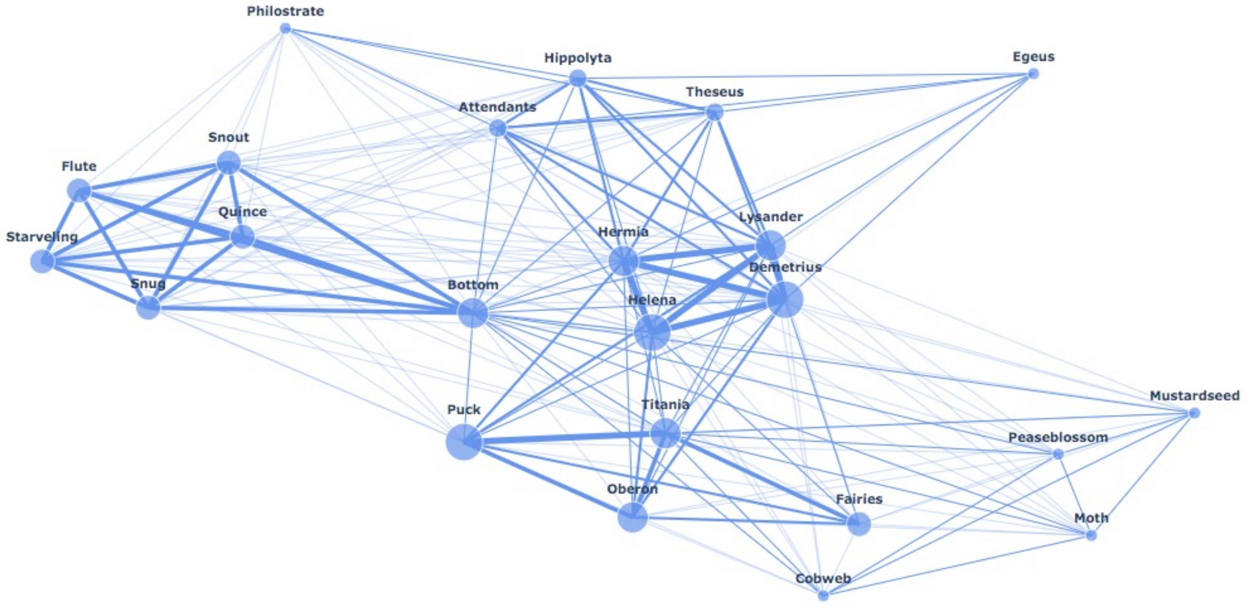
Paths in graphs

Shortest path and cycles

Fulvio Corno
Giuseppe Averta
Carlo Masone
Francesca Pistilli

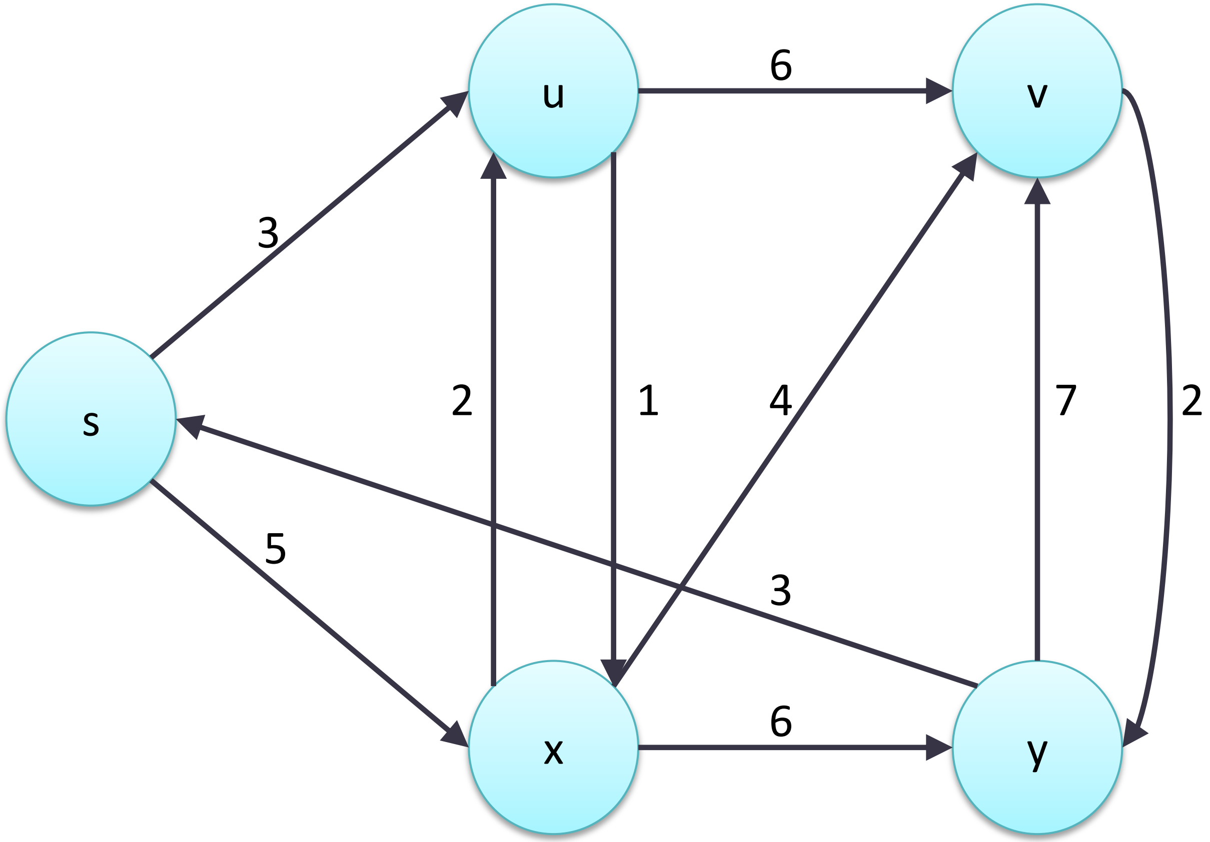


Searching for “optimal” paths between nodes

PATHS IN GRAPHS

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→$\mathbb{R}$
  - 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 d(u,v)
- If v is not reachable from u, then (by definition) d(u,v)=∞

Finding shortest paths

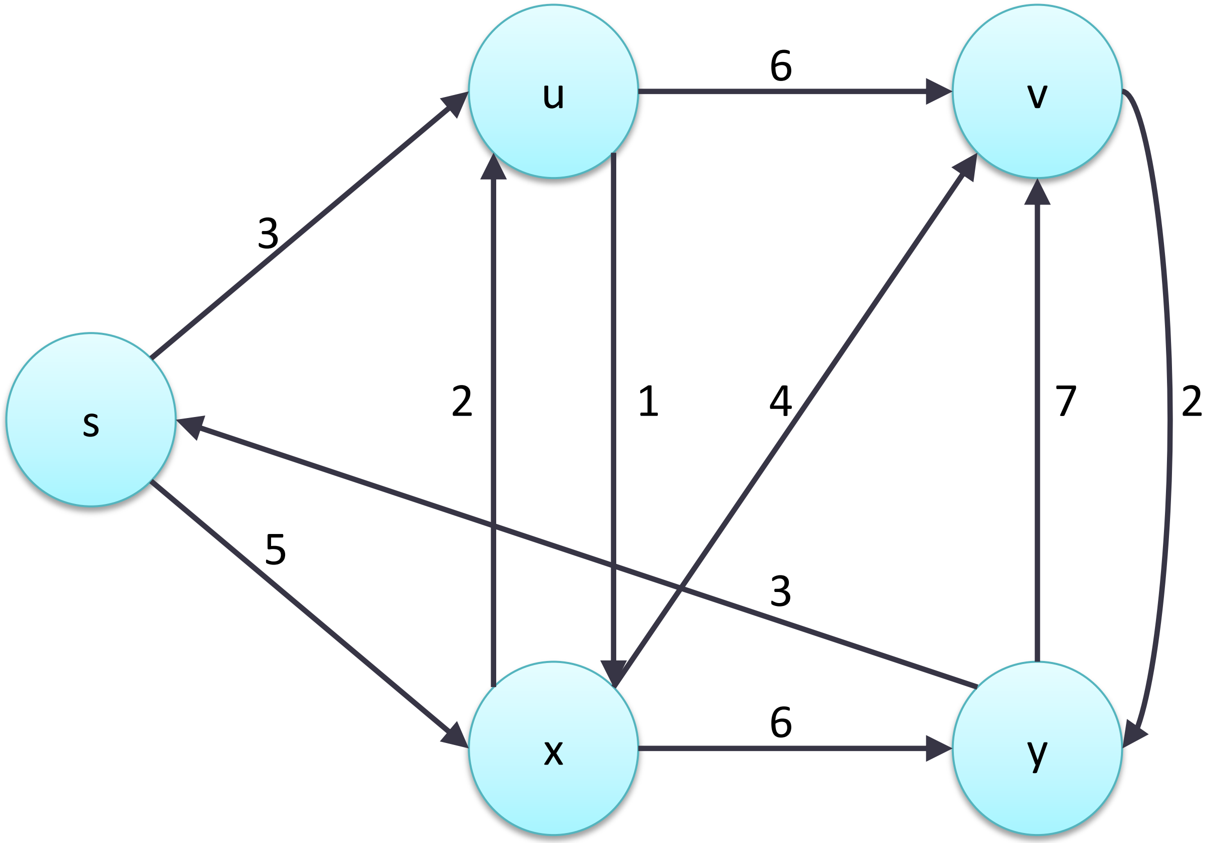
- Single-source shortest path (SS-SP)
  - Given u and v, find the shortest path between u and v
  - Given u, find the shortest path between u and any other vertex
- All-pairs shortest path (AP-SP)
  - Given a graph, find the shortest path between any pair of vertices

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

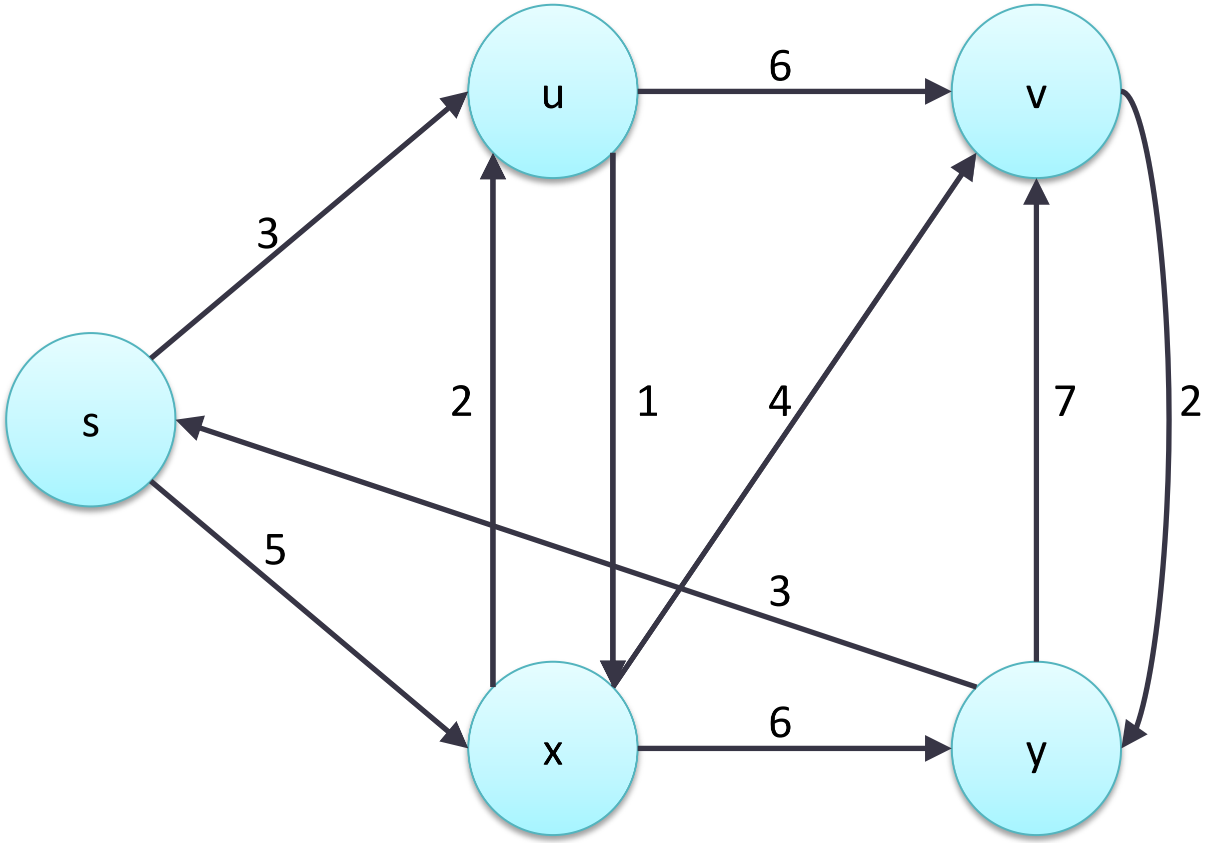
What is the shortest path between s and v ?



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 d(u,v) (double)
  - For each vertex v, the “preceding” vertex p(v) that allows to reach v in the shortest path (object)
    - For multigraphs, we need the preceding edge

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 p1 = shortest path between u and v1
  - Let p2 = shortest path between u and v2
  - Consider a vertex $w \in p1 \cap p2$
  - The value of $\pi(w)$ may be chosen in a unique way and still guarantees that both p1 and p2 are shortest

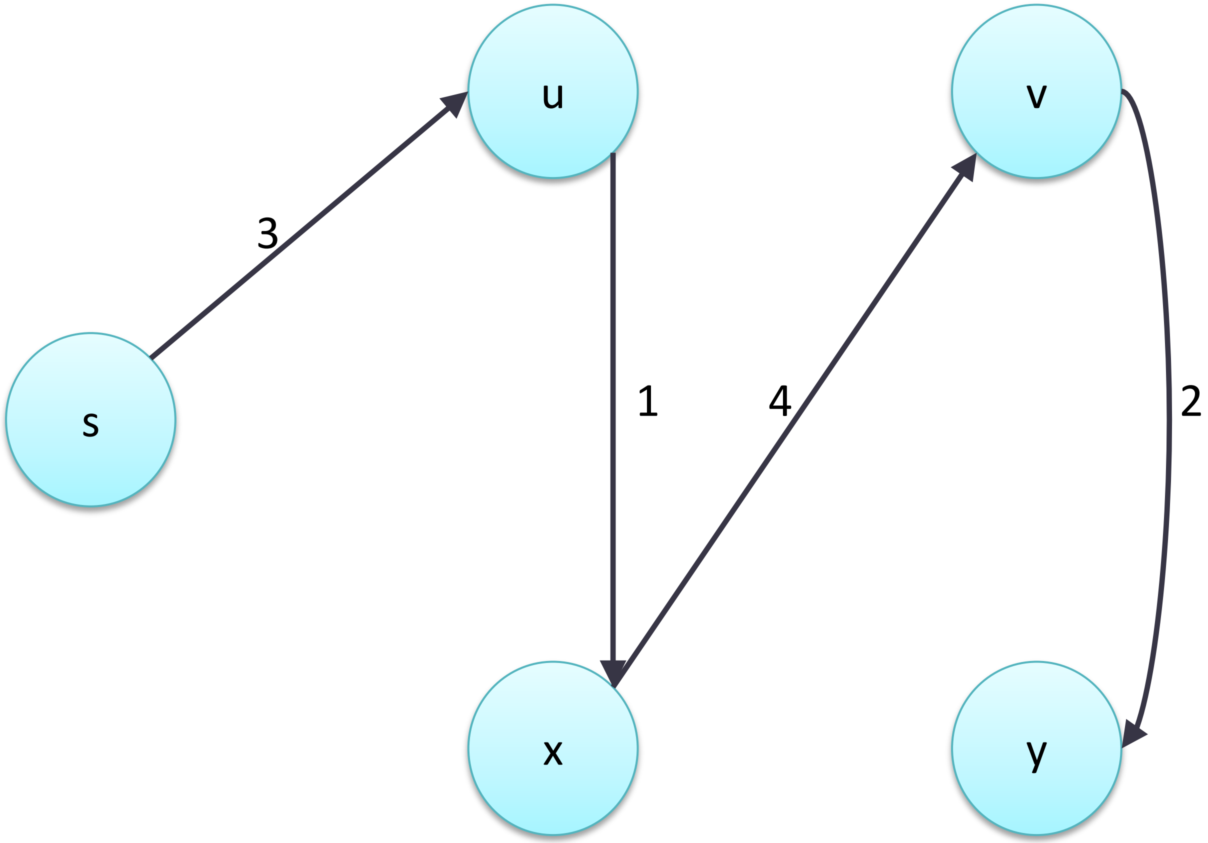
Shortest path graph

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

Shortest path tree

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

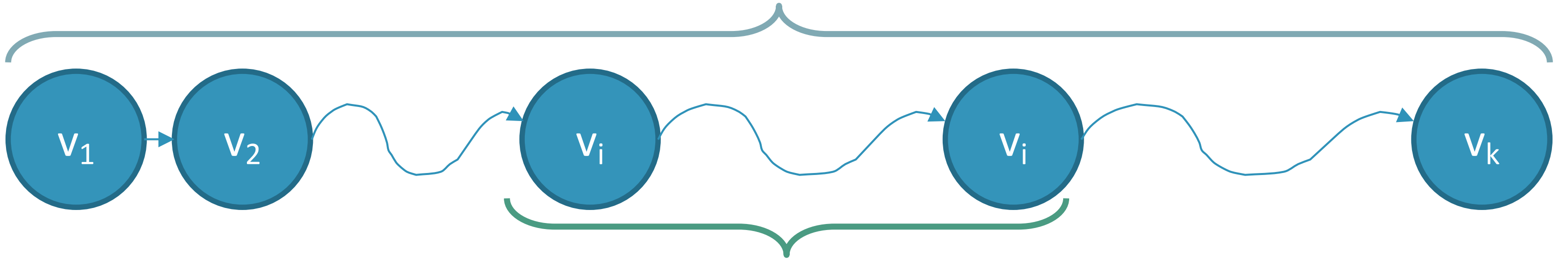
Special case

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

Lemma

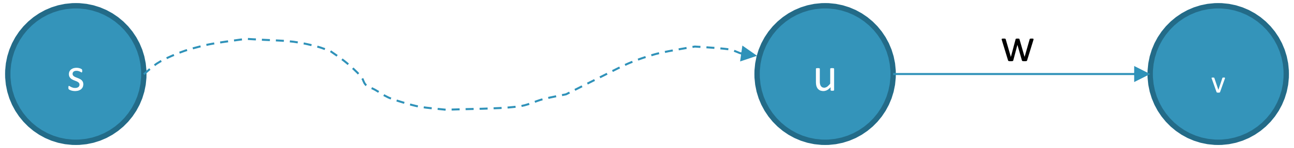
- Consider an ordered weighted graph G=(V,E), with weight function w: E→ℝ.
- Let p=<v1, v2, …, vk> a shortest path from vertex v1 to vertex vk.
- For all i,j such that 1≤i≤j≤k, let pij=<vi, vi+1, …, vj> be the sub-path of p, from vertex vi to vertex vj.

- Therefore, pij is a shortest path from vi to vj.



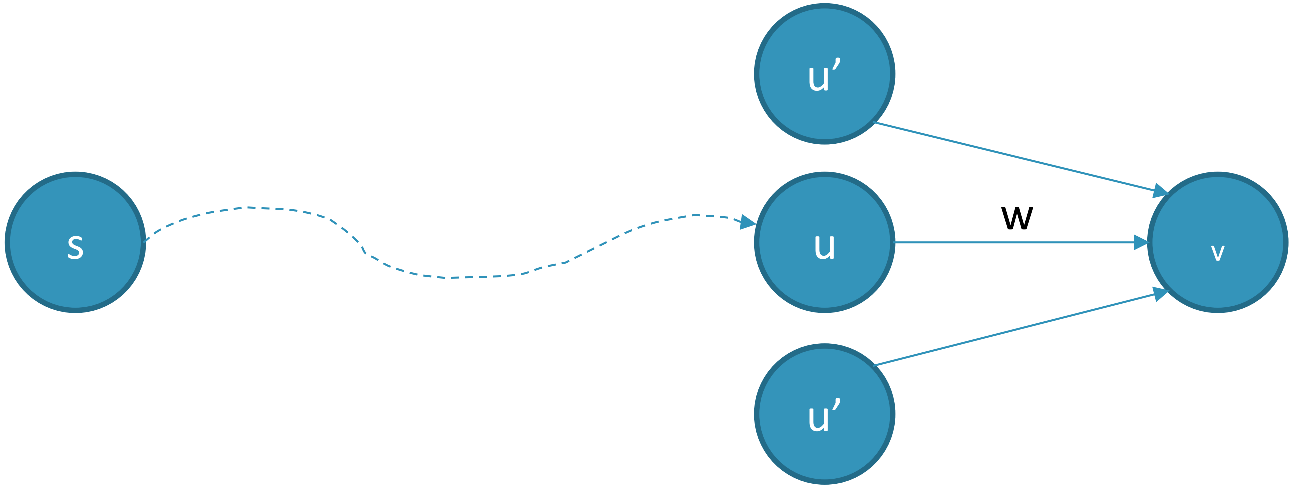
Corollary

- 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
- d(s,v)=d(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
- d(s,v)$\leq$d(s,u’)+w(u’,v)



Relaxation

- Most of the shortest-path algorithms are based on the relaxation technique:
  - Vector d[u] represents d(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 p[v]) whenever we discover that node v is more conveniently reached by traversing edge (u,v)

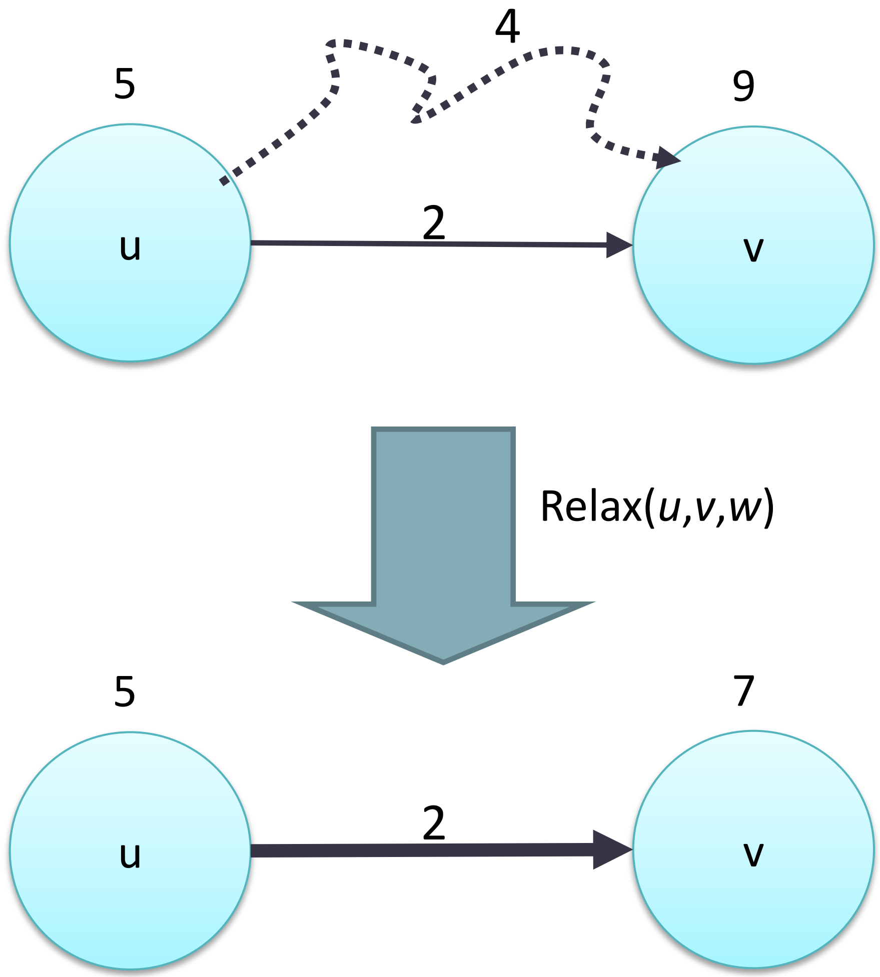
Initial state

- Initialize-Single-Source(G(V,E), s)
  - for all vertices v ∈ V
  - do
    - d[v]←∞
    - p[v]←NIL
  - d[s]←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] ← d[u]+w(u,v)
    - p[v] ← u

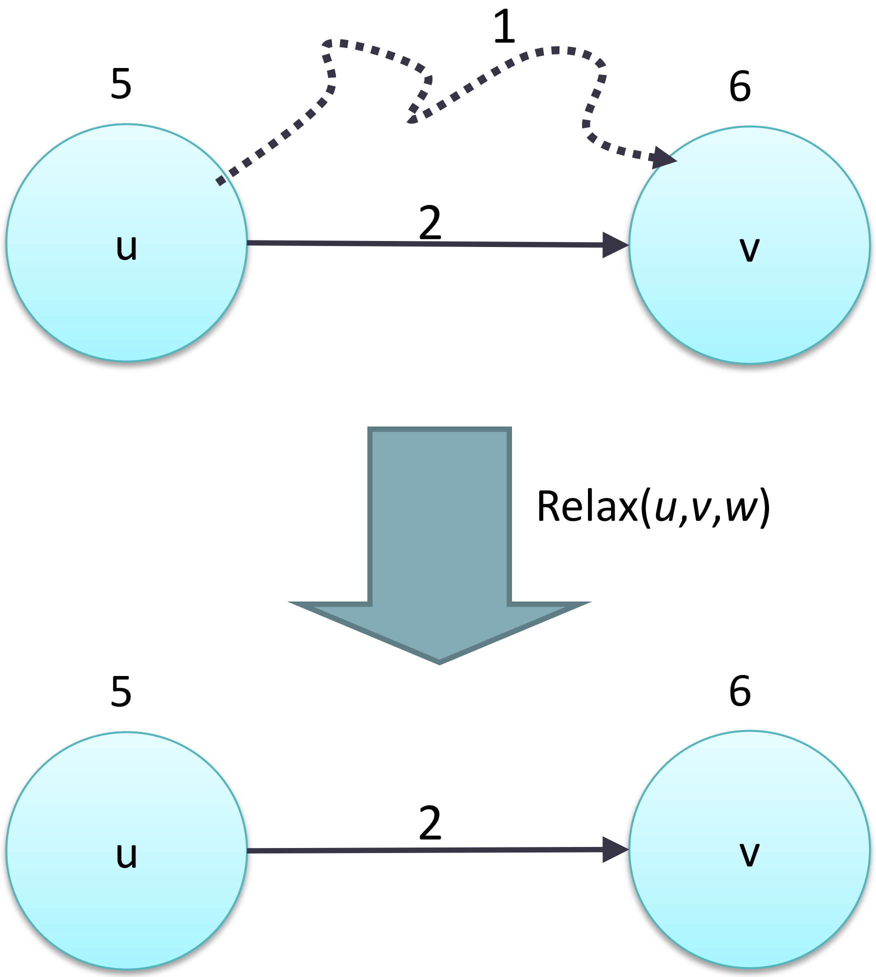
Example 1



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→$\mathbb{R}$.
- 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→$\mathbb{R}$ 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 Gp 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
  - $d[v] \geq d(s,v)$
- Additionally, if d[v] = d(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

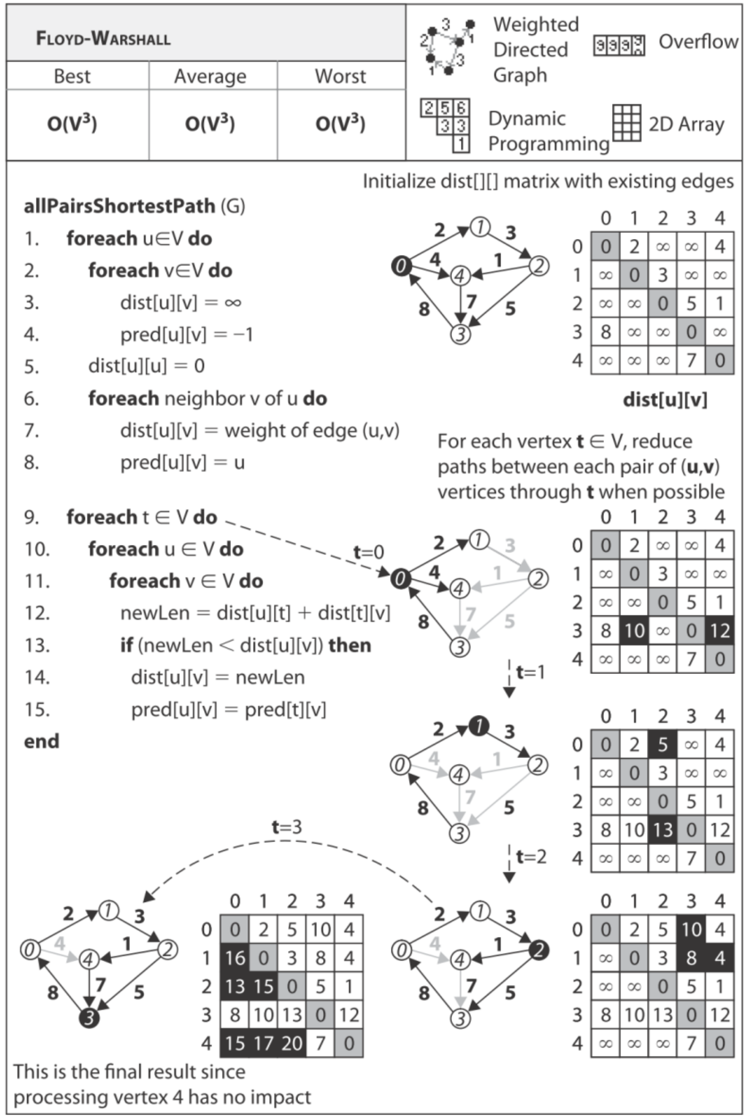
- https://networkx.org/documentation/stable/reference/algorithms/shortest_paths.html

Graphs: Finding shortest paths

FLOYD-WARSHALL ALGORITHM

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(V³)	**O(V³)**	**O(V³)**

```
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

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

	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

	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

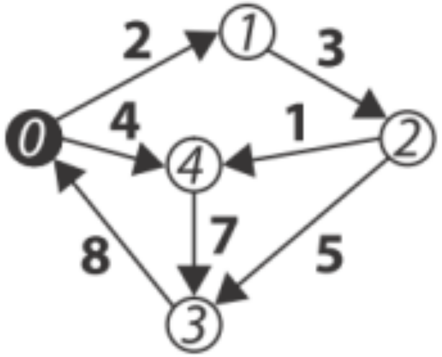
	0	1	2	3	4
0	0	2	5	10	4
1	16	0	3	8	4
2	13	15	0	5	1
3	8	10	13	0	12
4	15	17	20	7	0

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

Floyd-Warshall: initialization

```
def FLOYD_WARSHALL(V, E, w):

    #Initialise
    for u in V:
        for v in V:
            dist[u][v] = ∞
            pred[u][v] = None
        dist[u][u] = 0

        #Set distances with existing edges
        for n in neighborhood(u):
            dist[u][n] = w(u,n)
            pred[u][n] = u
```

Initialize dist[][] matrix with existing edges

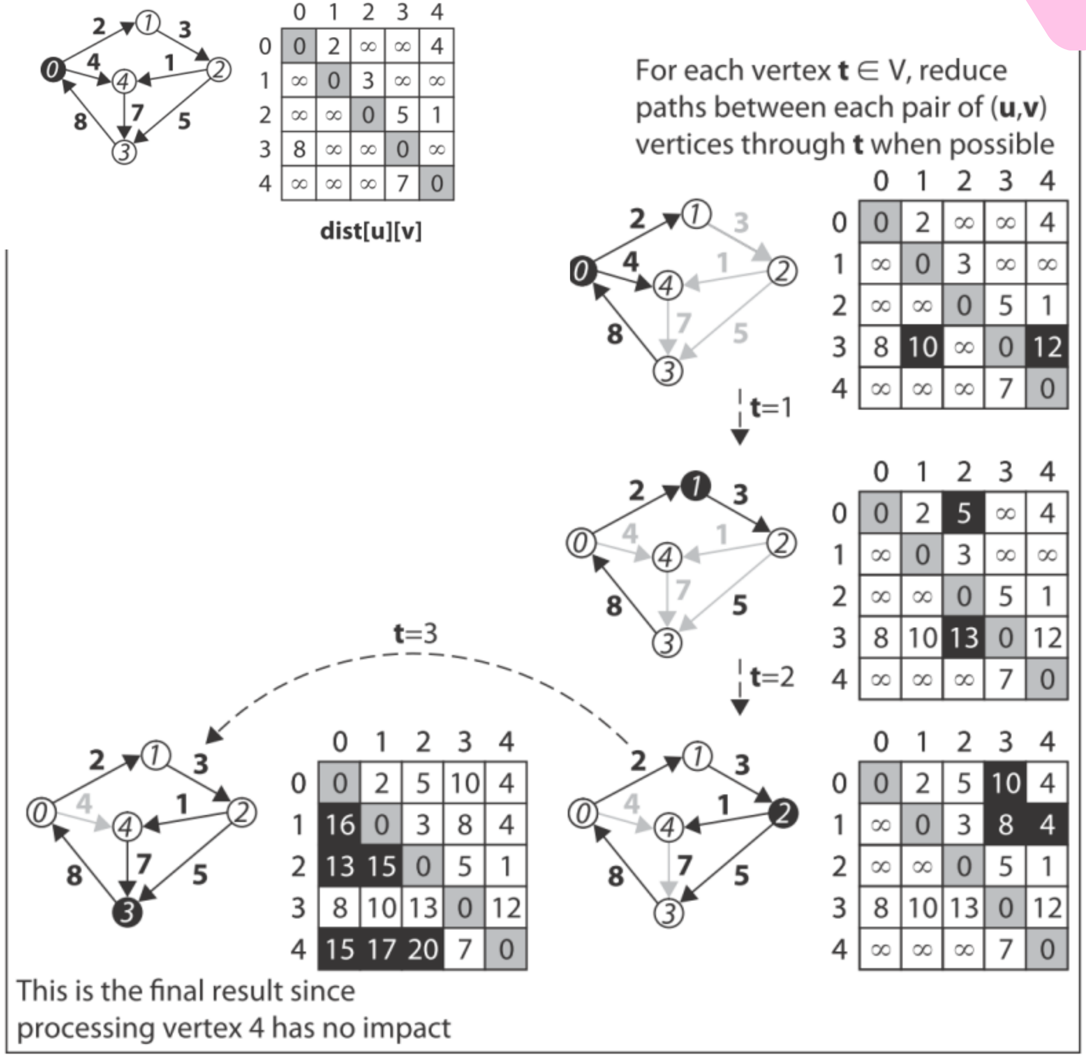


	0	1	2	3	4
0	0	2	∞	∞	4
1	∞	0	3	∞	∞
2	∞	∞	0	5	1
3	8	∞	∞	0	∞
4	∞	∞	∞	7	0

dist[u][v]

Floyd-Warshall: initialization

```
def FLOYD_WARSHALL(V, E, w):

    #Initialise
    for u in V:
        for v in V:
            dist[u][v] = ∞
            pred[u][v] = None
        dist[u][u] = 0

        #Set distances with existing edges
        for n in neighborhood(u):
            dist[u][n] = w(u,n)
            pred[u][n] = u

    #Relax by cycling on nodes (three times)
    for t in V:
        for u in V:
            for v in V:
                #Get a new path between u and v through t
                newDist = dist[u][t] + dist[t][v]

                #Check if the new path is better than previous best
                if (newDist < dist[u][v])
                  dist[u][v] = newLen
                  pred[u][v] = pred[t][v]

    return dist, pred
```



	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

	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

Complexity

- The Floyd-Warshall is basically executing 3 nested loops, each iterating over all vertices in the graph
- Complexity: O(V3)
- https://algorithms.discrete.ma.tum.de/graph-algorithms/spp-floyd-warshall/index_en.html

Implementation in NetworkX

floyd_warshall

`floyd_warshall(G, weight='weight')` [source]

Find all-pairs shortest path lengths using Floyd's algorithm.

Parameters:

G : *NetworkX graph*

weight: string, optional (default= 'weight')

Edge data key corresponding to the edge weight.

Returns:

distance : *dict*

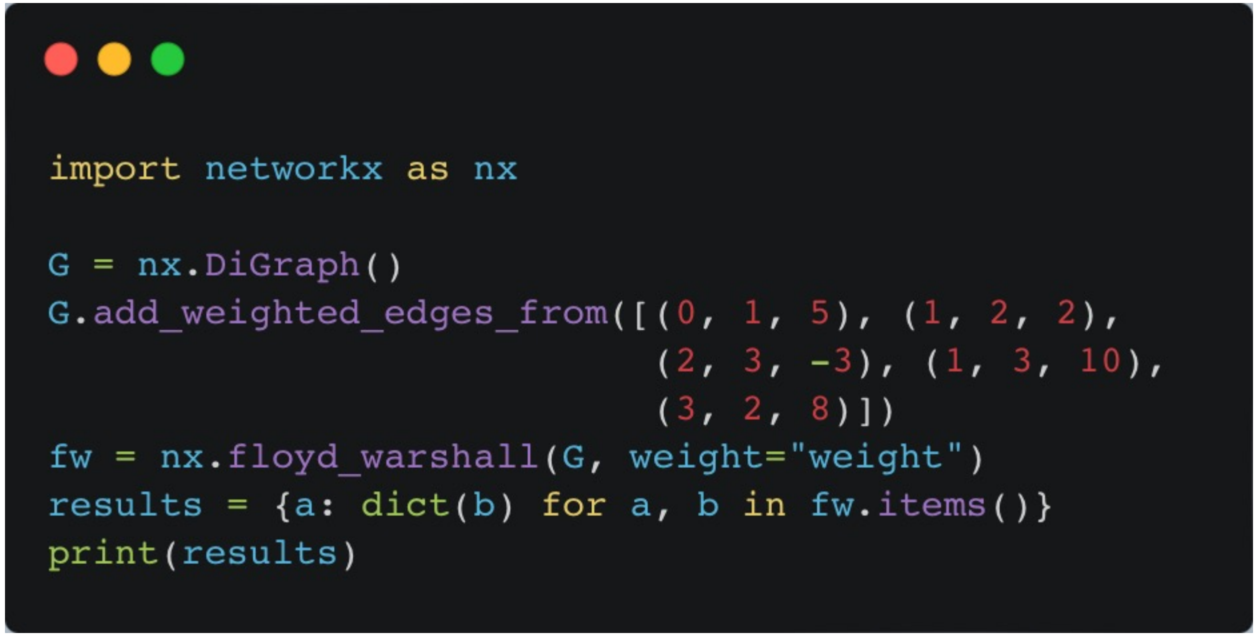
A dictionary, keyed by source and target, of shortest paths distances between nodes.

See also

`floyd_warshall_predecessor_and_distance`
`floyd_warshall_numpy`
`all_pairs_shortest_path`
`all_pairs_shortest_path_length`

Notes

Floyd's algorithm is appropriate for finding shortest paths in dense graphs or graphs with negative weights when Dijkstra's algorithm fails. This algorithm can still fail if there are negative cycles. It has running time $O(n^3)$ with running space of $O(n^2)$.



```
import networkx as nx

G = nx.DiGraph()
G.add_weighted_edges_from([(0, 1, 5), (1, 2, 2),
                           (2, 3, -3), (1, 3, 10),
                           (3, 2, 8)])
fw = nx.floyd_warshall(G, weight="weight")
results = {a: dict(b) for a, b in fw.items()}
print(results)
```

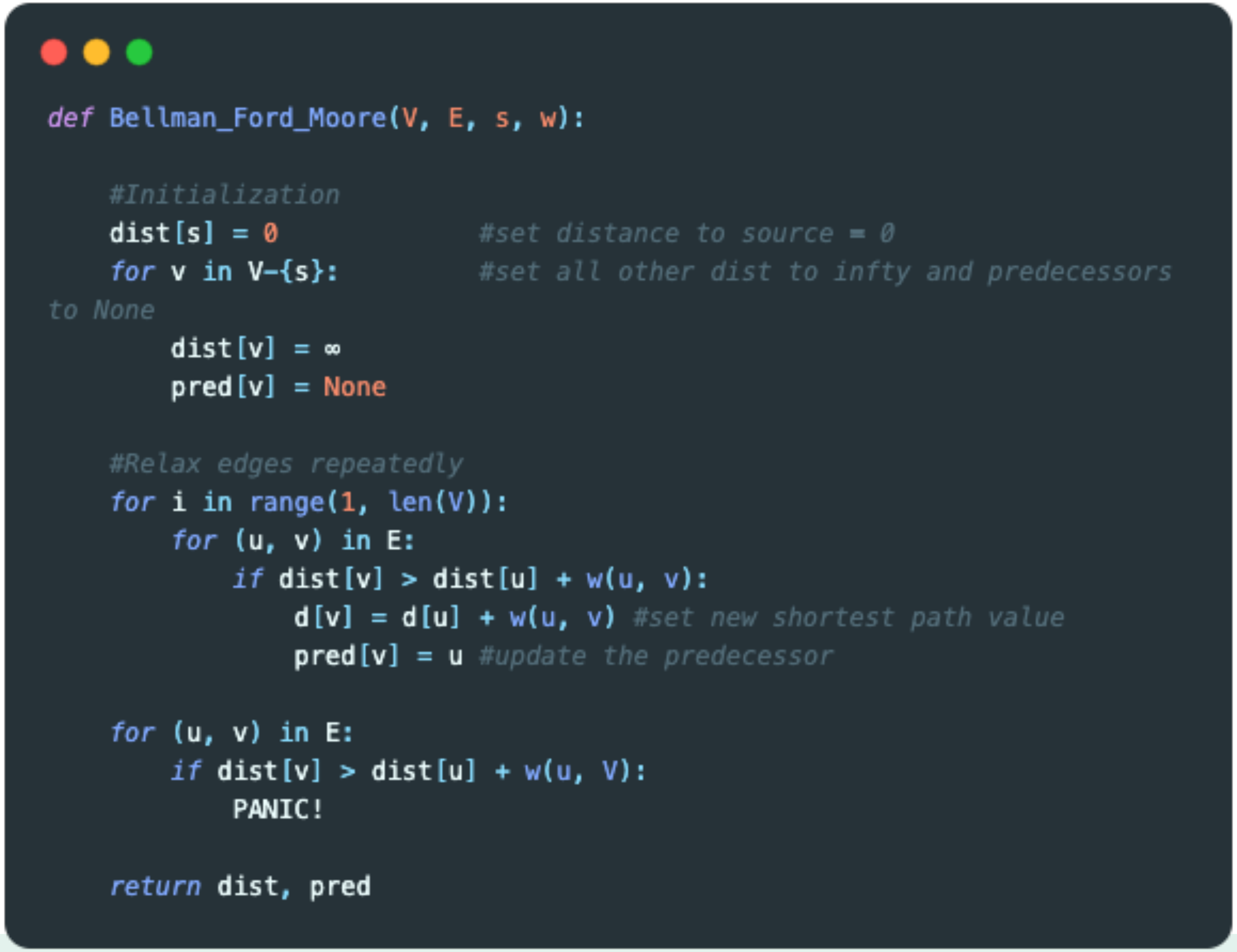
Graphs: Finding shortest paths

BELLMAN-FORD-MOORE ALGORITHM

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)

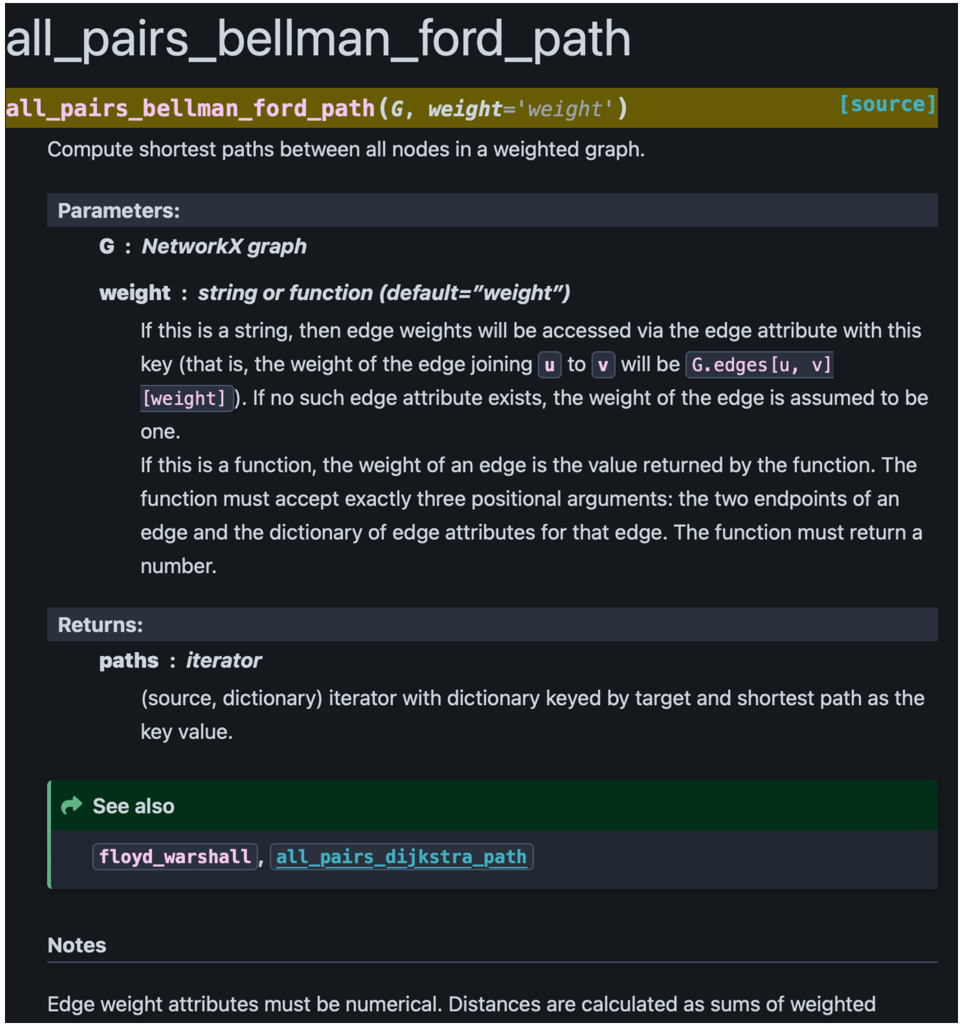
Bellman-Ford-Moore Algorithm



```
def Bellman_Ford_Moore(V, E, s, w):

    #Initialization
    dist[s] = 0             #set distance to source = 0
    for v in V-{s}:         #set all other dist to infty and predecessors
to None
        dist[v] = ∞
        pred[v] = None

    #Relax edges repeatedly
    for i in range(1, len(V)):
        for (u, v) in E:
            if dist[v] > dist[u] + w(u, v):
                d[v] = d[u] + w(u, v) #set new shortest path value
                pred[v] = u #update the predecessor

    for (u, v) in E:
        if dist[v] > dist[u] + w(u, V):
            PANIC!

    return dist, pred
```

https://algorithms.discrete.ma.tum.de/graph-algorithms/spp-bellman-ford/index_en.html

Implementation in NetworkX



all_pairs_bellman_ford_path

`all_pairs_bellman_ford_path(G, weight='weight')` [source]

Compute shortest paths between all nodes in a weighted graph.

Parameters:

G : *NetworkX graph*

weight : *string or function (default="weight")*

If this is a string, then edge weights will be accessed via the edge attribute with this key (that is, the weight of the edge joining `u` to `v` will be `G.edges[u, v][weight]`). If no such edge attribute exists, the weight of the edge is assumed to be one.

If this is a function, the weight of an edge is the value returned by the function. The function must accept exactly three positional arguments: the two endpoints of an edge and the dictionary of edge attributes for that edge. The function must return a number.

Returns:

paths : *iterator*

(source, dictionary) iterator with dictionary keyed by target and shortest path as the key value.

See also

`floyd_warshall`, `all_pairs_dijkstra_path`

Notes

Edge weight attributes must be numerical. Distances are calculated as sums of weighted

Graphs: Finding shortest paths

DIJKSTRA’S ALGORITHM

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!

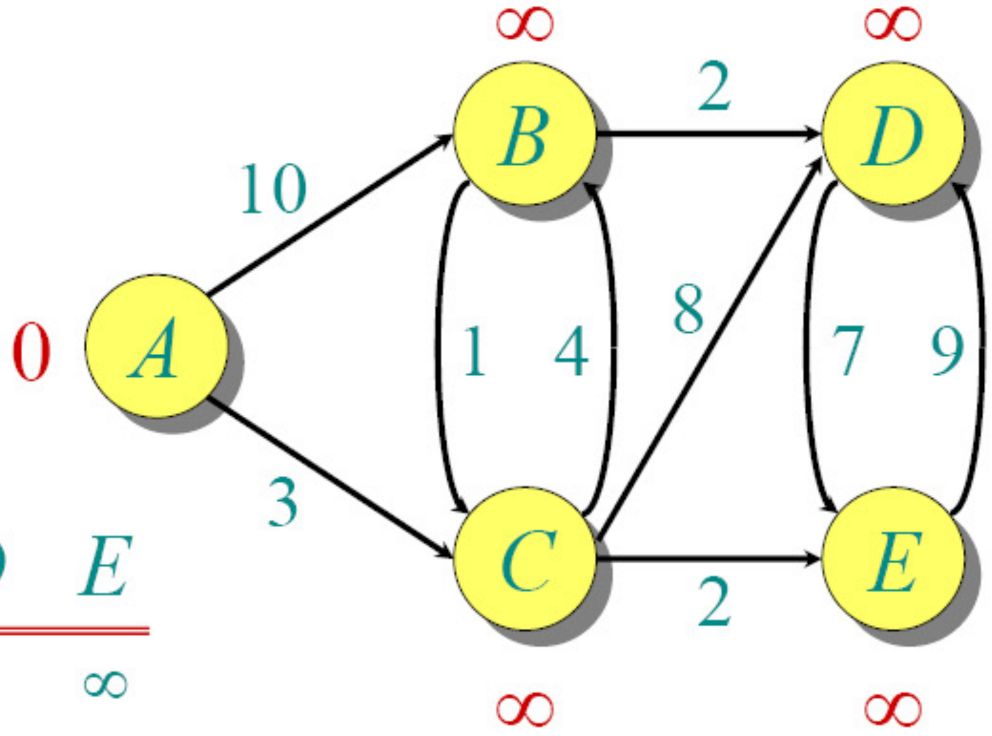
Dijkstra’s algorithm

```
def Dijkstra(V, E, s, w):

    #Initialise
    dist[s] = 0
    Q = []
    for v in V-{s}:
        dist[v] = ∞      # set initial dist to infty
        prev[v] = None   # set predecessors to None
        Q.append(v)      # Build a list of unvisited nodes

    #Run relaxation iteration
    while Q is not empty:
        u = q in Q with min dist[q]     # Pick one element of Q
        Q.remove(u)

        for v in neighborhood(u) still in Q:    # Cycle on neighbors of k
            newDist = dist[u] + w(u,v)          # Verify if path through u-v is better
            if newDist < dist[v]:
                dist[v] = newDist               # Update the new shortest path
                prev[v] = k

    return dist, pred
```

https://algorithms.discrete.ma.tum.de/graph-algorithms/spp-dijkstra/index_en.html

Dijkstra Animated Example

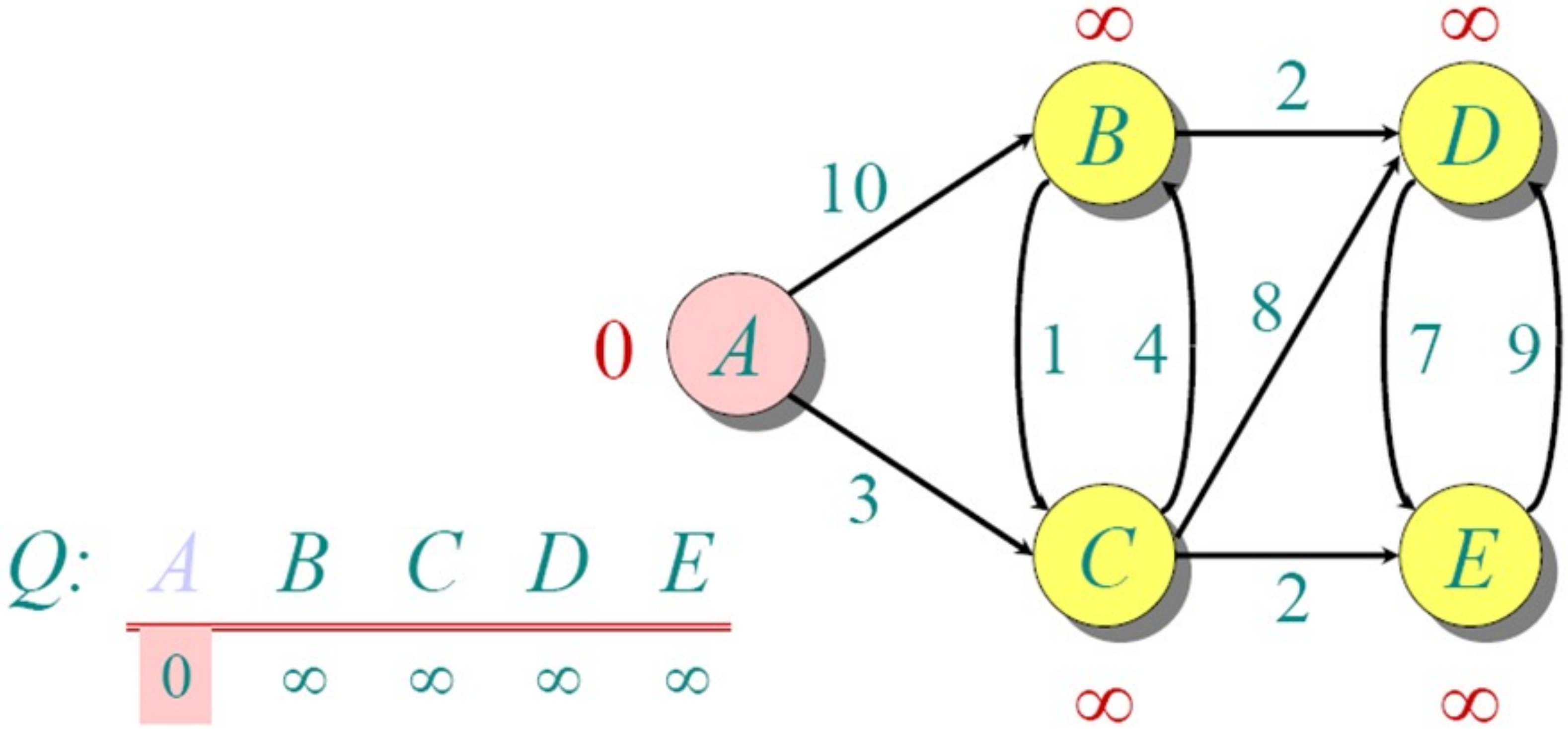
Initialize:



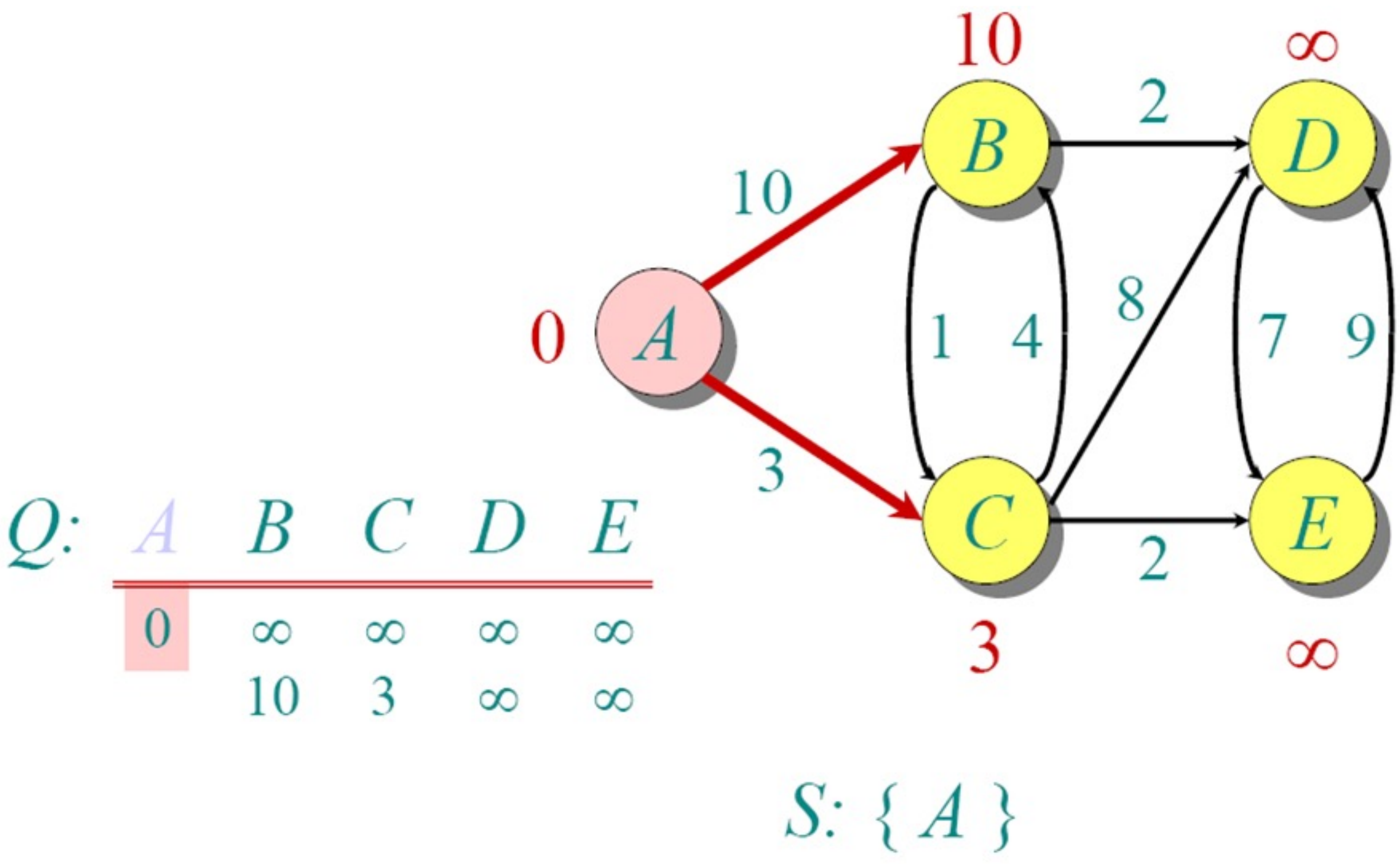
Q:	A	B	C	D	E
	0	∞	∞	∞	∞

S: {}

Dijkstra Animated Example



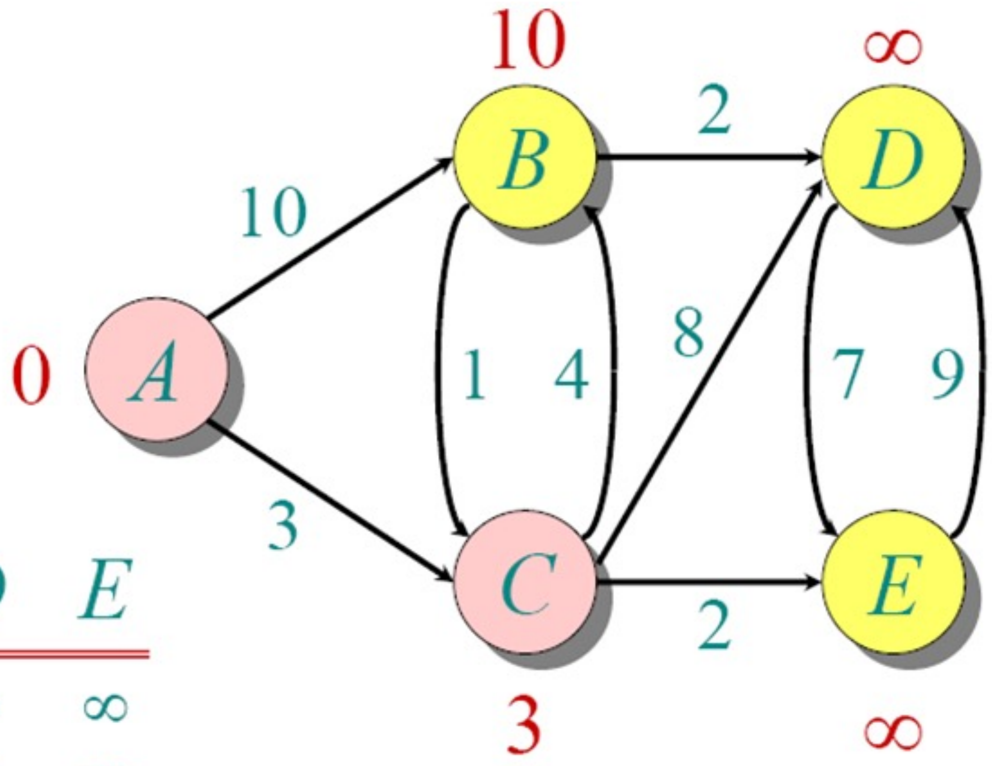
Dijkstra Animated Example



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞

S: { A }

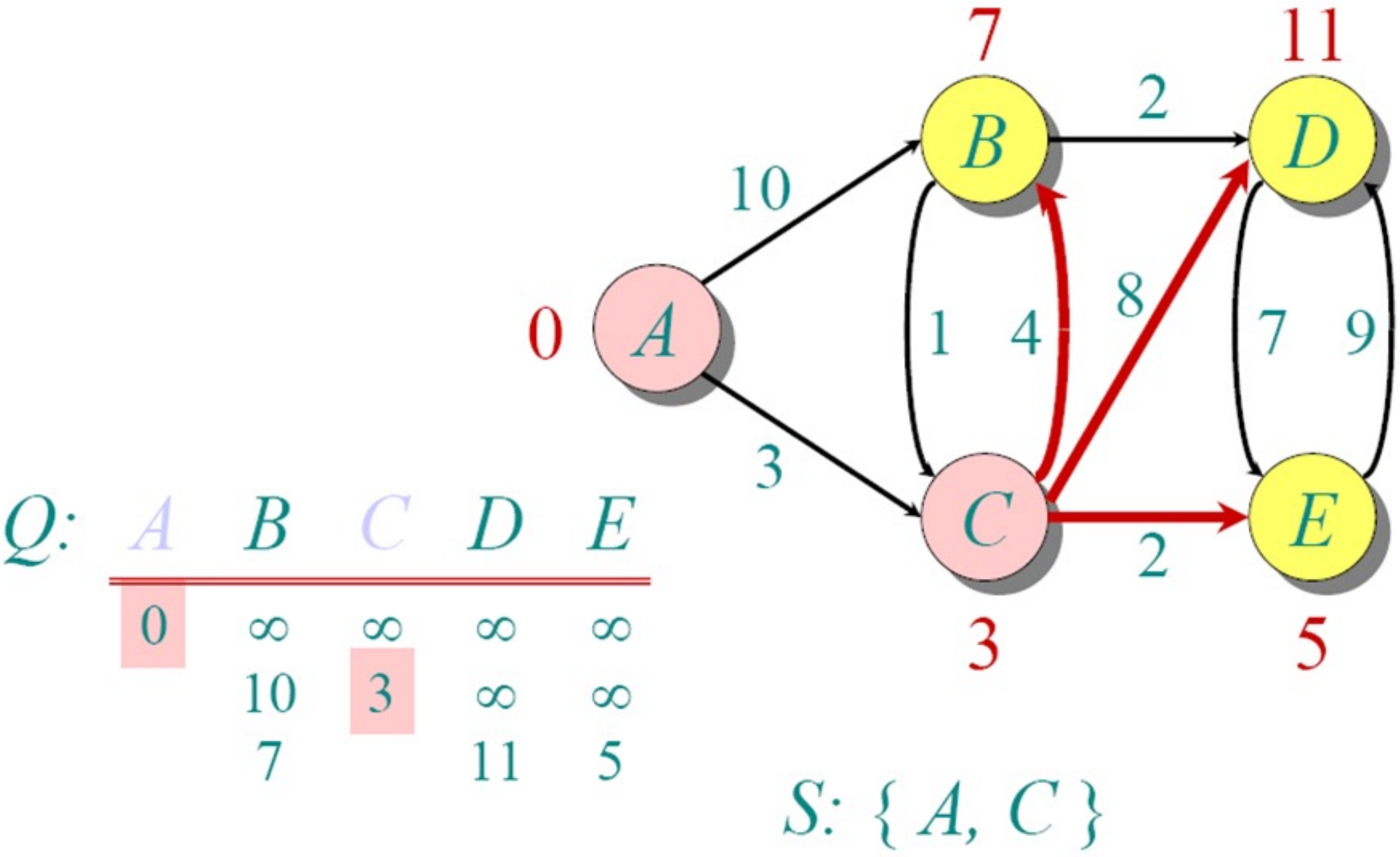
Dijkstra Animated Example



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞

S: { A, C }

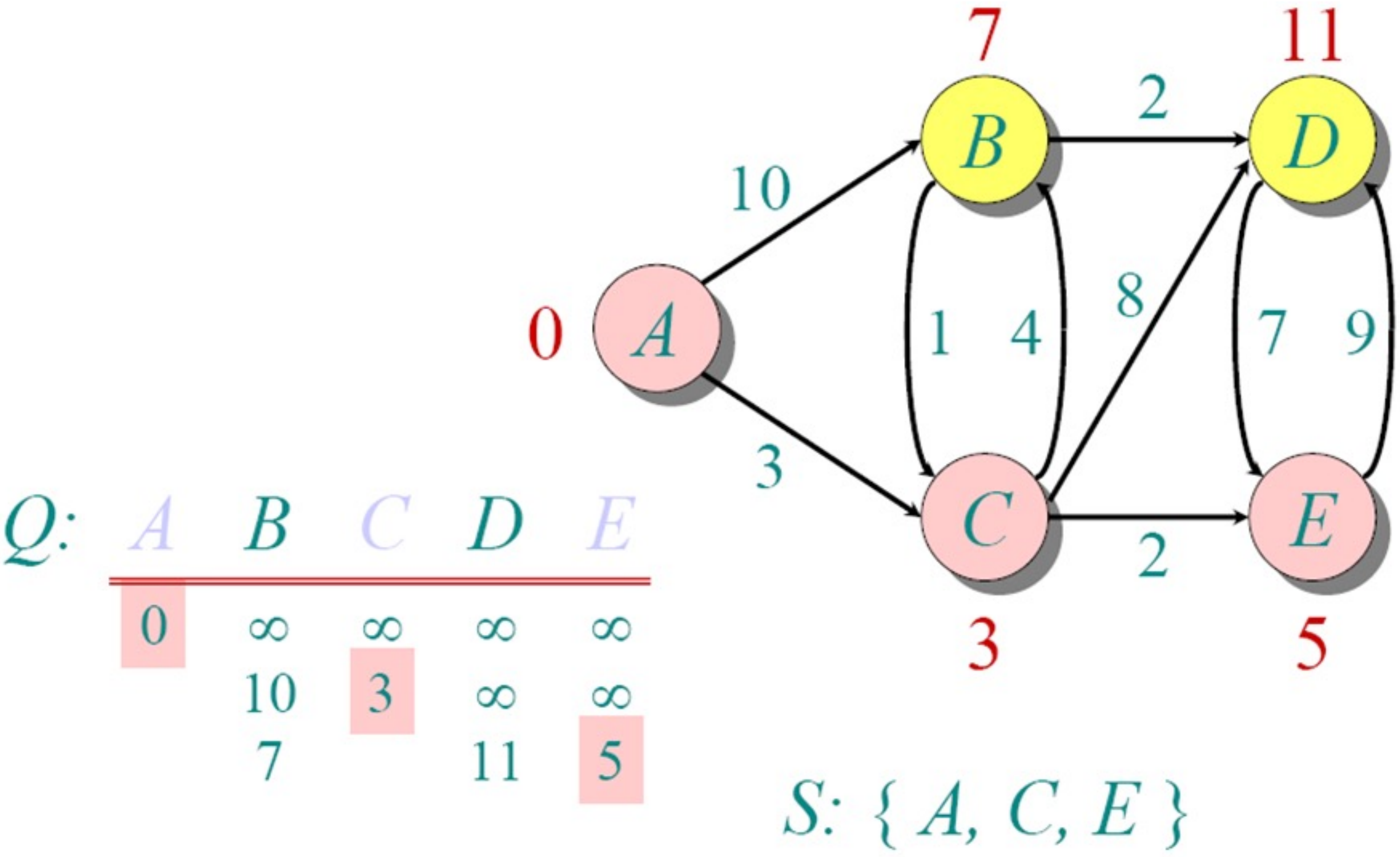
Dijkstra Animated Example



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

S: { A, C }

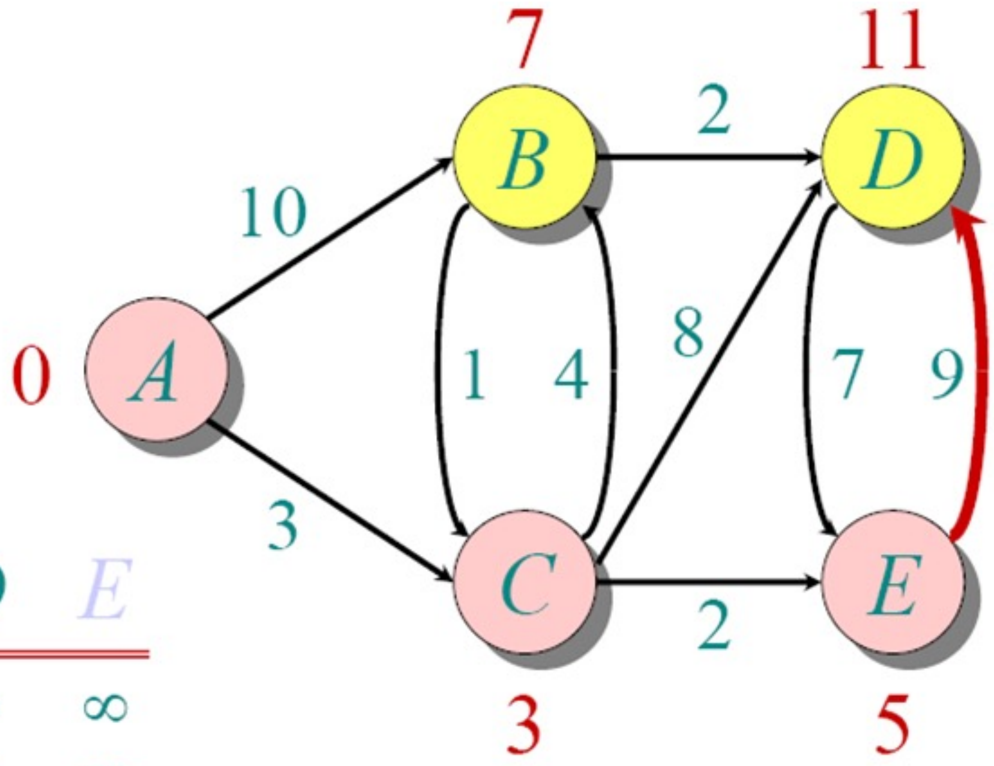
Dijkstra Animated Example



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

S: { A, C, E }

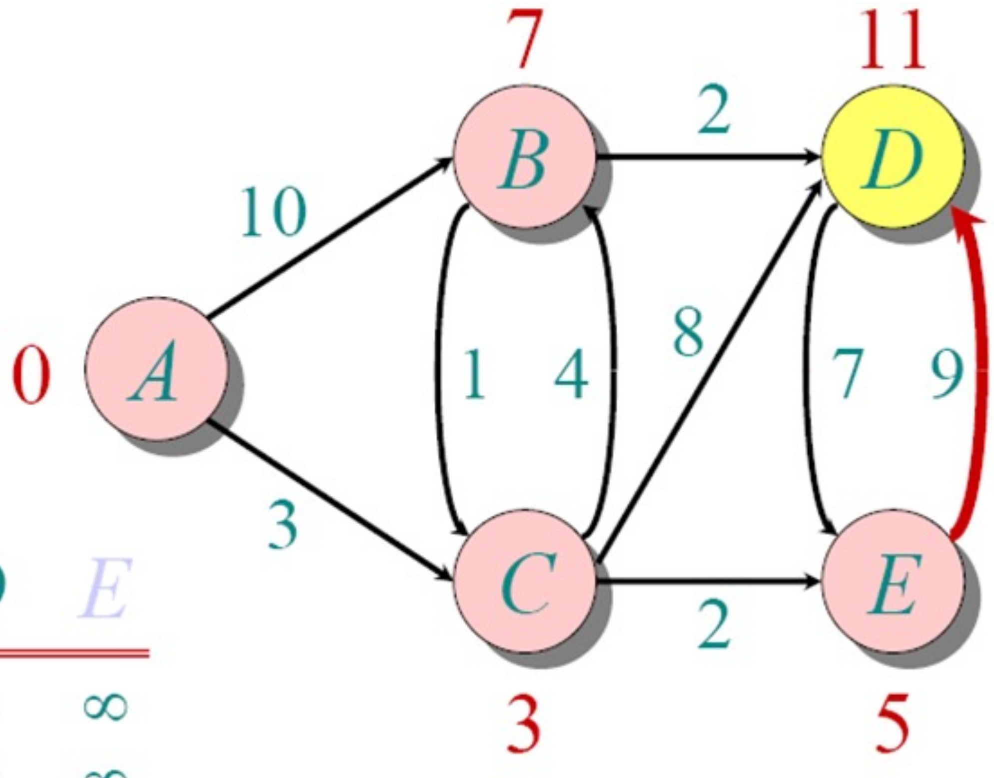
Dijkstra Animated Example



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

S: { A, C, E }

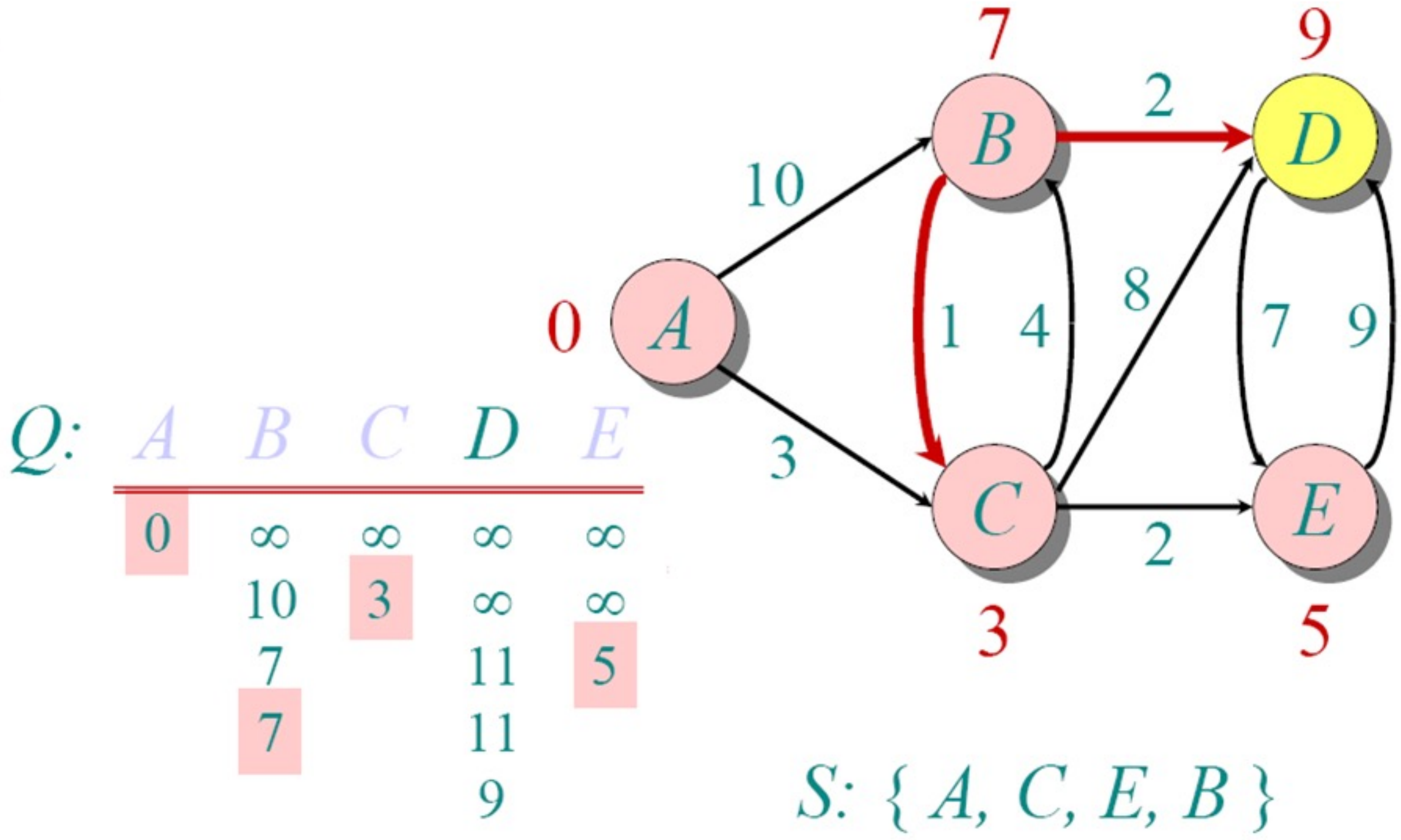
Dijkstra Animated Example



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

S: { A, C, E, B }

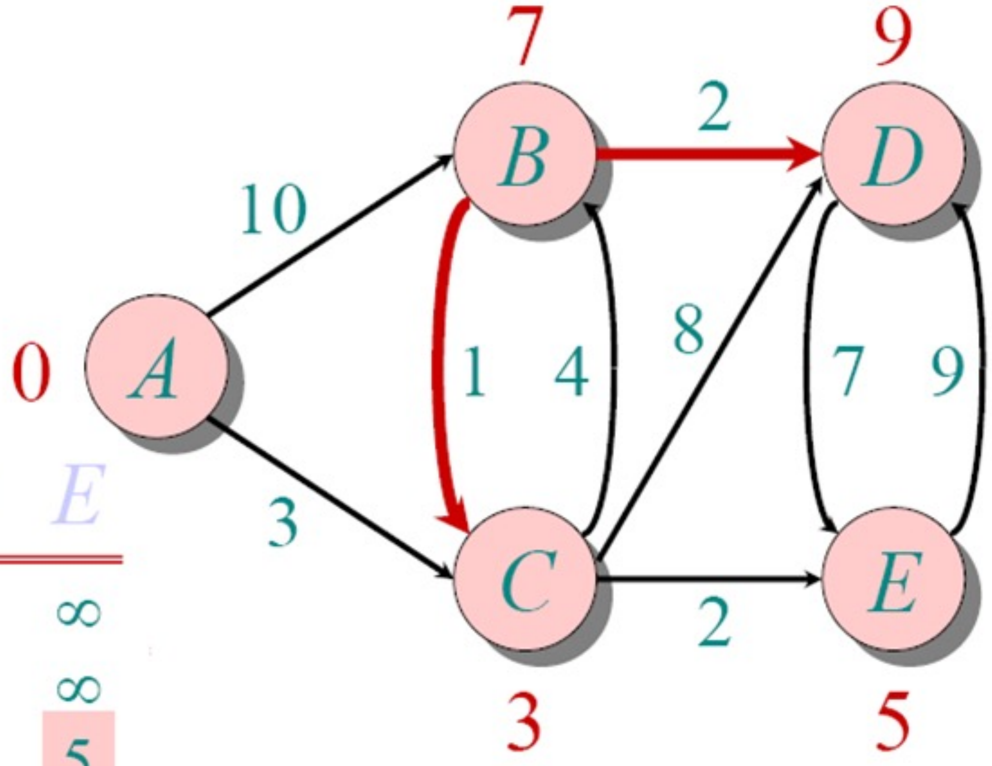
Dijkstra Animated Example



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 }

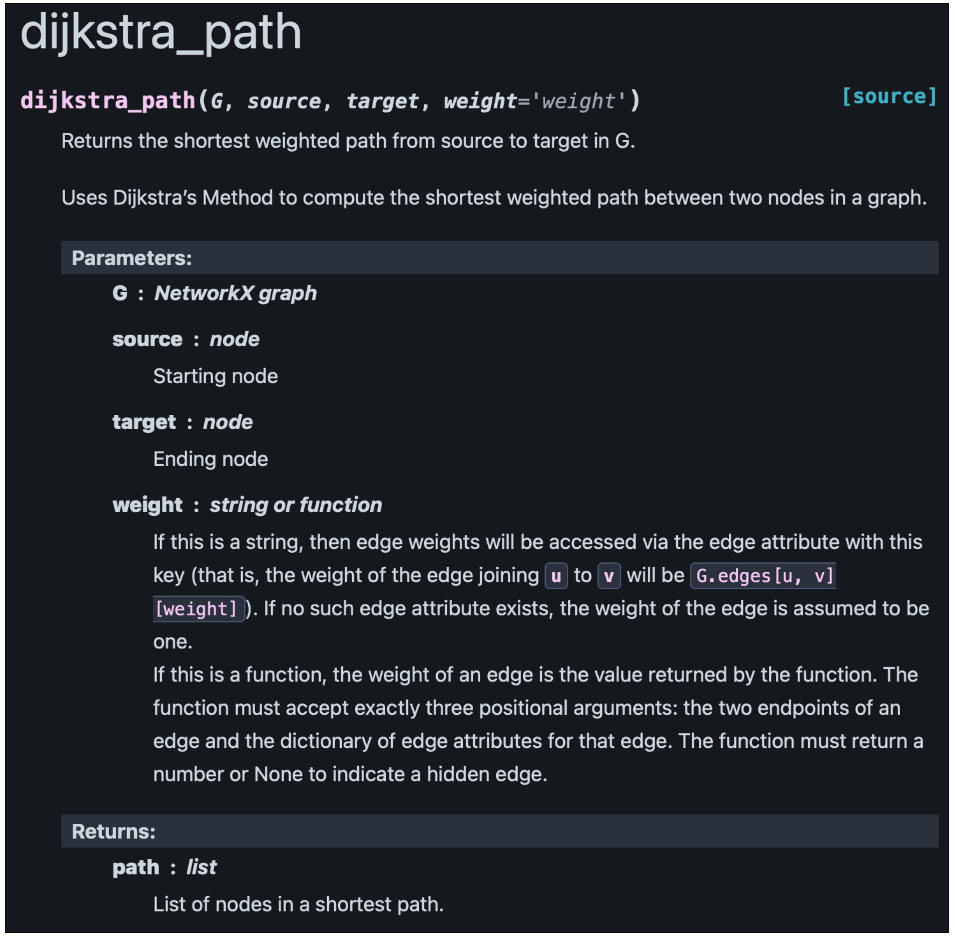
Dijkstra efficiency

- 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 in NetworkX



dijkstra_path

`dijkstra_path(G, source, target, weight='weight')` [source]

Returns the shortest weighted path from source to target in G.

Uses Dijkstra's Method to compute the shortest weighted path between two nodes in a graph.

Parameters:

G : *NetworkX graph*

source : *node*

Starting node

target : *node*

Ending node

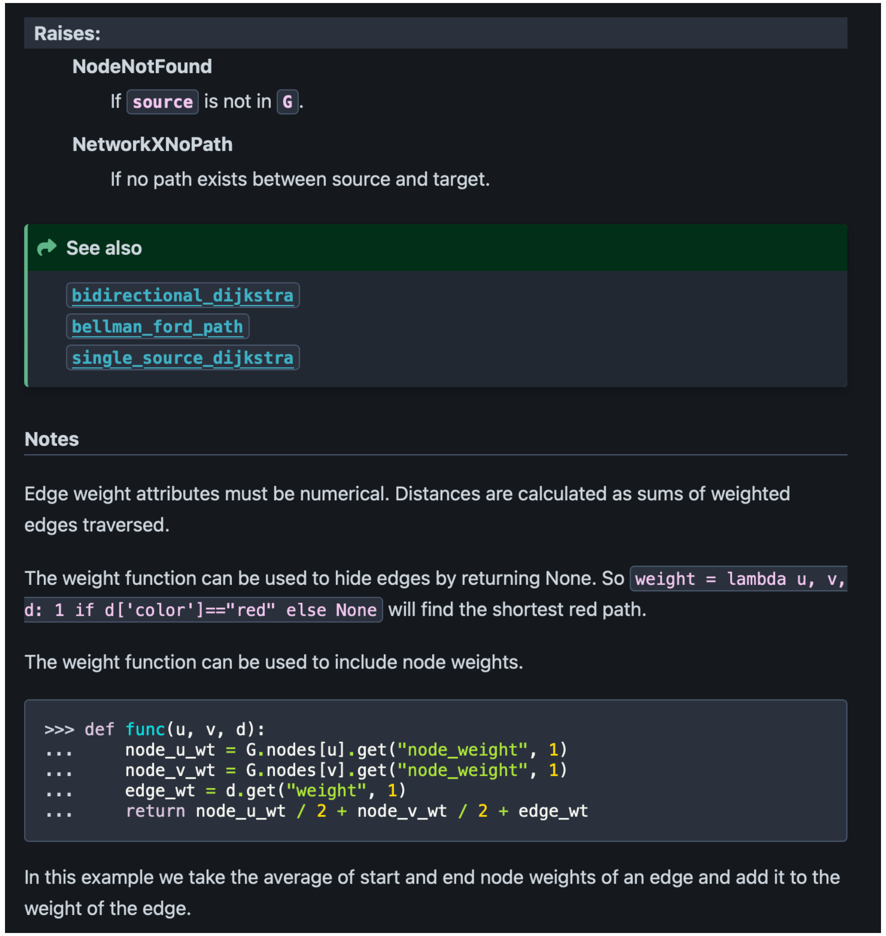
weight : *string or function*

If this is a string, then edge weights will be accessed via the edge attribute with this key (that is, the weight of the edge joining `u` to `v` will be `G.edges[u, v][weight]`). If no such edge attribute exists, the weight of the edge is assumed to be one.
If this is a function, the weight of an edge is the value returned by the function. The function must accept exactly three positional arguments: the two endpoints of an edge and the dictionary of edge attributes for that edge. The function must return a number or None to indicate a hidden edge.

Returns:

path : *list*

List of nodes in a shortest path.



Raises:

NodeNotFound

If `source` is not in `G`.

NetworkXNoPath

If no path exists between source and target.

See also

`bidirectional_dijkstra`
`bellman_ford_path`
`single_source_dijkstra`

Notes

Edge weight attributes must be numerical. Distances are calculated as sums of weighted edges traversed.

The weight function can be used to hide edges by returning None. So `weight = lambda u, v, d: 1 if d['color']=="red" else None` will find the shortest red path.

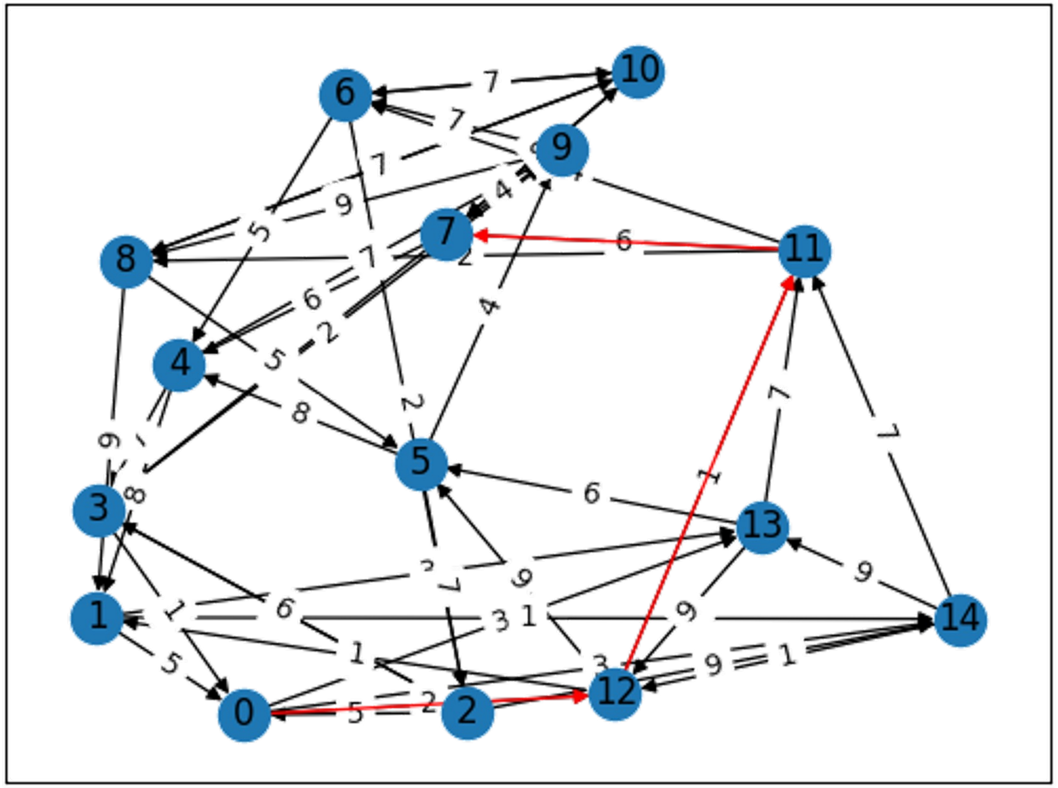
The weight function can be used to include node weights.

```
>>> def func(u, v, d):
...     node_u_wt = G.nodes[u].get("node_weight", 1)
...     node_v_wt = G.nodes[v].get("node_weight", 1)
...     edge_wt = d.get("weight", 1)
...     return node_u_wt / 2 + node_v_wt / 2 + edge_wt
```

In this example we take the average of start and end node weights of an edge and add it to the weight of the edge.

Example

Shortest path: 0-12-11-7
Shortest path length: 8



```
import networkx as nx
import matplotlib.pyplot as plt
import random

G = nx.directed_havel_hakimi_graph([3] * 15,
                                   [3] * 15,
                                   create_using=None)

for v in G.edges():
    print(G[v[0]][v[1]])
    G[v[0]][v[1]]['weight'] = random.randrange(1,10)
    print(G[v[0]][v[1]])
    print("---------")


print(G.nodes())
print(G.edges())
# nx.draw(G, with_labels=True, font_weight='bold')

pos=nx.spring_layout(G) # pos =
nx.nx_agraph.graphviz_layout(G)
nx.draw_networkx(G,pos)
labels = nx.get_edge_attributes(G,'weight')
nx.draw_networkx_edge_labels(G,pos,edge_labels=labels)

print(nx.dijkstra_path(G, 0, 7))
print(nx.dijkstra_path_length(G, 0, 7))

optpath = nx.dijkstra_path(G, 0, 7)
optedges = []
for i in range(0, len(optpath)-1):
    optedges.append([optpath[i], optpath[i+1]])


nx.draw_networkx_edges(G, pos, optedges,
edge_color="red")

plt.savefig("plot")
plt.show()
```

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

Graphs: Cycles

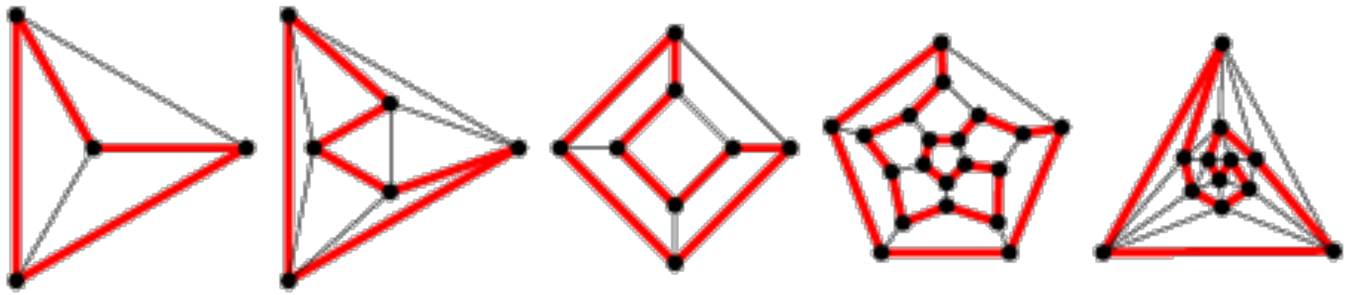
CYCLES: DEFINITIONS

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 which uses each 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.
- An Eulerian Graph is a graph which admits an Eulerian cycle.
- Euler showed (without proof) that a connected simple graph is Eulerian iff it has no graph vertices of odd degree (i.e., all vertices are of even degree).

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!

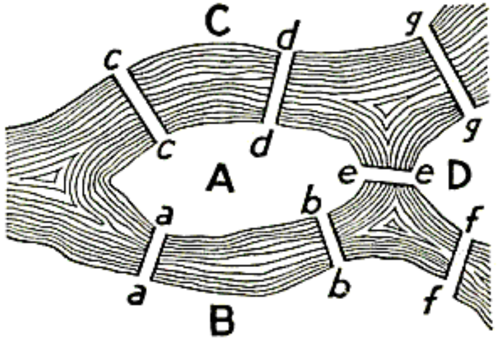
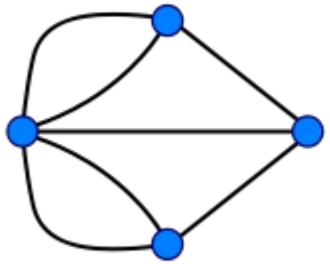


Figure 98. *Geographic Map: The Königsberg Bridges.*



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)

Graphs: Cycles

ALGORITHMS

Eulerian cycles: Hierholzer's algorithm (1)

- Let us assume that G is an Eulerian graph.
- 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, although it may not cover all the vertices and edges of the initial graph.

Eulerian cycles: Hierholzer's algorithm (2)

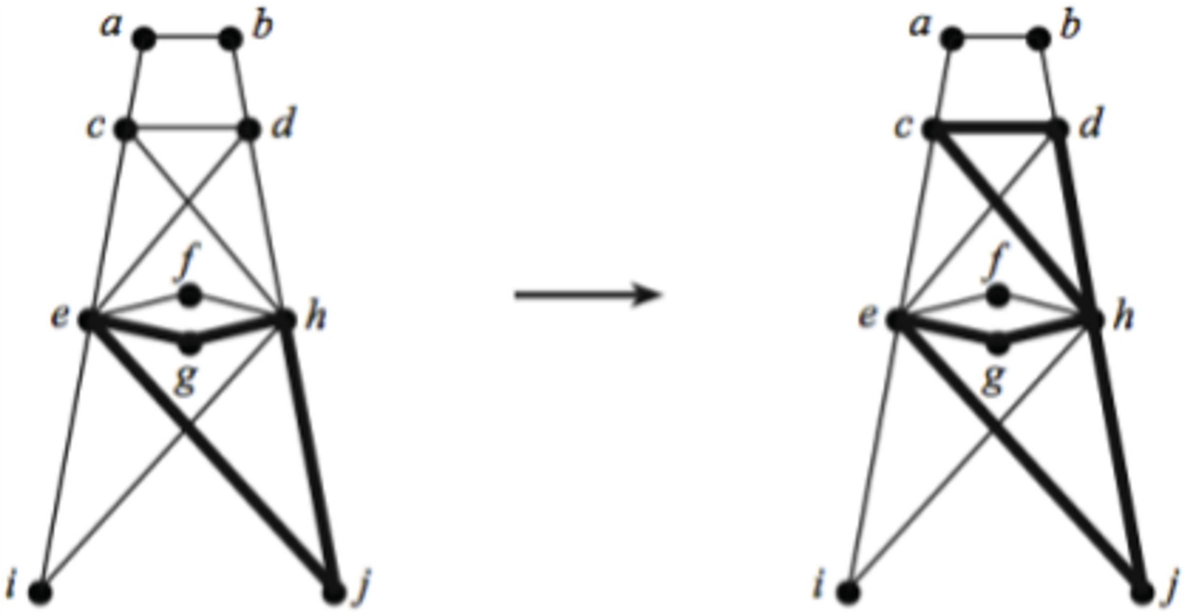
- As long as there exists a vertex v that belongs to the current tour but that has adjacent edges not part of the tour, start another trail from v, following unused edges until returning to v, and join the tour formed in this way to the previous tour.

Hierholzer's algorithm Pseudocode

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 $R_1$ as visited. Let $i=1$
2. If $R_i$ contains all edges of G, break.
3. If $R_i$ does not contains all edges of G, then let $v_i$ be a node of $R_i$ that is incident with an unmarked edge $e_i$
4. Build a new circuit $Q_i$ , starting from node $v_i$ and using edge $e_i$ . Mark edges of $Q_i$ as visited.
5. $R_{i+1}$ will result as the conjunction in $v_i$ of $R_i$ and $Q_i$
6. Increment i by 1 and go to step 2
```

Example

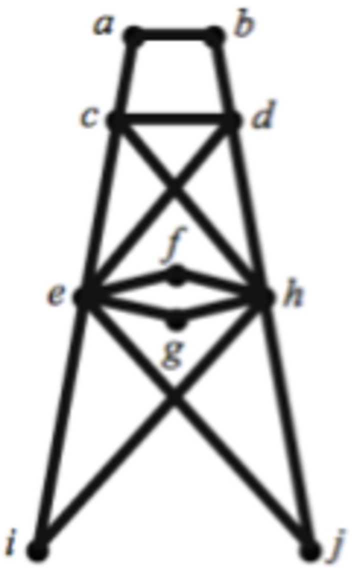


R_1: e, g, h, j, e

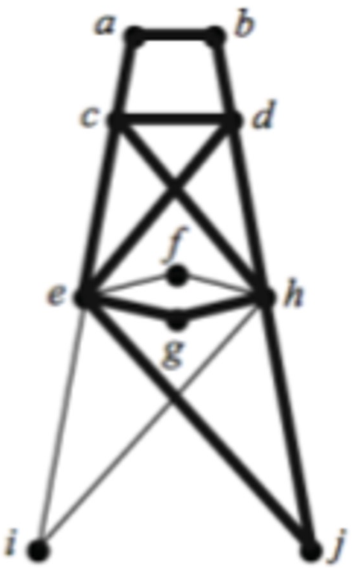
Q_1: h, d, c, h

R_2: e, g, **h**, **d**, **c**, **h**, j, e

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



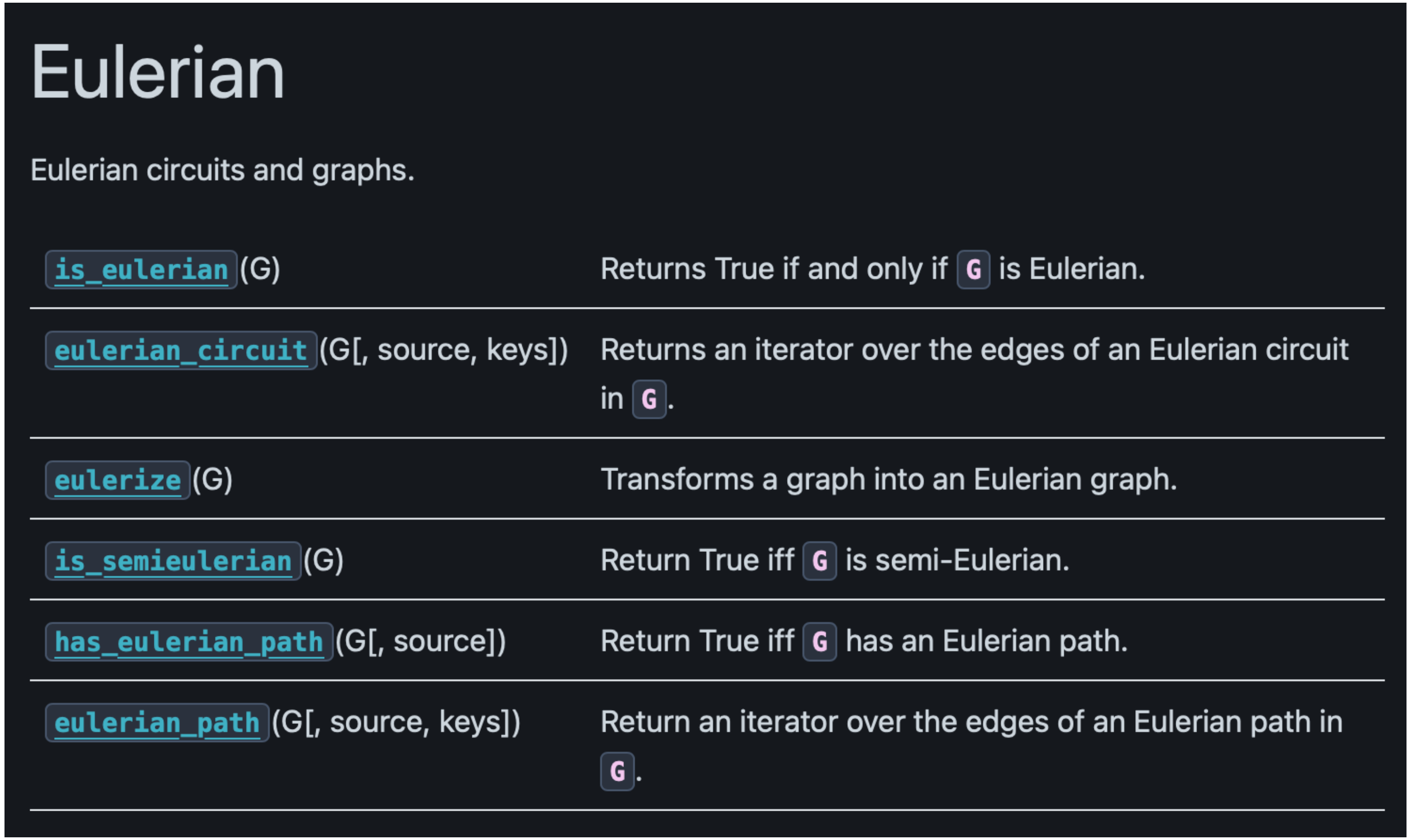
R_4: e, g, **h**, **f**, **e**, **i**, **h**, d, b, a, c, e, d, c, h, j, e



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 NetworkX



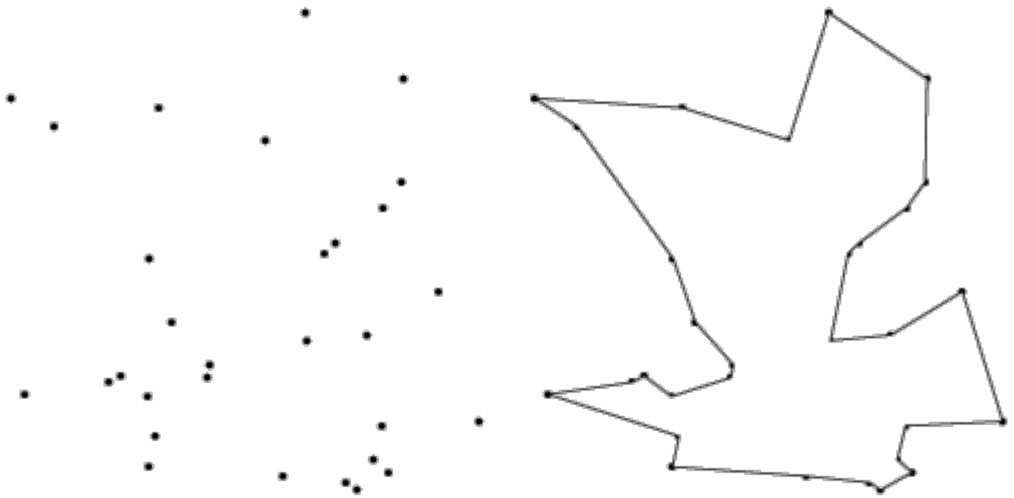
Eulerian

Eulerian circuits and graphs.

`is_eulerian`(G)	Returns True if and only if `G` is Eulerian.
`eulerian_circuit`(G[, source, keys])	Returns an iterator over the edges of an Eulerian circuit in `G`.
`eulerize`(G)	Transforms a graph into an Eulerian graph.
`is_semieulerian`(G)	Return True iff `G` is semi-Eulerian.
`has_eulerian_path`(G[, source])	Return True iff `G` has an Eulerian path.
`eulerian_path`(G[, source, keys])	Return an iterator over the edges of an Eulerian path in `G`.

Hamiltonian Cycles

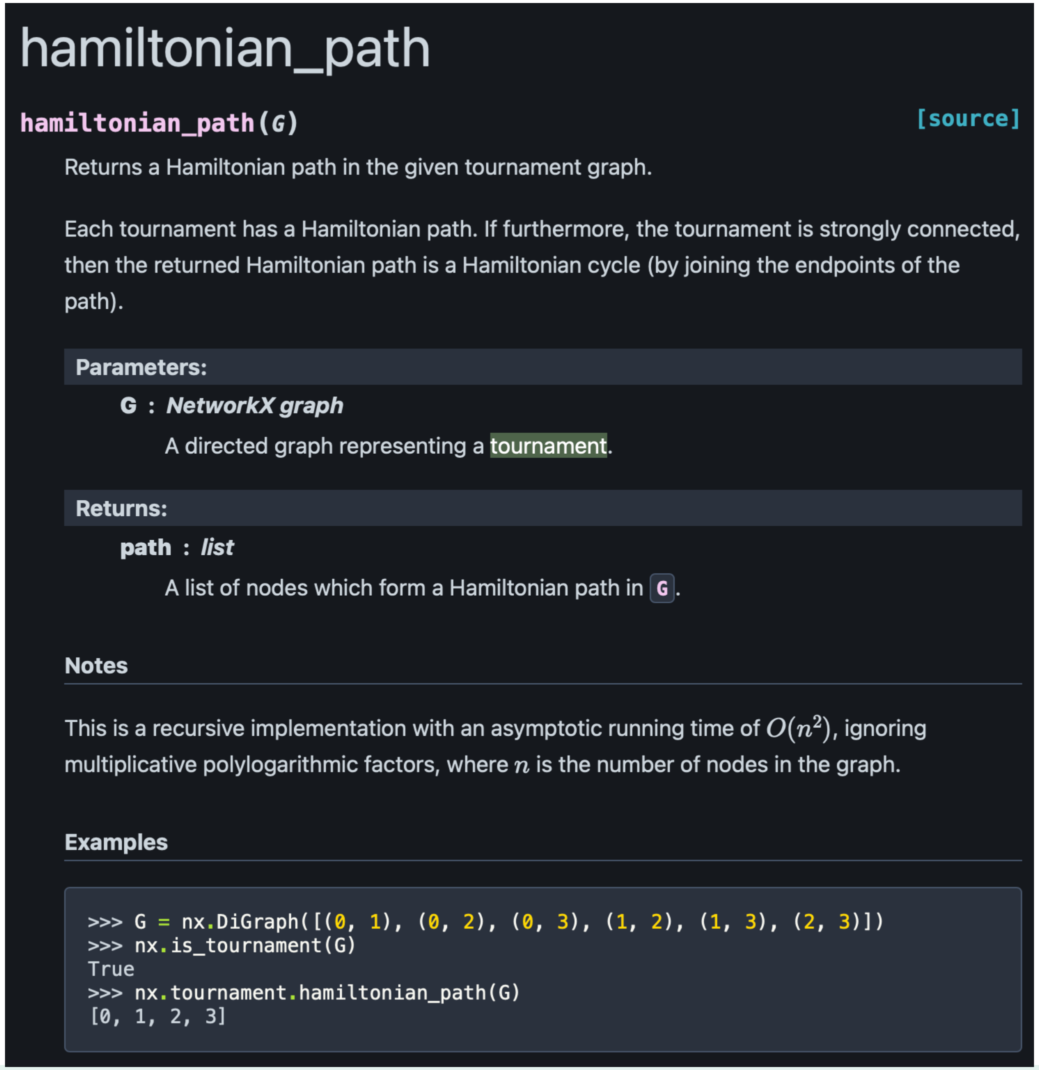
- There are theorems to identify whether a graph is Hamiltonian (i.e., whether it contains at least one Hamiltonian Cycle)
- Finding such a cycle has no known efficient solution, in the general case
- Example: the Traveling Salesman Problem (TSP)



The Traveling Salesman Problem (TSP)

- Given a collection of cities, find the shortest route to visit them exactly once.
- Most notorious NP-complete problem
- Typically is solved through backtracking:
  - 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 NetworkX



hamiltonian_path

`hamiltonian_path(G)` [source]

Returns a Hamiltonian path in the given tournament graph.

Each tournament has a Hamiltonian path. If furthermore, the tournament is strongly connected, then the returned Hamiltonian path is a Hamiltonian cycle (by joining the endpoints of the path).

Parameters:

G : *NetworkX graph*

A directed graph representing a tournament.

Returns:

path : *list*

A list of nodes which form a Hamiltonian path in `G`.

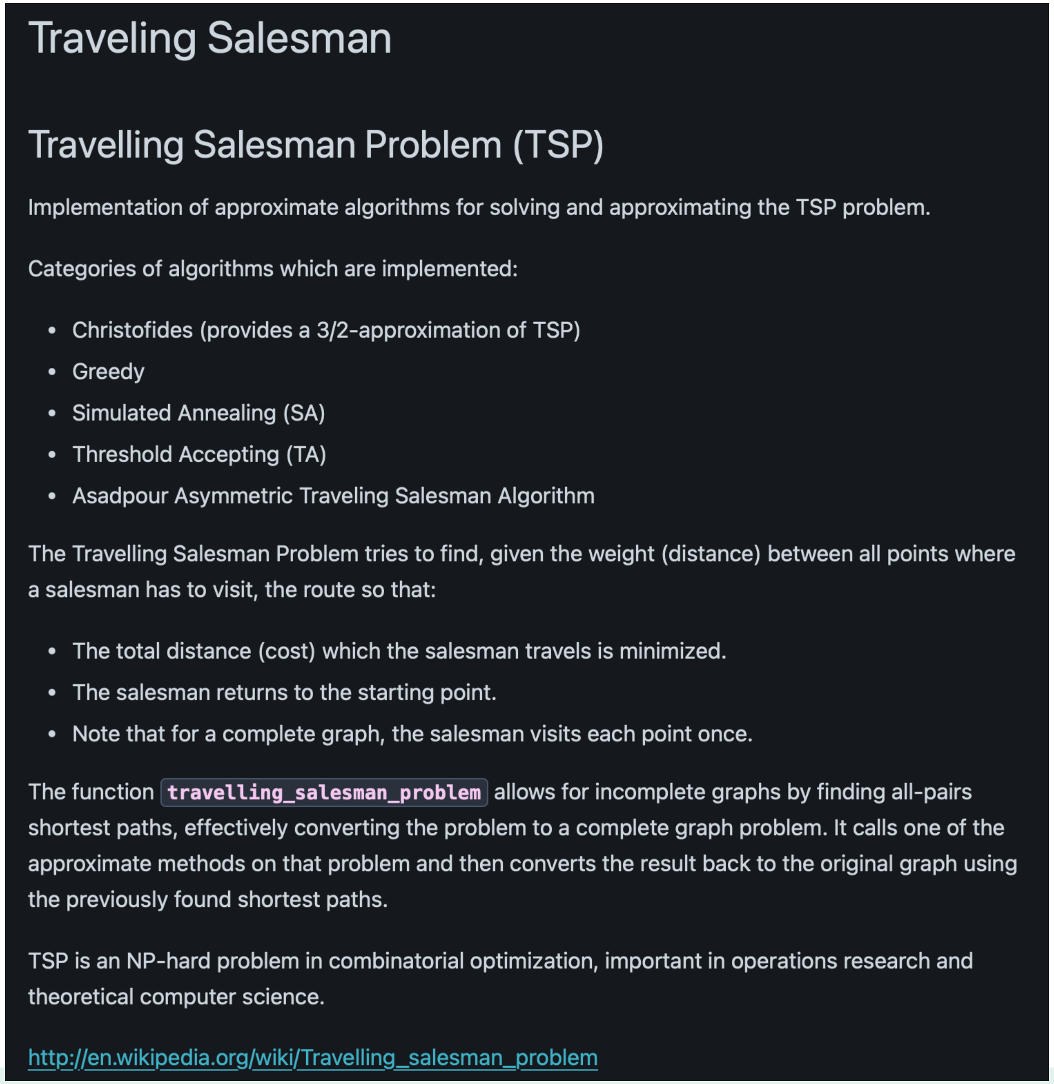
Notes

This is a recursive implementation with an asymptotic running time of $O(n^2)$, ignoring multiplicative polylogarithmic factors, where n is the number of nodes in the graph.

Examples

```
>>> G = nx.DiGraph([(0, 1), (0, 2), (0, 3), (1, 2), (1, 3), (2, 3)])
>>> nx.is_tournament(G)
True
>>> nx.tournament.hamiltonian_path(G)
[0, 1, 2, 3]
```

Alternatives on graphs



Traveling Salesman

Travelling Salesman Problem (TSP)

Implementation of approximate algorithms for solving and approximating the TSP problem.

Categories of algorithms which are implemented:

- Christofides (provides a 3/2-approximation of TSP)
- Greedy
- Simulated Annealing (SA)
- Threshold Accepting (TA)
- Asadpour Asymmetric Traveling Salesman Algorithm

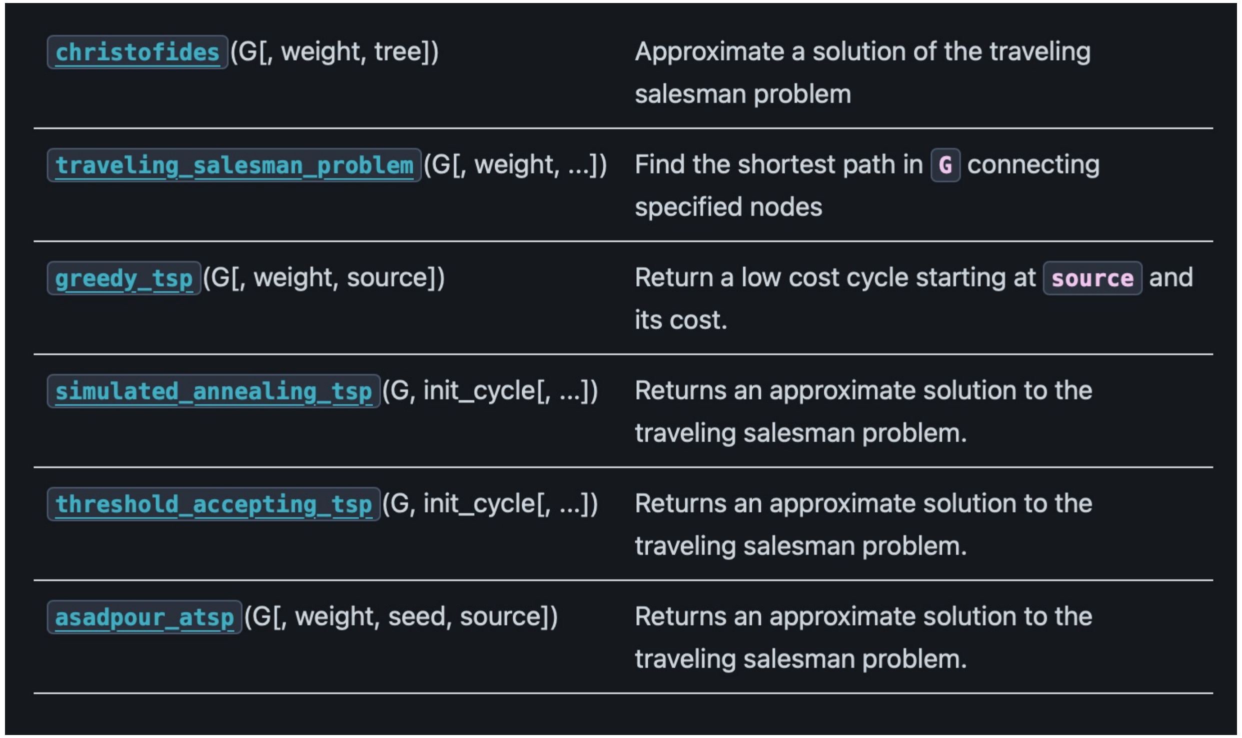
The Travelling Salesman Problem tries to find, given the weight (distance) between all points where a salesman has to visit, the route so that:

- The total distance (cost) which the salesman travels is minimized.
- The salesman returns to the starting point.
- Note that for a complete graph, the salesman visits each point once.

The function `travelling_salesman_problem` allows for incomplete graphs by finding all-pairs shortest paths, effectively converting the problem to a complete graph problem. It calls one of the approximate methods on that problem and then converts the result back to the original graph using the previously found shortest paths.

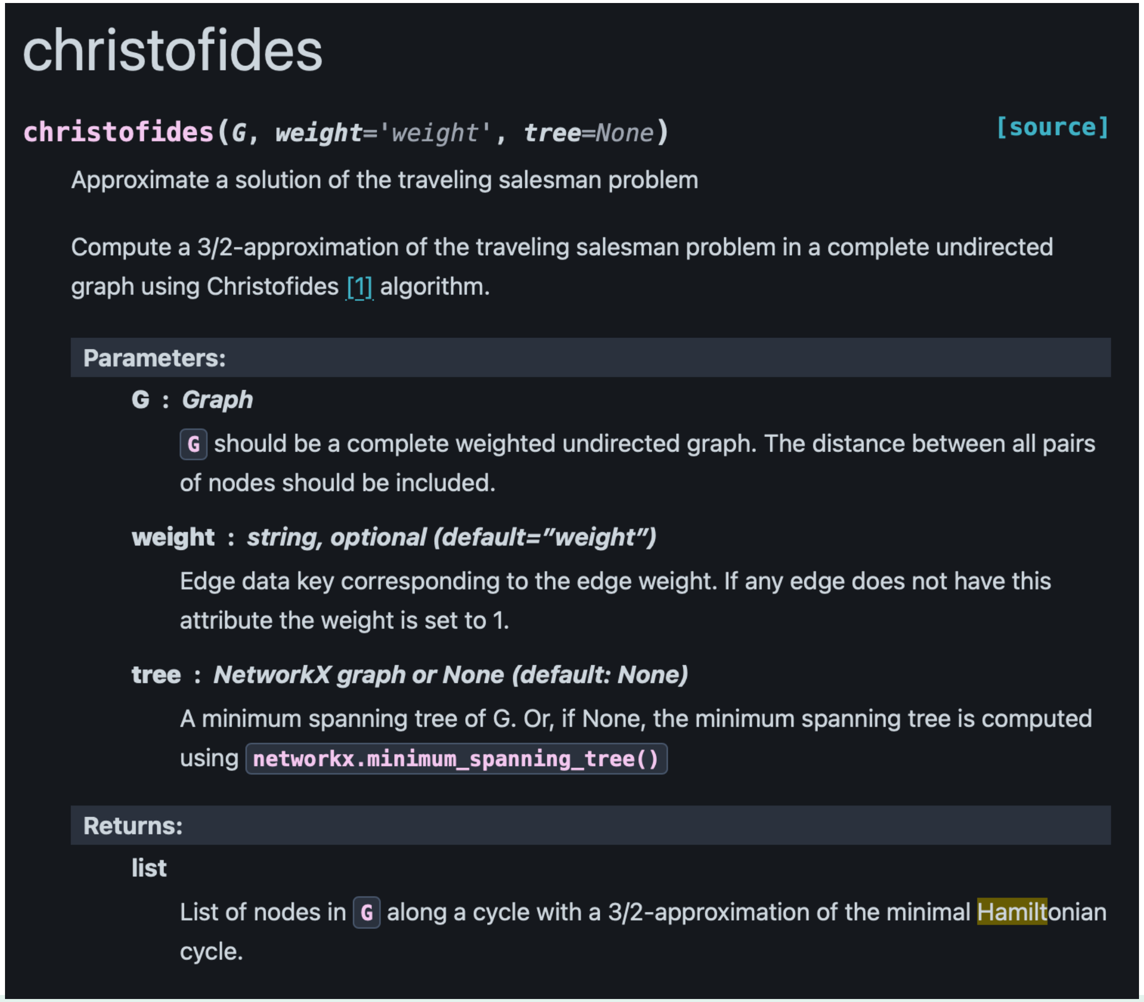
TSP is an NP-hard problem in combinatorial optimization, important in operations research and theoretical computer science.

http://en.wikipedia.org/wiki/Travelling_salesman_problem



`christofides`(G[, weight, tree])	Approximate a solution of the traveling salesman problem
`traveling_salesman_problem`(G[, weight, ...])	Find the shortest path in `G` connecting specified nodes
`greedy_tsp`(G[, weight, source])	Return a low cost cycle starting at `source` and its cost.
`simulated_annealing_tsp`(G, init_cycle[, ...])	Returns an approximate solution to the traveling salesman problem.
`threshold_accepting_tsp`(G, init_cycle[, ...])	Returns an approximate solution to the traveling salesman problem.
`asadpour_atsp`(G[, weight, seed, source])	Returns an approximate solution to the traveling salesman problem.

Christofides’ algorithm



christofides

`christofides(G, weight='weight', tree=None)` [source]

Approximate a solution of the traveling salesman problem

Compute a 3/2-approximation of the traveling salesman problem in a complete undirected graph using Christofides [1] algorithm.

Parameters:

G : ***Graph***

`G` should be a complete weighted undirected graph. The distance between all pairs of nodes should be included.

weight : ***string, optional (default=”weight”)***

Edge data key corresponding to the edge weight. If any edge does not have this attribute the weight is set to 1.

tree : ***NetworkX graph or None (default: None)***

A minimum spanning tree of G. Or, if None, the minimum spanning tree is computed using `networkx.minimum_spanning_tree()`

Returns:

list

List of nodes in `G` along a cycle with a 3/2-approximation of the minimal Hamiltonian cycle.

License

- These slides are distributed under a Creative Commons license “Attribution-NonCommercial-ShareAlike 4.0 International (CC BY-NC-SA 4.0)”
- You are free to:
  - Share — copy and redistribute the material in any medium or format
  - Adapt — remix, transform, and build upon the material
  - The licensor cannot revoke these freedoms as long as you follow the license terms.
- Under the following terms:
  - Attribution — You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.
  - NonCommercial — You may not use the material for commercial purposes.
  - ShareAlike — If you remix, transform, or build upon the material, you must distribute your contributions under th same license as the original.
  - No additional restrictions — You may not apply legal terms or technological measures that legally restrict others from doing anything the license permits.
- https://creativecommons.org/licenses/by-nc-sa/4.0/