Politecnico di Torino

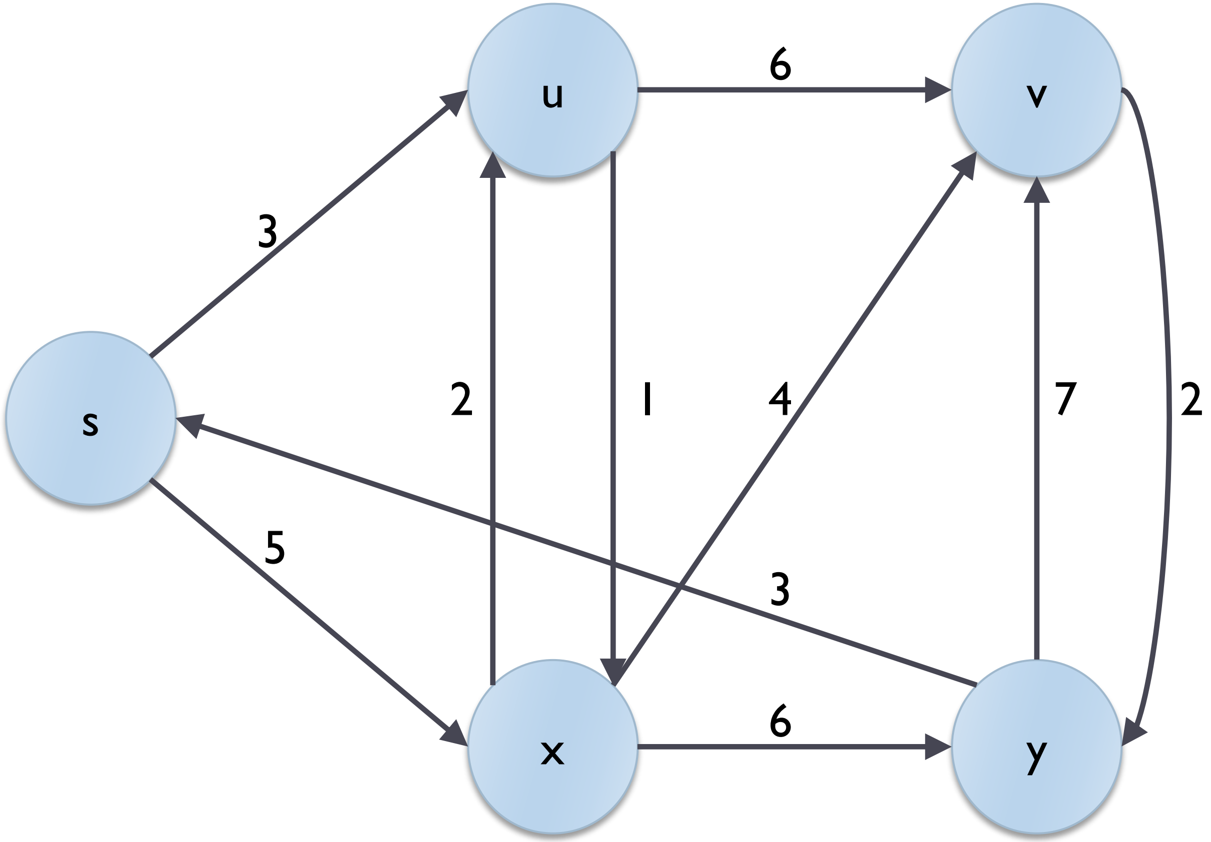
e-Lite

Graphs: paths and cycles

Tecniche di Programmazione – A.A. 2021/2022

Shortest Paths

What is the shortest path between s and v ?



Summary

- Shortest Paths
  - Definitions
  - Floyd-Warshall algorithm
  - Bellman-Ford-Moore algorithm
  - Dijkstra algorithm
- Cycles
  - Definitions
  - Algorithms

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 (by definition) $\delta(u,v)=\infty$

Finding shortest paths

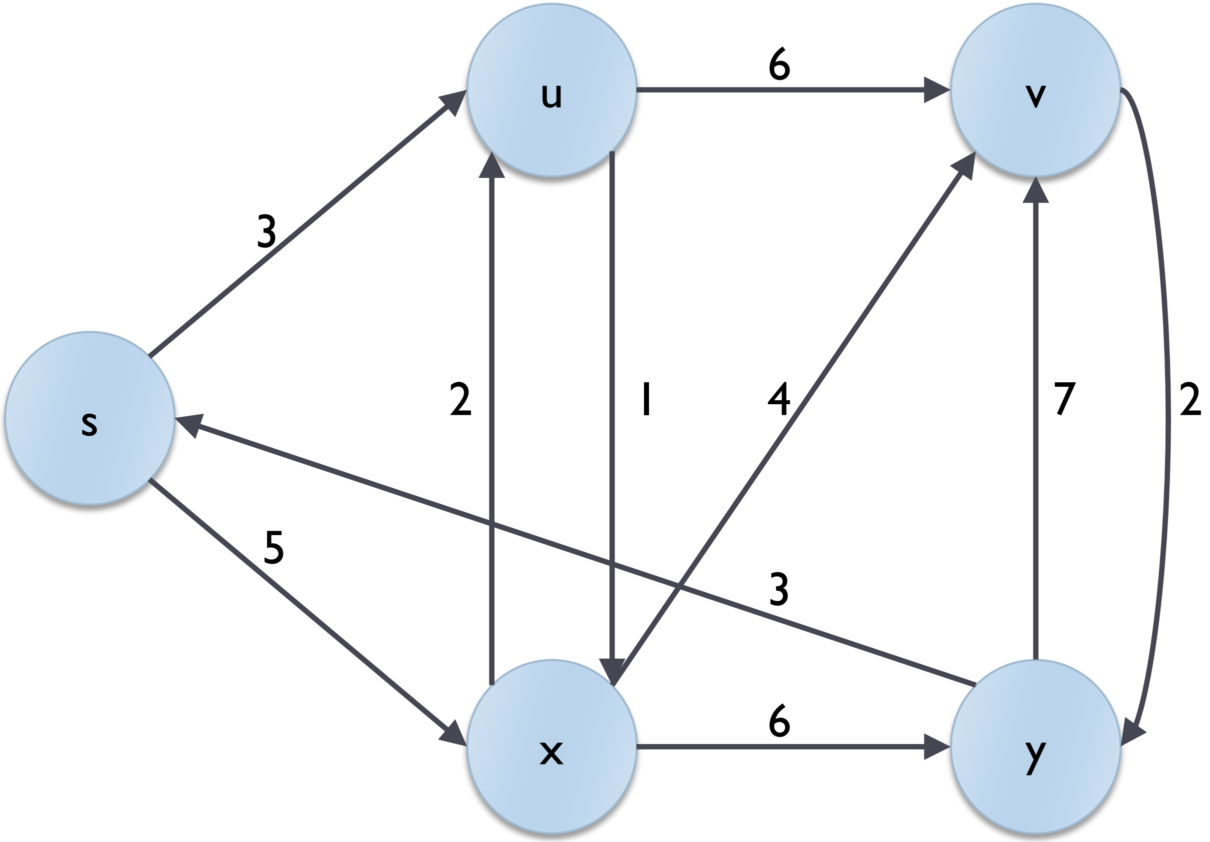
- Single-source shortest path (SS-SP)
  - Given *u* and *v*, find the shortest path between *u* and *v*
  - Given *u*, find the shortest path between *u* and any other vertex
- All-pairs shortest path (AP-SP)
  - Given a graph, find the shortest path between any pair of vertices

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

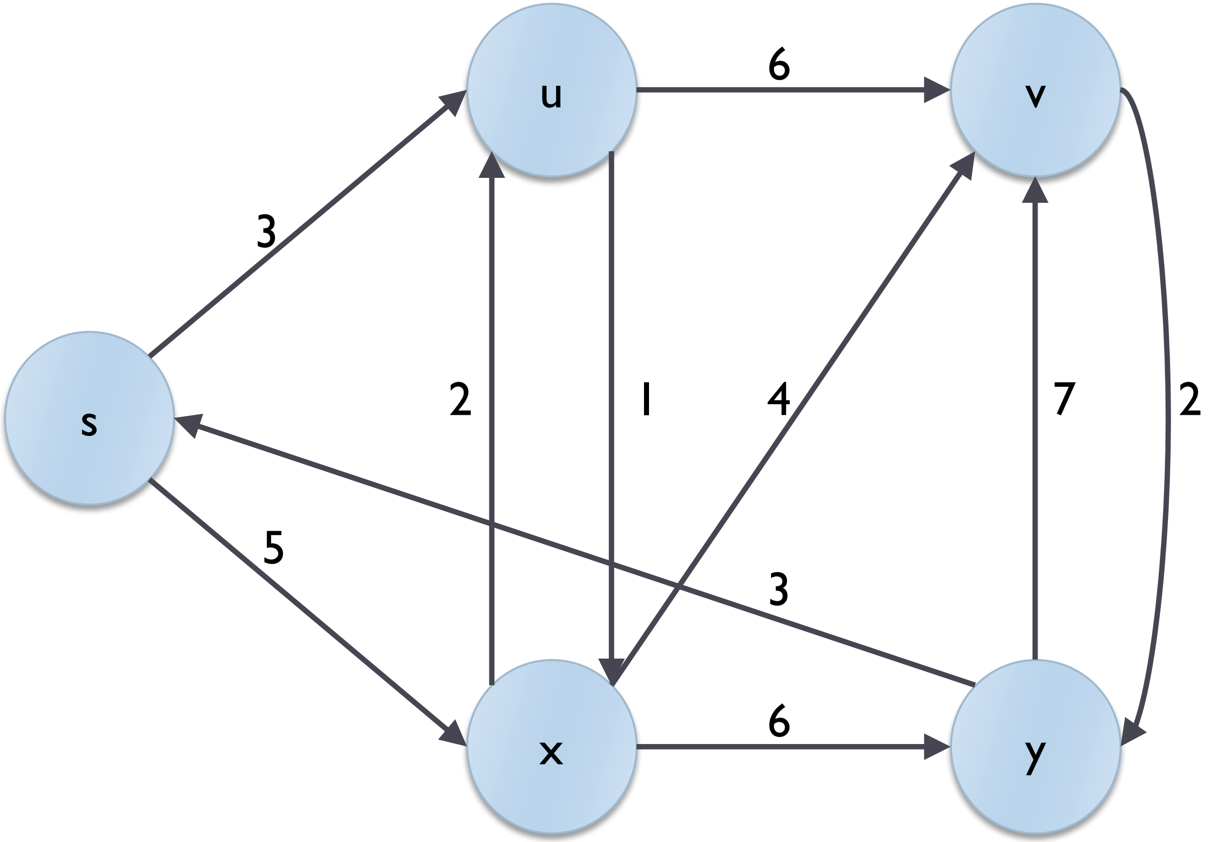
What is the shortest path between s and v ?



Representing shortest paths

- A data structure to represent all shortest paths from a single source *u*, may include
  - 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

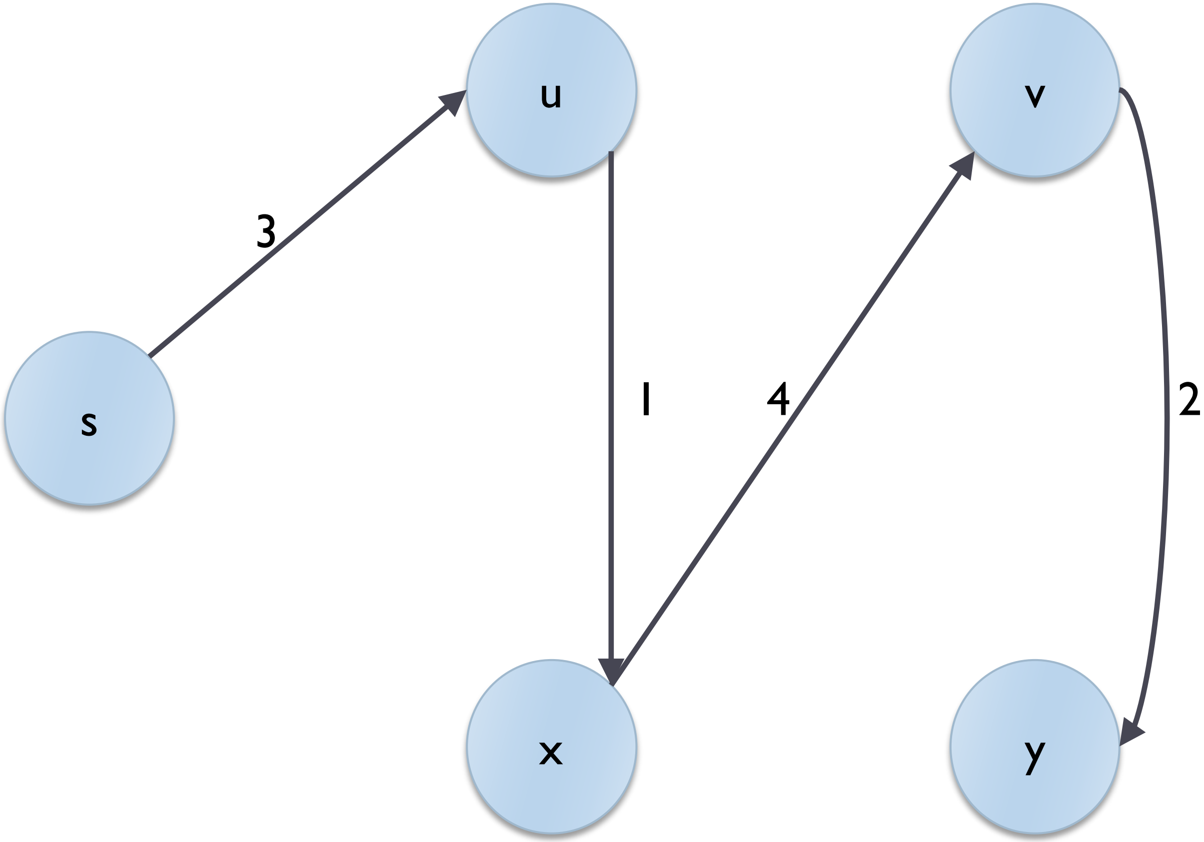
Shortest path graph

- 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

- $G\pi$ 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

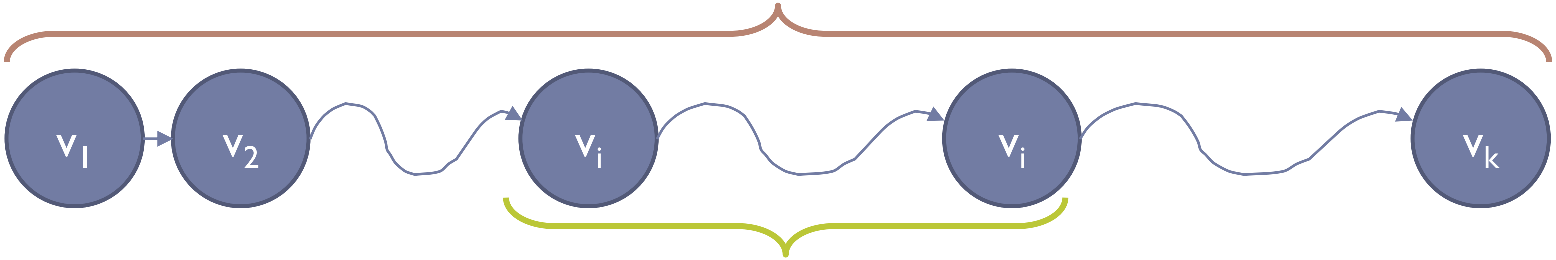
Special case

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

Lemma

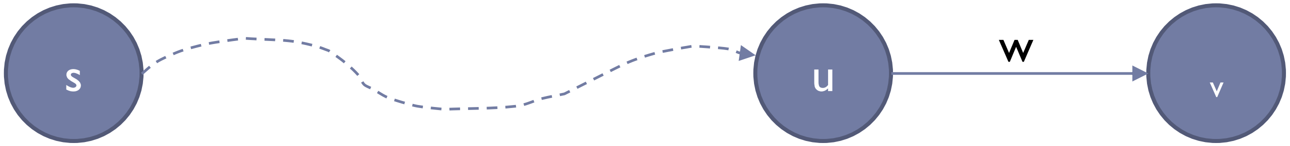
- Consider an ordered weighted graph G=(V,E), with weight function w: E→ℝ.
- Let $p=<v_1, v_2, \ldots, v_k>$ a shortest path from vertex v_1 to vertex v_k.
- For all i,j such that $1 \le i \le j \le k$, let $p_{ij}=<v_i, v_{i+1}, \ldots, v_j>$ be the sub-path of p, from vertex v_i to vertex v_j.

- Therefore, p_{ij} is a shortest path from v_i to v_j.



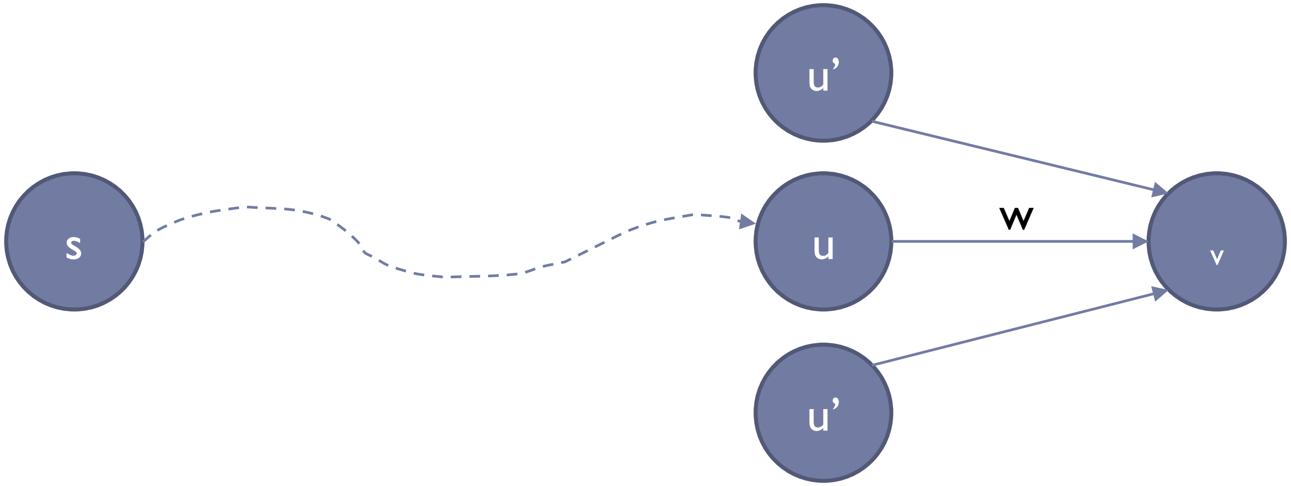
Corollary

- 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)

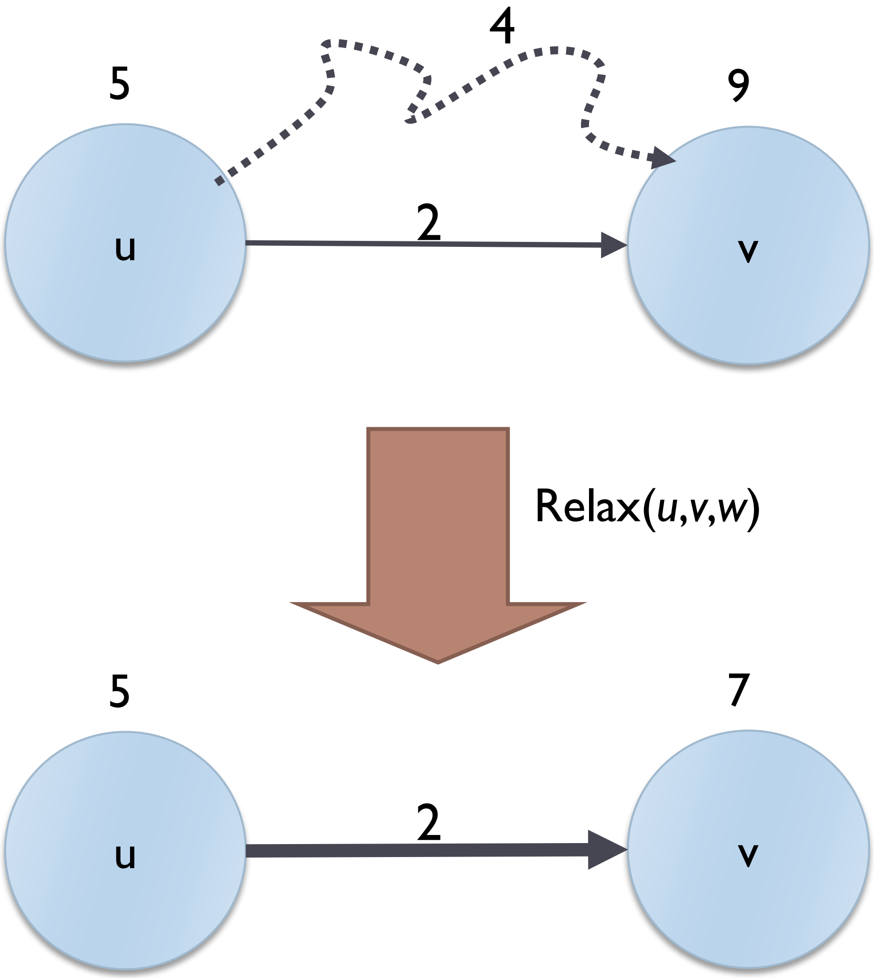
```
**for** all vertices $v \in V$
**do**
    $d[v] \leftarrow \infty$
    $\pi[v] \leftarrow NIL$
$d[s] \leftarrow 0$
```

Relaxation

- We consider an edge (*u*,*v*) with weight *w*
- Relax(*u*, *v*, *w*)

```
if $d[v] > d[u]+w(u,v)$
then
    $d[v] \leftarrow d[u]+w(u,v)$
    $\pi[v] \leftarrow u$
```

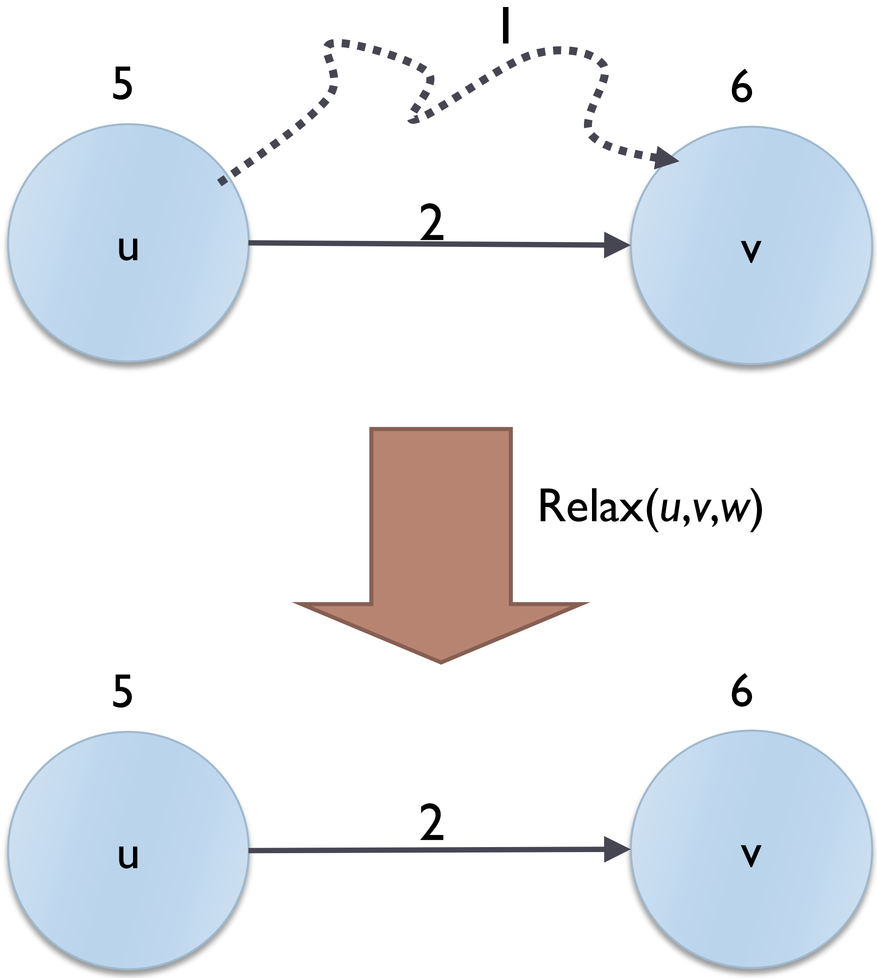
Example 1



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→$\mathbb{R}$.
- 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 $w: E \rightarrow \mathbb{R}$ 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 $G\pi$ is a rooted tree, with *s* as the root.
  - Any relaxation we may apply to the graph does not invalidate this property.

Lemma

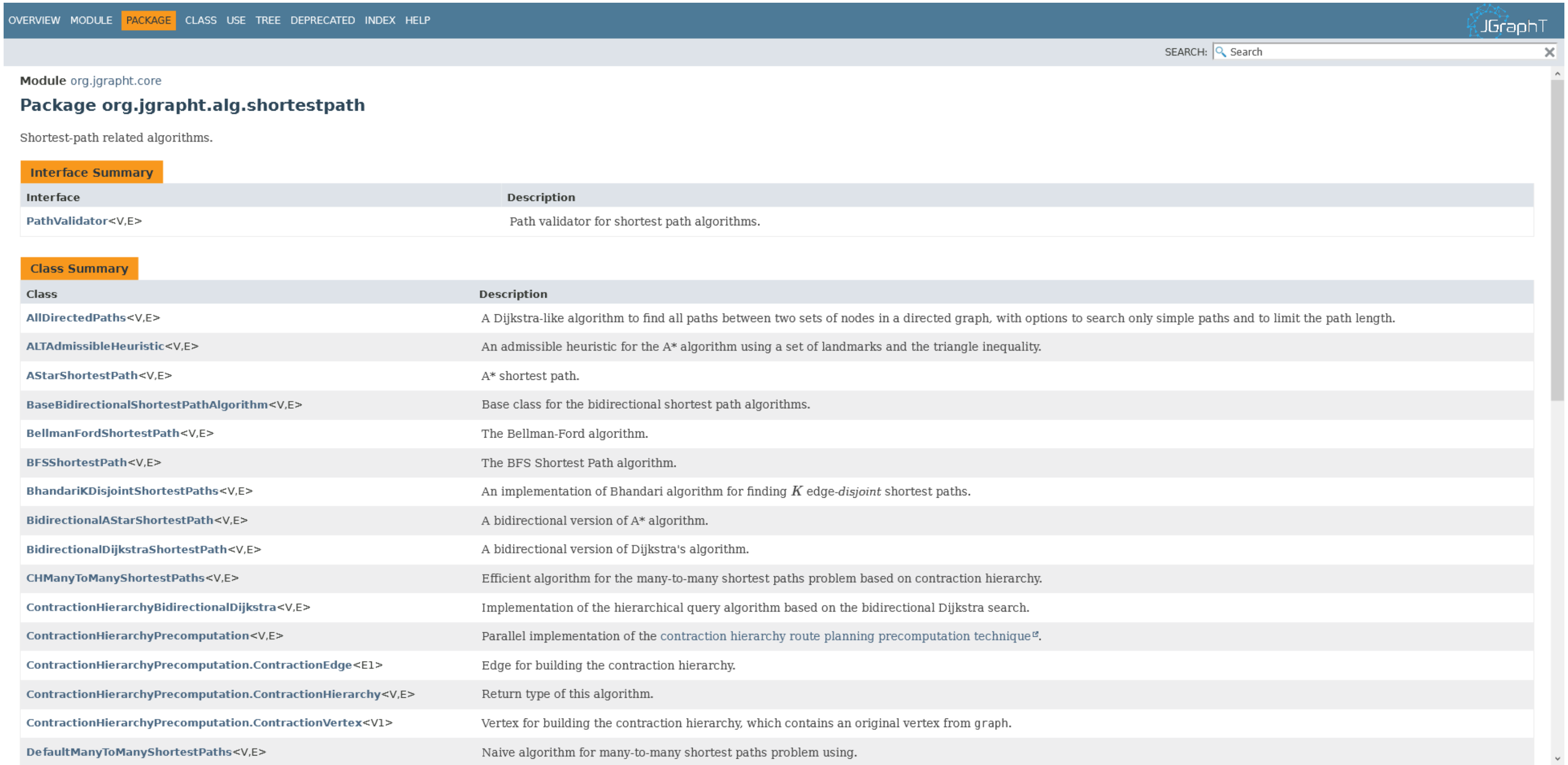
- Given the previous definitions.
- Apply any possible sequence of relaxation operations
- Therefore, for each vertex v
  - $d[v] \geq \delta(s,v)$
- Additionally, if $d[v] = \delta(s,v)$, then the value of $d[v]$ will not change anymore due to relaxation operations.

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`



Module org.jgrapht.core

Package org.jgrapht.alg.shortestpath

Shortest-path related algorithms.

Interface Summary

Interface	Description
PathValidator<V,E>	Path validator for shortest path algorithms.

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.
BaseBidirectionalShortestPathAlgorithm<V,E>	Base class for the bidirectional shortest path algorithms.
BellmanFordShortestPath<V,E>	The Bellman-Ford algorithm.
BFSShortestPath<V,E>	The BFS Shortest Path algorithm.
BhandariKDisjointShortestPaths<V,E>	An implementation of Bhandari algorithm for finding K edge-*disjoint* shortest paths.
BidirectionalAStarShortestPath<V,E>	A bidirectional version of A* algorithm.
BidirectionalDijkstraShortestPath<V,E>	A bidirectional version of Dijkstra's algorithm.
CHManyToManyShortestPaths<V,E>	Efficient algorithm for the many-to-many shortest paths problem based on contraction hierarchy.
ContractionHierarchyBidirectionalDijkstra<V,E>	Implementation of the hierarchical query algorithm based on the bidirectional Dijkstra search.
ContractionHierarchyPrecomputation<V,E>	Parallel implementation of the contraction hierarchy route planning precomputation technique.
ContractionHierarchyPrecomputation.ContractionEdge<E1>	Edge for building the contraction hierarchy.
ContractionHierarchyPrecomputation.ContractionHierarchy<V,E>	Return type of this algorithm.
ContractionHierarchyPrecomputation.ContractionVertex<V1>	Vertex for building the contraction hierarchy, which contains an original vertex from `graph`.
DefaultManyToManyShortestPaths<V,E>	Naive algorithm for many-to-many shortest paths problem using.

…and many more

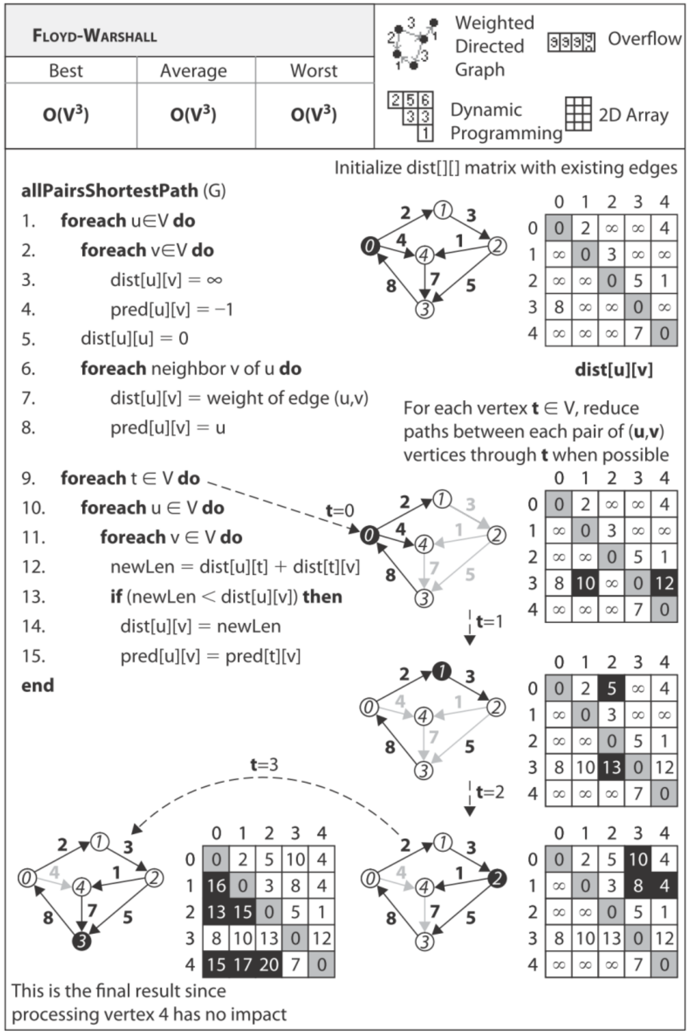
https://jgrapht.org/javadoc/org.jgrapht.core/org/jgrapht/alg/shortestpath/package-summary.html

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

Best	Average	Worst
$O(V^3)$	$O(V^3)$	$O(V^3)$

```
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

    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]
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** ∈ V, reduce paths between each pair of (**u**,**v**) vertices through **t** when possible

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

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

t=2

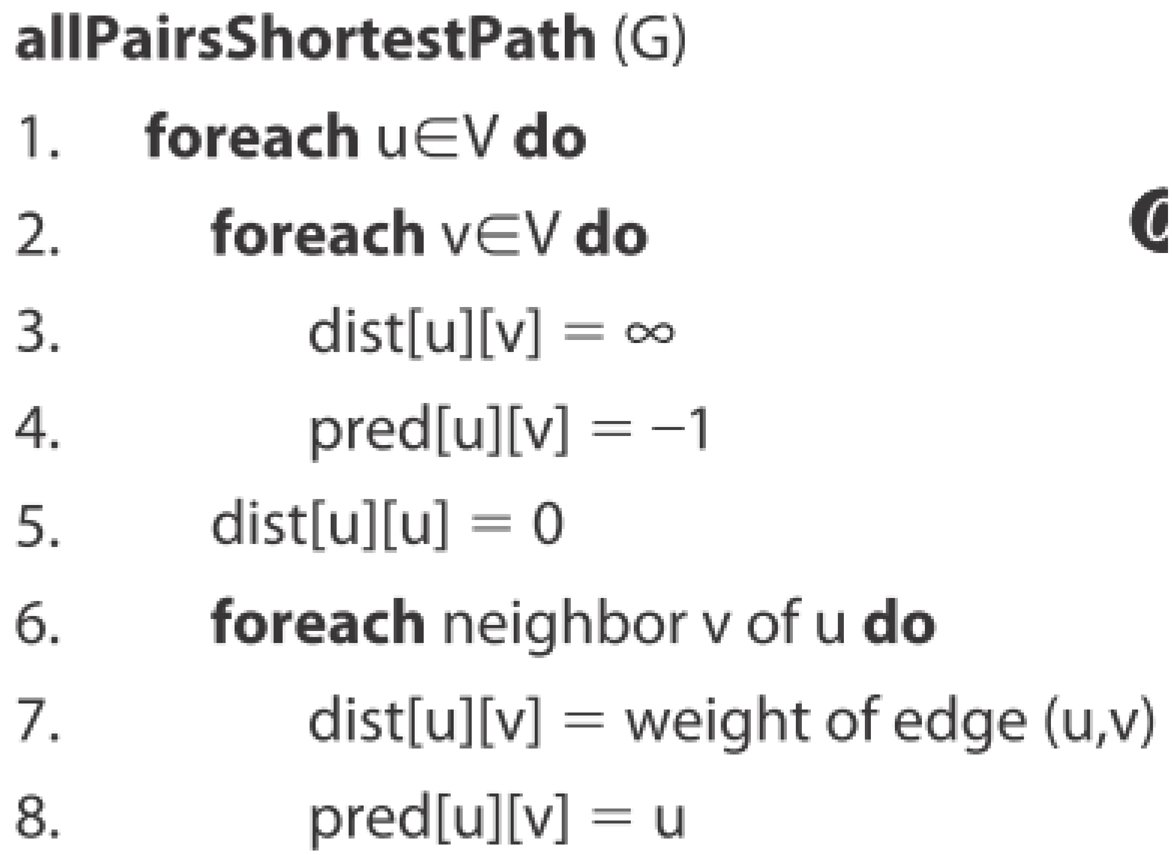
	0	1	2	3	4
0	0	2	5	10	4
1	∞	0	3	8	4
2	∞	∞	0	5	1
3	8	10	13	0	12
4	∞	∞	∞	7	0

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

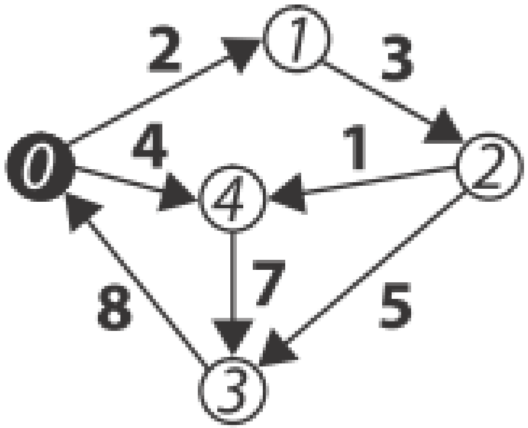
This is the final result since processing vertex 4 has no impact

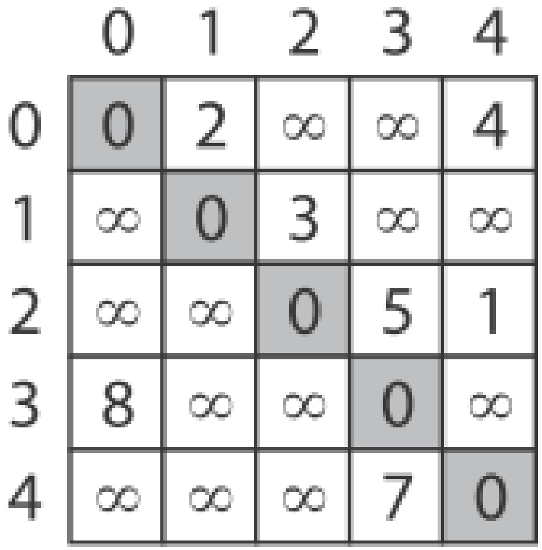
Floyd-Warshall: initialization



```
allPairsShortestPath (G)
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

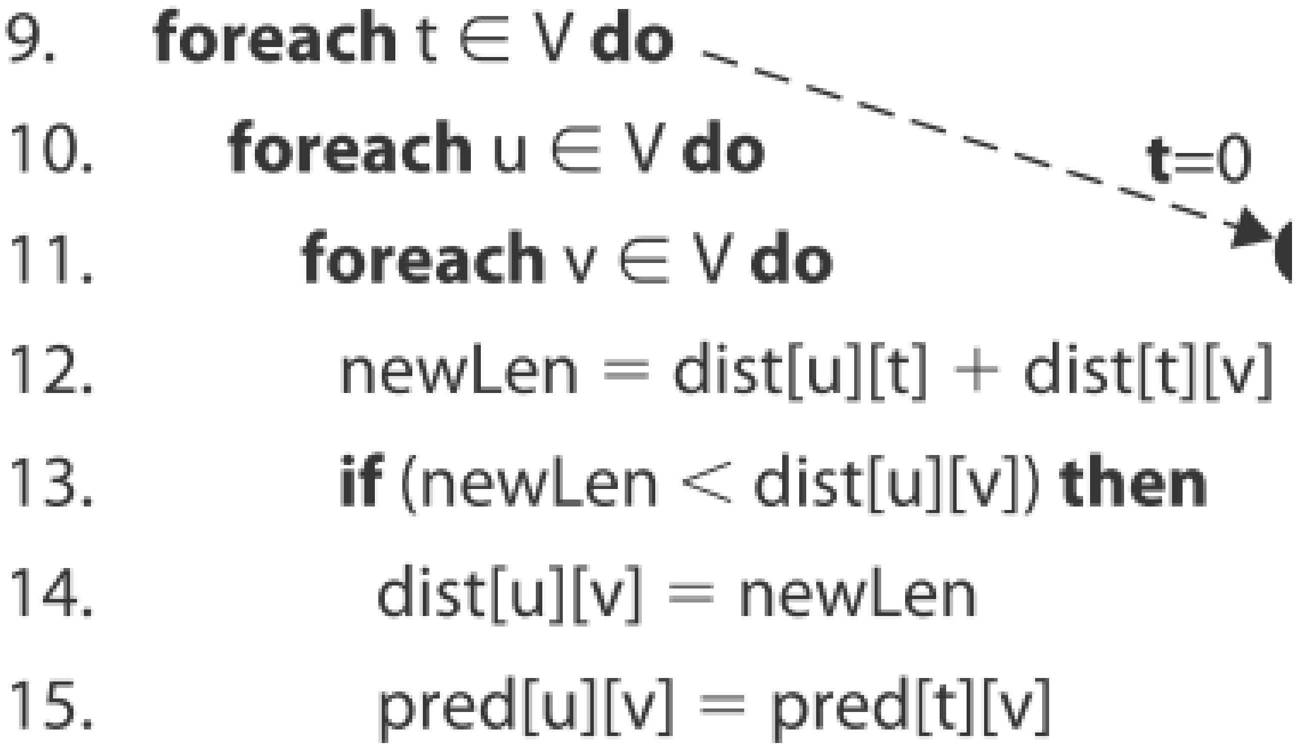




	0	1	2	3	4
0	0	2	∞	∞	4
1	∞	0	3	∞	∞
2	∞	∞	0	5	1
3	8	∞	∞	0	∞
4	∞	∞	∞	7	0

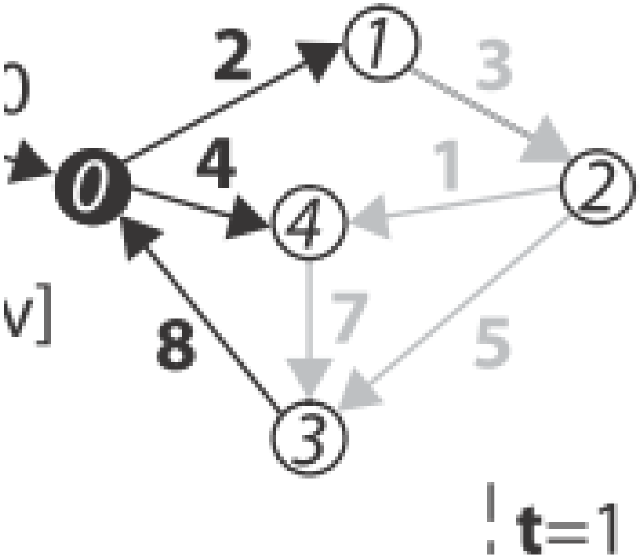
dist[u][v]

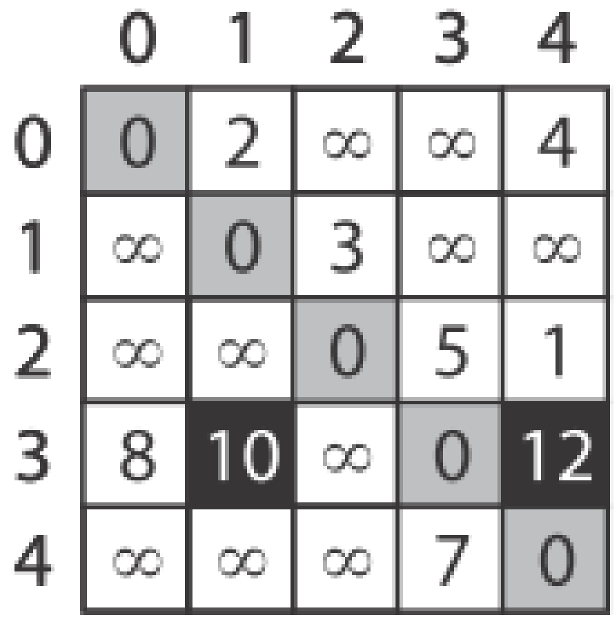
Floyd-Warshall: relaxation



```
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]
```

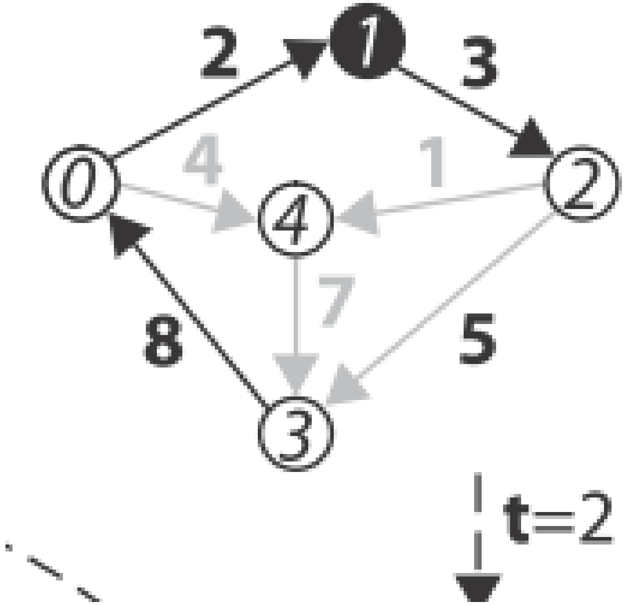
Example, after step t=0

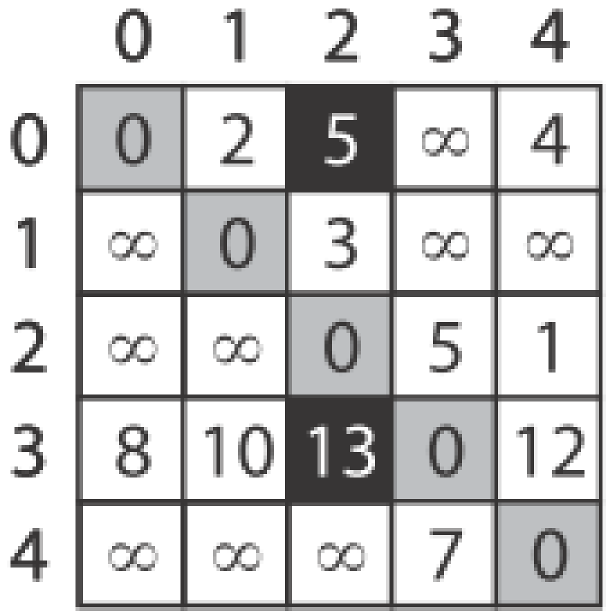




	0	1	2	3	4
0	0	2	∞	∞	4
1	∞	0	3	∞	∞
2	∞	∞	0	5	1
3	8	10	∞	0	12
4	∞	∞	∞	7	0

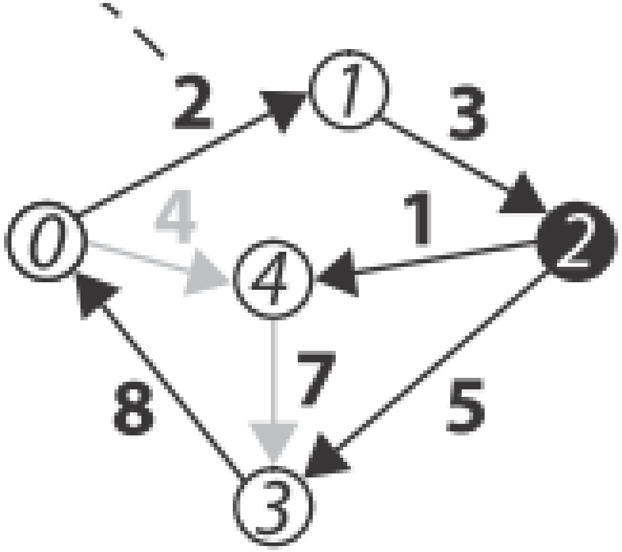
Example, after step t=1

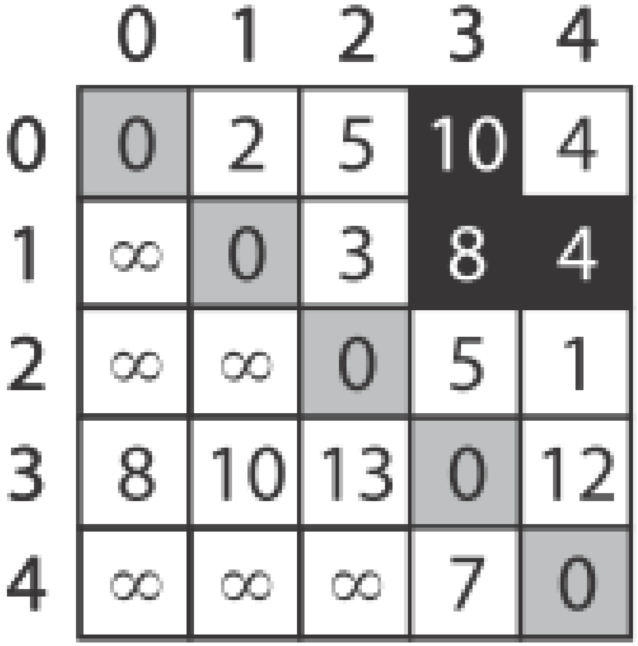




	0	1	2	3	4
0	0	2	5	∞	4
1	∞	0	3	∞	∞
2	∞	∞	0	5	1
3	8	10	13	0	12
4	∞	∞	∞	7	0

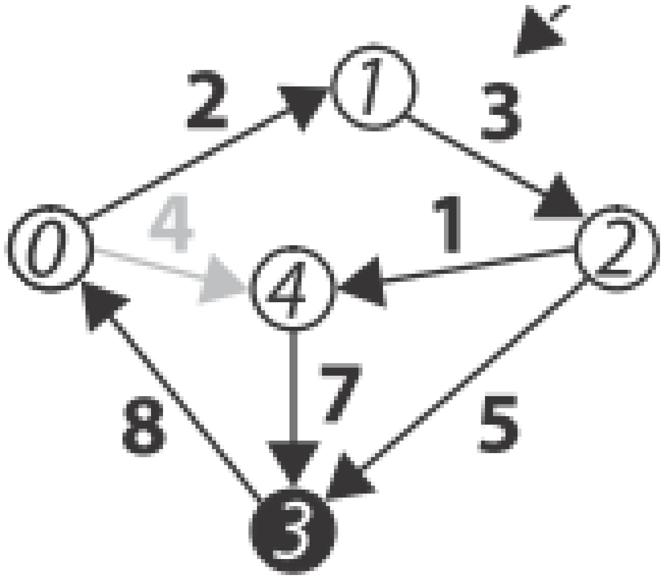
Example, after step t=2

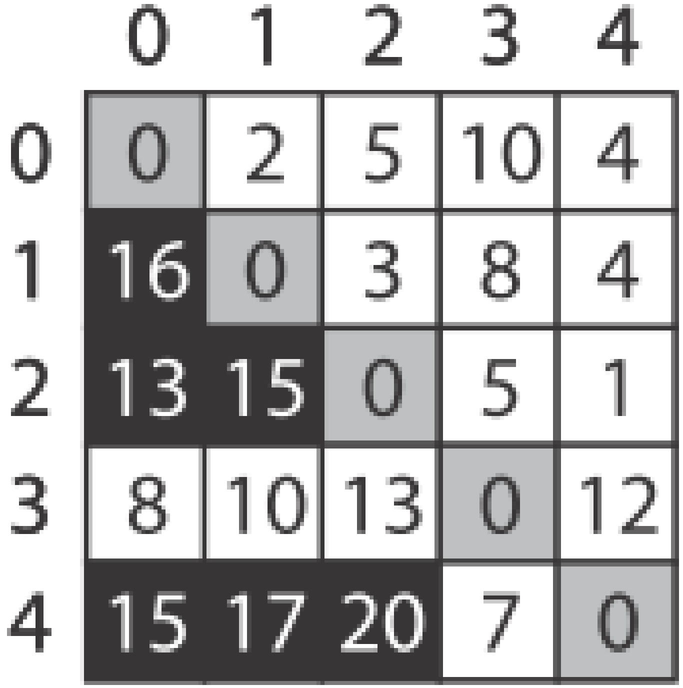




	0	1	2	3	4
0	0	2	5	10	4
1	∞	0	3	8	4
2	∞	∞	0	5	1
3	8	10	13	0	12
4	∞	∞	∞	7	0

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.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·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

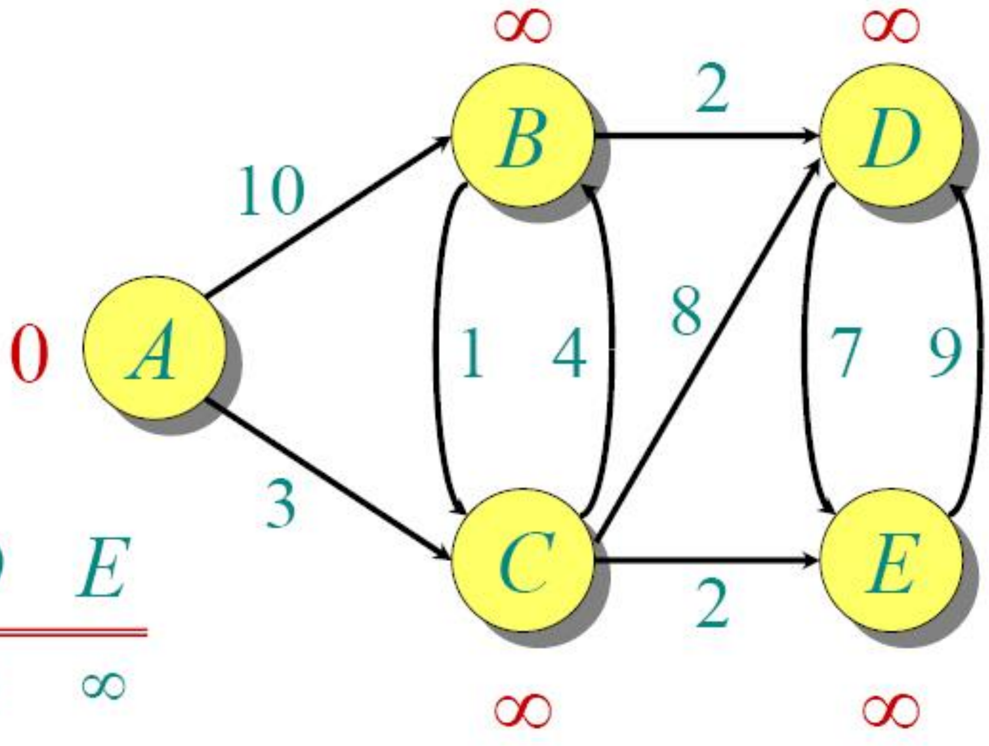
… but guarantees the optimum!

Dijkstra's algorithm

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

Dijkstra Animated Example

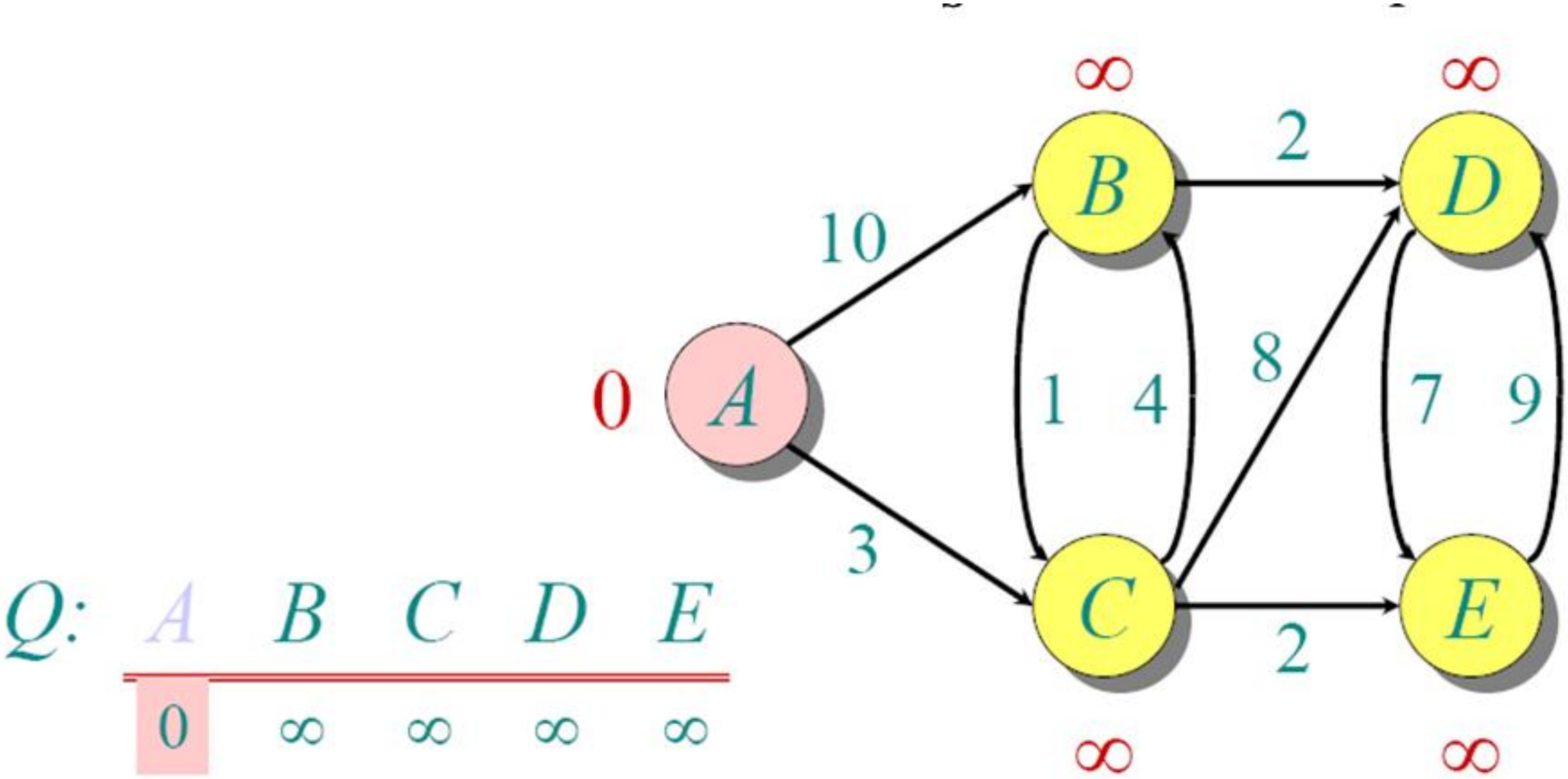
Initialize:



Q:	*A*	*B*	*C*	*D*	*E*
	0	∞	∞	∞	∞

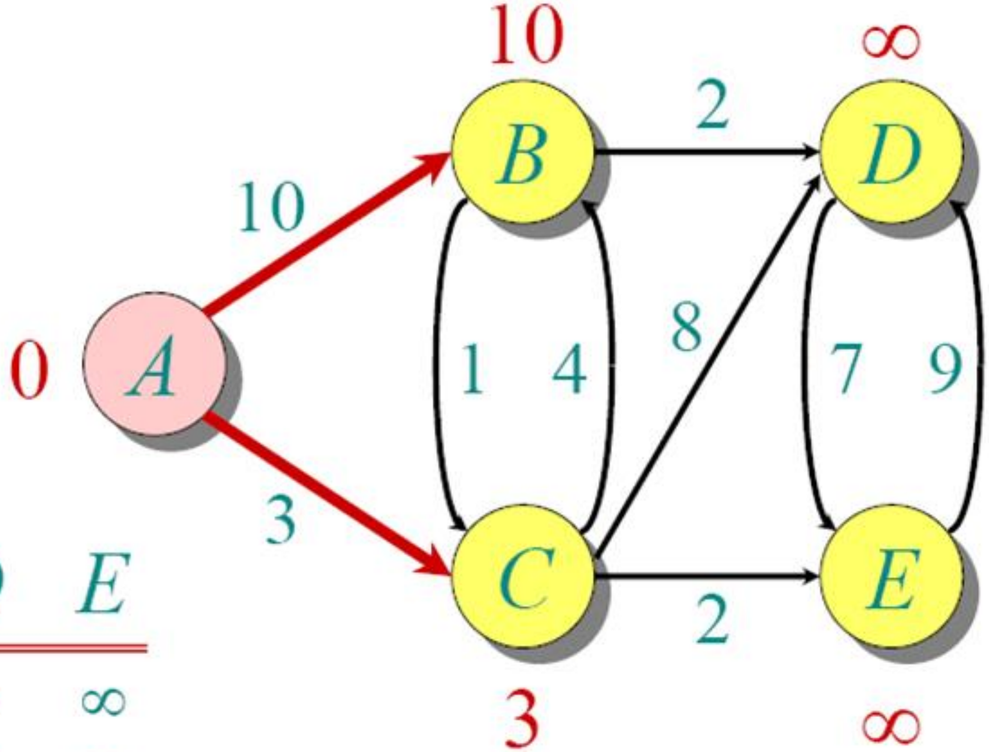
S: {}

Dijkstra Animated Example



Q:	A	B	C	D	E
	0	∞	∞	∞	∞

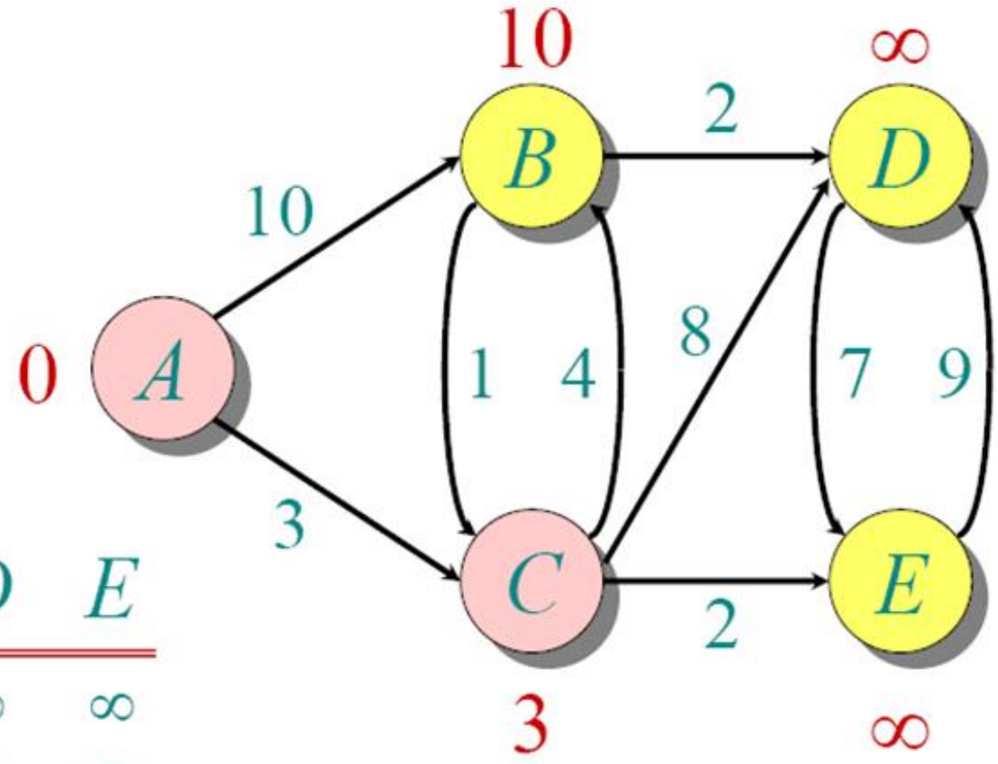
Dijkstra Animated Example



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞

S: { A }

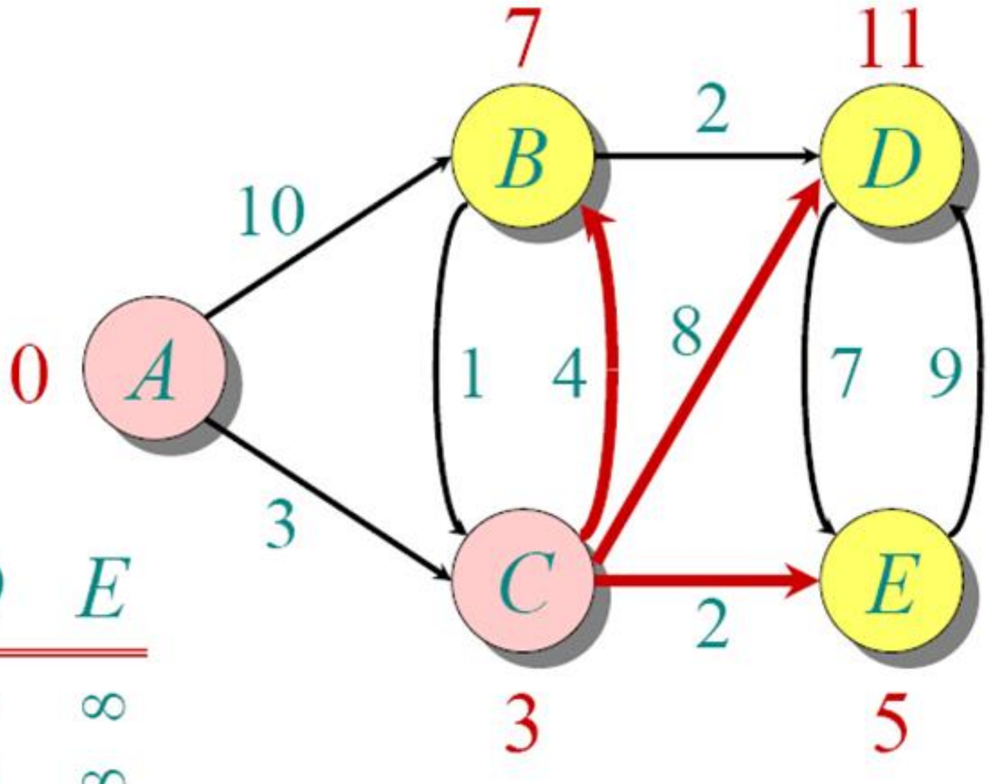
Dijkstra Animated Example



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞

S: { A, C }

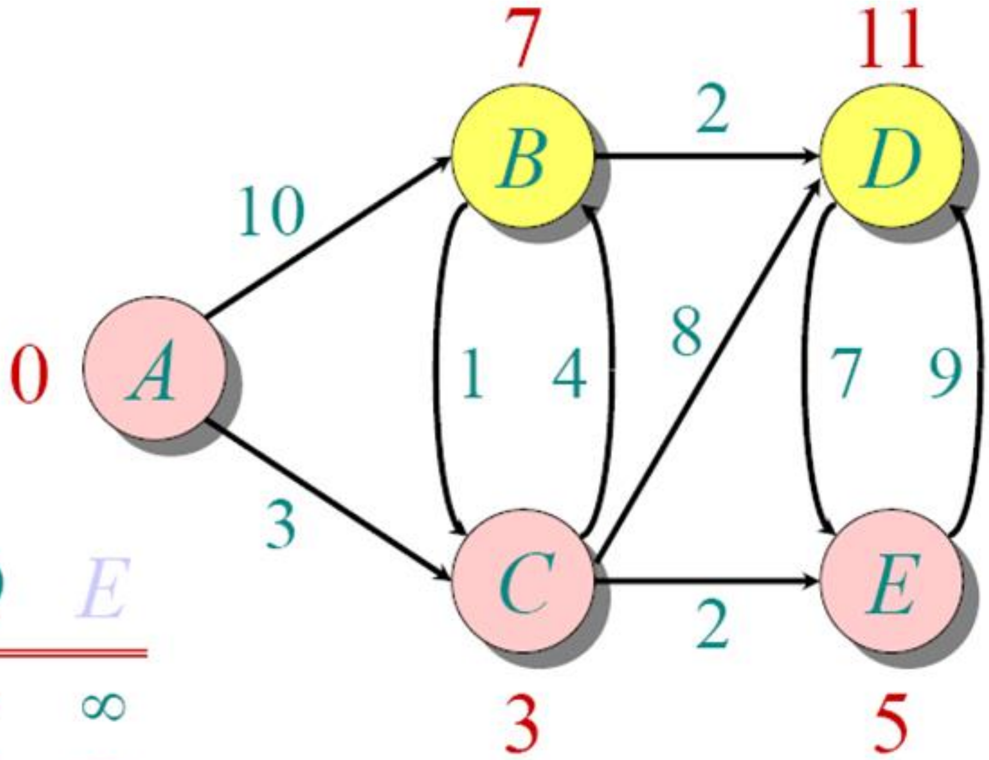
Dijkstra Animated Example



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

S: { A, C }

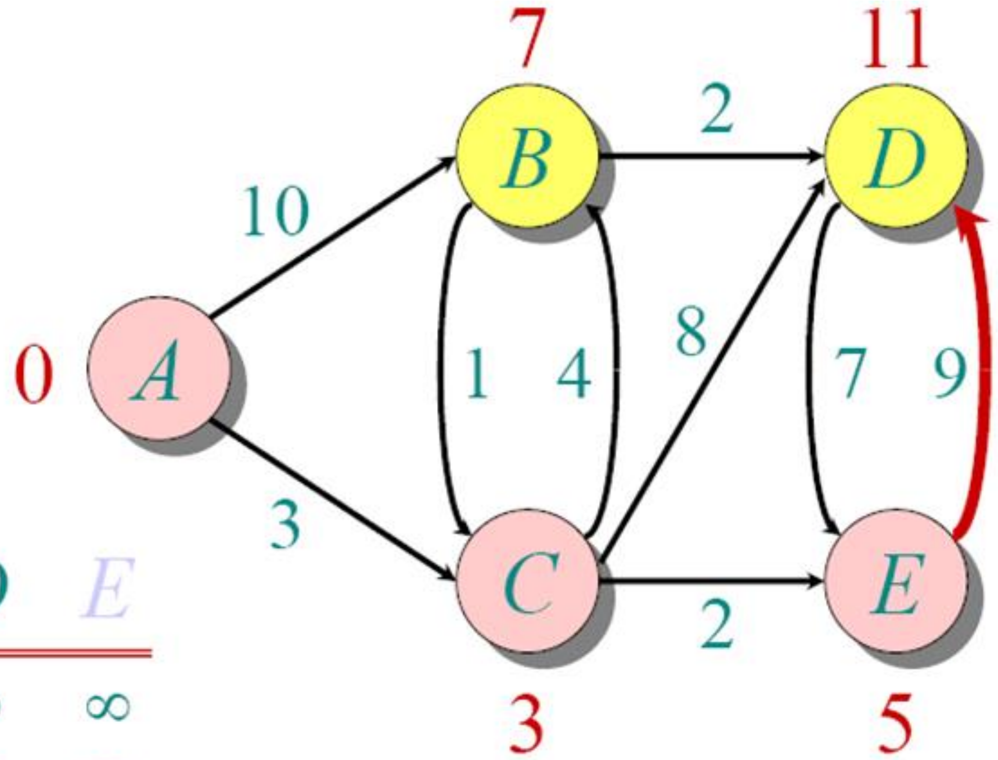
Dijkstra Animated Example



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

S: { A, C, E }

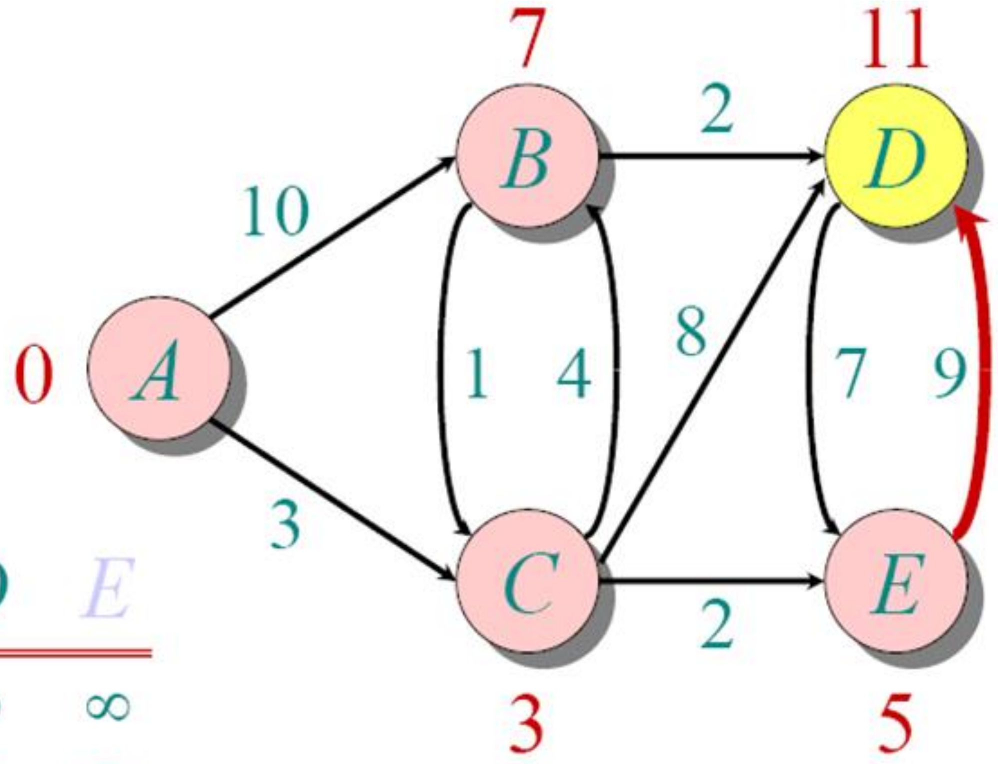
Dijkstra Animated Example



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

S: { A, C, E }

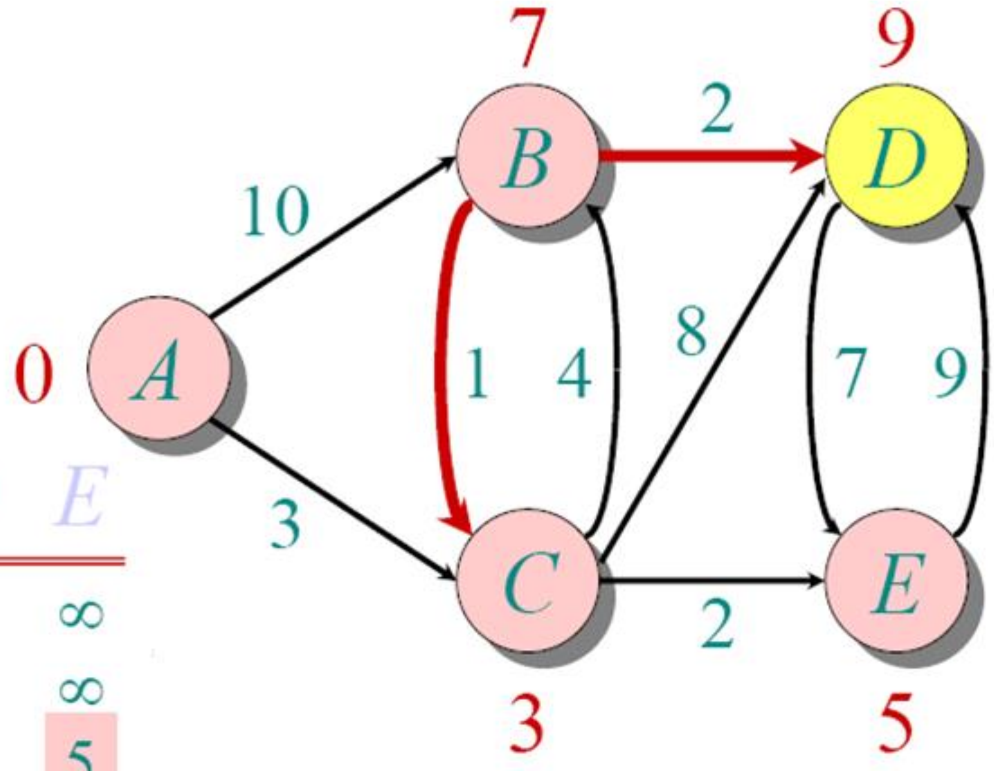
Dijkstra Animated Example



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

S: { A, C, E, B }

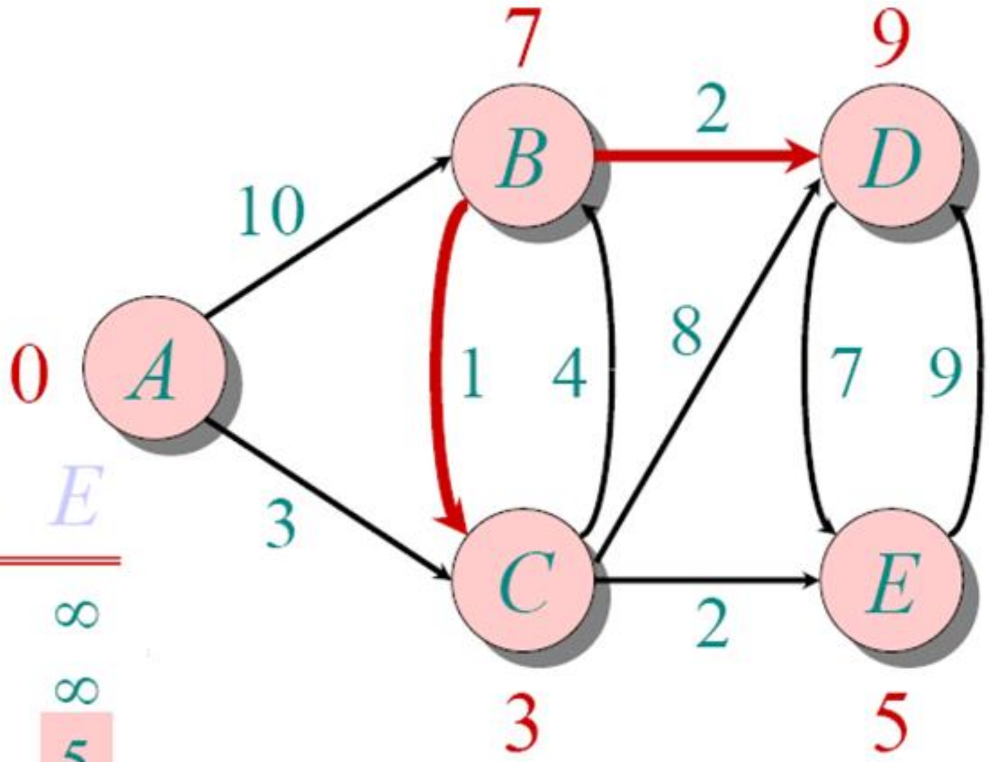
Dijkstra Animated Example



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* }

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 : $O(V \cdot E)$

Implementation

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

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

Cycles: Definitions

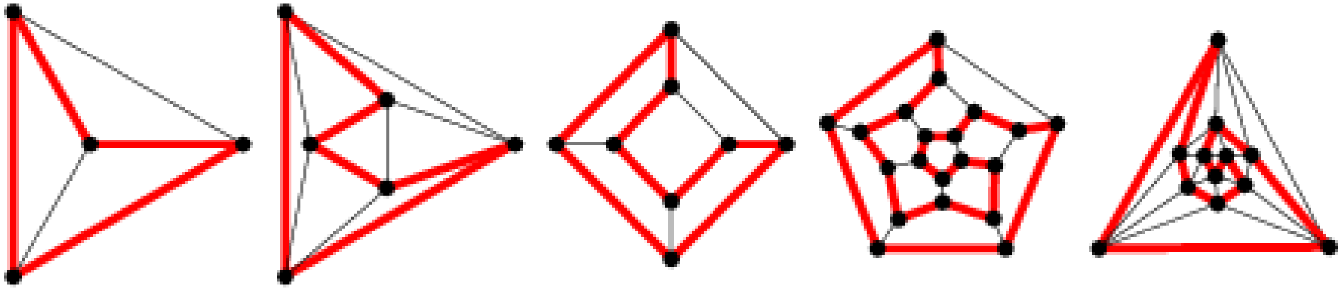
Graphs: Cycles

Cycle

- A **cycle** of a graph, sometimes also called a circuit, is a subset of the edge set of that forms a path such that the first node of the path corresponds to the last.

Hamiltonian cycle

- A cycle that uses each graph vertex of a graph exactly once is called a Hamiltonian cycle.



Hamiltonian path

- A Hamiltonian path, also called a Hamilton path, is a path between two vertices of a graph that visits each vertex exactly once.
  - N.B. does not need to return to the starting point

Eulerian Path and Cycle

- An **Eulerian path**, also called an Euler chain, Euler trail, Euler walk, or "Eulerian" version of any of these variants, is a walk on the graph edges of a graph which **uses each graph edge** in the original graph **exactly once**.
- An **Eulerian cycle**, also called an Eulerian circuit, Euler circuit, Eulerian tour, or Euler tour, is a trail which starts and ends at the **same** graph vertex.

Theorem

- A connected graph has an Eulerian **cycle** if and only if it **all vertices have even degree**.
- A connected graph has an Eulerian **path** if and only if it has **at most two graph vertices of odd degree**.
  - …easy to check!

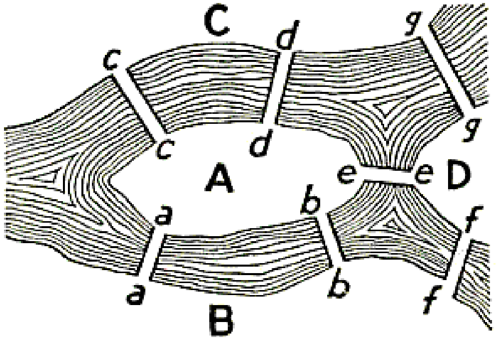
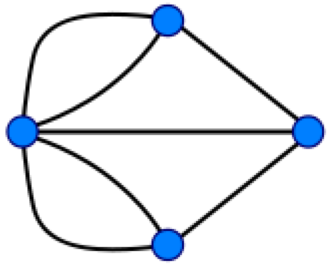


FIGURE 98. *Geographic Map: The Königsberg Bridges.*



Königsberg Bridges

Weighted vs. Unweighted

- Classical versions defined on Unweighted graphs
- Unweighted:
  - Does such a cycle exist?
  - If yes, find at least one
    - Optionally, find all of them
- Weighted
  - Does such a cycle exist?
    - Often, the graph is complete ☺
  - If yes, find at least one
  - If yes, find **the best one** (with **minimum** weight)

Algorithms

Graphs: Cycles

Eulerian cycles: Hierholzer's algorithm (1)

- Choose **any** starting vertex *v*, and **follow a trail** of edges from that vertex until returning to *v*.
  - It is **not** possible to get stuck at any vertex other than *v*, because the even degree of all vertices ensures that, when the trail enters another vertex *w* there must be an unused edge leaving *w*.
  - The tour formed in this way is a **closed** tour, but may **not** cover all the vertices and edges of the initial graph.

Eulerian cycles: Hierholzer's algorithm (2)

- As long as there exists a vertex *v* that belongs to the current tour but that has adjacent edges not part of the tour, **start another trail** from *v*, following **unused** edges until returning to *v*, **and join** the tour formed in this way to the previous tour.

Finding Eulerian circuits

Hierholzer's Algorithm

Given: an Eulerian graph G

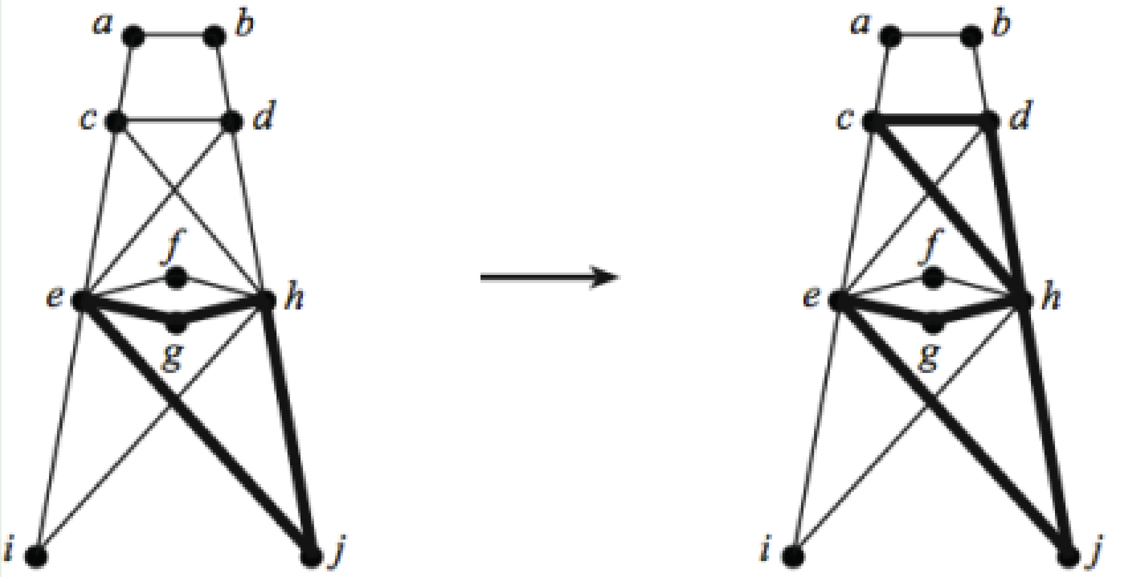
Find an Eulerian circuit of G.

1. Identify a circuit in G and call it R_1. Mark the edges of R_1. Let $i = 1$.
2. If R_i contains all edges of G, then stop (since R_i is an Eulerian circuit).
3. If R_i does not contain all edges of G, then let v_i be a node on R_i that is incident with an unmarked edge, e_i.
4. Build a circuit, Q_i, starting at node v_i and using edge e_i. Mark the edges of Q_i.
5. Create a new circuit, R_{i+1}, by patching the circuit Q_i into R_i at v_i.
6. Increment i by 1, and go to step (2).

Finding Eulerian circuits

Hierholzer's Algorithm

Example



R_1: e, g, h, j, e

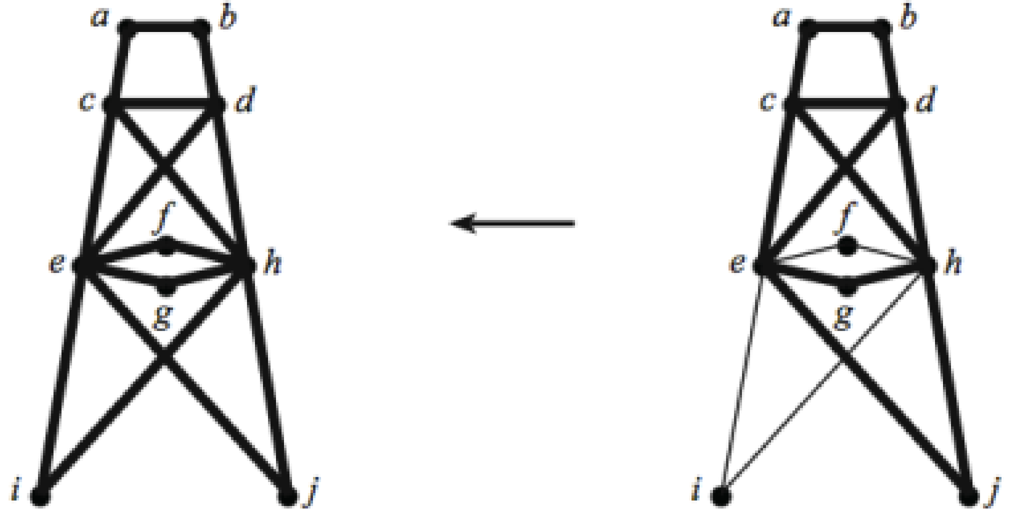
Q_1: h, d, c, h

R_2: $e, g, \mathbf{h}, \mathbf{d}, \mathbf{c}, \mathbf{h}, j, e$

Q_2: d, b, a, c, e, d

Finding Eulerian circuits
Hierholzer's Algorithm

Example (continued)



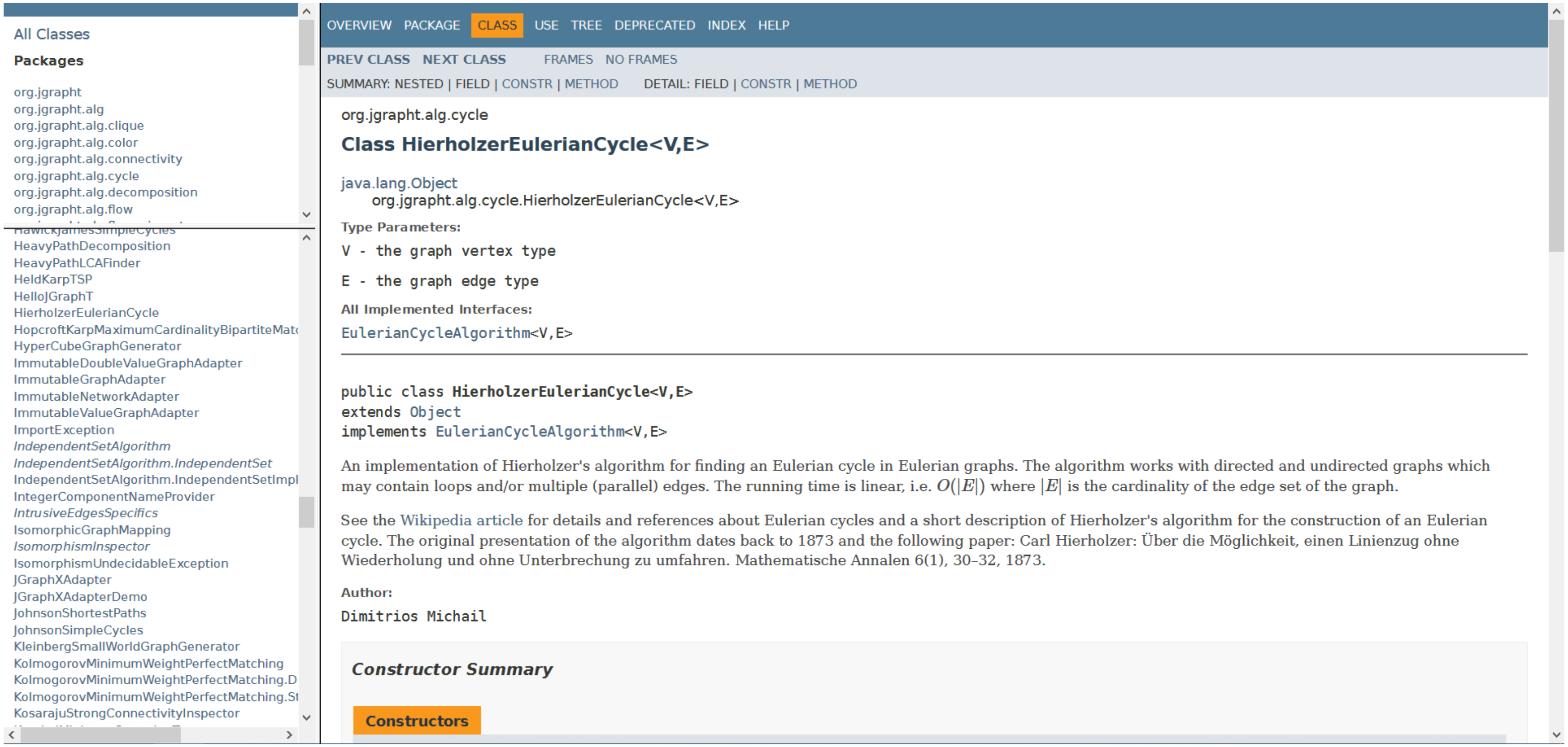
R_4: e, g, **h**, **f**, **e**, **i**, **h**, d, b, a,
c, e, d, c, h, j, e

R_3: e, g, h, **d**, **b**, **a**, **c**, **e**, **d**, c, h, j, e

Q_3: h, f, e, i, h

Eulerian Circuits in JGraphT

org.jgrapht.alg.cycle



org.jgrapht.alg.cycle

Class HierholzerEulerianCycle<V,E>

java.lang.Object
    org.jgrapht.alg.cycle.HierholzerEulerianCycle<V,E>

Type Parameters:

`V - the graph vertex type`

`E - the graph edge type`

All Implemented Interfaces:

`EulerianCycleAlgorithm<V,E>`

```
public class HierholzerEulerianCycle<V,E>
extends Object
implements EulerianCycleAlgorithm<V,E>
```

An implementation of Hierholzer's algorithm for finding an Eulerian cycle in Eulerian graphs. The algorithm works with directed and undirected graphs which may contain loops and/or multiple (parallel) edges. The running time is linear, i.e. $O(|E|)$ where $|E|$ is the cardinality of the edge set of the graph.

See the Wikipedia article for details and references about Eulerian cycles and a short description of Hierholzer's algorithm for the construction of an Eulerian cycle. The original presentation of the algorithm dates back to 1873 and the following paper: Carl Hierholzer: Über die Möglichkeit, einen Linienzug ohne Wiederholung und ohne Unterbrechung zu umfahren. Mathematische Annalen 6(1), 30–32, 1873.

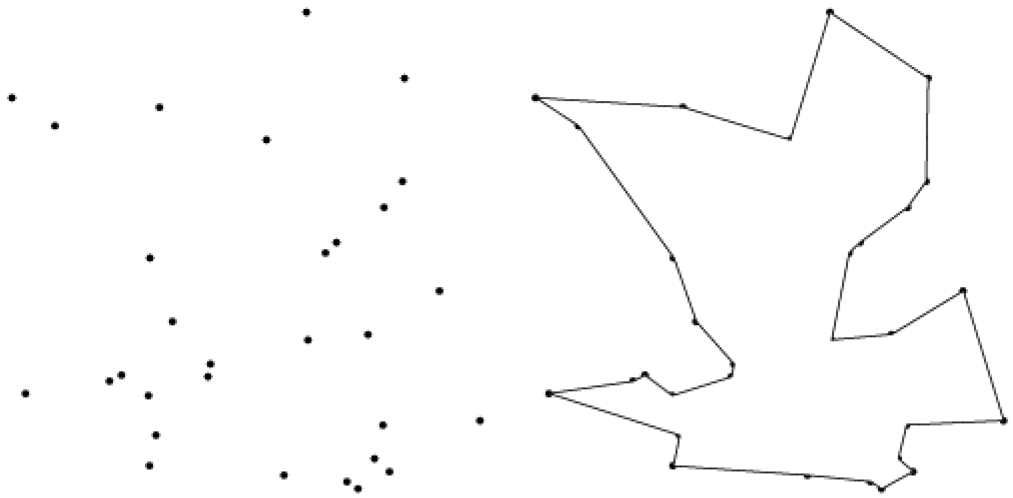
Author:

`Dimitrios Michail`

Constructor Summary

Hamiltonian Cycles

- There are theorems to identify **whether** a graph is Hamiltonian (i.e., whether it contains at least one Hamiltonian Cycle)
- **Finding** such a cycle has **no** known efficient solution, in the general case
- Example: the **Traveling Salesman Problem** (TSP)



The Traveling Salesman Problem (TSP)

Weighted or unweighted

Given a collection of cities connected by roads

Find the shortest route that visits each city exactly once.

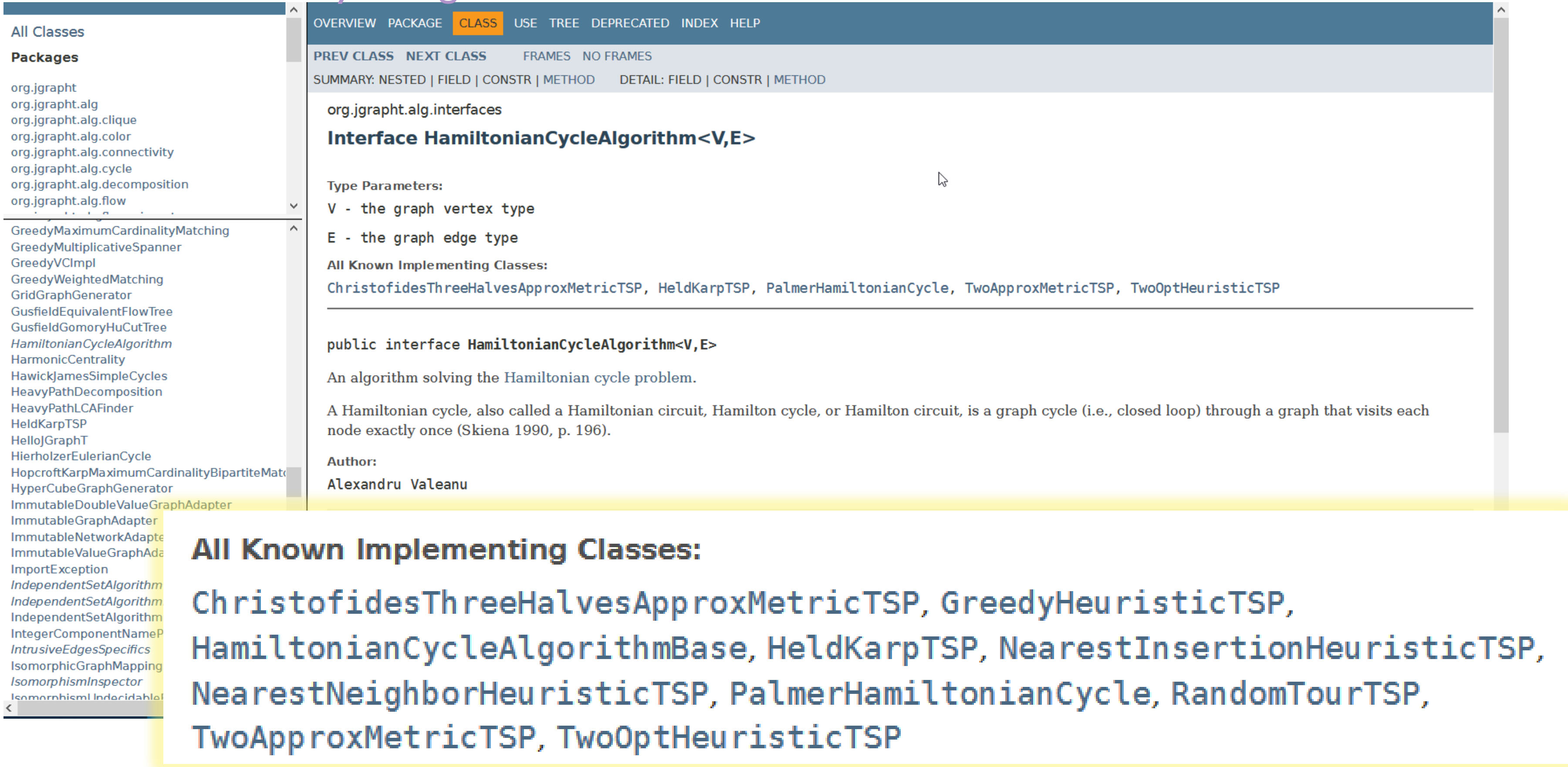
About TSP

- Most notorious NP-complete problem.
- Typically, it is solved with a backtracking algorithm:
  - The best tour found to date is saved.
  - The search backtracks unless the partial solution is cheaper than the cost of the best tour.

Hamiltonian Cycles in JGraphT

https://jgrapht.org/javadoc/org/jgrapht/alg/interfaces/HamiltonianCycleAlgorithm.html

org.jgrapht.alg.**interfaces**



org.jgrapht.alg.interfaces

Interface HamiltonianCycleAlgorithm<V,E>

Type Parameters:

`V` - the graph vertex type

`E` - the graph edge type

All Known Implementing Classes:

`ChristofidesThreeHalvesApproxMetricTSP`, `HeldKarpTSP`, `PalmerHamiltonianCycle`, `TwoApproxMetricTSP`, `TwoOptHeuristicTSP`

```
public interface HamiltonianCycleAlgorithm<V,E>
```

An algorithm solving the Hamiltonian cycle problem.

A Hamiltonian cycle, also called a Hamiltonian circuit, Hamilton cycle, or Hamilton circuit, is a graph cycle (i.e., closed loop) through a graph that visits each node exactly once (Skiena 1990, p. 196).

Author:

Alexandru Valeanu

All Known Implementing Classes:

`ChristofidesThreeHalvesApproxMetricTSP`, `GreedyHeuristicTSP`, `HamiltonianCycleAlgorithmBase`, `HeldKarpTSP`, `NearestInsertionHeuristicTSP`, `NearestNeighborHeuristicTSP`, `PalmerHamiltonianCycle`, `RandomTourTSP`, `TwoApproxMetricTSP`, `TwoOptHeuristicTSP`

Limitations…

- No exact solution (Approximate algorithms)
  - Class TwoApproxMetricTSP<V,E>
  - Class ChristofidesThreeHalvesApproxMetricTSP<V,E>
  - Class TwoOptHeuristicTSP<V,E>
- Or complete under extra conditions
  - Class PalmerHamiltonianCycle<V,E>
- Or complete but O(2^N)
  - Class HeldKarpTSP<V,E>

The Metric Traveling Salesman Problem

An approximation algorithm

ASSUMPTION: G is a metric graph.

1. Compute a minimum weight spanning tree T for G.
2. Perform a depth-first traversal of T starting from any node, and order the nodes of G as they were discovered in this traversal.

   $\Rightarrow$ a tour that is at most twice the optimal tour in G.

Class TwoApproxMetricTSP<V,E>

Resources

- http://mathworld.wolfram.com/
- http://en.wikipedia.org/wiki/Euler_cycle
- Mircea MARIN, Graph Theory and Combinatorics, Lectures 9 and 10, http://web.info.uvt.ro/~mmarin/

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 origina 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/