Grafi

Alessandro Pellegrini
pellegrini@diag.uniroma1.it

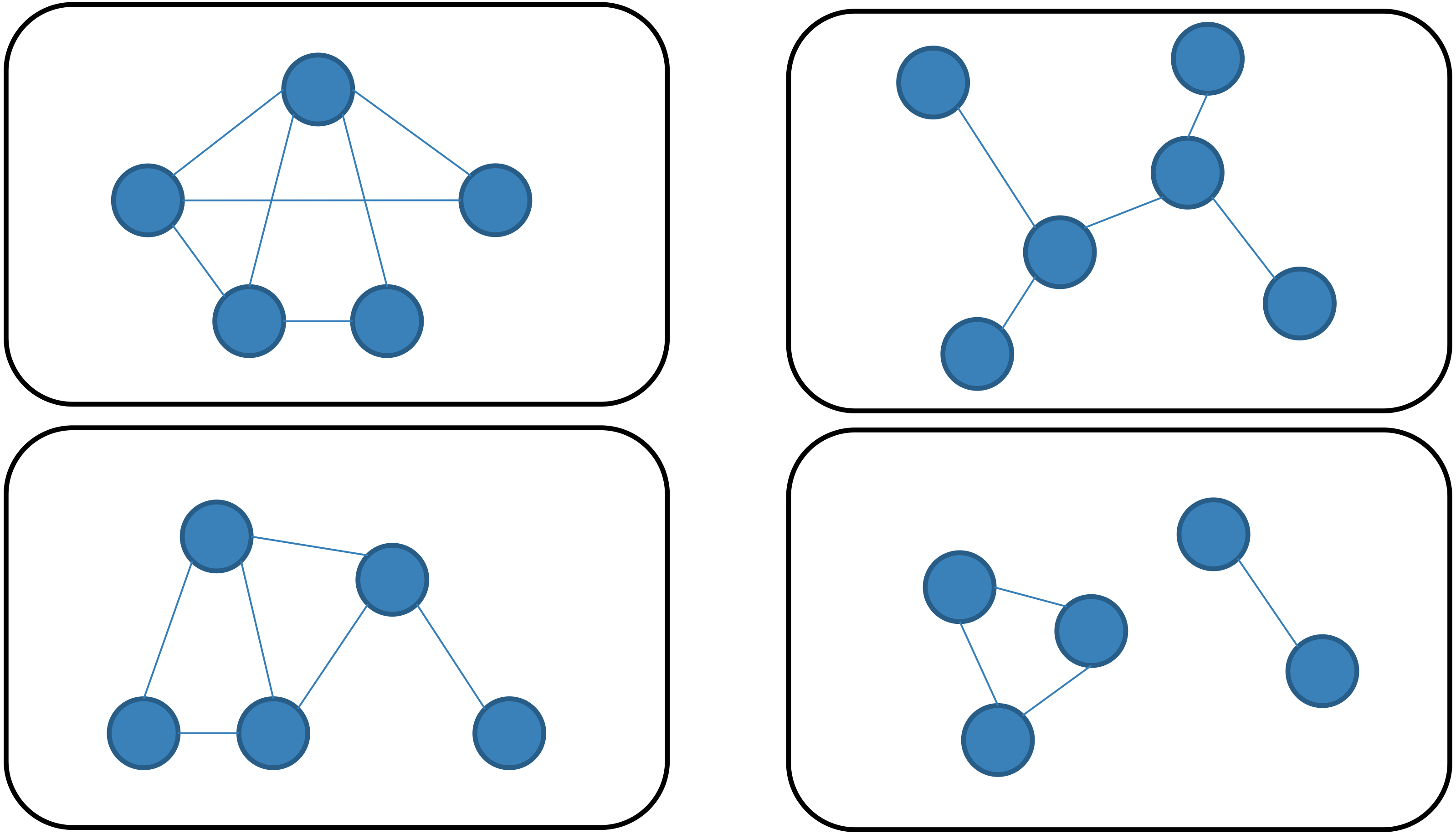
Università di Roma
Tor Vergata

Cos'è un grafo

- Un grafo (o una rete) è una struttura dati che permette di rappresentare delle *relazioni binarie*
  - questo è vero anche per gli alberi, ma le relazioni espresse dagli alberi sono più circoscritte

- Un grafo permette di rappresentare qualsiasi tipo di relazione matematica
  - Una relazione tra n insiemi $S_1, \cdots, S_n$ è un sottoinsieme del prodotto cartesiano $S_1 \times \cdots \times S_n$, ovvero un insieme di n-uple ordinate $(s_{1,} \cdots s_n)$. Viene anche chiamata relazione n-aria.

Alcuni esempi

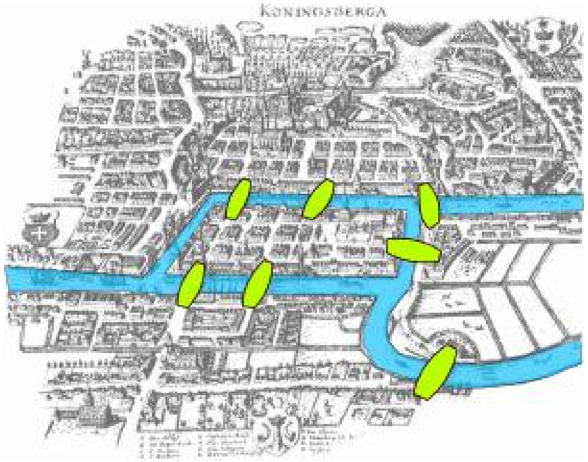


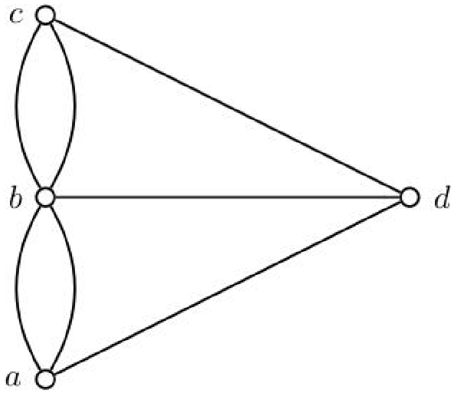
Applicazioni dei grafi

- Molti problemi possono essere visto come problemi su grafi
- I grafi sono potenti astrazioni che permettono di generalizzare molte soluzioni
- Alcune applicazioni:
  - reti sociali
  - reti di trasporti
  - reti di utenze (luce, acqua, gas, …)
  - collegamenti ipertestuali tra pagine
  - interazioni tra proteine
  - reti a pacchetto
  - griglie ad elementi finiti (e.g., scienze dei materiali)
  - reti neurali
  - pianificazione di azioni per robot
  - sistemi quantici
  - epidemiologia
  - sistemi a vincoli (ad esempio, la rete GSM)
  - dipendenze tra elementi

Le origini

- Eulero scrisse un articolo, nel 1736, sul problema dei *sette ponti di Königsberg*
  - La sua analisi si basa sui precedenti risultati sull'*analysis situs* di Leibniz
  - La sua formula fu generalizzata da Cauchy, dando origine alla branca della topologia



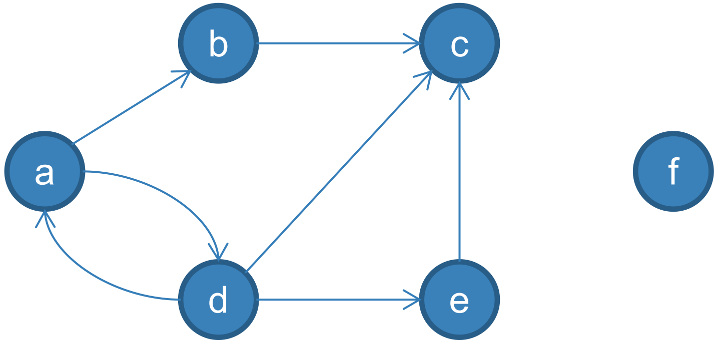


Esiste una soluzione se:
- Il grafo è *connesso*
- Il grafo ha zero o due nodi di grado dispari

Definizioni di base

- un **grafo orientato** (directed graph) è una coppia $G = (V, E)$ dove:
  - V è un insieme di *vertici* (vertex) o *nodi* (nodes)
  - E è un insieme di coppie ordinate (u, v) di nodi dette *archi* (edges) o *rami* (branches)

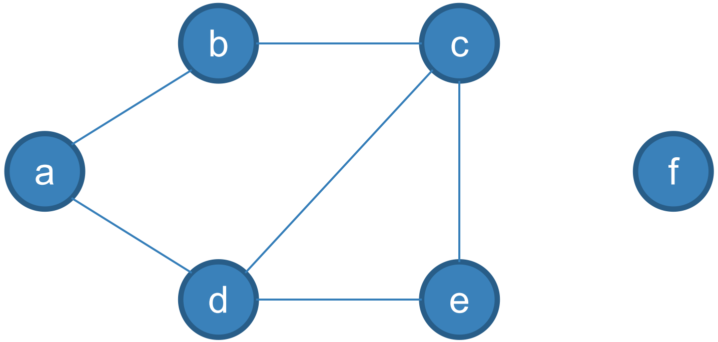
V = { a,b,c,d,e,f }
E = { (a,b), (a,d), (b,c)
(d,a), (d,c), (d,e) }



Definizioni di base

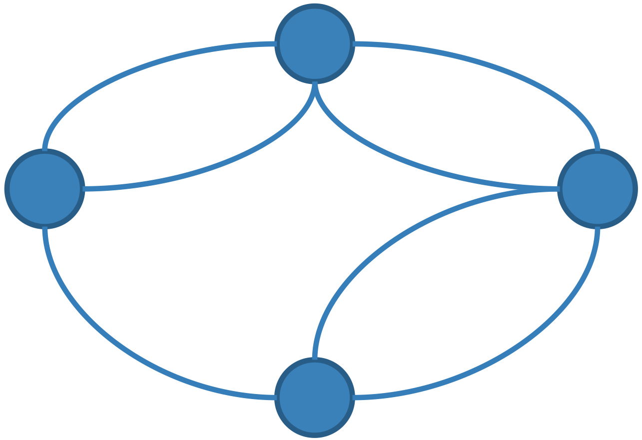
- un **grafo non orientato** (undirected graph) è una coppia $G = (V, E)$ dove:
  - V è un insieme di *vertici* (vertex) o *nodi* (nodes)
  - E è un insieme di coppie non ordinate (u, v) di nodi dette *archi* (edges)

V = { a,b,c,d,e,f }
E = { (a,b), (a,d), (b,c)
(d,c), (d,e) }



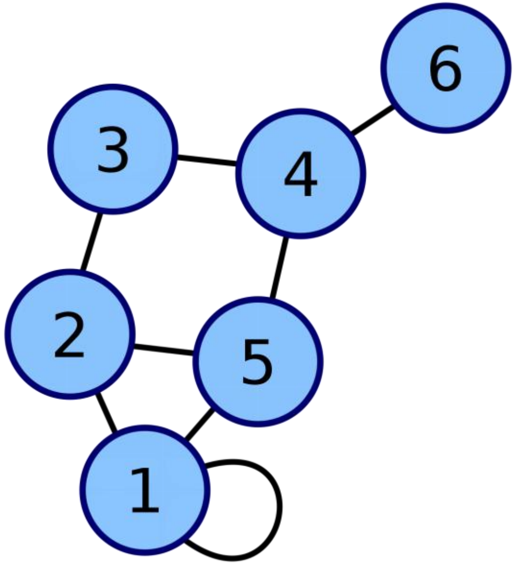
Definizioni di base

- **Multigrafo**: un grafo in cui E è un multi-insieme



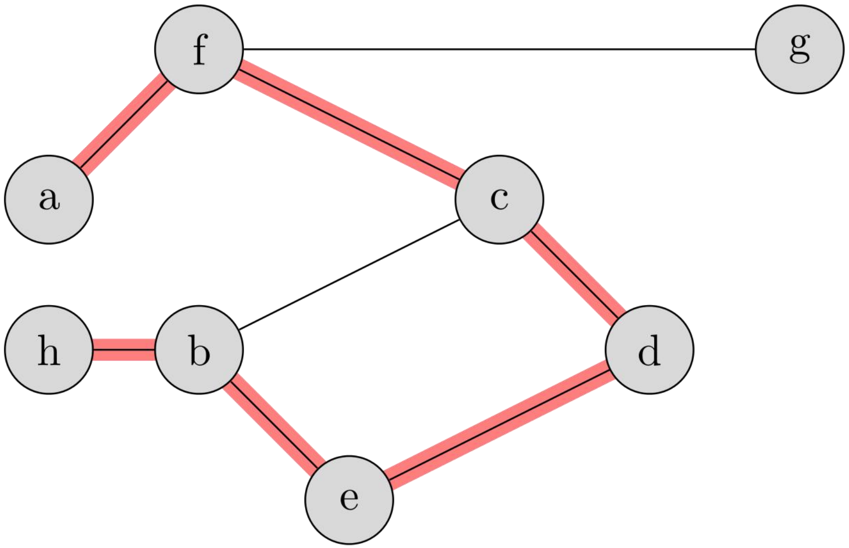
Definizioni di base

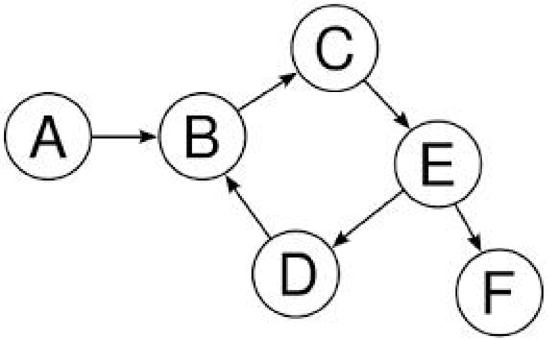
- **Pseudografo**: un grafo in cui E contiene anche coppie (v, v) dette *cappi* (loop)



Definizioni di base

- **Cammino** (di lunghezza k): una sequenza di vertici $v_1, v_2, \cdots, v_k$ tale che $(v_i, v_{i+1}) \in E$.
- **Circuito**: un cammino con $v_1 = v_k$
- **Ciclo**: un circuito senza vertici ripetuti

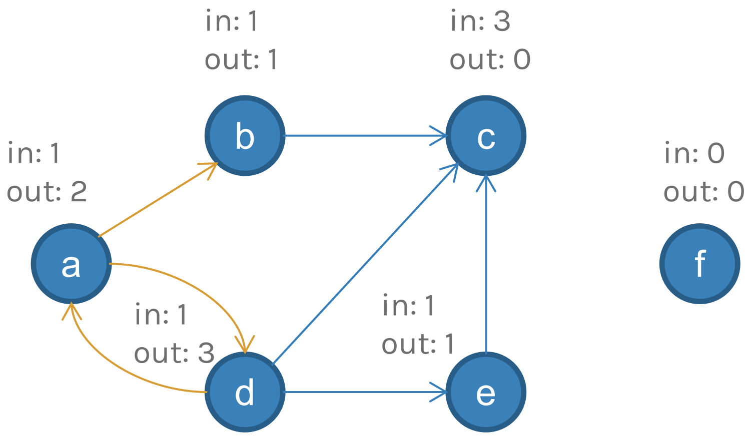




Definizioni di base

- Un vertice v è detto **adiacente** a u se esiste un arco (u, v)
- Un arco (u, v) è detto **incidente** a u e v
- In un grafo indiretto, la relazione di incidenza è simmetrica
- Il **grado** (degree) di un nodo è il numero di archi incidenti
  - nel caso di grafi orientati, si parla di grado entrante (in-degree) e uscente (out-degree)

(a, b) è incidente da a a b
(a, d) è incidente da a a d
(d, a) è incidente da d ad a
b è adiacente ad a
d è adiacente ad a
a è adiacente ad d



Definizioni di base

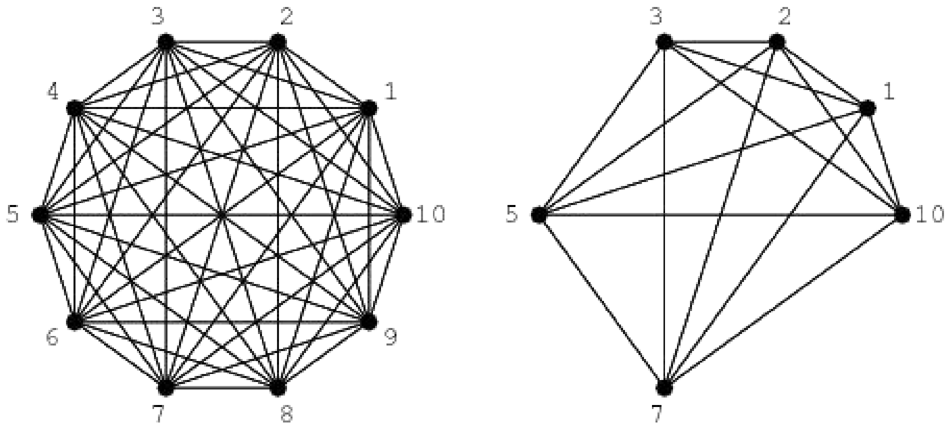
- **Dimensioni di un grafo**:
  - $n = |V|$
  - $m = |E|$

- Alcune relazioni tra n ed m:
  - In un grafo non orientato, $m \leq \frac{n(n-1)}{2} = O(n^2)$
  - In un grafo orientato, $m \leq n^2 - n = O(n^2)$

- La complessità degli algoritmi sui grafi viene tipicamente indicata in termini di n ed m, ad esempio $O(n + m)$

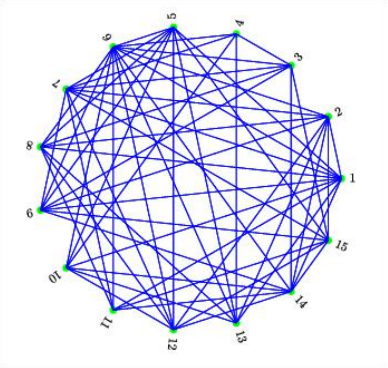
Definizioni di base

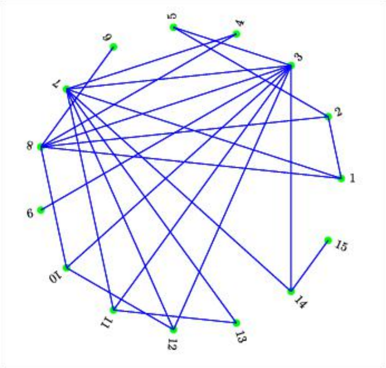
- $G' = (V', E')$ è un **sottografo** di $G = (V, E)$ se e solo se $V' \subset V$ e $E' \subset E$



Definizioni di base

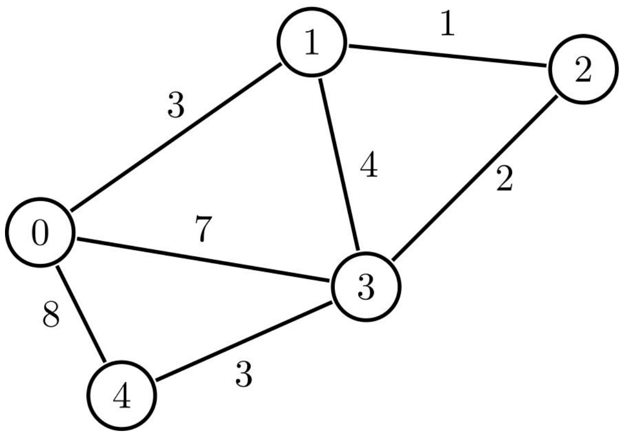
- Un grafo è detto **completo** se è presente un arco tra ogni coppia di nodi
  - Nel caso di un grafo non orientato, $m = \frac{n(n-1)}{2}$
- Un grafo è detto **sparso** se ha pochi archi
  - ad sempio, $m = O(n)$ o $m = O(n \log n)$
- Un grafo è detto **denso** se $m \gg n$
  - ad esempio, $m = \Omega(n)$





Definizioni di base

- Un **grafo pesato** (weighted) è un grafo in cui ogni arco è associato ad un valore numerico chiamato **peso** (costo, profitto, distanza, tempo, ...)
- Il peso è dato dalla funzione di peso $w : V \times V \mapsto \mathbb{R}$
- Se un arco $(u, v) \notin E$, il peso assume un valore che dipende dal problema
  - Tipicamente, $w(u, v) = 0$ o $w(u, v) = +\infty$



Operazioni fondamentali

- Tipicamente, i grafi possono cambiare nel tempo
- È necessario prevedere quindi operazioni per alterare nodi ed archi di un grafo:
  - VERTICES(): restituisce l'insieme di tutti i vertici
  - ADJ(v): restituisce l'insieme dei nodi adiacenti a v
  - INSERTNODE(v): inserisce il nodo v al grafo
  - INSERTEDGE(u,v): aggiunge l'arco (u,v) al grafo
  - DELETENODE(v): rimuove il nodo v e tutti gli archi in cui esso è coinvolto
  - DELETEEDGE(u,v): rimuove l'arco (u,v) dal grafo

Rappresentazione dei grafi

Rappresentazione dei grafi

- I grafi sono strutture dati complesse (il numero di archi e nodi può essere molto elevato)
- Tre modalità di rappresentazione principali:
  - **liste di adiacenza**: ad ogni vertice è associata la lista dei vertici adiacenti
    - si possono utilizzare array, liste collegate, dizionari, ...
  - **matrici di adiacenza**: una matrice quadrata A (di dimensione $n \times n$) tale che:

$$a_{ij} = \begin{cases} 1 & (v_i, v_j) \in E \\ 0 & \text{altrimenti} \end{cases}$$

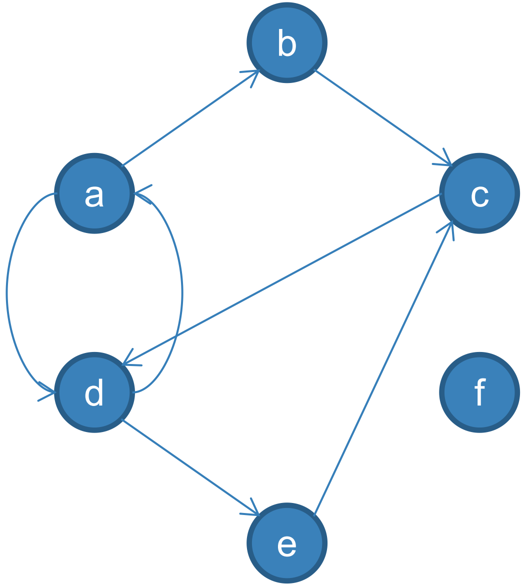
  - **matrici di incidenza**: una matrice rettangolare B (dimensione $n \times m$) tale che:

$$b_{ik} = \begin{cases} -1 & \text{se l'arco } k - \text{esimo esce dal nodo } i \\ 1 & \text{se l'arco } k - \text{esimo entra nel nodo } i \\ 0 & \text{altrimenti} \end{cases}$$

Matrice di adiacenza: grafi orientati

- La matrice occupa n^2 bit

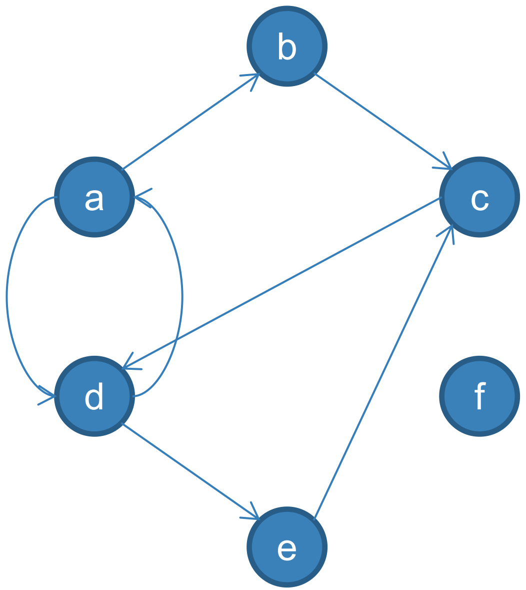
$$
\begin{array}{c|c}
 & \begin{array}{cccccc} \mathbf{a} & \mathbf{b} & \mathbf{c} & \mathbf{d} & \mathbf{e} & \mathbf{f} \end{array} \\
\begin{array}{c} \mathbf{a} \\ \mathbf{b} \\ \mathbf{c} \\ \mathbf{d} \\ \mathbf{e} \\ \mathbf{f} \end{array} &
\begin{pmatrix}
0 & 1 & 0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 \\
1 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0
\end{pmatrix}
\end{array}
$$



Matrice di incidenza: grafi orientati

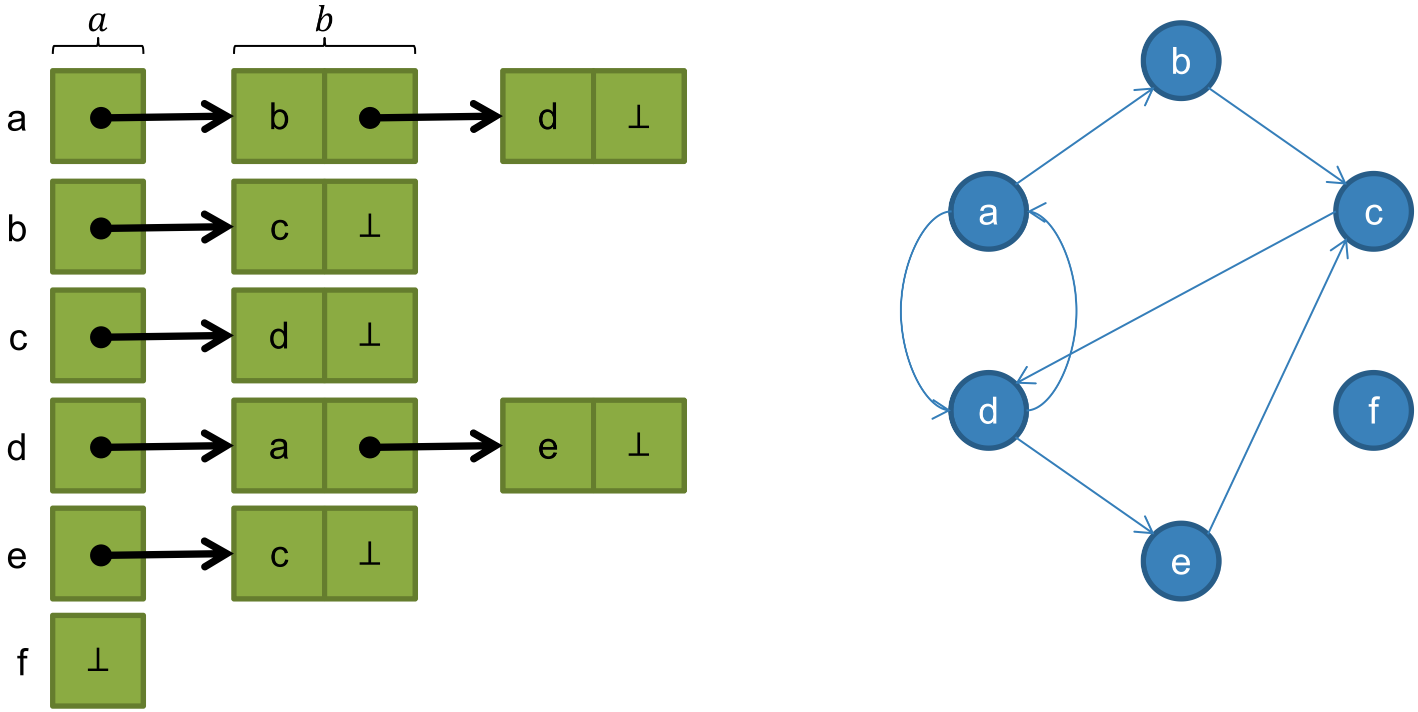
- La matrice occupa $n \times m$ byte

$$
\begin{array}{c|c}
 & \begin{array}{ccccccc} \textbf{ab} & \textbf{ad} & \textbf{bc} & \textbf{cd} & \textbf{da} & \textbf{de} & \textbf{ec} \end{array} \\
\begin{array}{c} \textbf{a} \\ \textbf{b} \\ \textbf{c} \\ \textbf{d} \\ \textbf{e} \\ \textbf{f} \end{array} &
\begin{pmatrix}
-1 & -1 & 0 & 0 & 1 & 0 & 0 \\
1 & 0 & -1 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & -1 & 0 & 0 & 1 \\
0 & 1 & 0 & 1 & -1 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 1 & -1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0
\end{pmatrix}
\end{array}
$$



Liste di adiacenza: grafi orientati

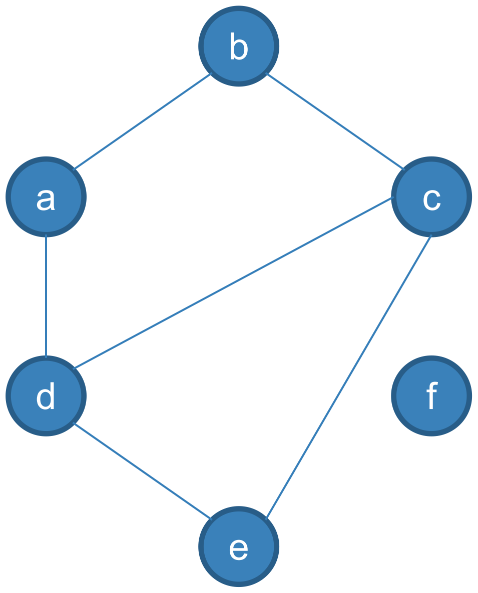
- Lo spazio occupato è $an + bm$ bit



Matrice di adiacenza: grafi non orientati

- La matrice occupa n^2 bit

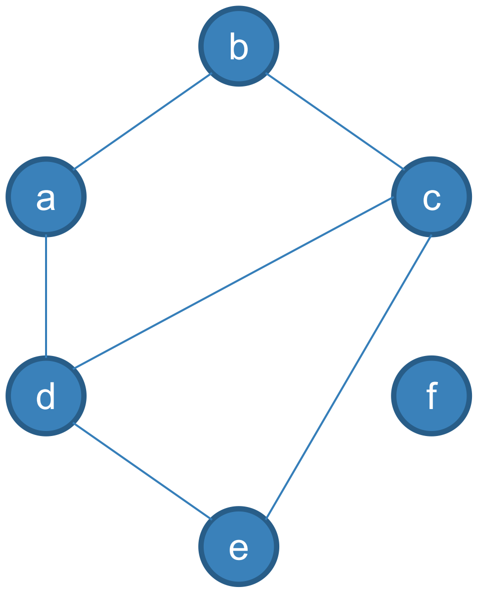
$$
\begin{array}{c c}
 & \begin{array}{cccccc} \mathbf{a} & \mathbf{b} & \mathbf{c} & \mathbf{d} & \mathbf{e} & \mathbf{f} \end{array} \\
\begin{array}{c} \mathbf{a} \\ \mathbf{b} \\ \mathbf{c} \\ \mathbf{d} \\ \mathbf{e} \\ \mathbf{f} \end{array} &
\begin{pmatrix}
0 & 1 & 0 & 1 & 0 & 0 \\
 & 0 & 1 & 0 & 0 & 0 \\
 & & 0 & 1 & 1 & 0 \\
 & & & 0 & 1 & 0 \\
 & & & & 0 & 0 \\
 & & & & & 0
\end{pmatrix}
\end{array}
$$



Matrice di incidenza: grafi non orientati

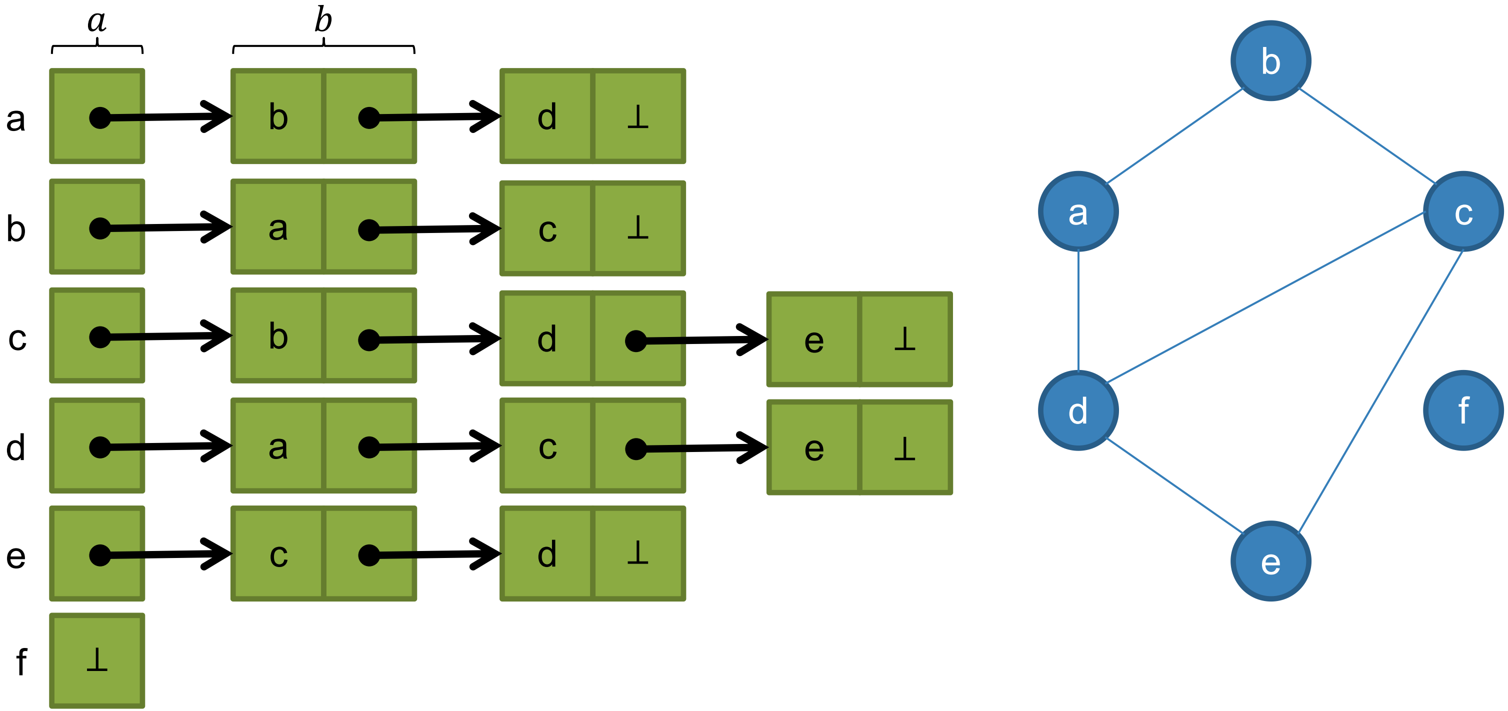
- La matrice occupa $n \times m$ bit

$$
\begin{array}{c}
\\ \mathbf{a} \\ \mathbf{b} \\ \mathbf{c} \\ \mathbf{d} \\ \mathbf{e} \\ \mathbf{f}
\end{array}
\begin{array}{c}
\begin{array}{cccccc} \mathbf{ab} & \mathbf{ad} & \mathbf{bc} & \mathbf{cd} & \mathbf{de} & \mathbf{ec} \end{array} \\
\begin{pmatrix}
1 & 1 & 0 & 0 & 0 & 0 \\
1 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 1 & 1 & 0 & 1 \\
0 & 1 & 0 & 1 & 1 & 0 \\
0 & 0 & 0 & 0 & 1 & 1 \\
0 & 0 & 0 & 0 & 0 & 0
\end{pmatrix}
\end{array}
$$



Liste di adiacenza: grafi non orientati

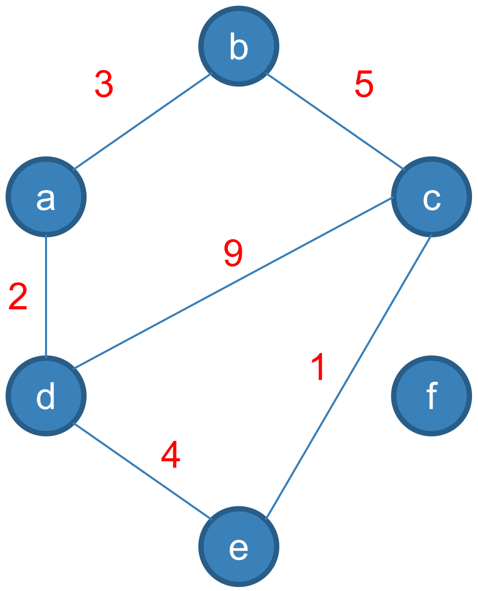
- Lo spazio occupato è $an + 2bm$ bit



Matrice di adiacenza: grafi pesati

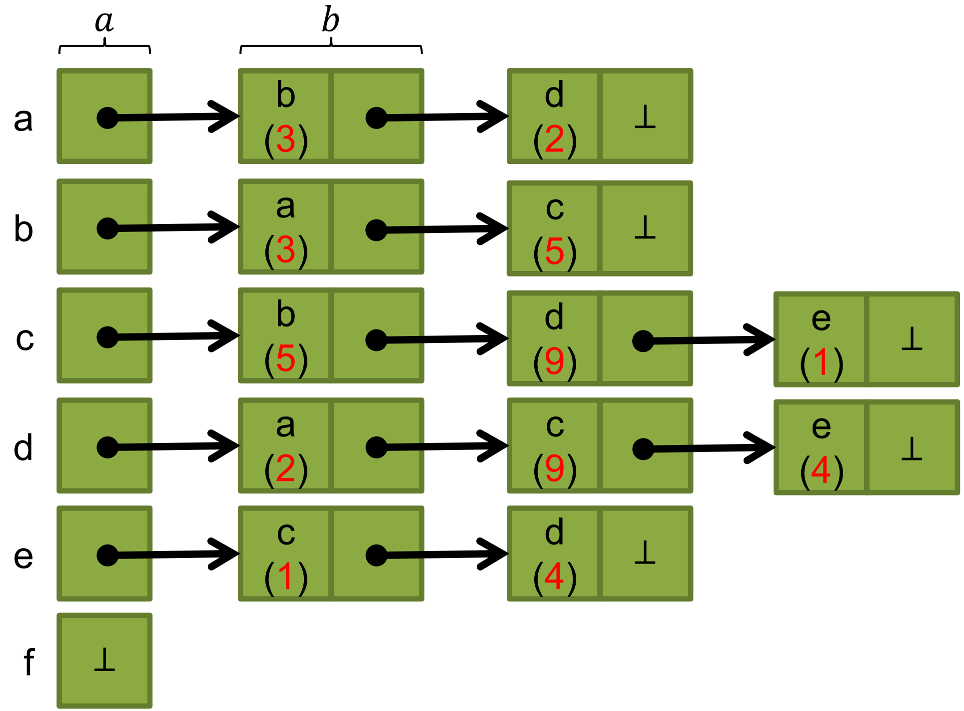
- La matrice occupa n^2 valori

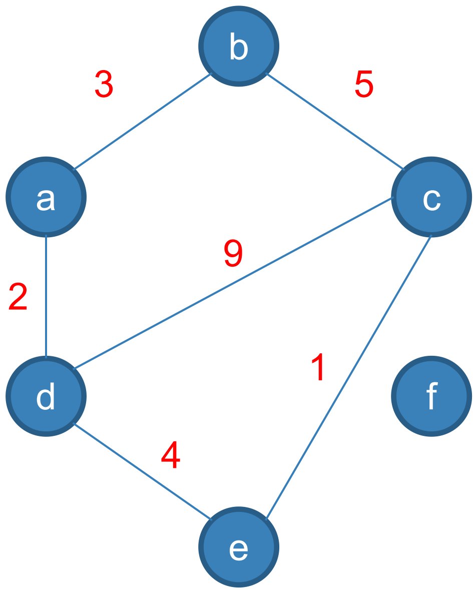
$$
\begin{array}{c c}
 & \begin{array}{cccccc} \mathbf{a} & \mathbf{b} & \mathbf{c} & \mathbf{d} & \mathbf{e} & \mathbf{f} \end{array} \\
\begin{array}{c} \mathbf{a} \\ \mathbf{b} \\ \mathbf{c} \\ \mathbf{d} \\ \mathbf{e} \\ \mathbf{f} \end{array} &
\begin{pmatrix}
0 & 3 & 0 & 2 & 0 & 0 \\
 & 0 & 5 & 0 & 0 & 0 \\
 & & 0 & 9 & 1 & 0 \\
 & & & 0 & 4 & 0 \\
 & & & & 0 & 0 \\
 & & & & & 0
\end{pmatrix}
\end{array}
$$



Liste di adiacenza: grafi pesati

- Lo spazio occupato è $an + 2bm$ bit





Confronto tra le rappresentazioni

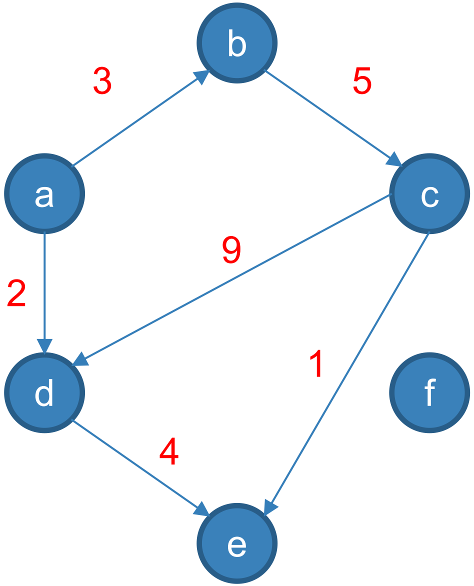
- Lista di adiacenza:
  - Pro: permette di individuare i nodi adiacenti al nodo v in $O(grado(v))$
  - Contro: gli inserimenti e le cancellazioni operano su liste concatenate, con un costo $O(grado(v))$
- Matrice di adiacenza:
  - Pro: Inserimento e cancellazione in $O(1)$ ammortizzato
  - Contro: permette di individuare i nodi adiacenti al nodo v in $O(n)$
- Matrice di incidenza:
  - Pro: Inserimento e cancellazione in $O(1)$ ammortizzato
  - Contro: permette di individuare i nodi adiacenti al nodo v in $O(m)$

Implementazione Python (pesata, con dizionari)

```
class Graph:
  def __init__(self):
    self.nodes = { }

  def vertices(self):
    return self.nodes.keys()

  def __len__(self):
    return len(self.nodes)

  def adj(self, u):
    if u in self.nodes:
      return self.nodes[u]

  def insertNode(self,u):
    if u not in self.nodes:
      self.nodes[u] = { }

  def insertEdge(self, u, v, w=0):
    self.insertNode(u)
    self.insertNode(v)
    self.nodes[u][v] = w
```

Implementazione Python (pesata, con dizionari)

```
graph = Graph()
for u,v,w in [ ('a','b',3), ('a','d',2), ('b','c',5),
        ('c','d',9), ('c','e',1), ('d','e',4)]:
  graph.insertEdge(u,v,w)
graph.insertNode('f')

for u in graph.V():
  print(u, "->", graph.adj(u))

a -> {'b': 3, 'd': 2}
b -> {'c': 5}
d -> {'e': 4}
c -> {'d': 9, 'e': 1}
e -> {}
f -> {}
```



Visite di grafi

Definizione del problema

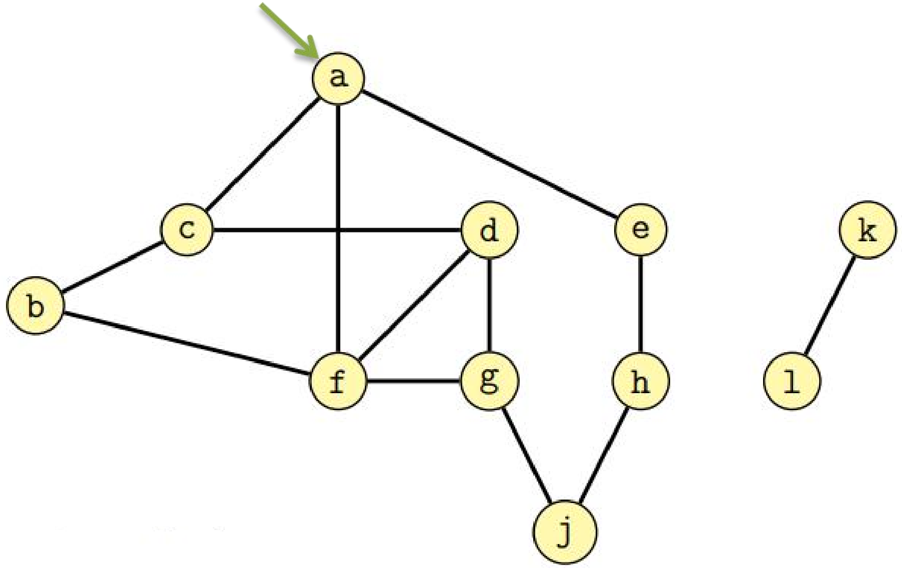
- Il problema della visita ricorda quello che abbiamo visto nel contesto degli alberi: vogliamo visitare tutti i nodi del grafo
- Dato un grafo $G = (V, E)$ e un vertice $r \in V$ (chiamato *radice* o *sorgente*), visitare <u>una e una sola volta</u> tutti i nodi del grafo

- Ancora due approcci:
  - Visita in ampiezza (BFS), o visita per livelli: si visita la sorgente, poi i nodi a distanza 1 dalla sorgente, poi quelli a distanza 2, …
  - Visita in profondità (DFS), o visita ricorsiva: per ogni nodo adiacente, si visita ricorsivamente tale nodo

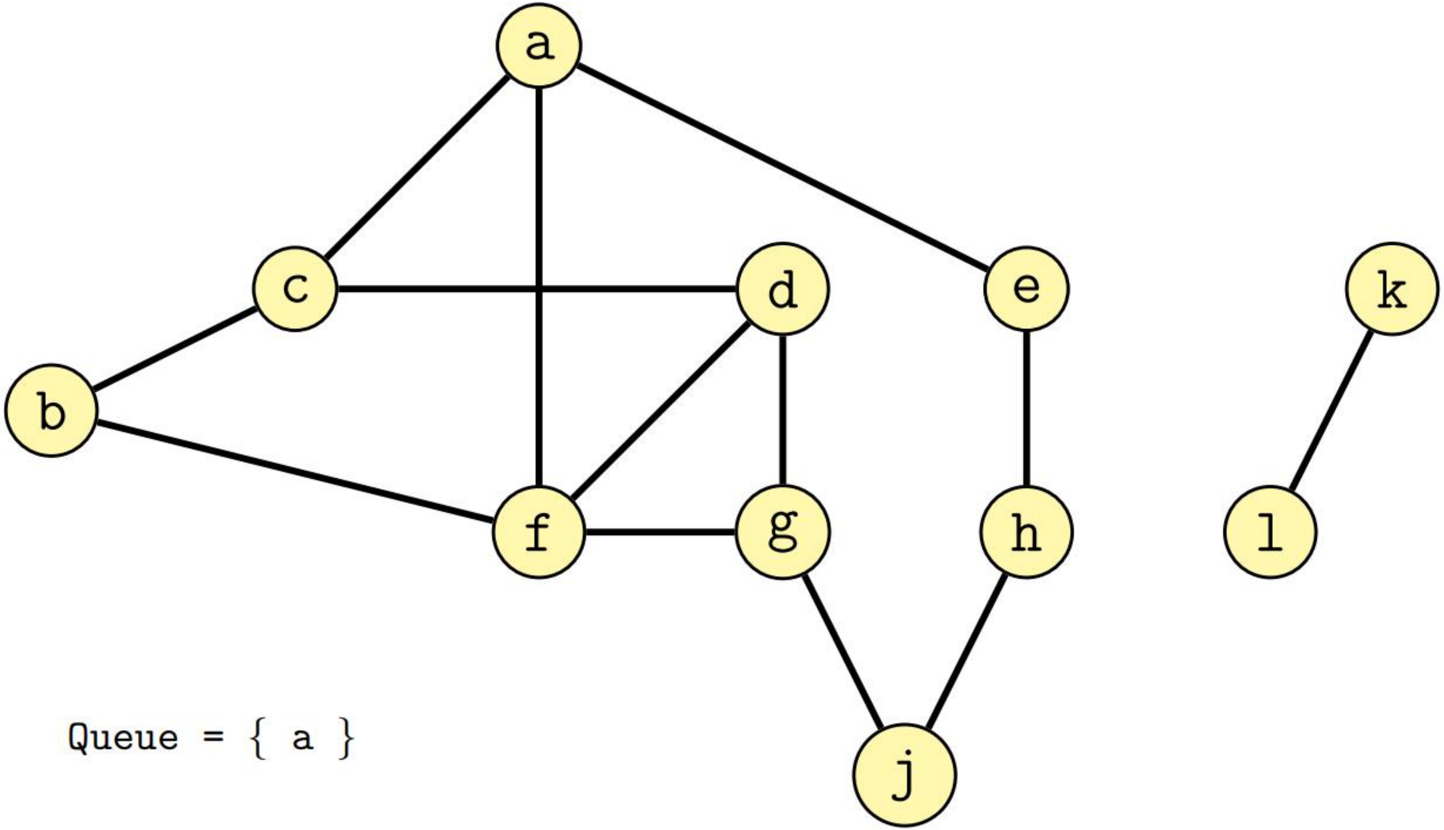
Un possibile approccio

- Un approccio immediato potrebbe essere quello di “riciclare” l'implementazione delle visite utilizza per gli alberi

```
BFS(root):
  q ← Queue()
  node ← root
  while node != ⊥:
    <do something with node>
    for each child of node:
      q.enqueue(child)
    node ← q.dequeue()
```

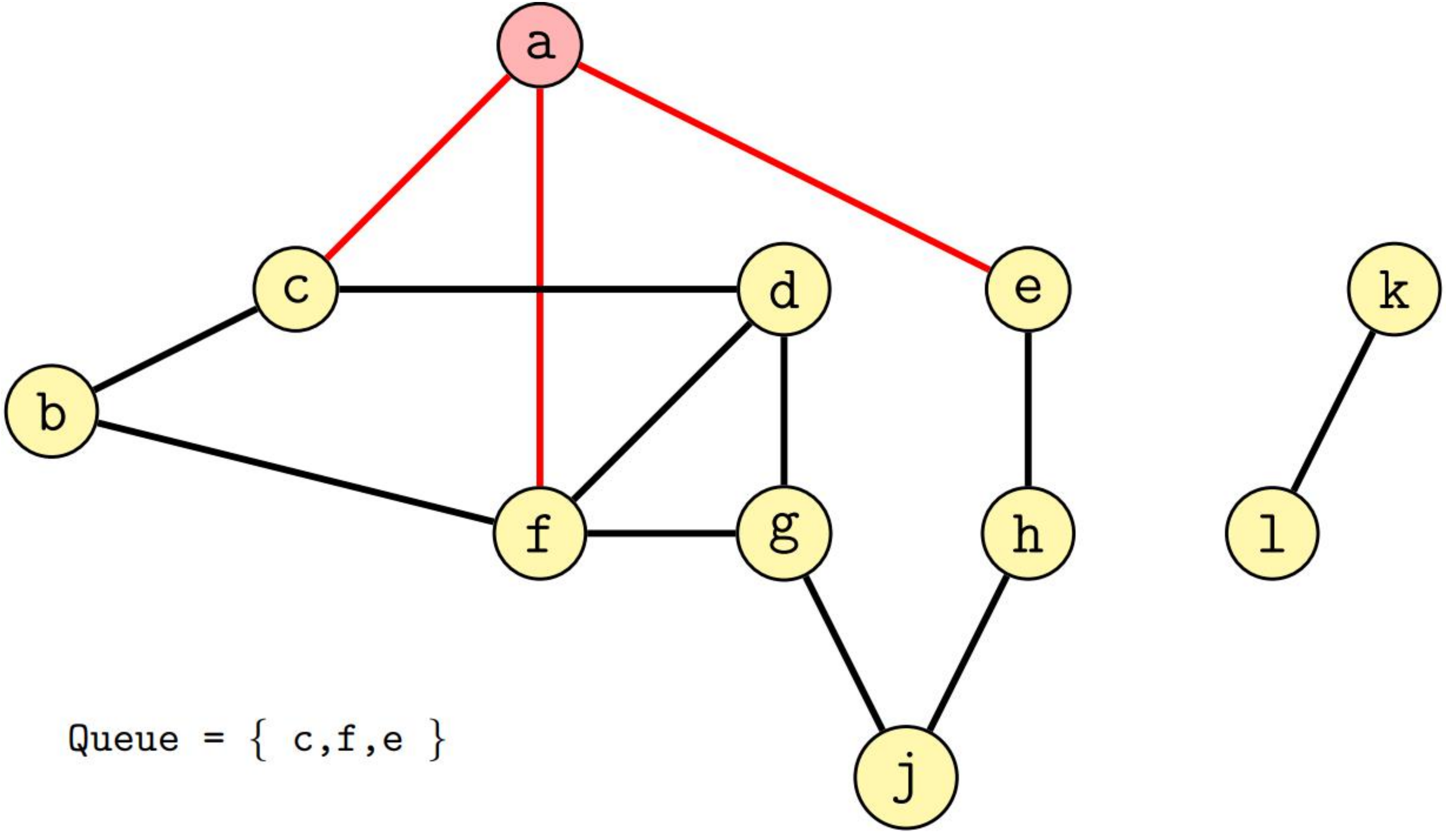


Meno facile di quanto possa sembrare!



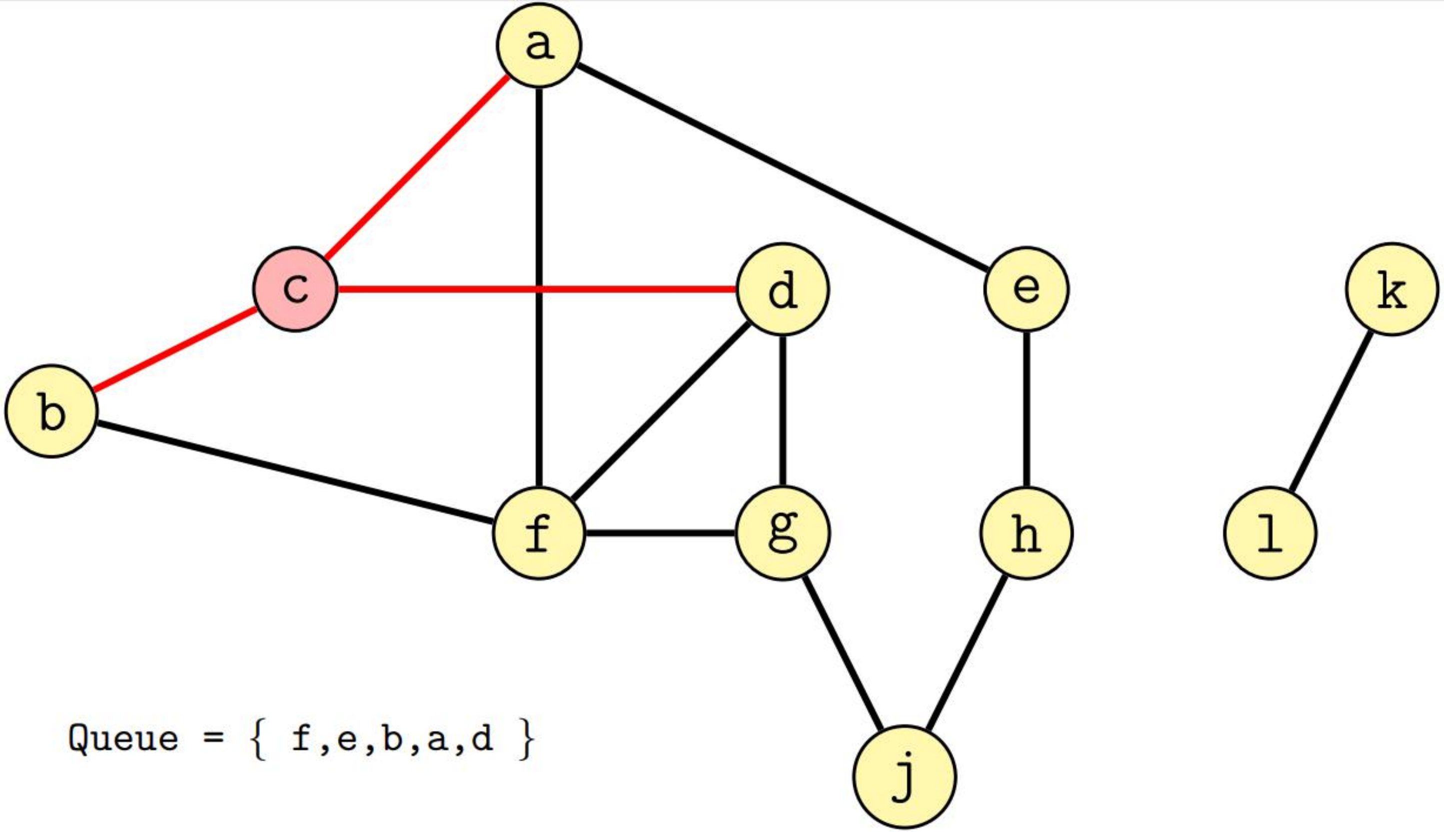
Queue = { a }

Meno facile di quanto possa sembrare!



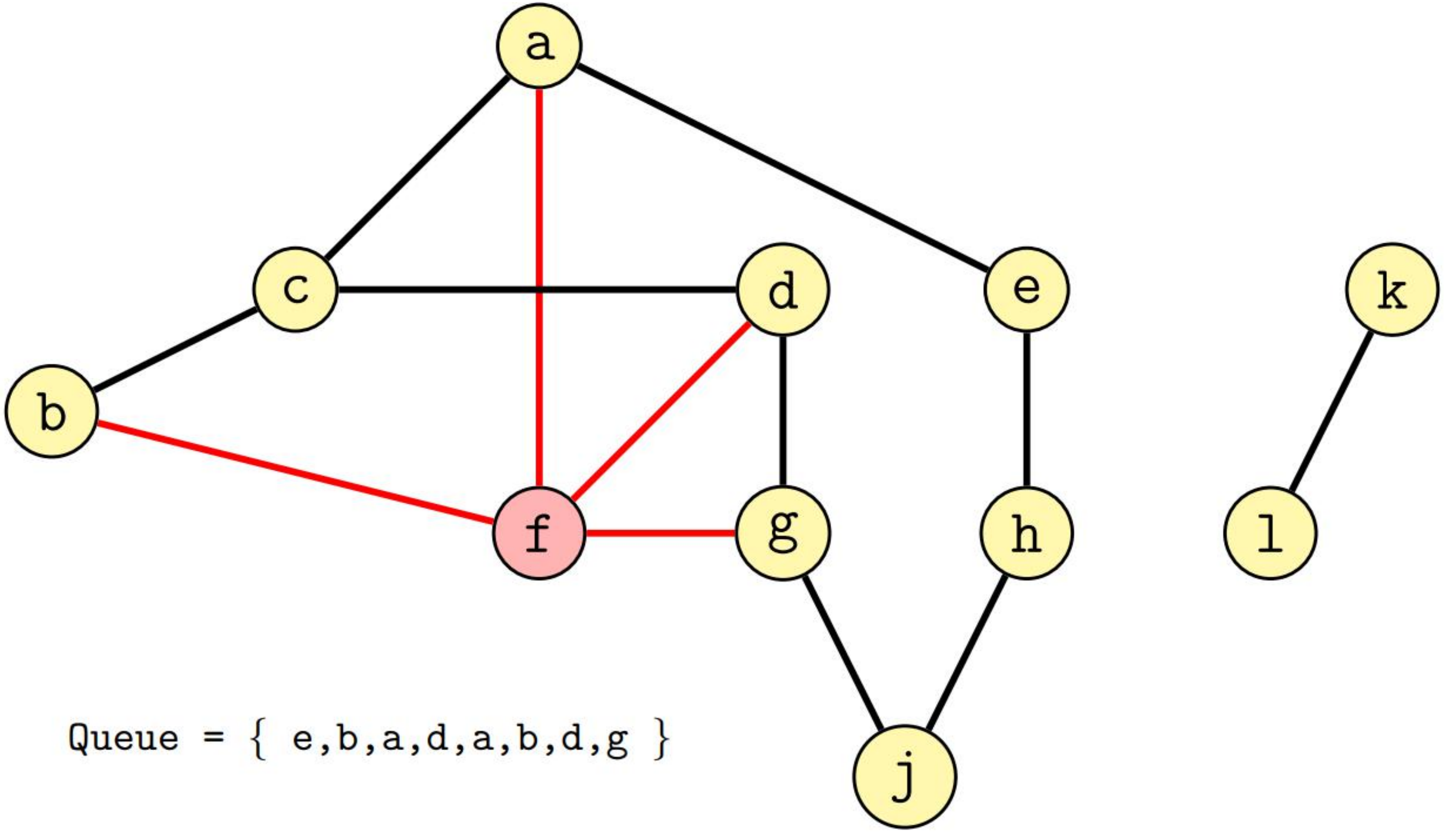
Queue = { c,f,e }

Meno facile di quanto possa sembrare!



Queue = { f,e,b,a,d }

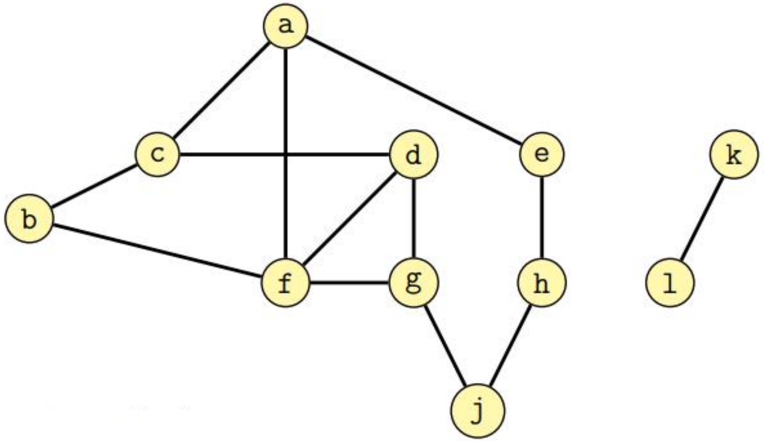
Meno facile di quanto possa sembrare!



Queue = { e,b,a,d,a,b,d,g }

Quali sono i problemi?

- Nei grafi possono esserci **cicli**: vogliamo visitare un nodo <u>una e una sola</u> volta
  - Occorre **marcare** i nodi visitati
- Presenza di **nodi isolati**: in generale, si vuole considerare tutte le componenti del grafo, anche in caso di **foresta**
  - La soluzione è banale, avendo a disposizione l'insieme dei vertici

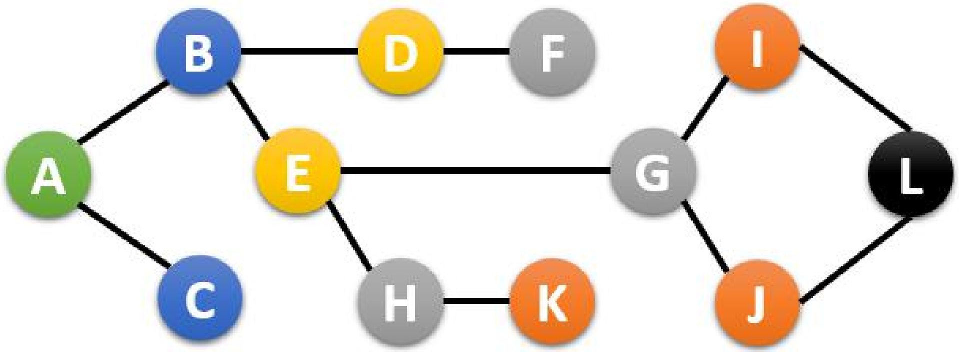


Algoritmo generale di attraversamento

```
GRAPHTRAVERSAL(G, r):
  Set S ← Ø
  S.insert(r)
  < marca il nodo r come già scoperto >
  while S.size() > 0:
    Node u ← S.remove()
    < visita il nodo u >
    foreach v in G.adj(u):
      < visita l'arco (u, v) >
      if v non è ancora stato marcato then:
        < marca il nodo v come già scoperto >
        S.insert(v)
```

Visita in ampiezza—BFS

- La visita in ampiezza fa uso di una coda per memorizzare tutti i nodi adiacenti al nodo v visitato, che portano ad un nodo non marcato come scoperto
- I nodi raggiungibili non marcati vengono quindi marcati
- La visita procede estraendo il nodo successivo dalla coda
- Cosa succede alla visita successiva?

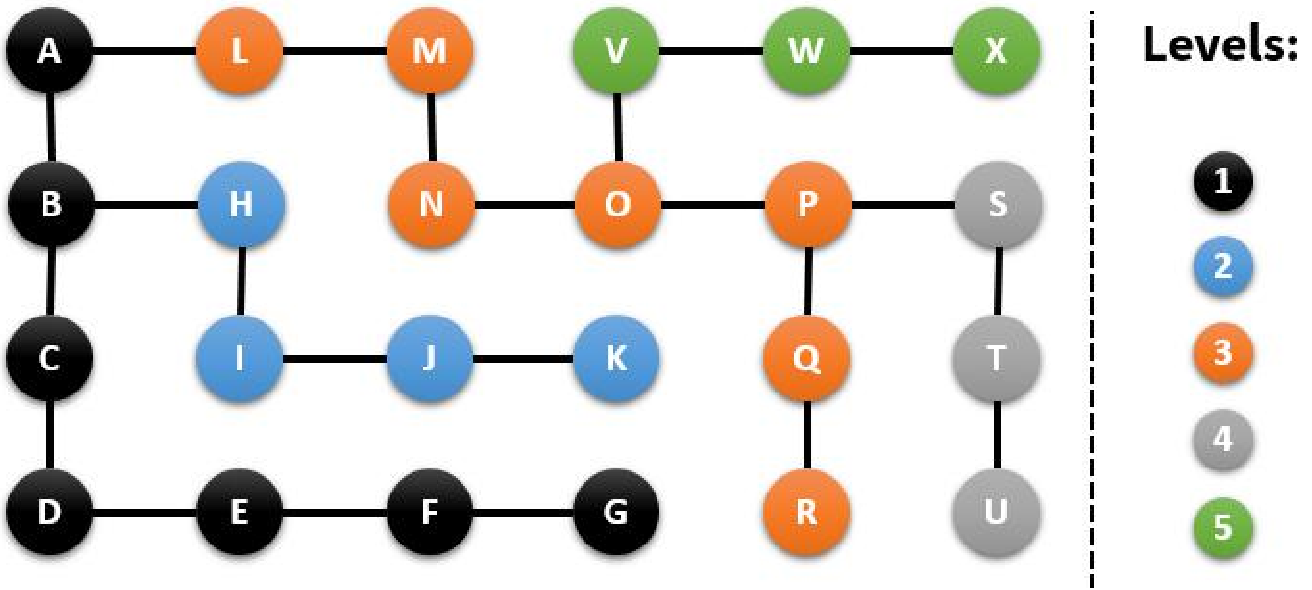


Visita in ampiezza—BFS

```
BFS(G, r):
    Queue Q ← Ø
    S.enqueue(r)
    foreach u in G.vertices() \ {r}:
        u.discovered ← false
    r.visited ← true
    while not Q.isEmpty():
        Node u ← Q.dequeue()
        < visita il nodo u >
        foreach v in G.adj(u):
                < visita l'arco (u, v) >
                if not v.discovered then:
                        v.discovered ← true
                        Q.enqueue(v)
```

- Complessità: $O(n + m)$

Visita in profondità—DFS



Visita in profondità—DFS

```
DFS(G, r):
    Stack S ← Ø
    S.push(r)
    foreach u in G.vertices():
        u.discovered ← false
    while not S.isEmpty():
        Node u ← S.pop()
        if not u.discovered then:
            < visita il nodo u >
            u.discovered ← true
            foreach v in G.adj(u):
                < visita l’arco (u, v) >
                S.push(v)
```

- Si può utilizzare la ricorsione per non dover utilizzare esplicitamente uno stack
- Complessità: $O(n+m)$

Cammini minimi

Il problema dei cammini minimi

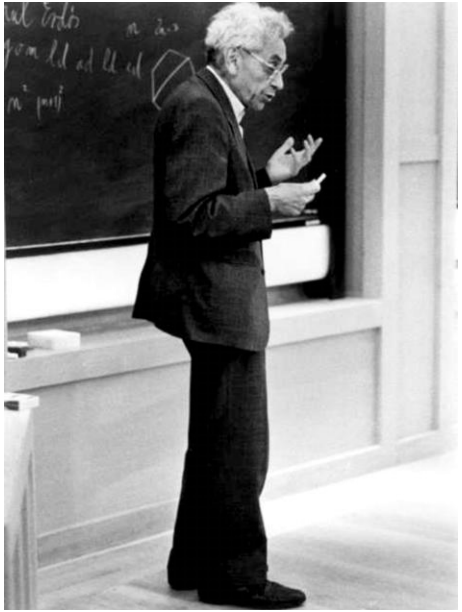
- Dato un grafo pesato $G = (V, E)$, con pesi non negativi, si vuole trovare un cammino tra due nodi del grafo che abbia peso minore.
- Alcune varianti:
  - Il cammino tra un nodo sorgente $s \in V$ e un nodo destinazione $t \in V$
    - In questo caso il cammino è una sequenza di nodi $v_1, v_k, \cdots v_k: (v_i, v_{i+1}) \in E, v_1 = s, v_k = t$
    - La lunghezza del cammino individuato è data da $\sum_1^{k-1} w(v_i, v_{i+1})$
  - Il cammino tra un nodo sorgente $s \in V$ e tutti gli altri nodi (Single Source Shortest Path—SSSP)
  - Il cammino tra tutte le coppie di nodi nel grafo (All Pairs Shortest Path—APSP)

Nel caso di archi con peso uguale

```
CAMMINI(G, r):
    Queue Q ← Ø
    Q.enqueue(r)
    foreach u in G.vertices() \ {r}:
        u.distance ← ∞
    r.distance ← 0
    r.parent ← ⊥
    while not Q.isEmpty():
        Node u ← Q.dequeue()
        foreach v in G.adj(u):
            if v.distance = ∞ then
            v.distance ← u.distance + 1
            v.parent ← u
            Q.enqueue(v)
```

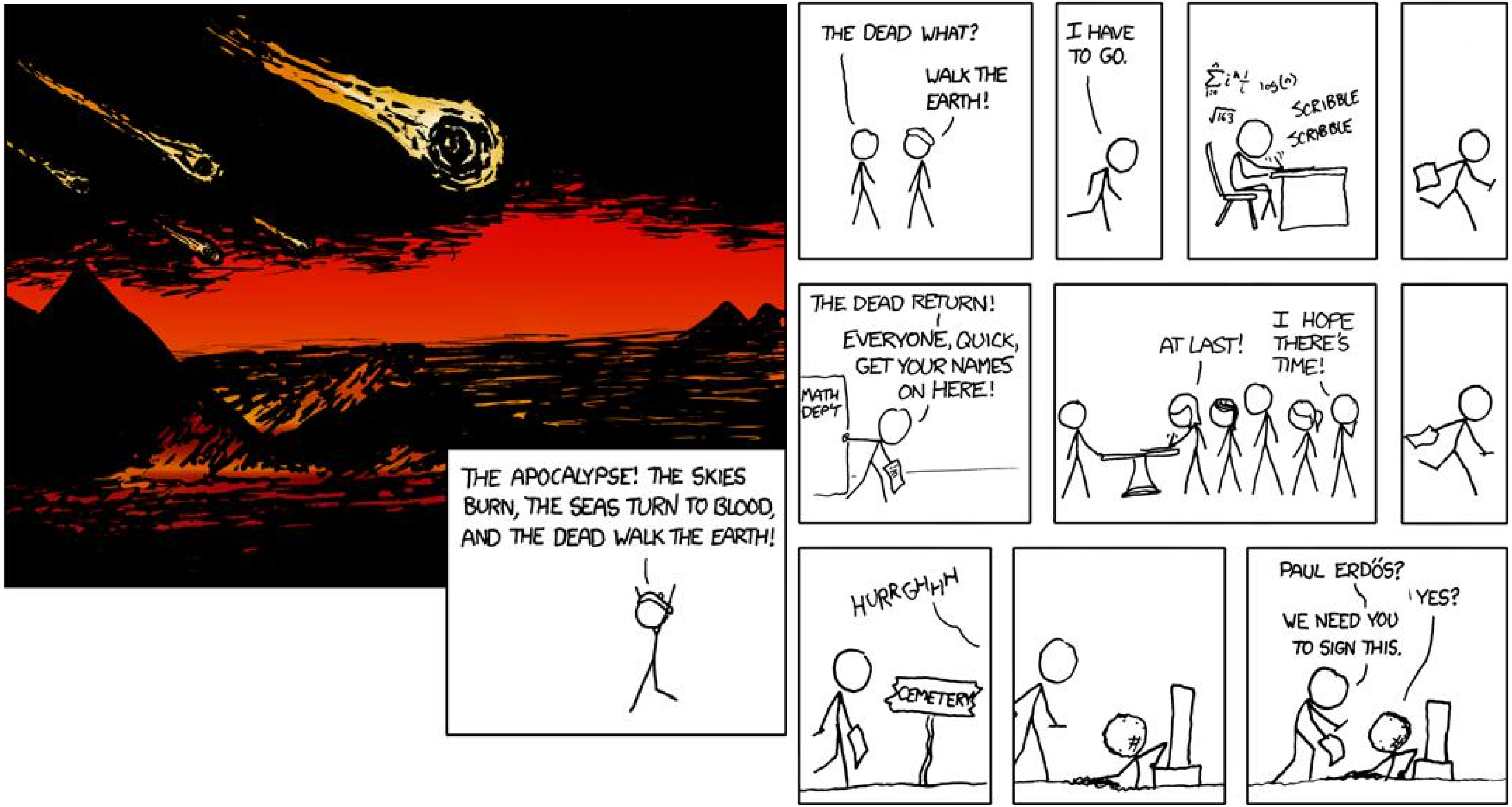
Un esempio: Erdős Number

- Paul Erdős (1913-1996):
  - Matematico ungherese incredibilmente prolifico
  - 1500+ articoli, 500+ coautori
  - Ha studiato la teoria dei grafi!

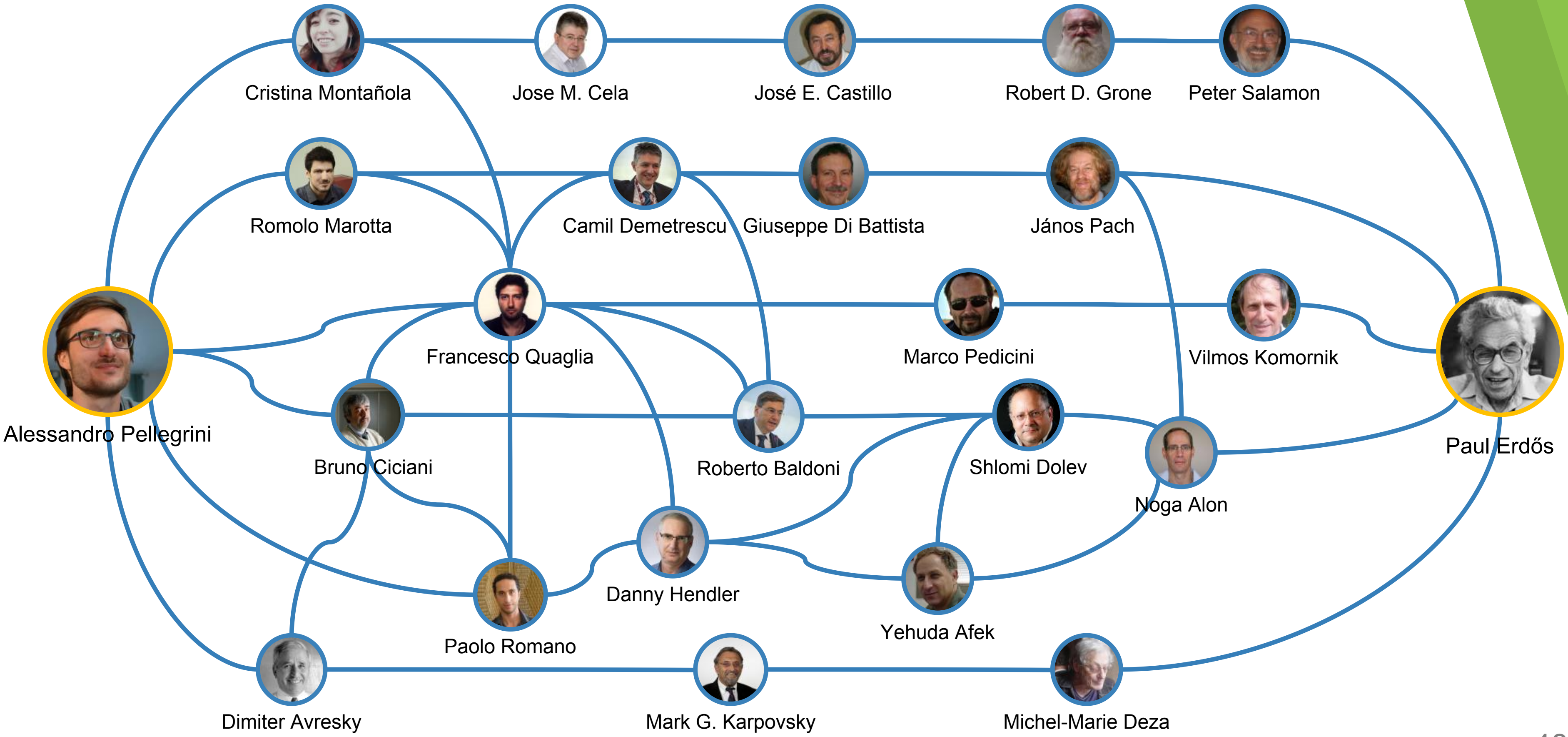


- Numero di Erdős
  - Erdős ha valore erdős = 0
  - I co-autori ei Erdős hanno erdős = 1
  - Se X è co-autore di qualcuno con erdős = k e non è coautore di nessun altro con erdős < k, allora X ha erdős = k + 1
  - Le persone non raggiunte, hanno erdős = +∞
  - È una forma di *grado di separazione*
  - Tutti gli archi sul grafo hanno peso 1

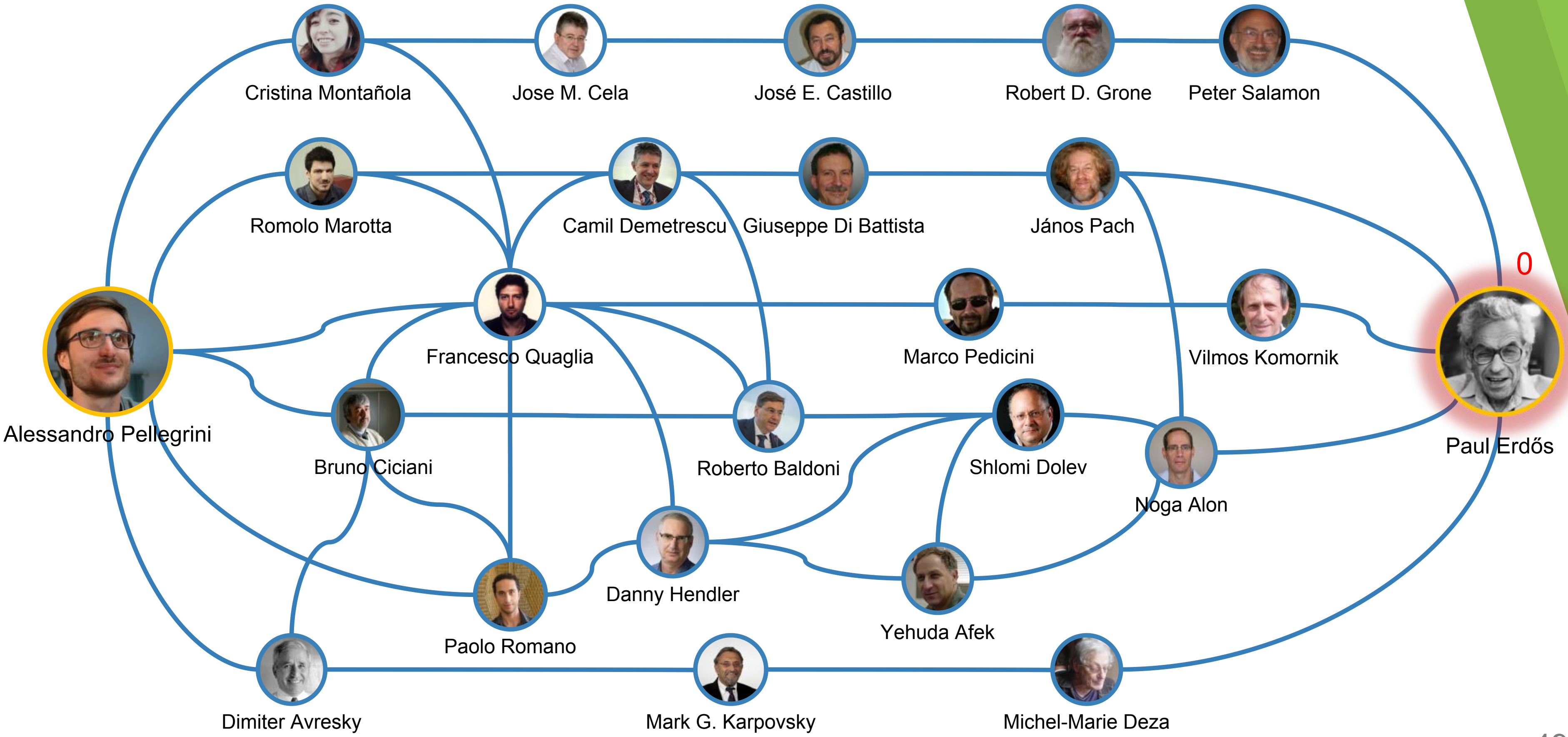
Erdős Number



Erdős Number

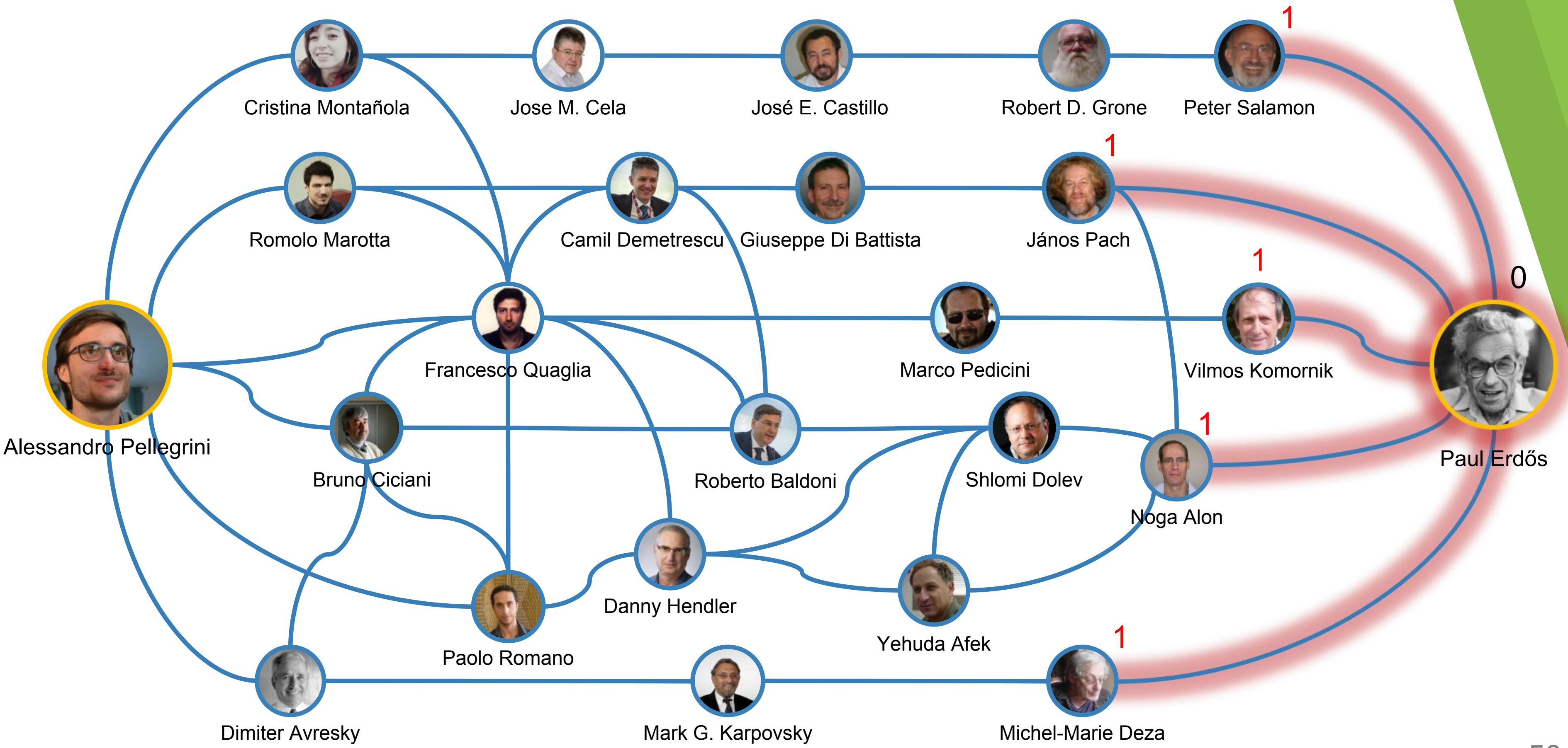


Erdős Number



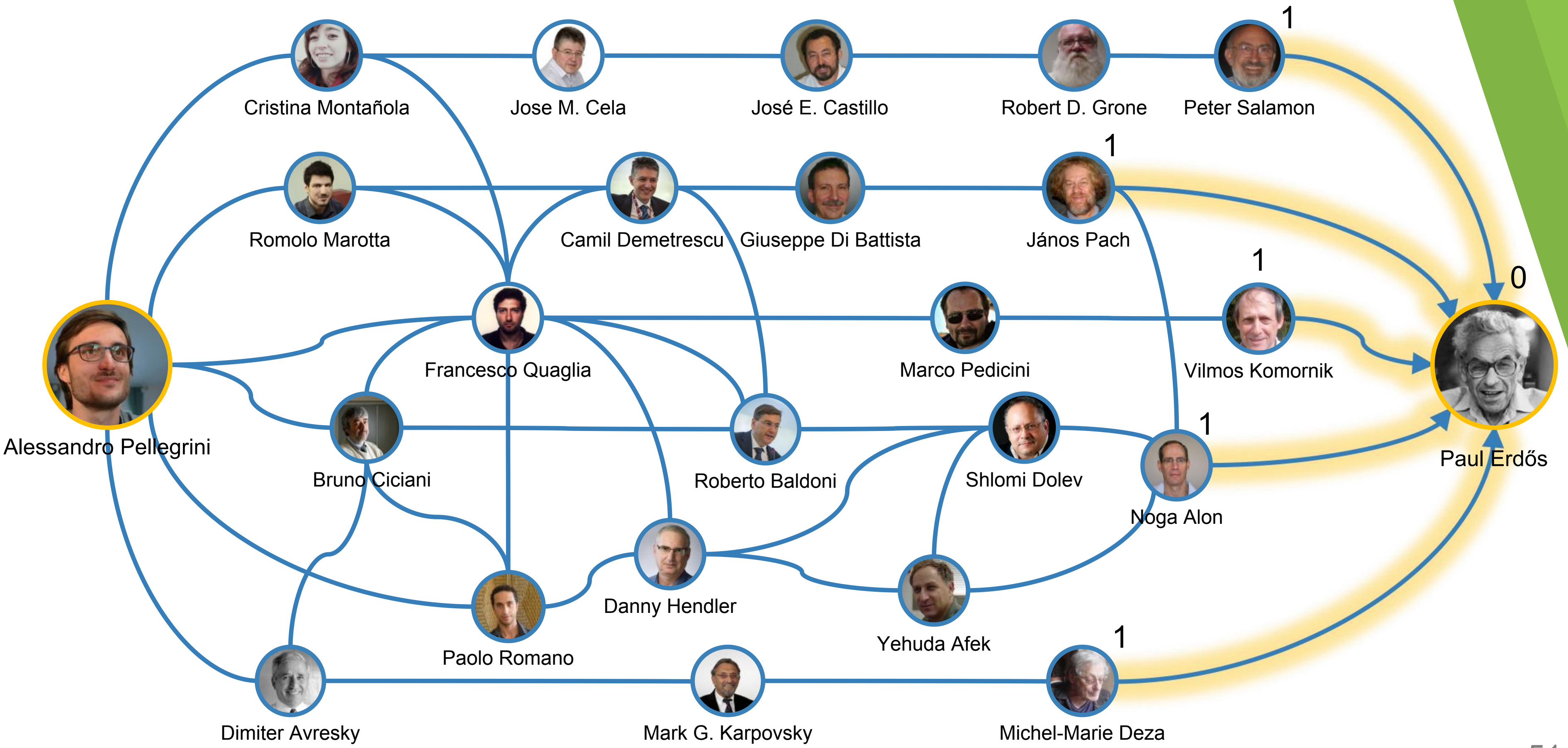
Queue = { }

Erdős Number



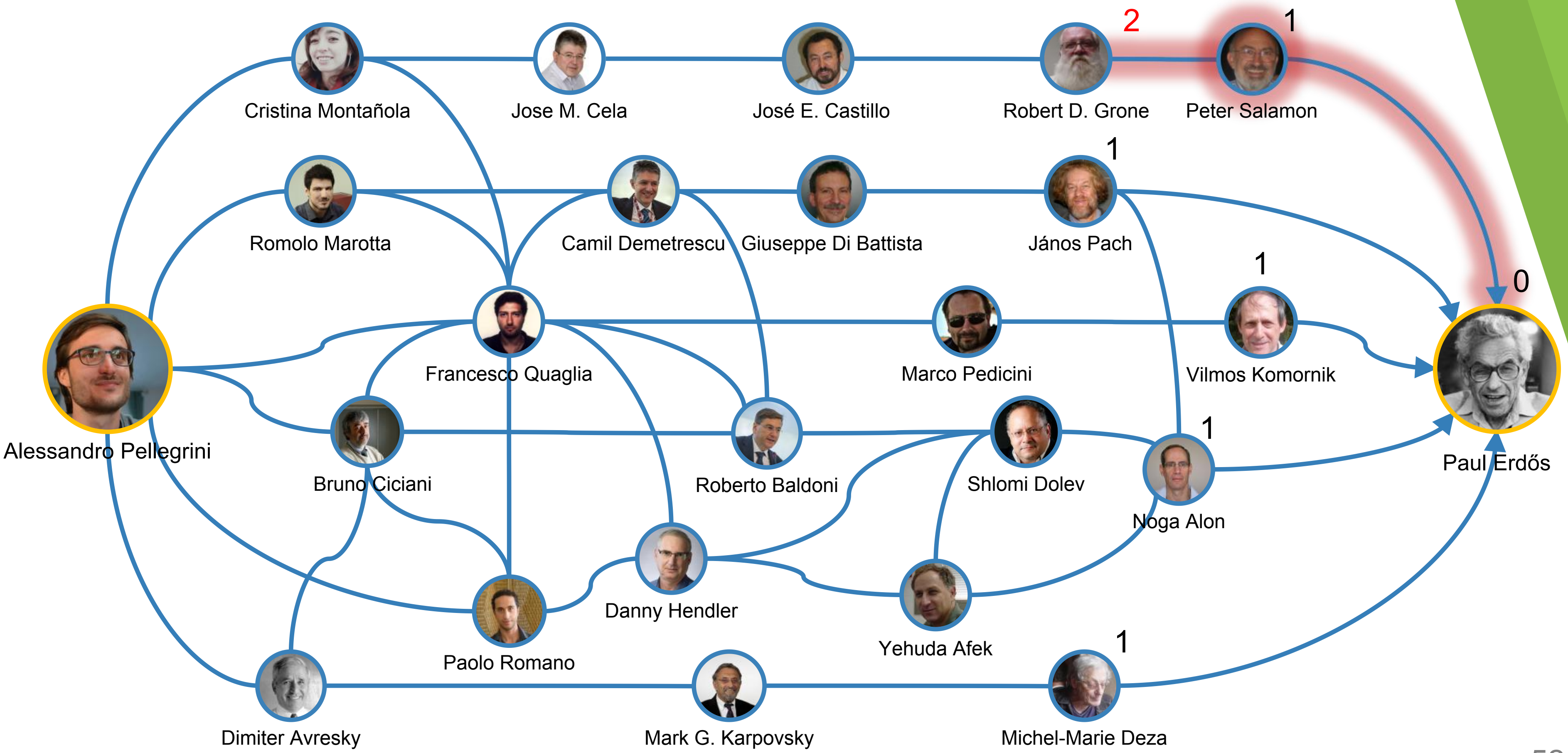
Queue = { Salamon, Pach, Komornik, Alon, Deza }

Erdős Number



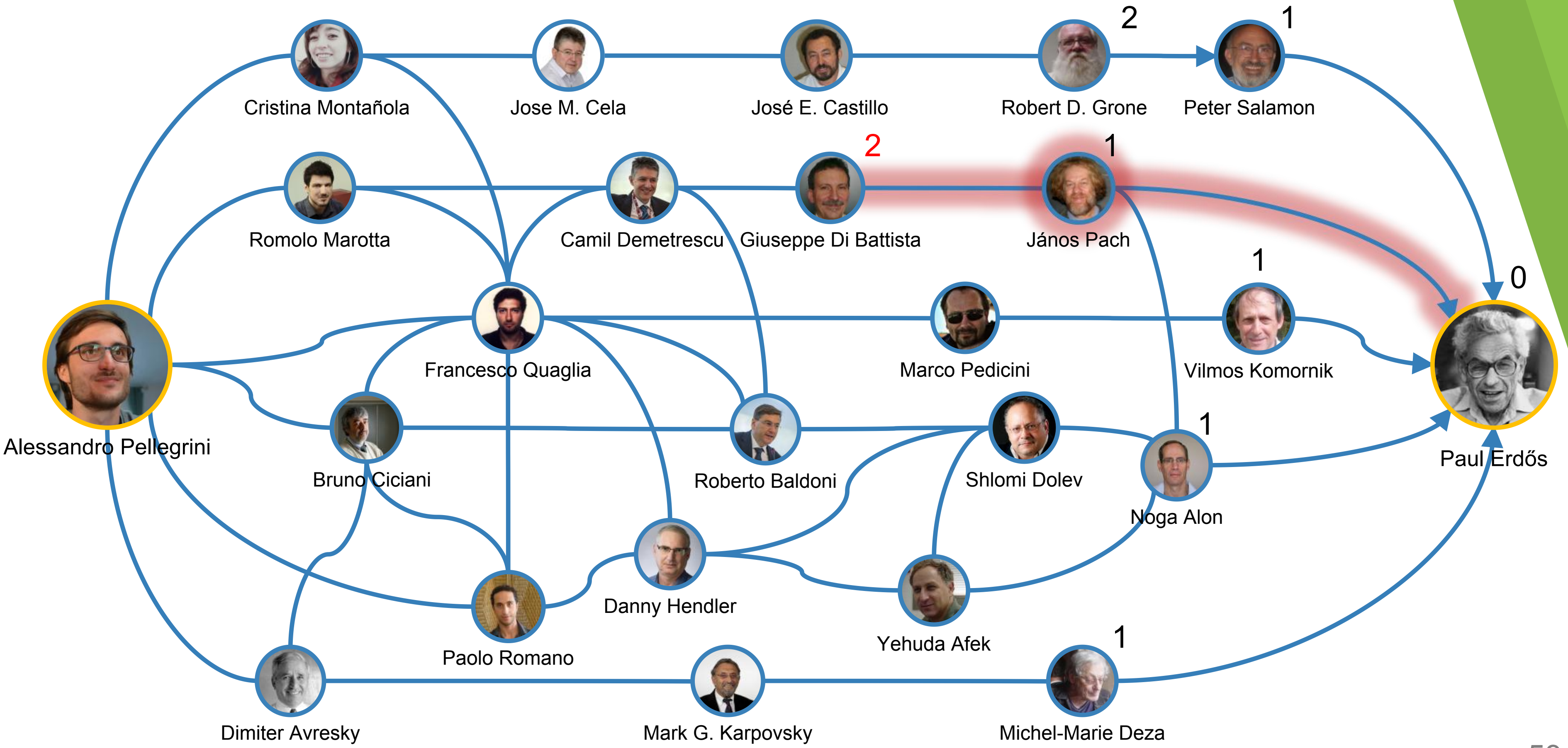
Queue = { Salamon, Pach, Komornik, Alon, Deza }

Erdős Number



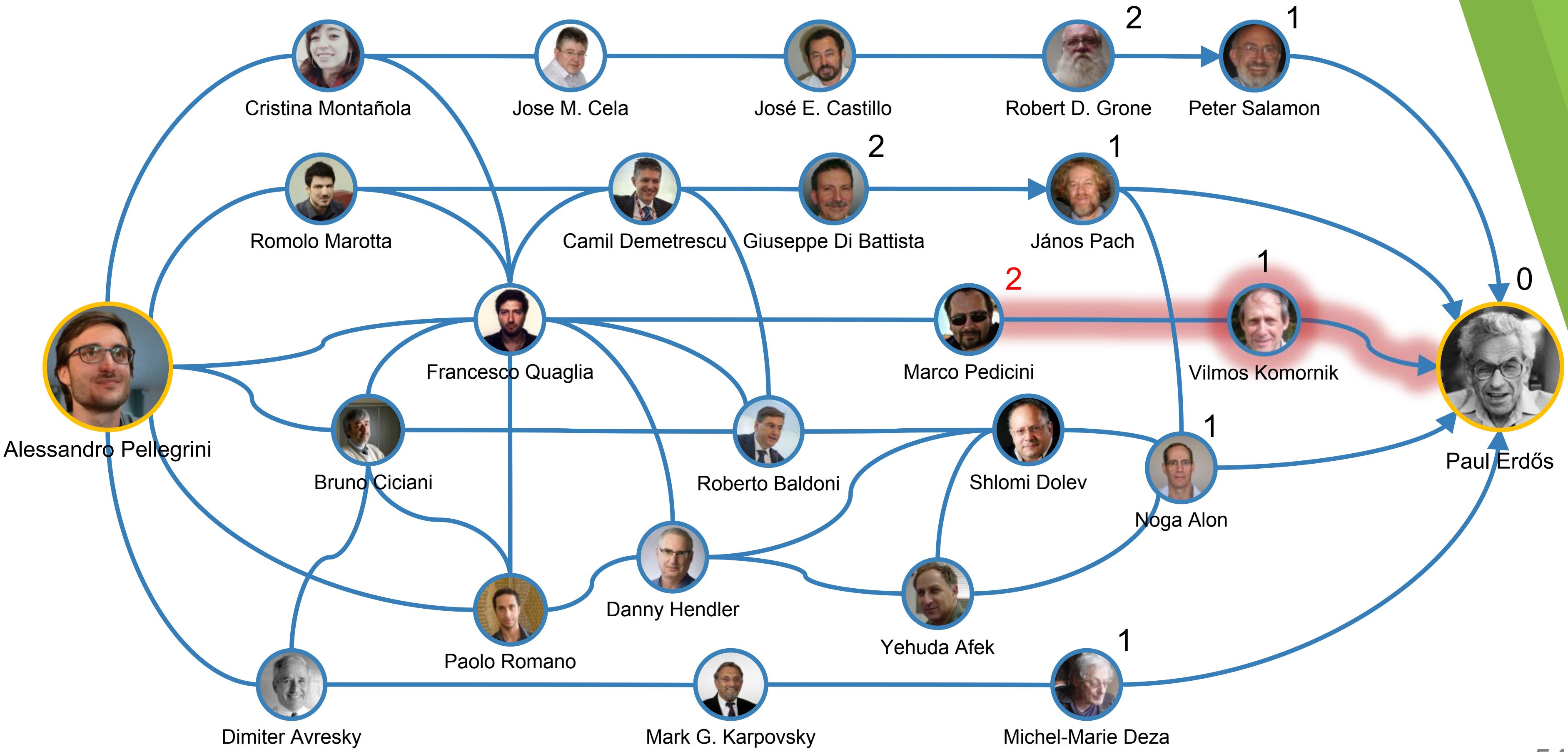
Queue = { Pach, Komornik, Alon, Deza, Grone }

Erdős Number



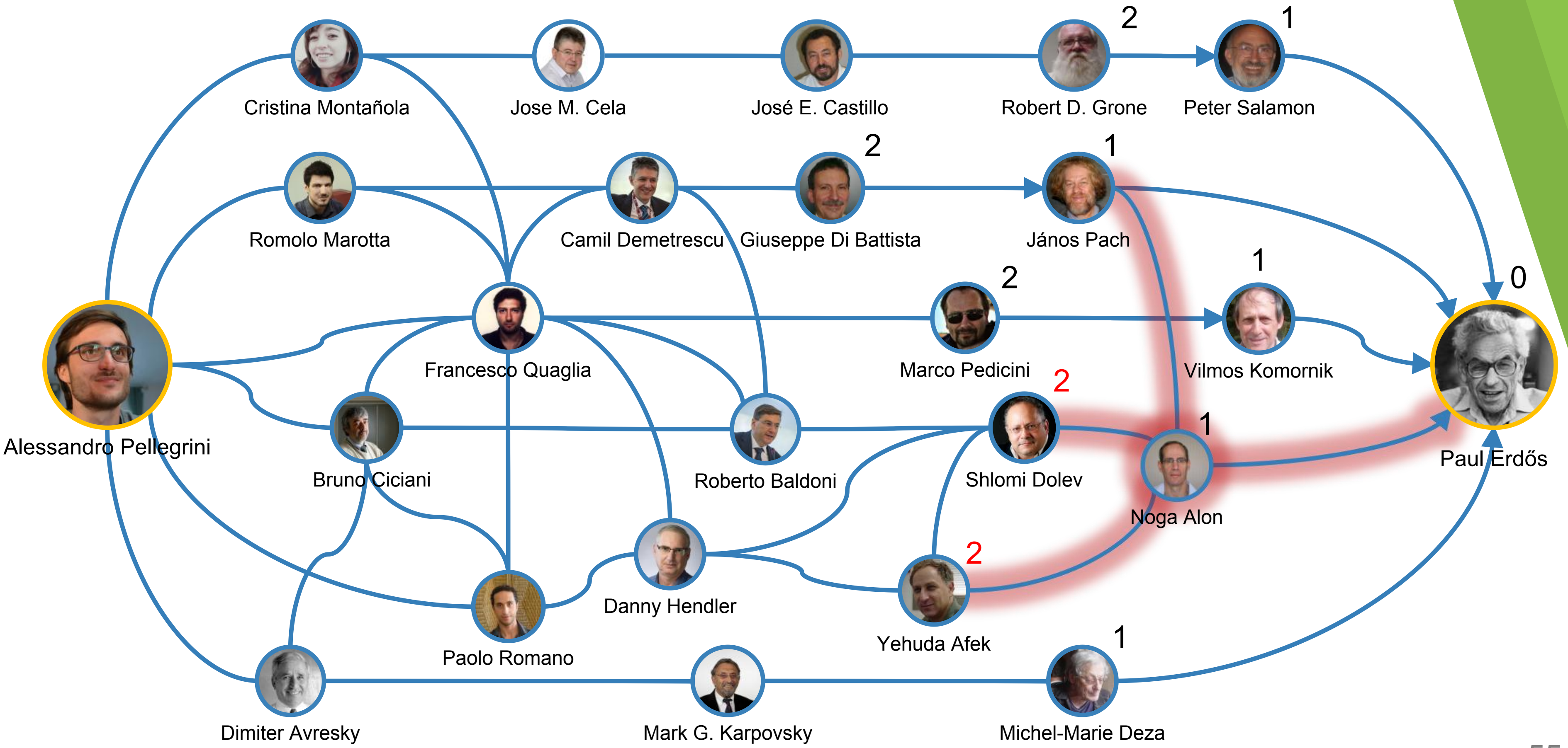
Queue = { Komornik, Alon, Deza, Grone, Di Battista }

Erdős Number



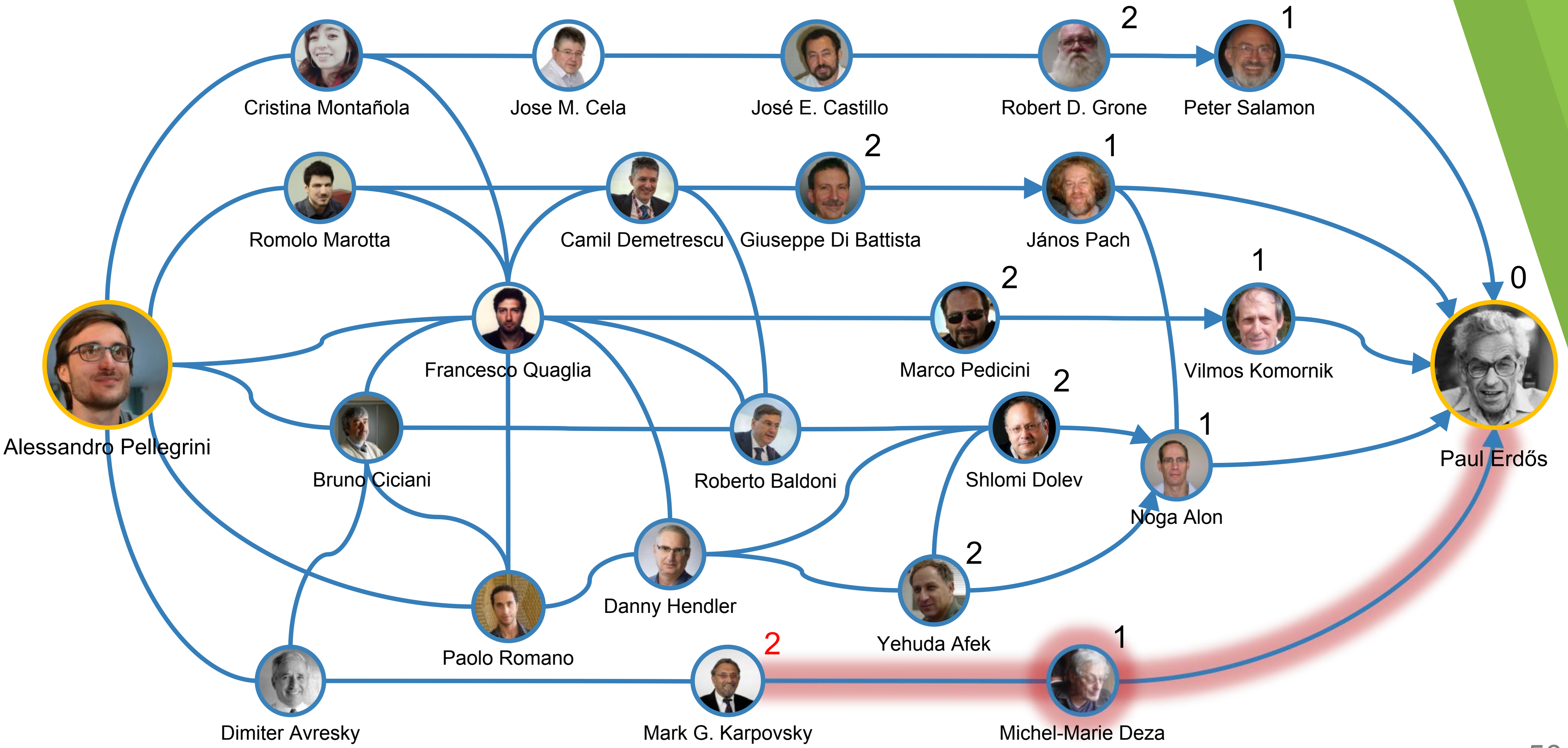
Queue = { Alon, Deza, Grone, Di Battista, Pedicini }

Erdős Number



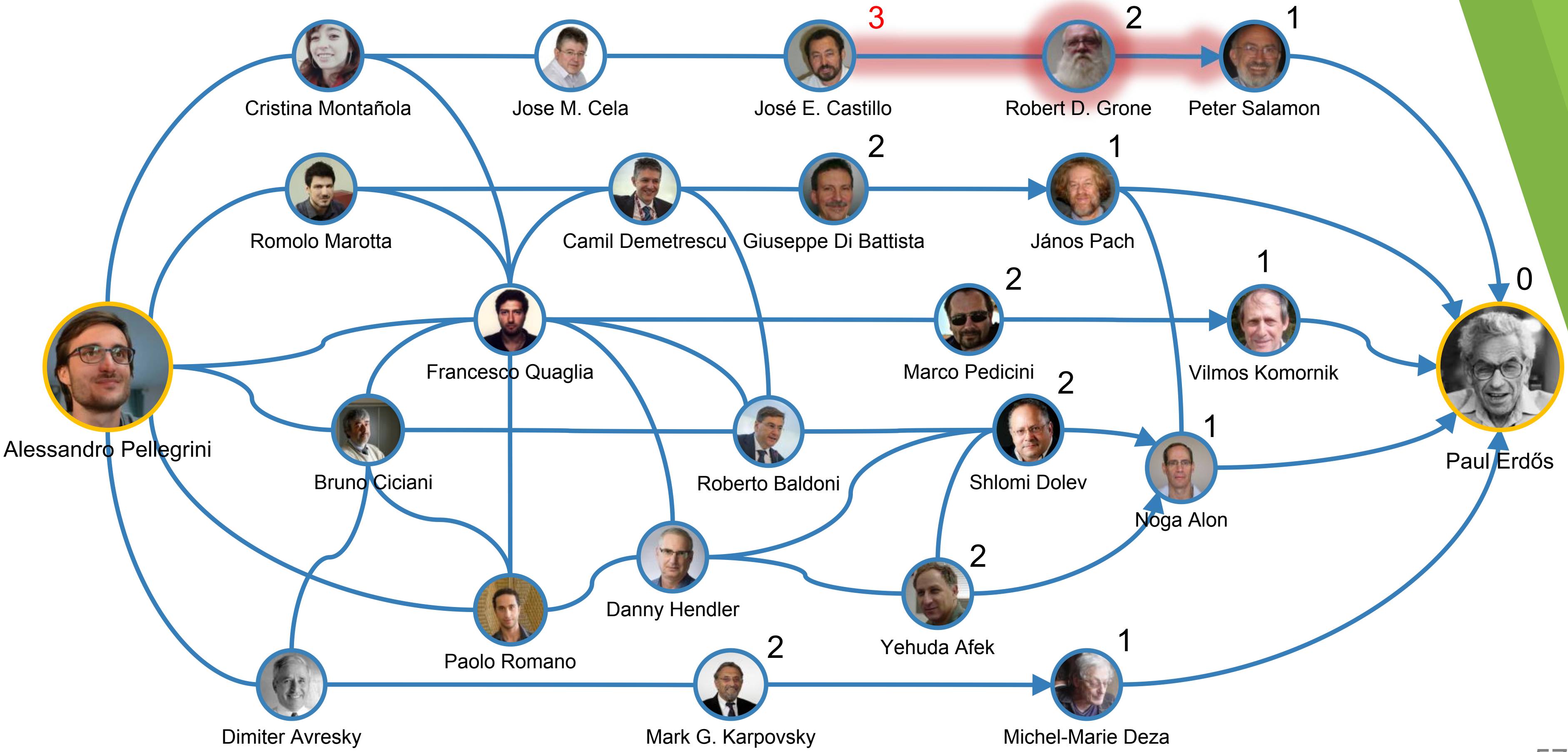
Queue = { Deza, Grone, Di Battista, Pedicini, Dolev, Afek }

Erdős Number



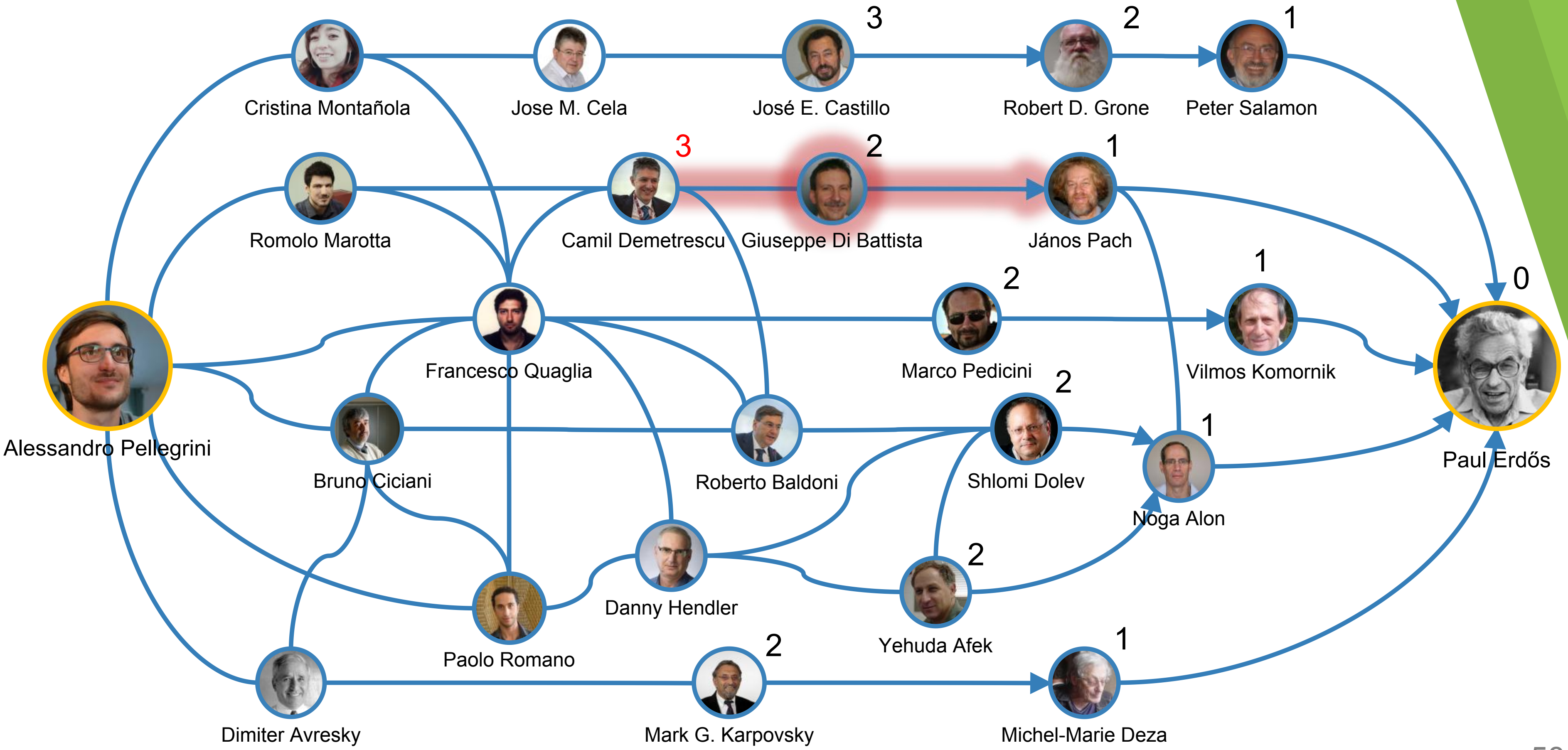
Queue = { Grone, Di Battista, Pedicini, Dolev, Afek, Karpovsky }

Erdős Number



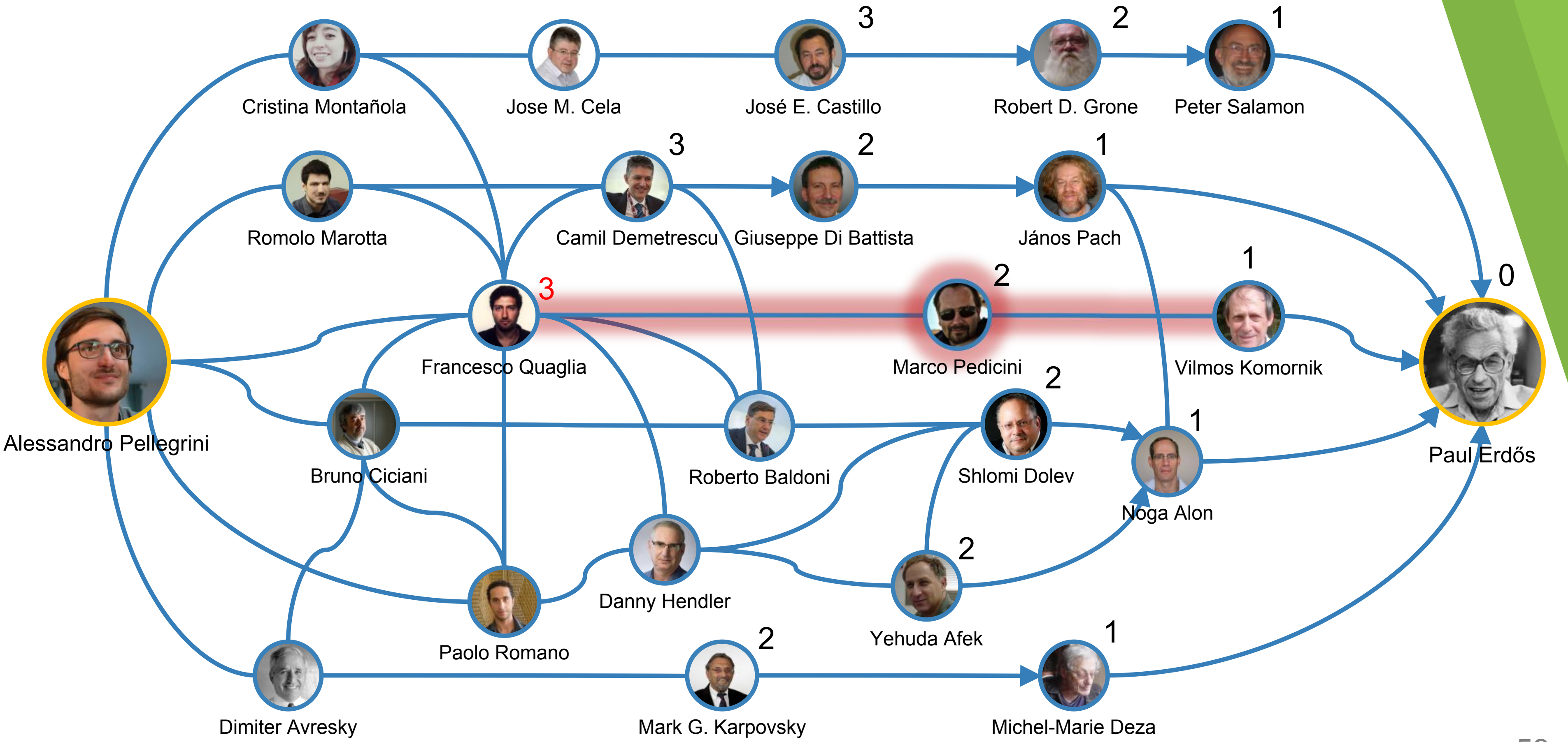
Queue = { Di Battista, Pedicini, Dolev, Afek, Karpovsky, Castillo }

Erdős Number



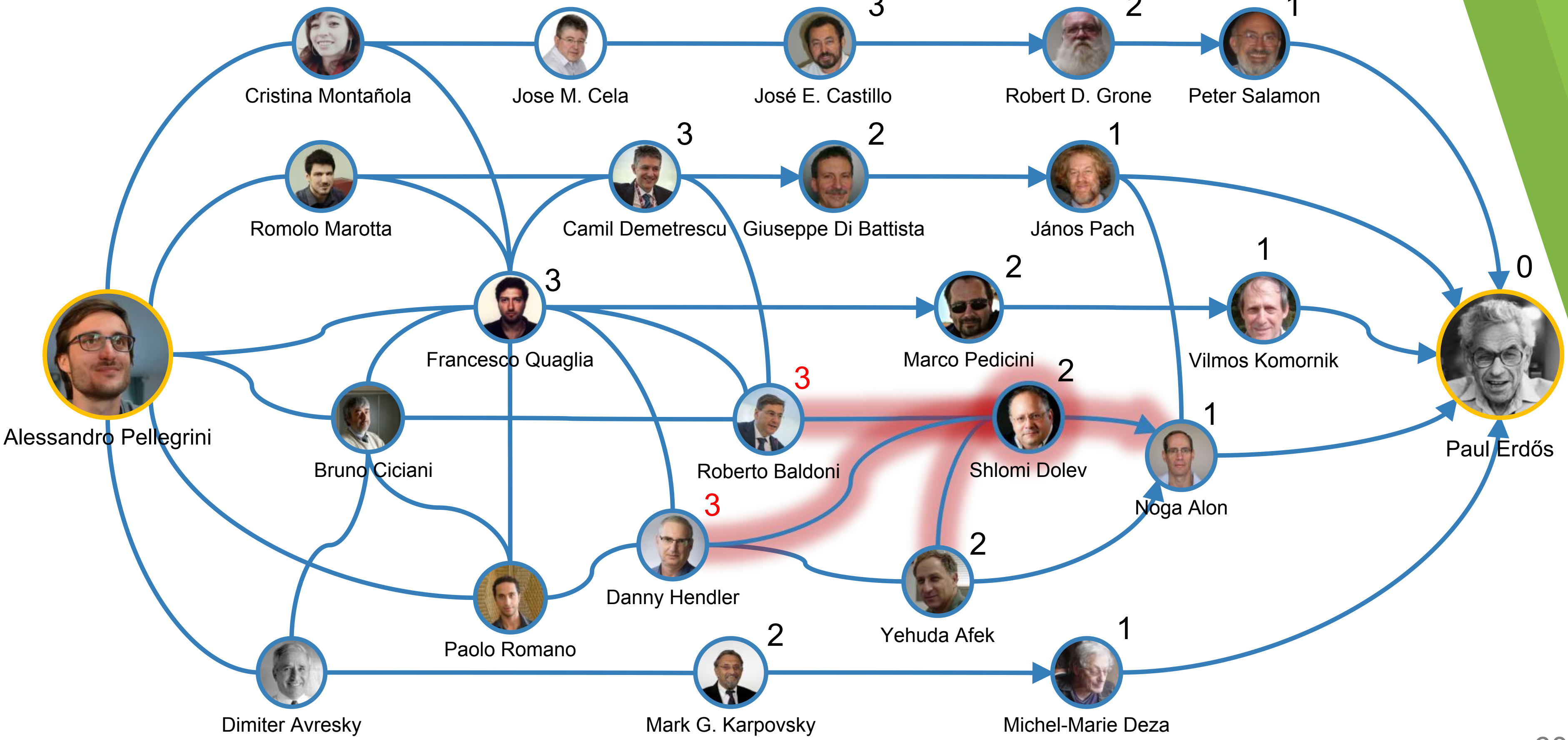
Queue = { Pedicini, Dolev, Afek, Karpovsky, Castillo, Demetrescu }

Erdős Number



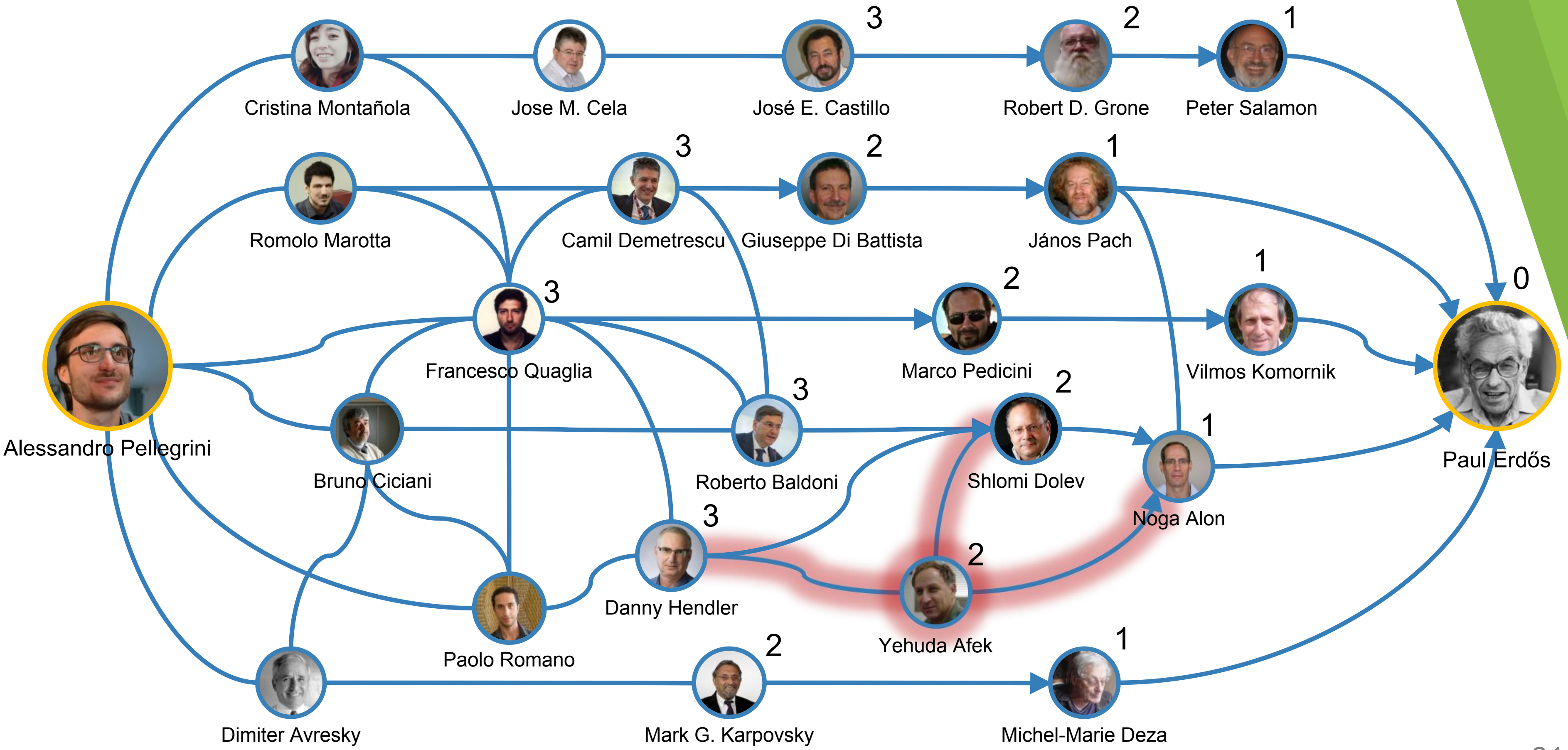
Queue = { Dolev, Afek, Karpovsky, Castillo, Demetrescu, Quaglia }

Erdős Number



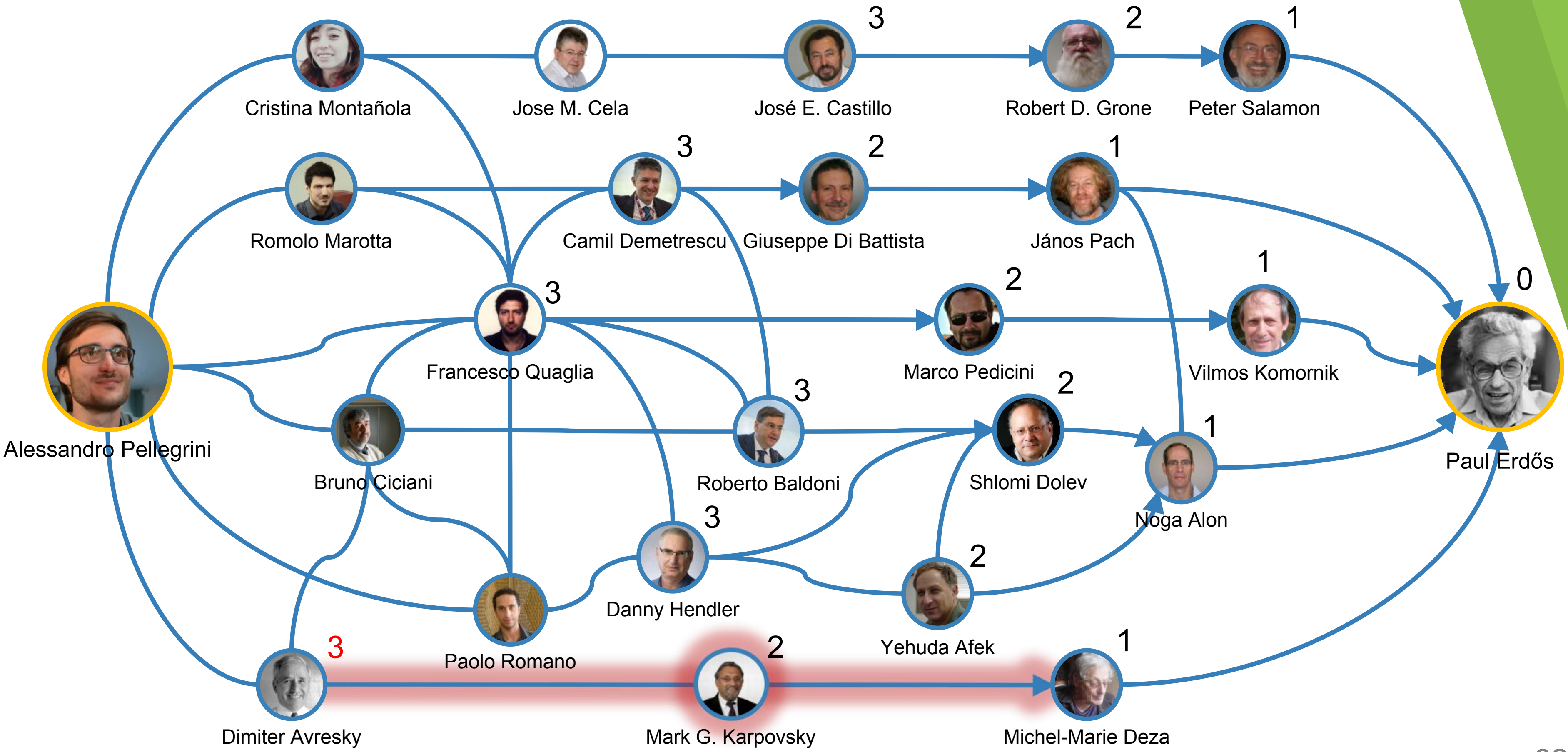
Queue = { Afek, Karpovsky, Castillo, Demetrescu, Quaglia, Baldoni, Hendler }

Erdős Number



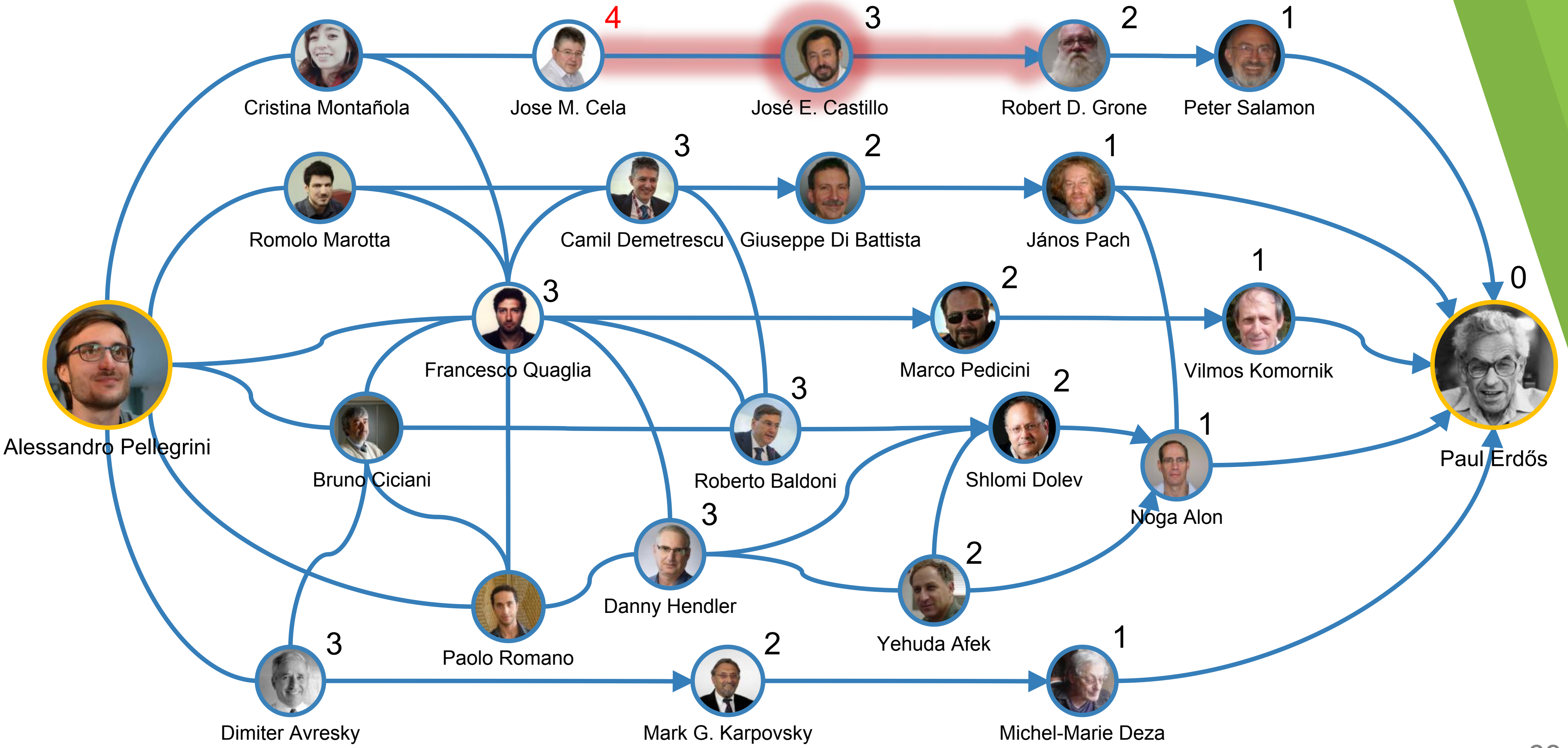
Queue = { Karpovsky, Castillo, Demetrescu, Quaglia, Baldoni, Hendler }

Erdős Number



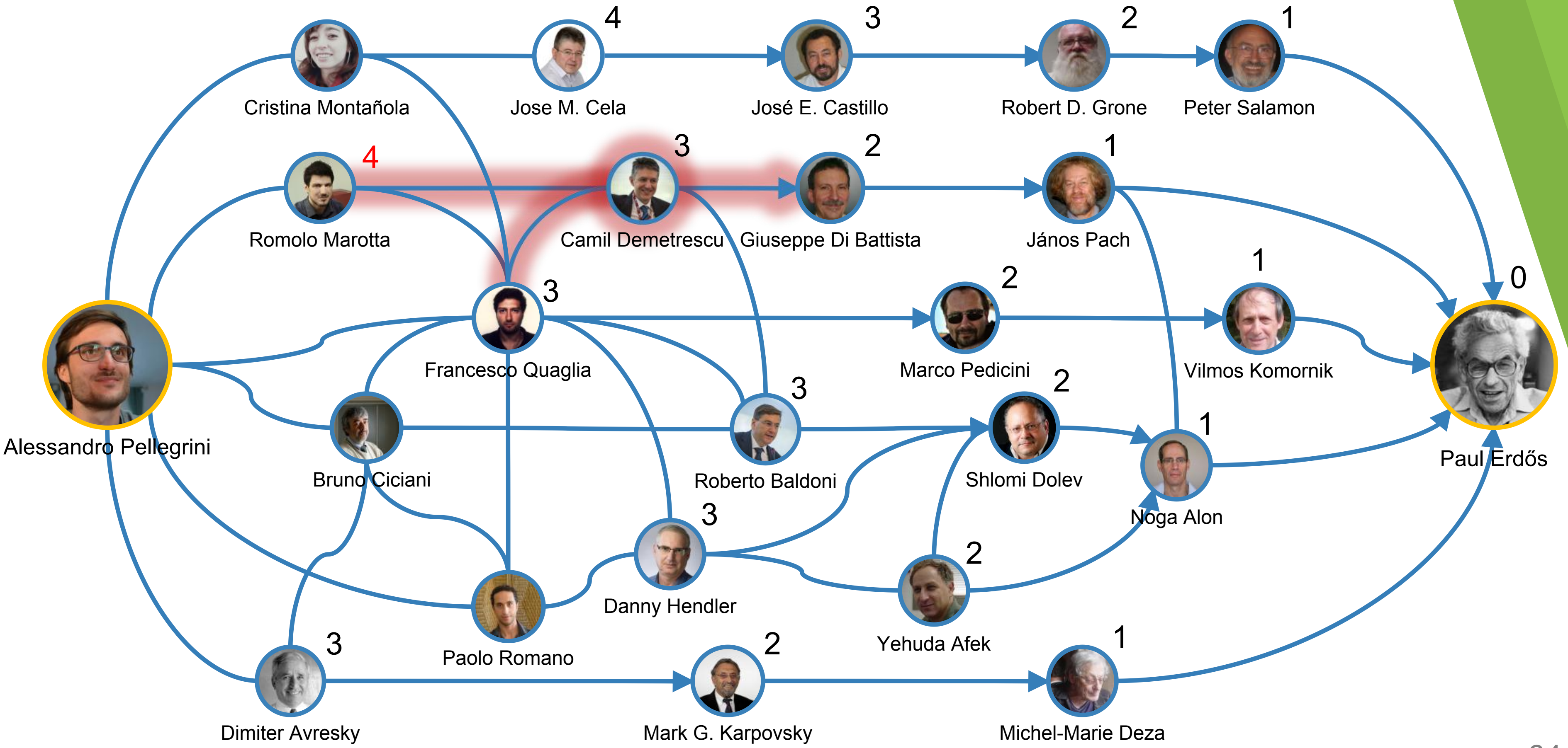
Queue = { Castillo, Demetrescu, Quaglia, Baldoni, Hendler }

Erdős Number



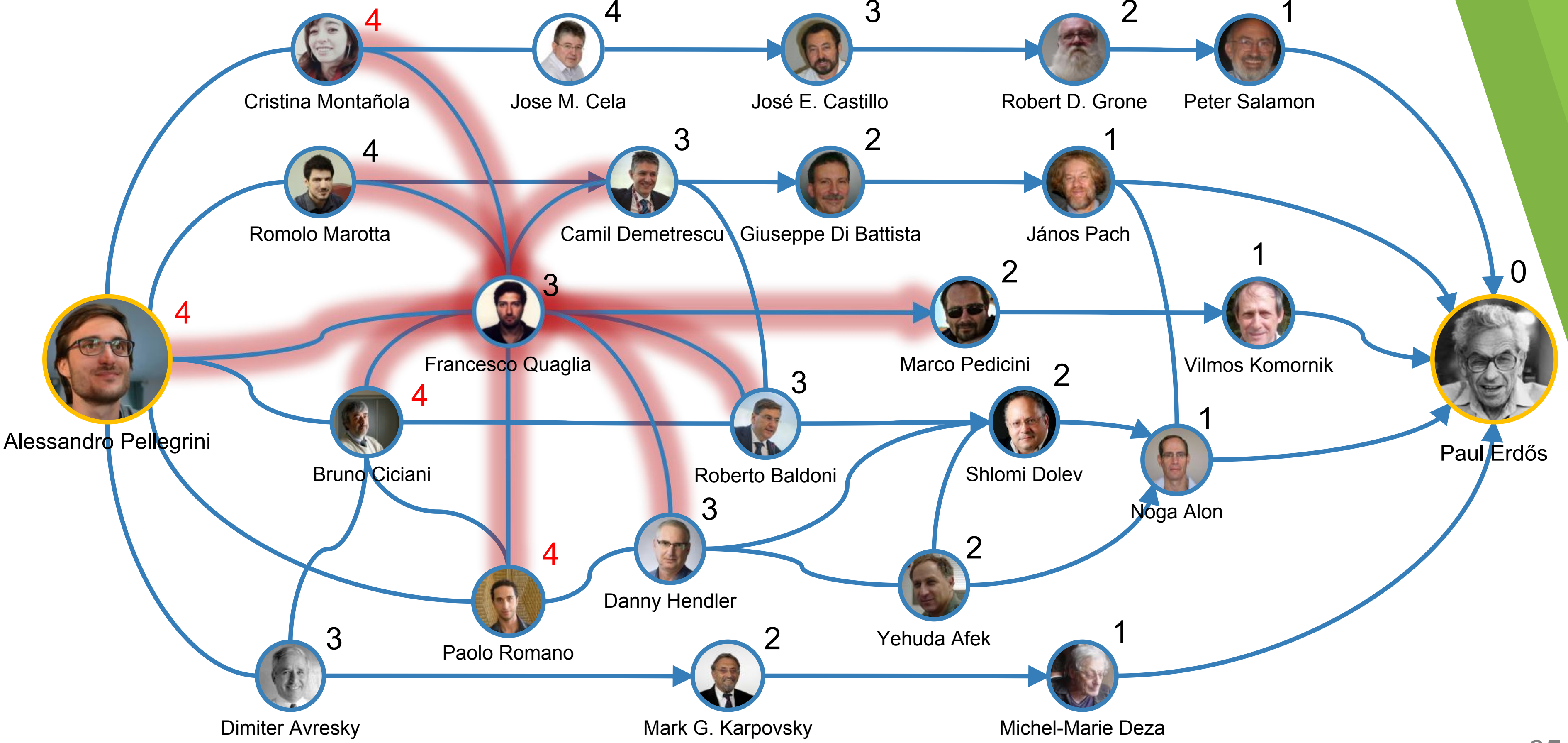
Queue = { Demetrescu, Quaglia, Baldoni, Hendler }

Erdős Number



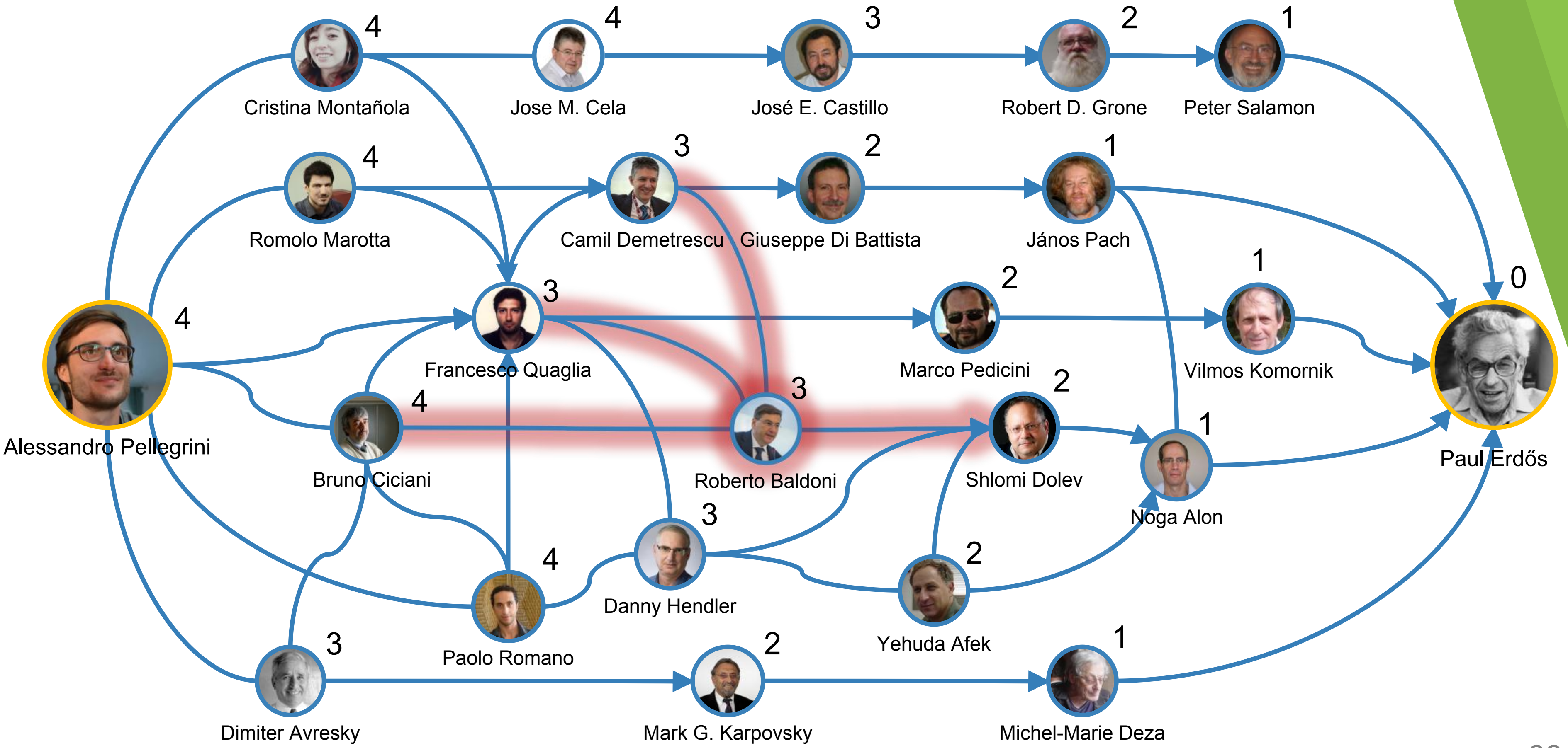
Queue = { Quaglia, Baldoni, Hendler, Marotta }

Erdős Number



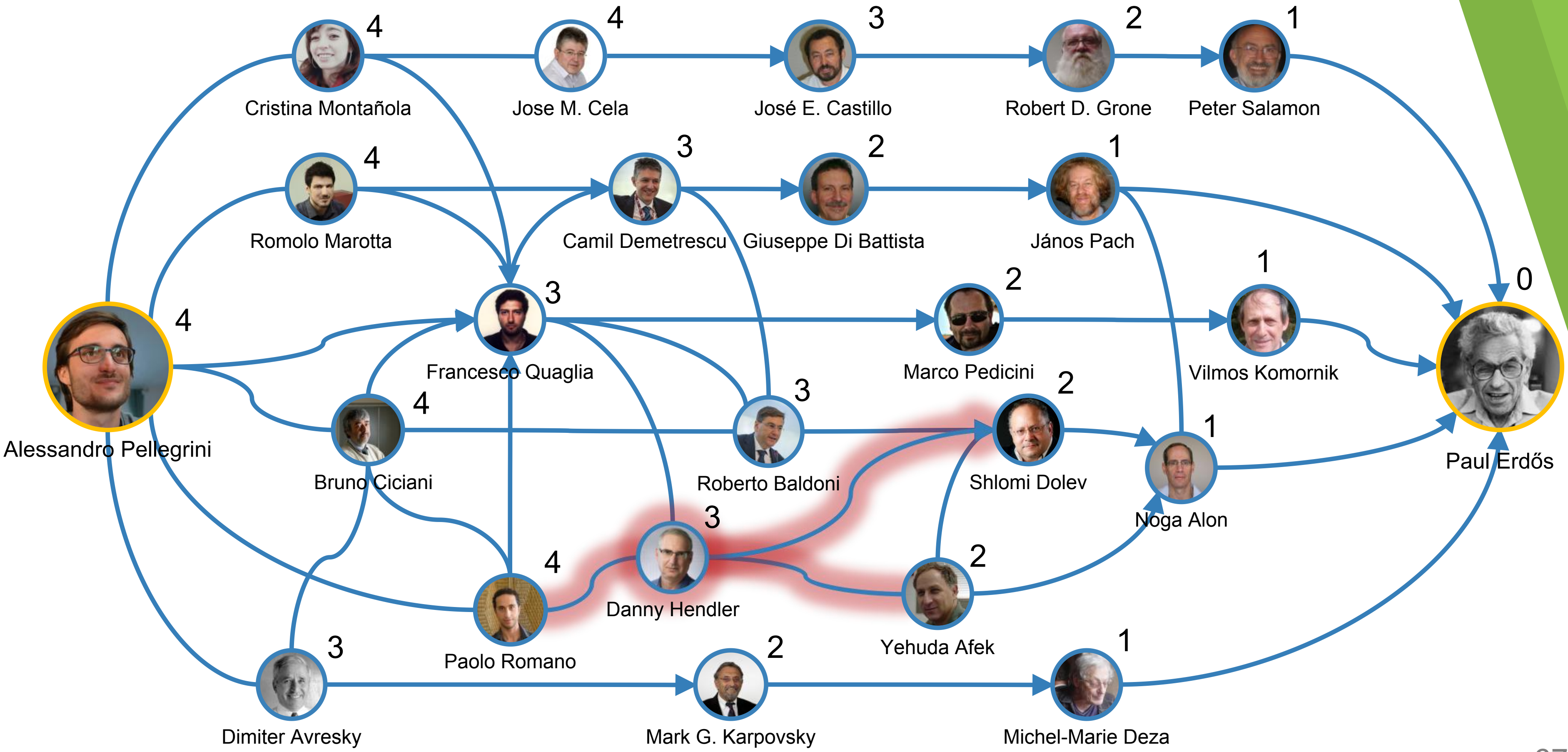
Queue = { Baldoni, Hendler, Marotta, Montañola, Pellegrini, Ciciani, Romano }

Erdős Number



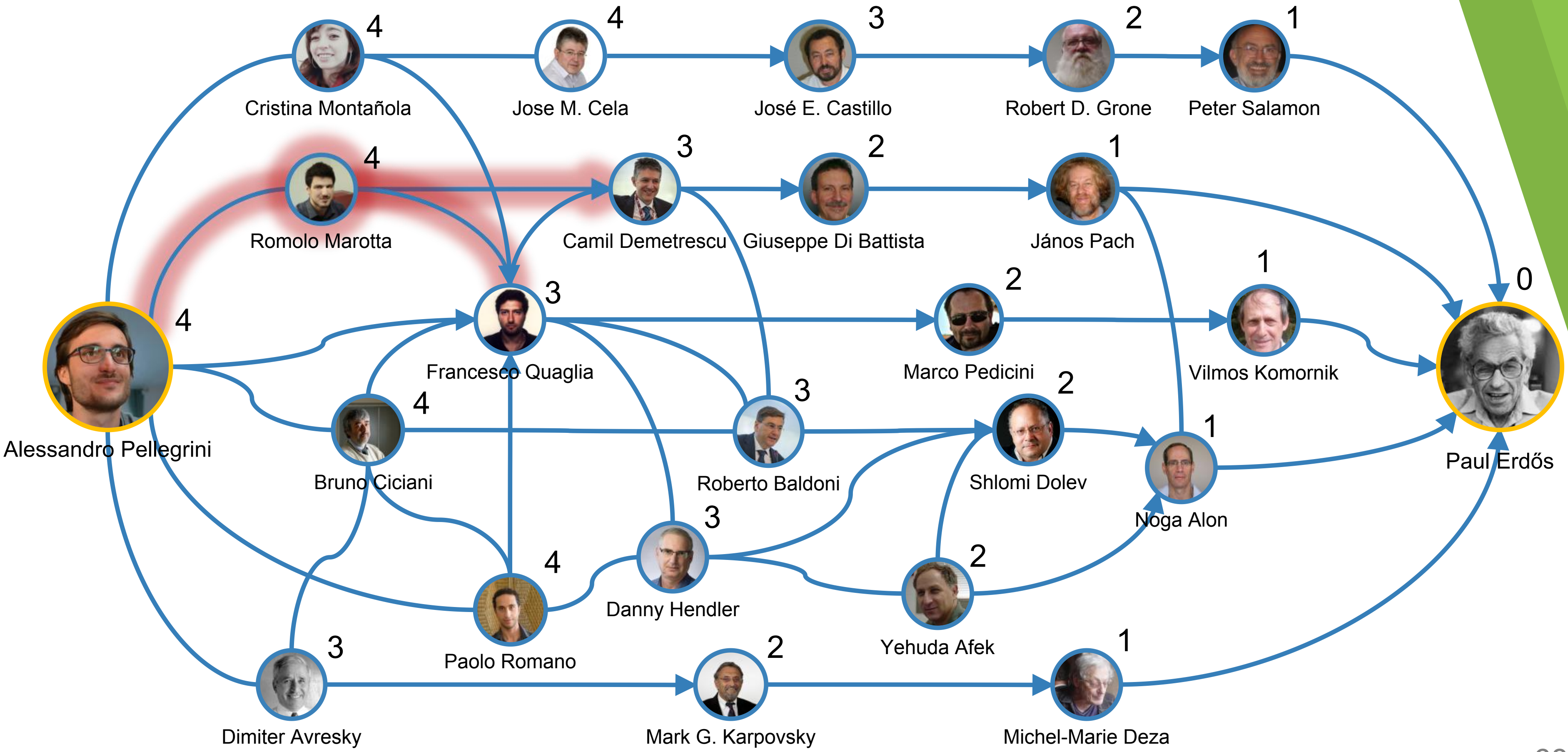
Queue = { Hendler, Marotta, Montañola, Pellegrini, Ciciani, Romano }

Erdős Number



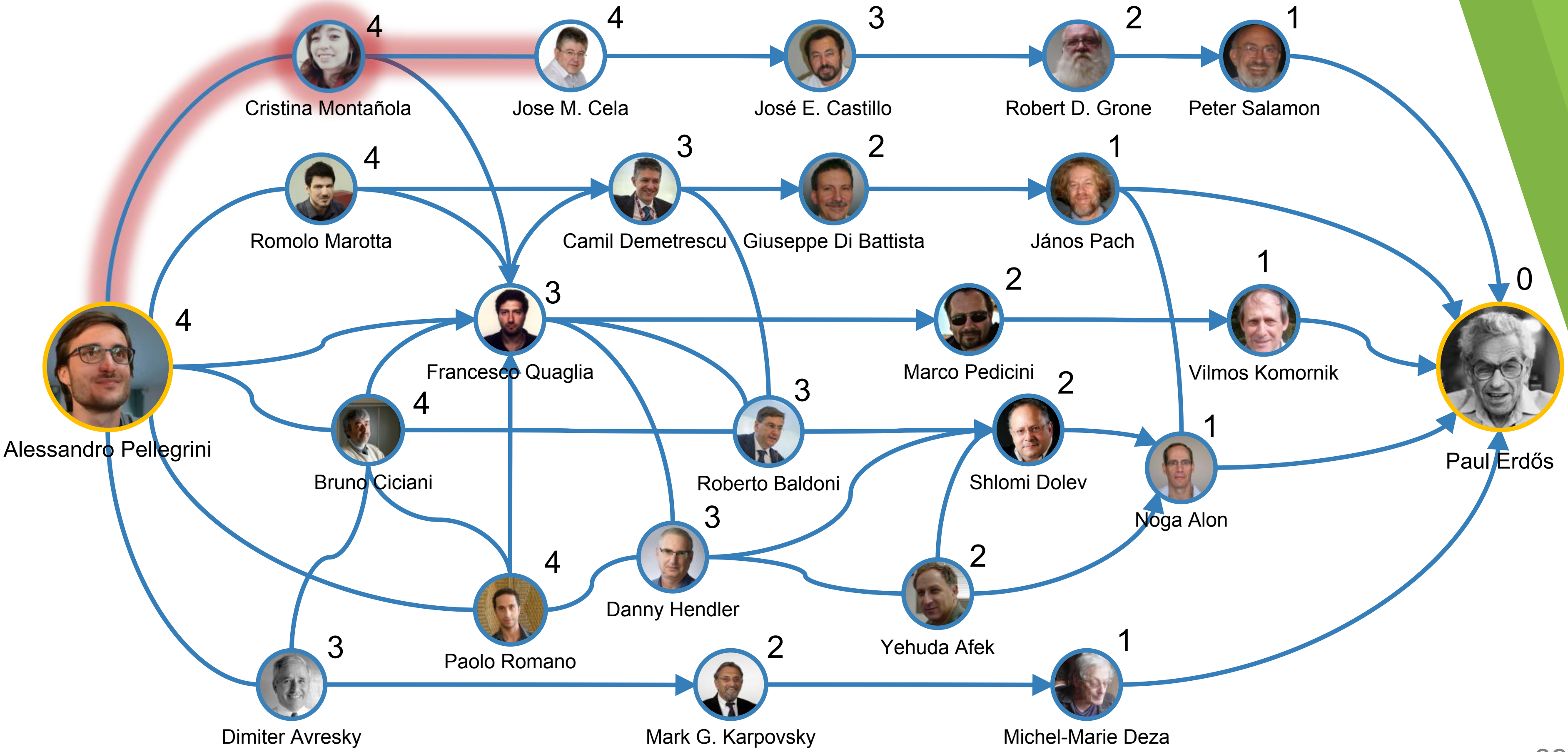
Queue = { Marotta, Montañola, Pellegrini, Ciciani, Romano }

Erdős Number



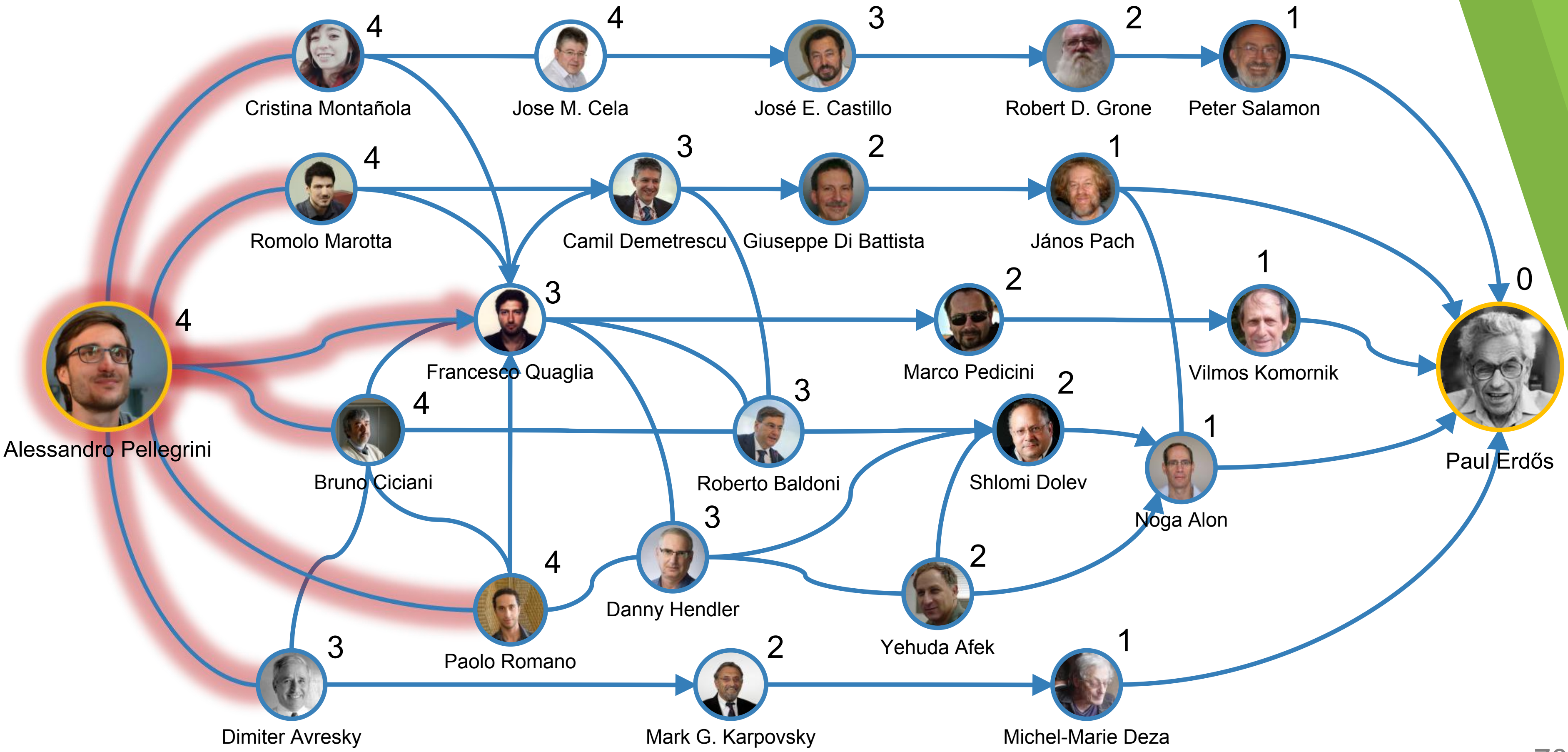
Queue = { Montañola, Pellegrini, Ciciani, Romano }

Erdős Number



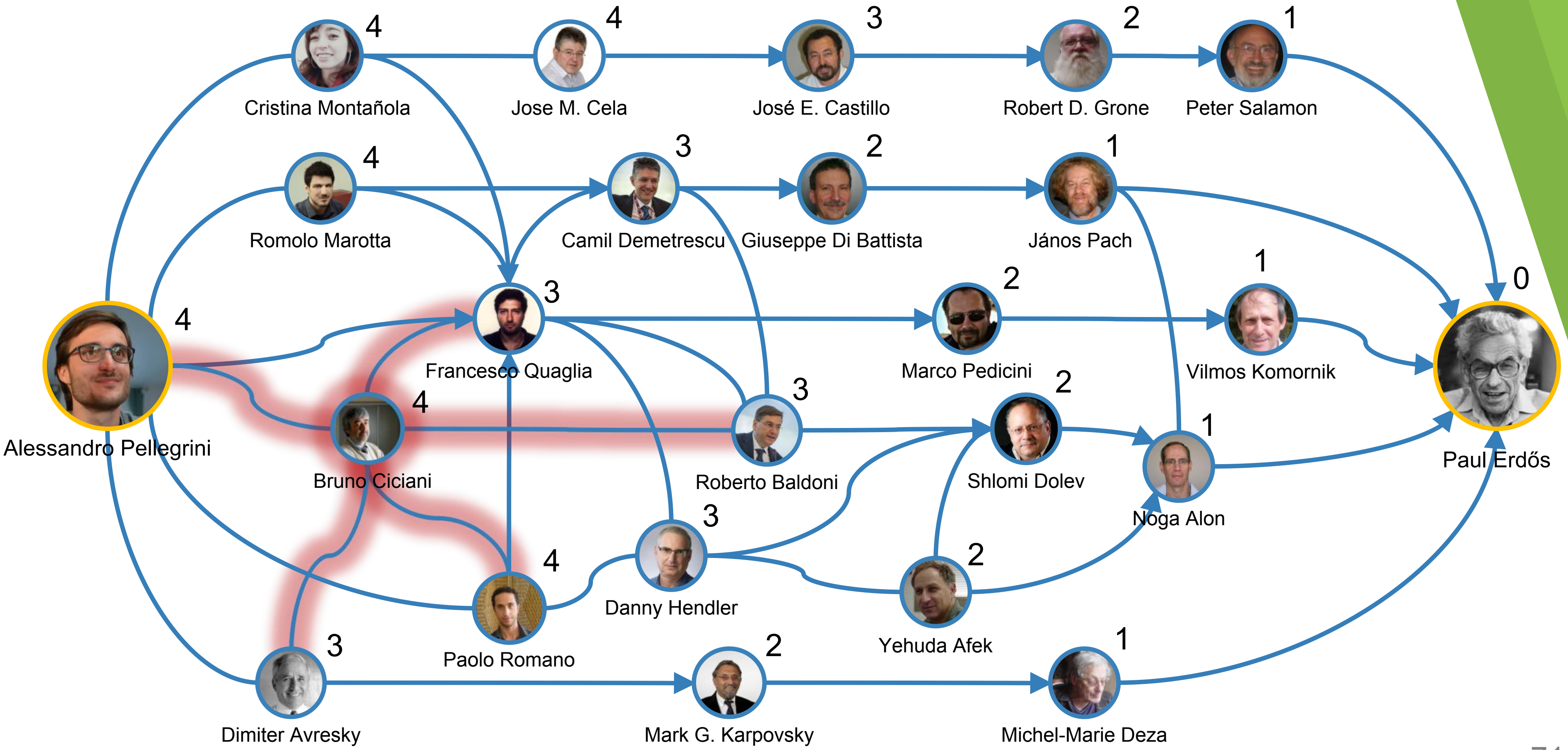
Queue = { Pellegrini, Ciciani, Romano }

Erdős Number



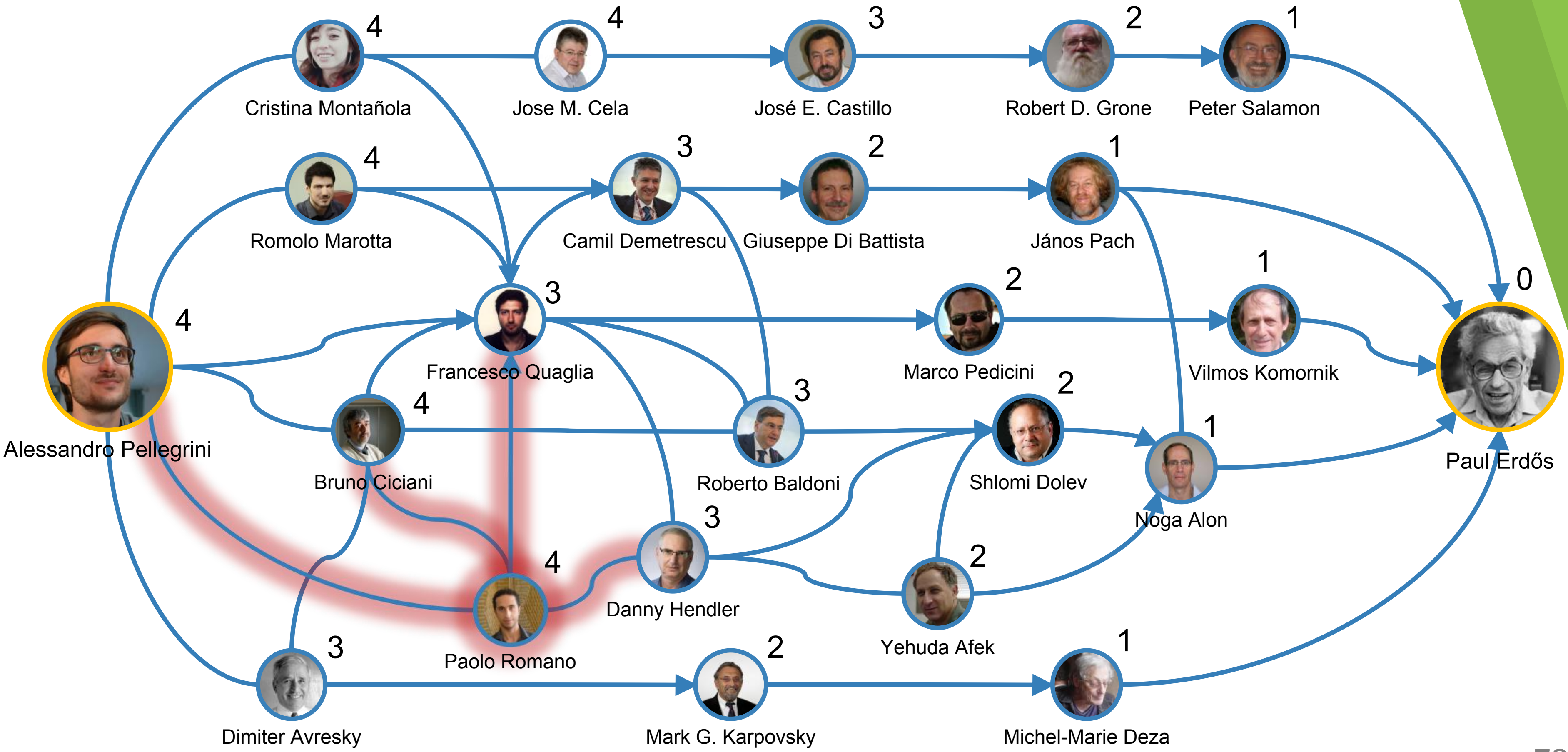
Queue = { Ciciani, Romano }

Erdős Number



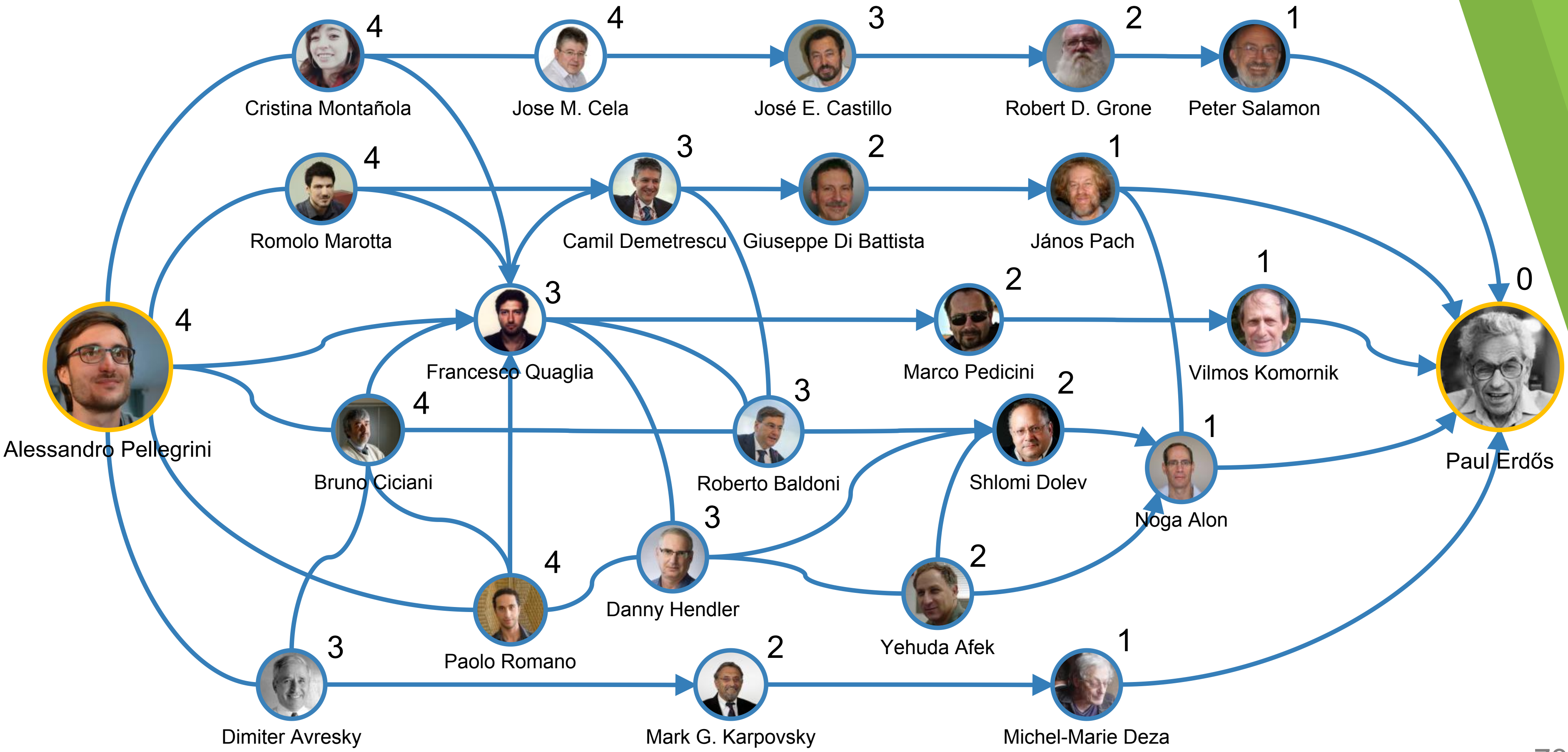
Queue = { Romano }

Erdős Number



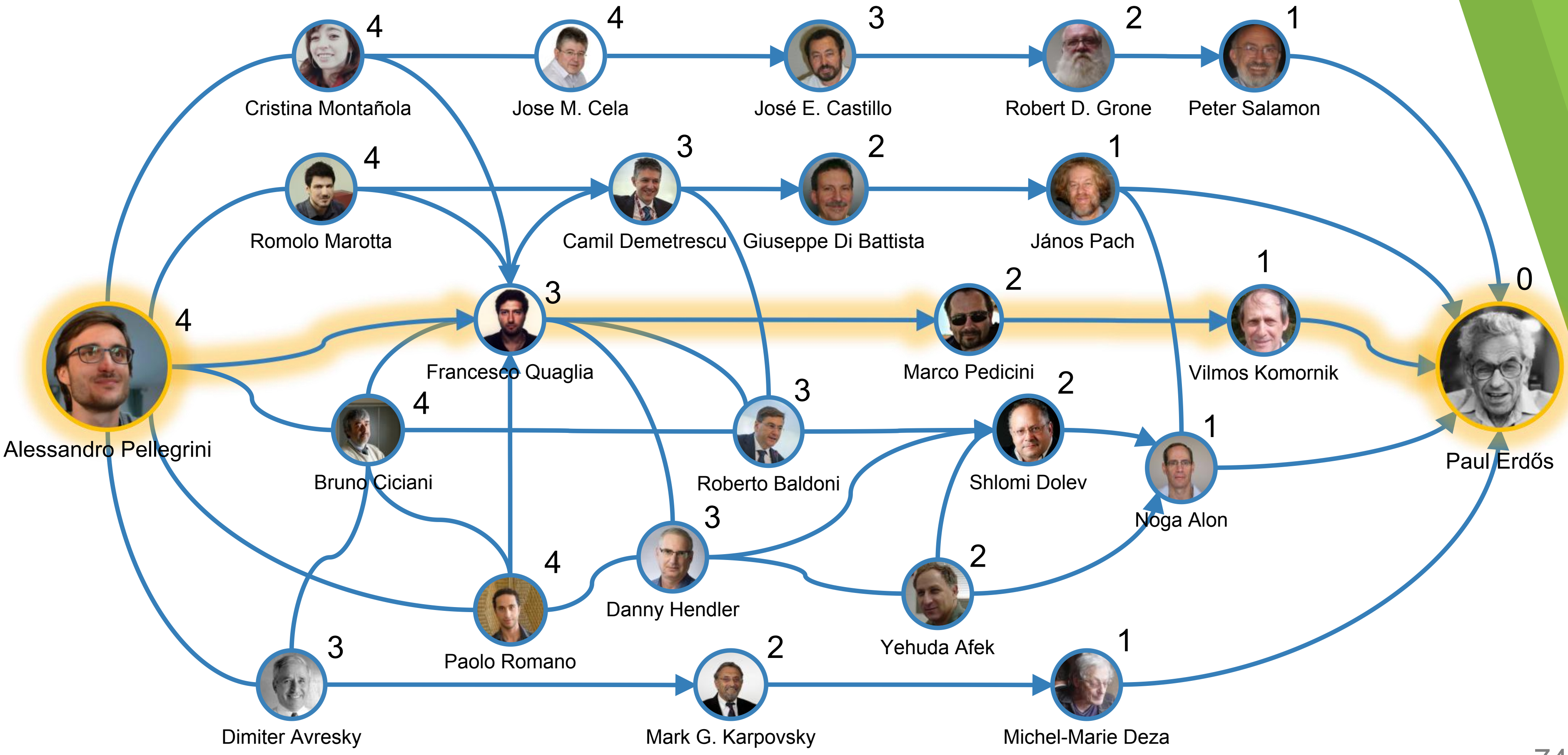
Queue = { }

Erdős Number

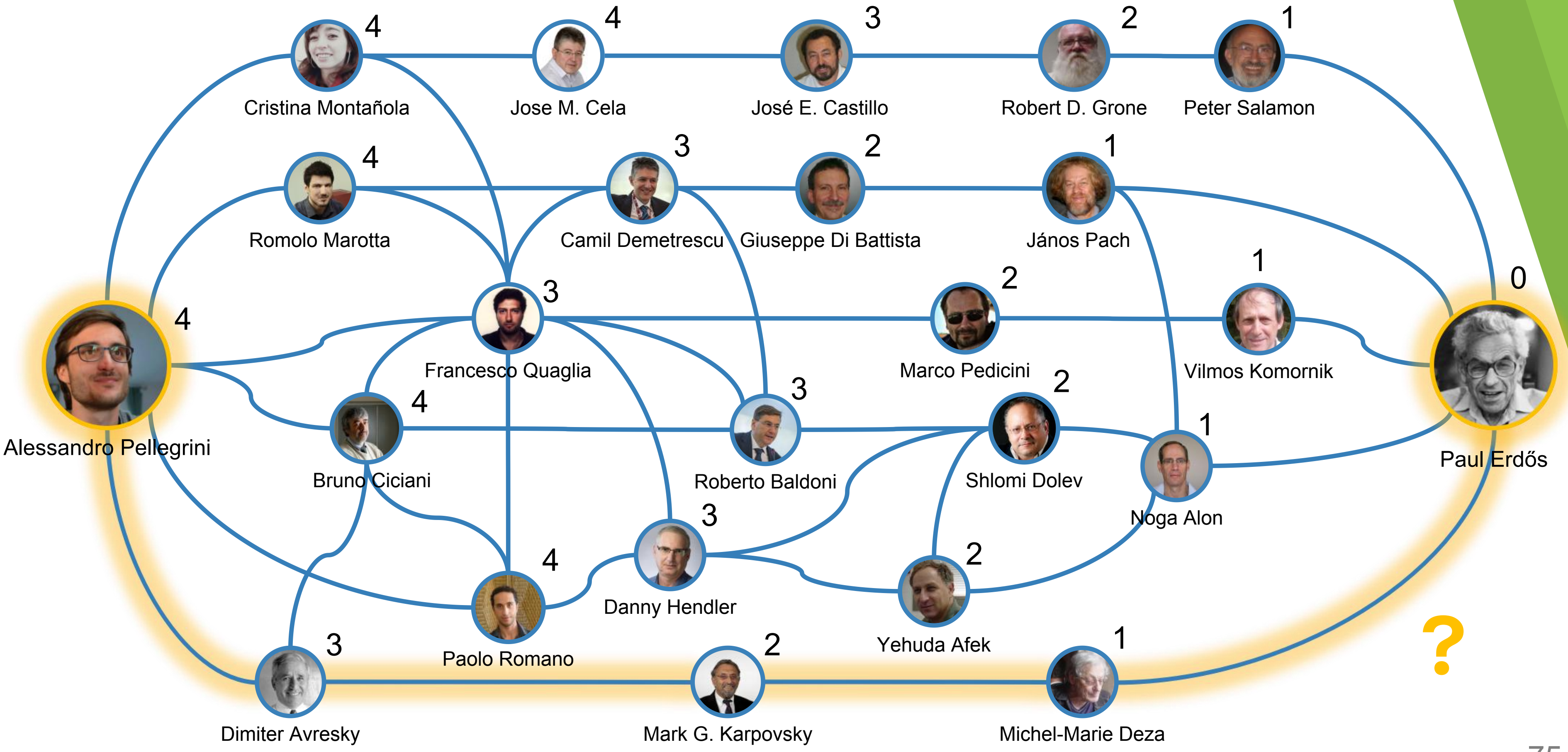


Queue = { }

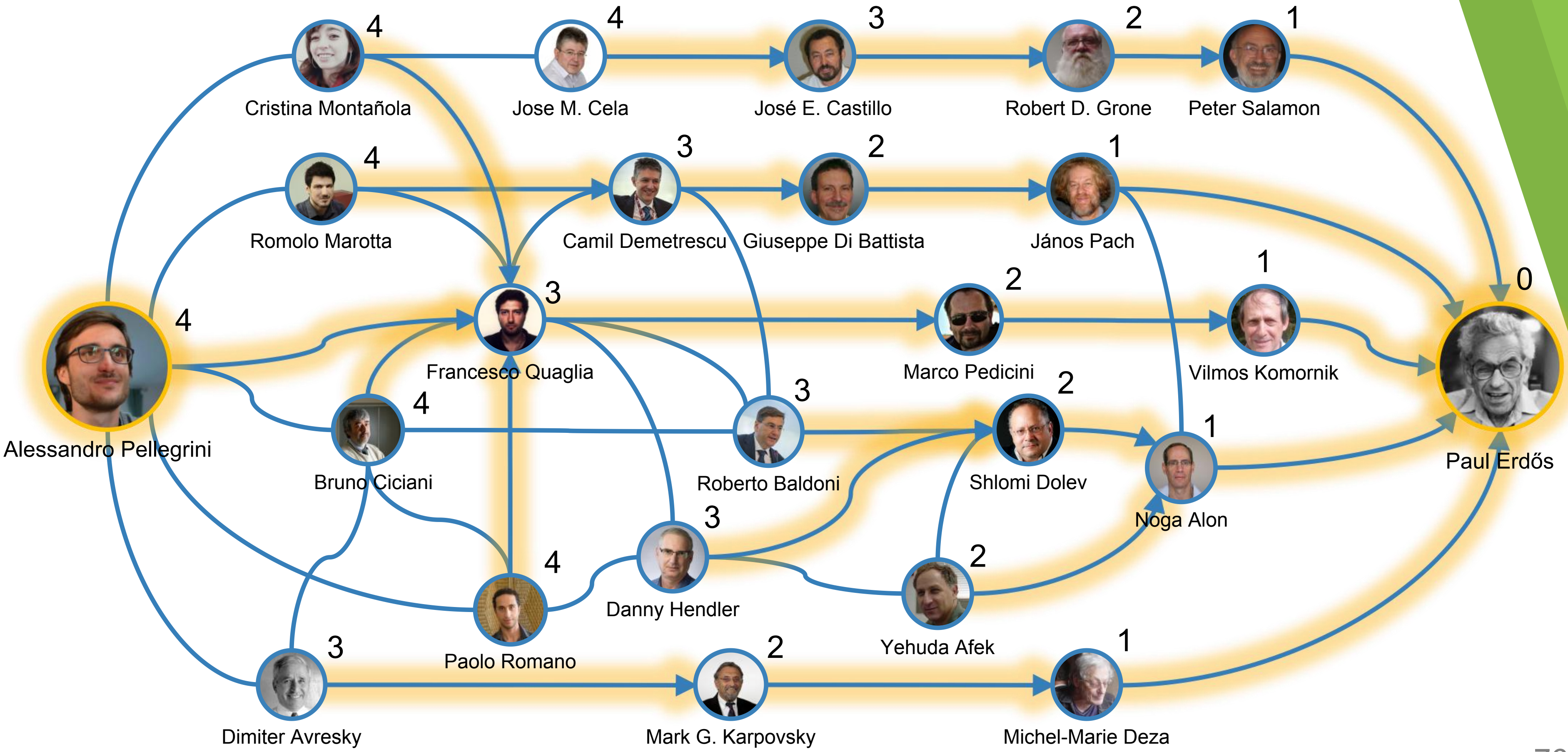
Erdős Number



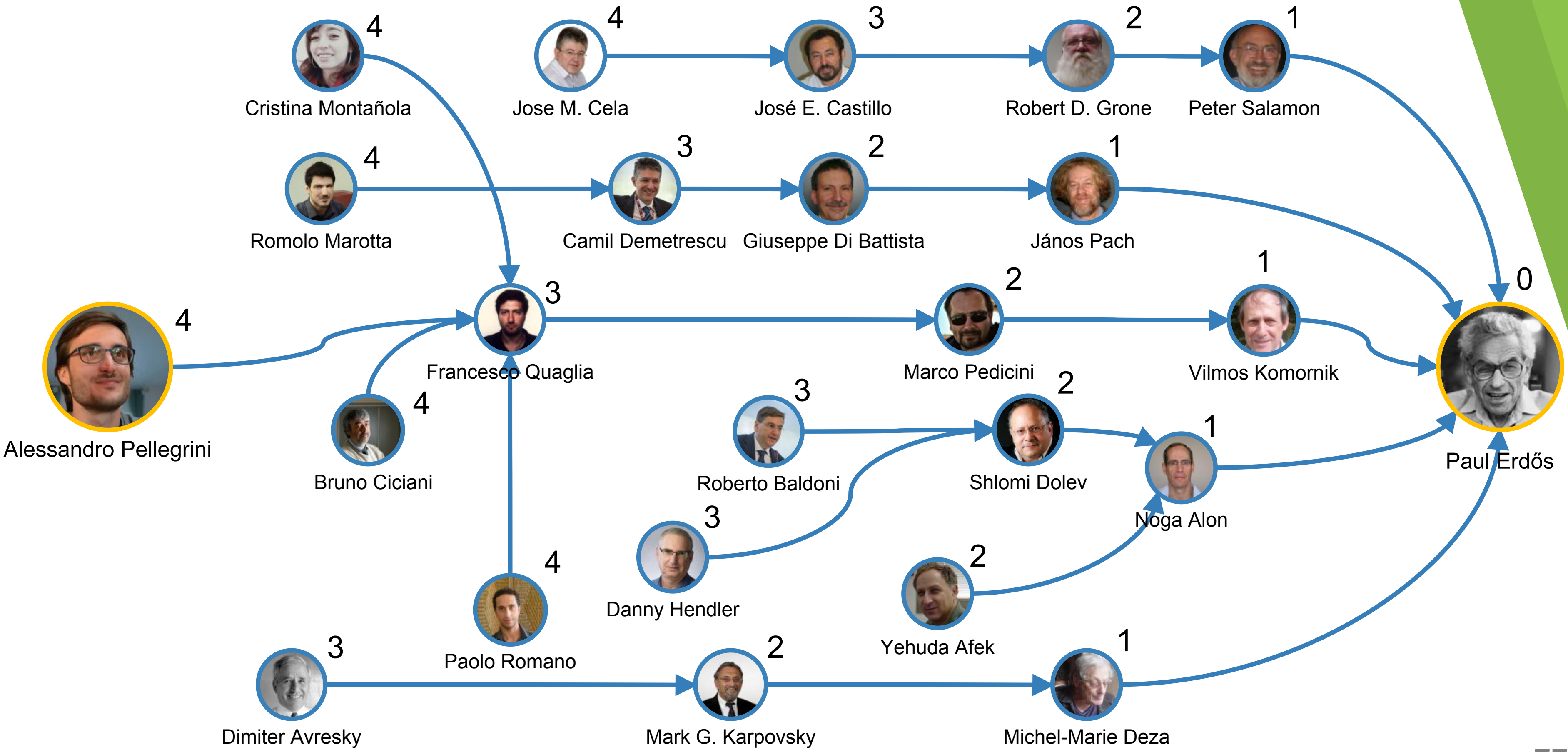
Erdős Number



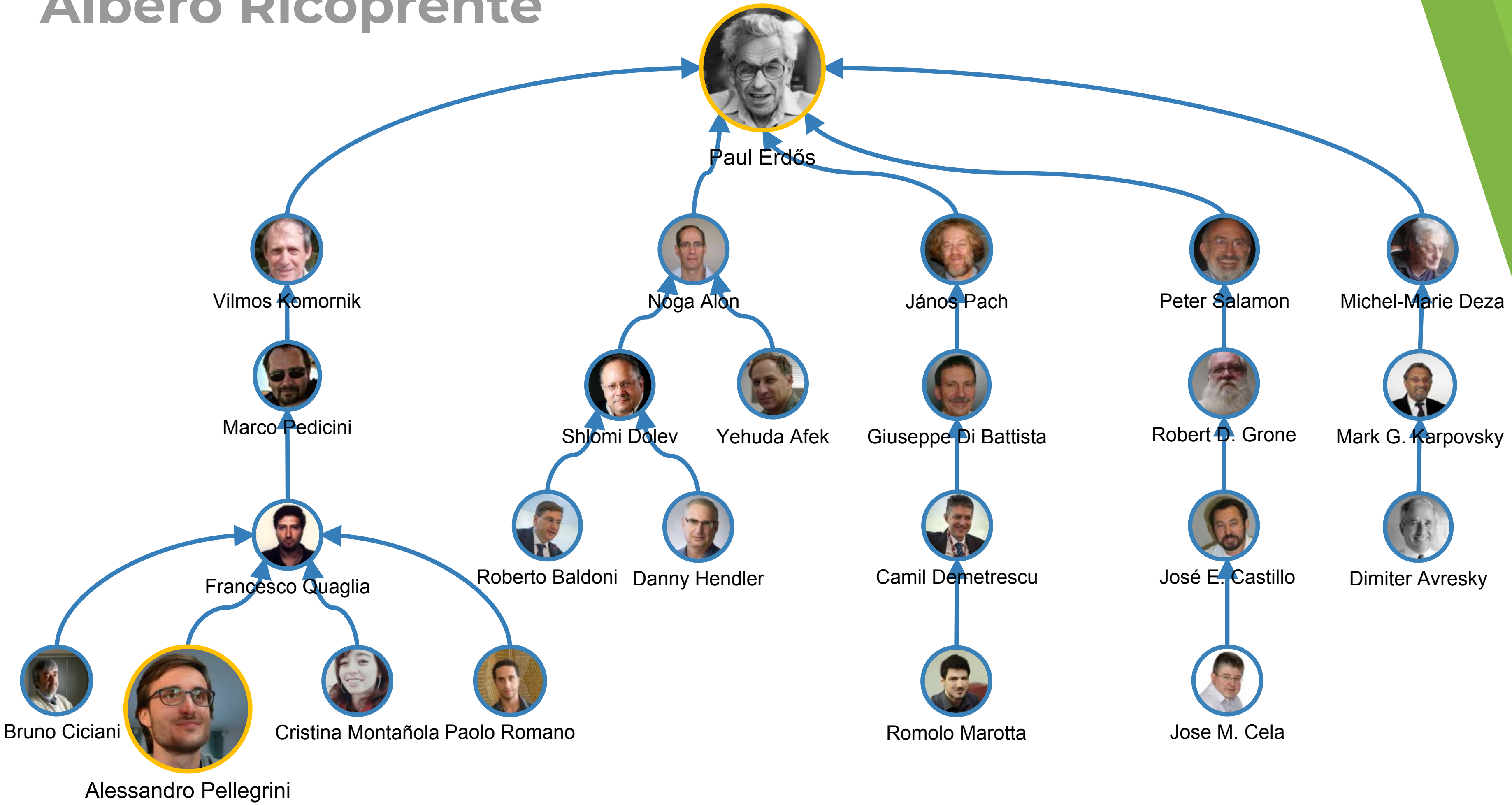
Erdős Number



Erdős Number

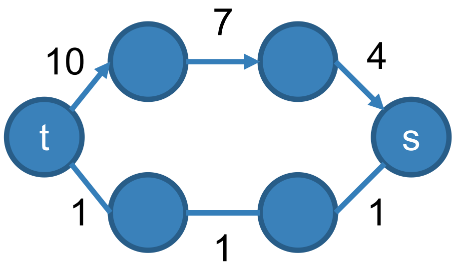


Albero Ricoprente



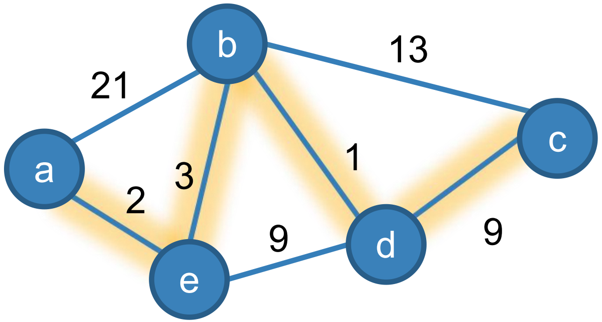
Cosa cambia nel caso di pesi differenti

- L'algoritmo per la ricerca dei cammini minimi considerato fin'ora trova il cammino minimo soltanto se i pesi di tutti gli archi sono identici
- Abbiamo osservato che il percorso con cui si raggiunge un nodo dipende dall'ordine di inserimento nella coda dei predecessori
- Nel caso di pesi, il primo percorso con cui raggiungo un nodo potrebbe non essere quello a peso minimo



SSSP—Algoritmo di Dijkstra

- Si tratta di un algoritmo greedy
- Si basa sull'idea che ogni sottocammino di un cammino minimo è esso stesso un cammino minimo
- Questa proprietà viene usata “al contrario”:
  - Si sovrastima la distanza di ciscun vertice dalla sorgente
  - Si visita ogni nodo ed i suoi nodi adiacenti per trovare il sottopercorso più breve verso i vicini



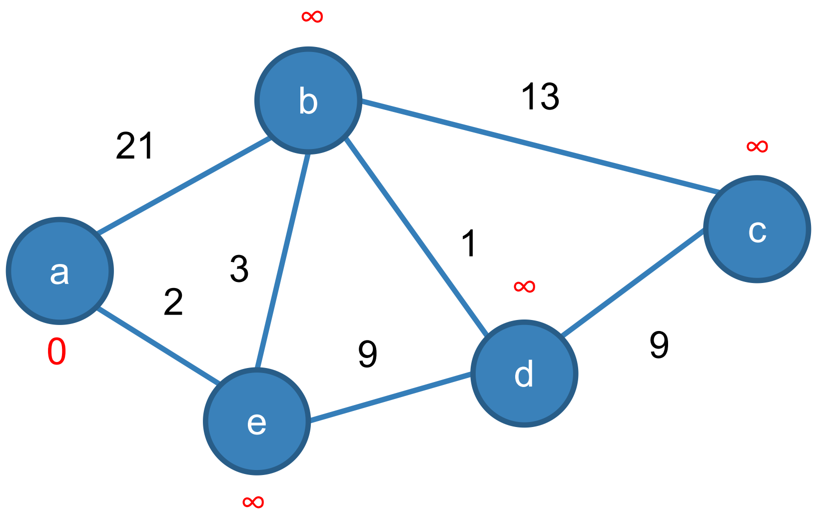
SSSP—Algoritmo di Dijkstra

- I nodi vengono divisi tra “visitati” e “non visitati”
- Viene preso il nodo non visitato con distanza minima
- Viene calcolata la distanza attraverso questo nodo verso ogni vicino non ancora visitato
- Se si trova una distanza più piccola, la distanza viene aggiornata
- Dopo aver visitato tutti i vicini, il nodo di partenza viene marcato come visitato

SSSP—Algoritmo di Dijkstra

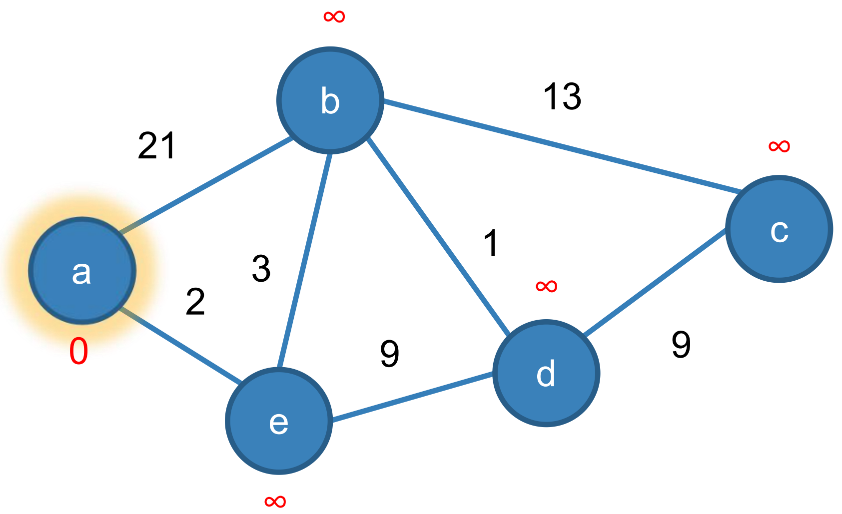
```
SSSP(G, r):
    r.distance ← 0
    MinHeap PQ ← Ø
    foreach v in G:
        if v ≠ r then:
            v.distance ← ∞
        v.parent ← nil
        PQ.enqueue(v)
    while PQ is not empty:
        u ← PQ.getMin()
        foreach v in G.adj(u):
            if v is in PQ then:
                newDist ← u.distance + w(u, v)
                if newDist < v.distance then:
                    v.distance ← newDist
                    v.parent ← u
                    PQ.decreasePrio(v, newDist)
```

SSSP—Algoritmo di Dijkstra



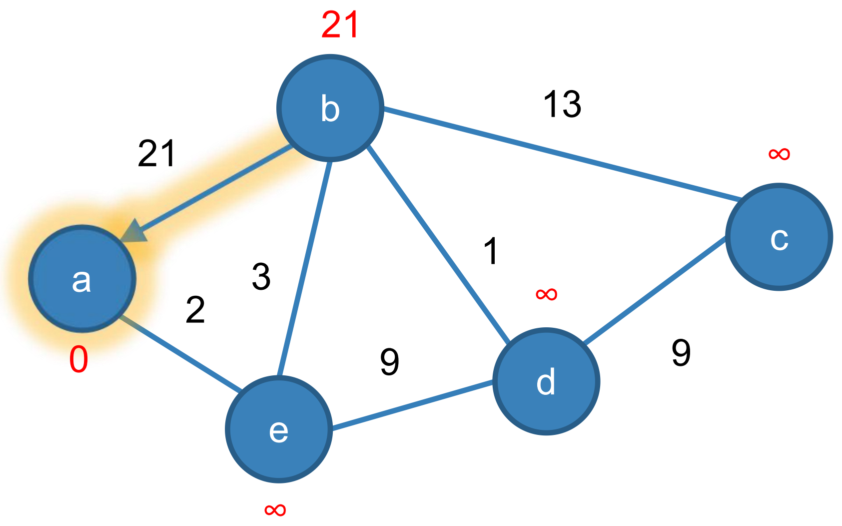
PriorityQueue = { a, b, c, d, e }

SSSP—Algoritmo di Dijkstra



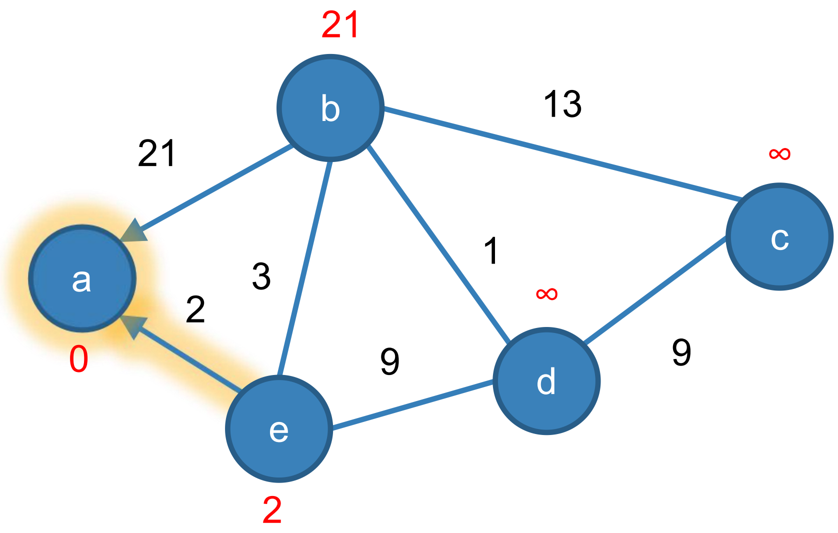
PriorityQueue = { b, c, d, e }

SSSP—Algoritmo di Dijkstra



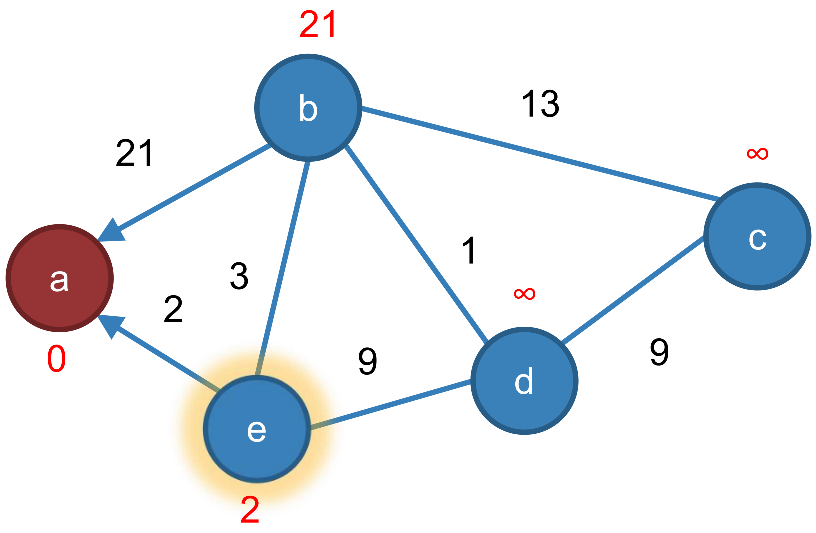
PriorityQueue = { b, c, d, e }

SSSP—Algoritmo di Dijkstra



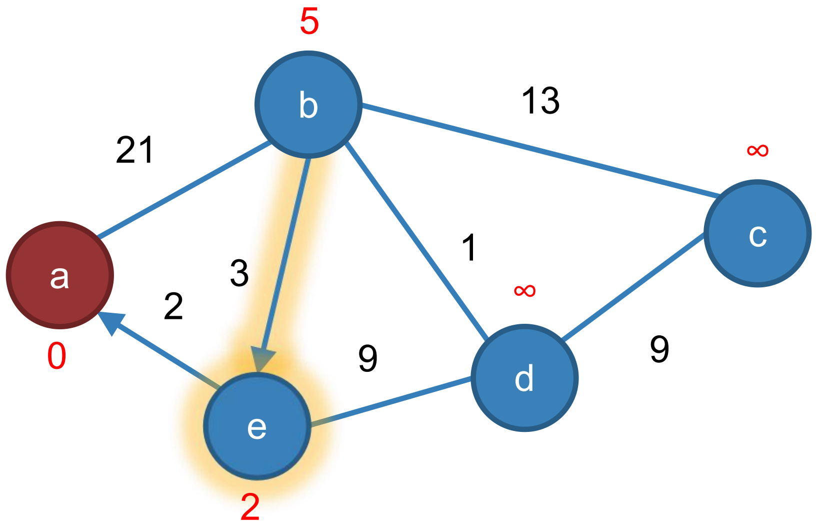
PriorityQueue = { e, b, c, d }

SSSP—Algoritmo di Dijkstra



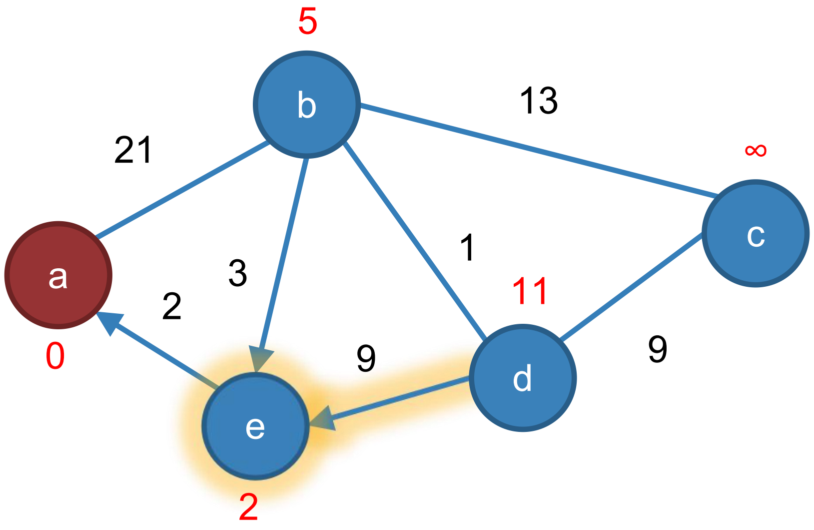
PriorityQueue = { b, c, d }

SSSP—Algoritmo di Dijkstra



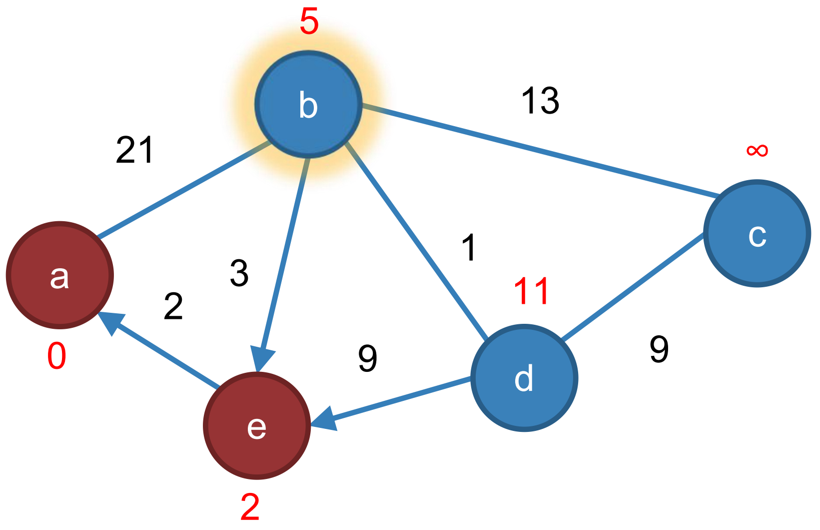
PriorityQueue = { b, c, d }

SSSP—Algoritmo di Dijkstra



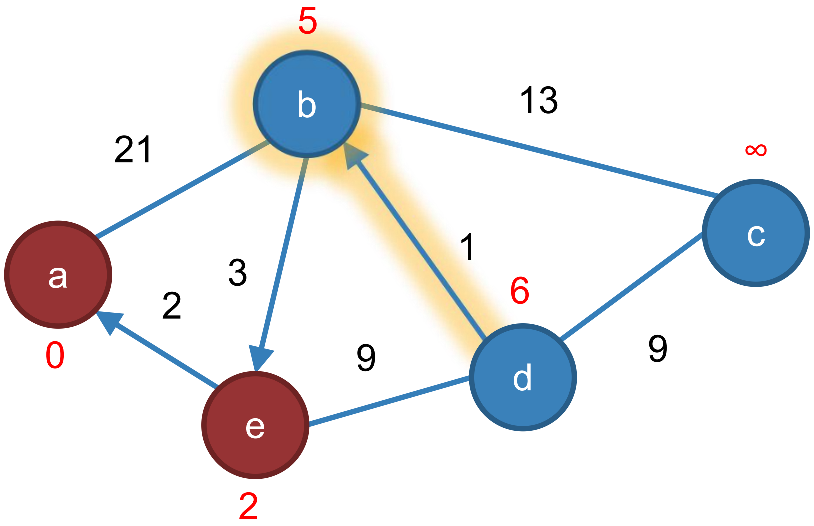
PriorityQueue = { b, d, c }

SSSP—Algoritmo di Dijkstra



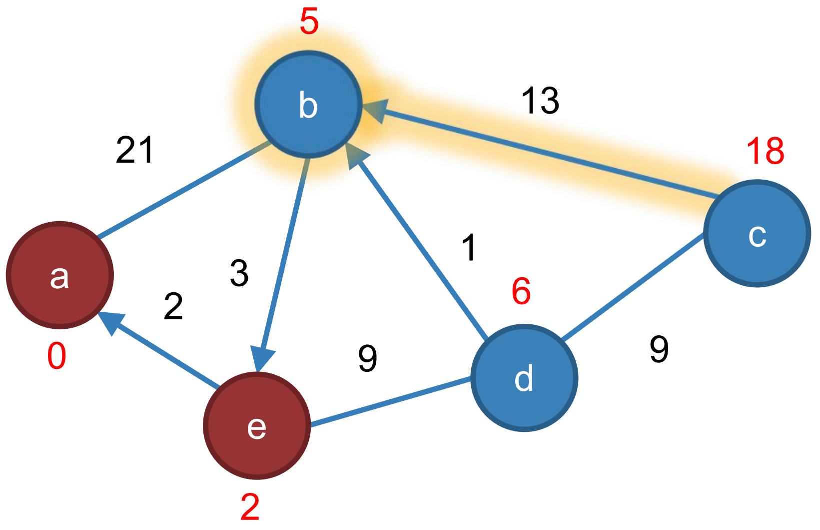
PriorityQueue = { d, c }

SSSP—Algoritmo di Dijkstra



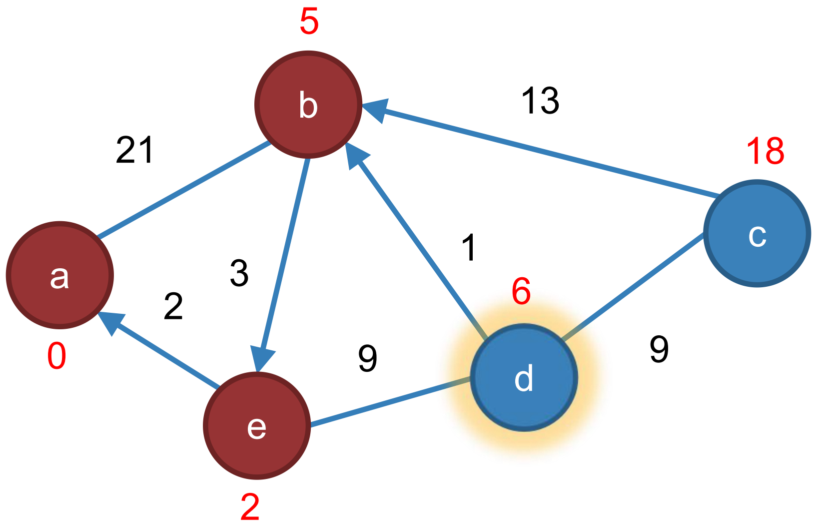
PriorityQueue = { d, c }

SSSP—Algoritmo di Dijkstra



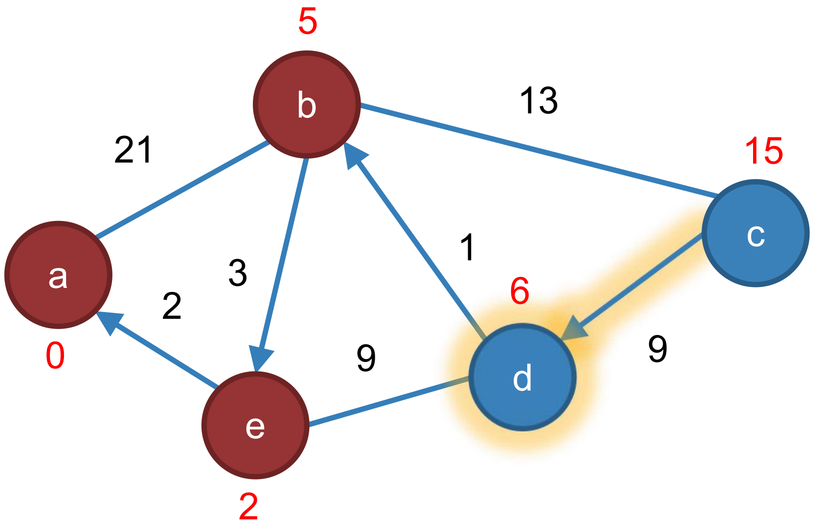
PriorityQueue = { d, c }

SSSP—Algoritmo di Dijkstra



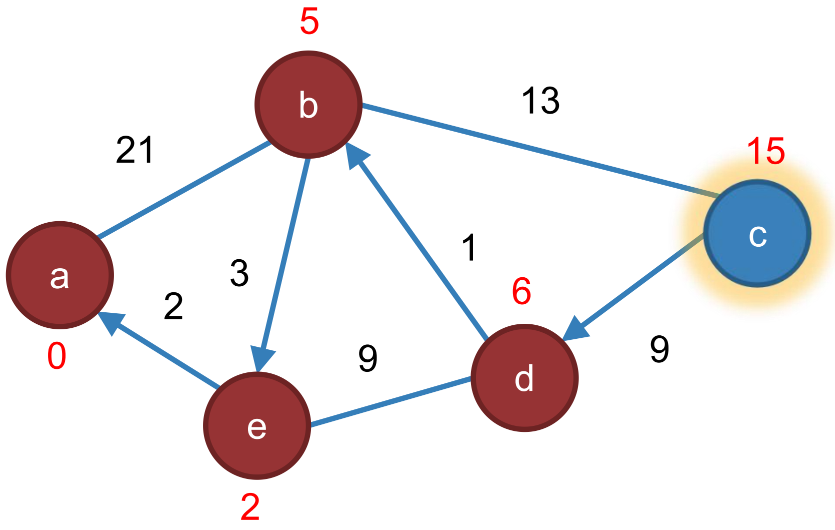
PriorityQueue = { c }

SSSP—Algoritmo di Dijkstra



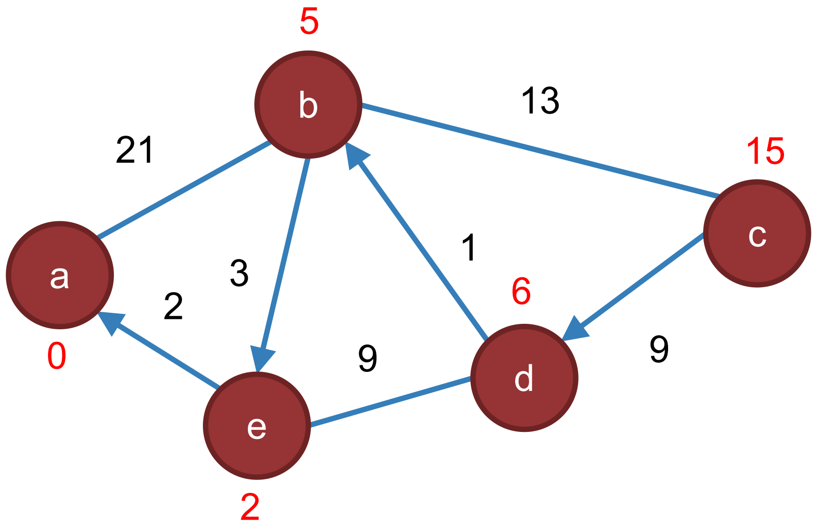
PriorityQueue = { c }

SSSP—Algoritmo di Dijkstra



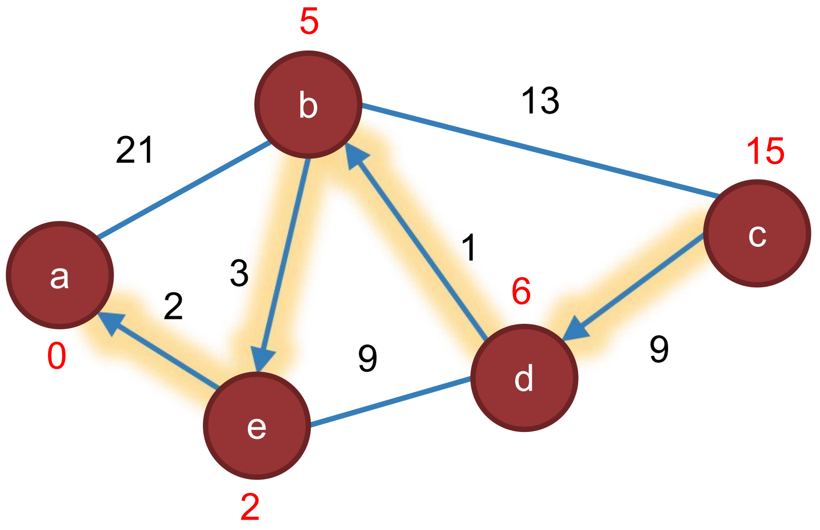
PriorityQueue = { }

SSSP—Algoritmo di Dijkstra



PriorityQueue = { }

SSSP—Algoritmo di Dijkstra

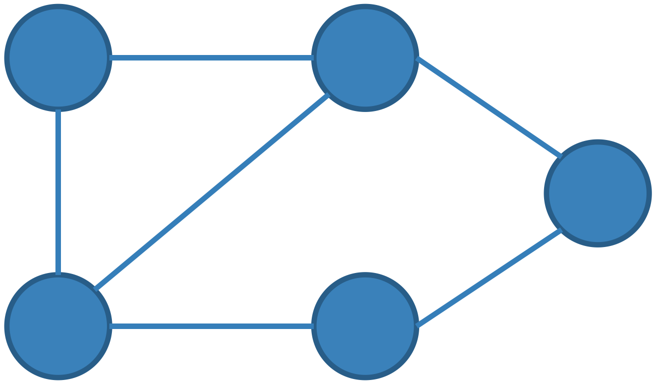


PriorityQueue = { }

Grafi aciclici

Cicli in grafi non orientati

- In un grafo non orientato $G = (V, E)$ un ciclo C di lunghezza $k > 2$ è una sequenza di nodi $u_0, u_1, \cdots, u_k$ tale che $(u_i, u_{i+1}) \in E$ per $0 \leq i \leq k - 1$ e $u_0 = u_k$
  - La condizione $k > 2$ esclude cicli banali composti da coppie di archi (u, v) e (v, u) che sono sempre presenti nei grafi non orientati

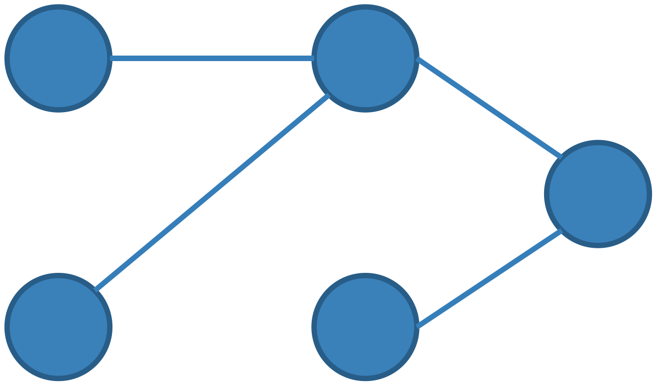


Grafi non orientati aciclici

- Un grafo è detto *aciclico* se non contiene alcun ciclo

```
HASCYCLE(G, u, p):
  u.visited ← true
  foreach v in G.adj(u) \ {p}:
    if v.visited then:
        return true
    else if hasCycle(G, v, u) then:
        return true
  return false
```

```
HASCYCLES(G):
  foreach u in G.V():
    u.visted ← false
  foreach u in G.V():
    if not u.visited then:
      if hasCycle(G, u, ⊥) then:
        return true
  return false
```

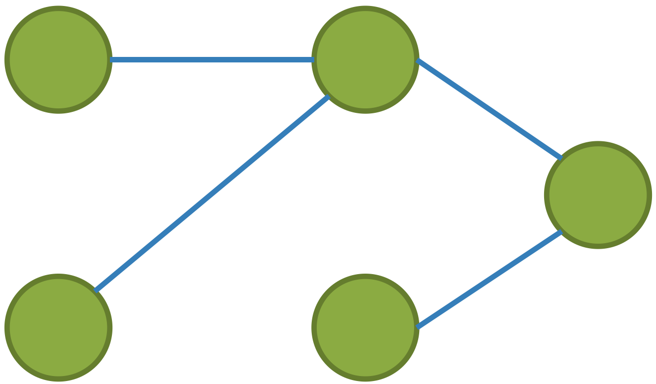


Grafi non orientati aciclici

- Un grafo è detto *aciclico* se non contiene alcun ciclo

```
HASCYCLE(G, u, p):
  u.visited ← true
  foreach v in G.adj(u) \ {p}:
    if v.visited then:
        return true
    else if hasCycle(G, v, u) then:
        return true
  return false
```

```
HASCYCLES(G):
  foreach u in G.V():
    u.visted ← false
  foreach u in G.V():
    if not u.visited then:
      if hasCycle(G, u, ⊥) then:
        return true
  return false
```

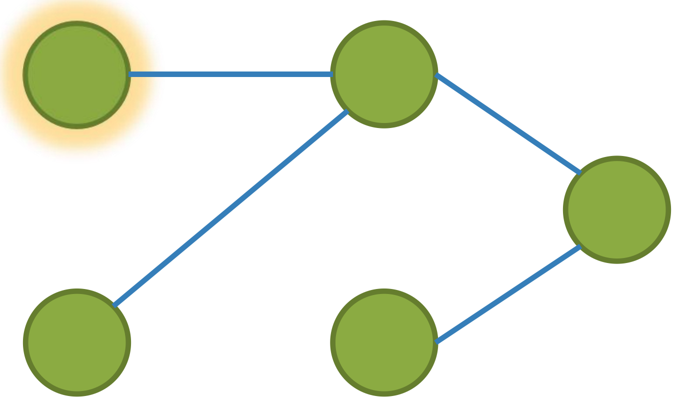


Grafi non orientati aciclici

- Un grafo è detto *aciclico* se non contiene alcun ciclo

```
HASCYCLE(G, u, p):
    u.visited ← true
    foreach v in G.adj(u) \ {p}:
        if v.visited then:
            return true
        else if hasCycle(G, v, u) then:
            return true
    return false
```

```
HASCYCLES(G):
    foreach u in G.V():
        u.visted ← false
    foreach u in G.V():
        if not u.visited then:
            if hasCycle(G, u, ⊥) then:
                return true
    return false
```

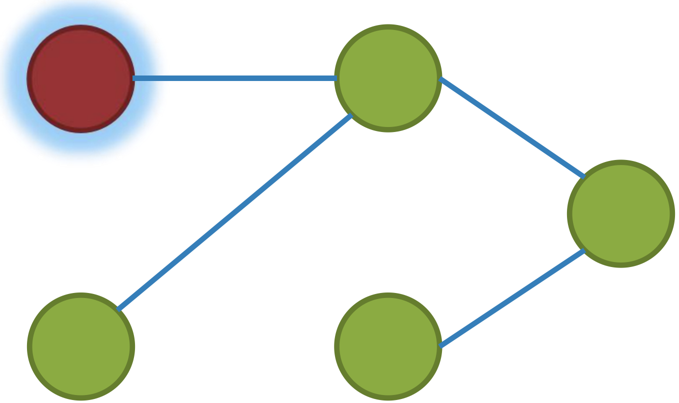


Grafi non orientati aciclici

- Un grafo è detto *aciclico* se non contiene alcun ciclo

```
HASCYCLE(G, u, p):
  u.visited ← true
  foreach v in G.adj(u) \ {p}:
    if v.visited then:
        return true
    else if hasCycle(G, v, u) then:
        return true
  return false
```

```
HASCYCLES(G):
  foreach u in G.V():
    u.visted ← false
  foreach u in G.V():
    if not u.visited then:
      if hasCycle(G, u, ⊥) then:
        return true
  return false
```

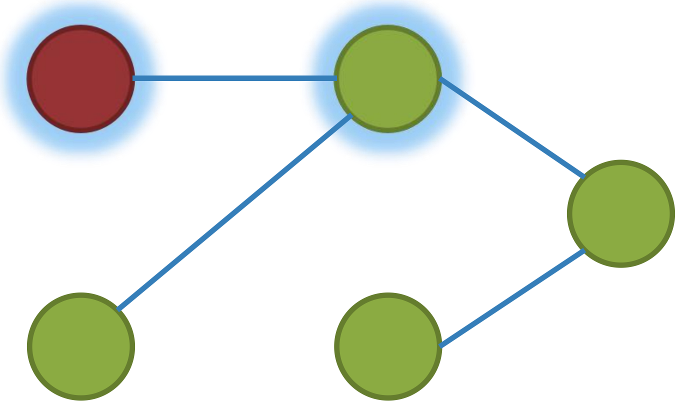


Grafi non orientati aciclici

- Un grafo è detto *aciclico* se non contiene alcun ciclo

```
HASCYCLE(G, u, p):
   u.visited ← true
   foreach v in G.adj(u) \ {p}:
      if v.visited then:
         return true
      else if hasCycle(G, v, u) then:
         return true
   return false
```

```
HASCYCLES(G):
   foreach u in G.V():
      u.visted ← false
   foreach u in G.V():
      if not u.visited then:
         if hasCycle(G, u, ⊥) then:
            return true
   return false
```

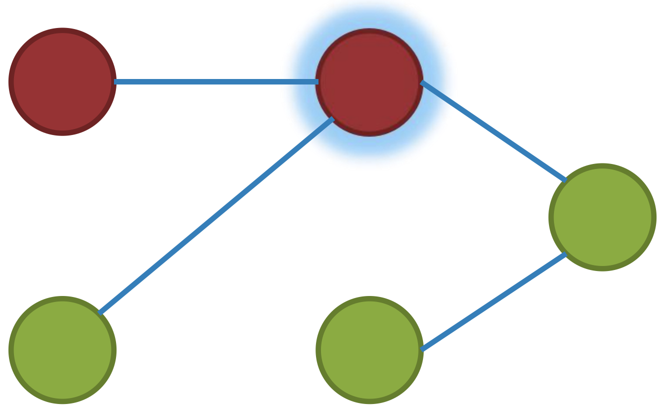


Grafi non orientati aciclici

- Un grafo è detto *aciclico* se non contiene alcun ciclo

```
HASCYCLE(G, u, p):
  u.visited ← true
  foreach v in G.adj(u) \ {p}:
    if v.visited then:
        return true
    else if hasCycle(G, v, u) then:
        return true
  return false
```

```
HASCYCLES(G):
  foreach u in G.V():
    u.visted ← false
  foreach u in G.V():
    if not u.visited then:
      if hasCycle(G, u, ⊥) then:
        return true
  return false
```

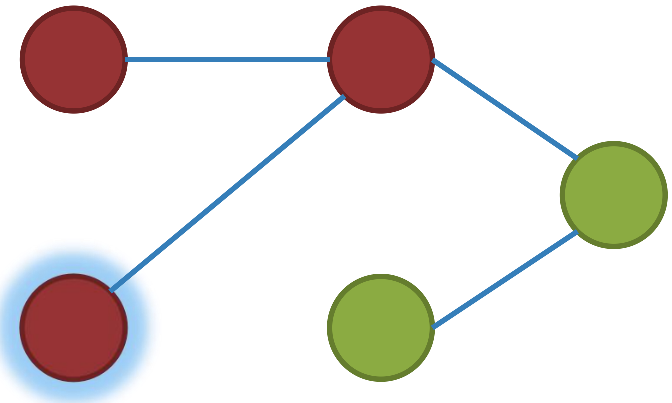


Grafi non orientati aciclici

- Un grafo è detto *aciclico* se non contiene alcun ciclo

```
HASCYCLE(G, u, p):
  u.visited ← true
  foreach v in G.adj(u) \ {p}:
    if v.visited then:
        return true
    else if hasCycle(G, v, u) then:
        return true
  return false
```

```
HASCYCLES(G):
  foreach u in G.V():
    u.visted ← false
  foreach u in G.V():
    if not u.visited then:
      if hasCycle(G, u, ⊥) then:
        return true
  return false
```

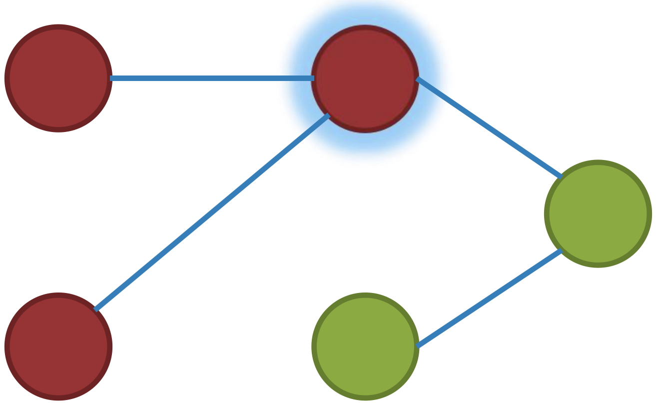


Grafi non orientati aciclici

- Un grafo è detto *aciclico* se non contiene alcun ciclo

```
HASCYCLE(G, u, p):
  u.visited ← true
  foreach v in G.adj(u) \ {p}:
    if v.visited then:
      return true
    else if hasCycle(G, v, u) then:
      return true
  return false
```

```
HASCYCLES(G):
  foreach u in G.V():
    u.visted ← false
  foreach u in G.V():
    if not u.visited then:
      if hasCycle(G, u, ⊥) then:
        return true
  return false
```

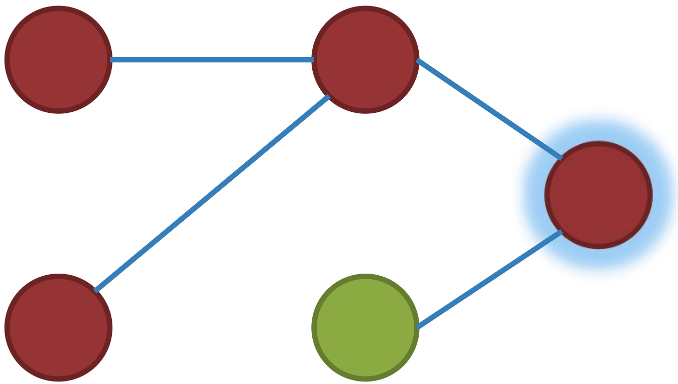


Grafi non orientati aciclici

- Un grafo è detto *aciclico* se non contiene alcun ciclo

```
HASCYCLE(G, u, p):
  u.visited ← true
  foreach v in G.adj(u) \ {p}:
    if v.visited then:
        return true
    else if hasCycle(G, v, u) then:
        return true
  return false
```

```
HASCYCLES(G):
  foreach u in G.V():
    u.visted ← false
  foreach u in G.V():
    if not u.visited then:
      if hasCycle(G, u, ⊥) then:
        return true
  return false
```

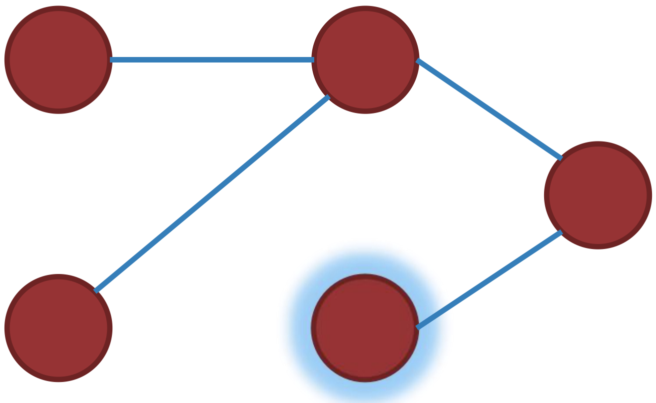


Grafi non orientati aciclici

- Un grafo è detto *aciclico* se non contiene alcun ciclo

```
HASCYCLE(G, u, p):
   u.visited ← true
   foreach v in G.adj(u) \ {p}:
      if v.visited then:
          return true
      else if hasCycle(G, v, u) then:
          return true
   return false
```

```
HASCYCLES(G):
   foreach u in G.V():
      u.visted ← false
   foreach u in G.V():
      if not u.visited then:
         if hasCycle(G, u, ⊥) then:
            return true
   return false
```

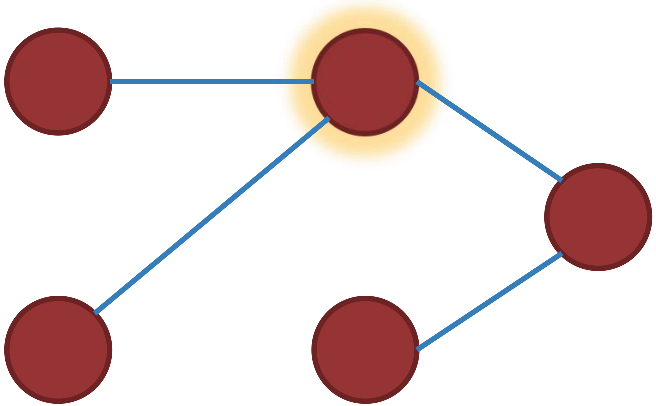


Grafi non orientati aciclici

- Un grafo è detto *aciclico* se non contiene alcun ciclo

```
HASCYCLE(G, u, p):
  u.visited ← true
  foreach v in G.adj(u) \ {p}:
    if v.visited then:
      return true
    else if hasCycle(G, v, u) then:
      return true
  return false
```

```
HASCYCLES(G):
  foreach u in G.V():
    u.visted ← false
  foreach u in G.V():
    if not u.visited then:
      if hasCycle(G, u, ⊥) then:
        return true
  return false
```

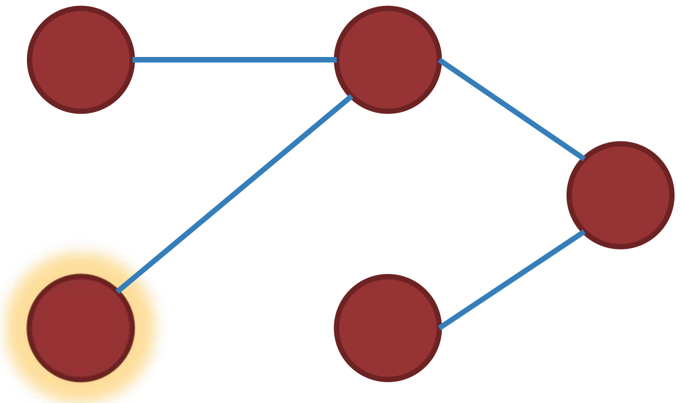


Grafi non orientati aciclici

- Un grafo è detto *aciclico* se non contiene alcun ciclo

```
HASCYCLE(G, u, p):
   u.visited ← true
   foreach v in G.adj(u) \ {p}:
      if v.visited then:
         return true
      else if hasCycle(G, v, u) then:
         return true
   return false
```

```
HASCYCLES(G):
   foreach u in G.V():
      u.visted ← false
   foreach u in G.V():
      if not u.visited then:
         if hasCycle(G, u, ⊥) then:
            return true
   return false
```

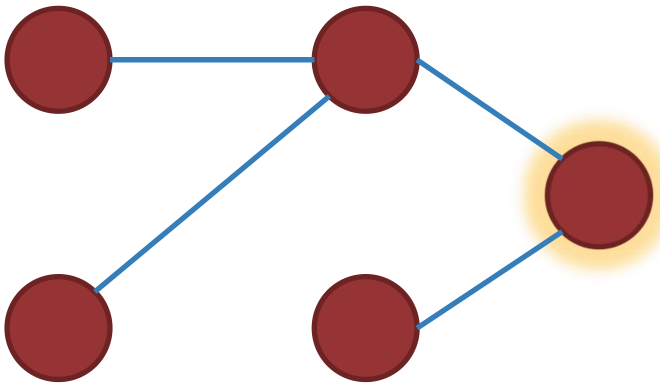


Grafi non orientati aciclici

- Un grafo è detto *aciclico* se non contiene alcun ciclo

```
HASCYCLE(G, u, p):
   u.visited ← true
   foreach v in G.adj(u) \ {p}:
      if v.visited then:
          return true
      else if hasCycle(G, v, u) then:
          return true
   return false
```

```
HASCYCLES(G):
   foreach u in G.V():
      u.visted ← false
   foreach u in G.V():
      if not u.visited then:
         if hasCycle(G, u, ⊥) then:
            return true
   return false
```

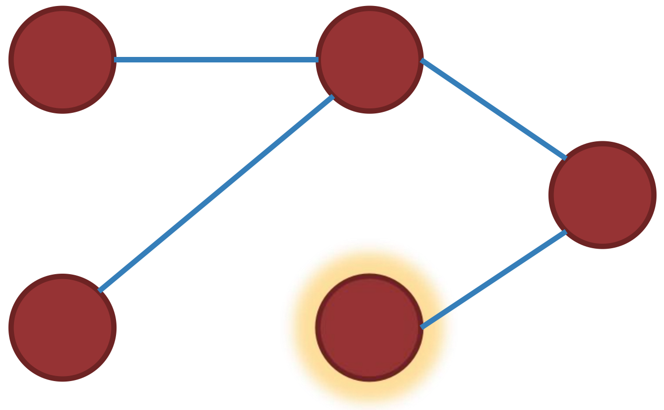


Grafi non orientati aciclici

- Un grafo è detto *aciclico* se non contiene alcun ciclo

```
HASCYCLE(G, u, p):
  u.visited ← true
  foreach v in G.adj(u) \ {p}:
    if v.visited then:
      return true
    else if hasCycle(G, v, u) then:
      return true
  return false
```

```
HASCYCLES(G):
  foreach u in G.V():
    u.visted ← false
  foreach u in G.V():
    if not u.visited then:
      if hasCycle(G, u, ⊥) then:
        return true
  return false
```

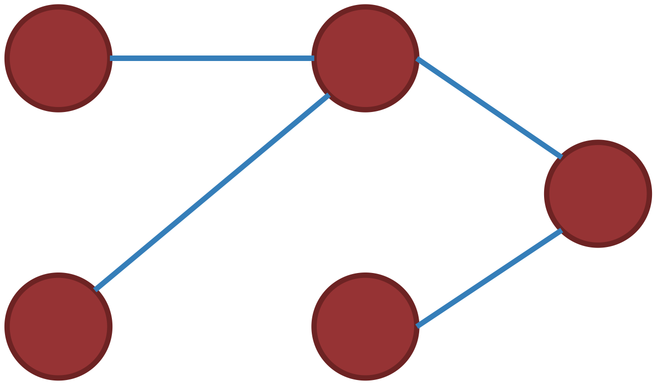


Grafi non orientati aciclici

- Un grafo è detto *aciclico* se non contiene alcun ciclo

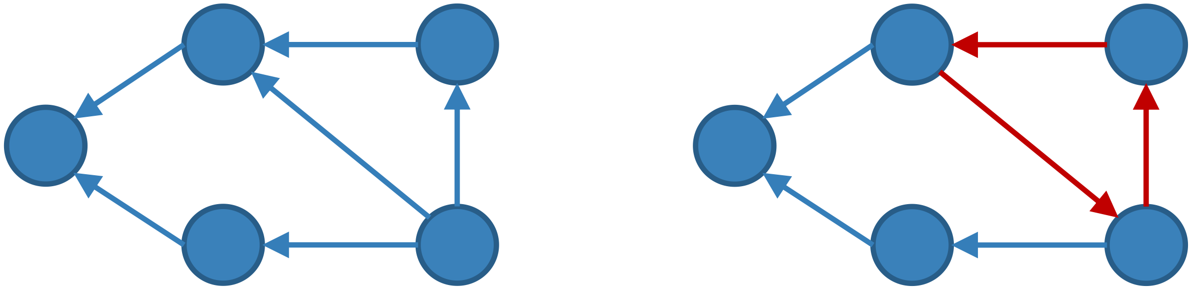
```
HASCYCLE(G, u, p):
   u.visited ← true
   foreach v in G.adj(u) \ {p}:
      if v.visited then:
          return true
      else if hasCycle(G, v, u) then:
          return true
   return false
```

```
HASCYCLES(G):
   foreach u in G.V():
      u.visted ← false
   foreach u in G.V():
      if not u.visited then:
         if hasCycle(G, u, ⊥) then:
            return true
   return false
```



Cicli in grafi orientati

- In un grafo orientato $G = (V, E)$ un ciclo C di lunghezza $k \geq 2$ è una sequenza di nodi $u_0, u_1, \cdots, u_k$ tale che $(u_i, u_{i+1}) \in E$ per $0 \leq i \leq k-1$ e $u_0 = u_k$
- Un ciclo è detto *semplice* se tutti i suoi nodi sono distinti (ad esclusione del primo e dell'ultimo)
- Un grafo orientato che non contiene cicli è detto DAG (Directed Acyclic Graph)
  - In tutti i DAG c'è sempre un nodo denominato *pozzo* (sink)

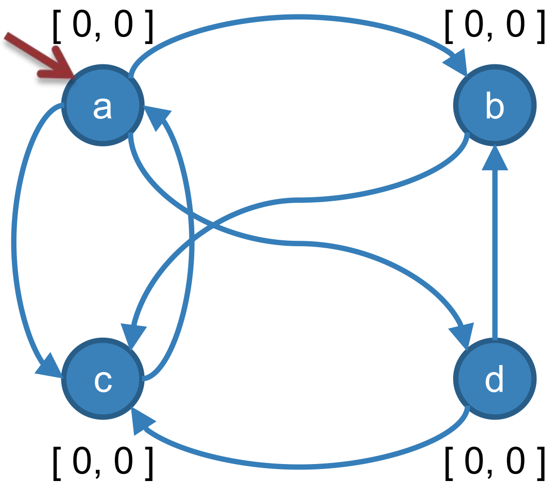


Identificazione di cicli in grafi orientati

- Intuitivamente, un DAG è un grafo che non ha nessun “arco all'indietro”
- Si tiene traccia del “tempo di visita” di ciascun nodo mediante un *timestamp*
  - Ogni nodo ha un timestamp di ingresso e di uscita
- Si possono quindi classificare gli archi (u,v), ogni volta che si raggiunge un nodo:
  - arco dell'albero: u.enterT = 0
  - arco all'indietro: u.enterT > v.enterT and v.exitT = 0
  - arco in avanti: u.enterT < v.enterT and v.exitT ≠ 0
  - arco di *attraversamento*: in tutti gli altri casi

Identificazione di cicli in grafi orientati

```
timestamp = 0
```

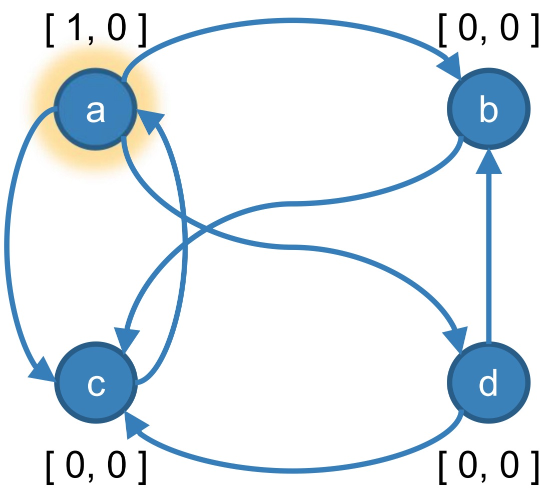


global timestamp

CLASSIFYEDGES(G, u)
  foreach v **in** G.adj(u):
    if v.enterTime = 0 **then**:
      < arco dell'albero >
      timestamp ← timestamp + 1
      u.enterT ← timestamp
      CLASSIFYEDGES(G, v) **then**:
    else if u.enterT > v.enterT **and** v.exitT = 0 **then**:
      < arco all'indietro >
    else if u.enterT < v.enterT and v.exitT ≠ 0 then:
      < arco in avanti >
    else:
      < arco di attraversamento >
  timestamp ← timestamp + 1
  u.exitT ← timestamp

Identificazione di cicli in grafi orientati

```
timestamp = 1
```

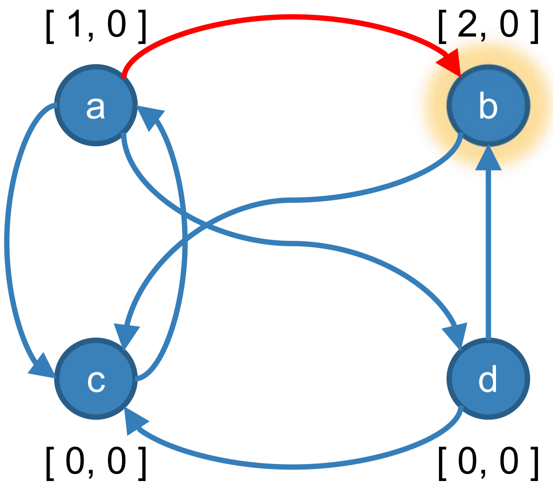


global timestamp

CLASSIFYEDGES(G, u)
  foreach v **in** G.adj(u):
    if v.enterTime = 0 **then**:
      < arco dell'albero >
      timestamp ← timestamp + 1
      u.enterT ← timestamp
      CLASSIFYEDGES(G, v) **then**:
    else if u.enterT > v.enterT **and** v.exitT = 0 **then**:
      < arco all'indietro >
    else if u.enterT < v.enterT and v.exitT ≠ 0 then:
      < arco in avanti >
    else:
      < arco di attraversamento >
  timestamp ← timestamp + 1
  u.exitT ← timestamp

Identificazione di cicli in grafi orientati

`timestamp = 2`

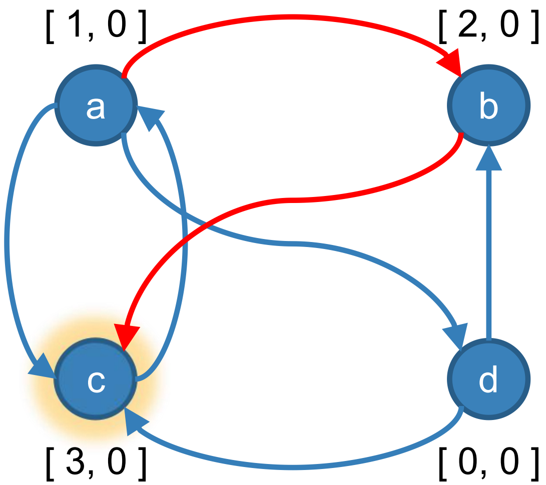


global timestamp

CLASSIFYEDGES(G, u)
    foreach v **in** G.adj(u):
        if v.enterTime = 0 **then**:
            < arco dell'albero >
            timestamp ← timestamp + 1
            u.enterT ← timestamp
            CLASSIFYEDGES(G, v) **then**:
        else if u.enterT > v.enterT **and** v.exitT = 0 **then**:
            < arco all'indietro >
        else if u.enterT < v.enterT and v.exitT ≠ 0 then:
            < arco in avanti >
        else:
            < arco di attraversamento >
    timestamp ← timestamp + 1
    u.exitT ← timestamp

Identificazione di cicli in grafi orientati

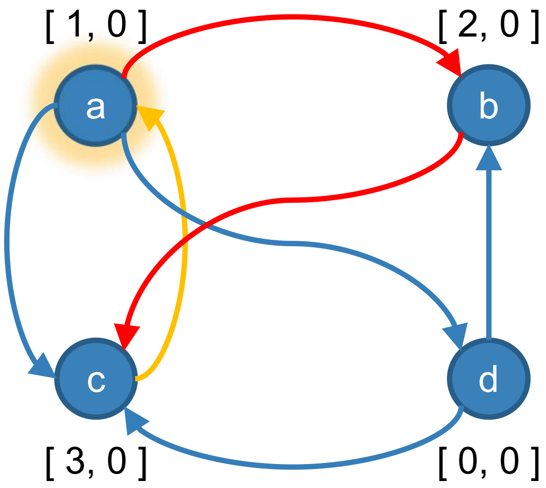
`timestamp = 3`




```
global timestamp

CLASSIFYEDGES(G, u)
    foreach v in G.adj(u):
        if v.enterTime = 0 then:
            < arco dell'albero >
            timestamp ← timestamp + 1
            u.enterT ← timestamp
            CLASSIFYEDGES(G, v) then:
        else if u.enterT > v.enterT and v.exitT = 0 then:
            < arco all'indietro >
        else if u.enterT < v.enterT and v.exitT ≠ 0 then:
            < arco in avanti >
        else:
            < arco di attraversamento >
    timestamp ← timestamp + 1
    u.exitT ← timestamp
```

Identificazione di cicli in grafi orientati

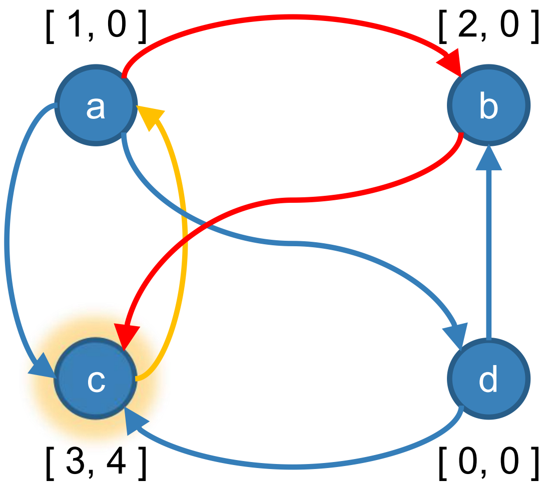
`timestamp = 3`




```
global timestamp

CLASSIFYEDGES(G, u)
   foreach v in G.adj(u):
      if v.enterTime = 0 then:
         < arco dell'albero >
         timestamp ← timestamp + 1
         u.enterT ← timestamp
         CLASSIFYEDGES(G, v) then:
      else if u.enterT > v.enterT and v.exitT = 0 then:
         < arco all'indietro >
      else if u.enterT < v.enterT and v.exitT ≠ 0 then:
         < arco in avanti >
      else:
         < arco di attraversamento >
   timestamp ← timestamp + 1
   u.exitT ← timestamp
```

Identificazione di cicli in grafi orientati

`timestamp = 4`

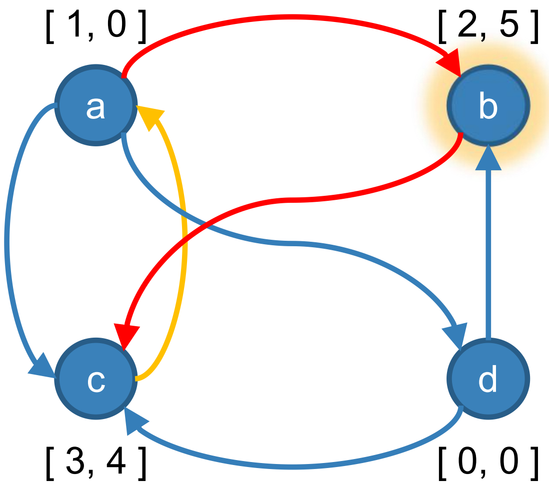


global timestamp

CLASSIFYEDGES(G, u)
  foreach v **in** G.adj(u):
    if v.enterTime = 0 **then**:
      < arco dell'albero >
      timestamp ← timestamp + 1
      u.enterT ← timestamp
      CLASSIFYEDGES(G, v) **then**:
    else if u.enterT > v.enterT **and** v.exitT = 0 **then**:
      < arco all'indietro >
    else if u.enterT < v.enterT and v.exitT ≠ 0 then:
      < arco in avanti >
    else:
      < arco di attraversamento >
  timestamp ← timestamp + 1
  u.exitT ← timestamp

Identificazione di cicli in grafi orientati

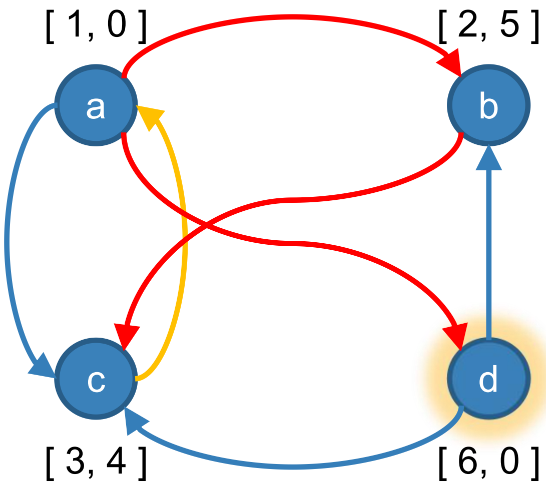
```
timestamp = 5
```




```
global timestamp

CLASSIFYEDGES(G, u)
   foreach v in G.adj(u):
      if v.enterTime = 0 then:
         < arco dell'albero >
         timestamp ← timestamp + 1
         u.enterT ← timestamp
         CLASSIFYEDGES(G, v) then:
      else if u.enterT > v.enterT and v.exitT = 0 then:
         < arco all'indietro >
      else if u.enterT < v.enterT and v.exitT ≠ 0 then:
         < arco in avanti >
      else:
         < arco di attraversamento >
   timestamp ← timestamp + 1
   u.exitT ← timestamp
```

Identificazione di cicli in grafi orientati

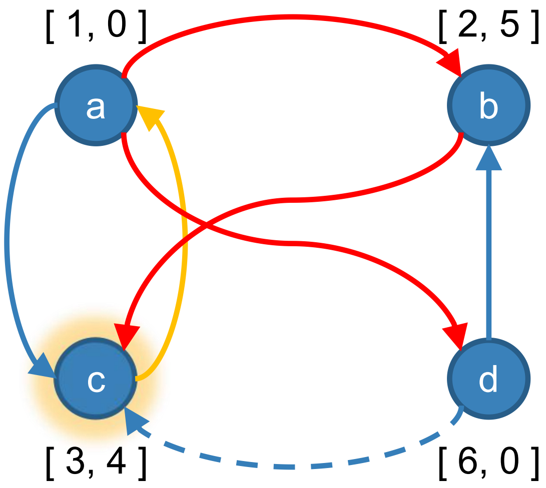
`timestamp = 6`




```
global timestamp

CLASSIFYEDGES(G, u)
   foreach v in G.adj(u):
      if v.enterTime = 0 then:
         < arco dell'albero >
         timestamp ← timestamp + 1
         u.enterT ← timestamp
         CLASSIFYEDGES(G, v) then:
      else if u.enterT > v.enterT and v.exitT = 0 then:
         < arco all'indietro >
      else if u.enterT < v.enterT and v.exitT ≠ 0 then:
         < arco in avanti >
      else:
         < arco di attraversamento >
   timestamp ← timestamp + 1
   u.exitT ← timestamp
```

Identificazione di cicli in grafi orientati

```
timestamp = 6
```

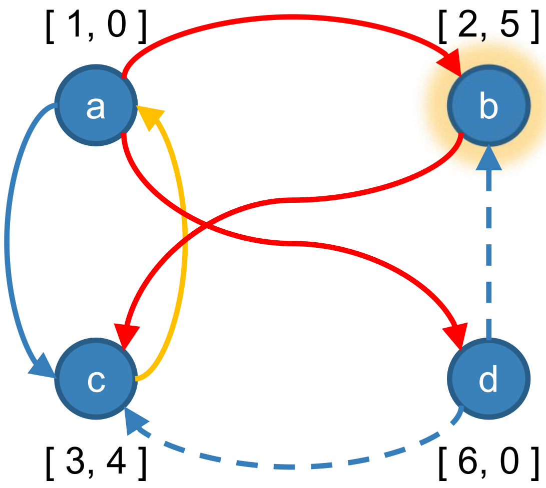


global timestamp

CLASSIFYEDGES(G, u)
  foreach v **in** G.adj(u):
    if v.enterTime = 0 **then**:
      < arco dell'albero >
      timestamp ← timestamp + 1
      u.enterT ← timestamp
      CLASSIFYEDGES(G, v) **then**:
    else if u.enterT > v.enterT **and** v.exitT = 0 **then**:
      < arco all'indietro >
    else if u.enterT < v.enterT and v.exitT ≠ 0 then:
      < arco in avanti >
    else:
      < arco di attraversamento >
  timestamp ← timestamp + 1
  u.exitT ← timestamp

Identificazione di cicli in grafi orientati

```
timestamp = 6
```

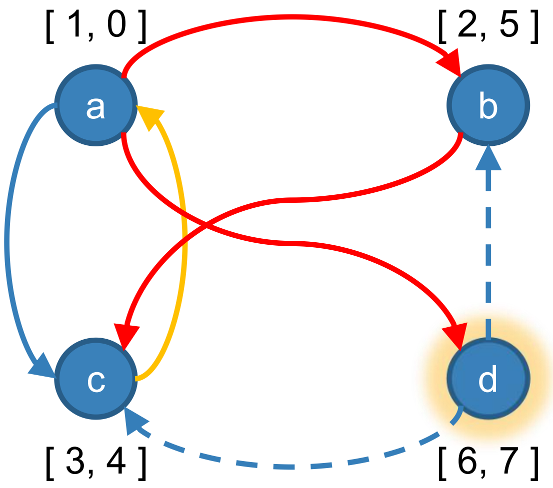


global timestamp

CLASSIFYEDGES(G, u)
  foreach v **in** G.adj(u):
    if v.enterTime = 0 **then**:
      < arco dell'albero >
      timestamp ← timestamp + 1
      u.enterT ← timestamp
      CLASSIFYEDGES(G, v) **then**:
    else if u.enterT > v.enterT **and** v.exitT = 0 **then**:
      < arco all'indietro >
    else if u.enterT < v.enterT and v.exitT ≠ 0 then:
      < arco in avanti >
    else:
      < arco di attraversamento >
  timestamp ← timestamp + 1
  u.exitT ← timestamp

Identificazione di cicli in grafi orientati

`timestamp = 7`

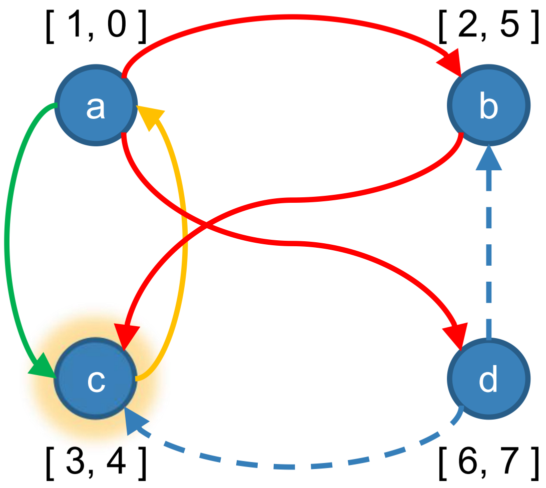


global timestamp

CLASSIFYEDGES(G, u)
  foreach v **in** G.adj(u):
    if v.enterTime = 0 **then**:
      < arco dell'albero >
      timestamp ← timestamp + 1
      u.enterT ← timestamp
      CLASSIFYEDGES(G, v) **then**:
    else if u.enterT > v.enterT **and** v.exitT = 0 **then**:
      < arco all'indietro >
    else if u.enterT < v.enterT and v.exitT ≠ 0 then:
      < arco in avanti >
    else:
      < arco di attraversamento >
  timestamp ← timestamp + 1
  u.exitT ← timestamp

Identificazione di cicli in grafi orientati

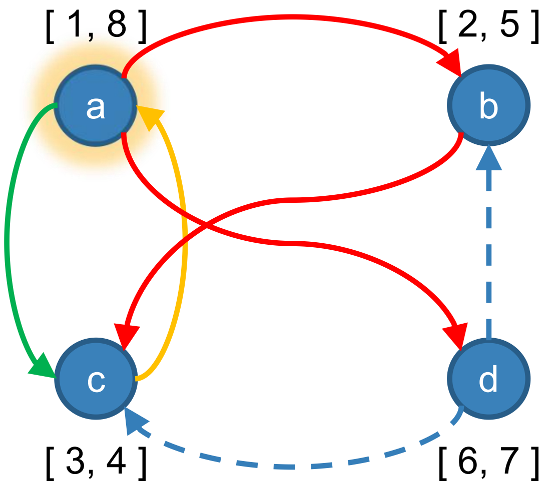
timestamp = 7




```
global timestamp

CLASSIFYEDGES(G, u)
  foreach v in G.adj(u):
    if v.enterTime = 0 then:
      < arco dell'albero >
      timestamp ← timestamp + 1
      u.enterT ← timestamp
      CLASSIFYEDGES(G, v) then:
    else if u.enterT > v.enterT and v.exitT = 0 then:
      < arco all'indietro >
    else if u.enterT < v.enterT and v.exitT ≠ 0 then:
      < arco in avanti >
    else:
      < arco di attraversamento >
  timestamp ← timestamp + 1
  u.exitT ← timestamp
```

Identificazione di cicli in grafi orientati

`timestamp = 8`

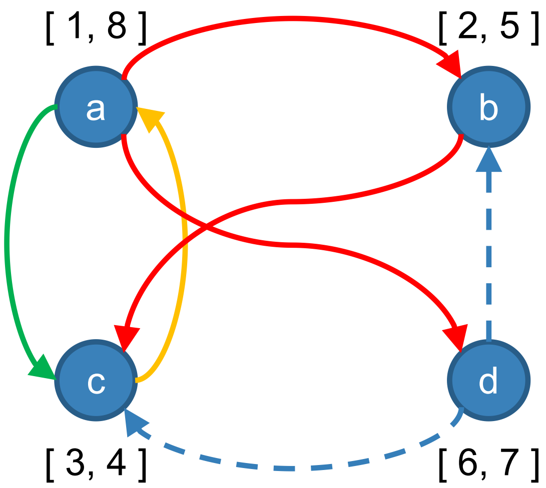


global timestamp

CLASSIFYEDGES(G, u)
  foreach v **in** G.adj(u):
    if v.enterTime = 0 **then**:
      < arco dell'albero >
      timestamp ← timestamp + 1
      u.enterT ← timestamp
      CLASSIFYEDGES(G, v) **then**:
    else if u.enterT > v.enterT **and** v.exitT = 0 **then**:
      < arco all'indietro >
    else if u.enterT < v.enterT and v.exitT ≠ 0 then:
      < arco in avanti >
    else:
      < arco di attraversamento >
  timestamp ← timestamp + 1
  u.exitT ← timestamp

Identificazione di cicli in grafi orientati

`timestamp = 8`




```
global timestamp

CLASSIFYEDGES(G, u)
  foreach v in G.adj(u):
    if v.enterTime = 0 then:
      < arco dell'albero >
      timestamp ← timestamp + 1
      u.enterT ← timestamp
      CLASSIFYEDGES(G, v) then:
    else if u.enterT > v.enterT and v.exitT = 0 then:
      < arco all'indietro >
    else if u.enterT < v.enterT and v.exitT ≠ 0 then:
      < arco in avanti >
    else:
      < arco di attraversamento >
  timestamp ← timestamp + 1
  u.exitT ← timestamp
```

Identificazione di cicli in grafi orientati

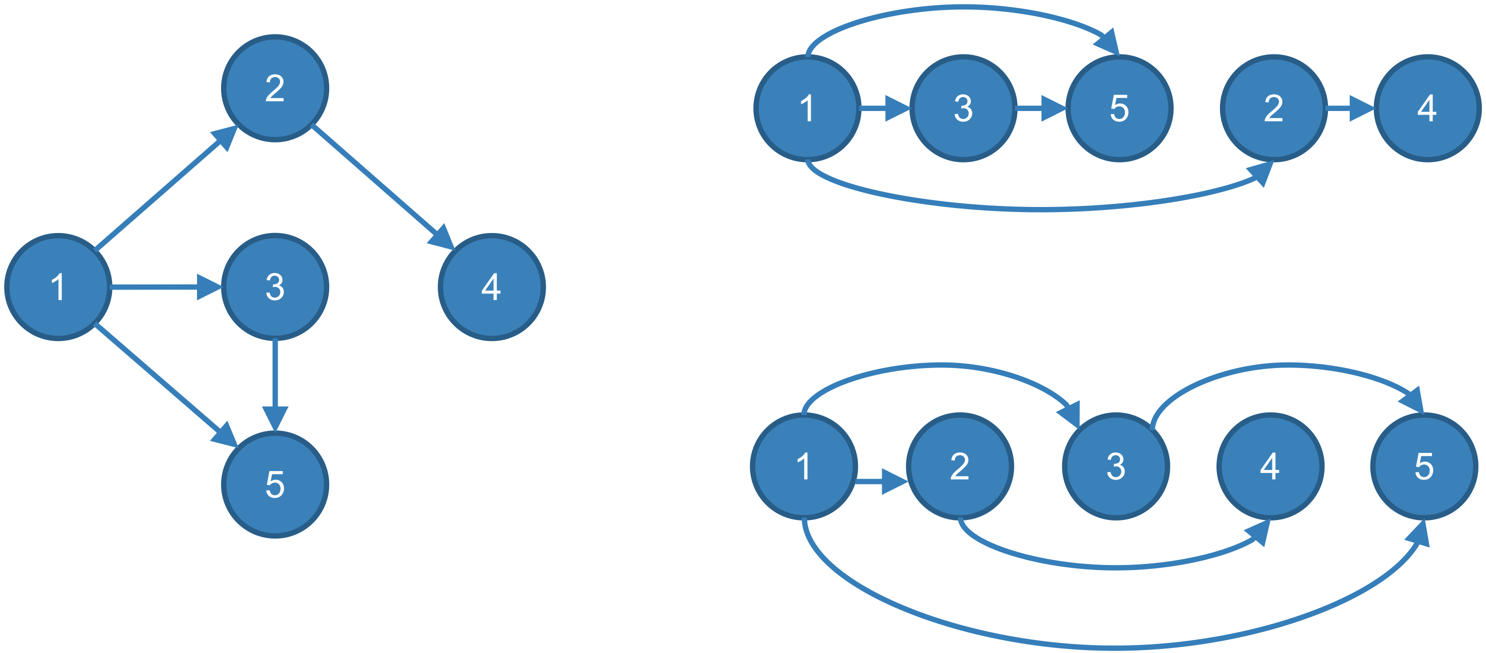
```
global timestamp

hasCycles(G, u)
   foreach v in G.adj(u):
      if v.enterTime = 0 then:
         timestamp ← timestamp + 1
         u.enterT ← timestamp
         if hasCycles(G, v) then:
            return true
      else if u.enterT > v.enterT and v.exitT = 0 then:
         return true
   timestamp ← timestamp + 1
   u.exitT ← timestamp
   return false
```

Ordinamento topologico

Ordinamento topologico

- Dato un DAG, un *ordinamento topologico* di G è un *ordinamento lineare* dei suoi nodi tale che se $(u, v) \in E$, allora u appare prima di v nell'ordinamento.
- L'ordinamento topologico è possibile solo per i DAG
- Dato un DAG, ci possono essere più ordinamenti topologici
- Molte applicazioni:
  - Ordine di compilazione in un Makefile
  - Ordine di valutazione delle celle in un foglio di calcolo
  - Risoluzione delle dipendenze nei linker
  - Risoluzione delle dipendenze nei gestori dei pacchetti

Ordinamento topologico



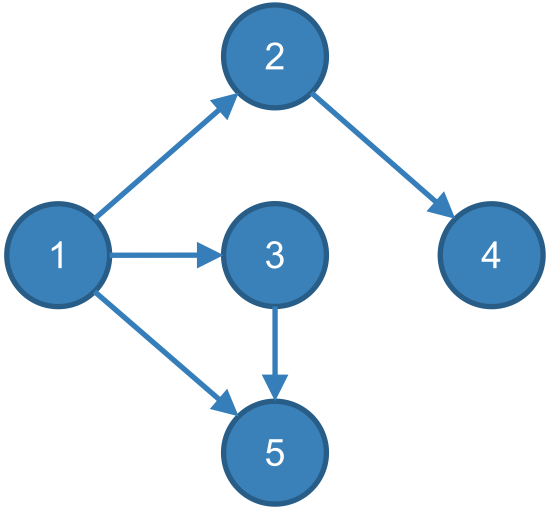
Algoritmo naïve

- Sfruttiamo la proprietà che un sottografo di un DAG è un DAG

```
TOPOLOGICALSORT(G):
  Sequence order ← Ø
  for i ← | G.vertices() | - 1 downto 0:
    v ← nodo pozzo in G.V()
    order.append(v)
    G.removeNode(v) // Rimuove anche tutti gli archi incidenti
```

Topological Sort con DFS e Stack

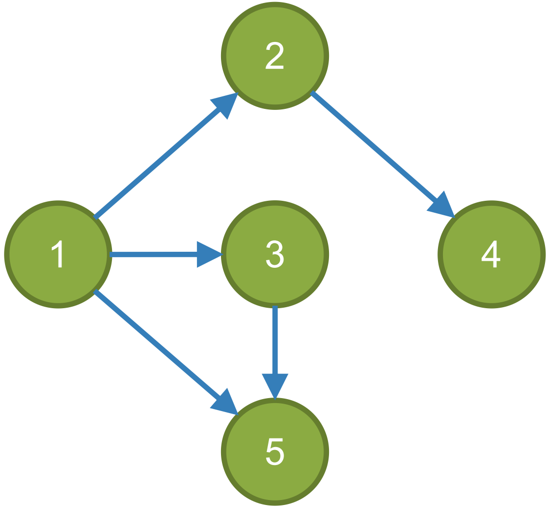
```
TOPOLOGICALSORT(G):
    Stack S ← Ø
    foreach u in G.V():
        u.vsited ← false
    foreach u in G.V():
        if not u.visited then:
            TOPOLOGICALSORTDFS(G, u, S)
    return S

TOPOLOGICALSORTDFS(G, u, S):
    u.visited ← true
    foreach v in G.adj(u):
        if not v.visited then:
            TOPOLOGICALSORTDFS(G, v, S)
        S.push(u)
```



Stack = { }

Topological Sort con DFS e Stack

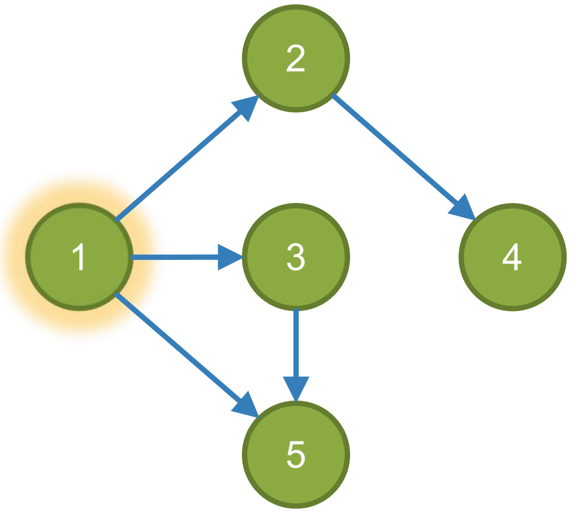
```
TOPOLOGICALSORT(G):
    Stack S ← Ø
    foreach u in G.V():
        u.vsited ← false
    foreach u in G.V():
        if not u.visited then:
            TOPOLOGICALSORTDFS(G, u, S)
    return S

TOPOLOGICALSORTDFS(G, u, S):
    u.visited ← true
    foreach v in G.adj(u):
        if not v.visited then:
            TOPOLOGICALSORTDFS(G, v, S)
        S.push(u)
```



Stack = { }

Topological Sort con DFS e Stack

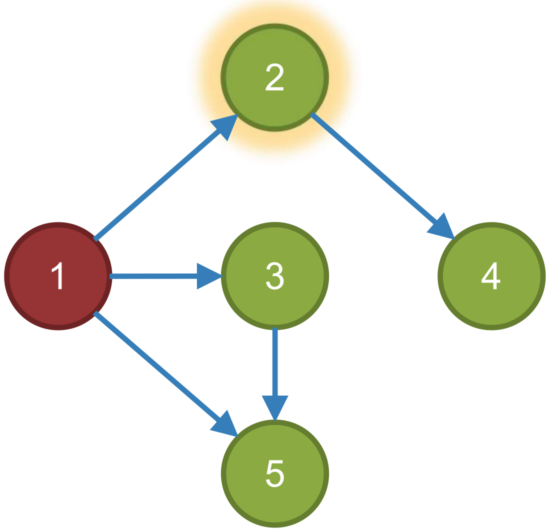
```
TOPOLOGICALSORT(G):
    Stack S ← Ø
    foreach u in G.V():
        u.vsited ← false
    foreach u in G.V():
        if not u.visited then:
            TOPOLOGICALSORTDFS(G, u, S)
    return S

TOPOLOGICALSORTDFS(G, u, S):
    u.visited ← true
    foreach v in G.adj(u):
        if not v.visited then:
            TOPOLOGICALSORTDFS(G, v, S)
        S.push(u)
```



Stack = { }

Topological Sort con DFS e Stack

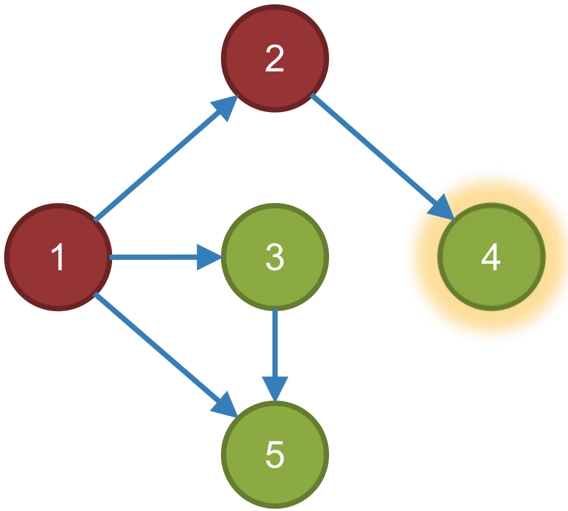
```
TOPOLOGICALSORT(G):
   Stack S ← Ø
   foreach u in G.V():
      u.vsited ← false
   foreach u in G.V():
      if not u.visited then:
         TOPOLOGICALSORTDFS(G, u, S)
   return S

TOPOLOGICALSORTDFS(G, u, S):
   u.visited ← true
   foreach v in G.adj(u):
      if not v.visited then:
         TOPOLOGICALSORTDFS(G, v, S)
   S.push(u)
```



Stack = { }

Topological Sort con DFS e Stack

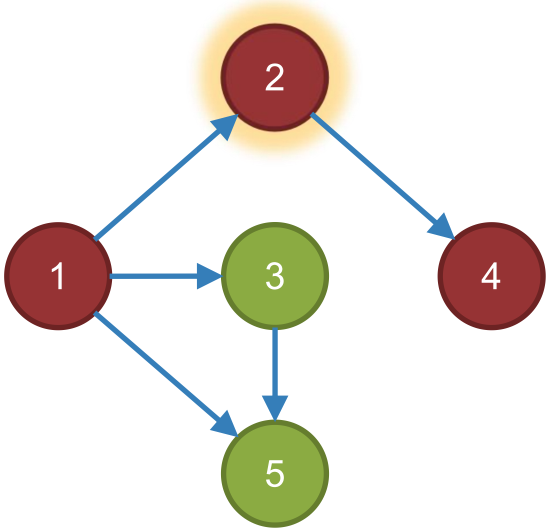
```
TOPOLOGICALSORT(G):
    Stack S ← Ø
    foreach u in G.V():
        u.vsited ← false
    foreach u in G.V():
        if not u.visited then:
            TOPOLOGICALSORTDFS(G, u, S)
    return S

TOPOLOGICALSORTDFS(G, u, S):
    u.visited ← true
    foreach v in G.adj(u):
        if not v.visited then:
            TOPOLOGICALSORTDFS(G, v, S)
        S.push(u)
```



Stack = { }

Topological Sort con DFS e Stack

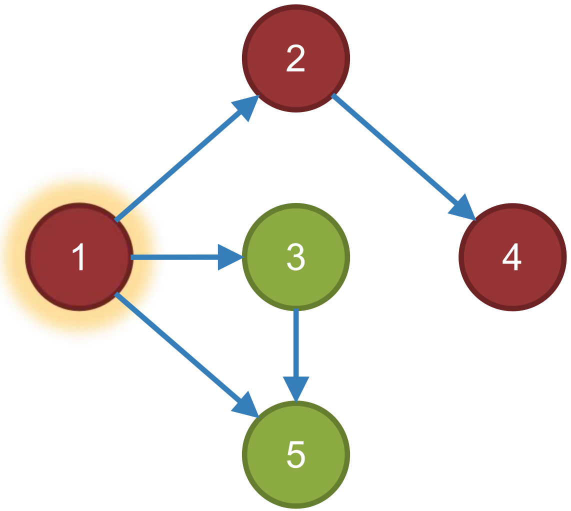
```
TOPOLOGICALSORT(G):
    Stack S ← Ø
    foreach u in G.V():
        u.vsited ← false
    foreach u in G.V():
        if not u.visited then:
            TOPOLOGICALSORTDFS(G, u, S)
    return S

TOPOLOGICALSORTDFS(G, u, S):
    u.visited ← true
    foreach v in G.adj(u):
        if not v.visited then:
            TOPOLOGICALSORTDFS(G, v, S)
        S.push(u)
```



Stack = { 4 }

Topological Sort con DFS e Stack

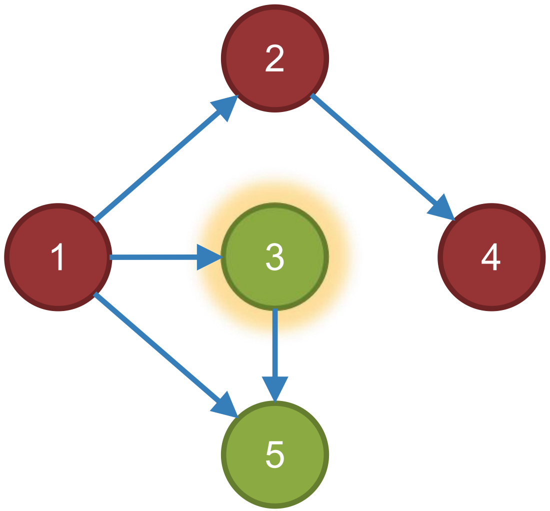
```
TOPOLOGICALSORT(G):
    Stack S ← Ø
    foreach u in G.V():
        u.vsited ← false
    foreach u in G.V():
        if not u.visited then:
            TOPOLOGICALSORTDFS(G, u, S)
    return S

TOPOLOGICALSORTDFS(G, u, S):
    u.visited ← true
    foreach v in G.adj(u):
        if not v.visited then:
            TOPOLOGICALSORTDFS(G, v, S)
    S.push(u)
```



Stack = { 2, 4 }

Topological Sort con DFS e Stack

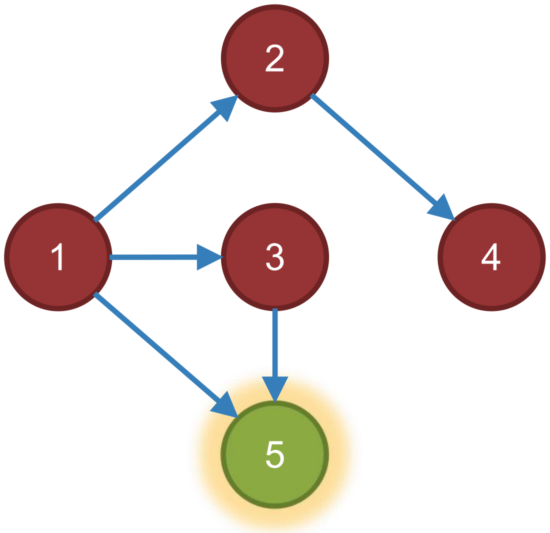
```
TOPOLOGICALSORT(G):
   Stack S ← Ø
   foreach u in G.V():
      u.vsited ← false
   foreach u in G.V():
      if not u.visited then:
         TOPOLOGICALSORTDFS(G, u, S)
   return S

TOPOLOGICALSORTDFS(G, u, S):
   u.visited ← true
   foreach v in G.adj(u):
      if not v.visited then:
         TOPOLOGICALSORTDFS(G, v, S)
      S.push(u)
```



Stack = { 2, 4 }

Topological Sort con DFS e Stack

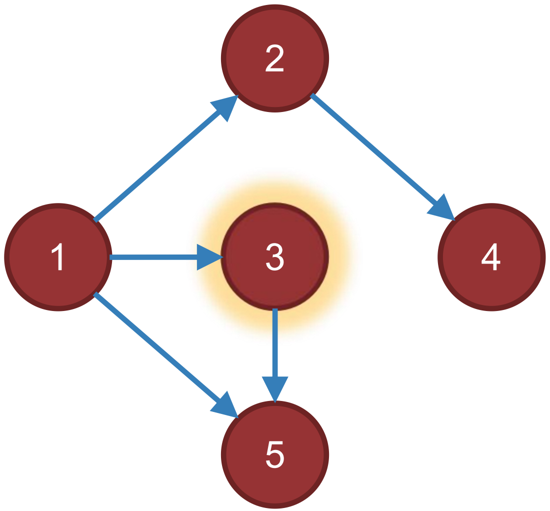
```
TOPOLOGICALSORT(G):
    Stack S ← Ø
    foreach u in G.V():
        u.vsited ← false
    foreach u in G.V():
        if not u.visited then:
            TOPOLOGICALSORTDFS(G, u, S)
    return S

TOPOLOGICALSORTDFS(G, u, S):
    u.visited ← true
    foreach v in G.adj(u):
        if not v.visited then:
            TOPOLOGICALSORTDFS(G, v, S)
        S.push(u)
```



Stack = { 2, 4 }

Topological Sort con DFS e Stack

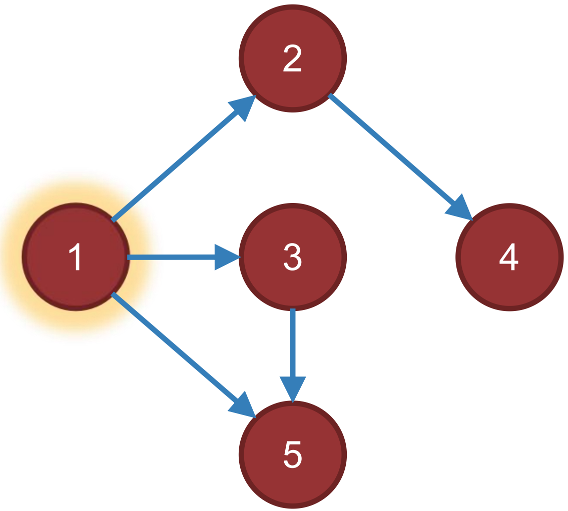
```
TOPOLOGICALSORT(G):
    Stack S ← Ø
    foreach u in G.V():
        u.vsited ← false
    foreach u in G.V():
        if not u.visited then:
            TOPOLOGICALSORTDFS(G, u, S)
    return S

TOPOLOGICALSORTDFS(G, u, S):
    u.visited ← true
    foreach v in G.adj(u):
        if not v.visited then:
            TOPOLOGICALSORTDFS(G, v, S)
    S.push(u)
```



Stack = { 5, 2, 4 }

Topological Sort con DFS e Stack

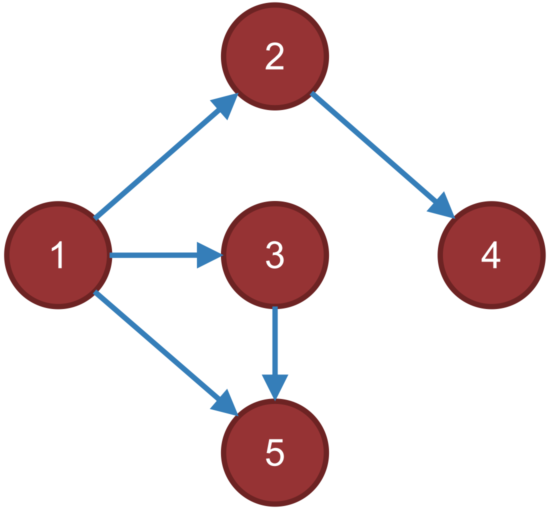
```
TOPOLOGICALSORT(G):
    Stack S ← Ø
    foreach u in G.V():
        u.vsited ← false
    foreach u in G.V():
        if not u.visited then:
            TOPOLOGICALSORTDFS(G, u, S)
    return S

TOPOLOGICALSORTDFS(G, u, S):
    u.visited ← true
    foreach v in G.adj(u):
        if not v.visited then:
            TOPOLOGICALSORTDFS(G, v, S)
    S.push(u)
```



Stack = { 3, 5, 2, 4 }

Topological Sort con DFS e Stack

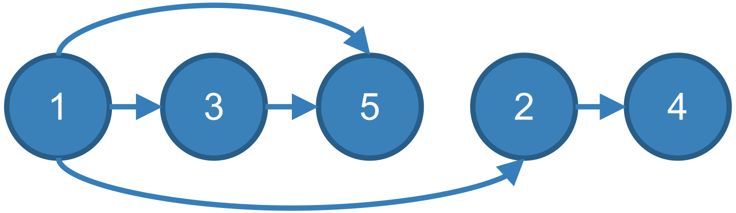
```
TOPOLOGICALSORT(G):
    Stack S ← Ø
    foreach u in G.V():
        u.vsited ← false
    foreach u in G.V():
        if not u.visited then:
            TOPOLOGICALSORTDFS(G, u, S)
    return S

TOPOLOGICALSORTDFS(G, u, S):
    u.visited ← true
    foreach v in G.adj(u):
        if not v.visited then:
            TOPOLOGICALSORTDFS(G, v, S)
        S.push(u)
```



Stack = { 1, 3, 5, 2, 4 }

Topological Sort con DFS e Stack

```
TOPOLOGICALSORT(G):
    Stack S ← Ø
    foreach u in G.V():
        u.vsited ← false
    foreach u in G.V():
        if not u.visited then:
            TOPOLOGICALSORTDFS(G, u, S)
    return S

TOPOLOGICALSORTDFS(G, u, S):
    u.visited ← true
    foreach v in G.adj(u):
        if not v.visited then:
            TOPOLOGICALSORTDFS(G, v, S)
        S.push(u)
```

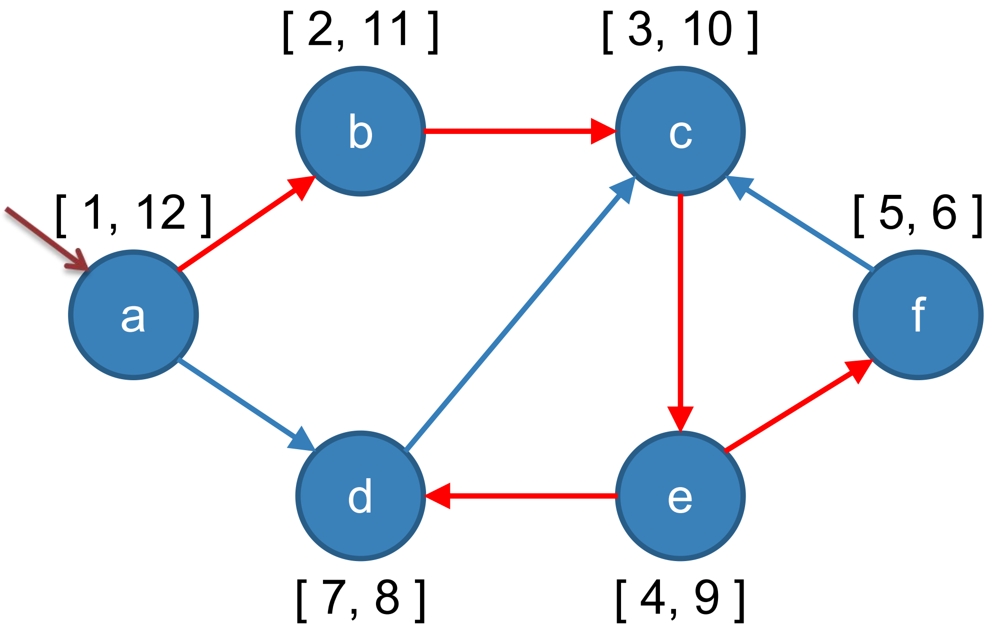


Stack = { 1, 3, 5, 2, 4 }

Ordinamento topologico in grafi generici

- Si può estendere l'algoritmo appena visto anche a grafici che contengono cicli
- Dall'esecuzione di TOPOLOGICALSORT() siamo sicuri che:
  - se un arco (u, v) non appartiene a un ciclo, u appare prima di v nell'ordinamento
  - Gli archi che appartengono a un ciclo appaiono in un qualche ordine, che non è influente ai fini dell'ordinamento
- Si può generalizzare quindi TOPOLOGICALSORT() semplicemente ordinando i nodi per timestamp decrescente di uscita

Ordinamento topologico in grafi generici



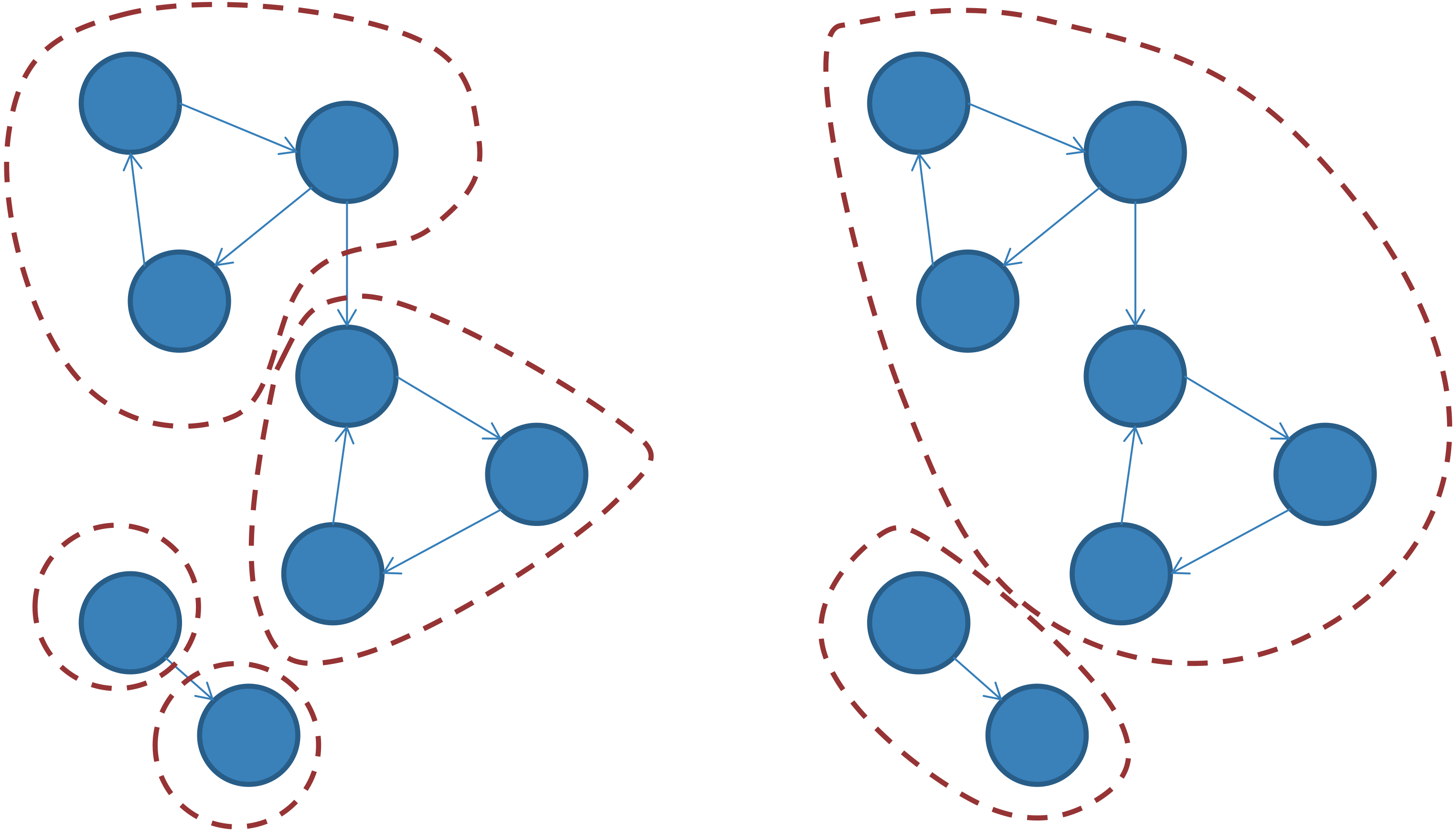
Stack = { a, b, c, e, d, f }

Componenti connesse

Connettività nei grafi diretti

- Due nodi u, v sono connessi in un grafo orientato se esiste un cammino che collega u a v.
- Un grafo diretto è *fortemente connesso* se per ogni coppia u, v esiste un cammino da u a v.
- Un grafo diretto è *debolmente connesso* se ogni coppia di nodi è connessa da un cammino solo sostituendo gli archi orientati con archi non orientati.
- Una *componente connessa* di un grafo G è un sottografo G' connesso e *massimale* di G
  - Un sottografo G' è massimale se non esiste un altro sottografo G'' tale che $G' \subset G''$

Componenti connesse e fortemente connesse

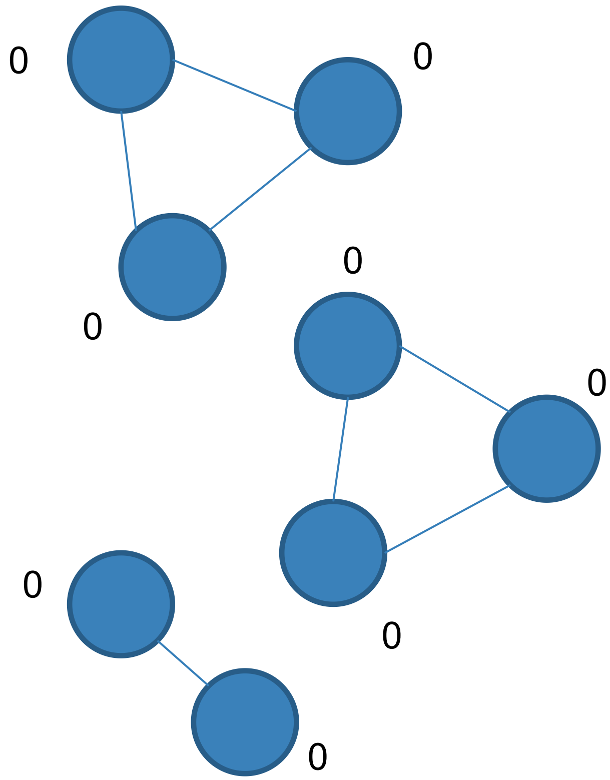


Individuazione delle componenti connesse

- Il problema è interessante quando si ha di fronte una *foresta*
  - Altrimenti, l'intero grafo è una singola componente connessa
- L'individuazione delle componenti si può effettuare tramite DFS
  - Ogni nodo viene “aumentato” con un intero che indica la componente connessa cui esso appartiene
  - Si itera su tutti i nodi del grafo
  - Se un nodo non appartiene ancora ad una componente connessa, si visita il grafo a partire da quel nodo associando tutti nodi visitati alla stessa componente connessa

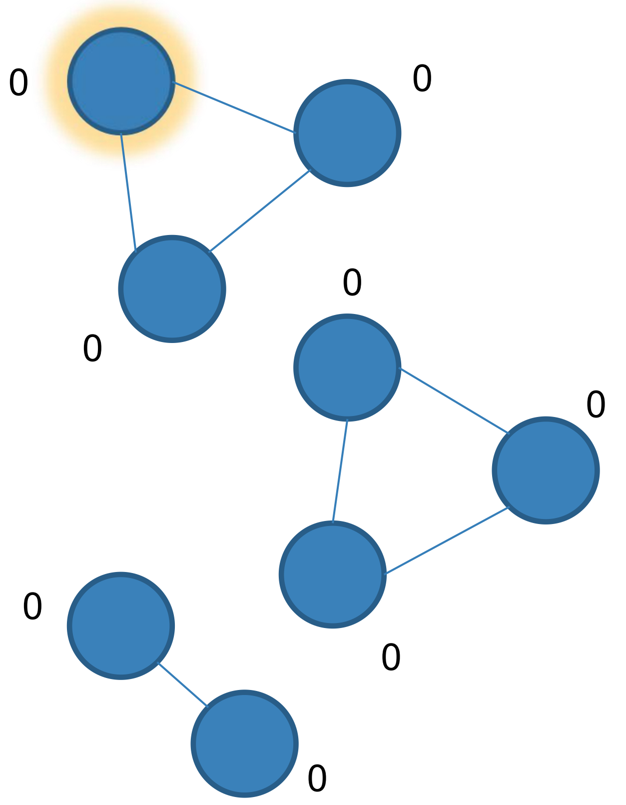
Individuazione delle componenti connesse

```
ccDFS(G, id, r):
    r.id ← id
    foreach v in G.adj(u):
        if v.id = 0 then:
            ccDFS(G, id, v)

cc(G):
    foreach u in G.V():
        u.id ← 0
    id ← 0
    foreach u in G.V():
        if u.id = 0 then:
            id ← id + 1
            ccdfs(G, id, u)
```



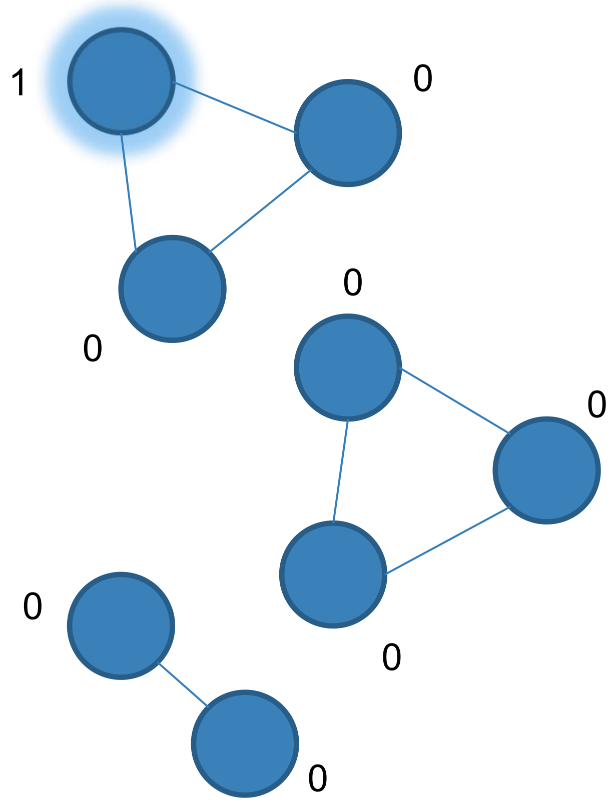
Individuazione delle componenti connesse

```
ccDFS(G, id, r):
    r.id ← id
    foreach v in G.adj(u):
        if v.id = 0 then:
            ccDFS(G, counter , v, id)

cc(G):
    foreach u in G.V():
        u.id ← 0
    id ← 0
    foreach u in G.V():
        if u.id = 0 then:
            id ← id + 1
            ccdfs(G, id, u)
```



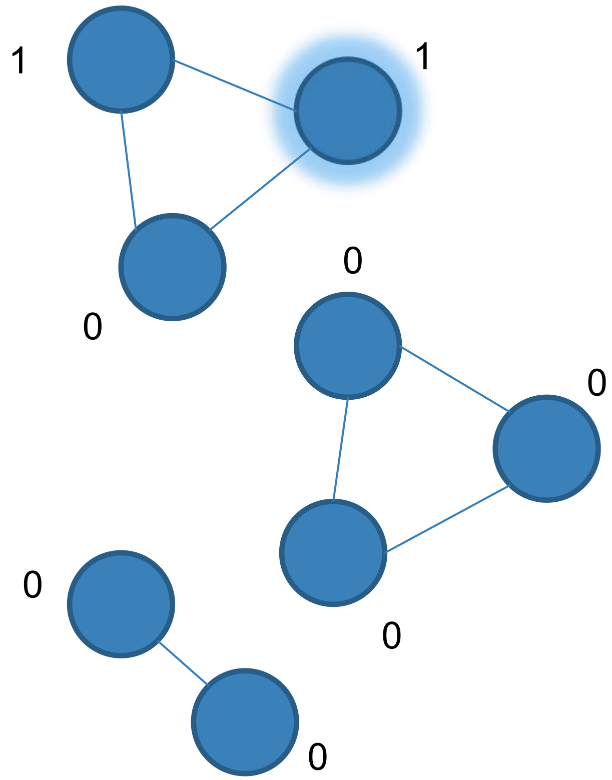
Individuazione delle componenti connesse

```
ccDFS(G, id, r):
    r.id ← id
    foreach v in G.adj(u):
        if v.id = 0 then:
            ccDFS(G, counter , v, id)

cc(G):
    foreach u in G.V():
        u.id ← 0
    id ← 0
    foreach u in G.V():
        if u.id = 0 then:
            id ← id + 1
            ccdfs(G, id, u)
```



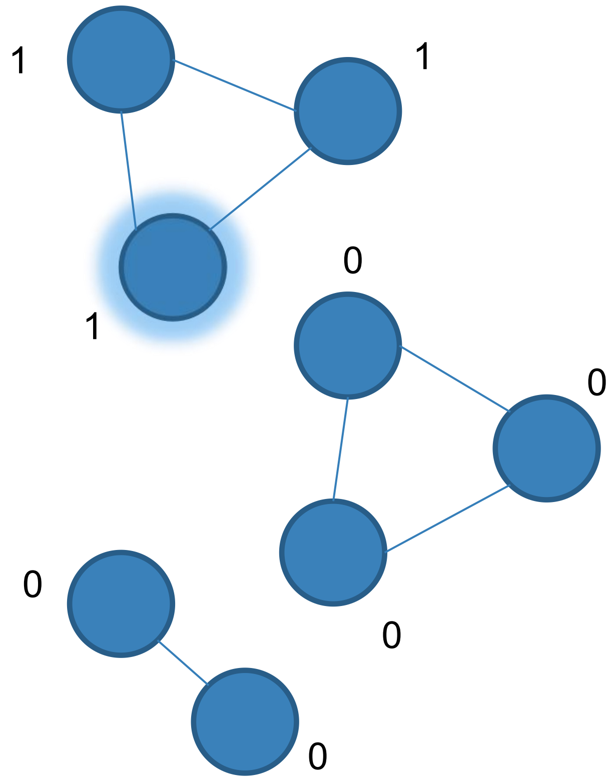
Individuazione delle componenti connesse

```
ccDFS(G, id, r):
    r.id ← id
    foreach v in G.adj(u):
        if v.id = 0 then:
            ccDFS(G, counter , v, id)

cc(G):
    foreach u in G.V():
        u.id ← 0
    id ← 0
    foreach u in G.V():
        if u.id = 0 then:
            id ← id + 1
            ccdfs(G, id, u)
```



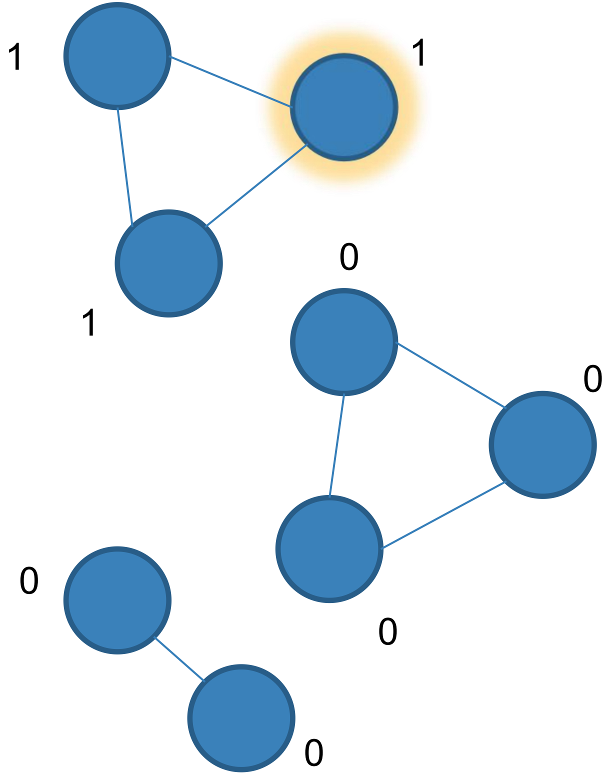
Individuazione delle componenti connesse

```
ccDFS(G, id, r):
   r.id ← id
   foreach v in G.adj(u):
      if v.id = 0 then:
         ccDFS(G, counter , v, id)

cc(G):
   foreach u in G.V():
      u.id ← 0
   id ← 0
   foreach u in G.V():
      if u.id = 0 then:
         id ← id + 1
         ccdfs(G, id, u)
```



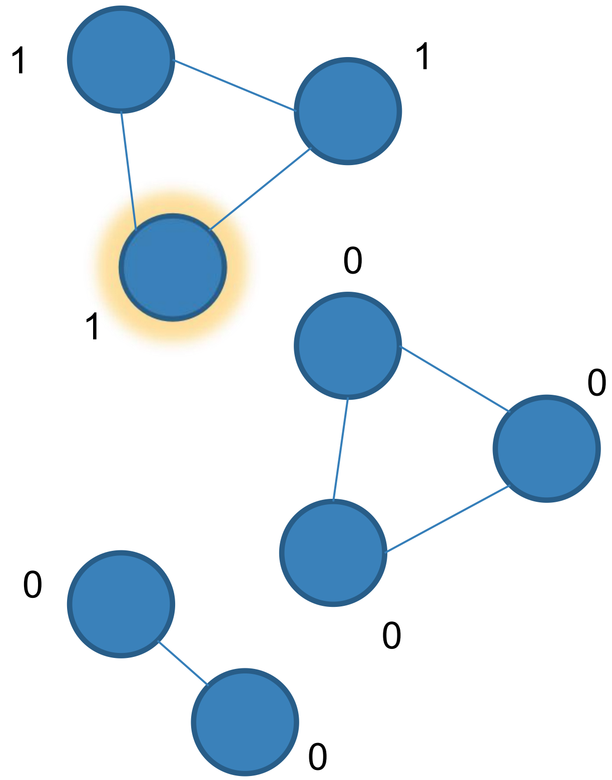
Individuazione delle componenti connesse

```
ccDFS(G, id, r):
   r.id ← id
   foreach v in G.adj(u):
      if v.id = 0 then:
         ccDFS(G, counter , v, id)

cc(G):
   foreach u in G.V():
      u.id ← 0
   id ← 0
   foreach u in G.V():
      if u.id = 0 then:
         id ← id + 1
         ccdfs(G, id, u)
```



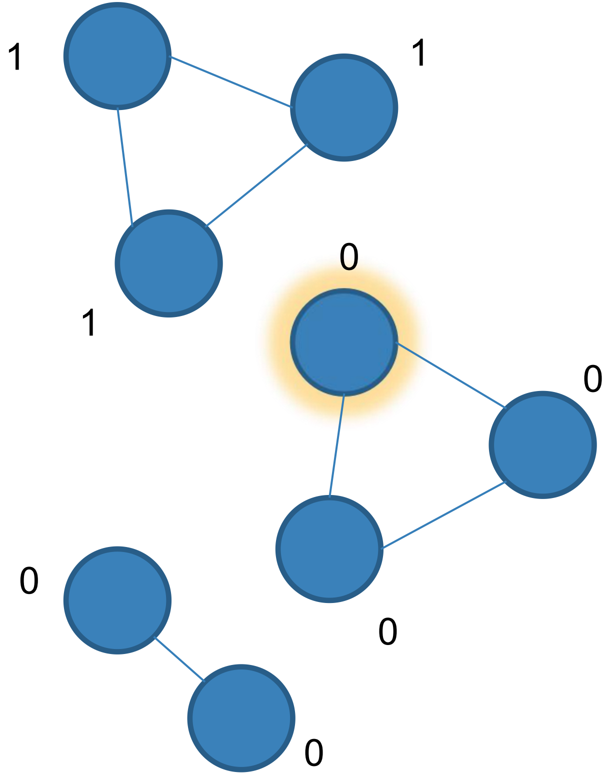
Individuazione delle componenti connesse

```
ccDFS(G, id, r):
    r.id ← id
    foreach v in G.adj(u):
        if v.id = 0 then:
            ccDFS(G, counter , v, id)

cc(G):
    foreach u in G.V():
        u.id ← 0
    id ← 0
    foreach u in G.V():
        if u.id = 0 then:
            id ← id + 1
            ccdfs(G, id, u)
```



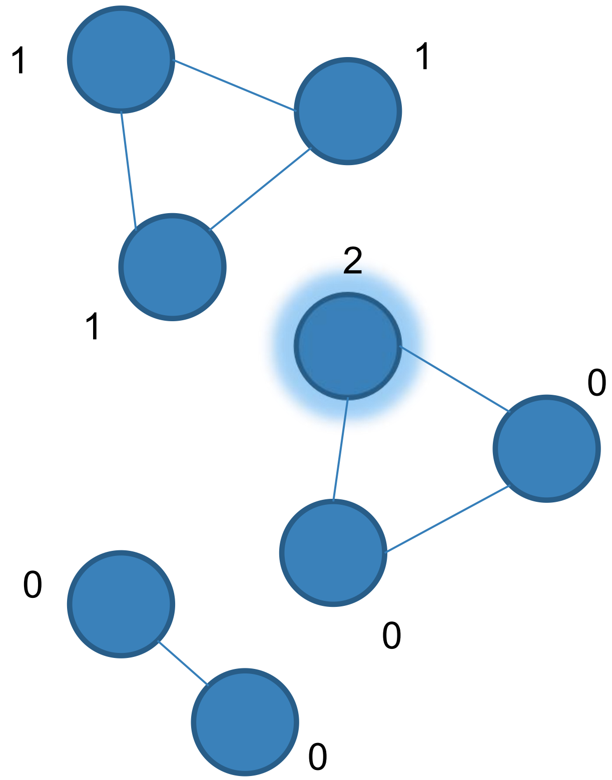
Individuazione delle componenti connesse

```
ccDFS(G, id, r):
    r.id ← id
    foreach v in G.adj(u):
        if v.id = 0 then:
            ccDFS(G, counter , v, id)

cc(G):
    foreach u in G.V():
        u.id ← 0
    id ← 0
    foreach u in G.V():
        if u.id = 0 then:
            id ← id + 1
            ccdfs(G, id, u)
```



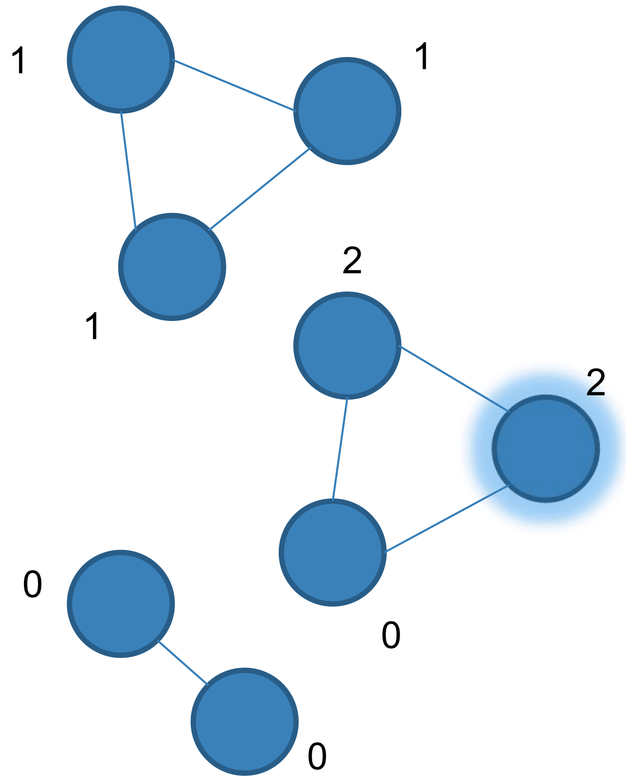
Individuazione delle componenti connesse

```
ccDFS(G, id, r):
    r.id ← id
    foreach v in G.adj(u):
        if v.id = 0 then:
            ccDFS(G, counter , v, id)

cc(G):
    foreach u in G.V():
        u.id ← 0
    id ← 0
    foreach u in G.V():
        if u.id = 0 then:
            id ← id + 1
            ccdfs(G, id, u)
```



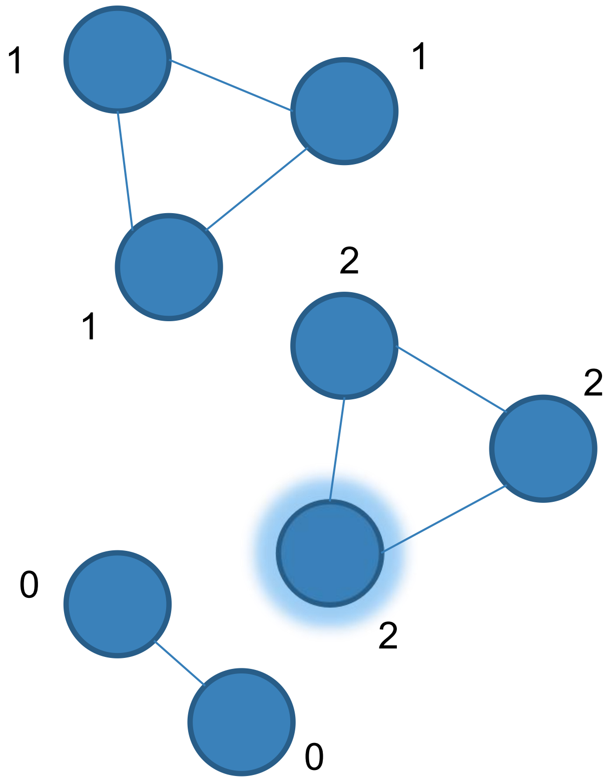
Individuazione delle componenti connesse

```
ccDFS(G, id, r):
    r.id ← id
    foreach v in G.adj(u):
        if v.id = 0 then:
            ccDFS(G, counter , v, id)

cc(G):
    foreach u in G.V():
        u.id ← 0
    id ← 0
    foreach u in G.V():
        if u.id = 0 then:
            id ← id + 1
            ccdfs(G, id, u)
```



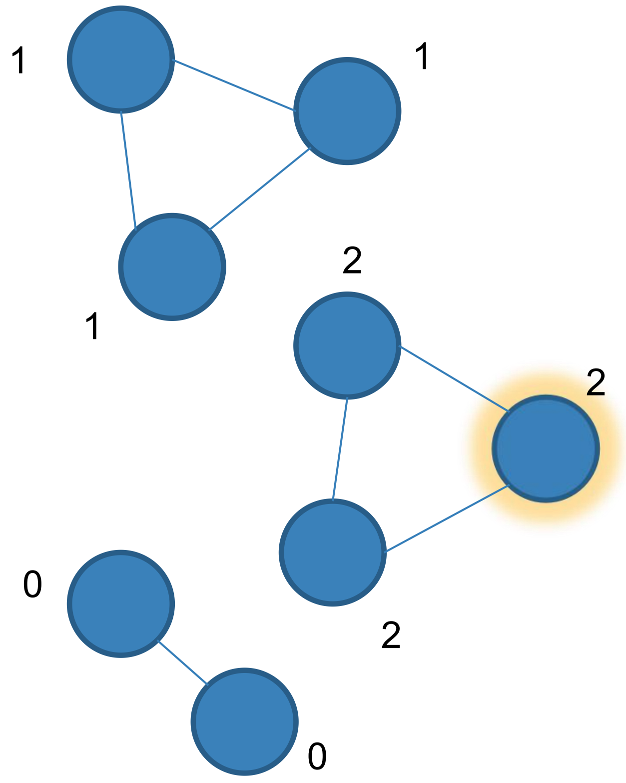
Individuazione delle componenti connesse

```
ccDFS(G, id, r):
    r.id ← id
    foreach v in G.adj(u):
        if v.id = 0 then:
            ccDFS(G, counter , v, id)

cc(G):
    foreach u in G.V():
        u.id ← 0
    id ← 0
    foreach u in G.V():
        if u.id = 0 then:
            id ← id + 1
            ccdfs(G, id, u)
```



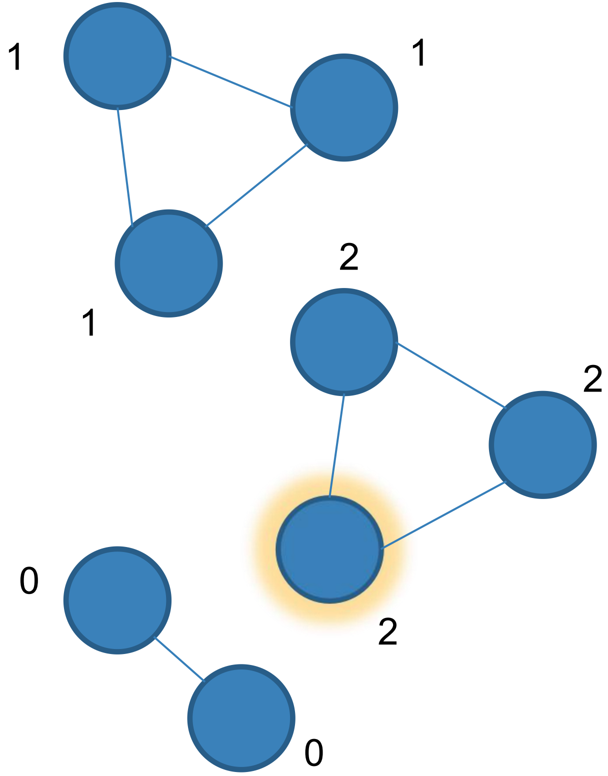
Individuazione delle componenti connesse

```
ccDFS(G, id, r):
    r.id ← id
    foreach v in G.adj(u):
        if v.id = 0 then:
            ccDFS(G, counter , v, id)

cc(G):
    foreach u in G.V():
        u.id ← 0
    id ← 0
    foreach u in G.V():
        if u.id = 0 then:
            id ← id + 1
            ccdfs(G, id, u)
```



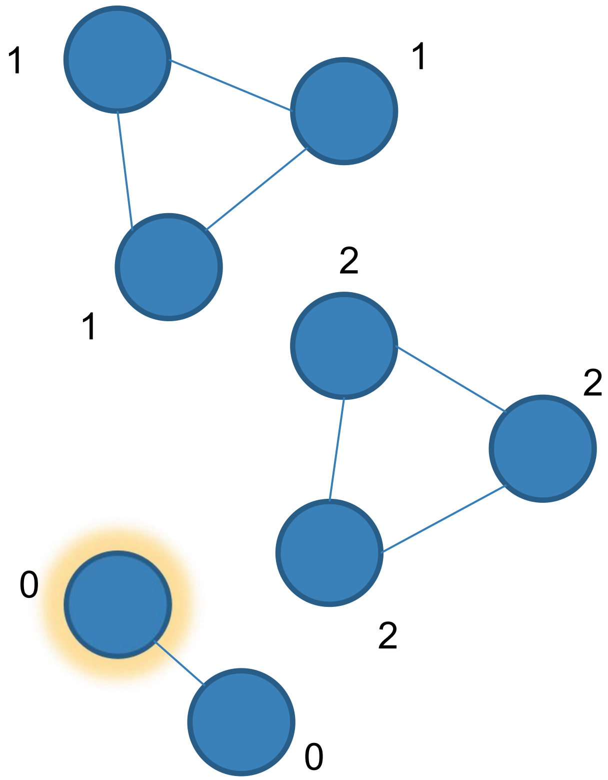
Individuazione delle componenti connesse

```
ccDFS(G, id, r):
    r.id ← id
    foreach v in G.adj(u):
        if v.id = 0 then:
            ccDFS(G, counter , v, id)

cc(G):
    foreach u in G.V():
        u.id ← 0
    id ← 0
    foreach u in G.V():
        if u.id = 0 then:
            id ← id + 1
            ccdfs(G, id, u)
```



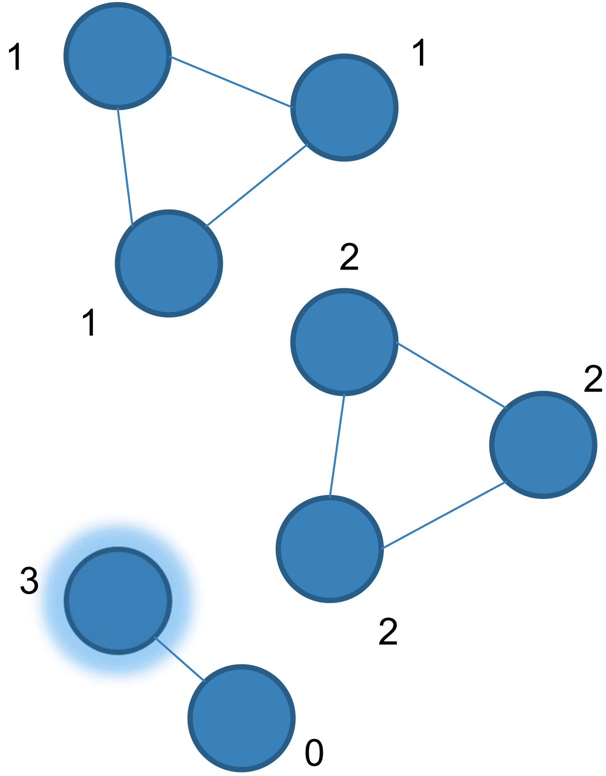
Individuazione delle componenti connesse

```
ccDFS(G, id, r):
    r.id ← id
    foreach v in G.adj(u):
        if v.id = 0 then:
            ccDFS(G, counter , v, id)

cc(G):
    foreach u in G.V():
        u.id ← 0
    id ← 0
    foreach u in G.V():
        if u.id = 0 then:
            id ← id + 1
            ccdfs(G, id, u)
```



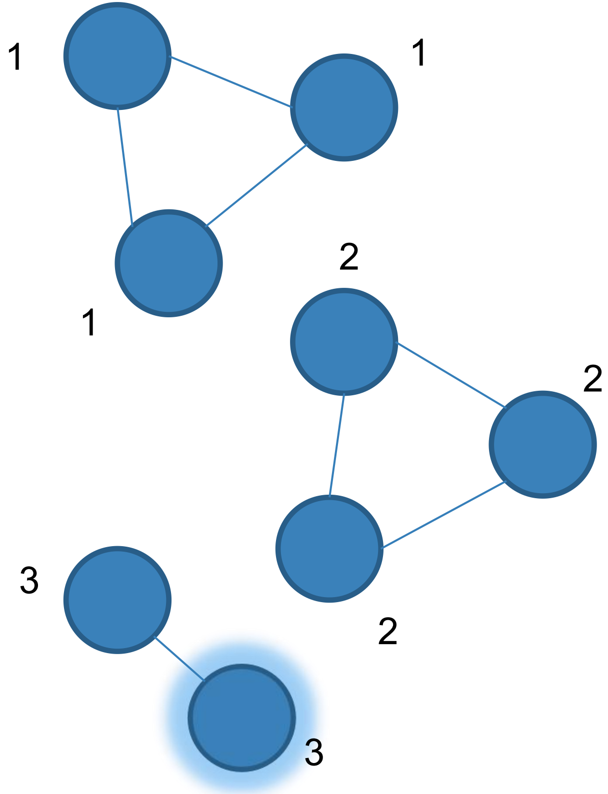
Individuazione delle componenti connesse

```
ccDFS(G, id, r):
    r.id ← id
    foreach v in G.adj(u):
        if v.id = 0 then:
            ccDFS(G, counter , v, id)

cc(G):
    foreach u in G.V():
        u.id ← 0
    id ← 0
    foreach u in G.V():
        if u.id = 0 then:
            id ← id + 1
            ccdfs(G, id, u)
```



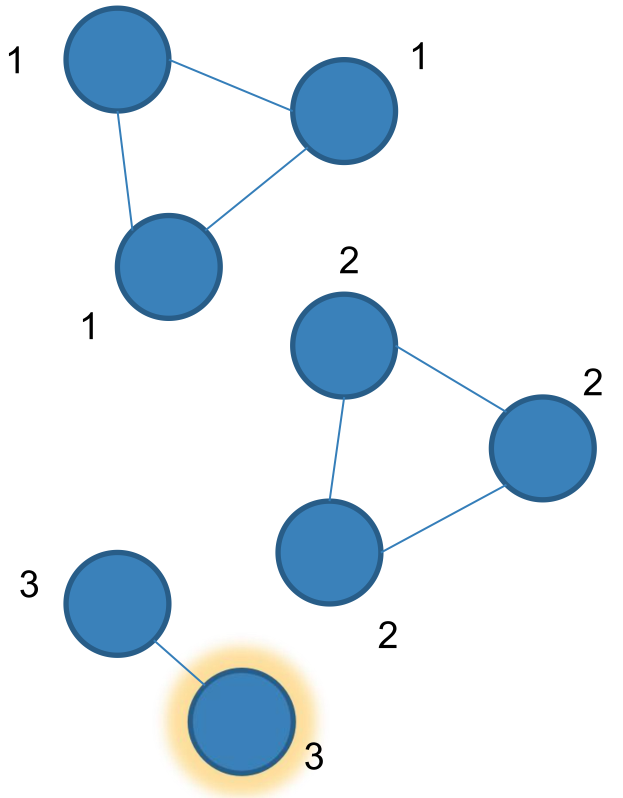
Individuazione delle componenti connesse

```
ccDFS(G, id, r):
    r.id ← id
    foreach v in G.adj(u):
        if v.id = 0 then:
            ccDFS(G, counter , v, id)

cc(G):
    foreach u in G.V():
        u.id ← 0
    id ← 0
    foreach u in G.V():
        if u.id = 0 then:
            id ← id + 1
            ccdfs(G, id, u)
```



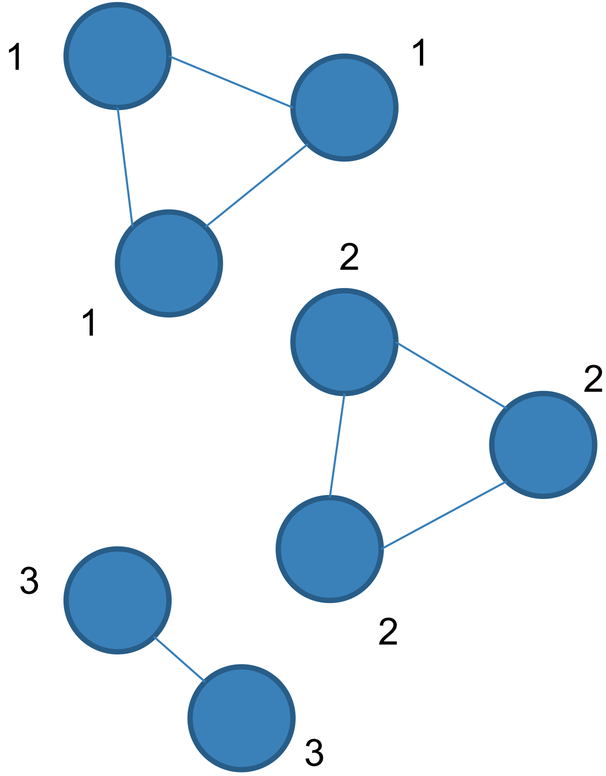
Individuazione delle componenti connesse

```
ccDFS(G, id, r):
    r.id ← id
    foreach v in G.adj(u):
        if v.id = 0 then:
            ccDFS(G, counter , v, id)

cc(G):
    foreach u in G.V():
        u.id ← 0
    id ← 0
    foreach u in G.V():
        if u.id = 0 then:
            id ← id + 1
            ccdfs(G, id, u)
```



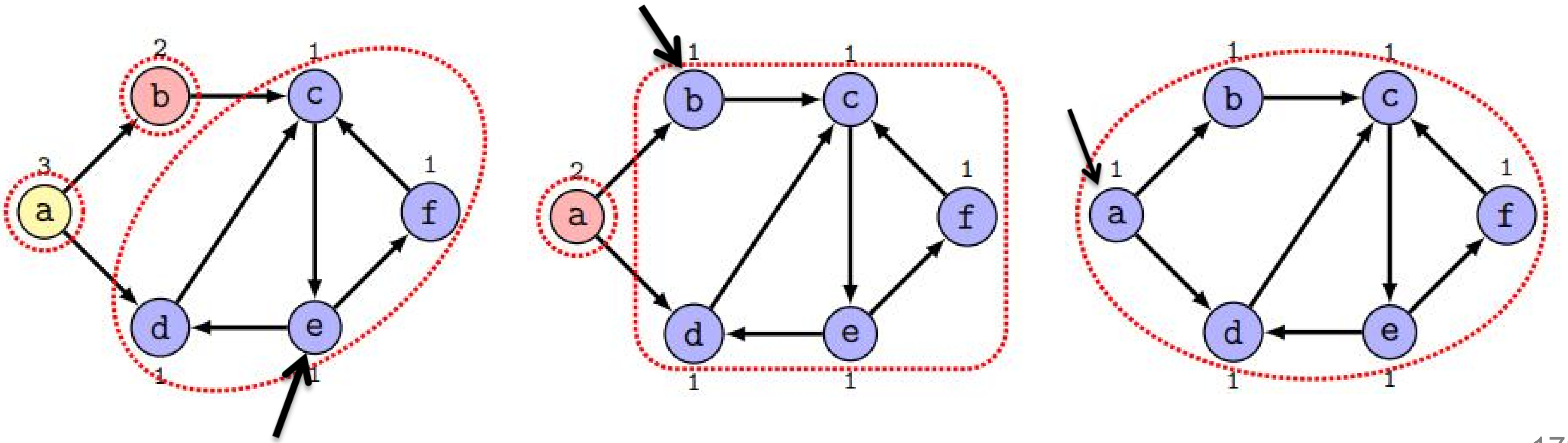
Individuazione delle componenti connesse

```
ccDFS(G, id, r):
    r.id ← id
    foreach v in G.adj(u):
        if v.id = 0 then:
            ccDFS(G, counter , v, id)

cc(G):
    foreach u in G.V():
        u.id ← 0
    id ← 0
    foreach u in G.V():
        if u.id = 0 then:
            id ← id + 1
            ccdfs(G, id, u)
```



Componenti fortemente connesse

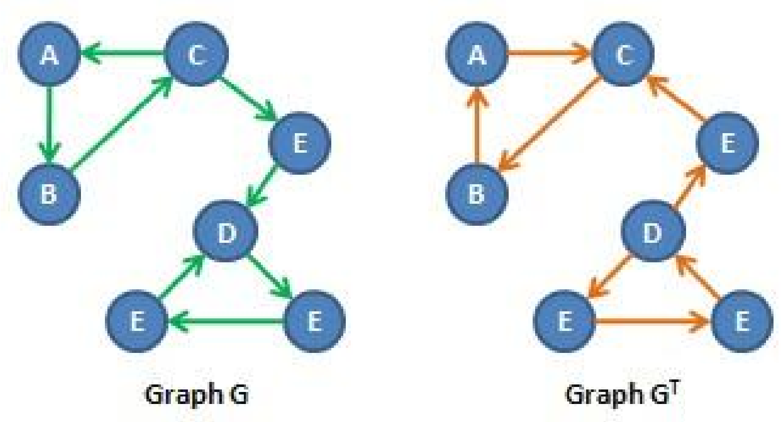
- Non è possibile applicare l'algoritmo precedente per l'individuazione delle componenti fortemente connesse
  - La correttezza del risultato dipende dal nodo di partenza



Grafo Trasposto

- Dato un grafo G=(V,E), il suo grafo trasposto $G^T = (V, E^T)$ ha gli stessi nodi e gli stessi archi orientati in verso opposto

```
TRANSPOSE(G):
    Graph Gᵀ ← Ø
    foreach u in G:
        Gᵀ.insertNode(u)
    foreach u in G:
        foreach v in G.adj(u):
            Gᵀ.insertEdge(v, u)
    return Gᵀ
```



Algoritmo di Kosaraju—1978

- L'algoritmo di Kosaraju individua le componenti fortemente connesse in tempo lineare $\Theta(n + m)$
- Si basa sulla generazione del grafo trasposto
- L'idea alla base è che un grafo G e il suo trasposto G^T hanno le stesse componenti fortemente connesse

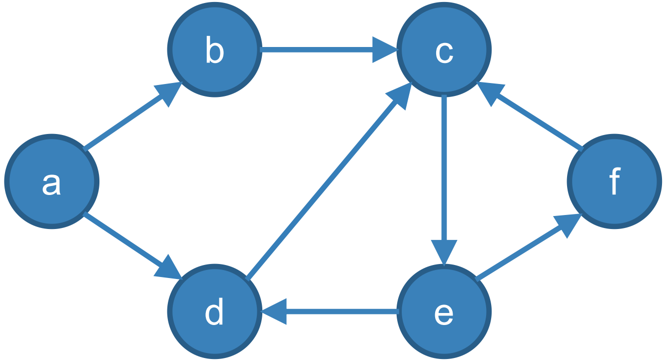
KOSARAJU(G):

  Stack S ← TOPOLOGICALSORT(G)

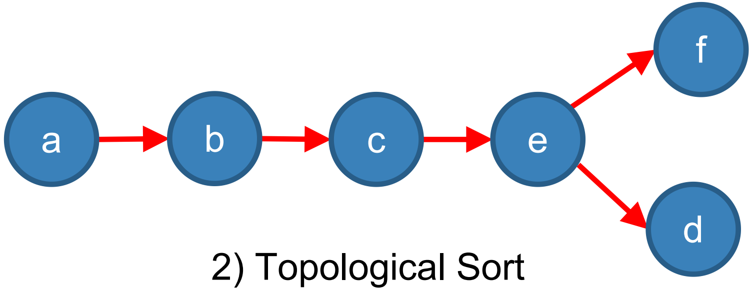
  G^T ← TRANSPOSE(G)

  return CC(G^T, S)

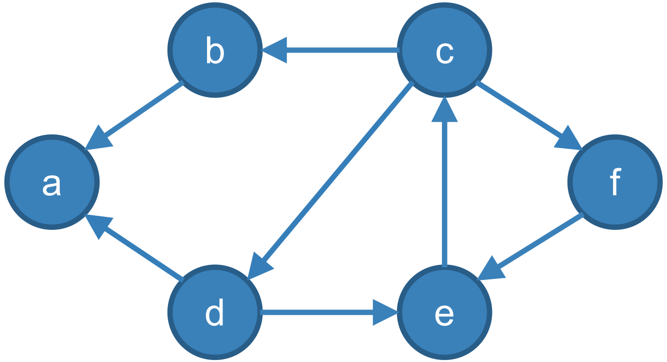
Algoritmo di Kosaraju—1978



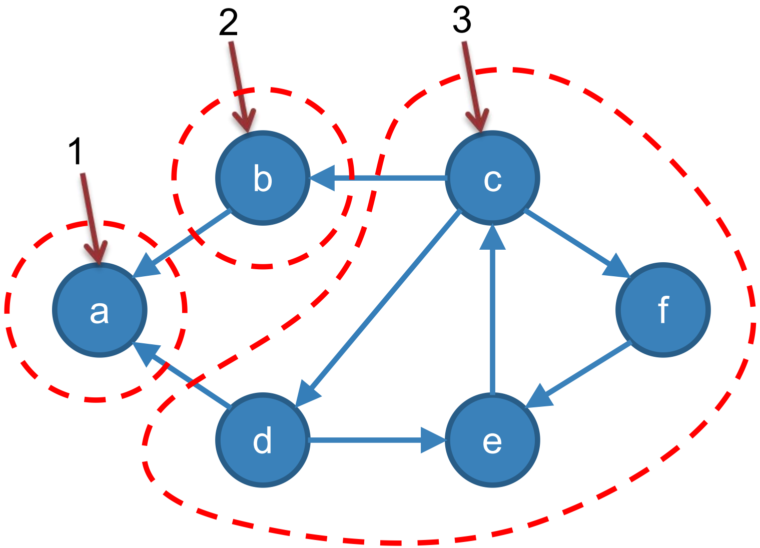
1) G



2) Topological Sort



3) G^T



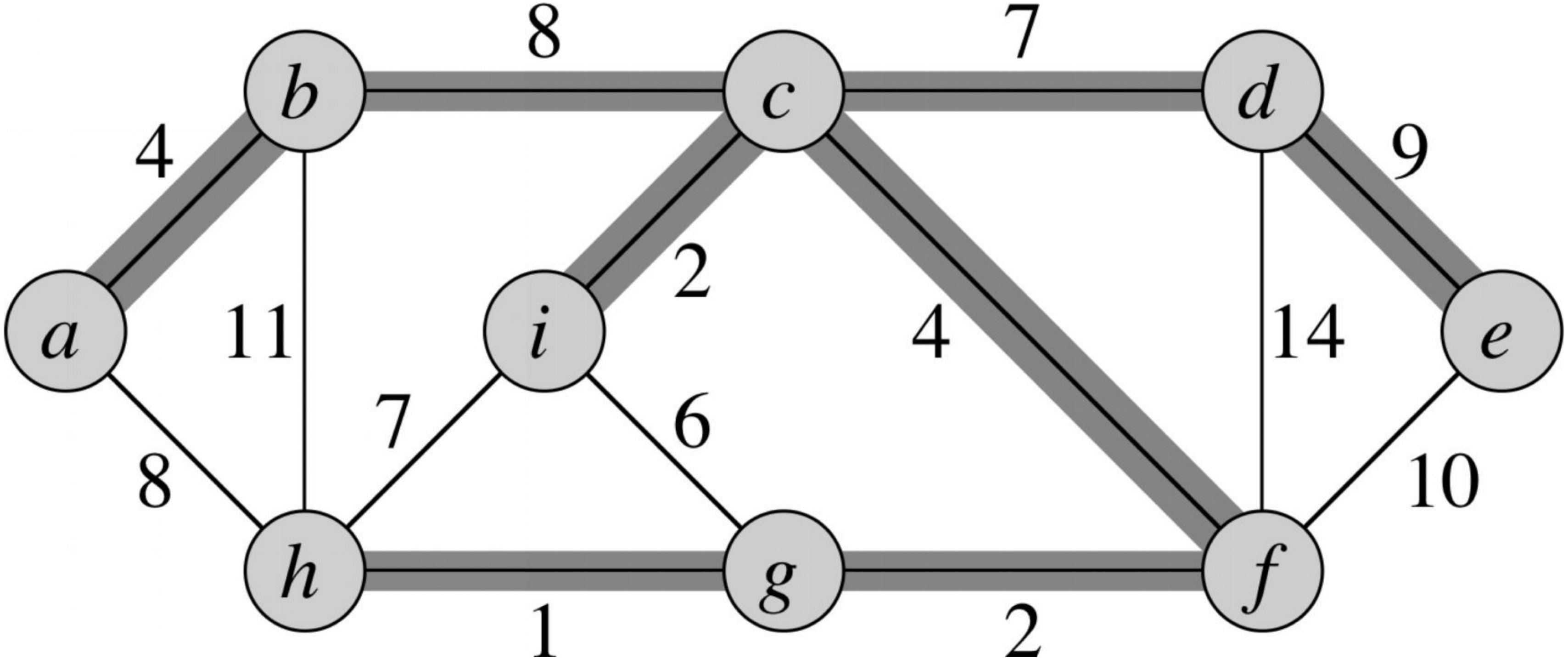
4) SCC

Minimo Albero Ricoprente

Minimum Spanning Tree

- Si vuole individuare un sottografo di un grafo che mantenga la connettività tra tutti i nodi *al minore costo possibile*
- La rete ottimale è sempre un albero
  - Gli alberi possono essere visti come grafi aciclici
- Nel caso di pesi non negativi, le applicazioni sono molteplici:
  - Identificazione di un sottoinsieme di tratte aeree per raggiungere tutte le destinazioni di interesse
  - Progettazione di infrastrutture (reti elettriche, reti idriche)
  - Permettono di trovare approssimazioni a problemi più difficili (es: il problema del commesso viaggiatore)
  - Clustering
  - Progettazione di circuiti elettronici

Minimum Spanning Tree



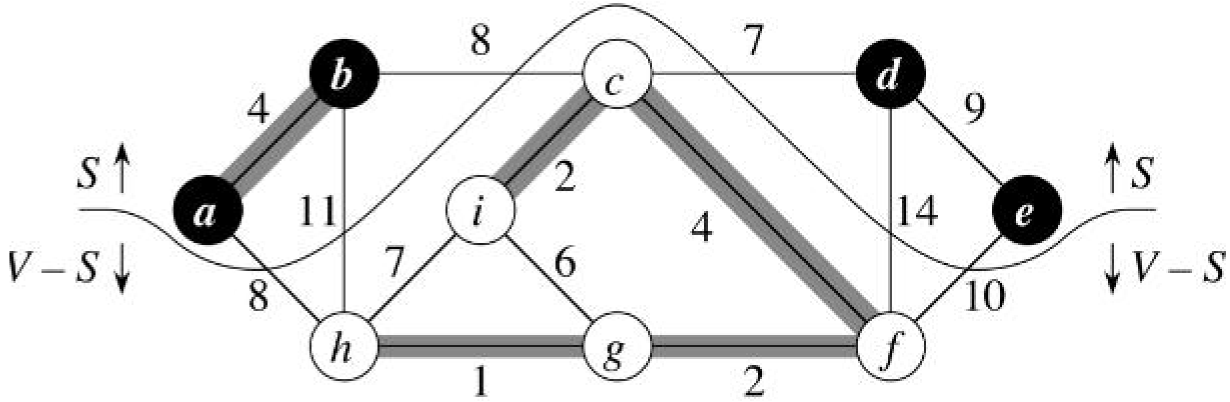
Minimum Spanning Tree

- La ricerca di un MST si basa su algoritmi greedy
- Questi algoritmi cercano di costruire incrementalmente il MST aggiungendo un arco alla volta
- L'ottimalità si ottiene se e solo se, ad ogni passo, non vengono mai inseriti archi che non appartengono al MST finale
  - È equivalente a dire che il MST costruito incrementalmente contiene archi che sono sempre un sottoinsieme degli archi che compongono il MST

- *Edge safety*: un arco è *sicuro* se può essere aggiunto al MST in costruzione senza violare l'ottimalità globale del MST

Edge Safety

- Un *taglio* è una partizione di V in due insiemi S e V\S
- Un arco (u,v) *attraversa il taglio* se u ∈ S e v ∈ V\S, o viceversa
- Un taglio *rispetta* un insieme di archi A ⊂ V se nessun arco in A attraversa il taglio
- Un *arco leggero* è l'arco a peso minimo tra quelli che attraversano un taglio



Costruzione di un MST

- Teorema: sia G = (V, E) un grafo connesso non diretto, con pesi non negativi per gli archi. Sia A un sottoinsieme di E, incluso in un qualche MST di G. Sia (S, V\S) un taglio qualsiasi di G che rispetta A, e sia (u, v) un arco leggero di questo taglio. Allora, (u, v) è un arco sicuro per A

TROVAMST(G):

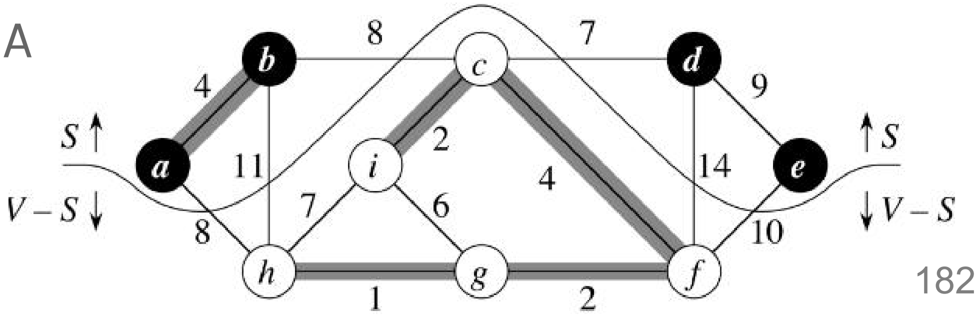
A ← Ø

while A non è un MST di G:

trova un arco (u, v) sicuro per A

A ← A ∪ {(u, v)}

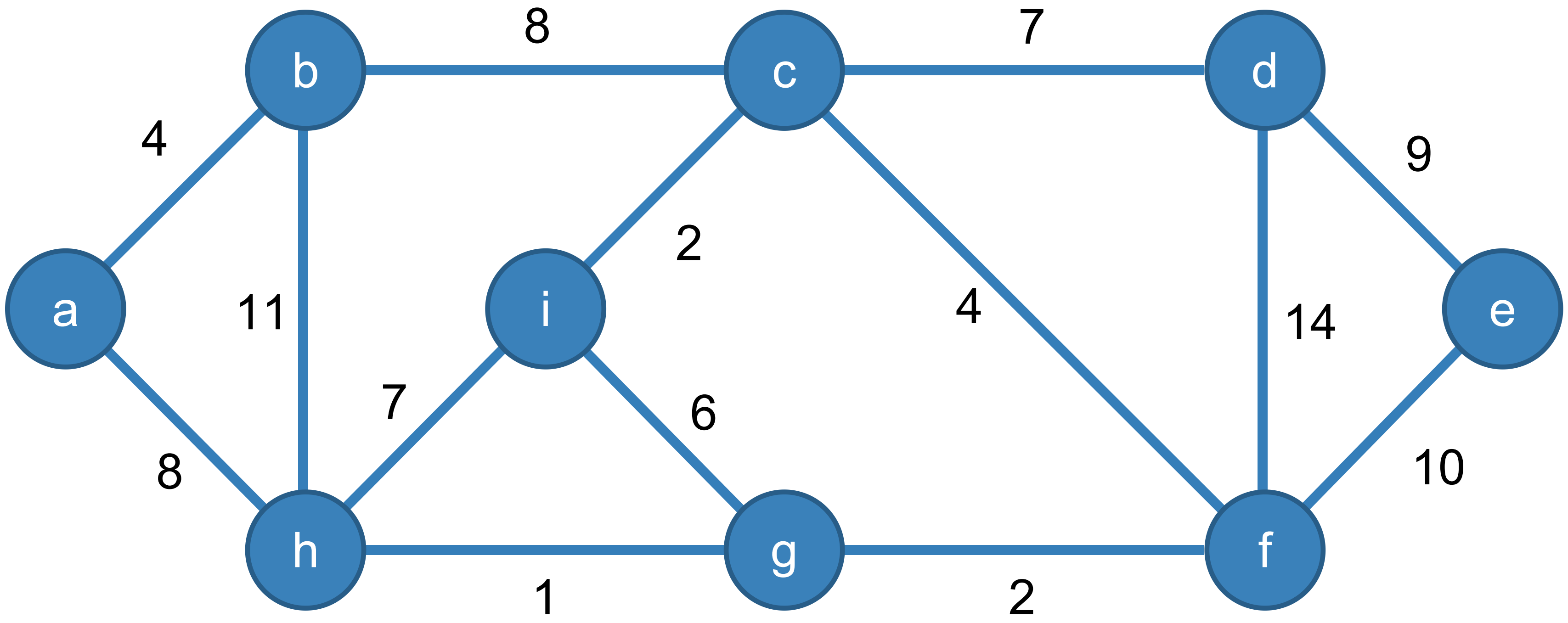
return A



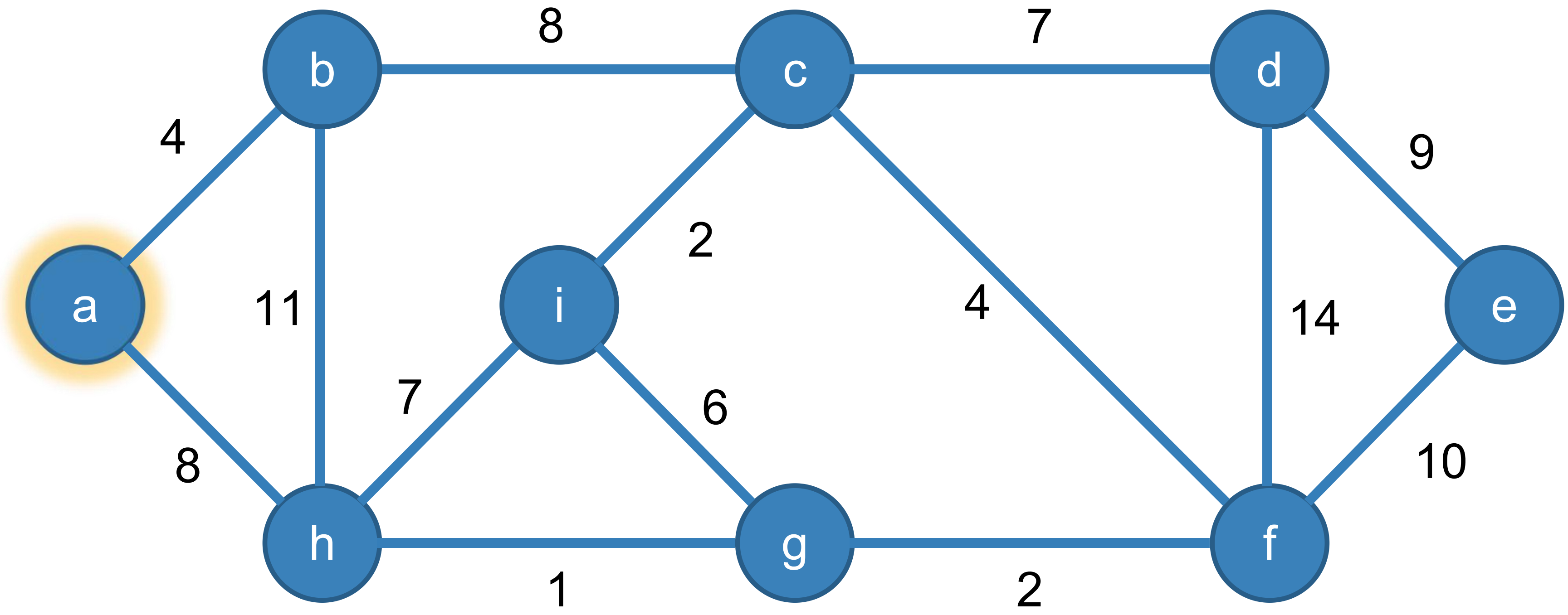
Algoritmo di Borůvka—1926

- L'insieme A forma un insieme di componenti connesse
  - Inizialmente, vi è una componente connessa per ciascun nodo
- Safety: viene determinato l'arco di costo minimo che connette due componenti connesse in A.
- Viene selezionata una componente connessa da A
- Si seleziona l'arco a peso minimo che connette la componente con un'altra componente di A
- Si rimuovono gli archi che connettono due vertici appartenenti alla stessa componente e non in A
- Complessità: $O(n \log m)$

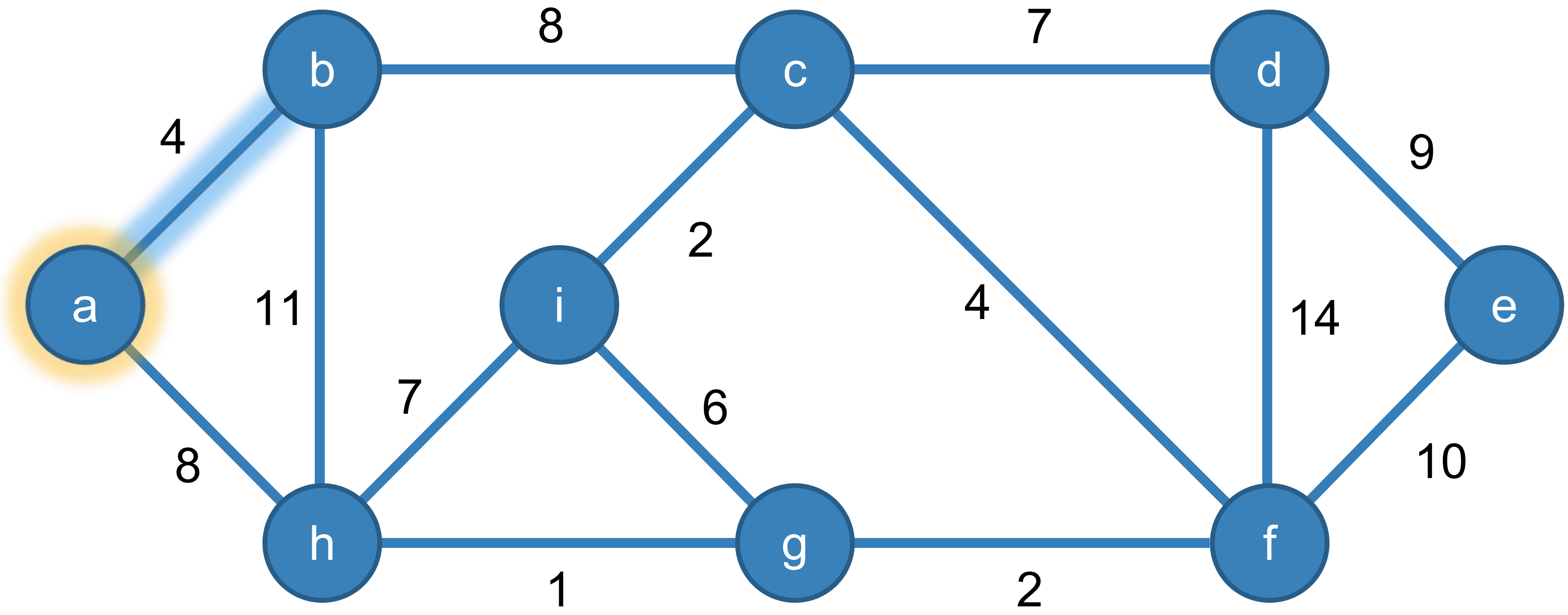
Algoritmo di Borůvka—1926



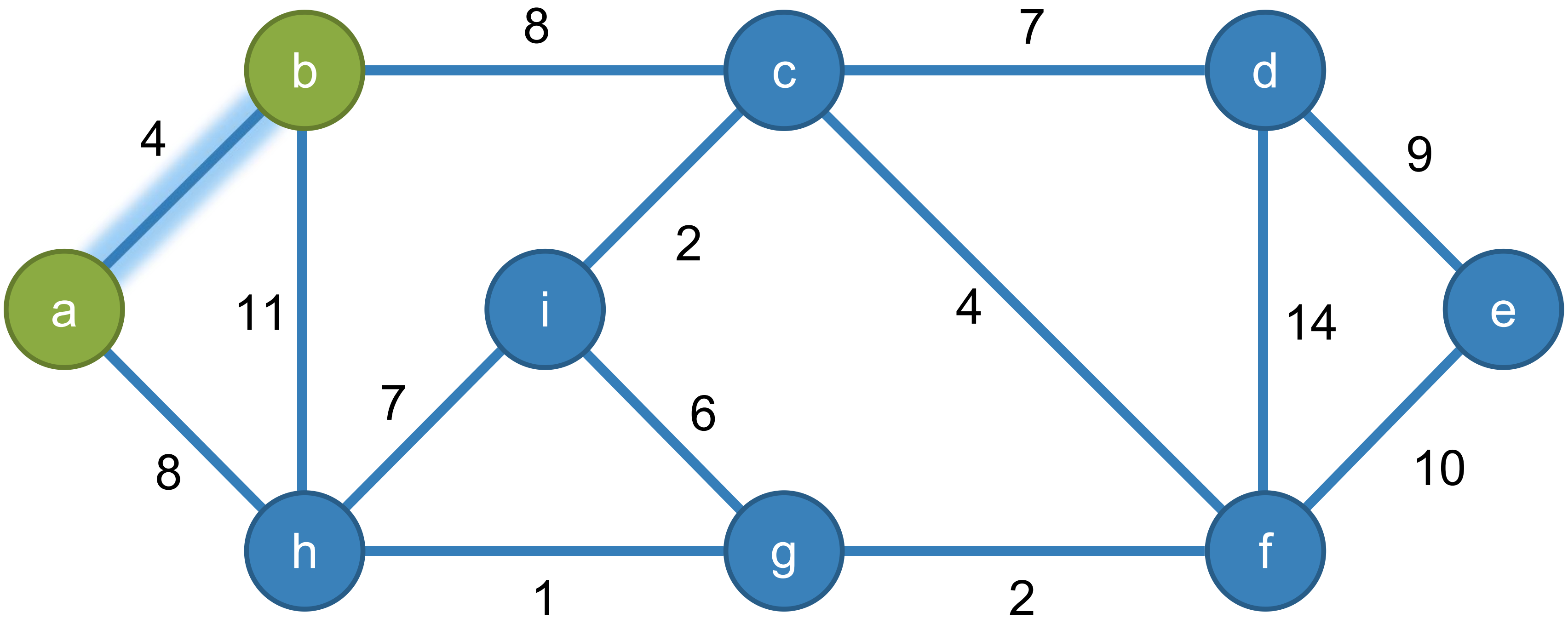
Algoritmo di Borůvka—1926



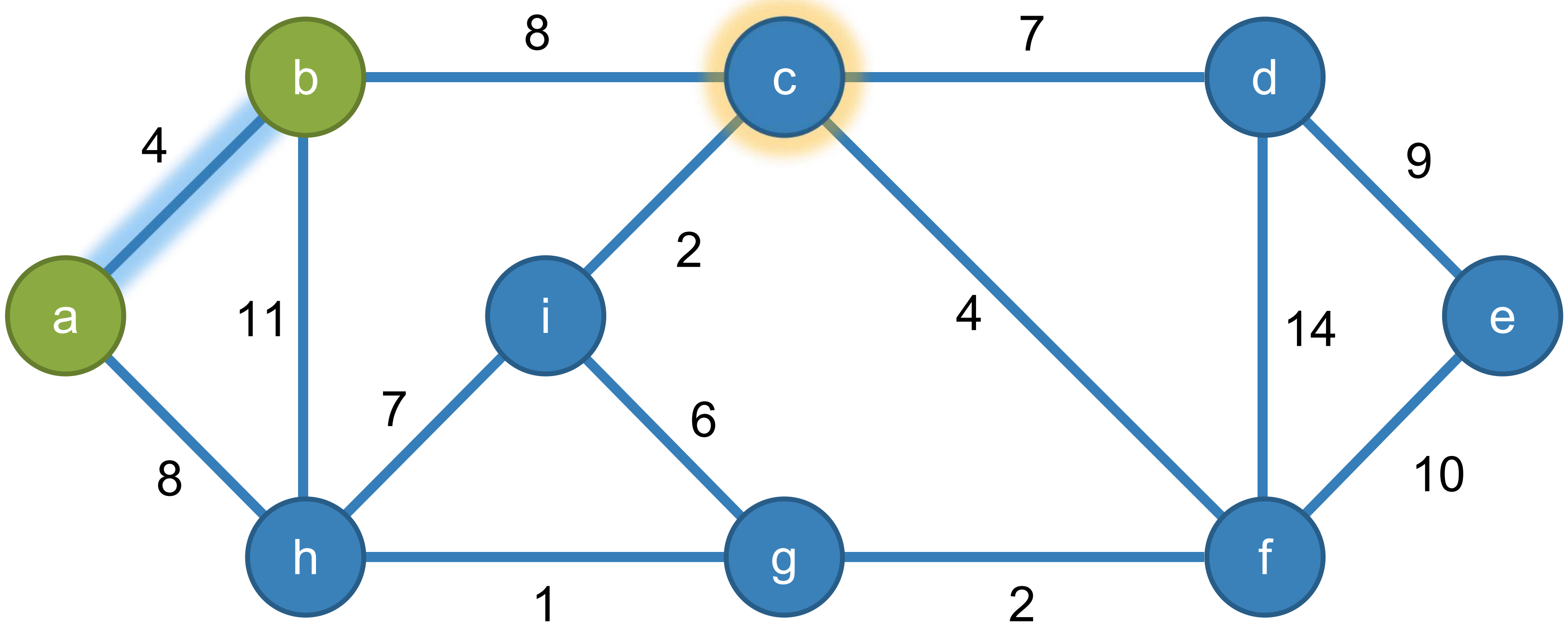
Algoritmo di Borůvka—1926



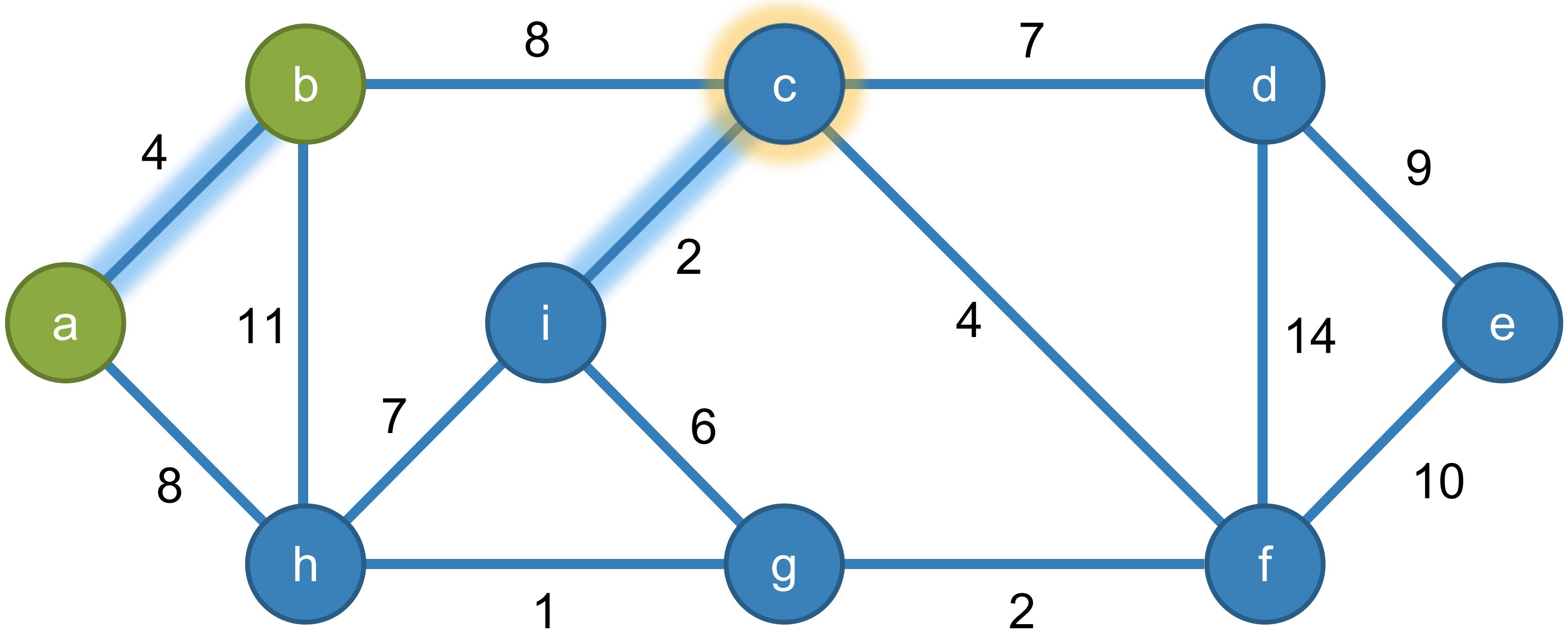
Algoritmo di Borůvka—1926



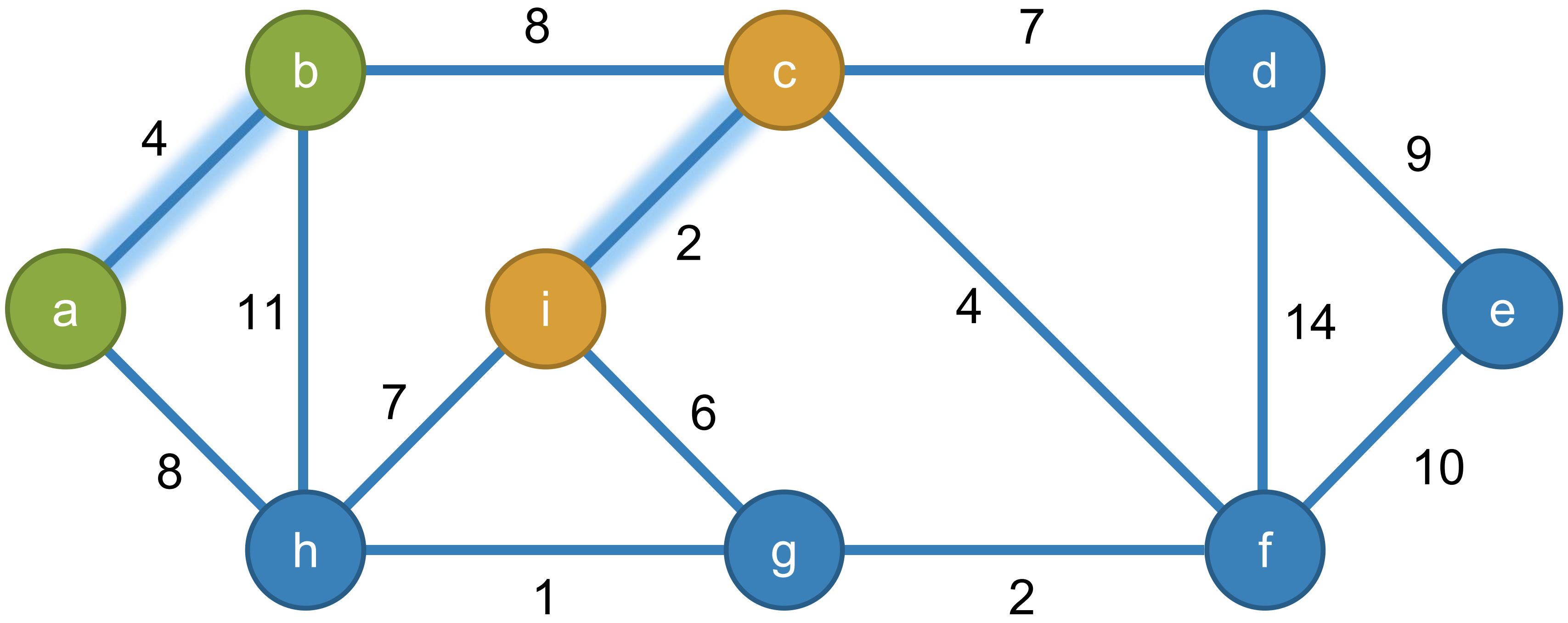
Algoritmo di Borůvka—1926



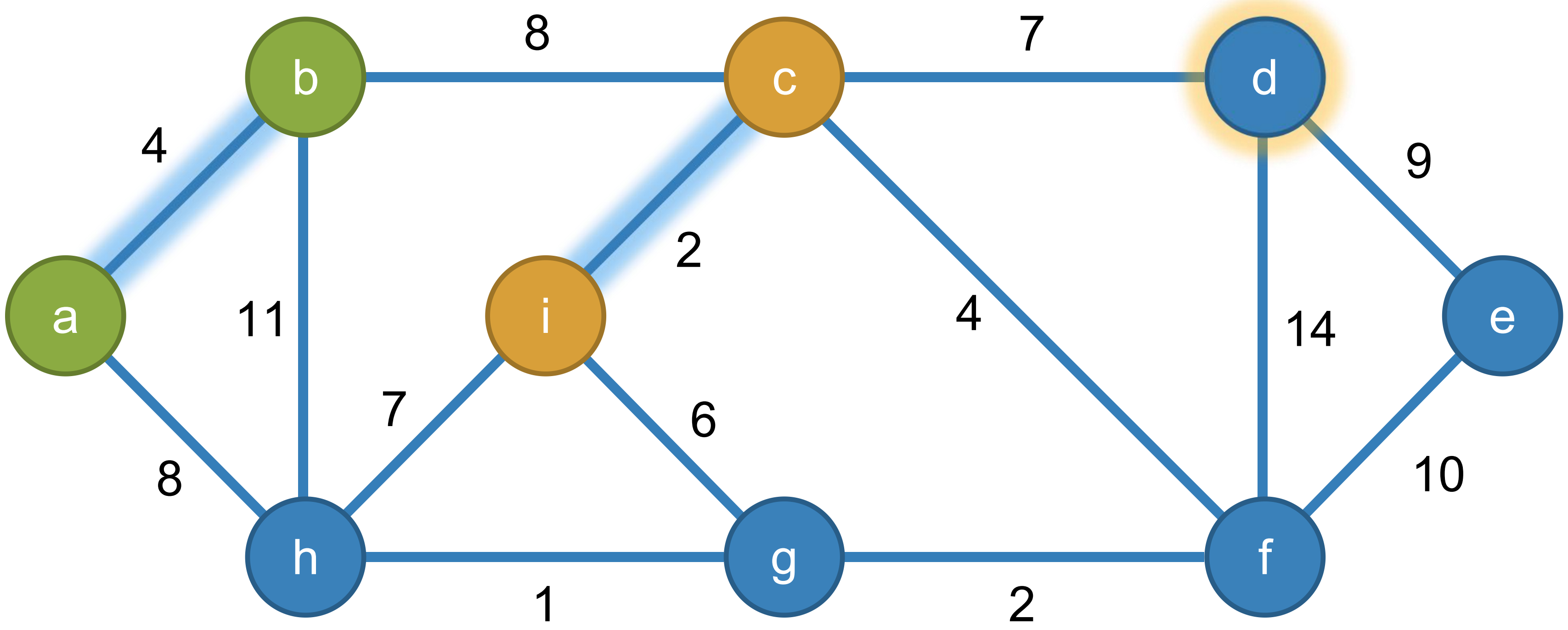
Algoritmo di Borůvka—1926



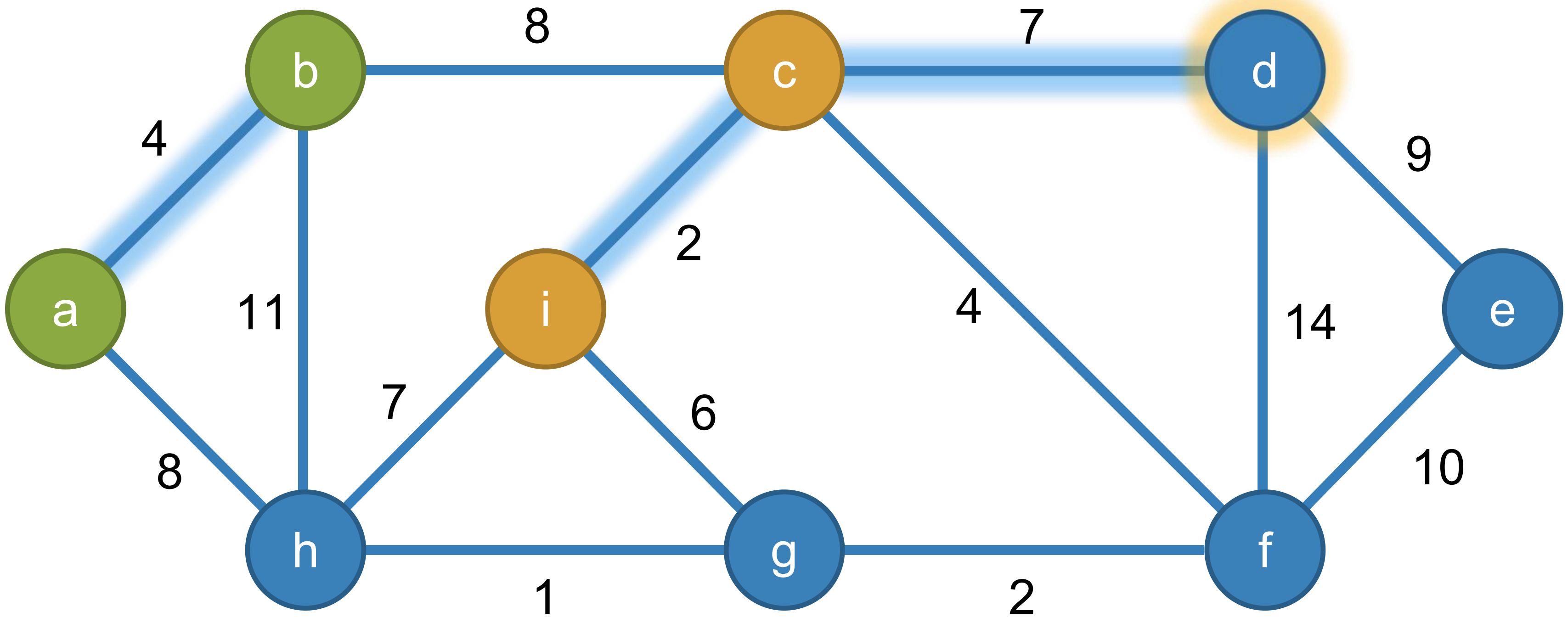
Algoritmo di Borůvka—1926



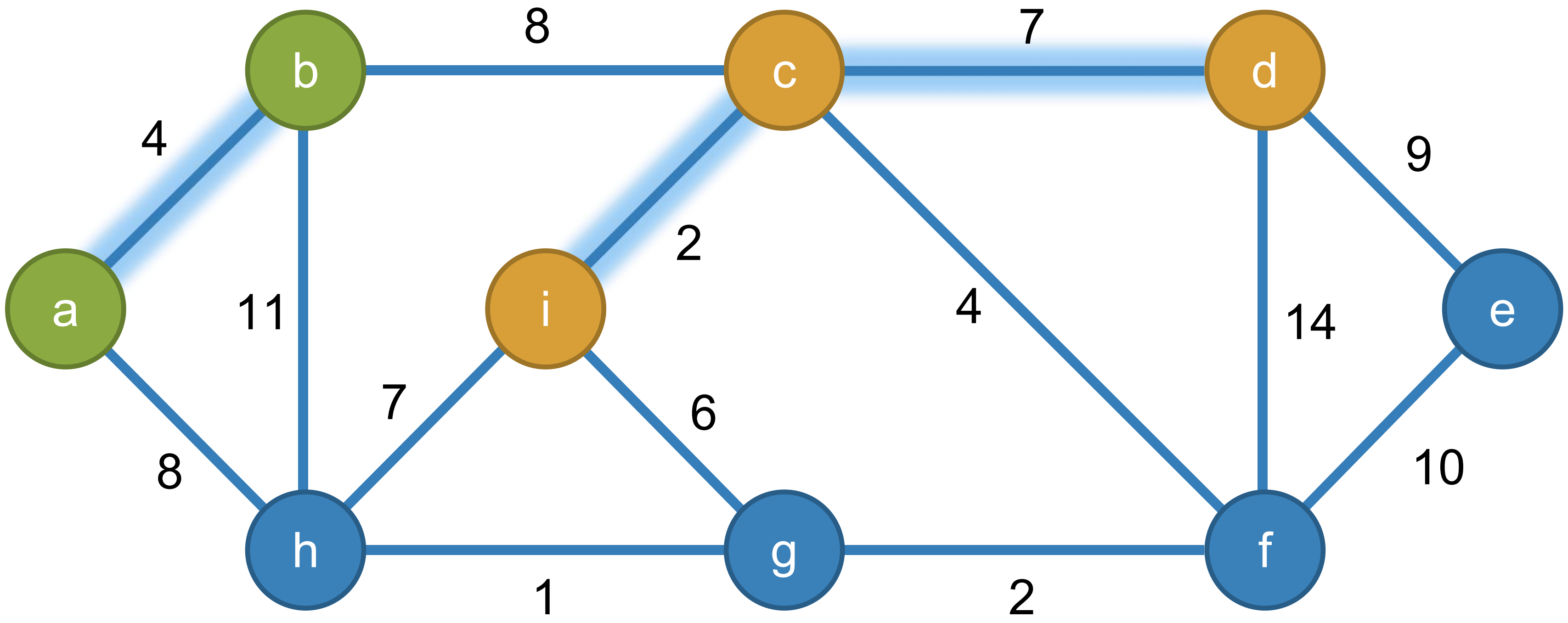
Algoritmo di Borůvka—1926



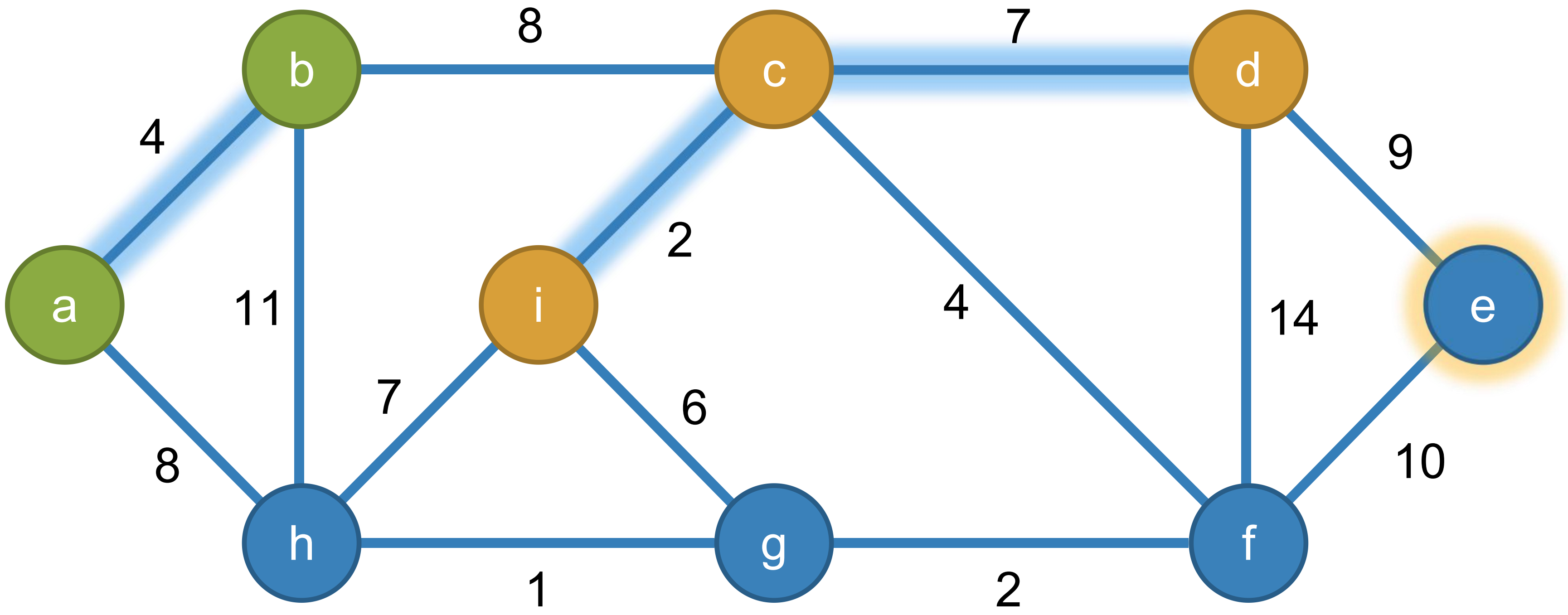
Algoritmo di Borůvka—1926



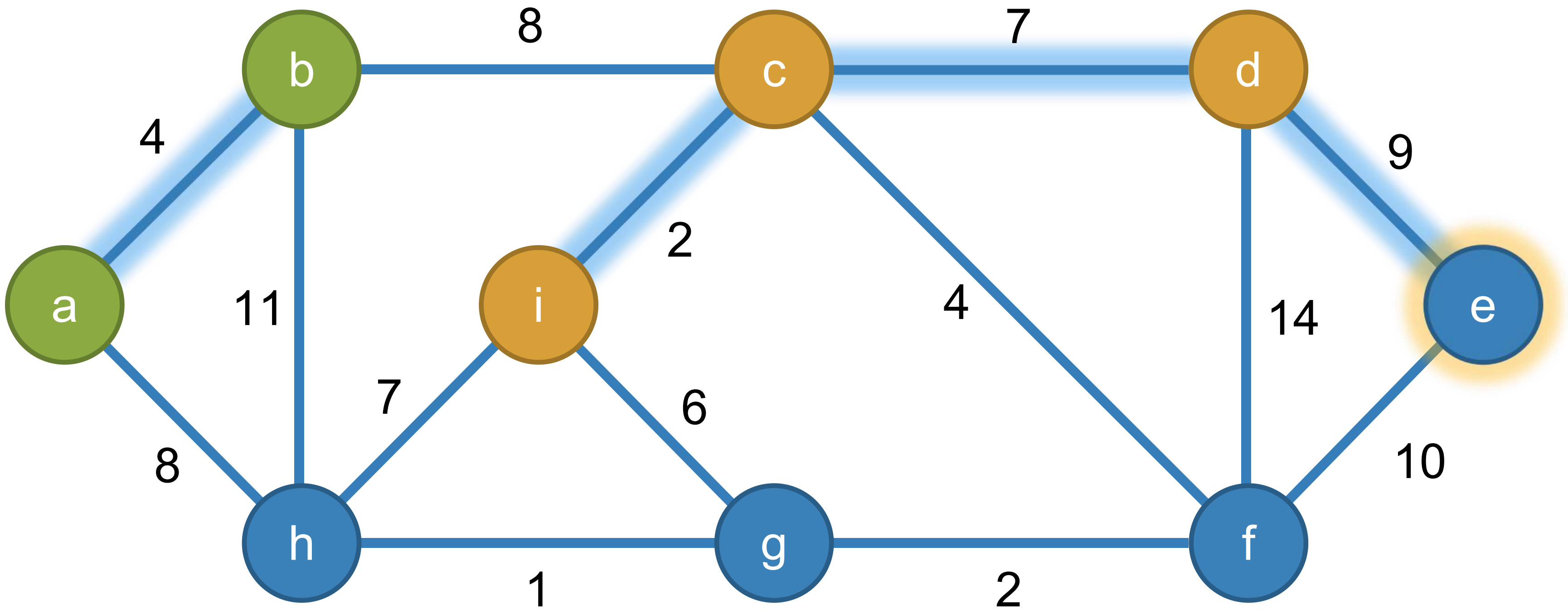
Algoritmo di Borůvka—1926



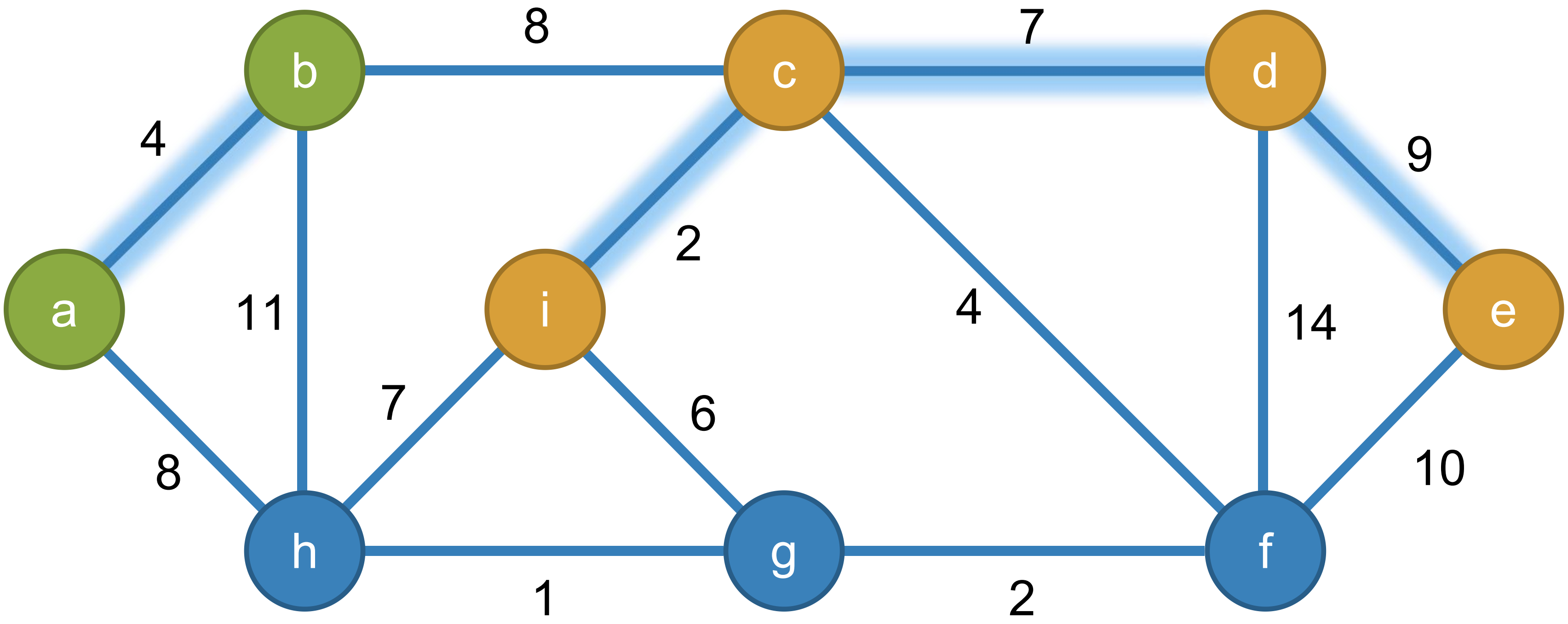
Algoritmo di Borůvka—1926



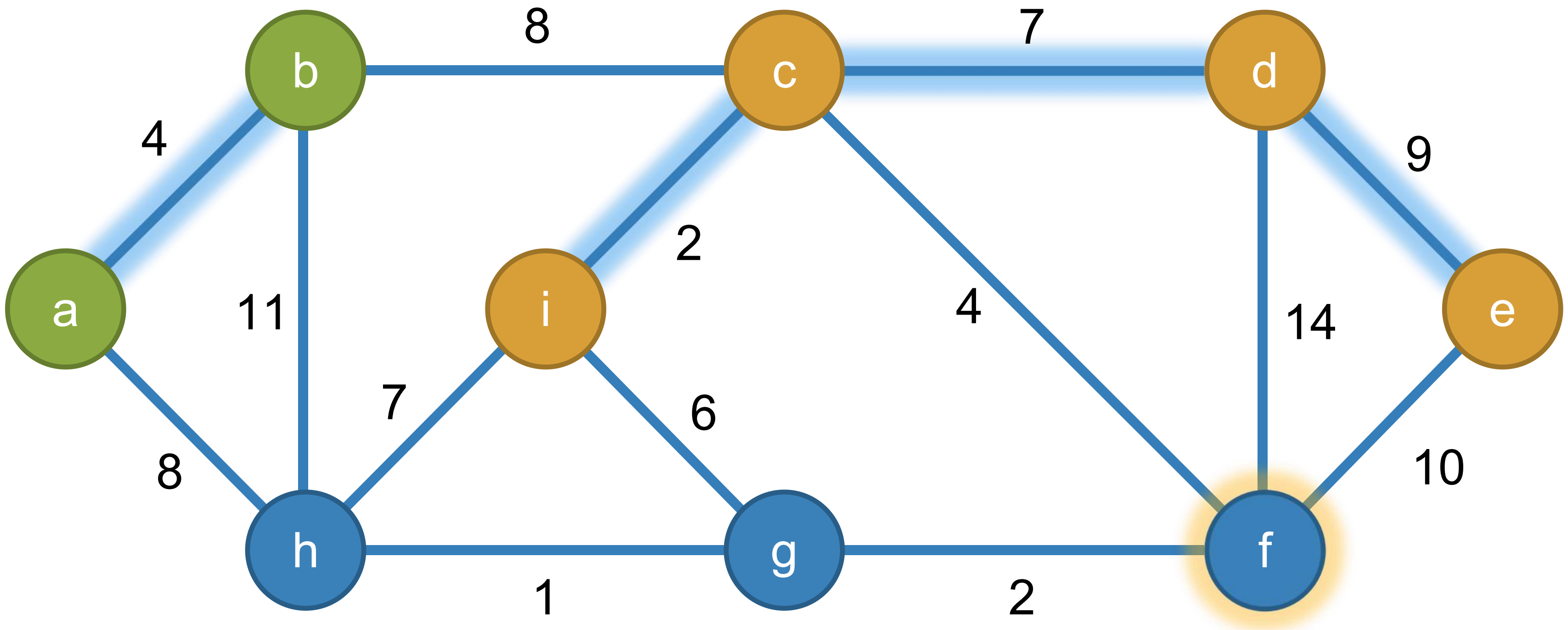
Algoritmo di Borůvka—1926



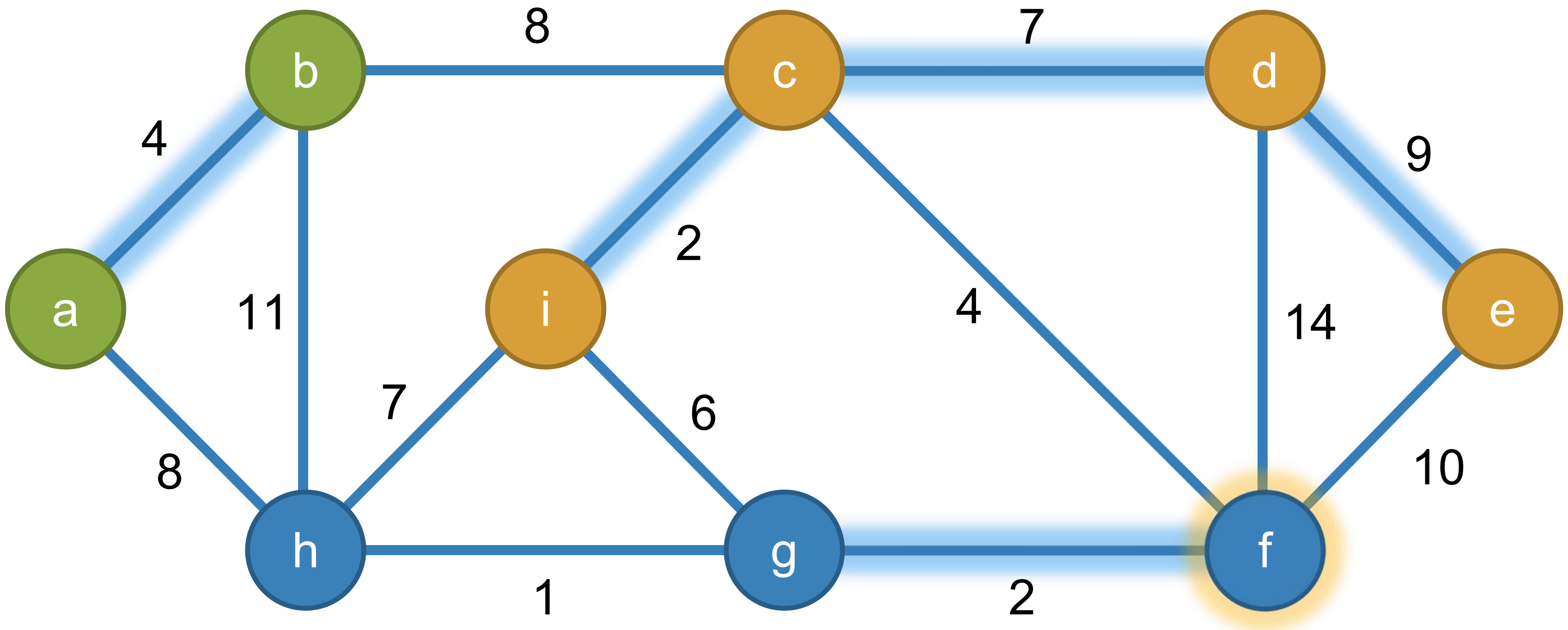
Algoritmo di Borůvka—1926



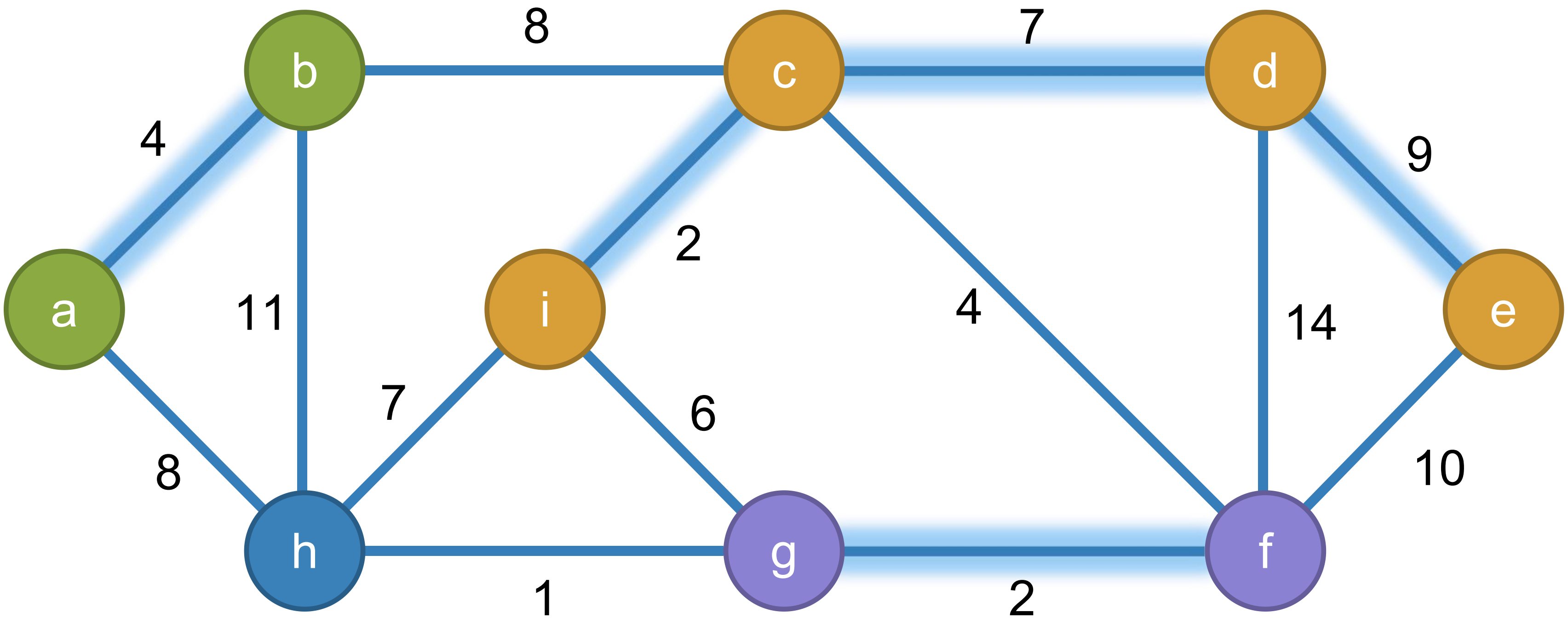
Algoritmo di Borůvka—1926



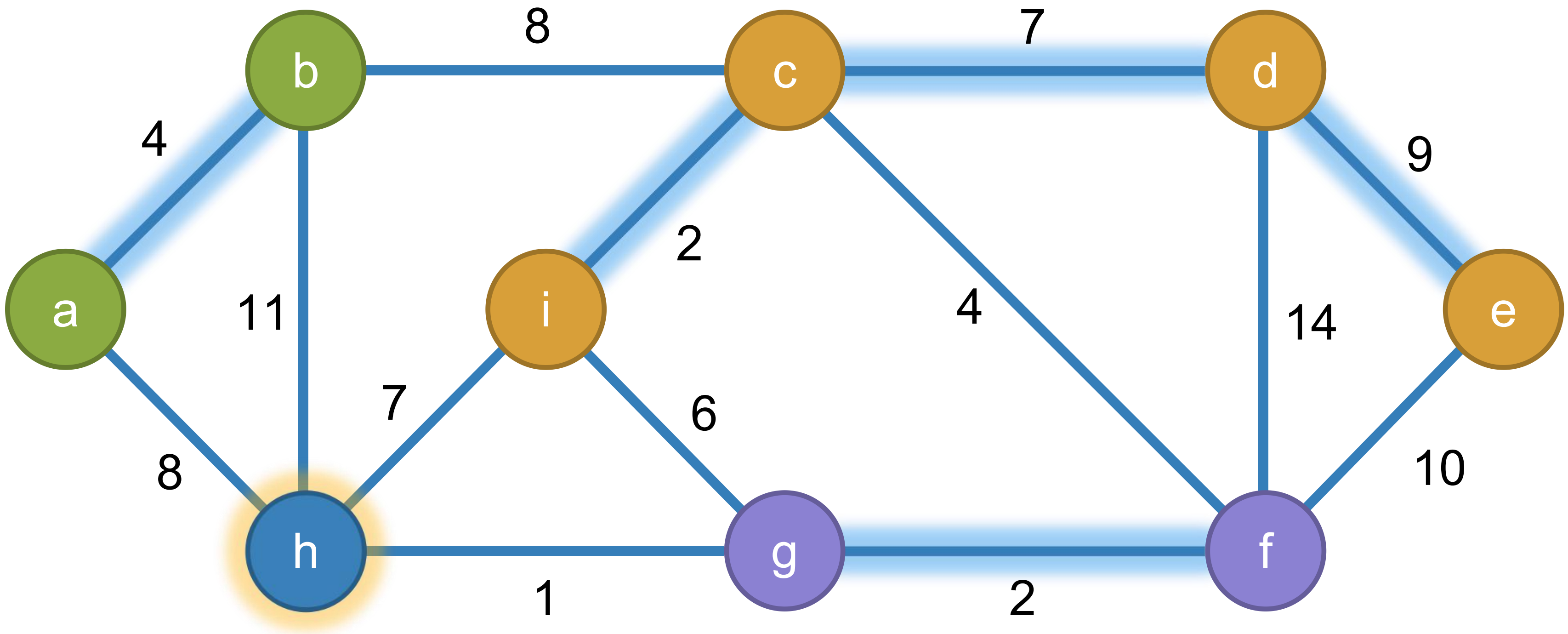
Algoritmo di Borůvka—1926



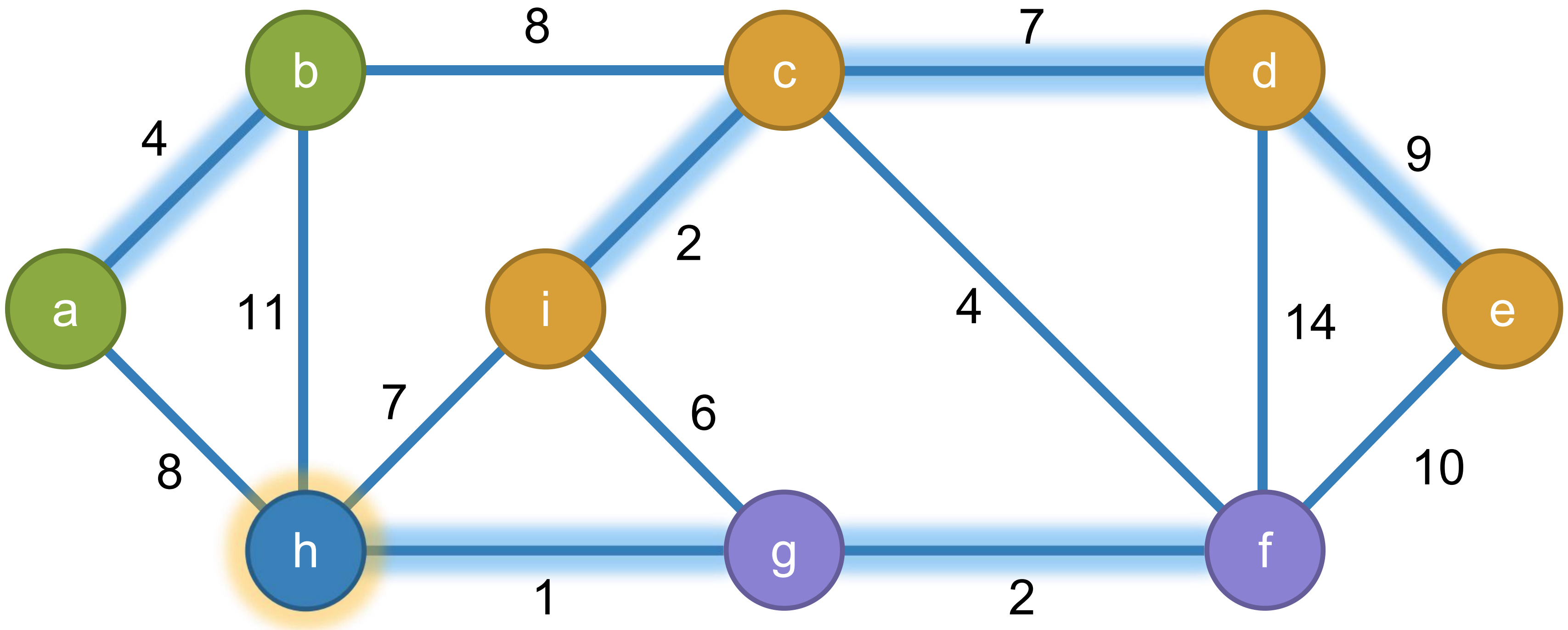
Algoritmo di Borůvka—1926



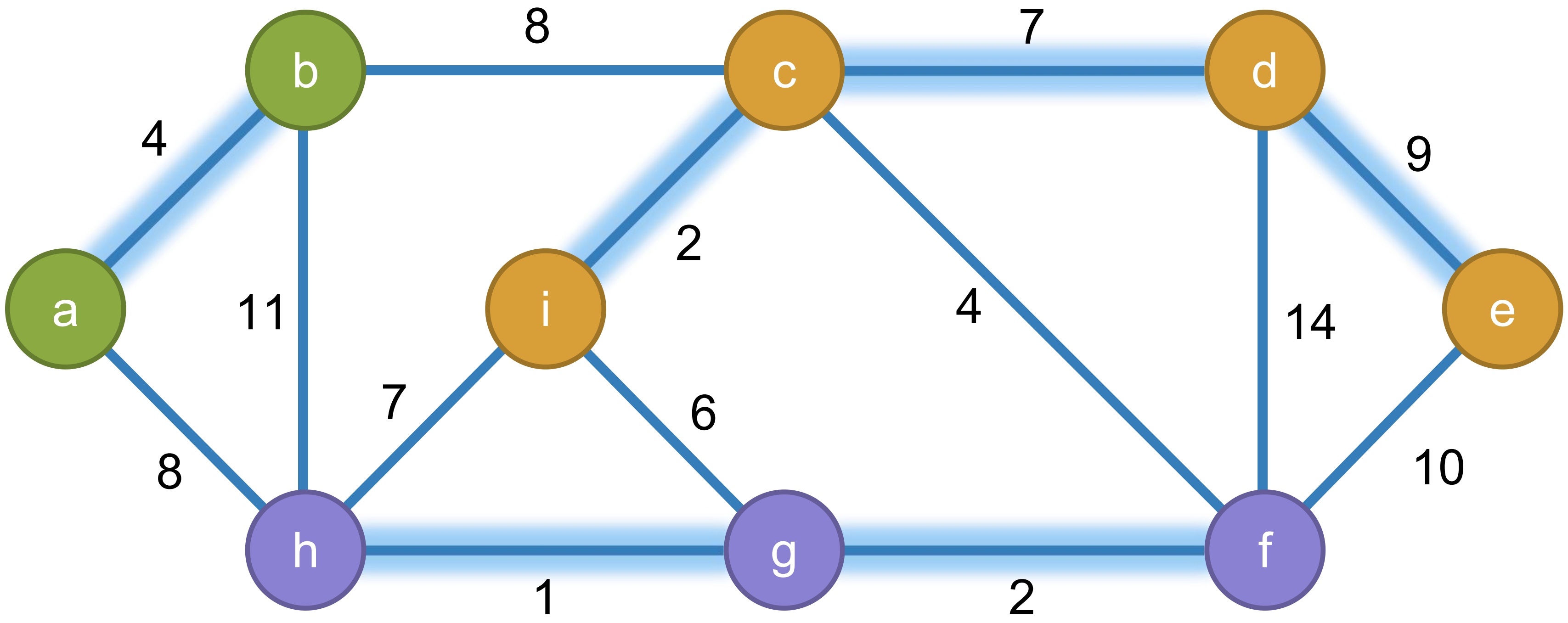
Algoritmo di Borůvka—1926



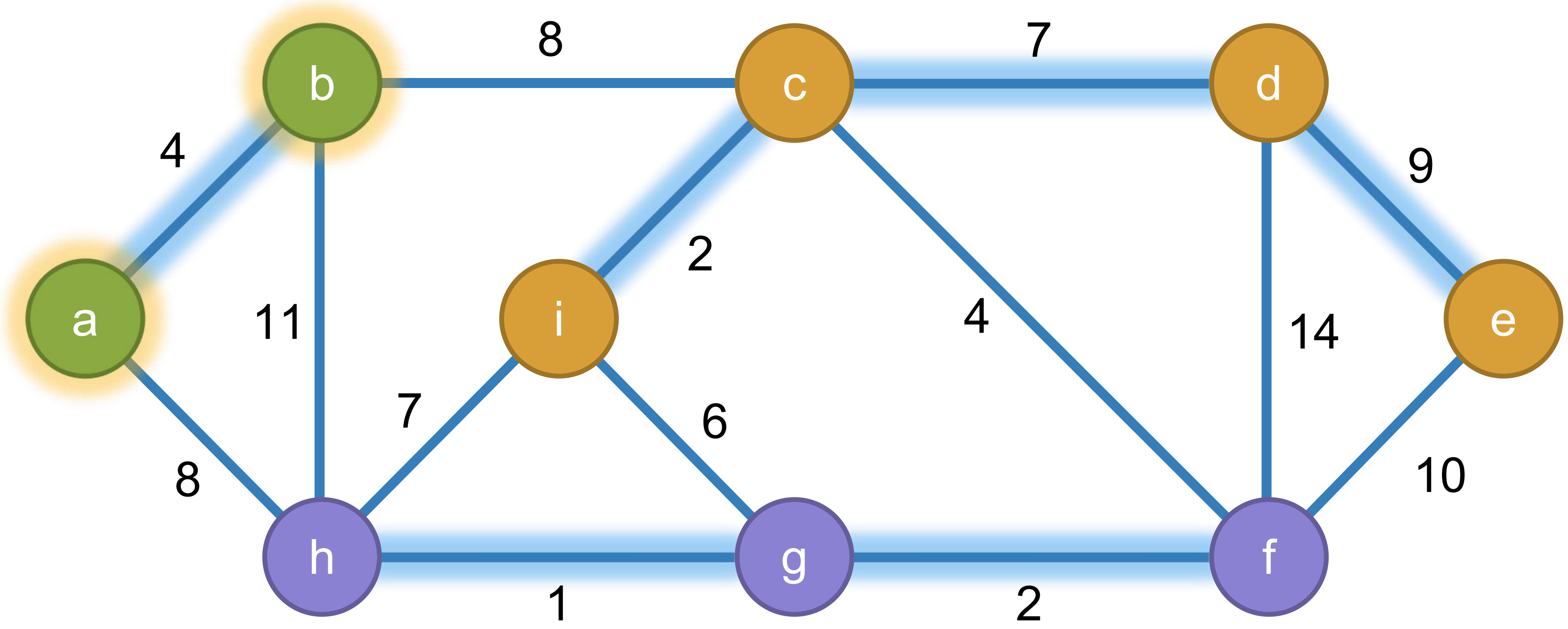
Algoritmo di Borůvka—1926



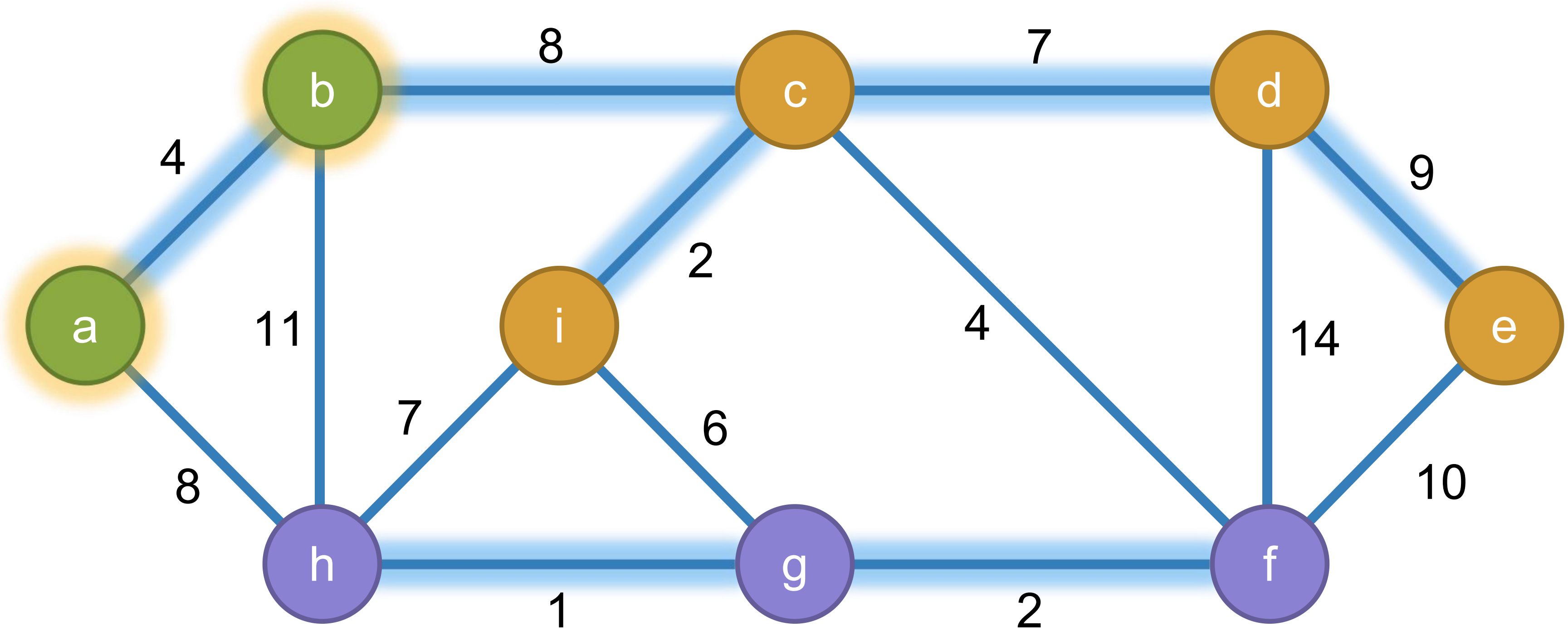
Algoritmo di Borůvka—1926



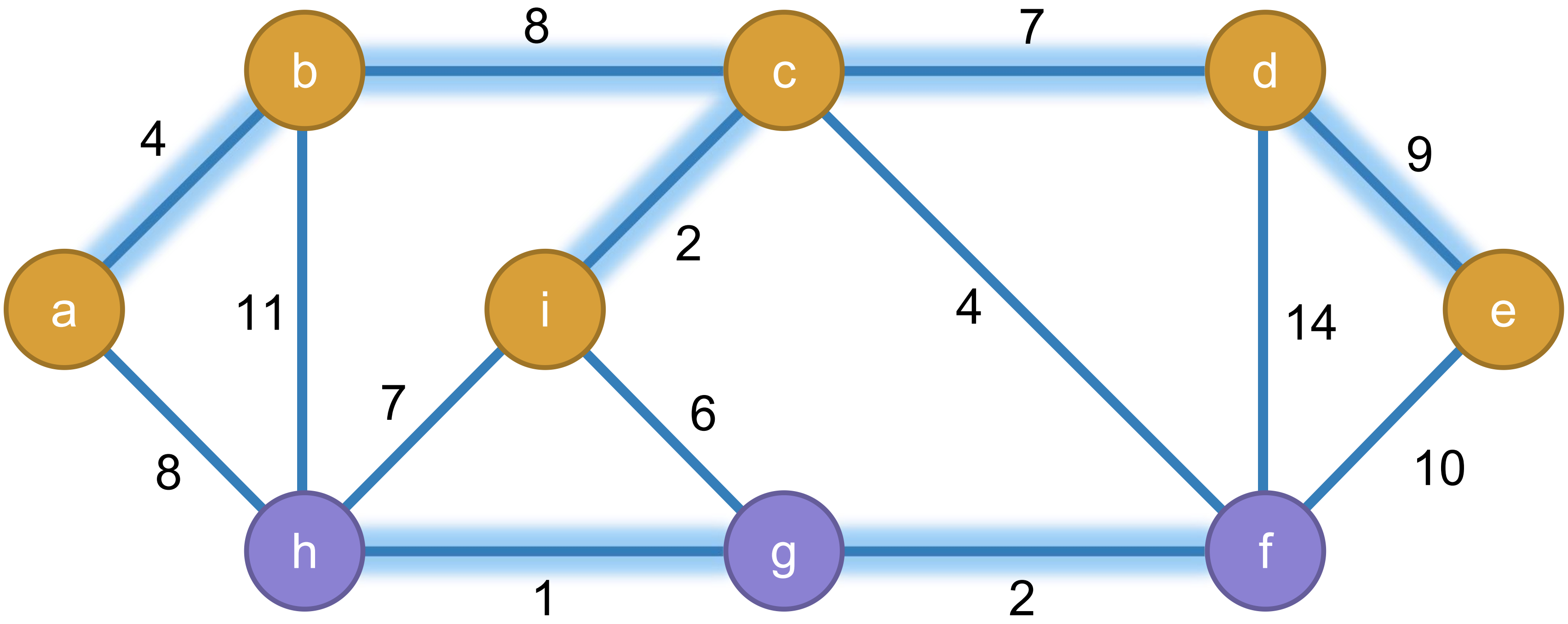
Algoritmo di Borůvka—1926



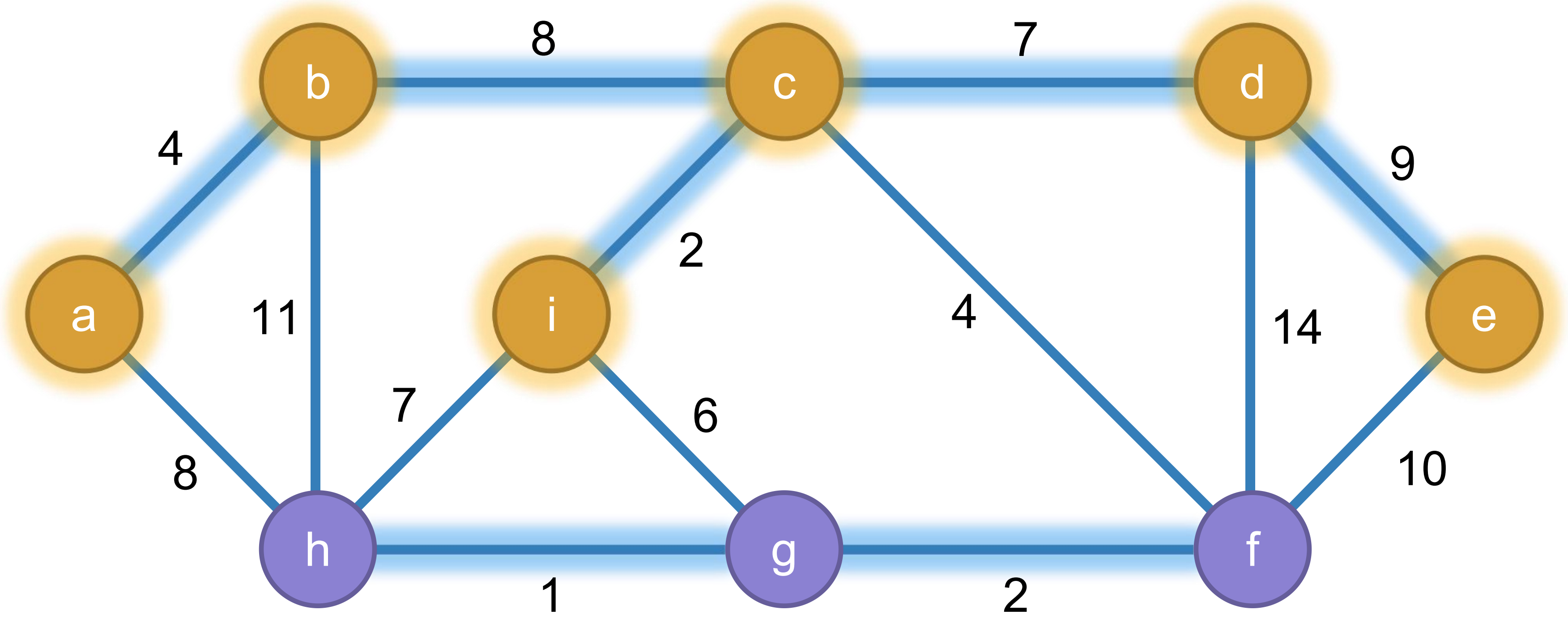
Algoritmo di Borůvka—1926



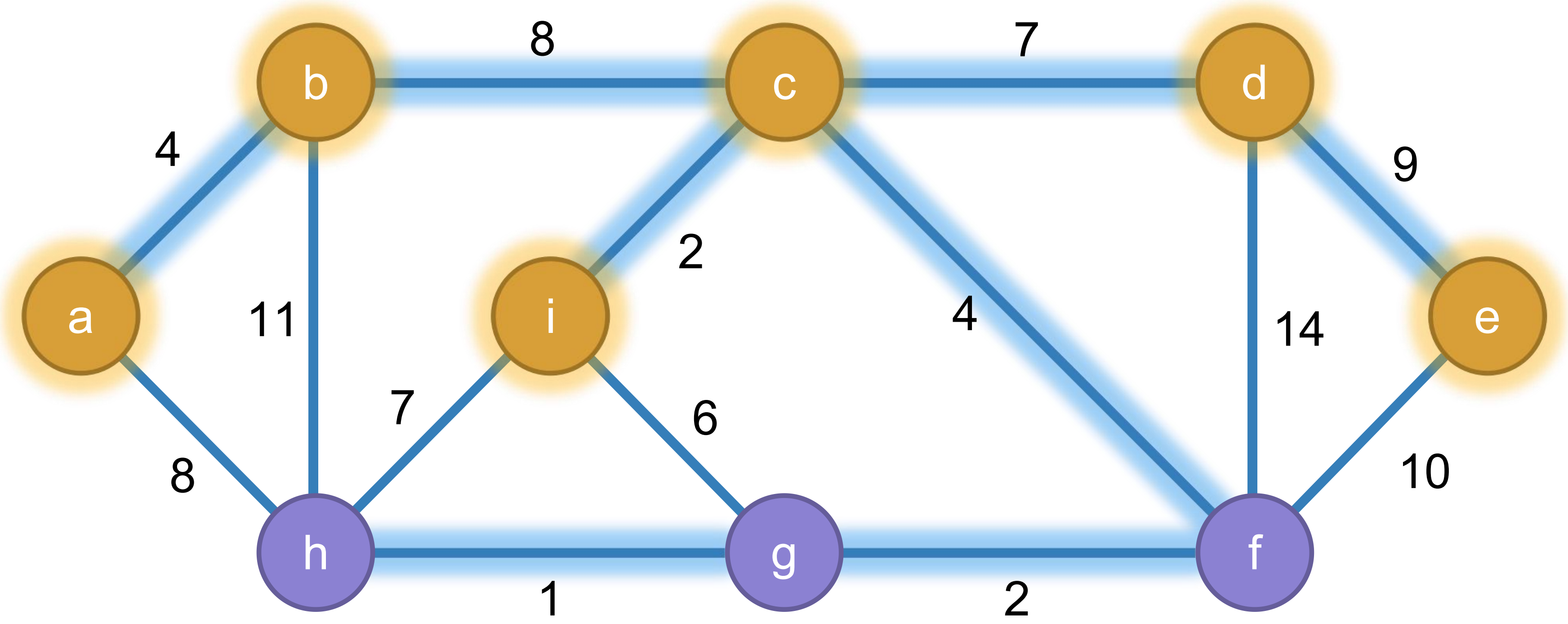
Algoritmo di Borůvka—1926



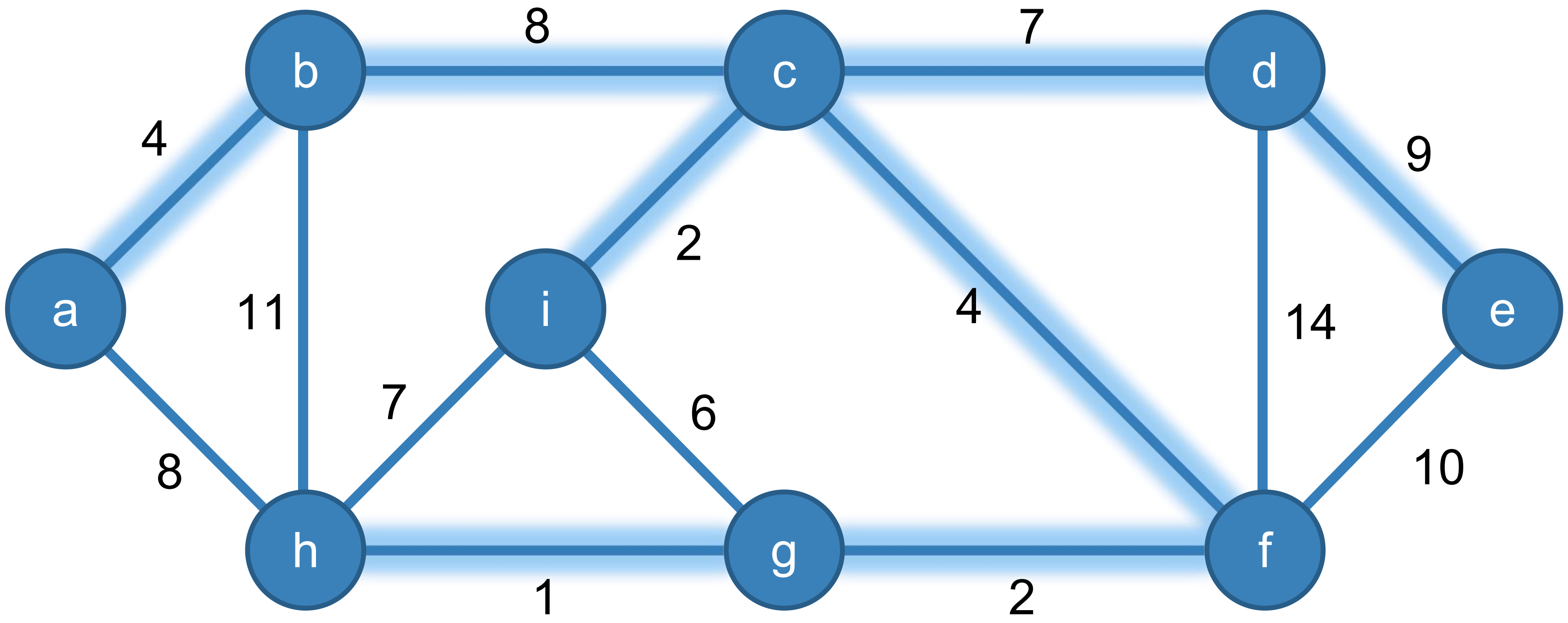
Algoritmo di Borůvka—1926



Algoritmo di Borůvka—1926



Algoritmo di Borůvka—1926

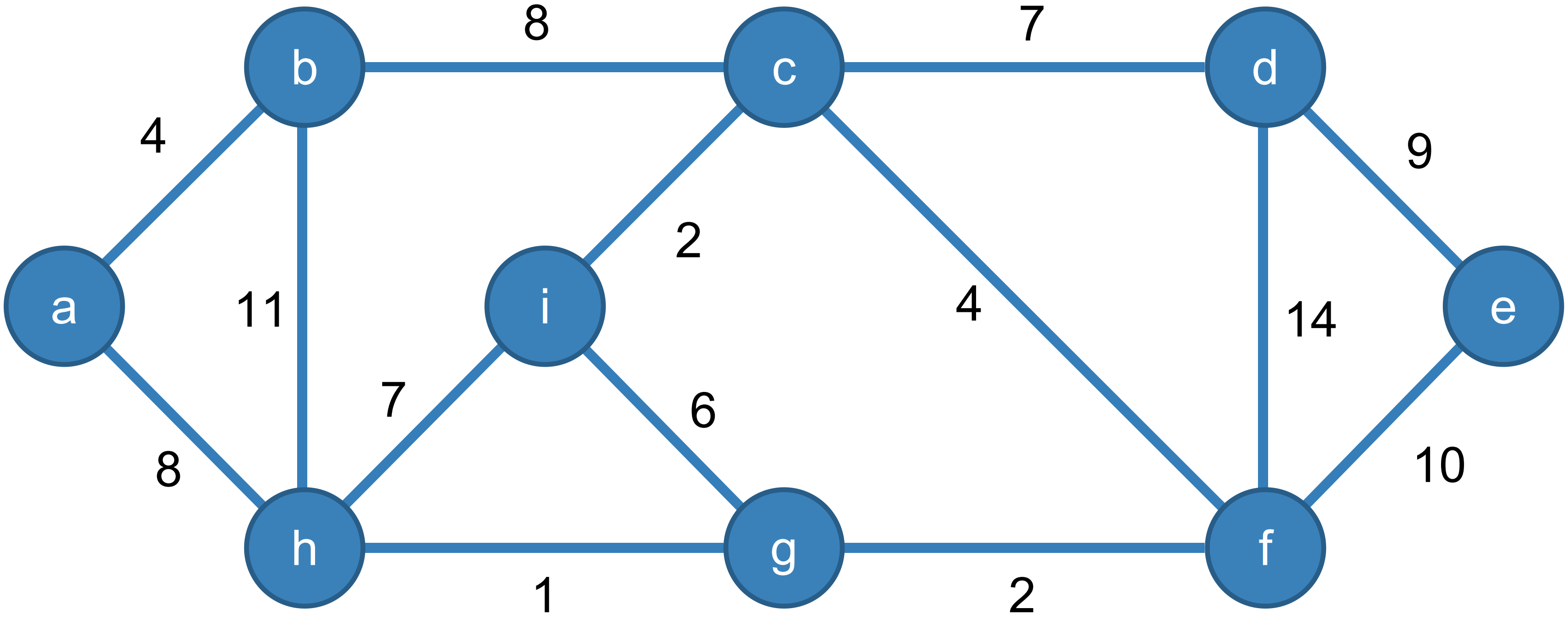


Algoritmo di Kruskal—1956

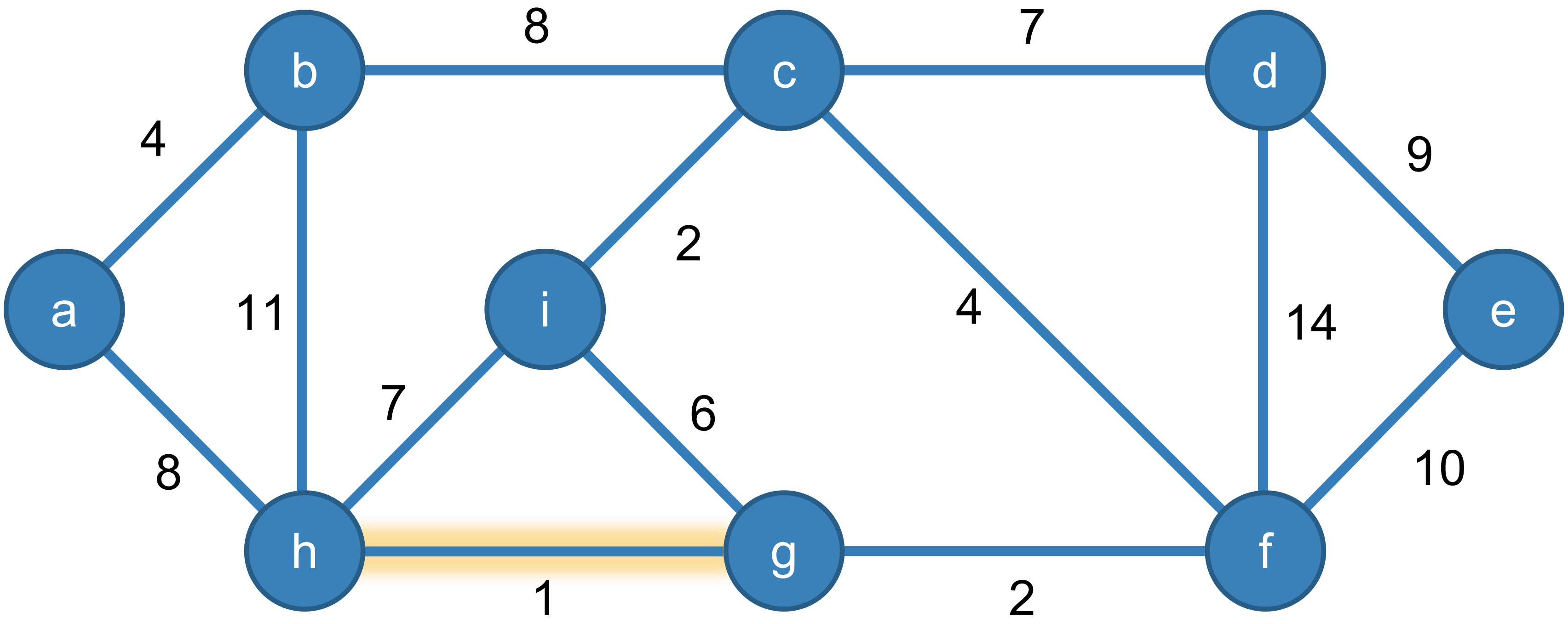
- Opera in maniera simile a Borůvska, ma sfruttando strutture dati differenti
- Richiede l'ordinamento degli archi per peso
- Complessità: O(E log V)

```
KRUSKAL(G):
  A ← Ø
  foreach v in G.V:
    MAKESET(v)
  foreach (u, v) in G.E ordinati per w(u, v) crescente:
    if FINDSET(u) ≠ FINDSET(v) then:
      A ← A ∪ {(u, v)}
      UNION(FINDSET(u), FINDSET(v))
  return A
```

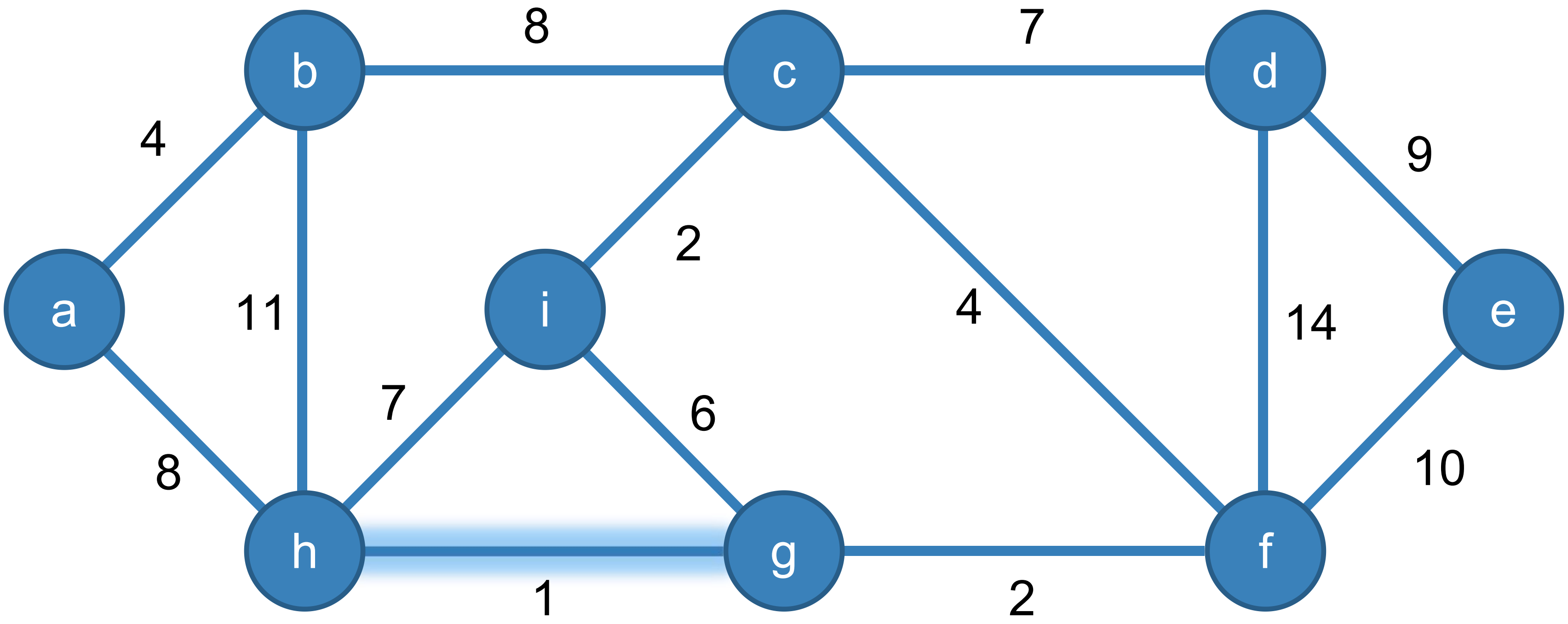
Algoritmo di Kruskal—1956



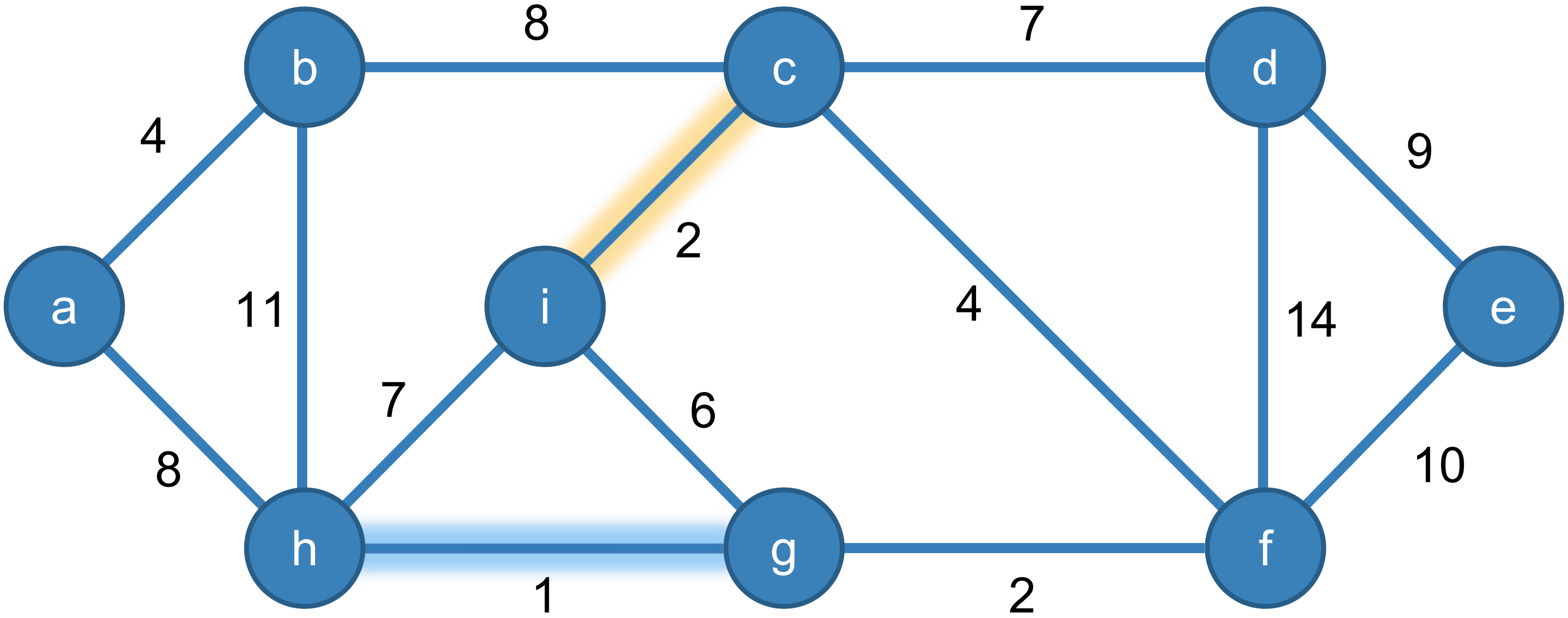
Algoritmo di Kruskal—1956



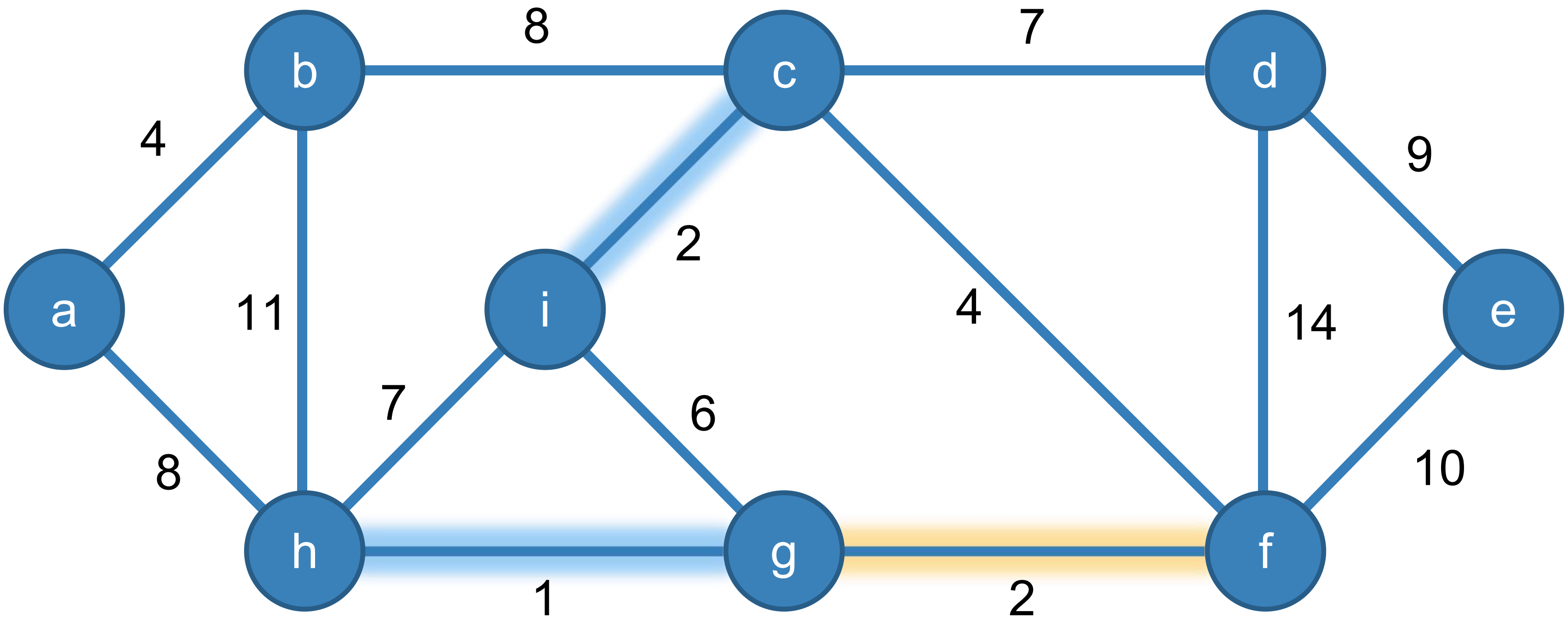
Algoritmo di Kruskal—1956



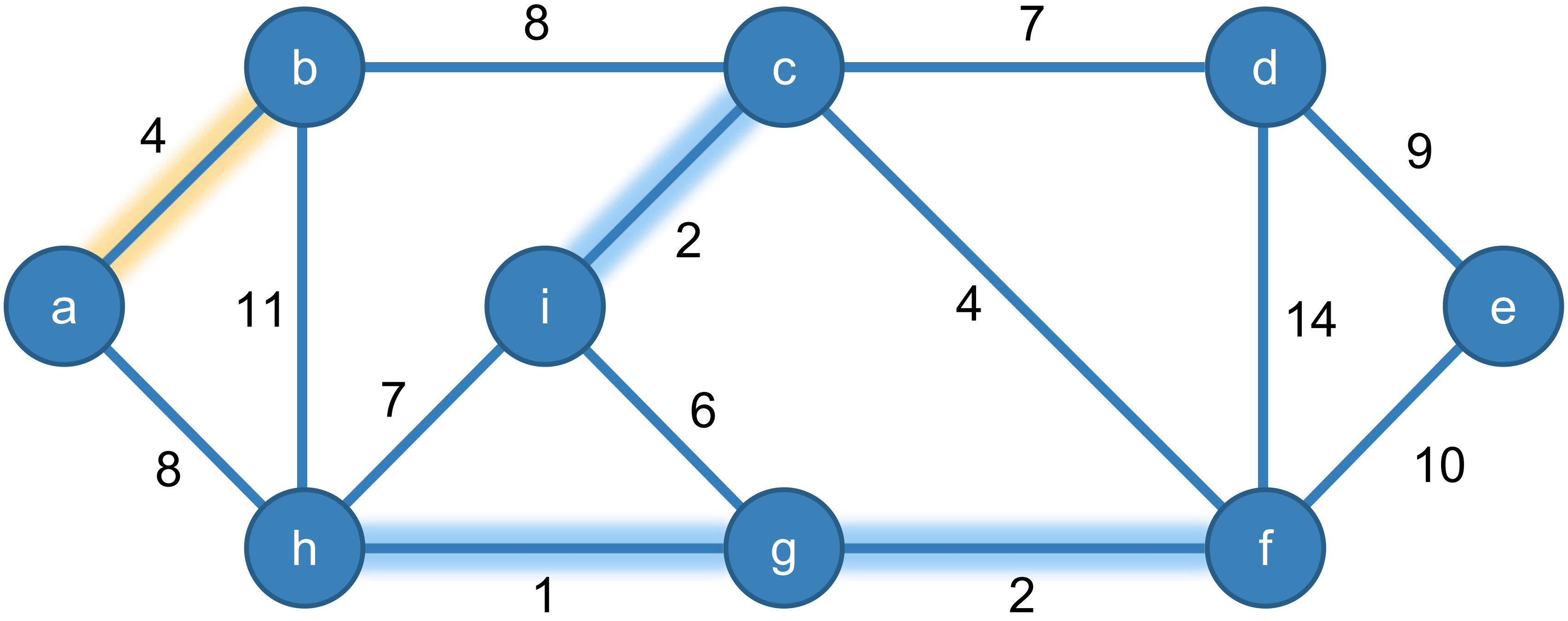
Algoritmo di Kruskal—1956



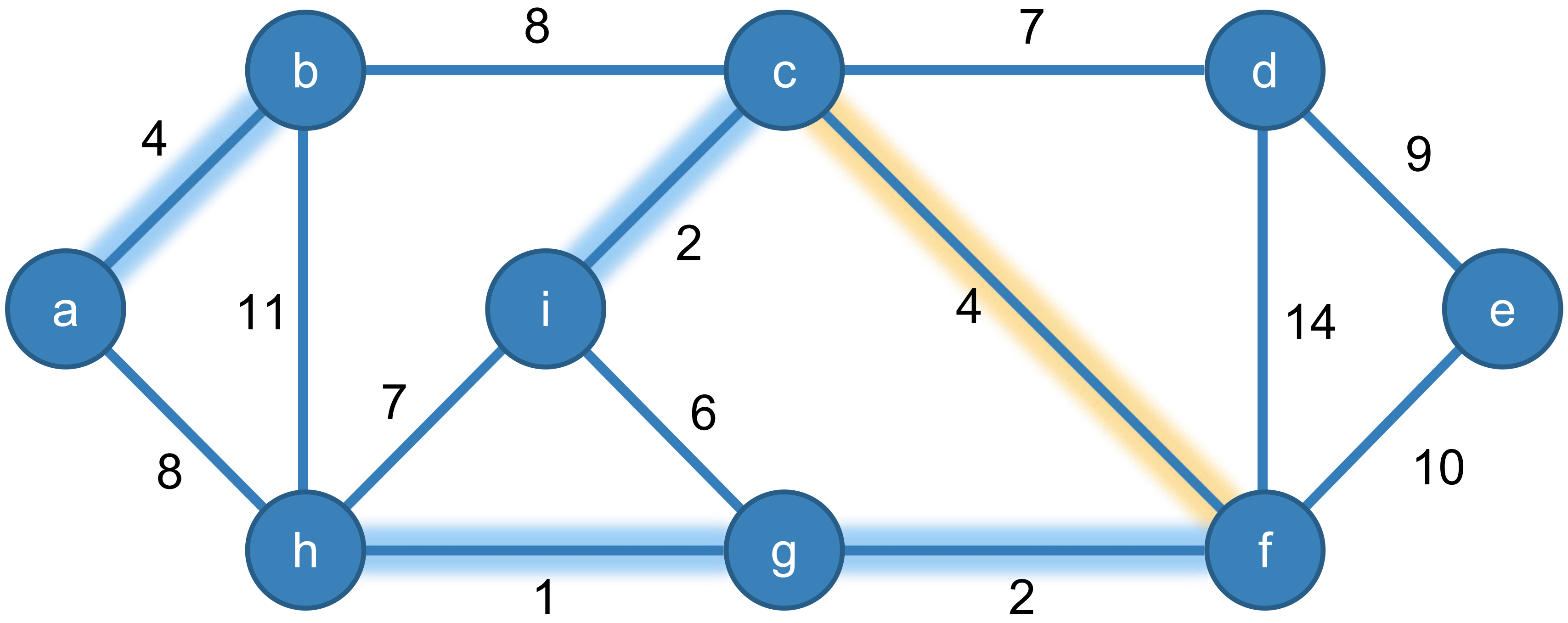
Algoritmo di Kruskal—1956



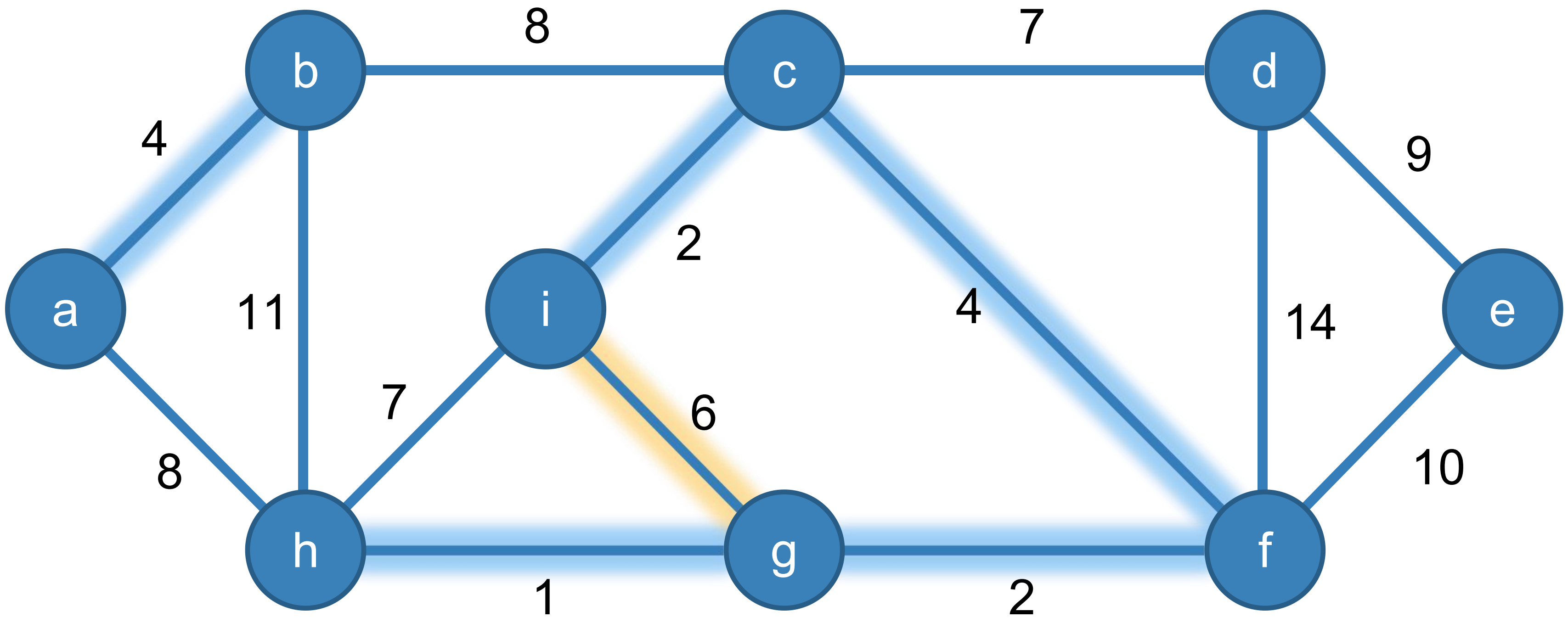
Algoritmo di Kruskal—1956



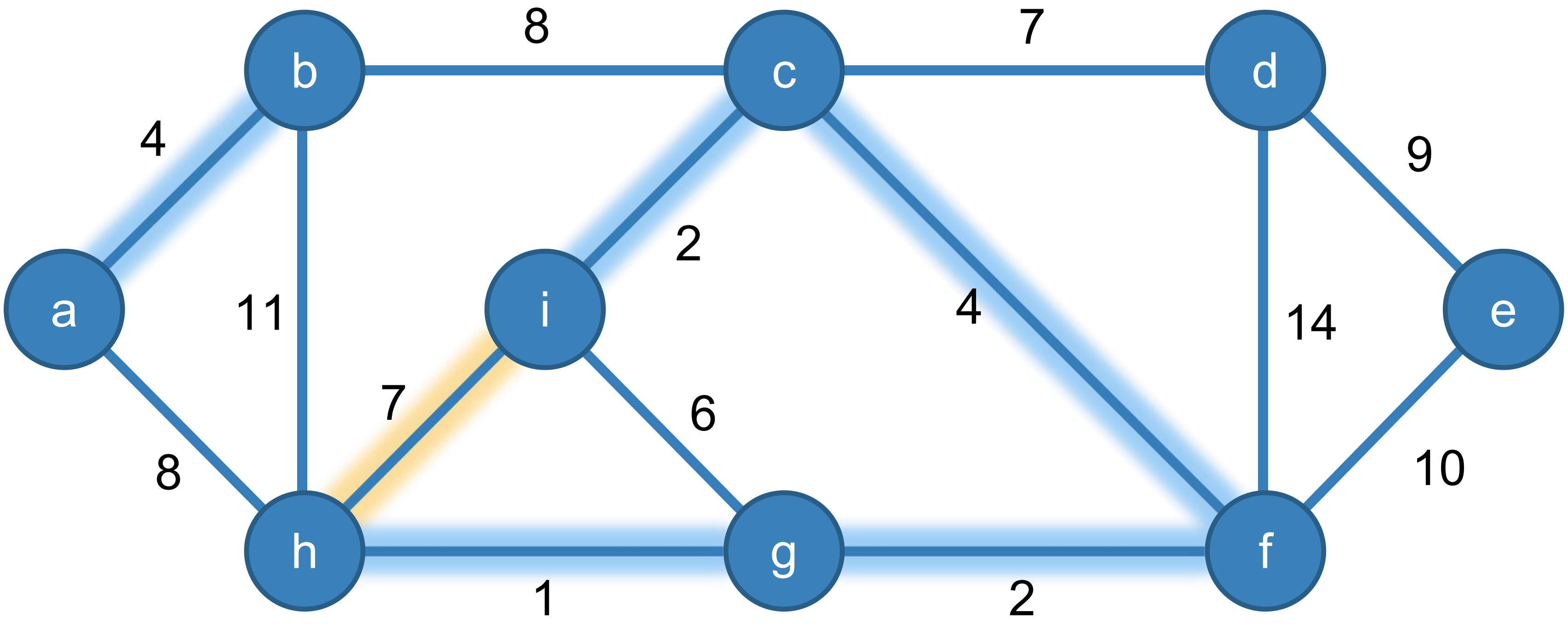
Algoritmo di Kruskal—1956



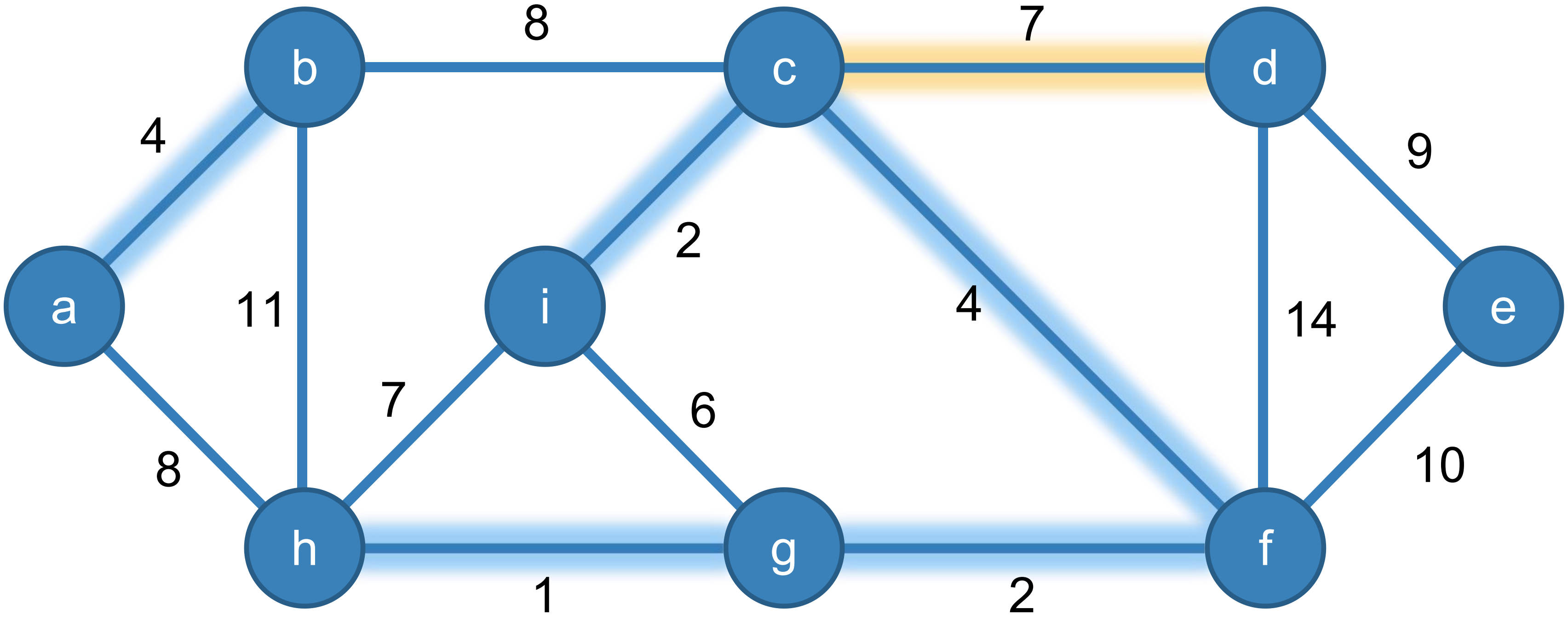
Algoritmo di Kruskal—1956



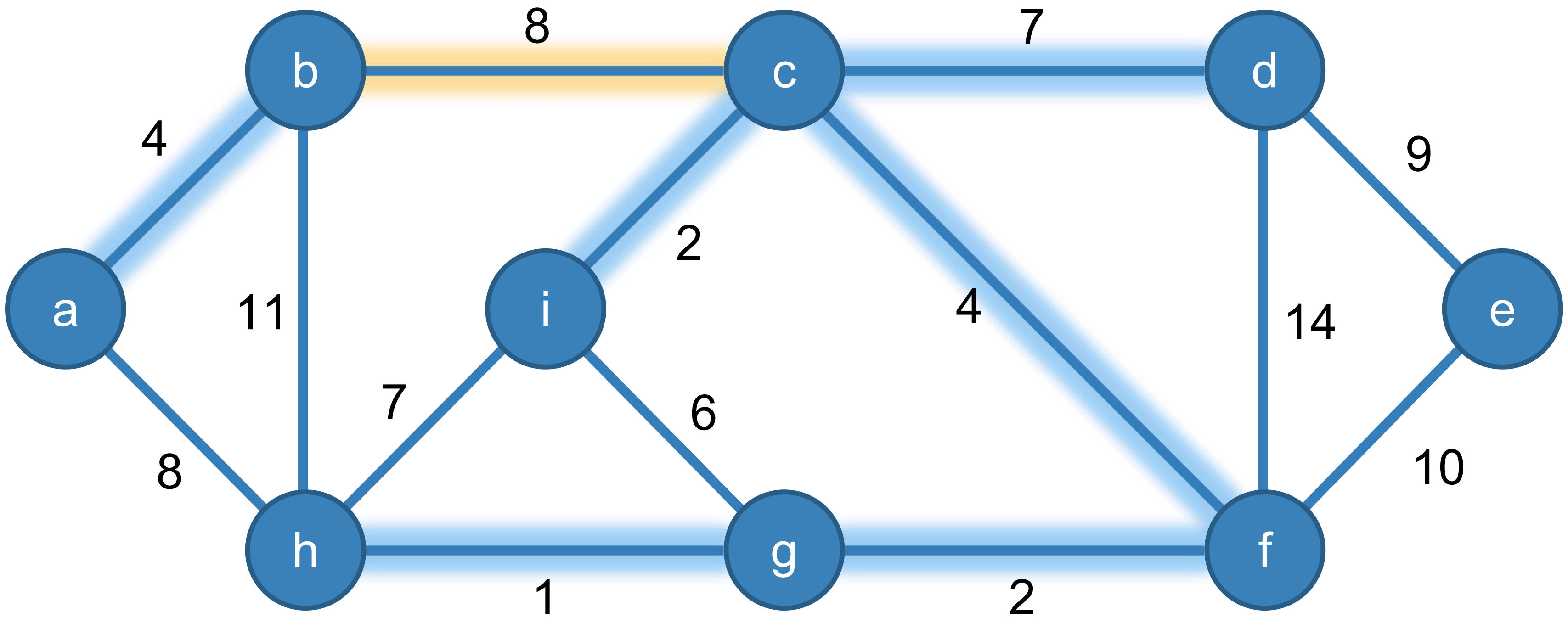
Algoritmo di Kruskal—1956



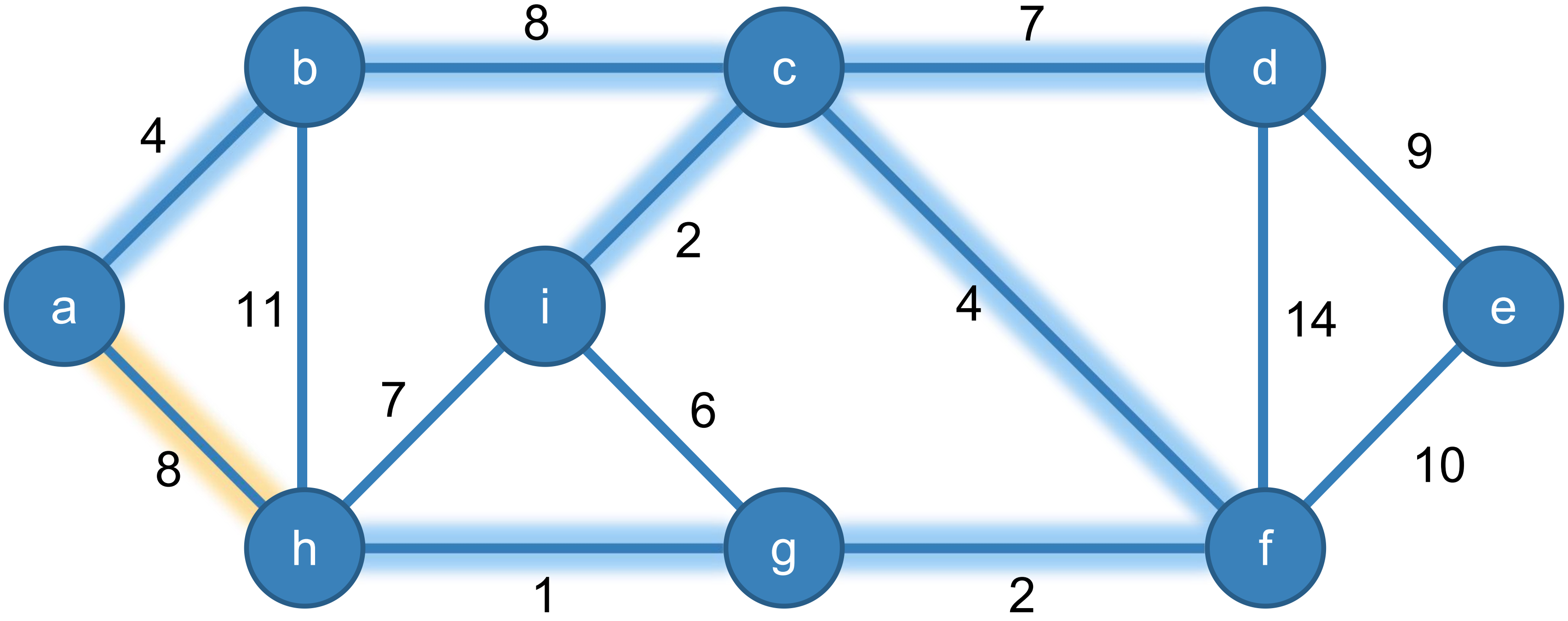
Algoritmo di Kruskal—1956



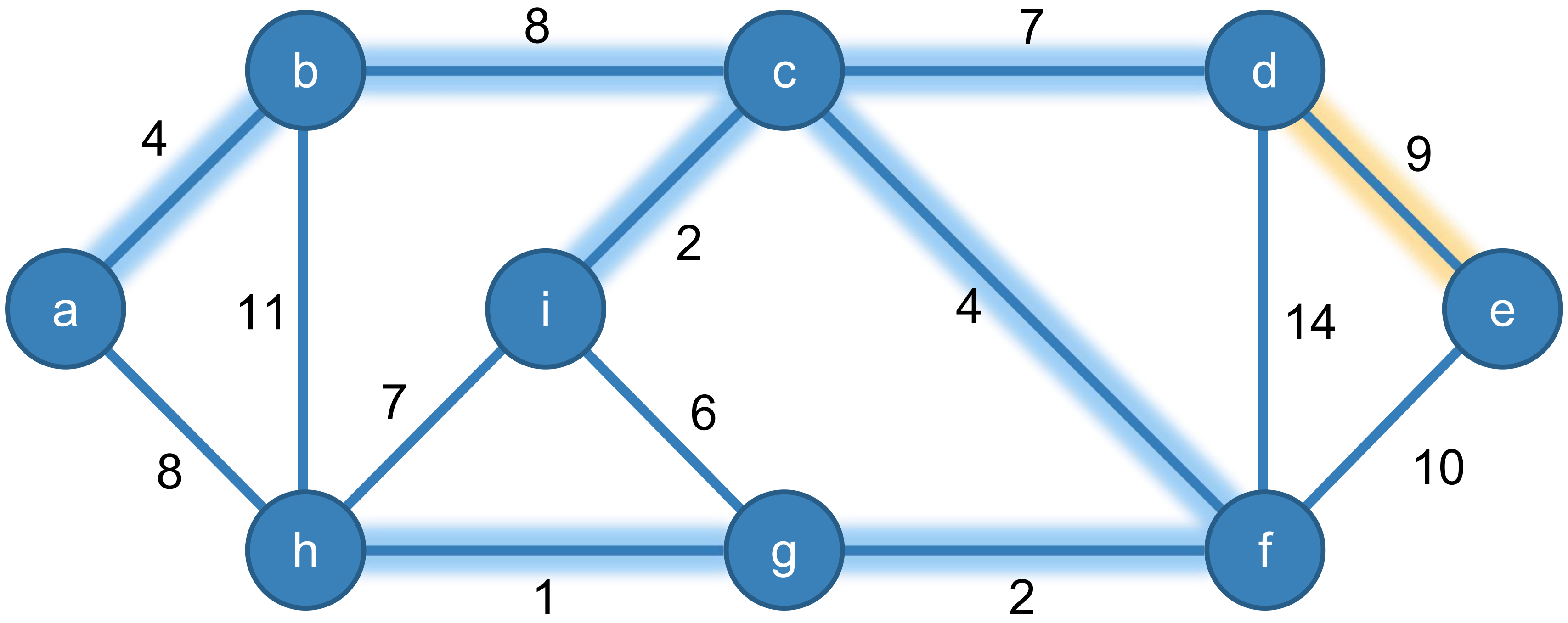
Algoritmo di Kruskal—1956



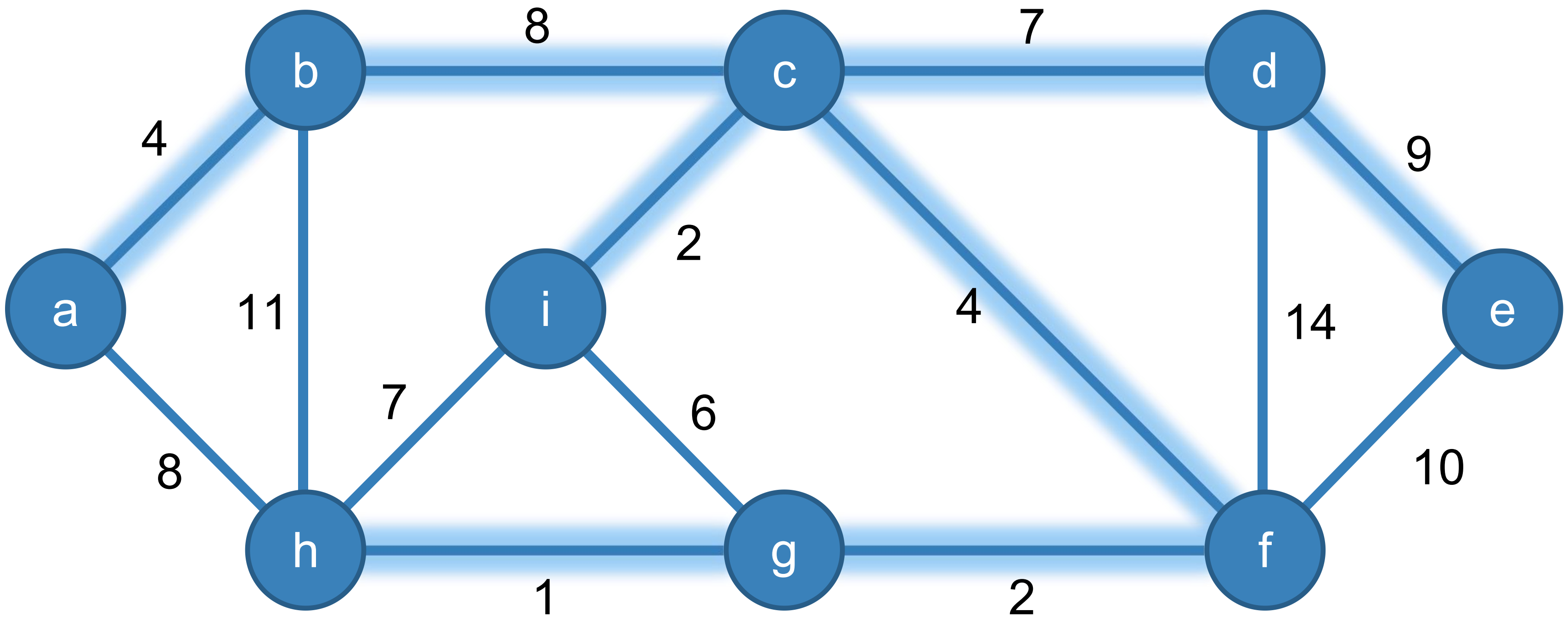
Algoritmo di Kruskal—1956



Algoritmo di Kruskal—1956



Algoritmo di Kruskal—1956



Algoritmo di Prim—1957

- Scoperto indipendentemente anche da Jarnik (1930) e Dijkstra (1959)

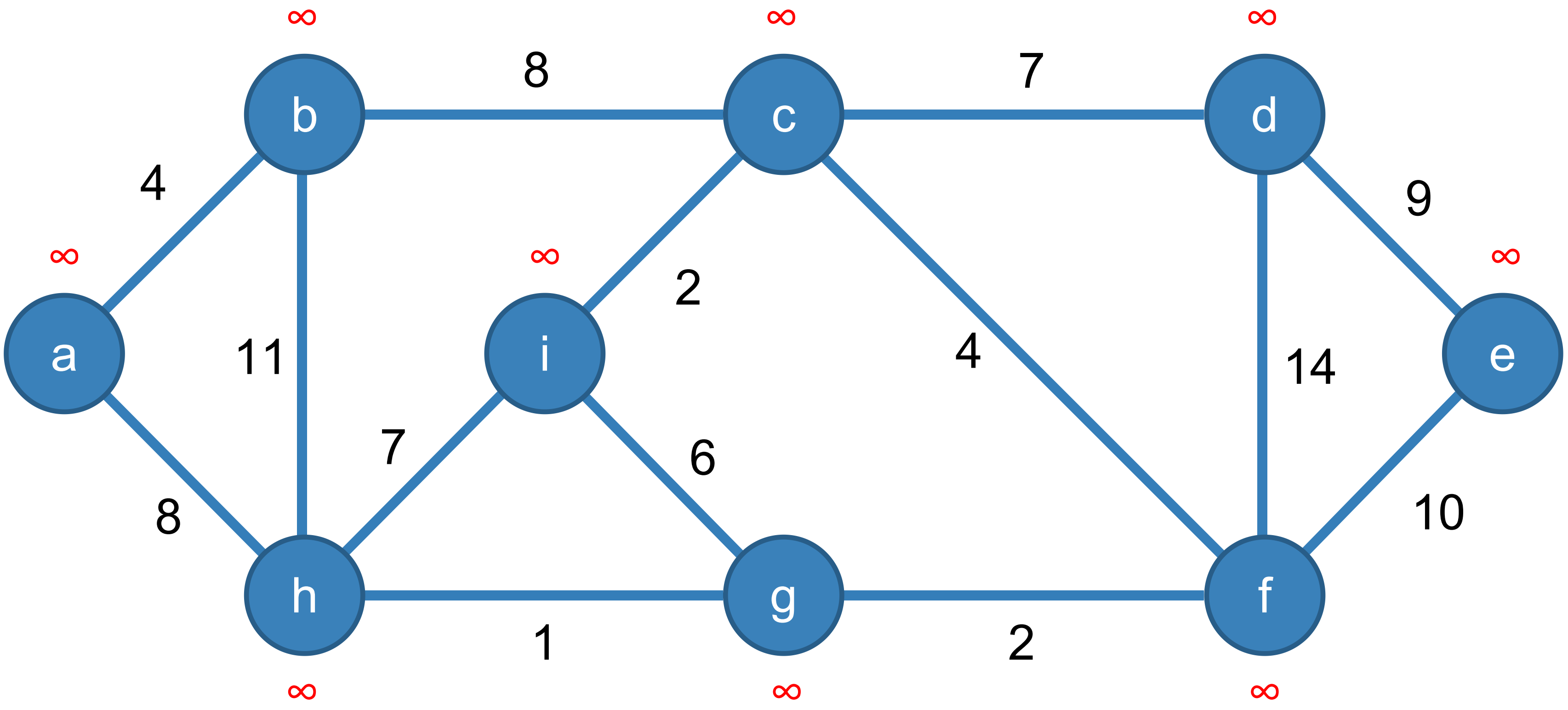
- L'insieme A costituisce sempre una ed una sola componente connessa
- A viene inizializzato con un nodo scelto casualmente
- Ad ogni passo si aggiunge ad A il nodo raggiunto dall'arco leggero che attraversa il taglio A, V\A

- Intuitivamente: ad ogni passo viene aggiunto il nodo raggiungibile dalla componente connessa a distanza minima

Algoritmo di Prim—1957

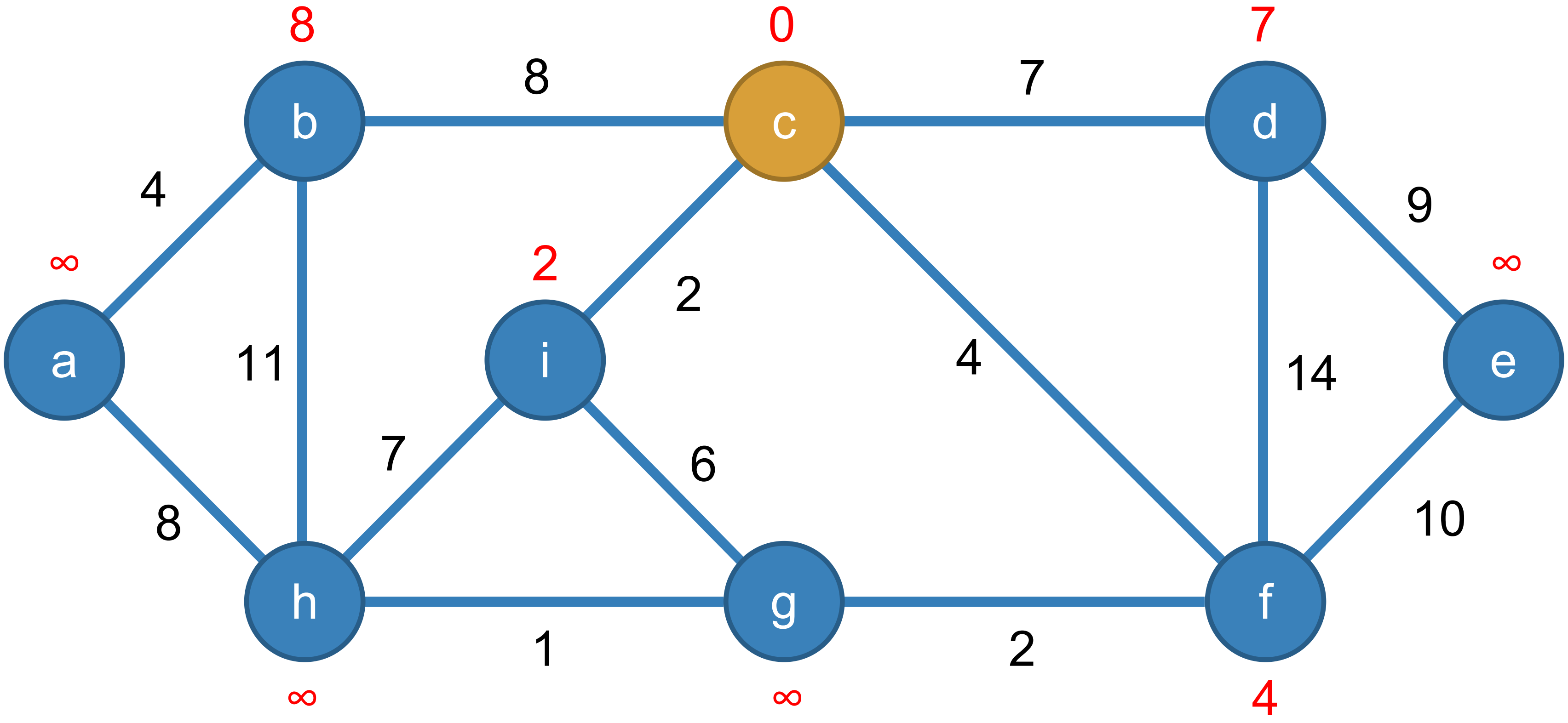
```
PRIM(G, r):
  A = {r};
  MinHeap PQ ← Ø ;
  foreach v in G.V:
    v.distance ← w(r, v); // vale ∞ se r e v non sono adiacenti;
    v.parent ← nil
    PQ.enqueue(v)
  while PQ is not empty:
    u ← PQ.getMin()
    (u, v) ← arco a minima distanza, con v ∈ A
    A ← A∪u
    u.parent ← v
    foreach u in PQ tale che (u,v) ∈ E:
      if u.distance > w(v, u) then:
        PQ.updatePrioritu(u, w(v, u))
```

Algoritmo di Prim—1957



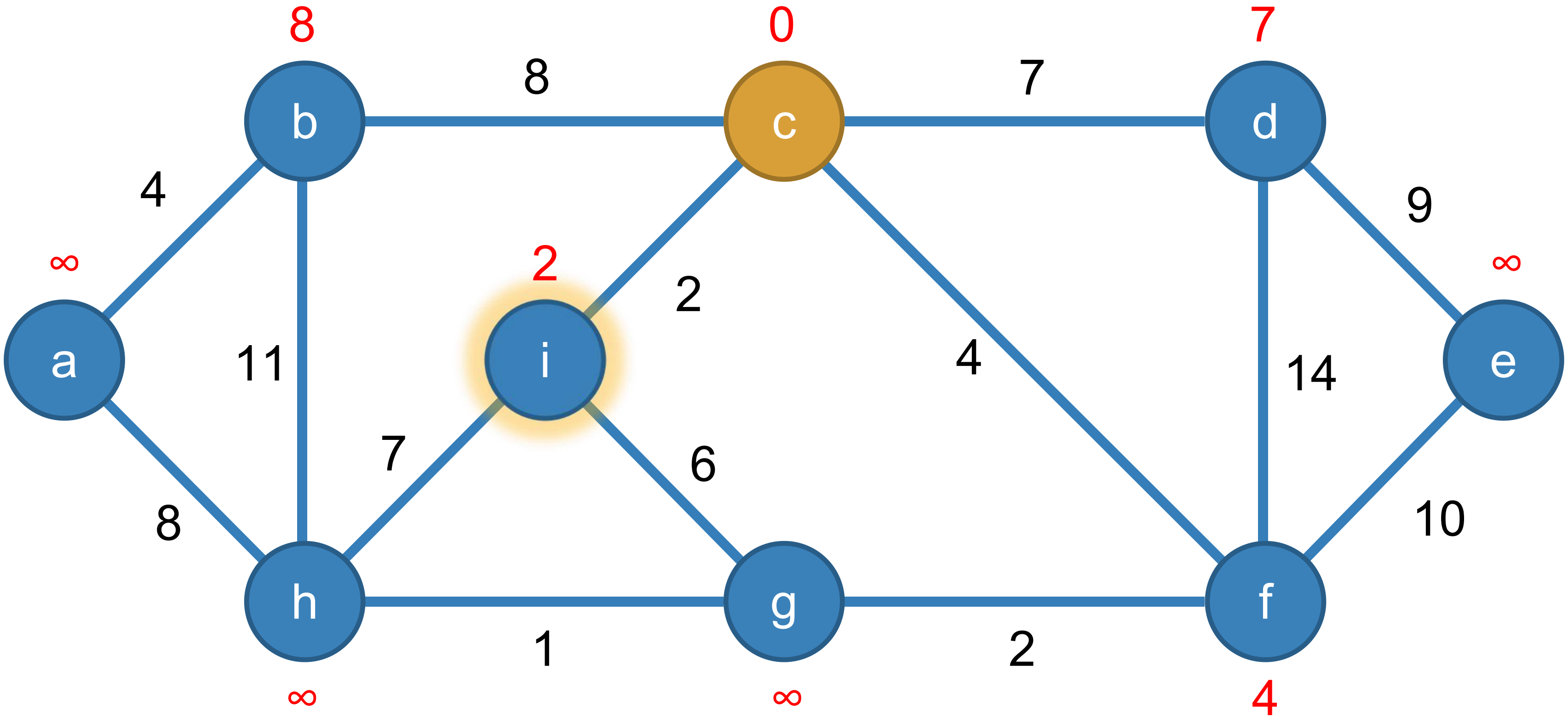
PriorityQueue = { }

Algoritmo di Prim—1957



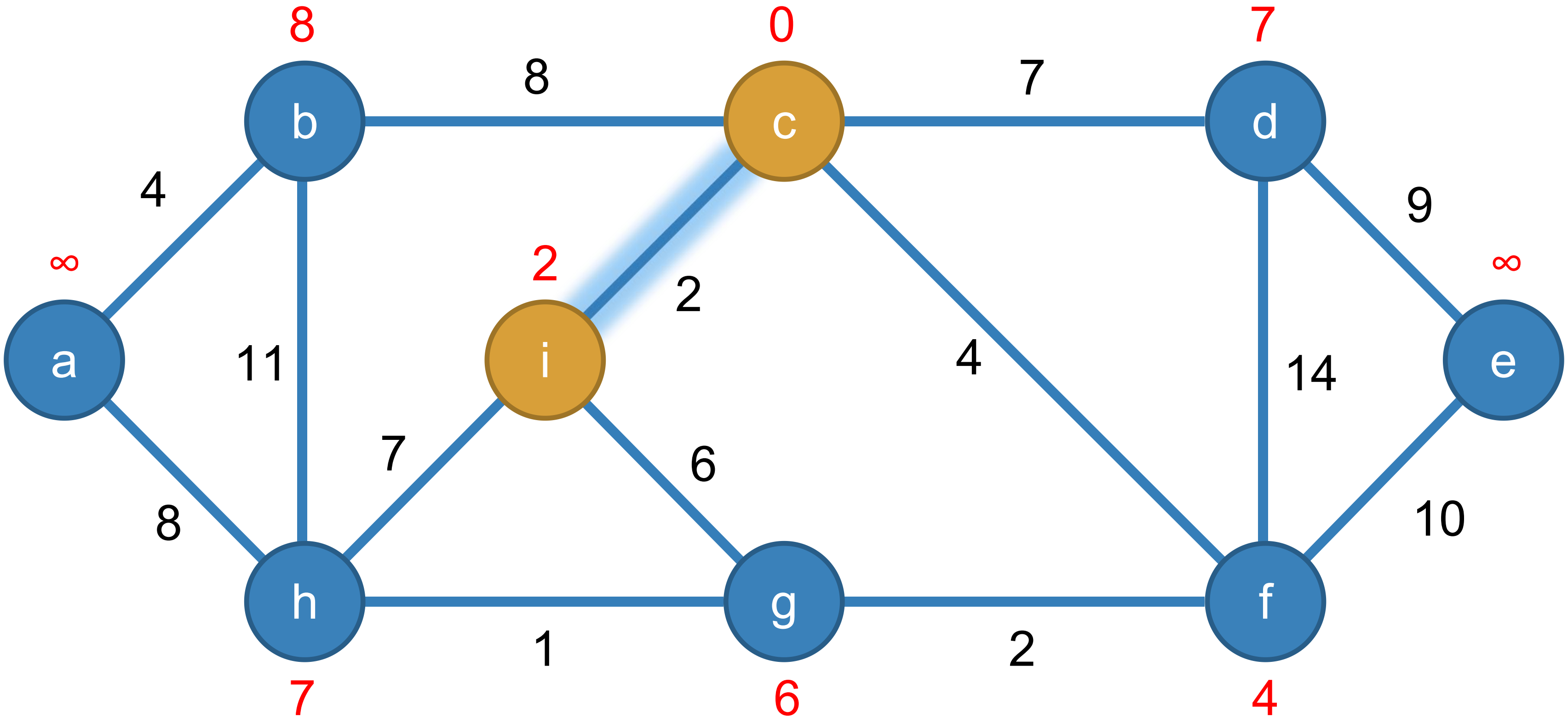
PriorityQueue = { i, f, d, b, a, h, g, e }

Algoritmo di Prim—1957



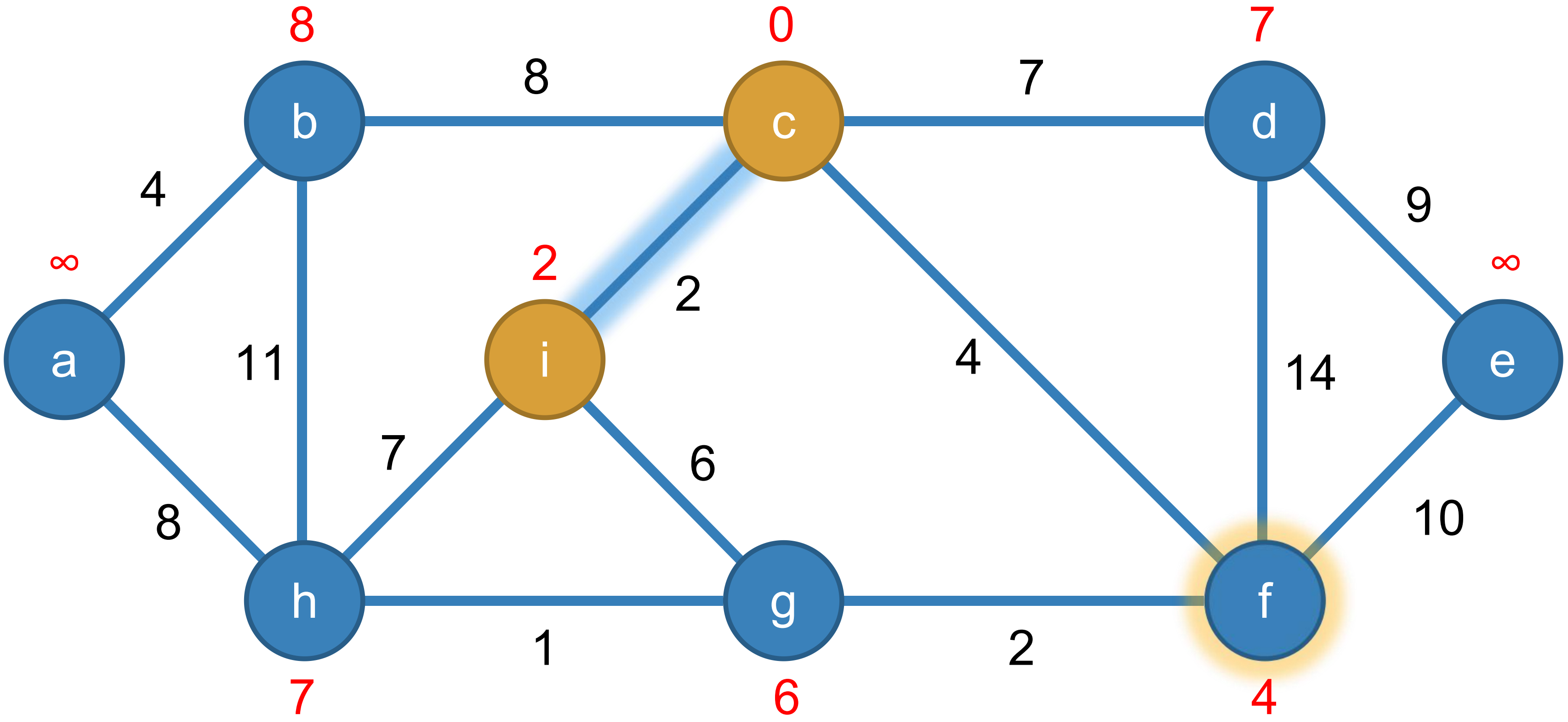
PriorityQueue = { f, d, b, a, h, g, e }

Algoritmo di Prim—1957



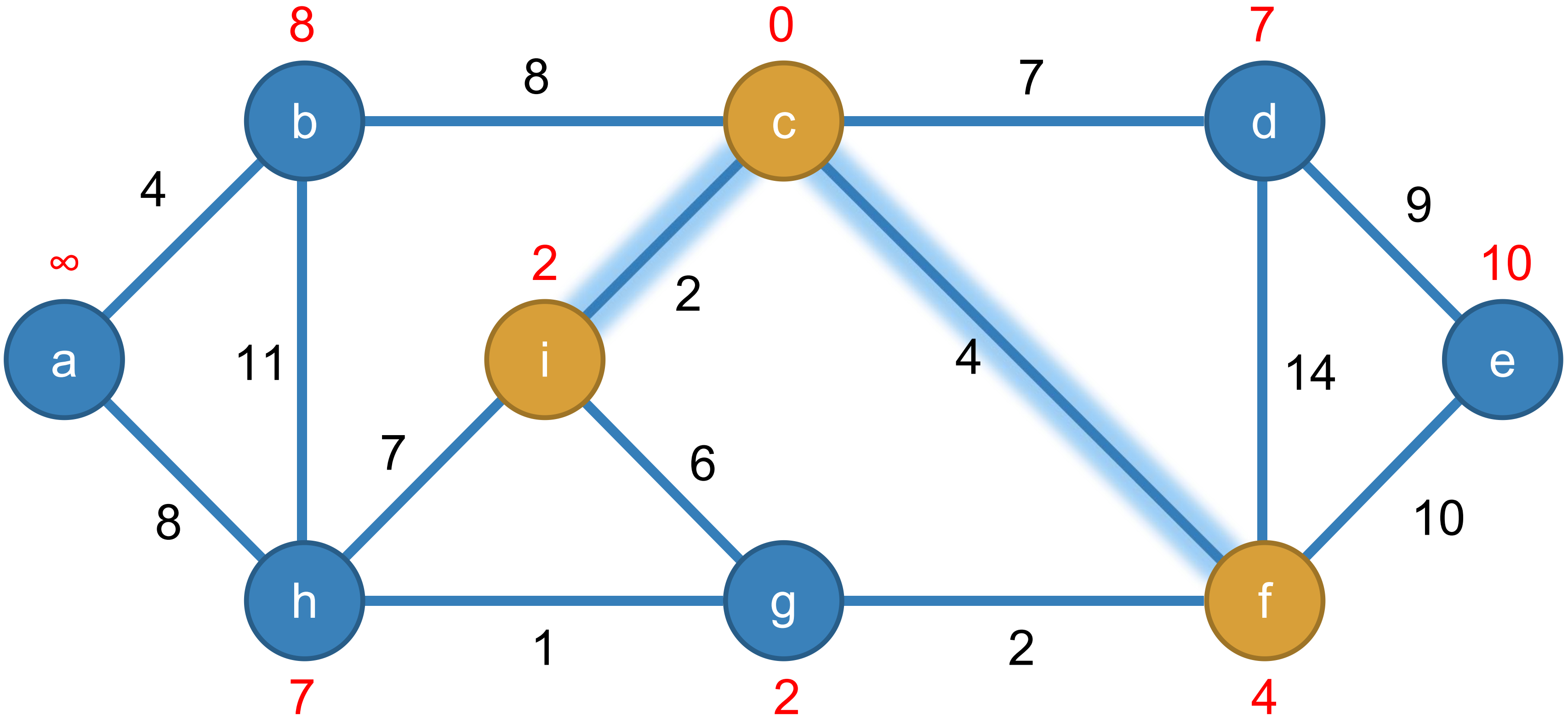
PriorityQueue = { f, g, h, d, b, a, e }

Algoritmo di Prim—1957



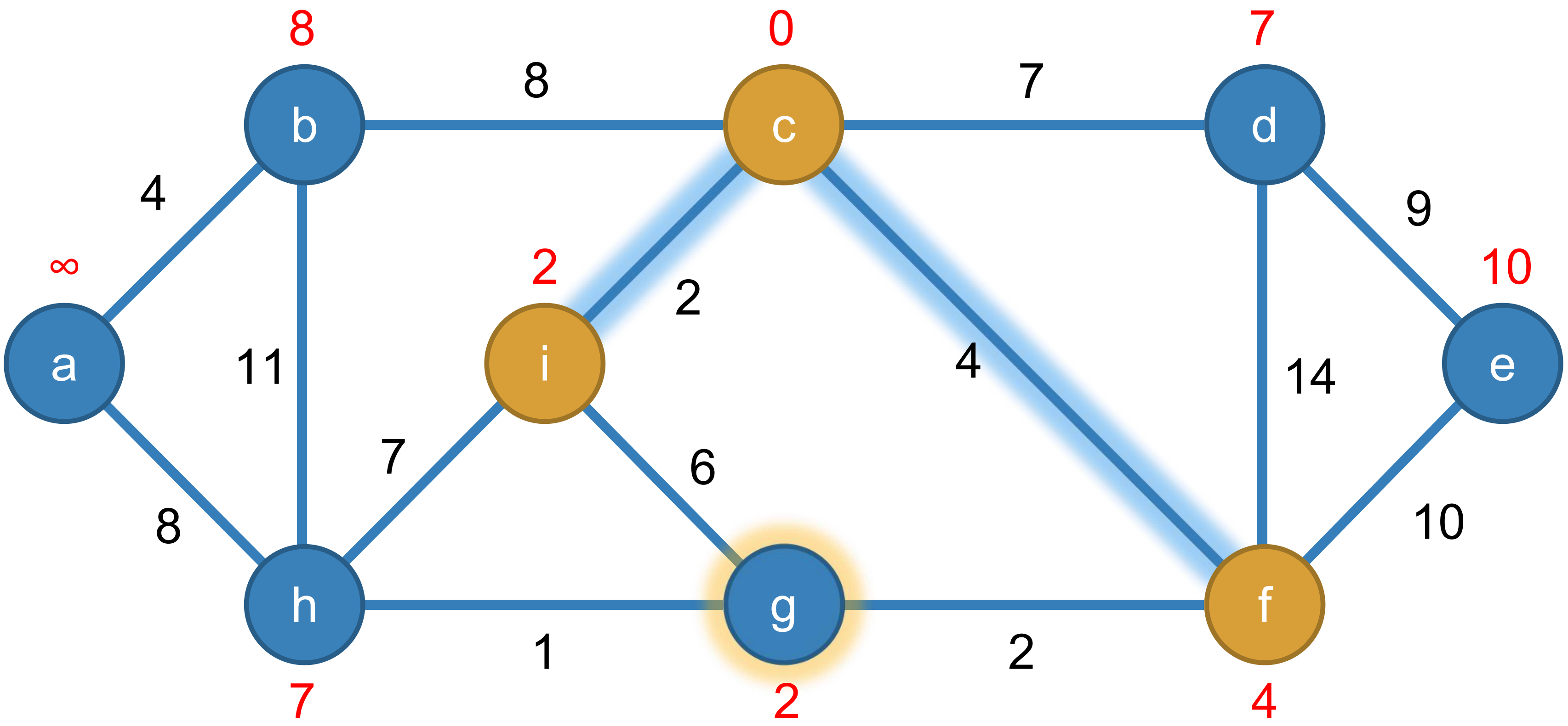
PriorityQueue = { g, h, d, b, a, e }

Algoritmo di Prim—1957



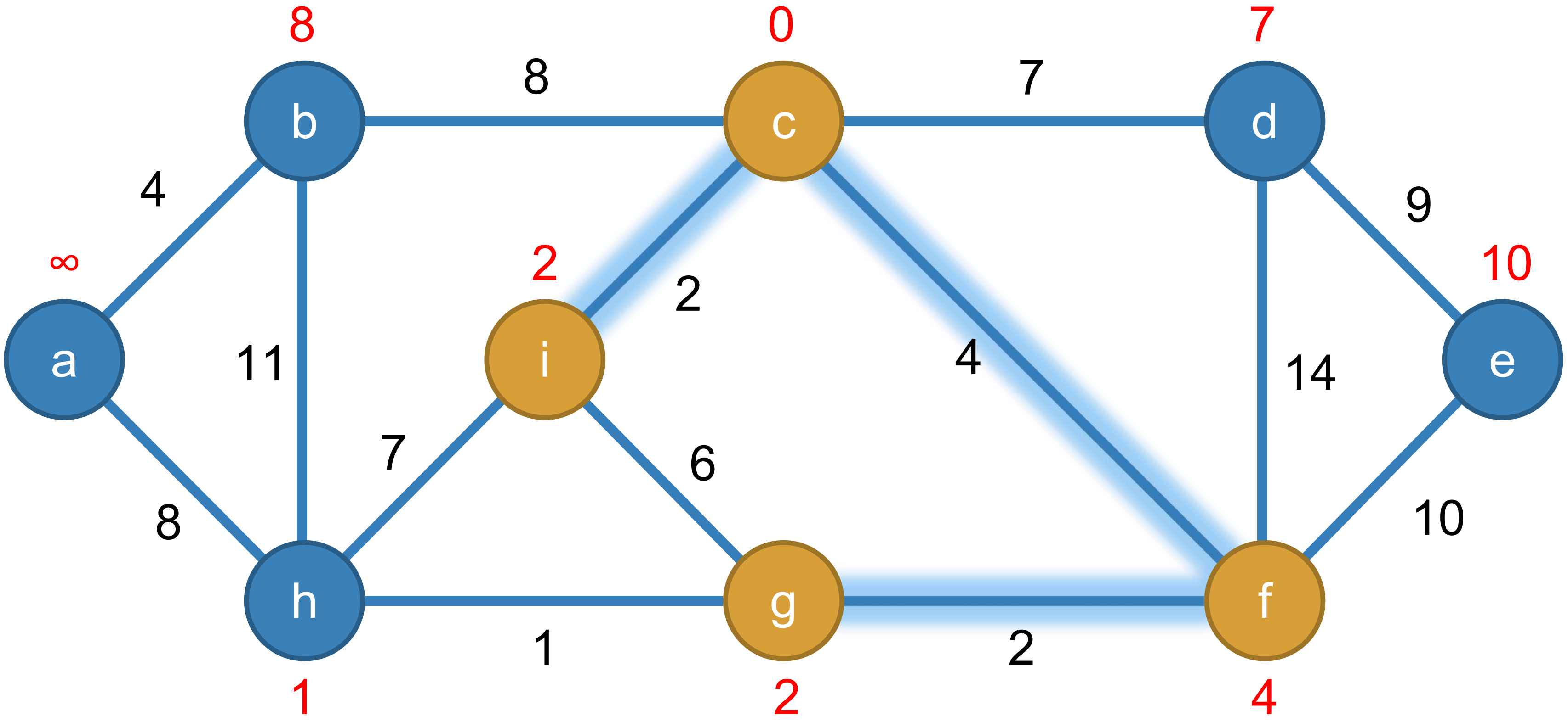
PriorityQueue = { g, h, d, b, e, a }

Algoritmo di Prim—1957



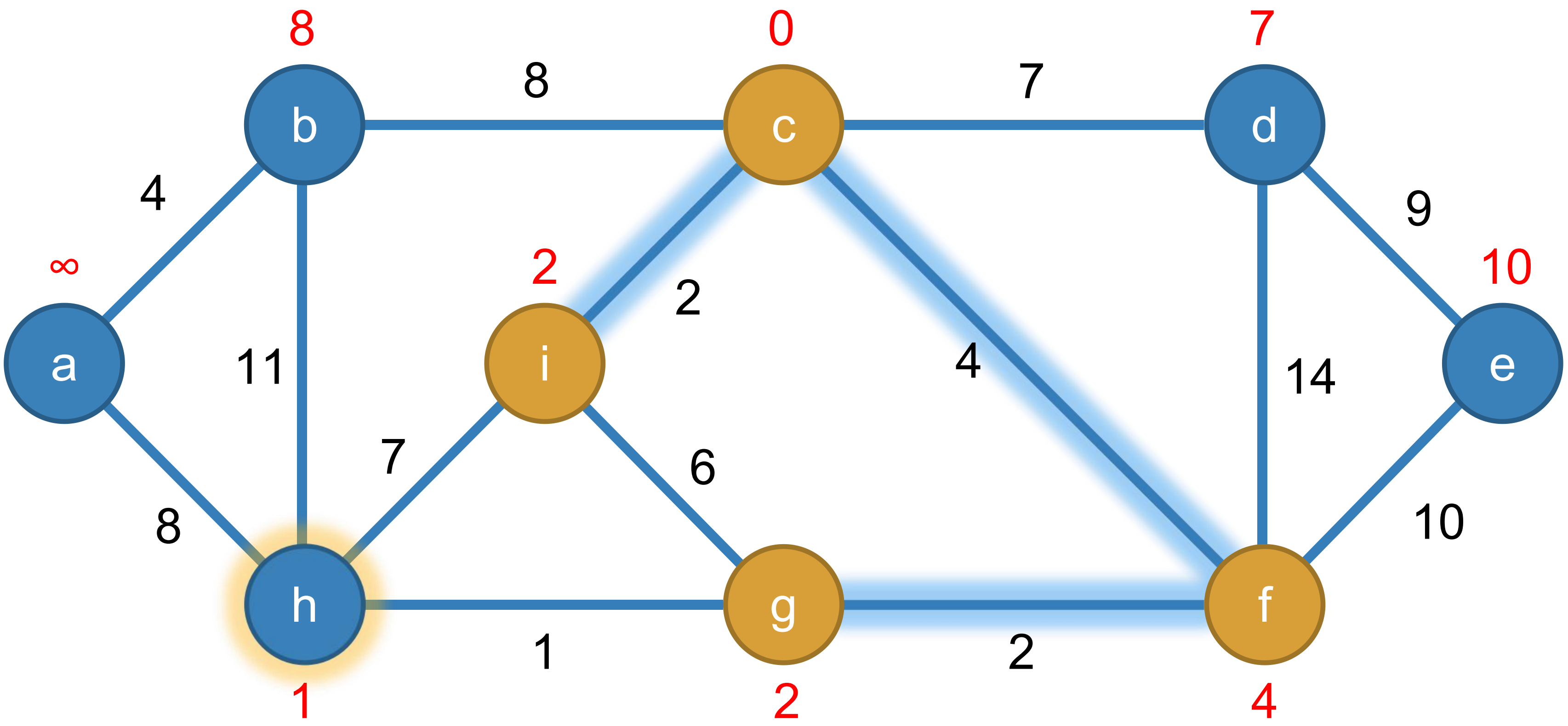
PriorityQueue = { h, d, b, e, a }

Algoritmo di Prim—1957



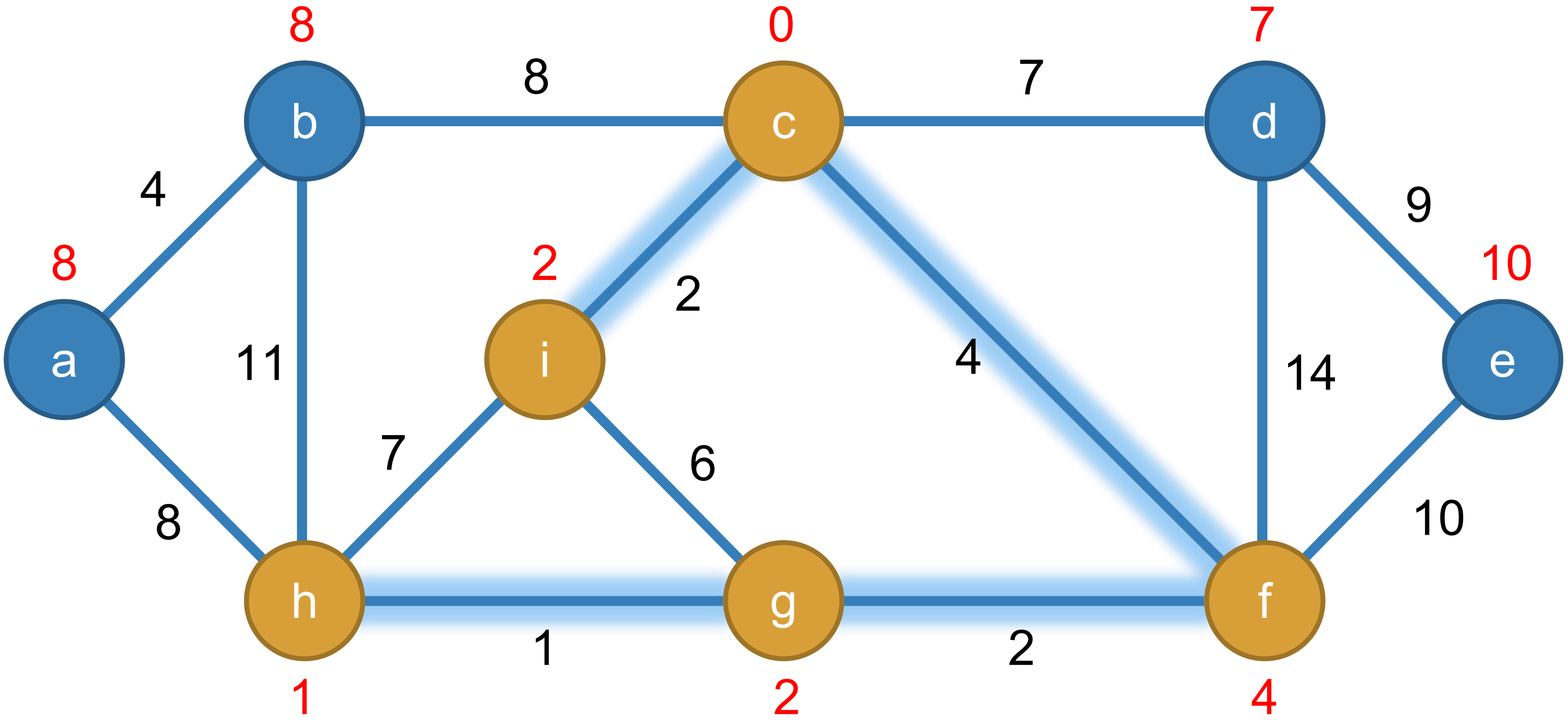
PriorityQueue = { h, d, b, e, a }

Algoritmo di Prim—1957



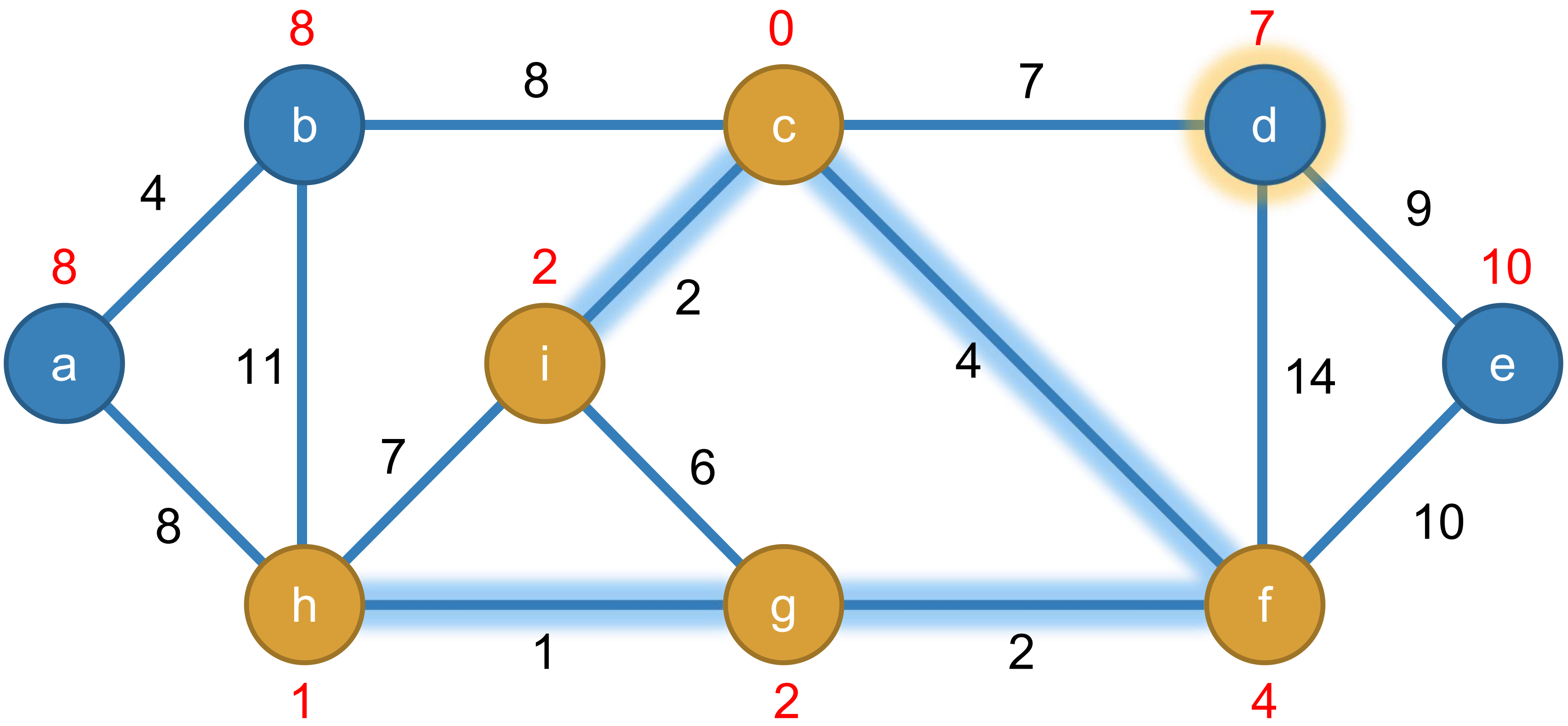
PriorityQueue = { d, b, e, a }

Algoritmo di Prim—1957



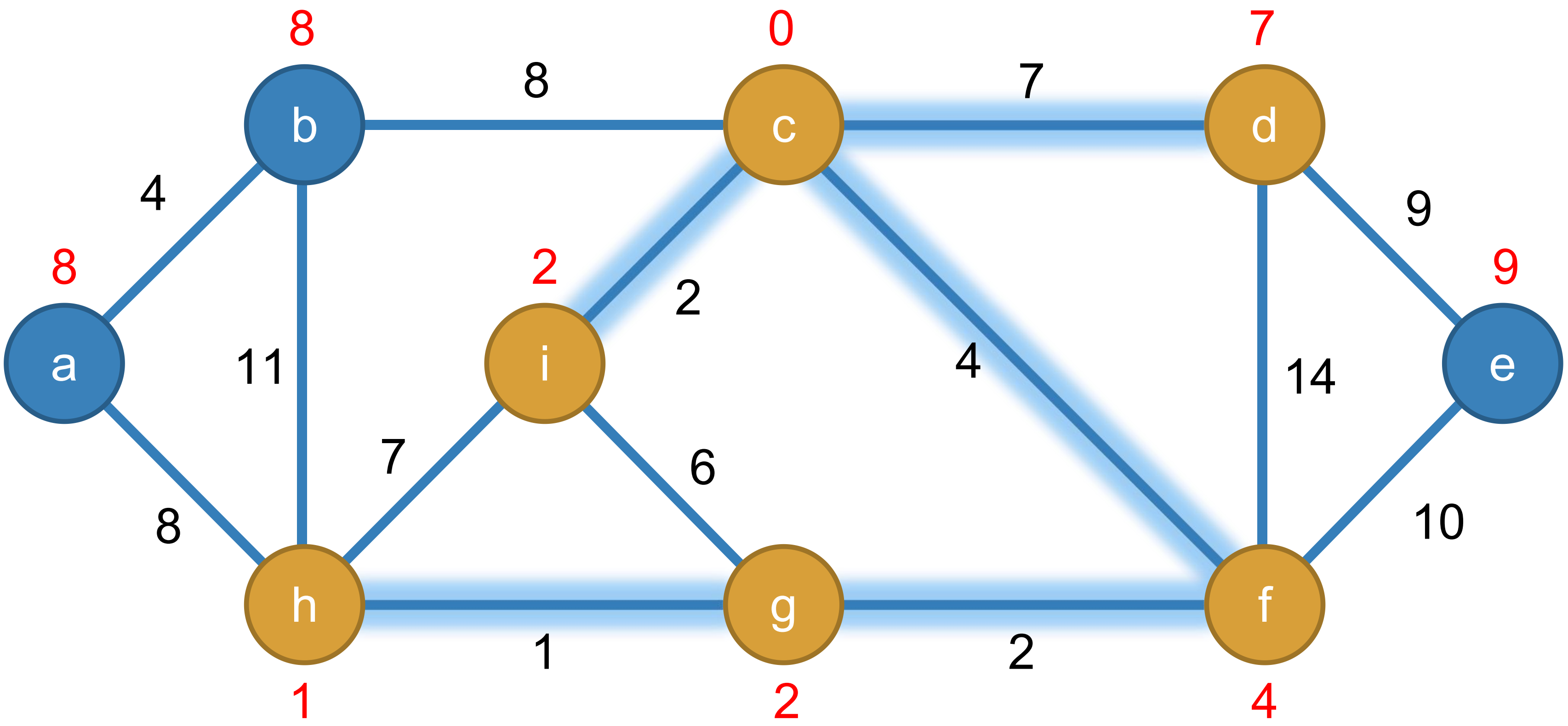
PriorityQueue = { d, a, b, e }

Algoritmo di Prim—1957



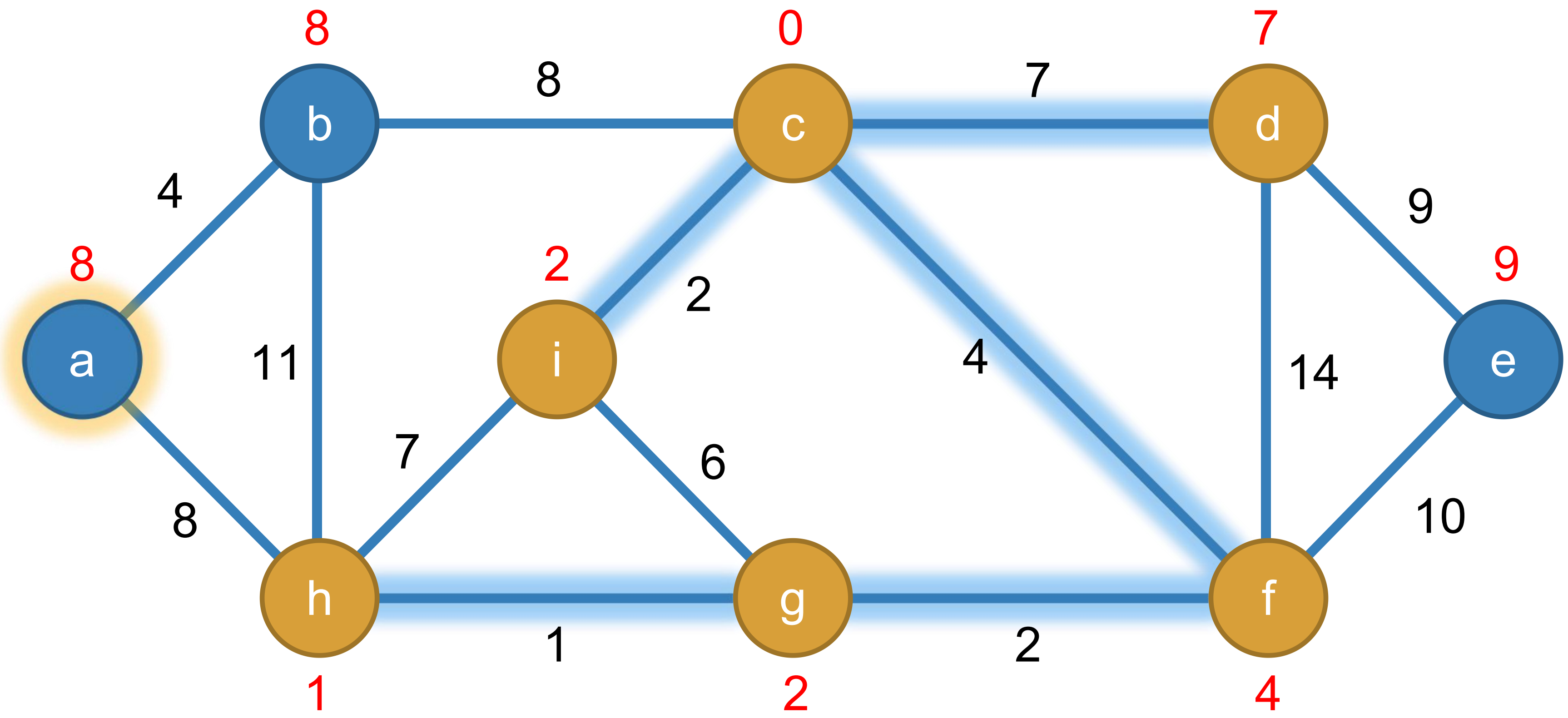
PriorityQueue = { a, b, e }

Algoritmo di Prim—1957



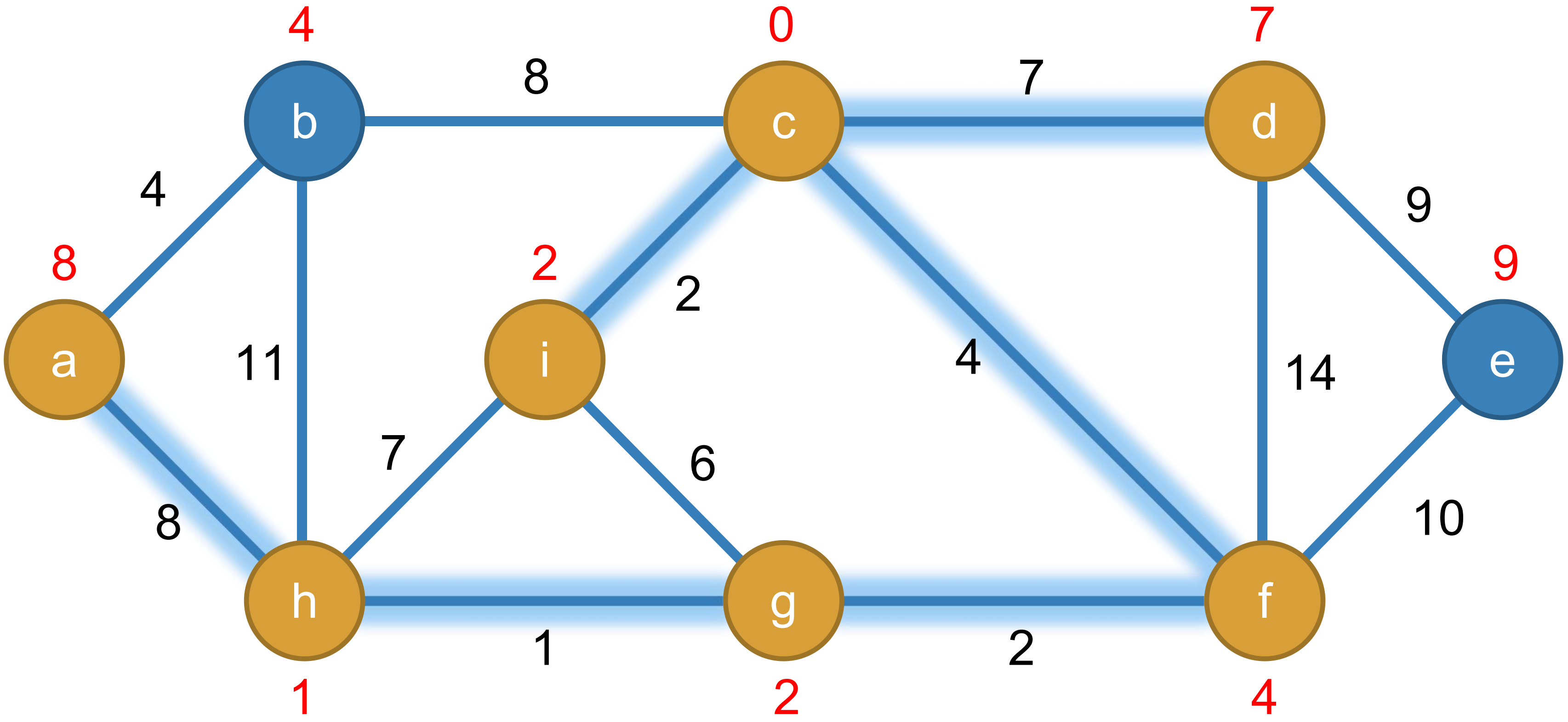
PriorityQueue = { a, b, e }

Algoritmo di Prim—1957



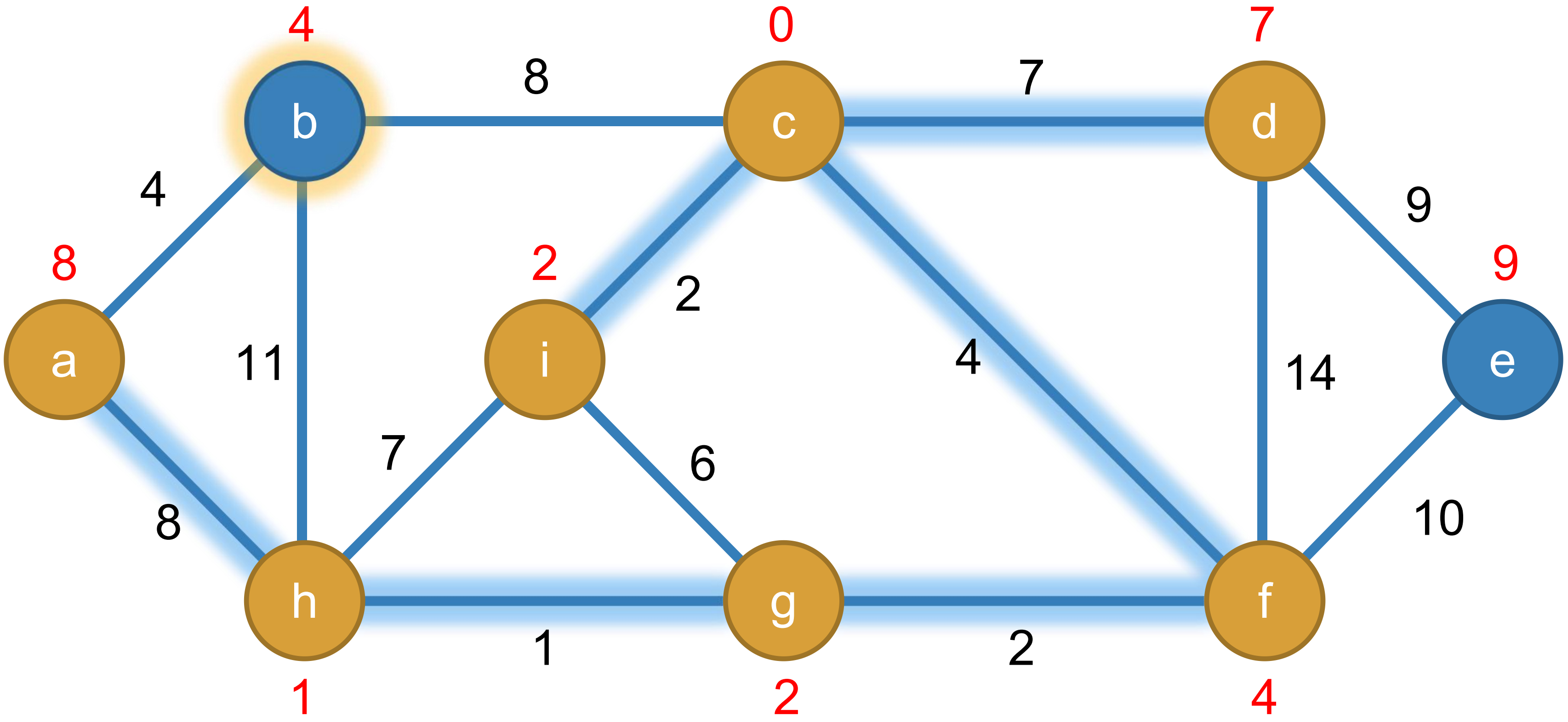
PriorityQueue = { b, e }

Algoritmo di Prim—1957



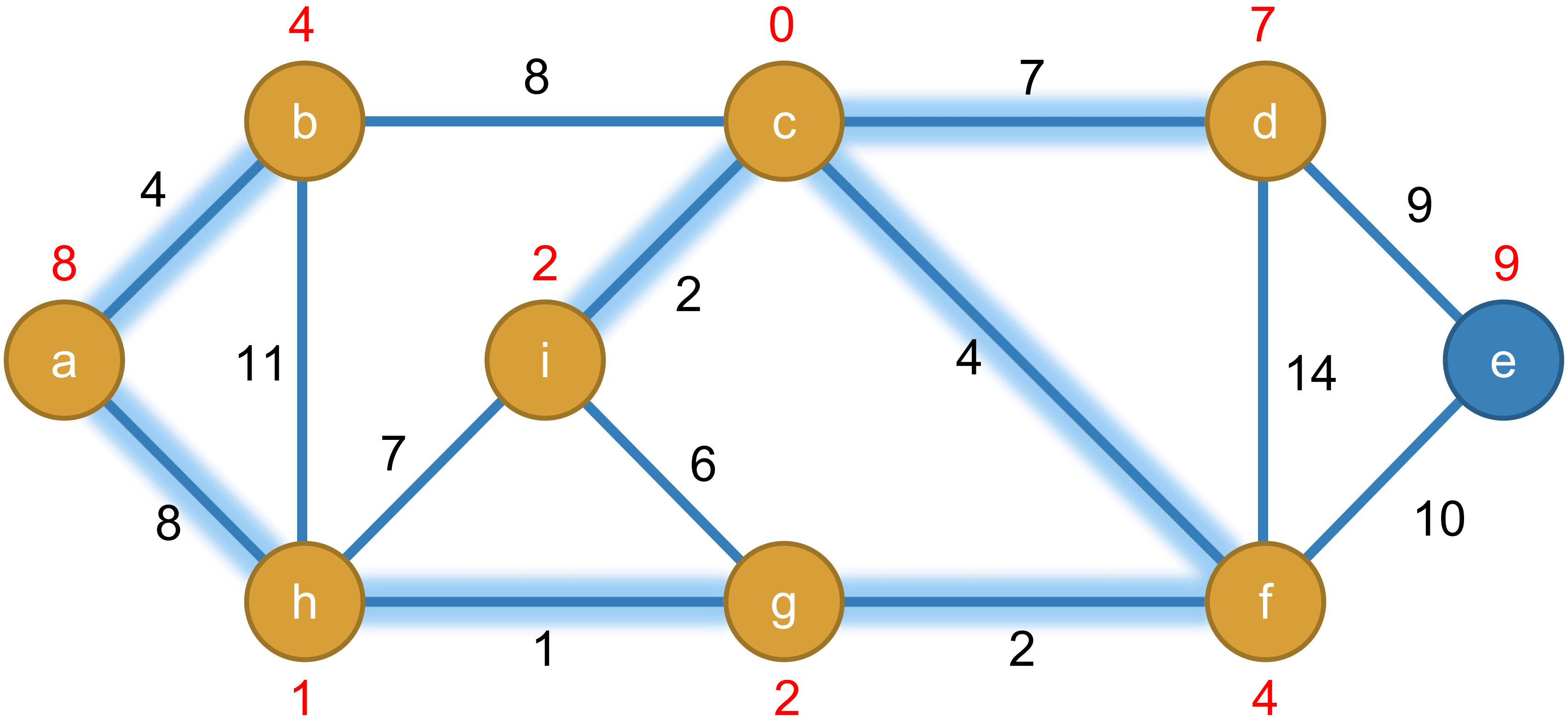
PriorityQueue = { b, e }

Algoritmo di Prim—1957



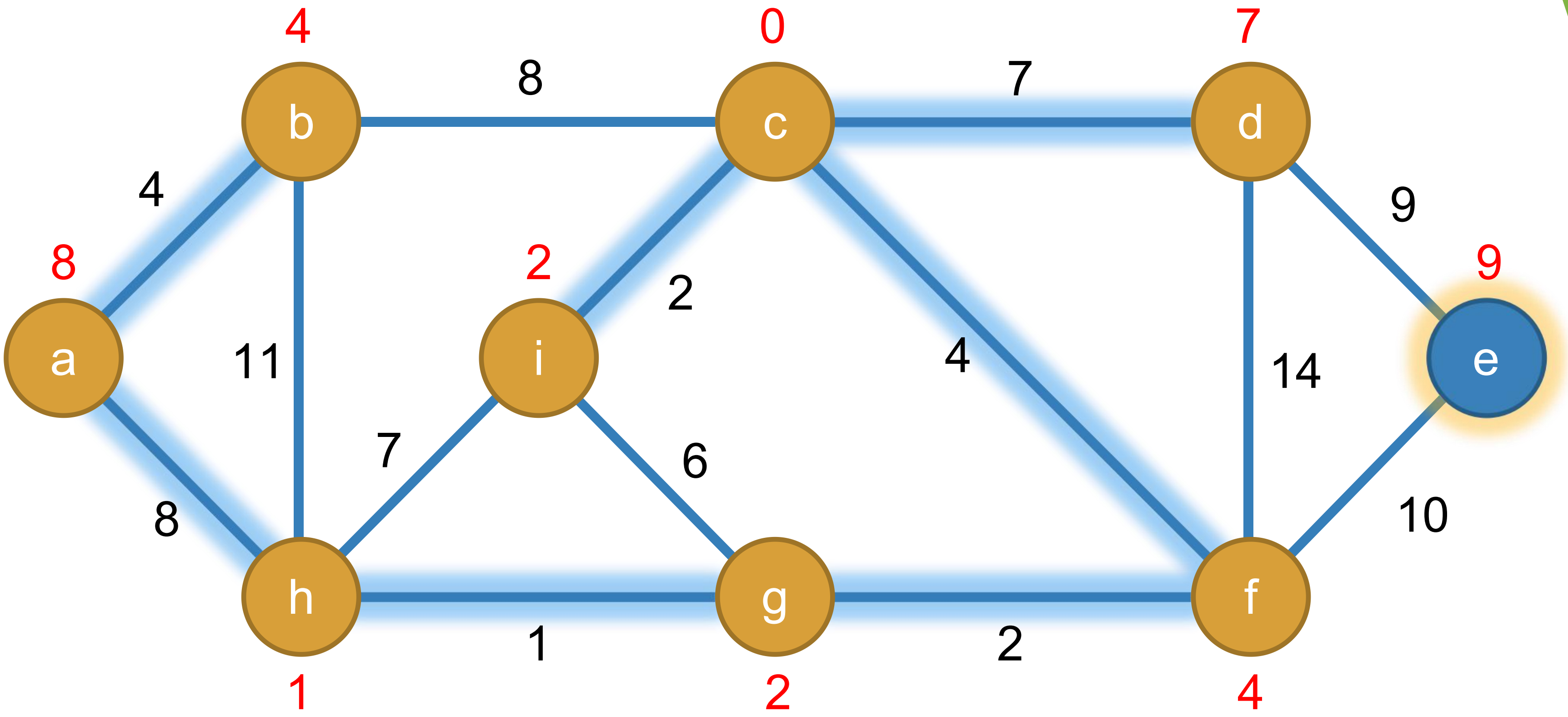
PriorityQueue = { e }

Algoritmo di Prim—1957



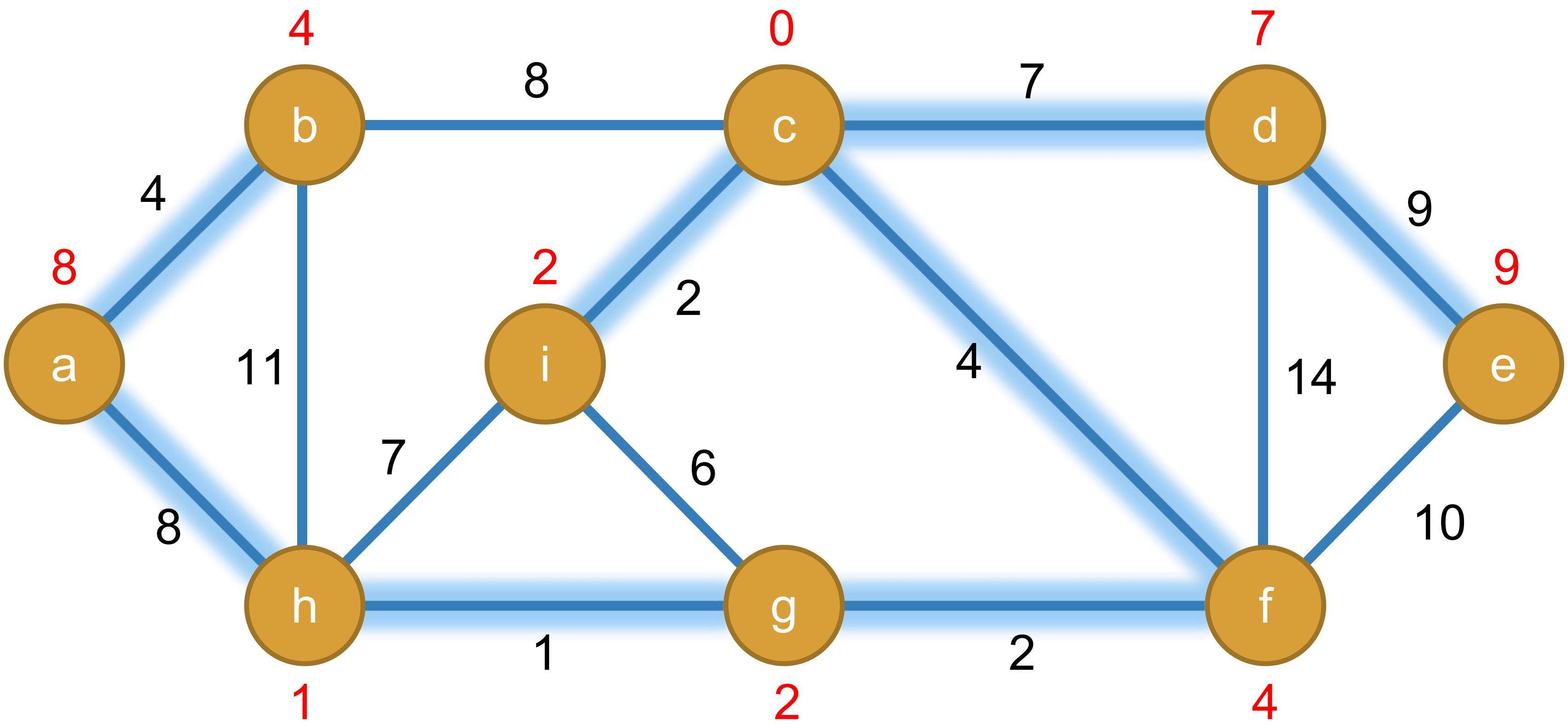
PriorityQueue = { e }

Algoritmo di Prim—1957



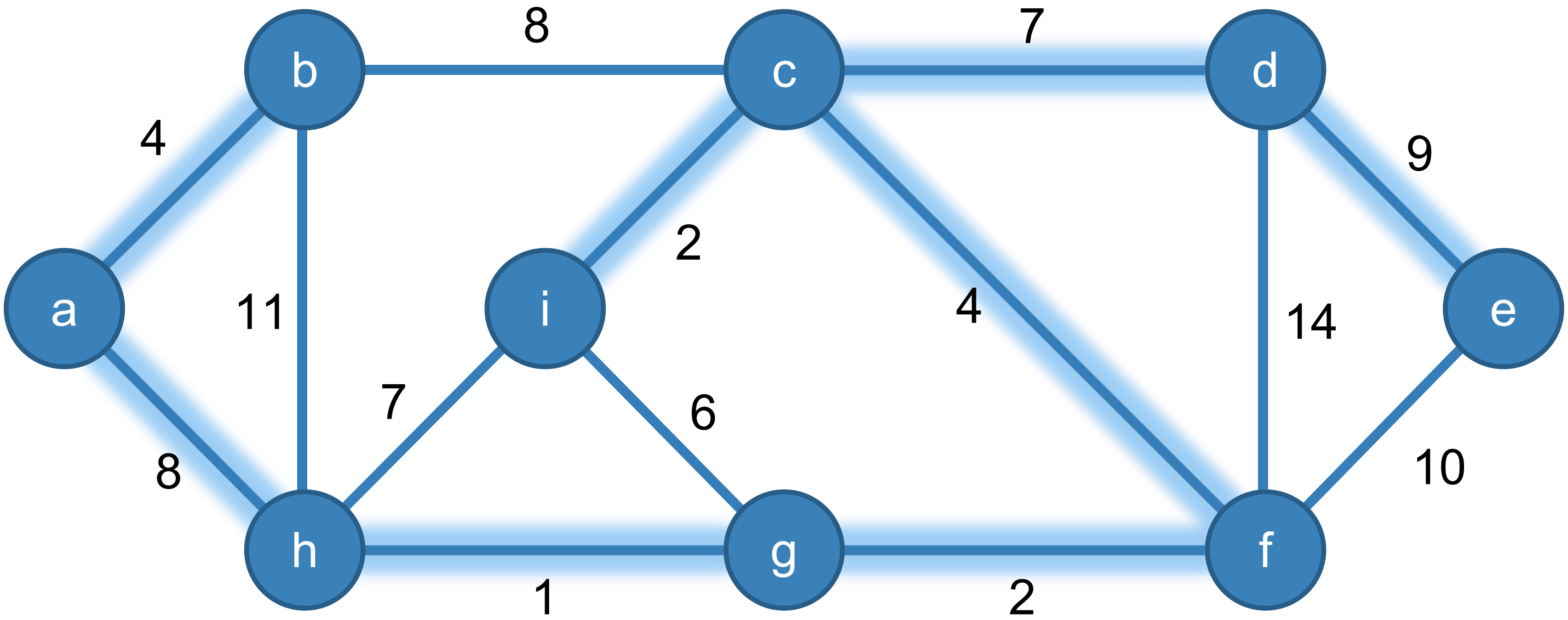
PriorityQueue = { }

Algoritmo di Prim—1957

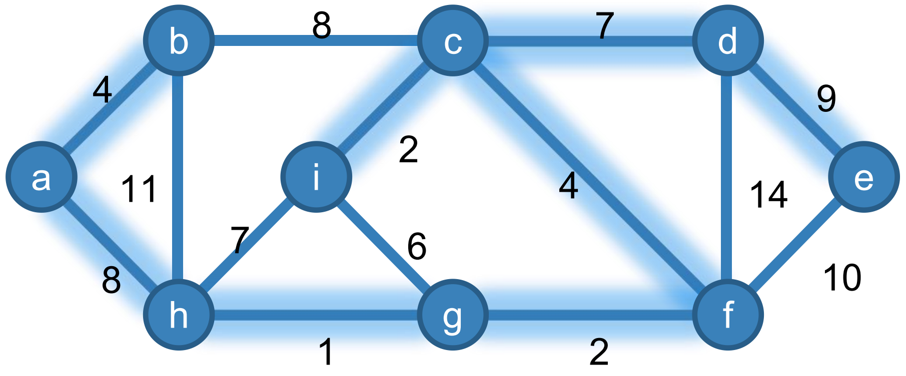


PriorityQueue = { }

Algoritmo di Prim—1957



Confrontro tra MST



Algoritmo di Prim

Costo: 4+8+1+2+4+2+7+9=37

Algoritmo di Kruskal/Borůvka

Costo: 4+8+7+9+2+4+2+1 = 37