



# Paths in graphs

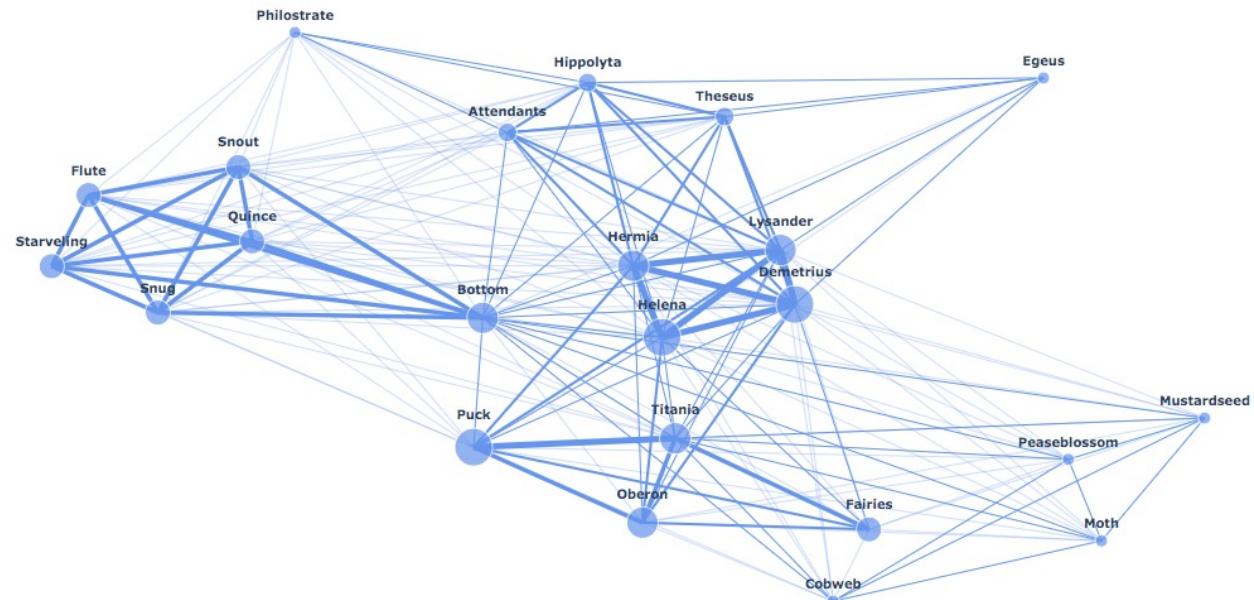
## Shortest path and cycles

Fulvio Corno

Giuseppe Averta

Carlo Masone

Francesca Pistilli





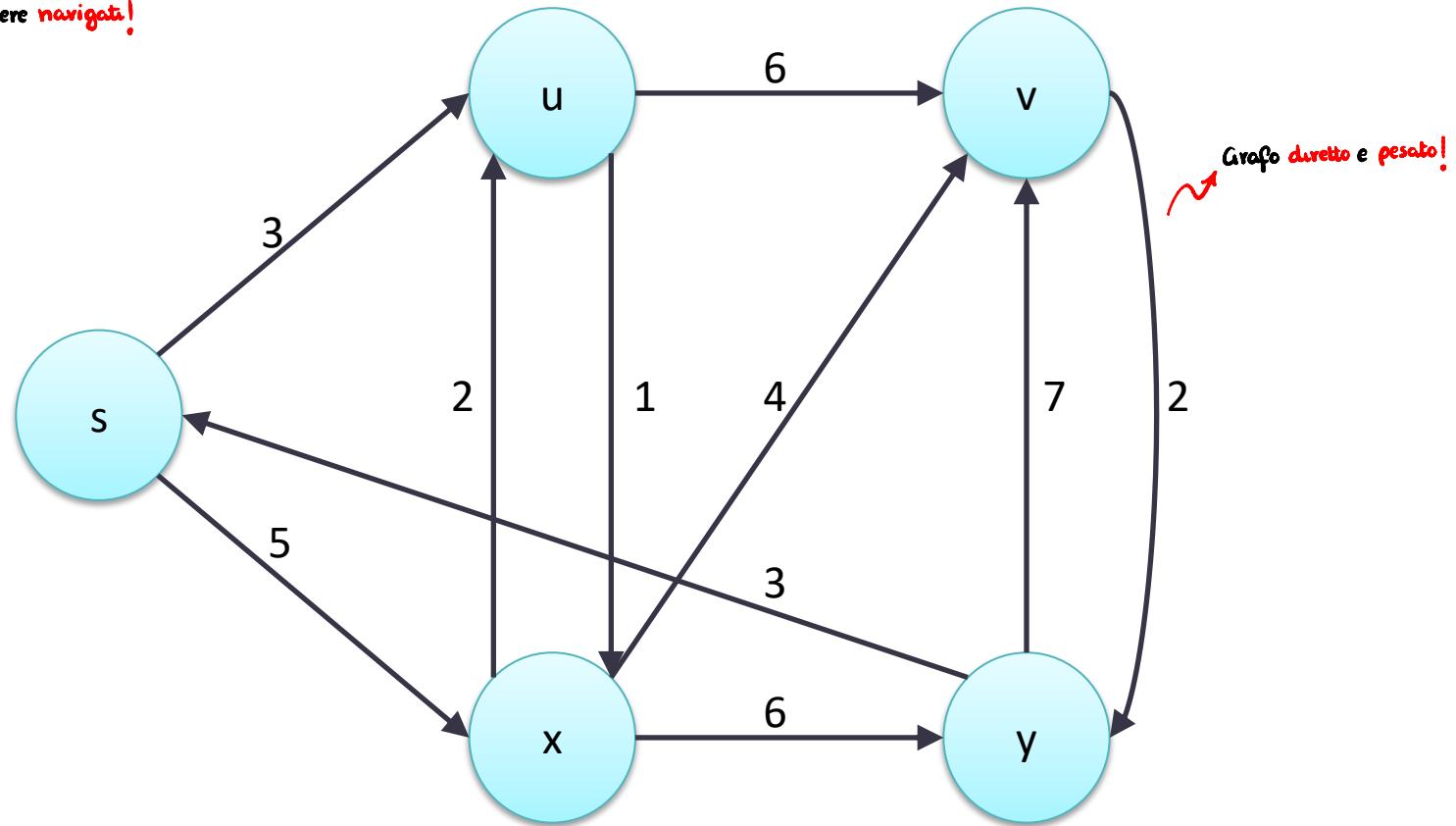
Searching for “optimal” paths between nodes

# PATHS IN GRAPHS

# Shortest Paths

What is the shortest path between s and v ?

• La cosa + comoda dei grafi è che possono essere **navigati!**



# Summary

- Shortest Paths
    - Definitions
    - Floyd-Warshall algorithm
    - Bellman-Ford-Moore algorithm
    - Dijkstra algorithm
  - Cycles
    - Definitions
    - Algorithms
- Algoritmi usati x trovare percorsi ottimi in un grafo!

# Definitions

- Graphs: Finding shortest paths

# Definition: weight of a path

- Consider a directed, weighted graph  $G=(V, E)$ , with weight function  $w: E \rightarrow \mathbb{R}$ 
  - 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

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

*Dato un cammino fra due nodi, il peso di un cammino è la somma dei pesi degli archi che compongono il cammino.*

# 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)  $d(u,v)=\infty$

Il cammino minimo tra  $u$  e  $v$  è, tra tutti i possibili cammini che collegano  $u$  e  $v$ , quello che ha il costo minimo.  
Noi definiamo il peso del cammino  $d(u,v)$  come la somma dei pesi degli archi che collegano  $u$  e  $v$ ; se non esiste un cammino che collega  $u$  e  $v$ , allora  $d(u,v) = \infty$

# 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

• Due tipi di algoritmi:  
• **Single-source**: dato due nodi, calcola il cammino minimo fra i nodi; dato un nodo, calcola il cammino minimo verso tutti gli altri nodi.  
• **all-pairs**: dato un grafo, restituisce i cammini minimi fra tutte le coppie di nodi.

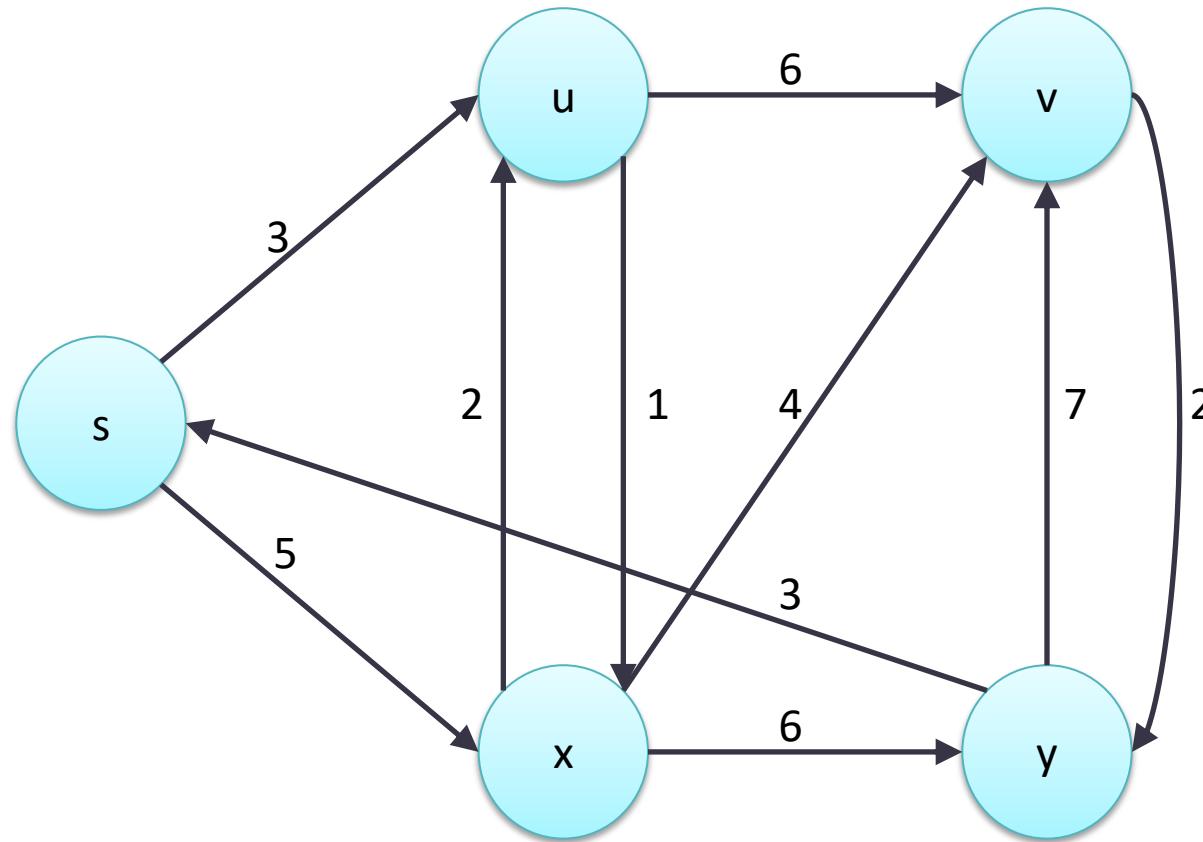
# What to find?

- 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

• Possiamo cercare solo il **costo minimo** del cammino; in altri casi potrei voler conoscere nodi e archi del cammino minimo!

# Example

What is the shortest path between s and v ?



# Representing shortest paths

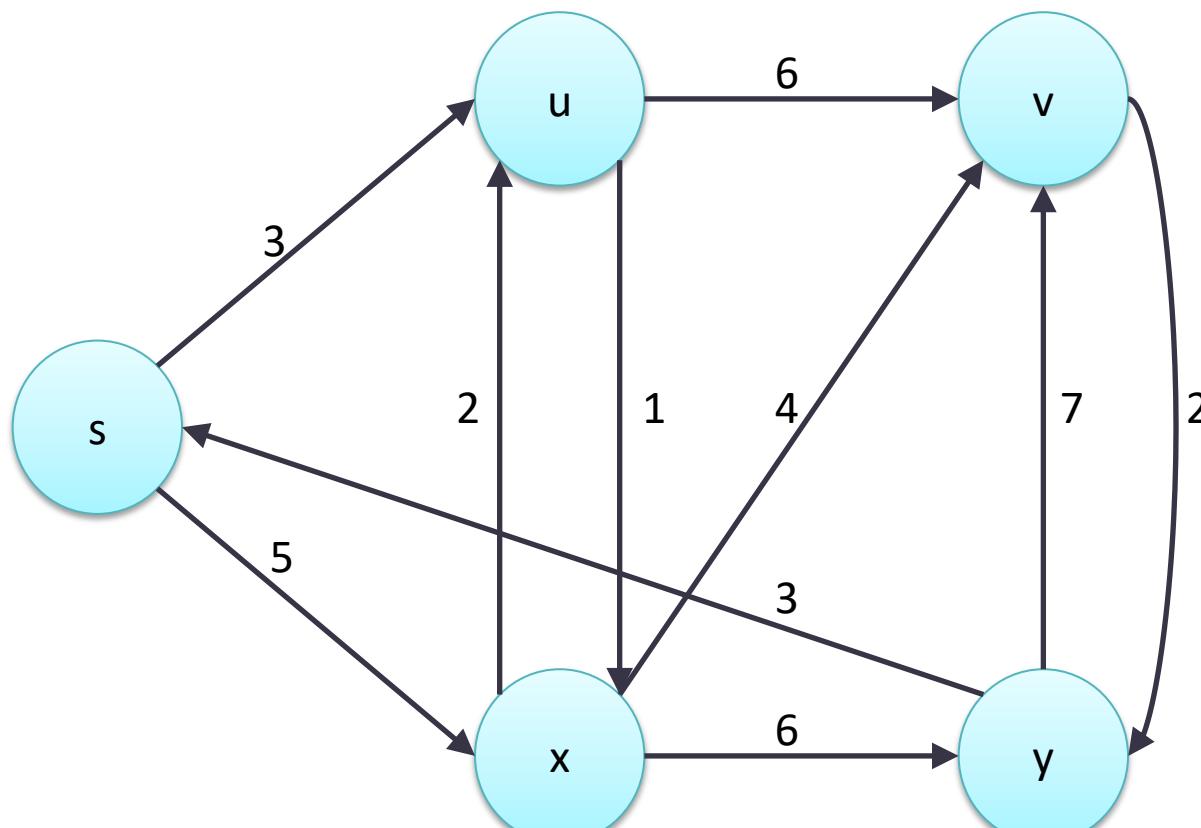
- A data structure to represent all shortest paths from a single source  $u$ , may include
  - For each vertex  $v$ , the weight of the shortest path  $d(u,v)$  (double)
  - For each vertex  $v$ , the “preceding” vertex  $p(v)$  that allows to reach  $v$  in the shortest path (object)
    - For multigraphs, we need the preceding edge

Per rappresentare uno shortest path, possiamo utilizzare diversi modi:  
• dato un vertice  $u$  (source), mi salvo in una lista i costi minimi per andare verso gli altri nodi.  
• se voglio oltre ai costi anche i diversi percorsi, dovremo usare un dizionario in cui avrò, per ogni nodo  $v$ , qual è il predecessore di  $v$  che appartiene al cammino minimo.  
Questi metodi non funzionano sui multi grafi, dato che potrei avere tra gli stessi nodi più archi.

# Example

$\pi$  è la struttura dati che salva il percorso del cammino minimo; si tratta di un dizionario in cui  $\forall$  nodo come chiave, il valore sarà il suo nodo predecessore.

$\delta$  rappresenta i costi: il costo associato al nodo  $n$  è il costo del cammino minimo considerando come nodo source il nodo  $s$ !



| $\pi$ | Vertex | Previous |
|-------|--------|----------|
|       | s      | NULL     |
|       | u      | s        |
|       | x      | u        |
|       | v      | x        |
|       | y      | v        |

| $\delta$ | Vertex | Weight |
|----------|--------|--------|
|          | s      | 0      |
|          | u      | 3      |
|          | x      | 4      |
|          | v      | 8      |
|          | y      | 10     |

# Lemma

Dobbiamo ricordarci che il vertice precedente in un algoritmo di ricerca di un cammino minimo non dipende dal nodo target. Perchè se il costo minimo per arrivare a un nodo s è x, sarà x indipendentemente dai nodi intermedi che attraverso per arrivare al nodo s.  
Questo ci permette di "spezzare" il problema e di risolverlo per step.  
Se p1 fa parte dello shortest path tra u e v1, e p2 è lo shortest path tra u e v2, allora se prendo un nodo che fa parte di entrambi i path w, allora il valore dello cammino minimo da u a w è indifferente che io utilizzi il cammino p1 o il cammino p2.

- 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 unique way and still guarantees that both  $p_1$  and  $p_2$  are shortest

# Shortest path graph

Quando noi cerchiamo lo shortest path tra il nodo source e il nodo target di fatto stiamo costruendo un albero orientato dal source verso il target, con la garanzia che i percorsi su questo albero siano quelli a costo minimo. Consideriamo un nodo source  $u$  e assumiamo di avere un algoritmo che mi riesca a trovare tutti i cammini minimi da  $u$  a tutti gli altri nodi del grafo; possiamo definire il vettore  $E_p$  che contiene le tuple  $(v.\text{preceding}, v)$ . L'insieme degli archi  $E_p$  sarà sicuramente un sottoinsieme dell'insieme contenente tutti gli archi del grafo  $E$ . I nodi di questo albero di visita saranno tutti i nodi del grafo che sono raggiungibili dal nodo source. Si può dire che  $G_p$ , il nostro albero di visita, è un sottografo del grafo di partenza  $G$ .

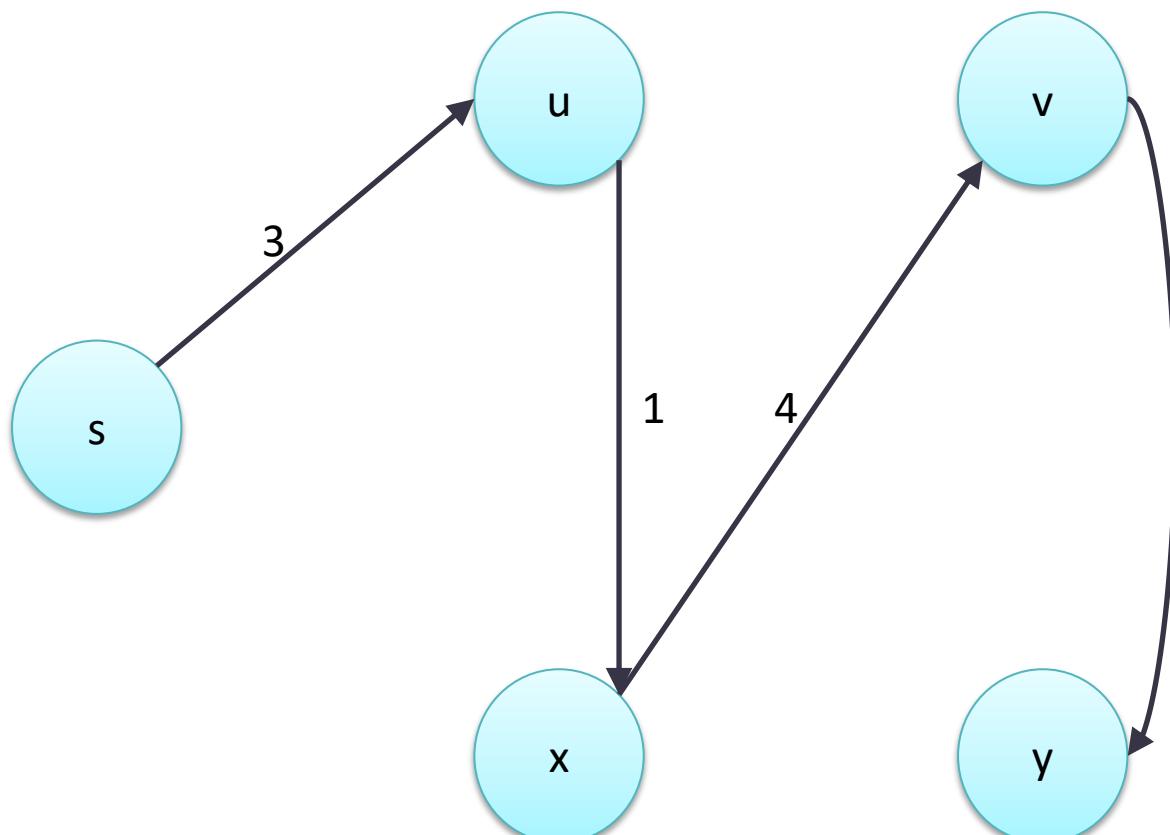
- Consider a source node  $u$
- Compute all shortest paths from  $u$
- Consider the relation  $E_p = \{ (v.\text{preceding}, v) \}$
- $E_p \subseteq E$
- $V_p = \{ v \in V : v \text{ reachable from } u \}$
- $G_p = G(V_p, E_p)$  is a subgraph of  $G(V, E)$
- $G_p$ : the predecessor-subgraph

# Shortest path tree

G<sub>p</sub> è un albero che ha un come radice il nodo u.  
G<sub>p</sub> ha la proprietà che tutti i percorsi che hanno come source u e target un qualsiasi altro nodo del grafo, saranno i percorsi minimi possibili.  
G<sub>p</sub> non è univoco; possono esistere più sottografi contenenti i cammini minimi da u a tutti gli altri nodi, ma se esistono altri alberi di cammini minimi, allora il peso dei diversi cammini minimi sarà sempre uguale anche negli altri sottografi.

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

# Example



| $\pi$ | Vertex | Previous |
|-------|--------|----------|
|       | s      | NULL     |
|       | u      | s        |
|       | x      | u        |
|       | v      | x        |
|       | y      | v        |

| $\delta$ | 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

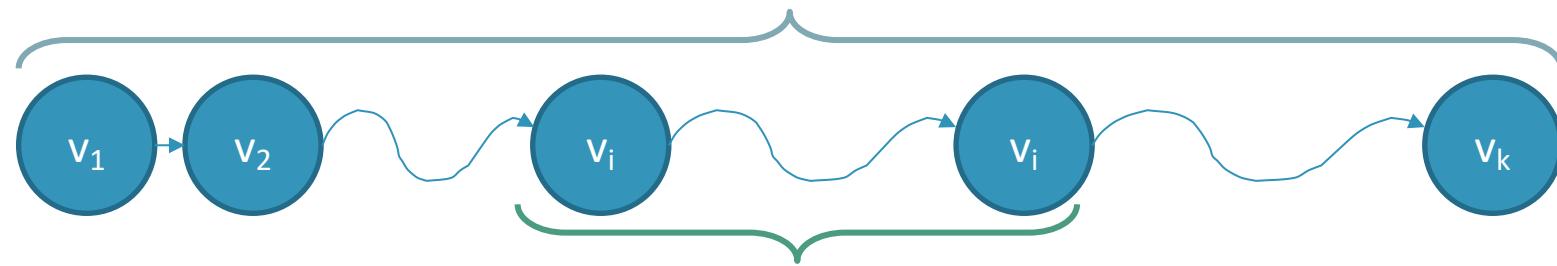


Se il grafo è **NON pesato**, i cammini minimi tra un nodo source e tutti gli altri nodi possono essere calcolati con un **algoritmo BFS**, perché il BFS mi restituisce i cammini minimi in termini di numero di archi.

# Lemma

Considerando un grafo ordinato e pesato  $G$ , e considerato un path che già sappiamo che è ottimo tra  $v_1$  e  $v_k$  che chiameremo  $p$ , allora presi a caso due nodi di quel path,  $v_i$  e  $v_j$ , il percorso tra quei i due nodi  $v_i$  e  $v_j$  che è compreso nel percorso generale  $p$ , tra  $v_1$  e  $v_k$ , sarà sicuramente il percorso di costo minimo.

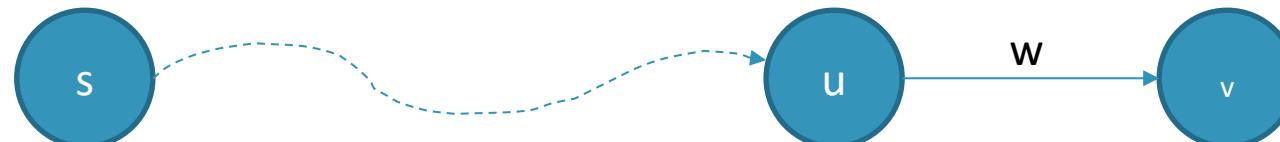
- Consider an ordered weighted graph  $G=(V,E)$ , with weight function  $w: E \rightarrow \mathbb{R}$ .
- Let  $p = \langle v_1, v_2, \dots, v_k \rangle$  a shortest path from vertex  $v_1$  to vertex  $v_k$ .
- For all  $i,j$  such that  $1 \leq i \leq j \leq k$ , let  $p_{ij} = \langle v_i, v_{i+1},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



# Corollary

Proprietà **IMPORTANTE**: sia  $p$  il percorso minimo tra i nodi  $s$  e  $v$ , e considerato il nodo  $u$  che è il penultimo del percorso (ovvero il predecessore del nodo target), allora io posso disaccoppiare il costo di questo cammino come il costo per arrivare da  $s$  a  $u$ , + il costo dell'arco tra  $u$  e  $v$ . =>  $d(s,v) = d(s,u) + w(u,v)$ !

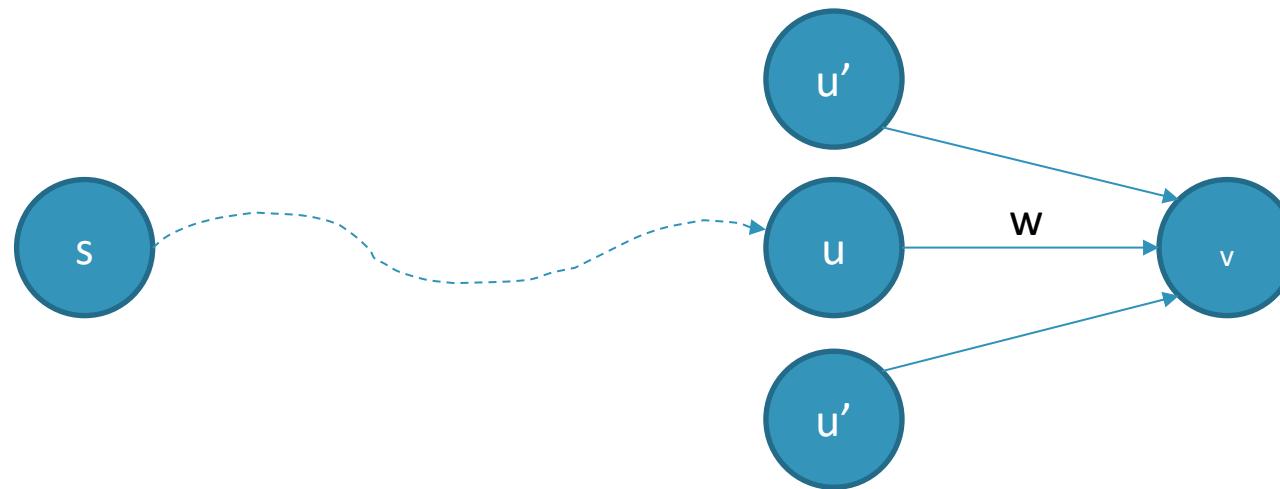
- Let  $p$  be a shortest path from  $s$  to  $v$
- Consider the vertex  $u$ , such that  $(u,v)$  is the last edge in the shortest path
- We may decompose  $p$  (from  $s$  to  $v$ ) into:
  - A sub-path from  $s$  to  $u$
  - The final edge  $(u,v)$
- Therefore
- $d(s,v) = d(s,u) + w(u,v)$



# Lemma

Se sceglieremo arbitrariamente un altro nodo connesso a  $v$ , che chiamiamo  $u'$ , allora sicuramente il costo minimo per arrivare da  $s$  a  $v$  mediante il nodo  $u$ , sarà più piccolo (o al massimo uguale) al costo del cammino minimo tra  $s$  e  $u'$  + il costo dell'arco tra  $u'$  e  $v$ .

- 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

Tutti gli algoritmi che calcolano i cammini minimi nei grafi si basano sulla Relaxation: cosa vuol dire Relaxation?  
Vuol dire che dato un vettore  $d[u]$  che mi salva i costi minimi io, quando sto ciclando con il mio algoritmo, vado a tener traccia del costo minimo che ho trovato prima; poi provo una strada diversa cambiando arco e vado a vedere se il nuovo percorso aggiungendo un arco o cambiando l'arco con il percorso minimo attuale, mi permette di abbassare il mio costo.  
Tutti gli algoritmi si basano su questo principio.

- 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  $v \in V$
  - do
    - $d[v] \leftarrow \infty$
    - $p[v] \leftarrow \text{NIL}$
  - $d[s] \leftarrow 0$

**Inizializzazione** di tutti gli algoritmi:

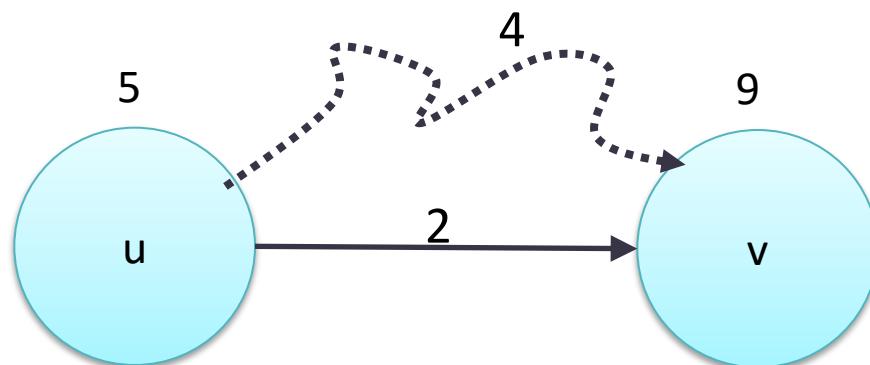
- distanza tra source e source è zero;
- distanza tra source e qualsiasi altro nodo è  $\infty$ .

# Relaxation

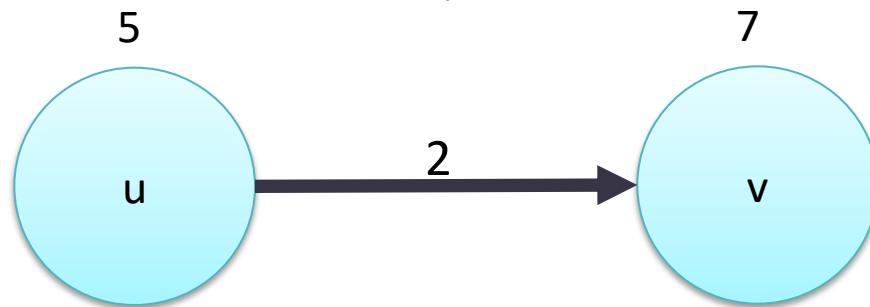
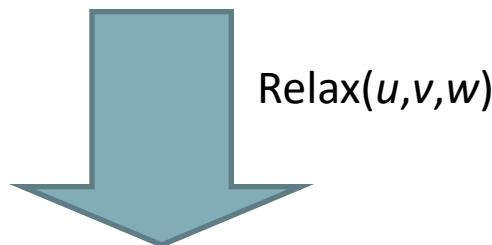
- We consider an edge  $(u,v)$  with weight  $w$
- $\text{Relax}(u, v, w)$ 
  - if  $d[v] > d[u] + w(u,v)$
  - then
    - $d[v] \leftarrow d[u] + w(u,v)$
    - $p[v] \leftarrow u$

• considerato un arco tra  $u$  e  $v$  di peso  $w$ , verifico se il **cammino minimo** fino a  $v$  calcolato fino a quest'iterazione è maggiore del cammino minimo fino a  $v$  calcolato fino a quest'iterazione + il peso dell'arco tra  $u$  e  $v$ . Se così fosse, imposto  $d[v] = d[u] + w(u,v)$  come nuovo cammino minimo fino a  $v$ , e  $p[v] = u$  ( $u$  come predecessore di  $v$ ) ; altrimenti non faccio nulla!

# Example 1

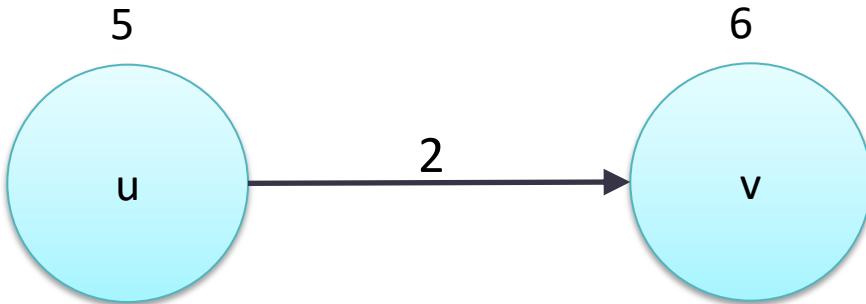
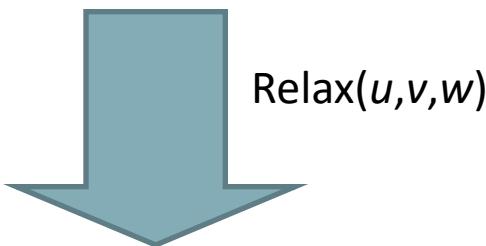
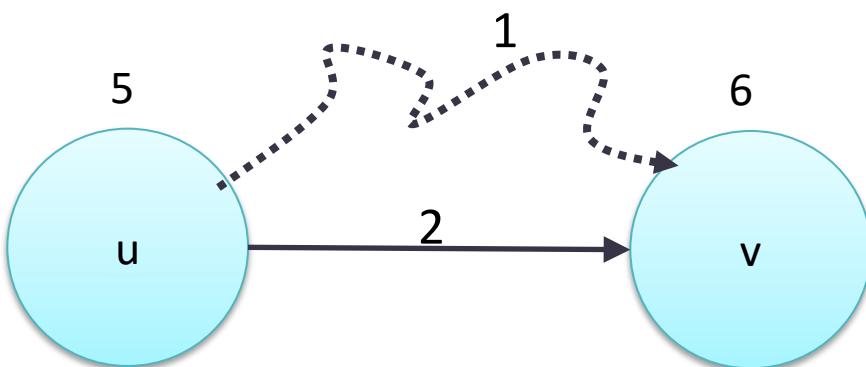


Before:  
Shortest known path to  $v$   
weights 9, does not  
contain  $(u,v)$



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

Consideriamo un grafo ordinato e pesato  $G$ . Prendiamo un arco tra  $u$  e  $v$   $(u,v)$  appartenente al grafo.  
DOPO la Relaxation, possiamo affermare SICURAMENTE che  $d(v) \leq d(u) + w(u,v)$  !

- Consider an ordered weighted graph  $G=(V, E)$ , with weight function  $w: E \rightarrow \mathbb{R}$ .
- 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

Se il nostro grafo  $G$  non ha cicli con peso sommato negativo, allora dopo aver inizializzato l'algoritmo di ricerca (considerando come nodo source  $s$ ), il grafo dei predecessori  $G_p$  è un albero che avrà come nodo radice  $s$ .  
Qualsiasi update noi facciamo sui cammini, non potrà cambiare questa proprietà, quindi l'albero rimarrà sempre incardinato in  $s$ .

- 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  $G_p$  is a rooted tree, with  $s$  as the root.
  - Any relaxation we may apply to the graph does not invalidate this property.

# Lemma

Date queste definizioni che abbiamo visto fin'ora, assumendo qualsiasi possibile operazione di Relaxation, per ogni vertice  $v$ , il costo ottenuto per arrivare da  $s$  a un qualsiasi nodo del grafo  $v$  sarà  $\leq$  rispetto a un qualsiasi altro percorso del grafo fatto per arrivare da  $s$  a  $v$ .

- 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

• Gli algoritmi x trovare i cammini minimi sono già implementati!

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

→ Documentazione di networkX sui shortest paths!



Graphs: Finding shortest paths

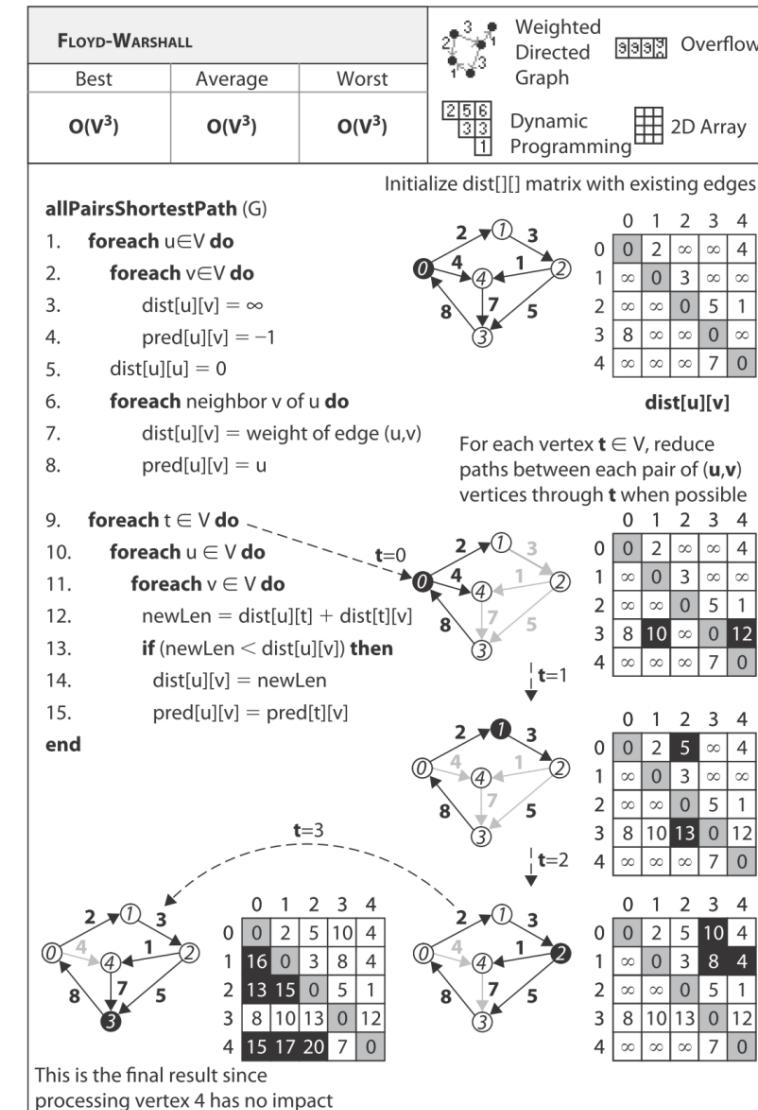
# FLOYD-WARSHALL ALGORITHM

# Floyd-Warshall algorithm

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

L'algoritmo di Floyd-Warshall è un metodo per la soluzione al problema all-source shortest path, quindi è un algoritmo che ci consente di calcolare tutti i cammini minimi a partire da un nodo qualunque del grafo. Io avrò il mio grafo con il mio set di nodi, e in questo modo sarò in grado di identificare il costo per andare da qualsiasi nodo a qualsiasi altro nodo.

- $\text{dist}[i][j]$  sarà il costo del percorso minimo da  $i$  a  $j$  e sarà una matrice, perché calcolo il costo del percorso da un nodo a tutti gli altri nodi.
- se la distanza tra due nodi è infinita, significa che non esiste un percorso tra i due nodi.
- con  $\text{pred}[s][j]$  è un dizionario in cui associamo a ogni nodo il suo predecessore. Da  $\text{dist}$  otteniamo le distanze dei percorsi minimi, da  $\text{pred}$  otteniamo gli effettivi percorsi minimi.



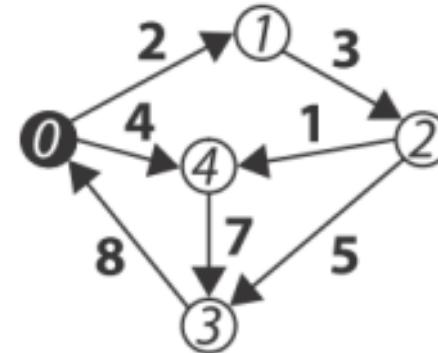
# Floyd-Warshall: initialization

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

    #Initialise
    for u in V:
        for v in V:
            dist[u][v] =  $\infty$       } Creo la matrice delle distanze fra i nodi: imposto la distanza
            pred[u][v] = None       } tra due nodi + a  $\infty$  e la distanza tra un nodo e se stesso
            dist[u][u] = 0           } pari a 0!

    #Set distances with existing edges
    for n in neighborhood(u):
        dist[u][n] = w(u,n)      } Per tutti i nodi vicini al nodo u (collegati da un
        pred[u][n] = u           } arco), imposto la distanza tra u e il nodo pari al
                                } peso dell'arco!
```

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



|   | 0        | 1        | 2        | 3        | 4        |
|---|----------|----------|----------|----------|----------|
| 0 | 0        | 2        | $\infty$ | $\infty$ | 4        |
| 1 | $\infty$ | 0        | 3        | $\infty$ | $\infty$ |
| 2 | $\infty$ | $\infty$ | 0        | 5        | 1        |
| 3 | 8        | $\infty$ | $\infty$ | 0        | $\infty$ |
| 4 | $\infty$ | $\infty$ | $\infty$ | 7        | 0        |

$dict[u][v]$

# 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

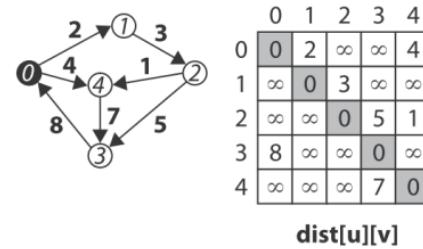
    #Set distances with existing edges
    for n in neighborhood(u):
        dist[u][n] = w(u,n)
        pred[u][n] = u

    #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] implementazione dell'algoritmo

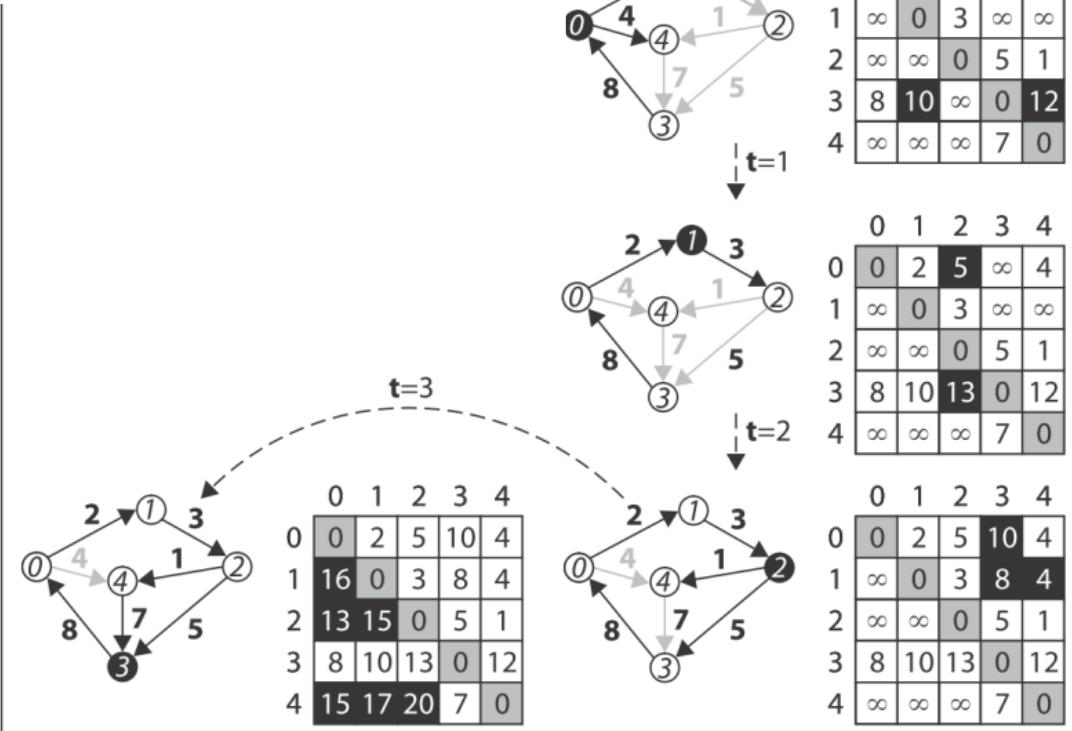
                #Check if the new path is better than previous best
                if (newDist < dist[u][v])
                    dist[u][v] = newLen
                    pred[u][v] = pred[t][v]

    return dist, pred

```



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



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:  $O(V^3)$  ↗ Algoritmo non molto efficiente! ➔ Dipende dal numero di vertici; va quindi usato solo su piccoli grafi.
- [https://algorithms.discrete.ma.tum.de/graph-algorithms/spp-floyd-warshall/index\\_en.html](https://algorithms.discrete.ma.tum.de/graph-algorithms/spp-floyd-warshall/index_en.html)

# Implementation in NetworkX

## floyd\_marshall

`floyd_marshall(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\\_marshall\\_predecessor\\_and\\_distance](#)

[floyd\\_marshall\\_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  $O(n^2)$ .

- L'algoritmo è implementato in networkX con il metodo `nx.floyd_marshall()` che riceve in input il grafo e la stringa che identifica i pesi degli archi; restituisce un dizionario di dizionari con i pesi dei cammini minimi fra i nodi. Con `dist[u][v]` accederò al costo del cammino fra u e v!



```
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_marshall(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
- $O(V \cdot E)$

L'algoritmo di Bellman-Ford-Moore è una soluzione al problema del single-source shortest path. Quindi assumendo di partire da un nodo source, mi calcola il costo dei cammini minimi dal nodo source verso tutti gli altri nodi del grafo.  
Non funziona in presenza di cicli negativi all'interno del grafo.  
La complessità è pari al numero di vertici moltiplicato per il numero di archi.  
Invece che ciclare sui nodi, ciclo sugli archi.

# 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] =  $\infty$   
        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
```

Algoritmo di Bellman-Ford!

[https://algorithms.discrete.ma.tum.de/graph-algorithms/spp-bellman-ford/index\\_en.html](https://algorithms.discrete.ma.tum.de/graph-algorithms/spp-bellman-ford/index_en.html)

# Implementation in NetworkX

- In NetworkX posso usare il metodo `all_pairs_bellman_ford_path()`, che riceve in input il grafo e una stringa che rappresenta un attributo degli archi del grafo che identifica il peso dell'arco fra 2 nodi.

Il metodo restituisce un iteratore così organizzato:

- Ogni iterazione del generatore restituisce una tupla con due elementi.
- Il primo elemento della tupla è il nodo di partenza (la "sorgente") da cui si stanno generando i percorsi minimi.
- Il secondo elemento è un dizionario che contiene tutte le informazioni sui percorsi minimi da quella sorgente a tutti gli altri nodi nel grafo. In questo dizionario, le chiavi sono i nodi di destinazione e i valori sono i percorsi minimi rappresentati come liste di nodi.

- Invece, il metodo `nx.bellman_ford_path_length()` riceve il grafo, il nodo source e il target e restituisce il costo del cammino minimo tra i due nodi!

**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_marshall`, `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

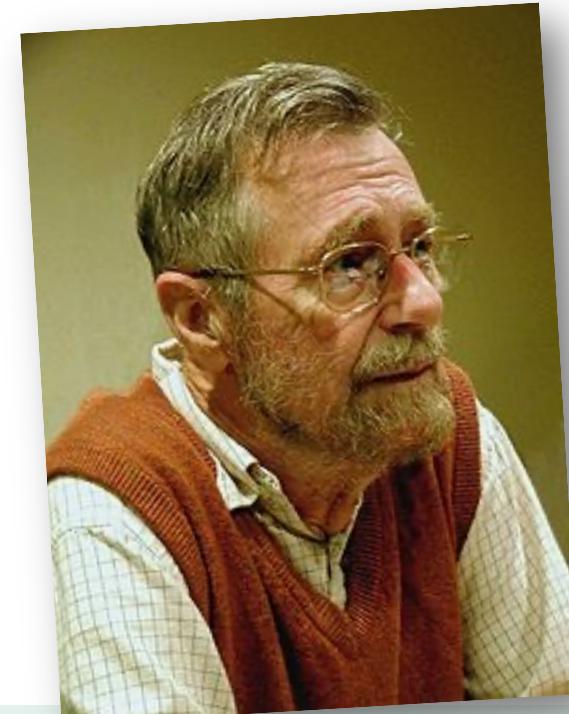
• ALGORITMO + EFFICIENTE

- 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!

Fisso un nodo source e calcolo  
il cammino minimo verso gli  
altri nodi.

Funziona sia su grafi diretti che indiretti!

Tutti gli archi devono avere  
pesi positivi!



# 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 infinity
        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)

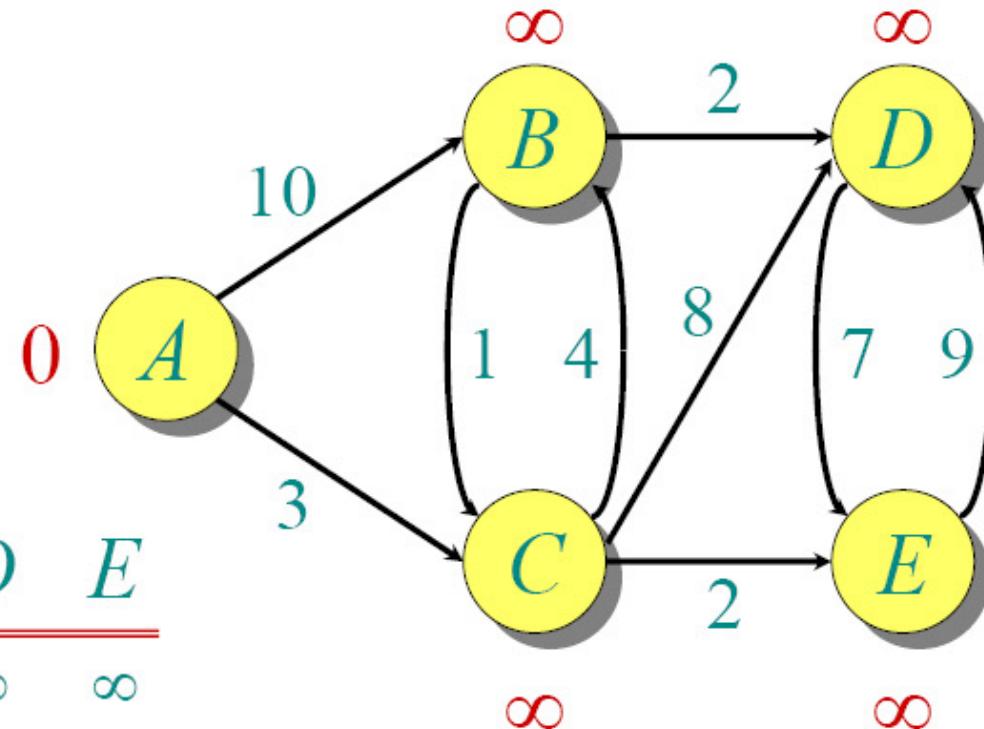
        for v in neighborhood(u) still in Q:    # Cycle on neighbors of k
            newDist = dist[u] + w(u,v)          # Verify if path through u-v is better
            if newDist < dist[v]:
                dist[v] = newDist              # Update the new shortest path
                prev[v] = u

    return dist, pred
```

# Dijkstra Animated Example

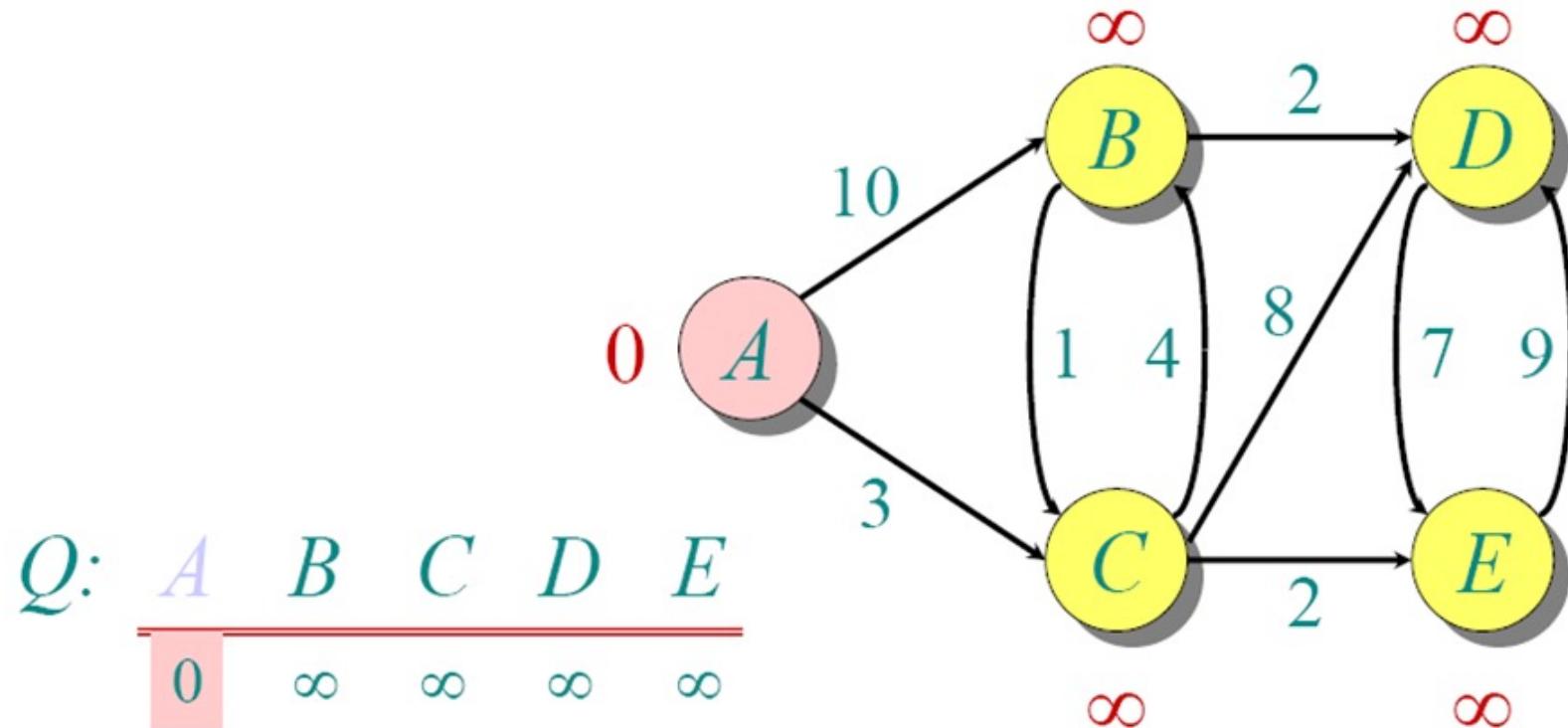
**Initialize:**

$$Q: \begin{array}{ccccc} A & B & C & D & E \\ \hline 0 & \infty & \infty & \infty & \infty \end{array}$$

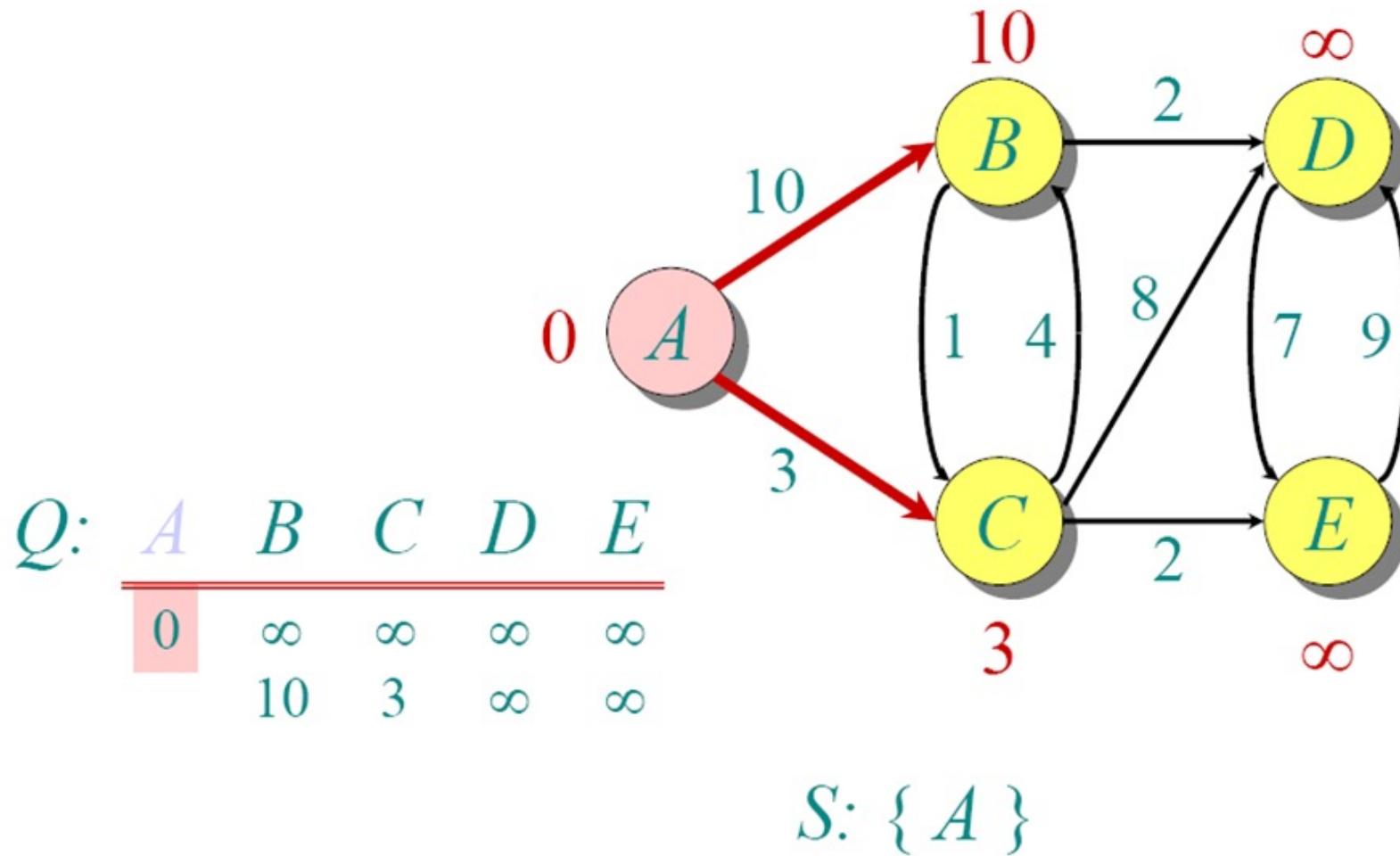


$S: \{\}$

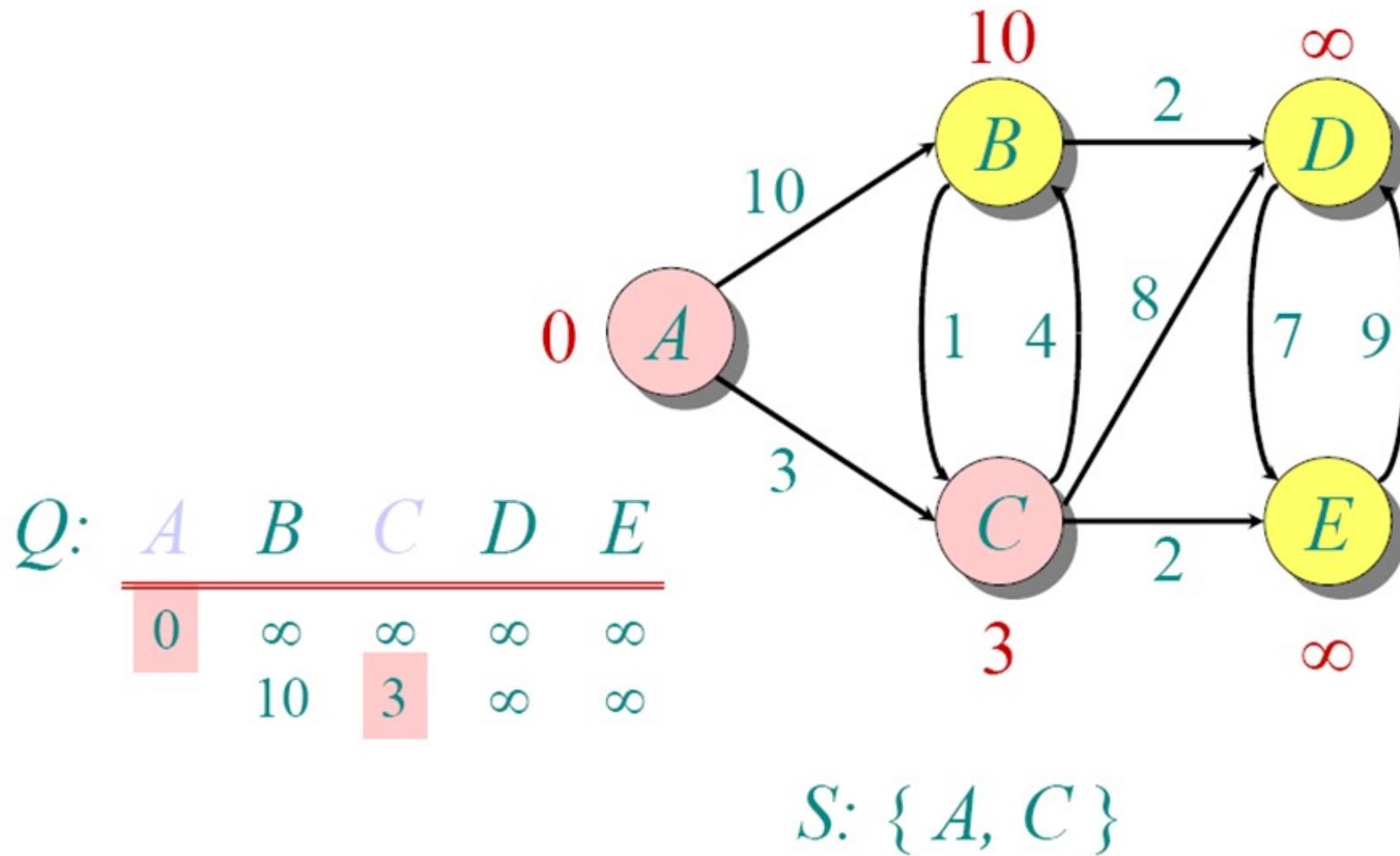
# Dijkstra Animated Example



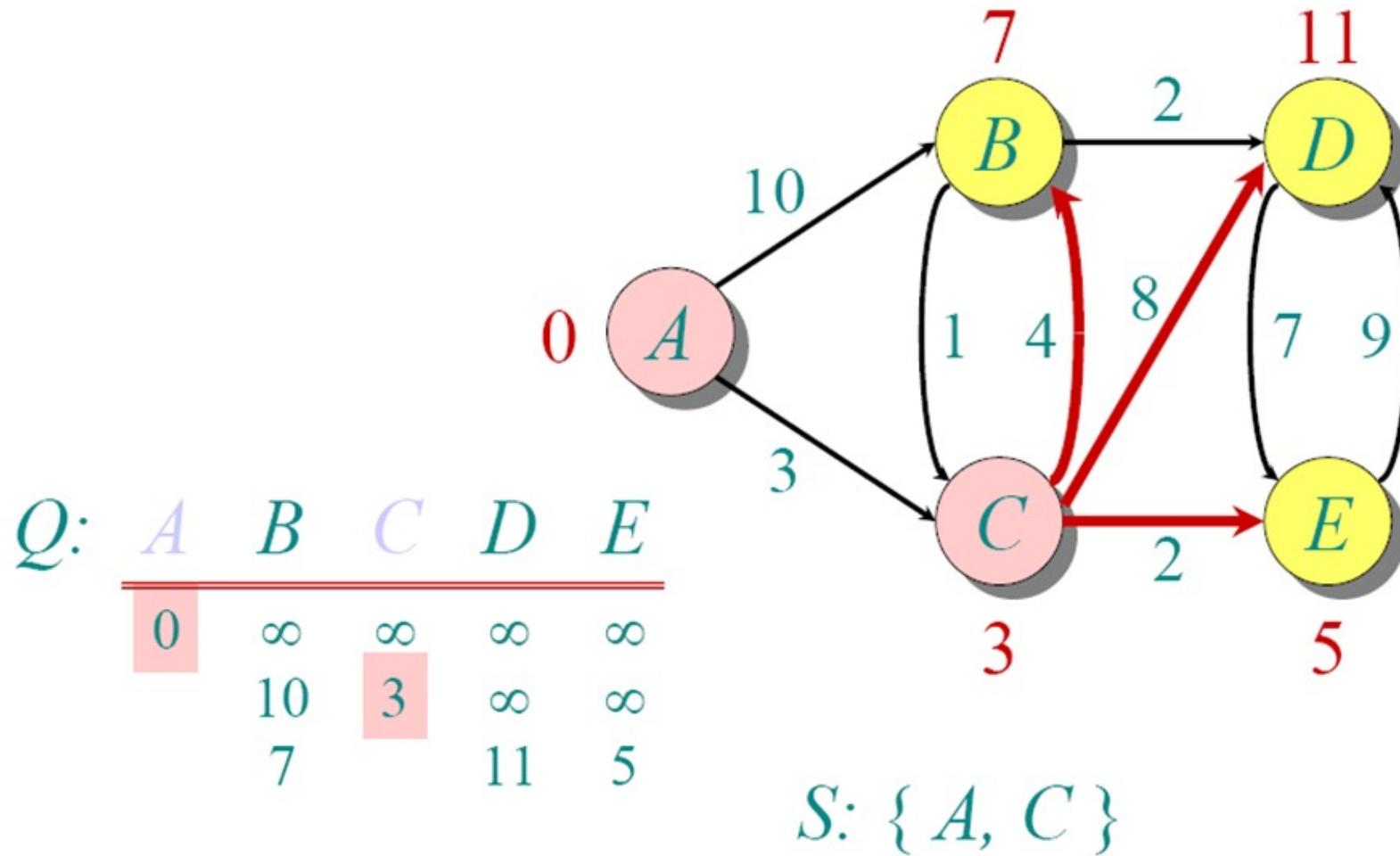
# Dijkstra Animated Example



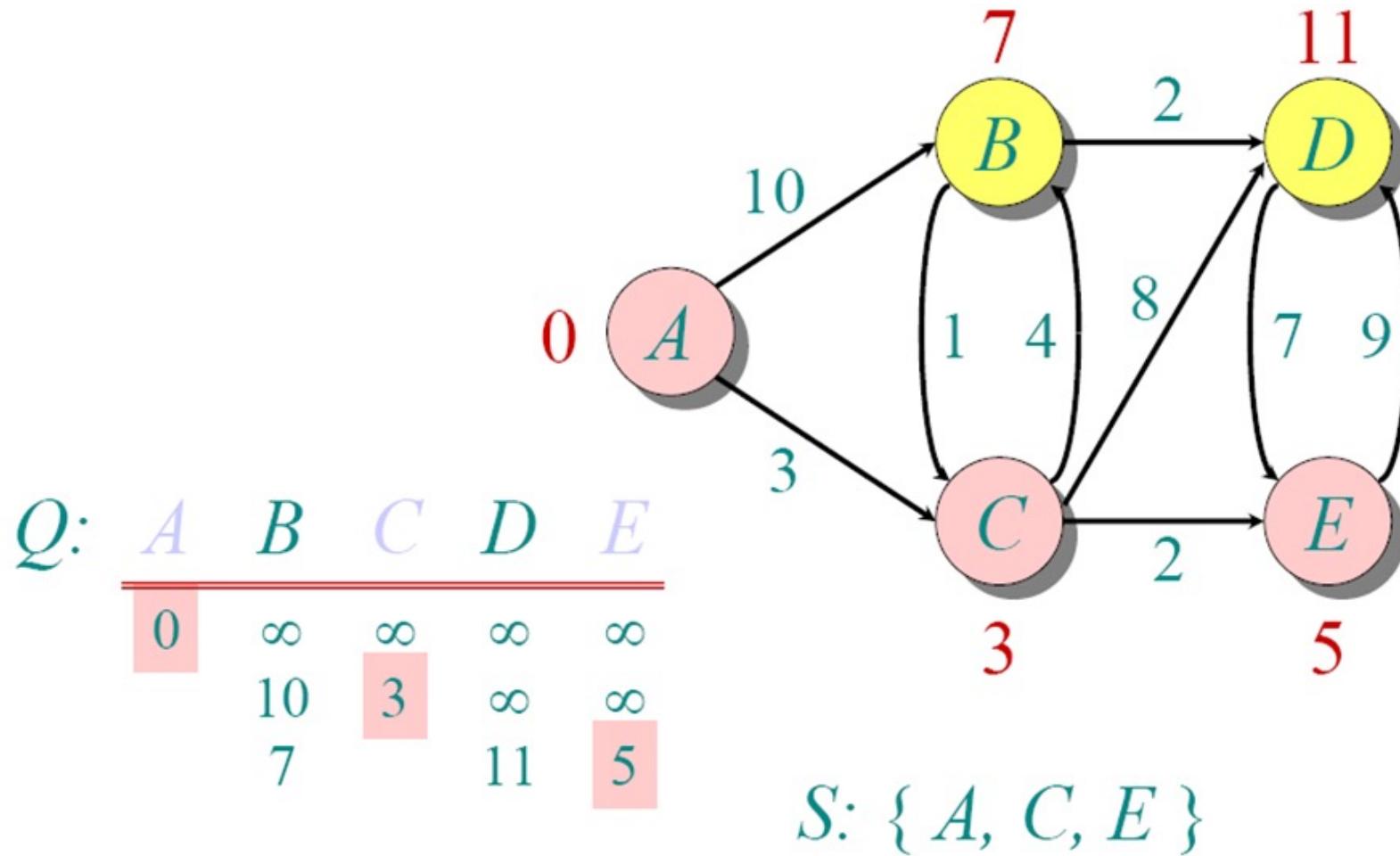
# Dijkstra Animated Example



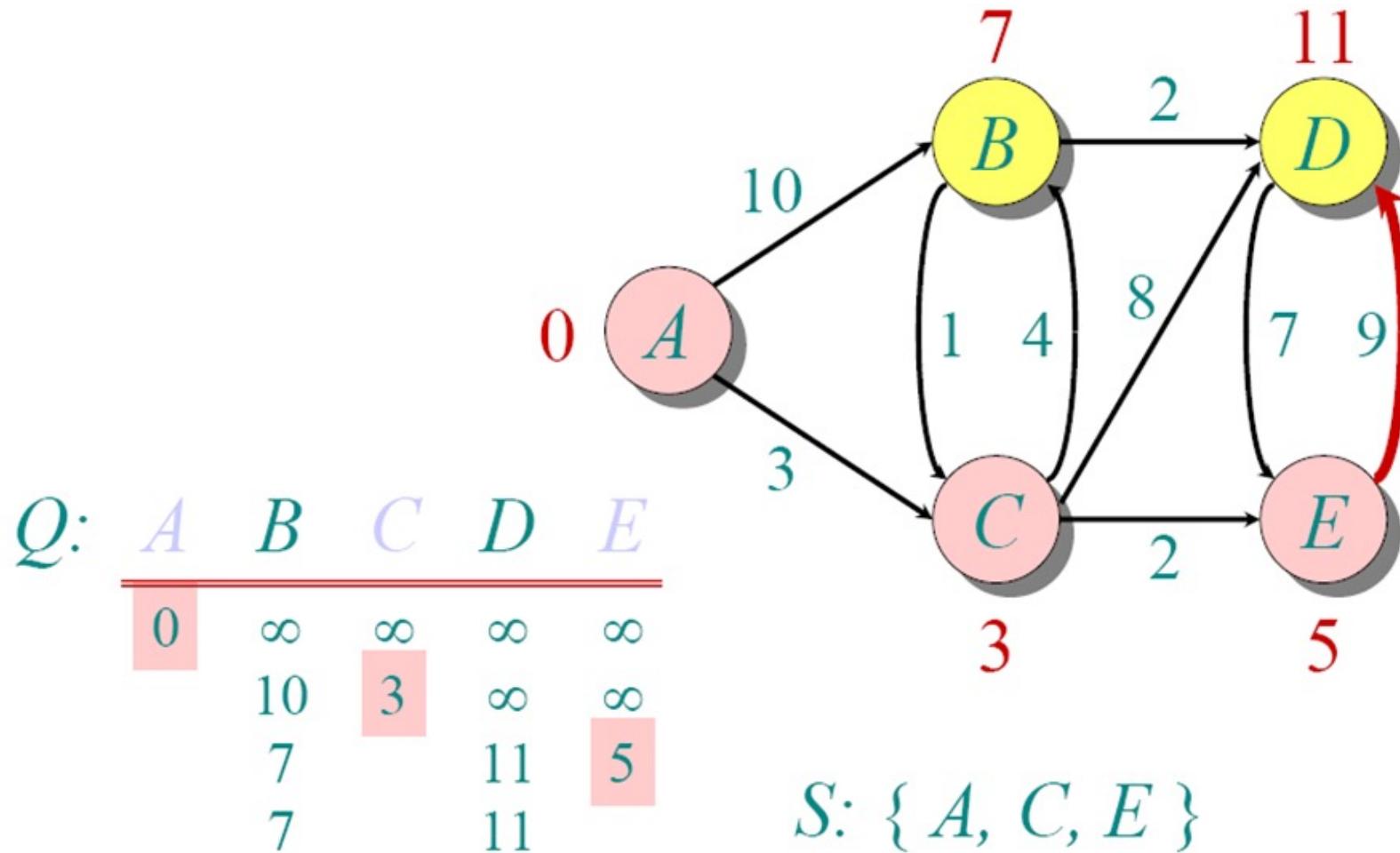
# Dijkstra Animated Example



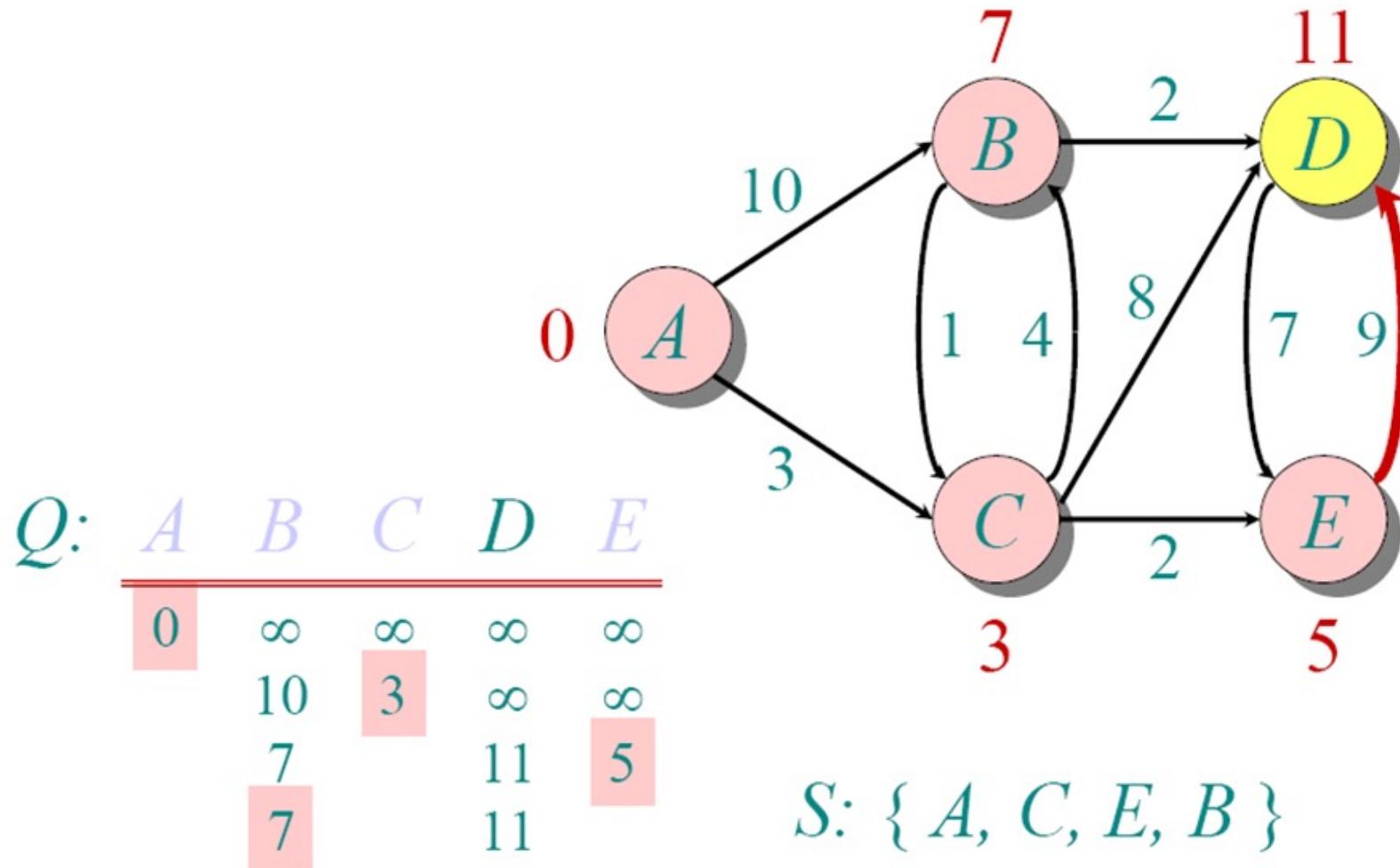
# Dijkstra Animated Example



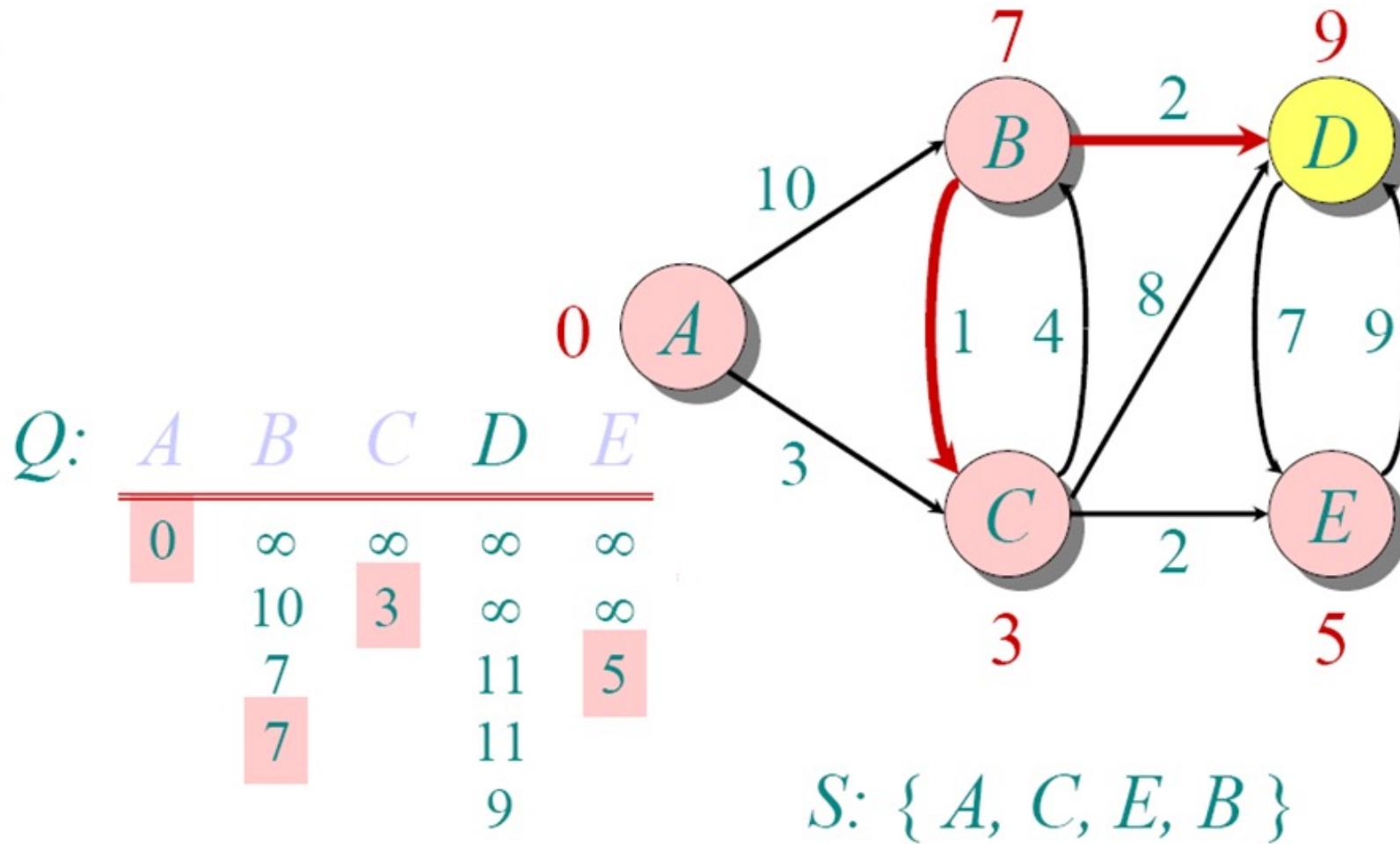
# Dijkstra Animated Example



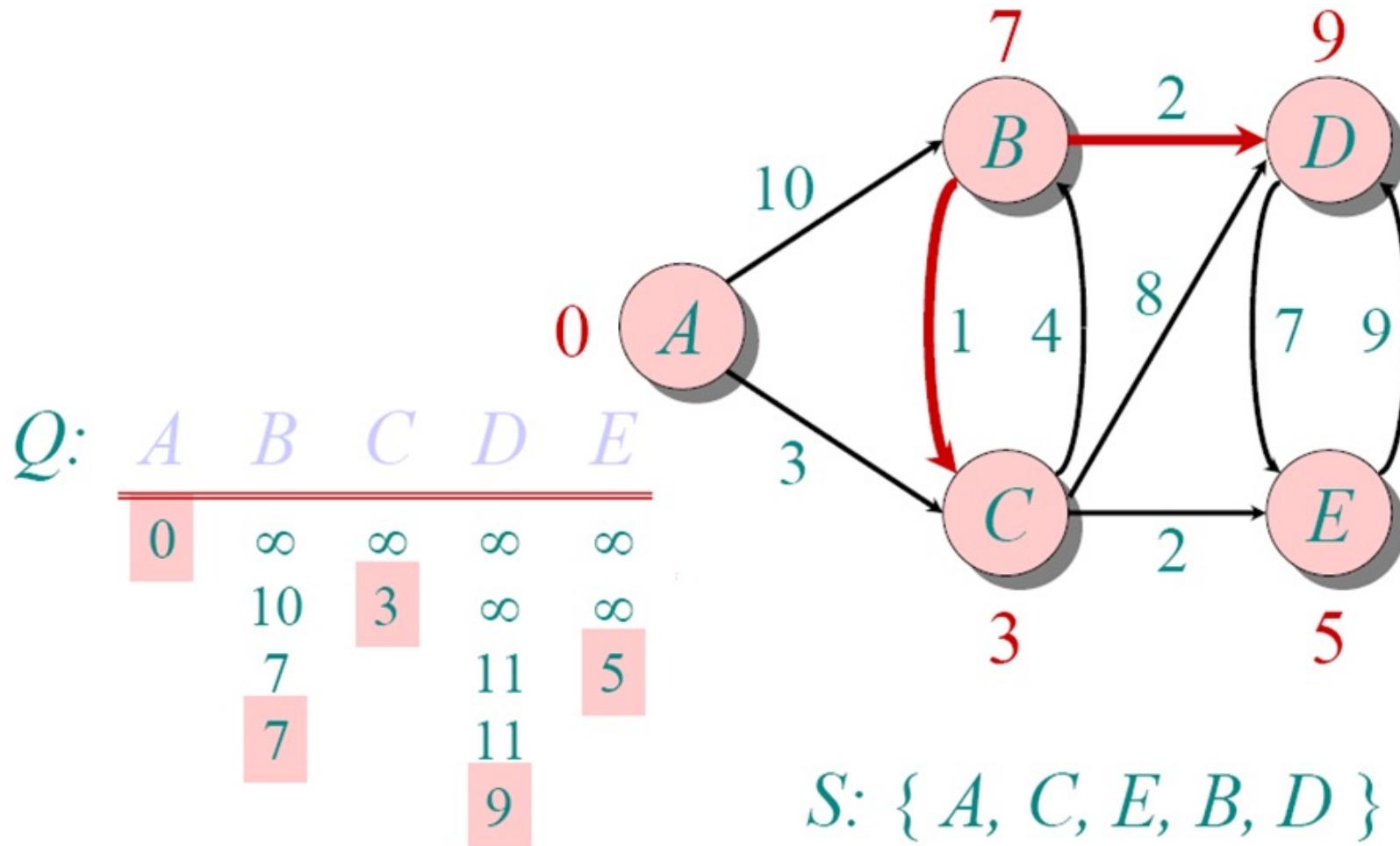
# Dijkstra Animated Example



# Dijkstra Animated Example



# Dijkstra Animated Example



# Dijkstra efficiency

- The simplest implementation is:
- $O(E + V^2)$

• Dijkstra è l'algoritmo più efficiente per la ricerca dei cammini minimi.

- But it can be implemented more efficiently:
- $O(E + V \cdot \log V)$



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

# 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.

- `dijkstra_path()` riceve il grafo, il nodo source e il nodo target e restituisce la lista di nodi attraversati nel cammino minimo.
- `single_source_dijkstra_path()`: riceve il grafo, il nodo source e restituisce un dizionario in cui le chiavi sono i nodi target e i valori il costo del cammino minimo dal source fino al nodo target rispettivo.

• `dijkstra_path.length()`: riceve il grafo, il nodo source e il nodo target e restituisce il peso del cammino minimo fra source e target.

## single\_source\_dijkstra()

### Parameters:

`G : NetworkX graph`

`source : node label`

Starting node for path

`target : node label, optional`

Ending node for path

`cutoff : integer or float, optional`

Length (sum of edge weights) at which the search is stopped. If cutoff is provided, only return paths with summed weight <= cutoff.

`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:

`distance, path : pair of dictionaries, or numeric and list.`

If target is None, paths and lengths to all nodes are computed. The return value is a tuple of two dictionaries keyed by target nodes. The first dictionary stores distance to each target node. The second stores the path to each target node. If target is not None, returns a tuple (`distance, path`), where distance is the distance from source to target and path is a list representing the path from source to target.

### 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.

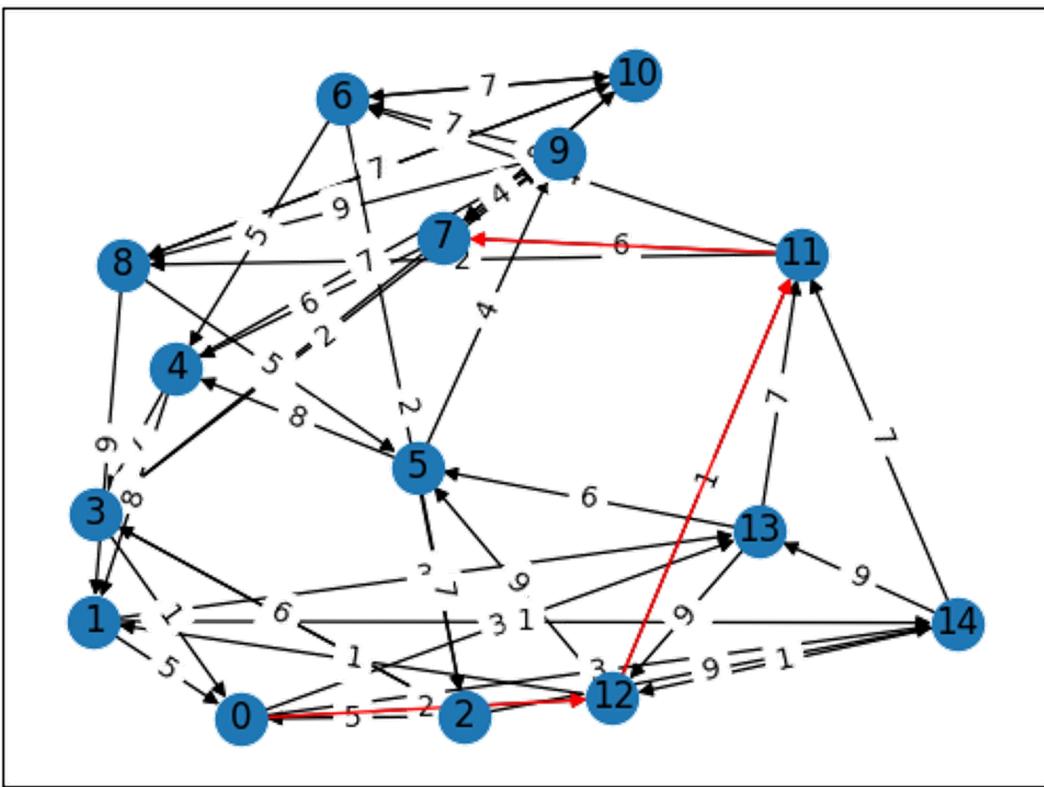
```
>>> def func(u, v, d):
...     node_u_wt = G.nodes[u].get("node_weight", 1)
...     node_v_wt = G.nodes[v].get("node_weight", 1)
...     edge_wt = d.get("weight", 1)
...     return node_u_wt / 2 + node_v_wt / 2 + edge_wt
```

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

- Tendenzialmente, a meno che non ci siano archi a costo negativo nel grafo, va usato Dijkstra!

| 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

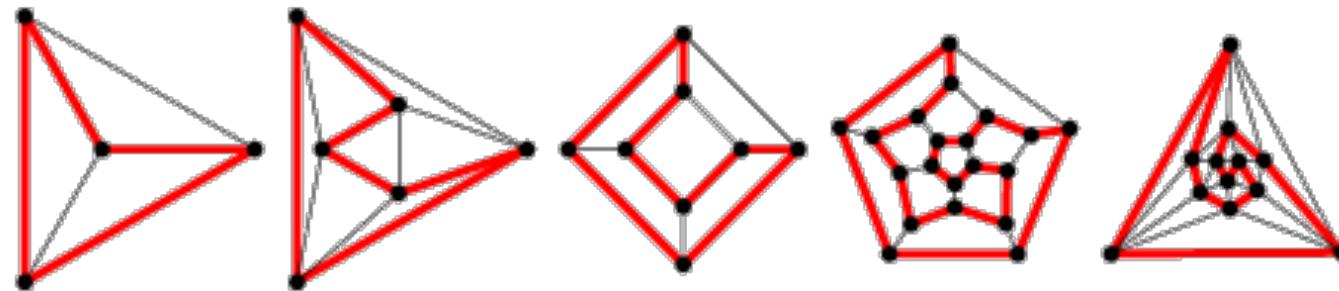
- A cycle of a graph, sometimes also called a circuit, is a subset of the edge set of that forms a path such that the first node of the path corresponds to the last.

Un ciclo, per definizione, è un cammino che parte e finisce nello stesso nodo.

# Hamiltonian cycle

Un ciclo è detto Hamiltoniano se riesce a passare da tutti i nodi esattamente una volta.

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



# Hamiltonian path

Un cammino Hamiltoniano è un cammino all'interno del grafo che attraversa tutti i nodi del grafo ESATTAMENTE una sola volta.

- 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

Un cammino di Eulero è un cammino fra due nodi qualsiasi che passa attraverso tutti gli archi del grafo, e ci passa ESATTAMENTE una volta.  
Se il nodo di partenza e di arrivo sono uguali, parliamo di CICLO di Eulero.  
I Grafi di Eulero sono grafi che ammettono un ciclo euleriano al loro interno.

# 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).

Un grafo connesso è un grafo euleriano se tutti i vertici hanno grado (cardinalità) pari.

# 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!

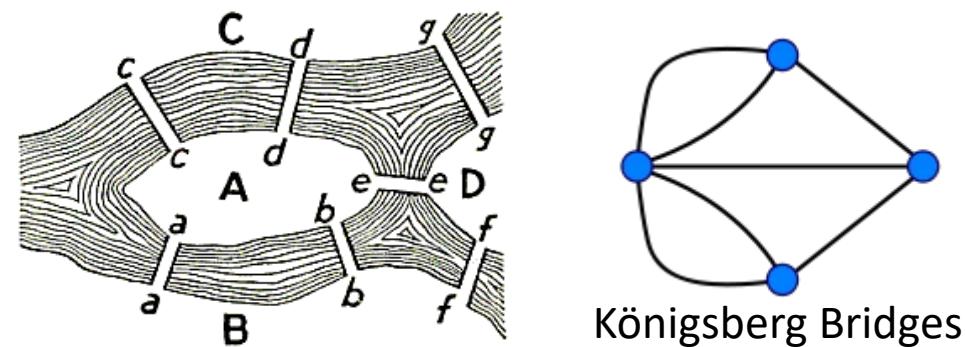


FIGURE 98. Geographic Map:  
The Königsberg Bridges.

# 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)

Ci sono algoritmi che ci permettono in grafi non pesati di verificare se un ciclo esiste e nel caso trovarlo (o trovarli tutti)  
Oppure in grafi pesati possiamo trovare, se esiste, almeno un ciclo. Oppure il ciclo migliore (quello con peso minimo)



### 1. Definizioni:

- **Cammino euleriano:** è un cammino in un grafo che visita ogni arco esattamente una volta e può finire nello stesso nodo da cui è partito (è un cammino che può tornare all'inizio).
- **Ciclo euleriano:** è un ciclo (cammino chiuso) che visita ogni arco esattamente una volta e ritorna al punto di partenza.
- **Cammino hamiltoniano:** è un cammino in un grafo che visita ogni nodo esattamente una volta (può visitare più volte lo stesso arco, ma non il nodo).
- **Ciclo hamiltoniano:** è un ciclo (cammino chiuso) che visita ogni nodo esattamente una volta.

### 2. Condizioni di esistenza:

- **Cammino euleriano:** Esiste in un grafo connesso se e solo se ogni nodo ha grado pari (tutti i nodi hanno un numero pari di archi incidenti).
- **Ciclo euleriano:** Esiste in un grafo connesso se e solo se ogni nodo ha grado pari (tutti i nodi hanno un numero pari di archi incidenti).
- **Cammino hamiltoniano:** Non esistono condizioni generali semplici per la sua esistenza, anche se ci sono alcuni teoremi e condizioni specifiche per determinati tipi di grafi.
- **Ciclo hamiltoniano:** Non esistono condizioni generali semplici per la sua esistenza, ma ci sono diversi teoremi e condizioni specifiche che possono essere verificate per particolari tipi di grafi.

### 3. Visita degli elementi:

- **Cammino euleriano:** Visita ogni arco una sola volta.
- **Ciclo euleriano:** Visita ogni arco una sola volta e ritorna al nodo di partenza.
- **Cammino hamiltoniano:** Visita ogni nodo una sola volta.
- **Ciclo hamiltoniano:** Visita ogni nodo una sola volta e ritorna al nodo di partenza.

In sintesi, la differenza principale sta nel tipo di elementi visitati (nodi o archi) e nel numero di visite permesso (una sola o più volte). I cammini e cicli euleriani sono definiti in base agli archi, mentre quelli hamiltoniani si basano sui nodi.

# 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.

Nell'algoritmo di Hierholzer si assume di lavorare su un grafo Euleriano, che quindi ammetterà un ciclo di Eulero.  
L'algoritmo ci dice: scelto un qualsiasi vertice  $v$ , scegli un percorso che ti riporta in quel vertice, qualunque esso sia.  
Non è possibile rimanere bloccati e non concludere il ciclo perché stiamo lavorando su un grafo di Eulero. Il ciclo che io faccio seguendo questo percorso non è detto che sia un ciclo di Eulero (e che quindi attraverso tutti i nodi e tutti gli archi una sola volta).  
Una volta fatto questo ciclo iniziale, vado a verificare se esiste un vertice nel cammino che ha un arco che non ho attraversato; se esiste, vado a cercare un ciclo nel grafo che attraversa quell'arco. Una volta trovato, lo unisco al mio ciclo trovato inizialmente. E itero così finché non finisco tutti gli archi. Alla fine, troverò il ciclo di Eulero.

# Eulerian cycles: Hierholzer's algorithm (2)

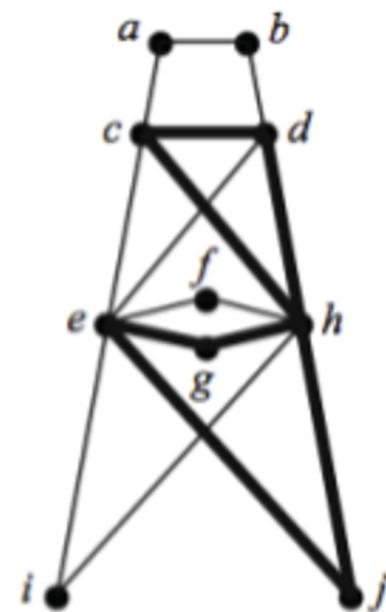
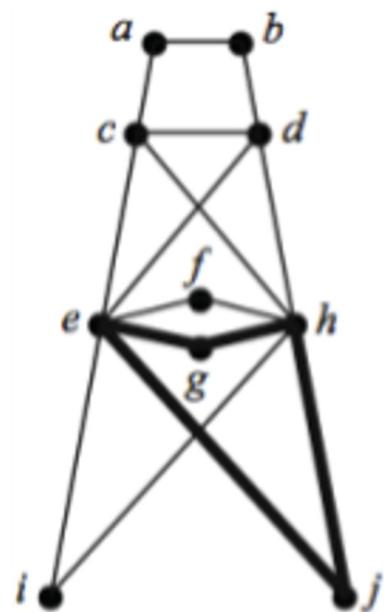
- As long as there exists a vertex  $v$  that belongs to the current tour but that has adjacent edges not part of the tour, start another trail from  $v$ , following unused edges until returning to  $v$ , and join the tour formed in this way to the previous tour.

# 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 contain 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

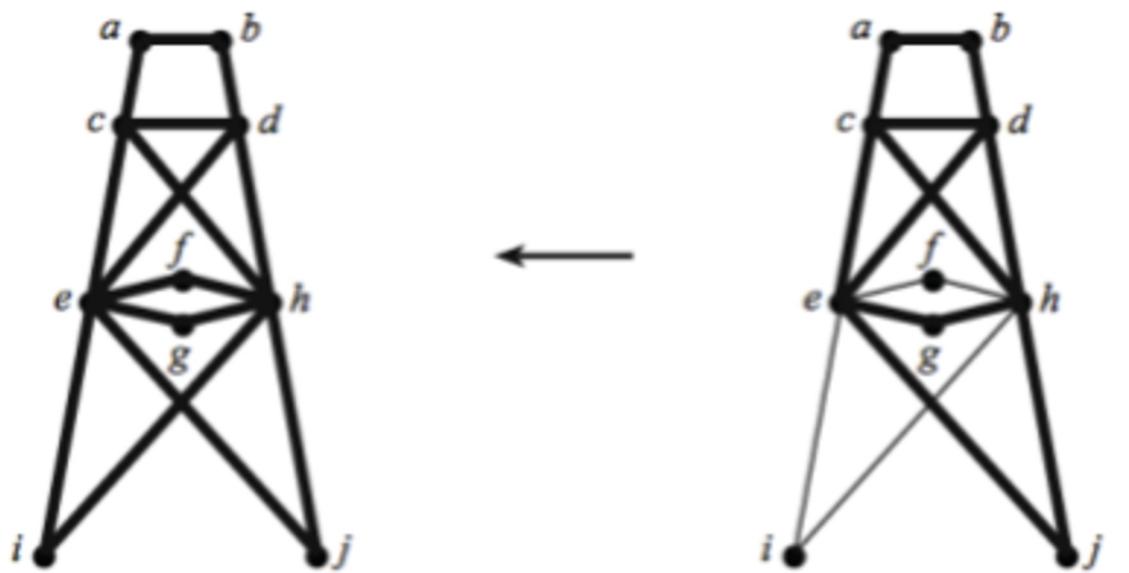


$R_1: e, g, h, j, e$

$Q_1: h, d, c, h$

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

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



$R_4$ : **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

- `IS_eulerian()`: riceve un grafo e ritorna `True` se è un grafo Euleriano.
- `eulerian_circuit()`: riceve il grafo e ritorna un iteratore contenente il ciclo di Eulero del grafo.
- `eulerize()`: riceve un grafo e ritorna un grafo reso euleriano.
- `has_eulerian_path()`: riceve un grafo e ritorna `True` se il grafo ha un cammino di Eulero.

## 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.

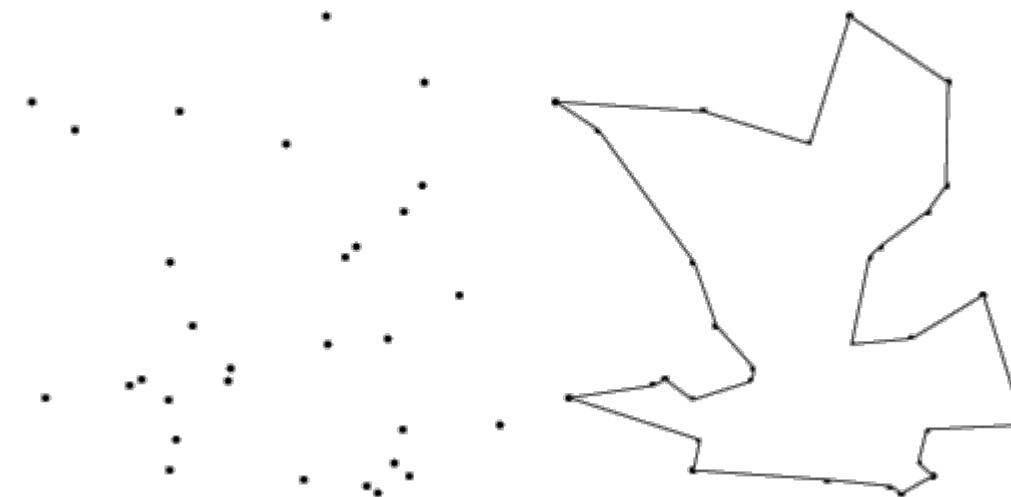
`eulerian_path(G[, source, keys])`

↳ Come circuit ma restituisce un cammino

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

Un problema Hamiltoniano molto famoso è quello del postino che deve attraversare tutte le case a cui deve consegnare la posta minimizzando il percorso, quindi il numero di archi traversati.  
Questo problema va risolto con la RICORSIONE.

# Hamiltonian Cycles in NetworkX

- **hamiltonian\_path()**: riceve in input un grafo **NON PESATO** e ritorna la lista dei nodi in un **CAMMINO HAMILTONIANO** all'interno del grafo.
- Se il grafo fosse pesato, dovrà utilizzare un metodo ricorsivo a trovare il cammino/ciclo Hamiltoniano ottimo; non posso utilizzare un metodo di nx/

**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

• Non li vediamo!

## 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](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

0 BUONA APPROSSIMAZIONE DEL PROBLEMA  
DEL VENDITORE!

## 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 the 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/>