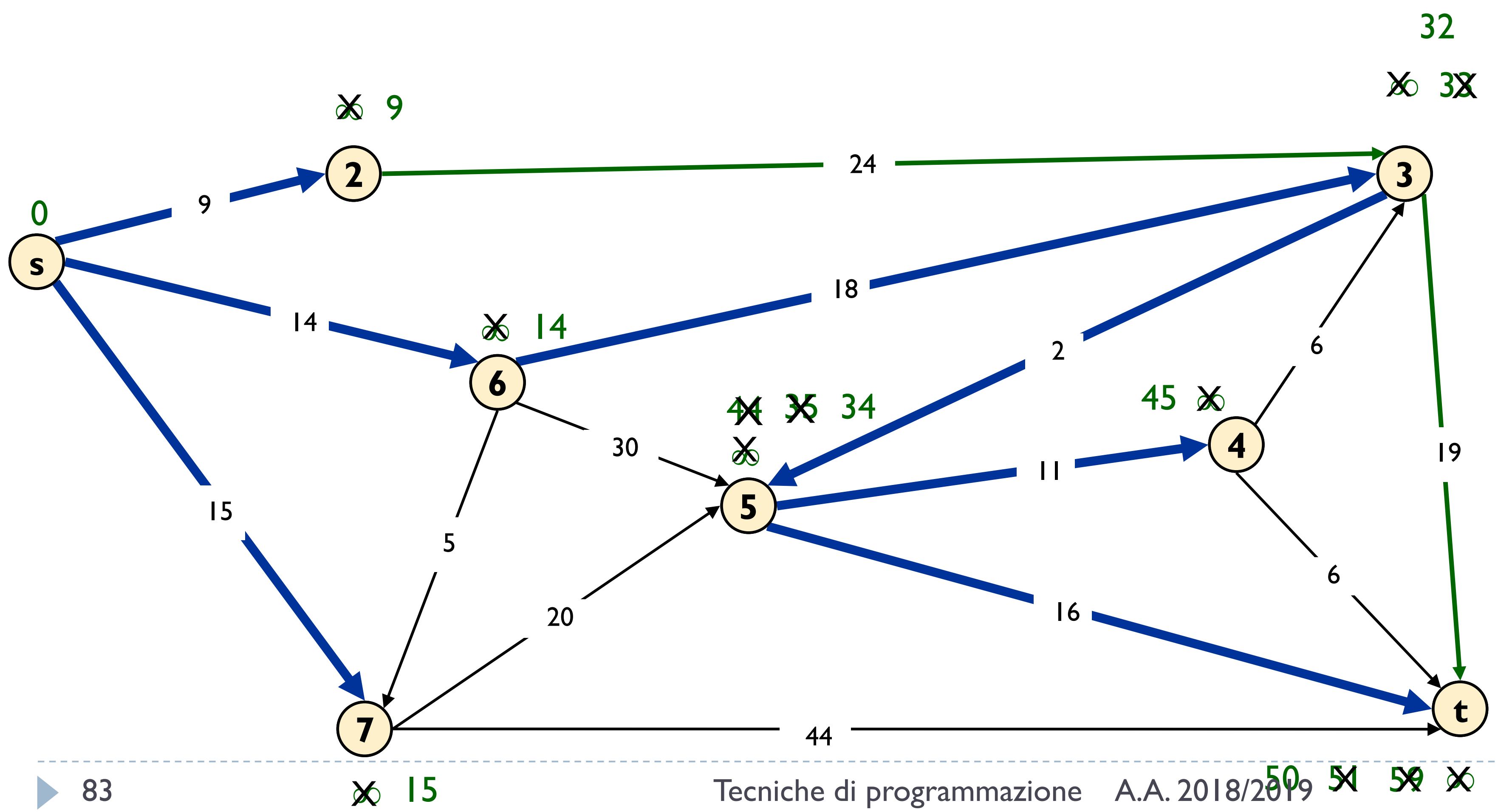
Dijkstra's Shortest Path Algorithm

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/