Alberi di copertura minimi

Sommario

- Alberi di copertura minimi per grafi pesati
  - Algoritmo di Kruskal
  - Algoritmo di Prim

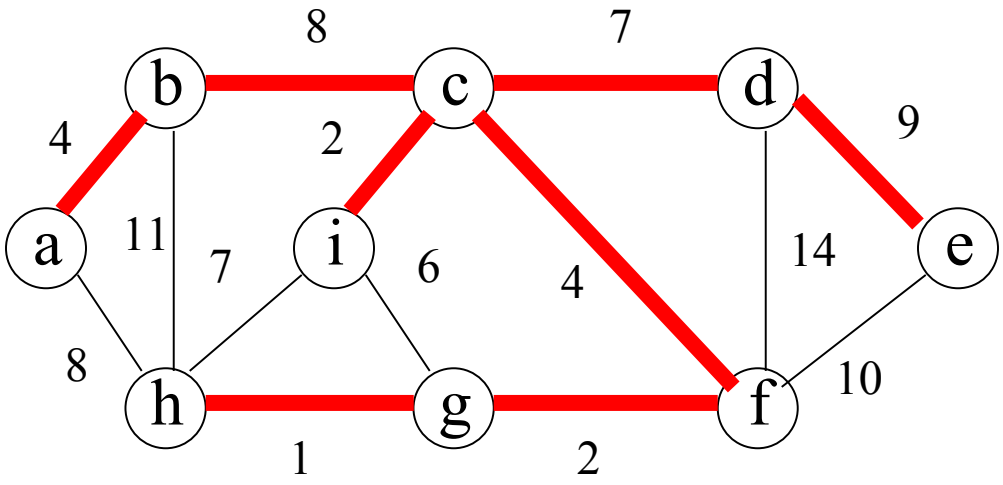
Albero di copertura minimo

- Un problema di notevole importanza consiste nel determinare come interconnettere fra di loro diversi elementi minimizzando certi vincoli sulle connessioni
- Un esempio classico è quello della progettazione dei circuiti elettronici dove si vuole minimizzare la quantità di filo elettrico per collegare fra loro i morsetti di diversi componenti

Albero di copertura minimo

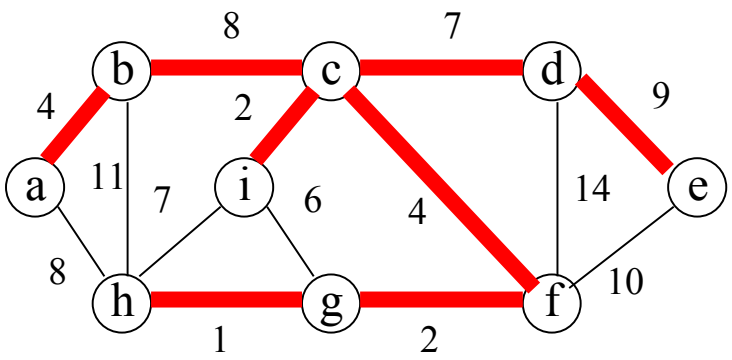
- Il problema può essere modellato con un grafo non orientato e connesso in cui le interconnessioni sono archi pesati (u,v) e dove il peso specifica il costo per connettere u con v
- La soluzione del problema consiste nella determinazione di un sottografo aciclico $T \subseteq E$ che connetta tutti i vertici in modo da minimizzare il peso totale $w(t)=\sum_{(u,v)\in T} w(u,v)$
- Dato che T è aciclico e collega tutti i vertici deve essere un albero
- Tale albero è chiamato *albero di copertura minimo* o *minimum spanning tree MST*

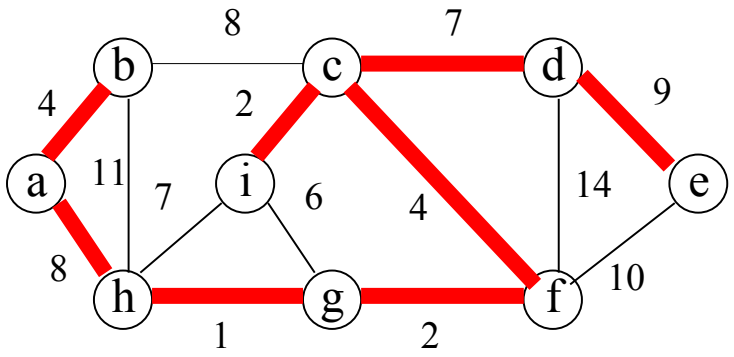
Esempio



Unicità

- L'albero di copertura minimo non è unico
- Ad esempio si possono dare due albero di copertura minimi per il grafo in esame





Algoritmo generico

- Verranno illustrati due algoritmi di tipo "*greedy*" o *golosi*:
  - Algoritmo di Kruskal
  - Algoritmo di Prim
- L'approccio greedy consiste nello scegliere fra più alternative quella più conveniente sul momento
- Nota: in generale non è detto che in ogni tipo di problema questo porti ad una soluzione globalmente ottima.
- Per la soluzione del problema dell'albero di copertura minima una soluzione greedy coincide con una soluzione globalmente ottima

Arco sicuro

- L'idea è di accrescere un sottoinsieme A di archi di un albero di copertura aggiungendo un arco alla volta
- Ad ogni passo si determina un arco che può essere aggiunto ad A mantenendo la proprietà per A di essere un sottoinsieme di archi di un albero di copertura
- Un arco di questo tipo è detto *arco sicuro*

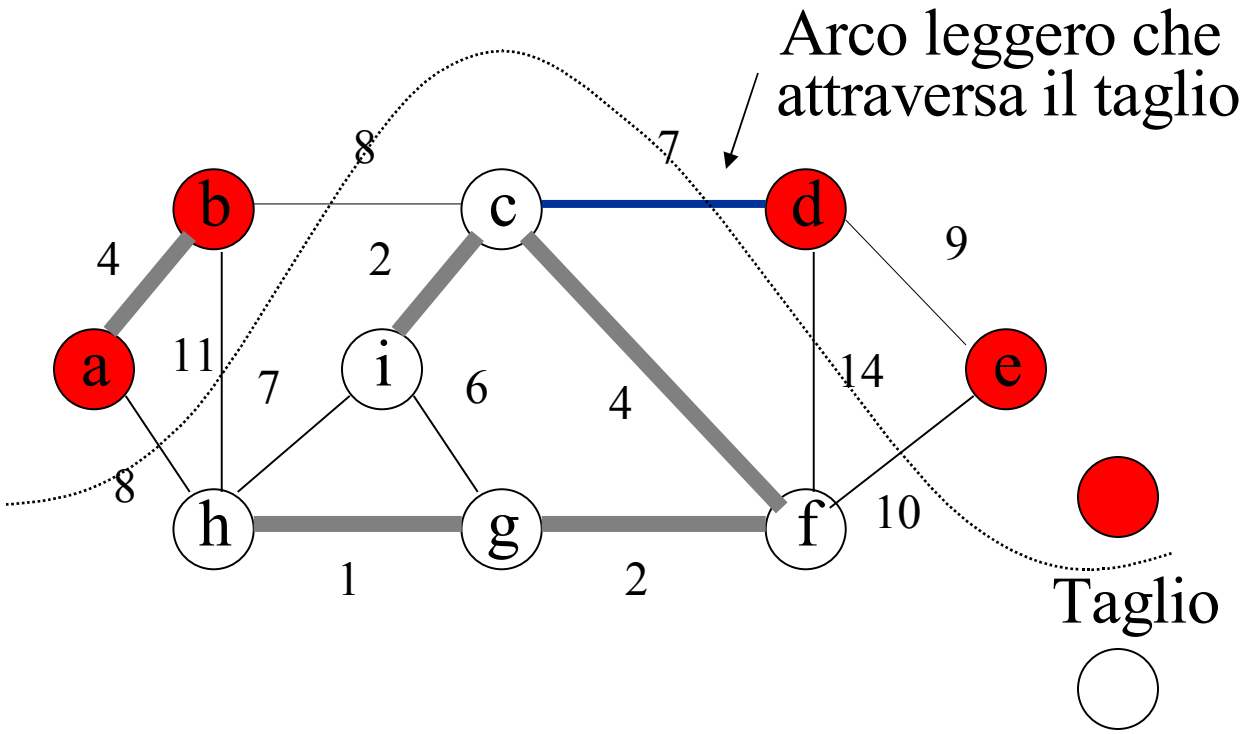
Pseudocodice

```
Generic-MST(G,w)
1  A ← ∅
2  while A non forma un albero di copertura
3  do  trova un arco sicuro (u,v)
4      A ← A ∪ {(u,v)}
5  return A
```

Algoritmi concreti

- Per poter implementare l'algoritmo generico abbiamo bisogno di determinare gli archi sicuri
- Per caratterizzare gli archi sicuri dobbiamo introdurre alcune definizioni:
  - un *taglio* (S,V-S) di un grafo non orientato G=(V,E) è una partizione di V
  - un arco *attraversa* il taglio se uno dei suoi estremi è in S e l'altro è in V-S
  - un taglio *rispetta* un insieme di archi A se nessun arco di A attraversa il taglio
  - un arco *leggero* è un arco con peso minimo

Visualizzazione dei concetti

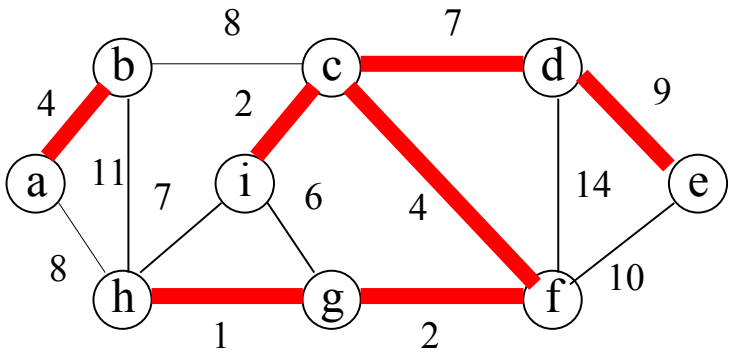


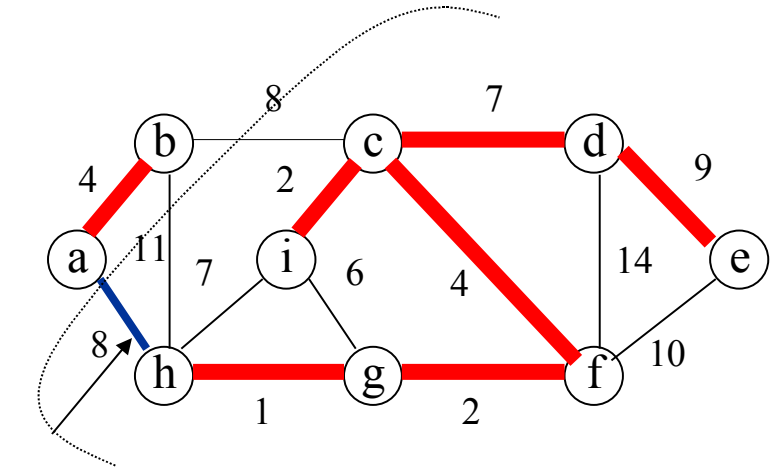
Insieme A: archi in grigio
il taglio rispetta A

Archi sicuri

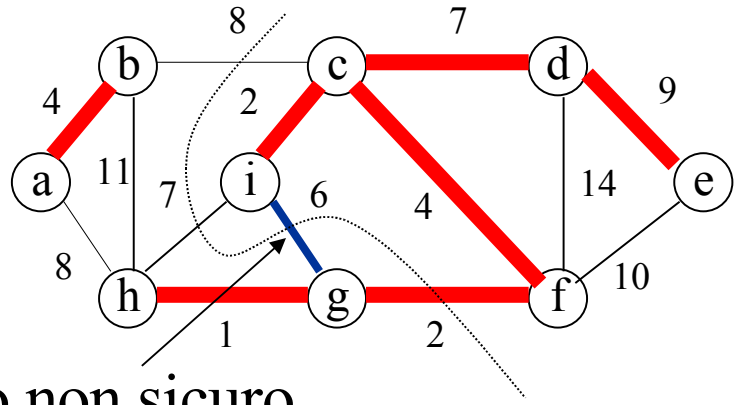
- La regola per riconoscere gli archi sicuri è data dal seguente:
- **Teorema**:
  - Sia G=(V,E) un grafo non orientato e connesso con una funzione peso w a valori reali definita su E.
  - Sia A un sottoinsieme di E contenuto in un qualche albero di copertura minimo per G.
  - Sia (S,V-S) un qualunque taglio che rispetta A.
  - Sia (u,v) un arco leggero che attraversa il taglio.
  - Allora l'arco (u,v) è sicuro per A

Visualizzazione
arco non sicuro perché taglio non rispetta



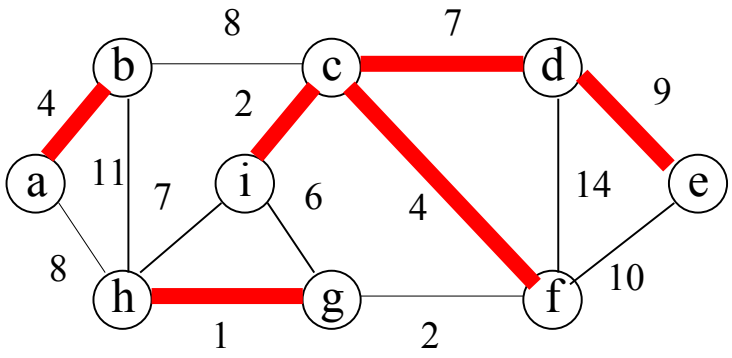


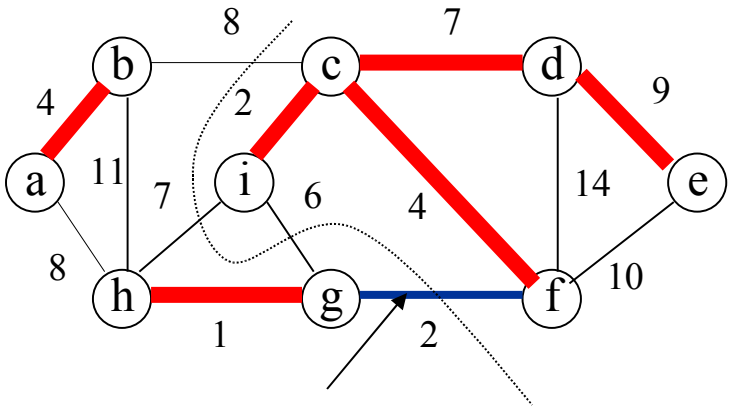
Arco sicuro



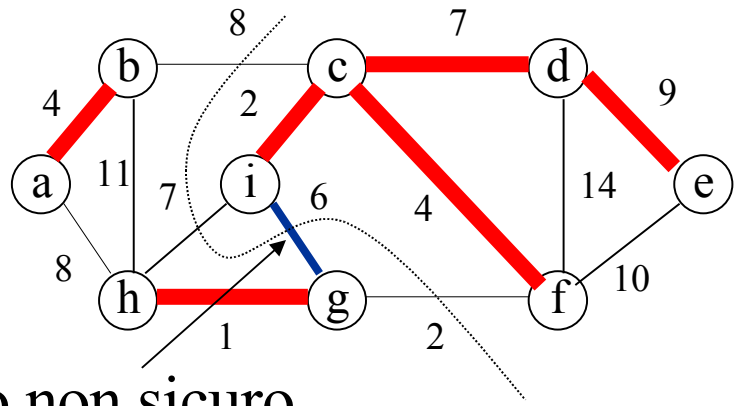
Arco non sicuro

Visualizzazione
arco non sicuro perché non leggero





Arco sicuro



Arco non sicuro

Archi sicuri

- Dimostrazione:
  - sia T l'albero di copertura minimo che contiene A
  - l'arco (u,v) o appartiene a T, e quindi è sicuro per A
  - oppure non appartiene a T:
  - in questo caso faremo vedere che sostituendo un arco (x,y)in T con il nuovo arco (u,v) otteniamo un T' che è sempre un albero di copertura minimo, infatti:
    - deve esserci un altro arco (x,y) di T che attraversa il taglio perché T è un insieme connesso su tutti i vertici (un qualsiasi taglio non rispetta T) e quindi deve esserci un percorso fra u e v che sono da parti opposte del taglio
    - inoltre (x,y) non è in A perché il taglio rispetta A
    - se costruisco T' come T-(x,y)+(u,v) ho creato un insieme connesso con costo complessivo $\leq$ di T che copre tutti i vertici e che ha costo minimo e quindi deve essere un albero di copertura minimo
    - in pratica (u,v) deve essere un arco con costo equivalente a (x,y)

Archi sicuri

- cosi' abbiamo fatto vedere che (u,v) e' sicuro per un sottoinsieme di T', ma noi vogliamo far vedere che (u,v) e' sicuro per A in T e non in T', ma...
- se T' è un albero di copertura minimo allora dato che A è compreso in T e che (x,y) non era in A allora A è anche compreso in T'
- infatti l'unico cambiamento da T in T' è stato solo per l'arco (x,y) e (u,v) che non sono in A, gli altri archi sono rimasti inalterati
- di conseguenza aggiungere (u,v) ad A mantiene A un sottoinsieme dell'albero di copertura (T' questa volta) e dunque è un arco sicuro per A

Archi sicuri

- Corollario:
  - Sia G=(V,E) un grafo non orientato e connesso con una funzione peso w a valori reali definita su E.
  - Sia A un sottoinsieme di E contenuto in un albero di copertura minimo per G
  - Sia C una componente connessa (un albero) nella foresta G_A=(V,A)
  - Se (u,v) è un arco leggero che connette C a qualche altra componente in G_A
  - allora (u,v) è sicuro per A
- Dimostrazione:
  - il taglio (C,V-C) rispetta A: quindi l'arco leggero (u,v) è un arco sicuro per A per il teorema precedente

Algoritmo di Kruskal

- L'idea dell'algoritmo di Kruskal è di ingrandire sottoinsiemi disgiunti dell'albero di copertura minimo connettendoli fra di loro fino ad avere l'albero complessivo
- In particolare si individua un arco sicuro da aggiungere alla foresta scegliendo un arco (u,v) di peso minimo tra tutti gli archi che connettono due distinti alberi (componenti connesse) della foresta
- L'algoritmo è greedy perché ad ogni passo si aggiunge alla foresta un arco con il peso minore possibile

Algoritmo di Kruskal

- L’idea è che, come richiesto dall’algoritmo astratto:
  - ogni componente connessa in A appartiene all’albero di copertura
  - unendo componenti connesse tramite archi leggeri (per il corollario) si stanno aggiungendo archi sicuri
  - ad ogni fusione di componenti connesse stiamo espandendo A
  - si può continuare fino a quando non si sono acquisiti tutti i vertici del grafo

Implementazione

- L’algoritmo presentato è simile a quello usato per calcolare le componenti connesse
- Si usa una struttura dati per insiemi disgiunti
- Ogni insieme contiene i vertici di un albero della foresta corrente
- Si può determinare se due vertici appartengono allo stesso albero verificando l'eguaglianza degli elementi rappresentanti restituiti da Find-Set
- Si fondono due alberi tramite la Union

Pseudocodice Kruskal

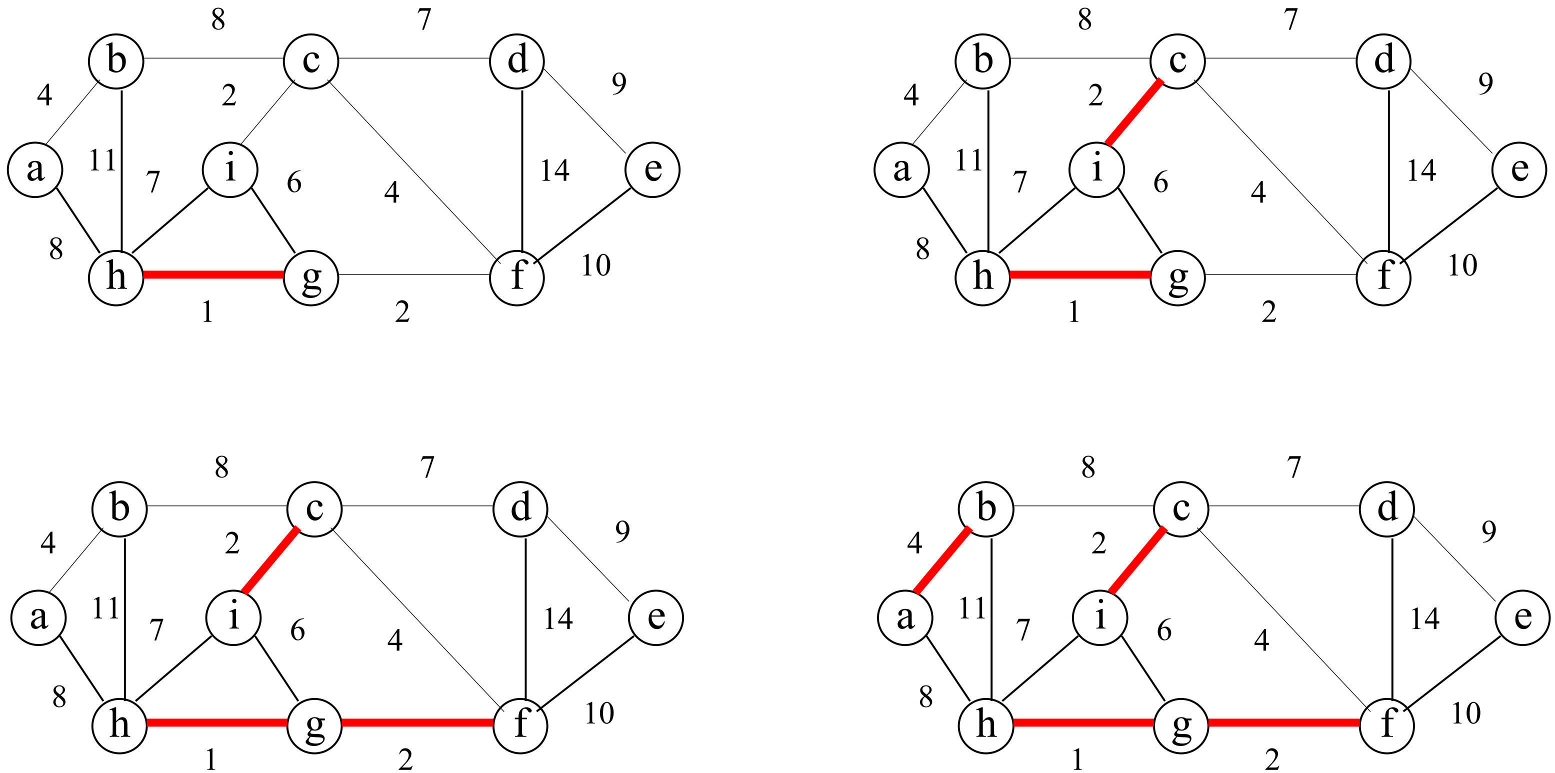
```
MST-Kruskal(G,w)
1  A ← ∅
2  for all v in V[G]
3  do  Make-Set(v)
4  ordina gli archi di E per peso w non decrescente
5  for all (u,v) in E in ordine di peso non decrescente
6  do  if Find-Set(u)≠ Find-Set(v)
7       then   A ← A ∪ {(u,v)}
8              Union(u,v)
9  return A
```

Spiegazione dello pseudocodice

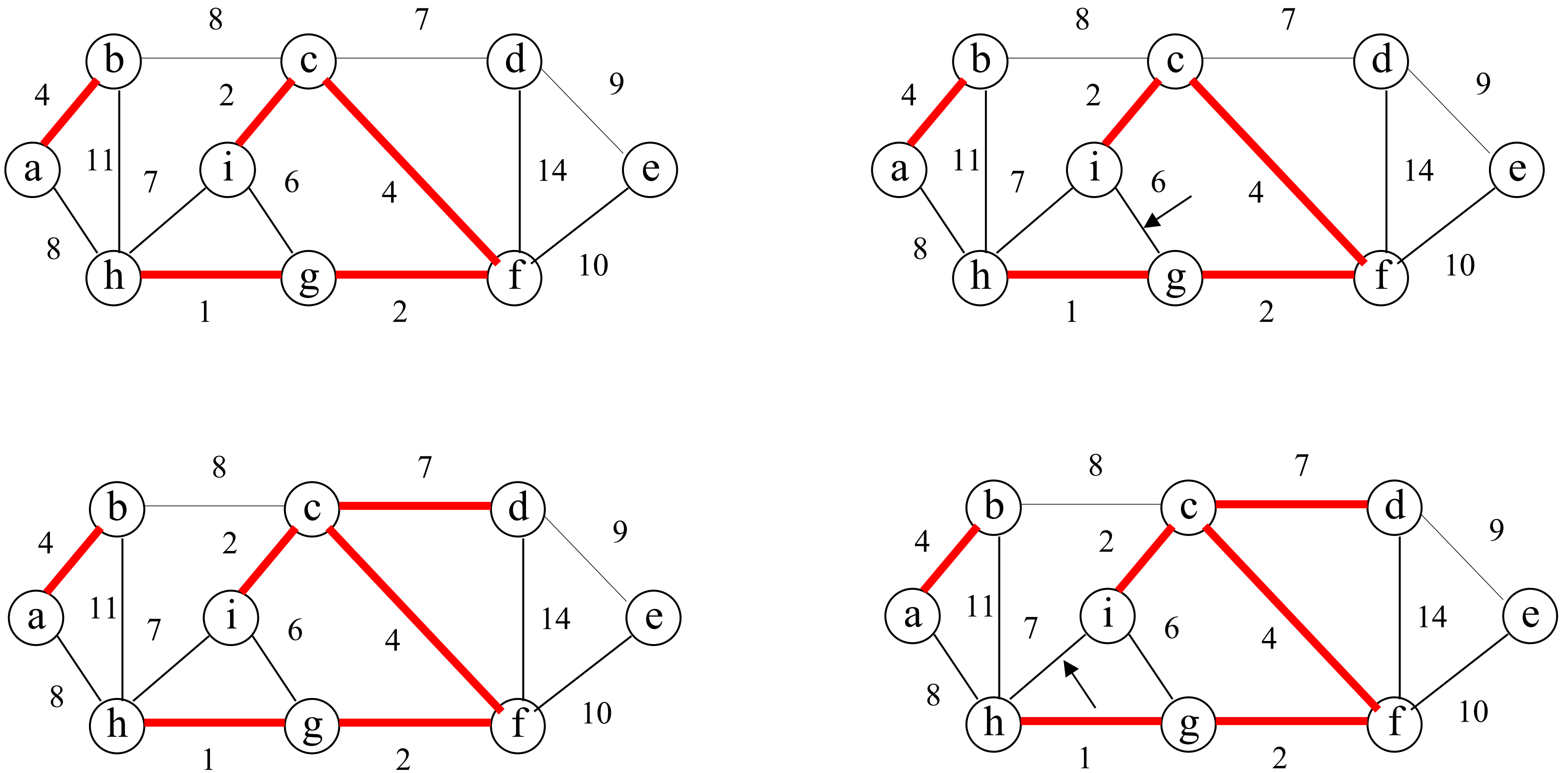
- Le linee 1-3 inizializzano l'insieme A con l'insieme vuoto e creano |V| alberi, uno per ogni vertice
- la linea 4 ordina gli archi per peso
- nelle linee 5-8 il ciclo for controlla che i vertici di ogni arco appartengano ad alberi diversi
- in caso affermativo
  - l'arco viene aggiunto ad A
  - la linea 8 fonde i due alberi in un unico insieme

  - Nota: se i vertici appartenessero allo stesso albero collegheremmo due vertici di un albero ottenendo un ciclo, facendo venire meno la condizione di aciclicità del sottografo di ricoprimento

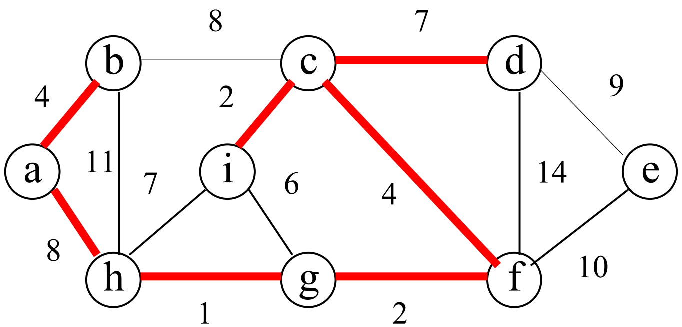
Visualizzazione

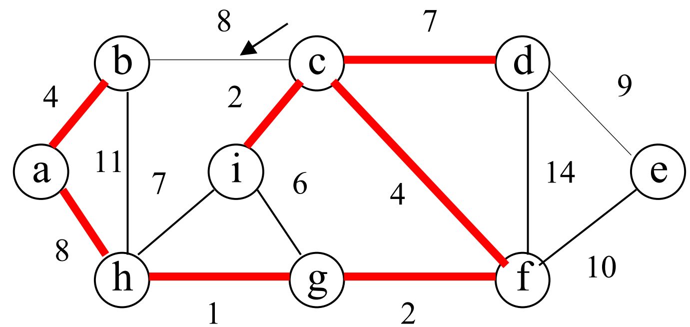


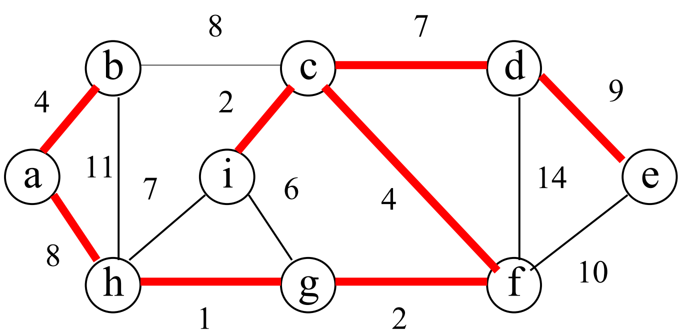
Visualizzazione

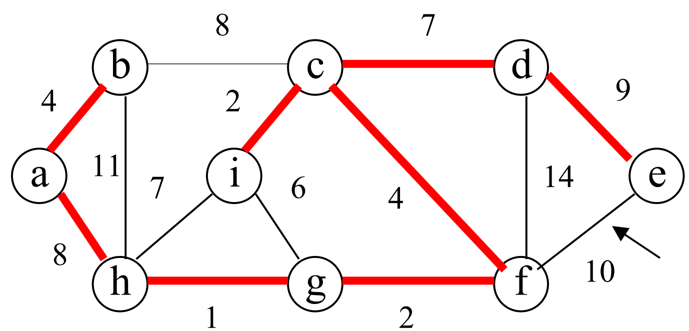


Visualizzazione

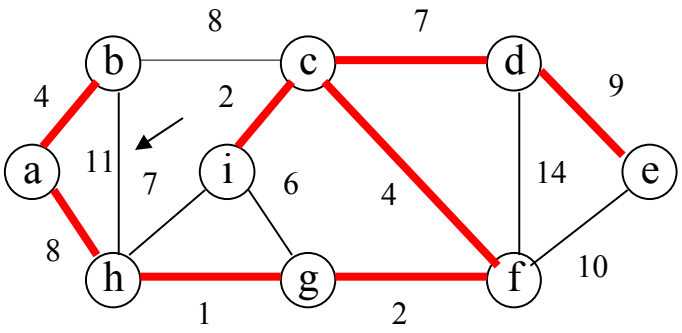


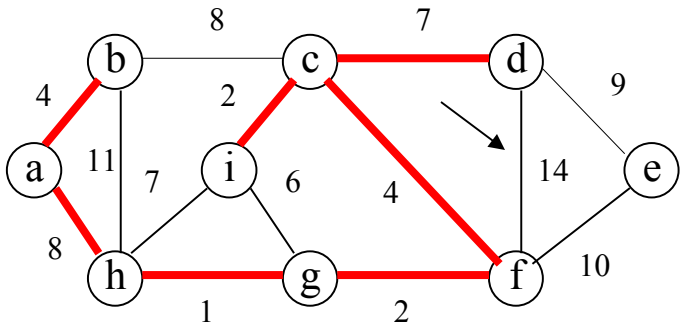






Visualizzazione





Analisi

- Il tempo di esecuzione per l'algoritmo di Kruskal dipende dalla realizzazione della struttura dati per insiemi disgiunti
- Se si utilizza la realizzazione con foreste con le euristiche del rango e della compressione dei cammini:
  - l'inizializzazione richiede O(V)
  - il tempo necessario per ordinare gli archi è O(E lg E)
  - in totale si fanno O(E) operazioni sulla foresta di insiemi disgiunti (O(E) find-set e O(E) union), ovvero complessivamente un tempo O(E)
- In totale il tempo di esecuzione dell'algoritmo di Kruskal è $O(V+E \lg E + E)=O(E \lg E)$

Algoritmo di Prim

- L'algoritmo di Prim procede mantenendo in A un singolo albero (una singola componente connessa)
- L'albero parte da un vertice arbitrario r (la radice) e cresce fino a quando non ricopre tutti i vertici
- Ad ogni passo viene aggiunto un arco leggero che collega un vertice in A con un vertice in V-A
- Per il corollario questi archi sono sicuri per A e quindi quando l'algoritmo termina, in A vi è un albero di copertura minimo
- Anche questo algoritmo è greedy poiché l'albero viene esteso ad ogni passo scegliendo l'arco di peso minimo tra quelli possibili

Algoritmo di Prim

- In questo caso A ha una unica componente connessa che è l'intero albero di copertura che sta crescendo
- Il taglio (A, V-A) rispetta A
- ..e quindi per il corollario qualsiasi arco leggero che attraversa il taglio è un arco sicuro

Algoritmo di Prim

- Per avere un algoritmo efficiente si deve prestare attenzione a come rendere facile la scelta di un nuovo arco da aggiungere ad A
- Questo viene fatto memorizzando tutti i vertici che non sono nell'albero in costruzione in una coda con priorità Q
- Per ogni nodo la priorità è basata su un campo key[v] che contiene il minimo tra i pesi degli archi che collegano v ad un qualunque vertice dell'albero in costruzione
- Per ogni nodo si introduce un campo parent p[x] che serve per poter ricostruire l'albero

Algoritmo di Prim

- L'insieme A è mantenuto implicitamente come

$$A=\{(v,p[v]) : v \text{ in } V - \{r\} - Q\}$$

- Quando l'algoritmo termina Q è vuota e l'albero di copertura in A è dunque:

$$A=\{(v,p[v]) : v \text{ in } V - \{r\}\}$$

Pseudocodice Prim

```
MST-Prim(G,w,r)
1  Q ← V[G]
2  for all u in Q
3  do  key[u] ← ∞
4  key[r] ← 0
5  p[r] ← NIL
6  while Q ≠ ∅
7  do  u ← Extract-Min(Q)
8      for all v in Adj[u]
9      do     if v in Q and w(u,v)<key[v]
10            then   p[v] ← u
11                   key[v] ← w(u,v)
```

Spiegazione pseudocodice

- Le linee 1-4 inizializzano la coda Q con tutti i vertici e pongono a ∞ l'attributo key per ogni vertice
- ad eccezione del vertice r per il quale key[r]=0 in modo da estrarre r come elemento minimo nella fase iniziale
- durante l'algoritmo A e' implicitamente costituito dagli archi (v,p[v]) con v nell'insieme V-Q, cioe' l'albero di copertura in costruzione ha vertici in V-Q
- la linea 7 identifica un vertice u incidente su di un arco leggero che attraversa il taglio (V-Q,Q)
- si elimina u da Q e lo si aggiunge ai vertici dell'albero

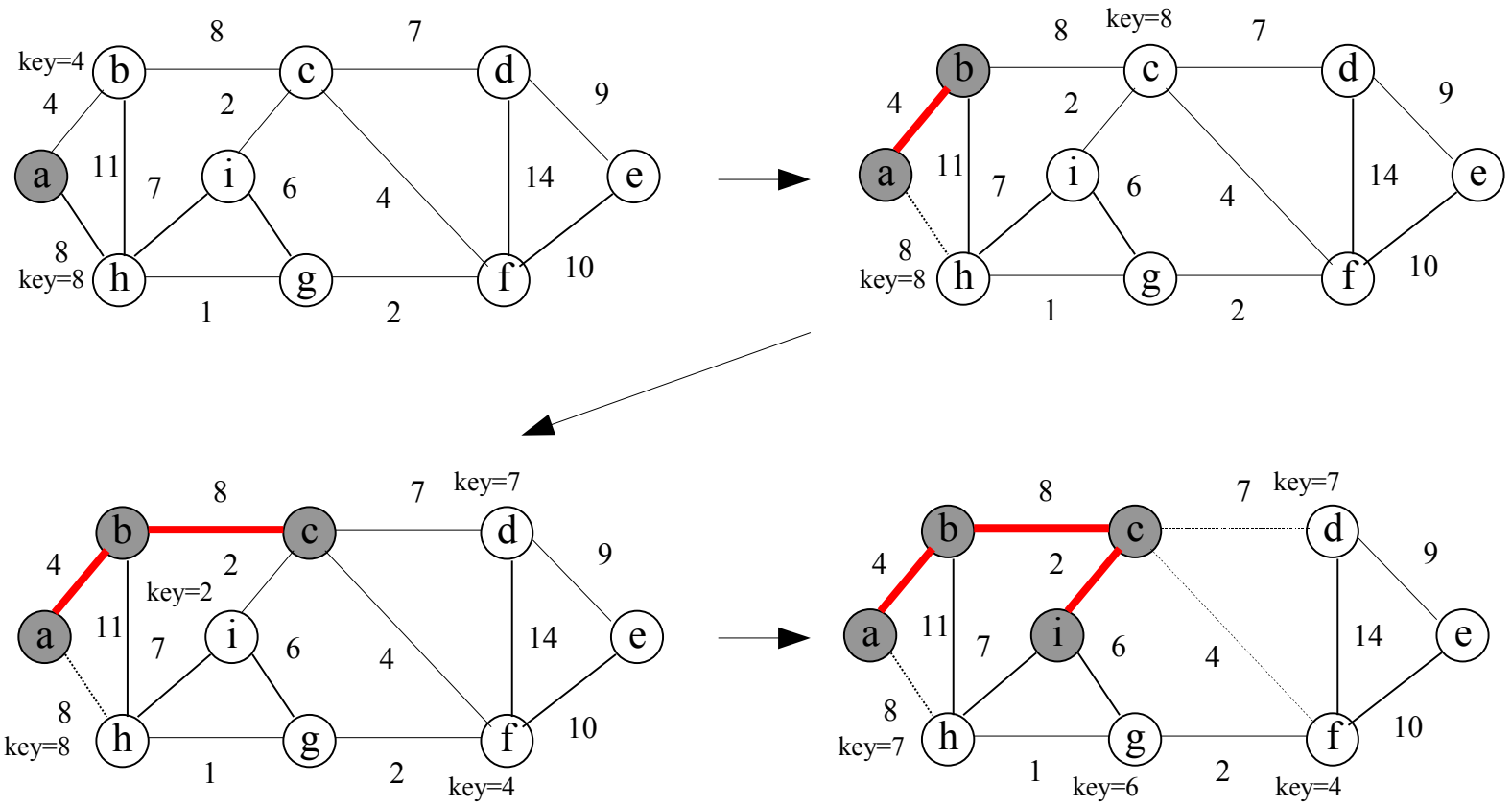
Spiegazione pseudocodice

- Le linee 8-11 aggiornano i campi key e p di ogni vertice v adiacente a u che non appartiene ancora all'albero
- durante l'esecuzione dell'algoritmo il campo key[v] rappresenta sempre il costo minimo tra i pesi degli archi che collegano v ad un qualunque vertice dell'albero in costruzione
- questa proprieta' è preservata perché se si trova un arco che collega v con l'albero di costo inferiore, si aggiorna key al nuovo valore minimo
- durante l'esecuzione dell'algoritmo si aggiorna il valore key dei nodi adiacenti al nodo sotto esame e che verrà inserito nell'albero di copertura

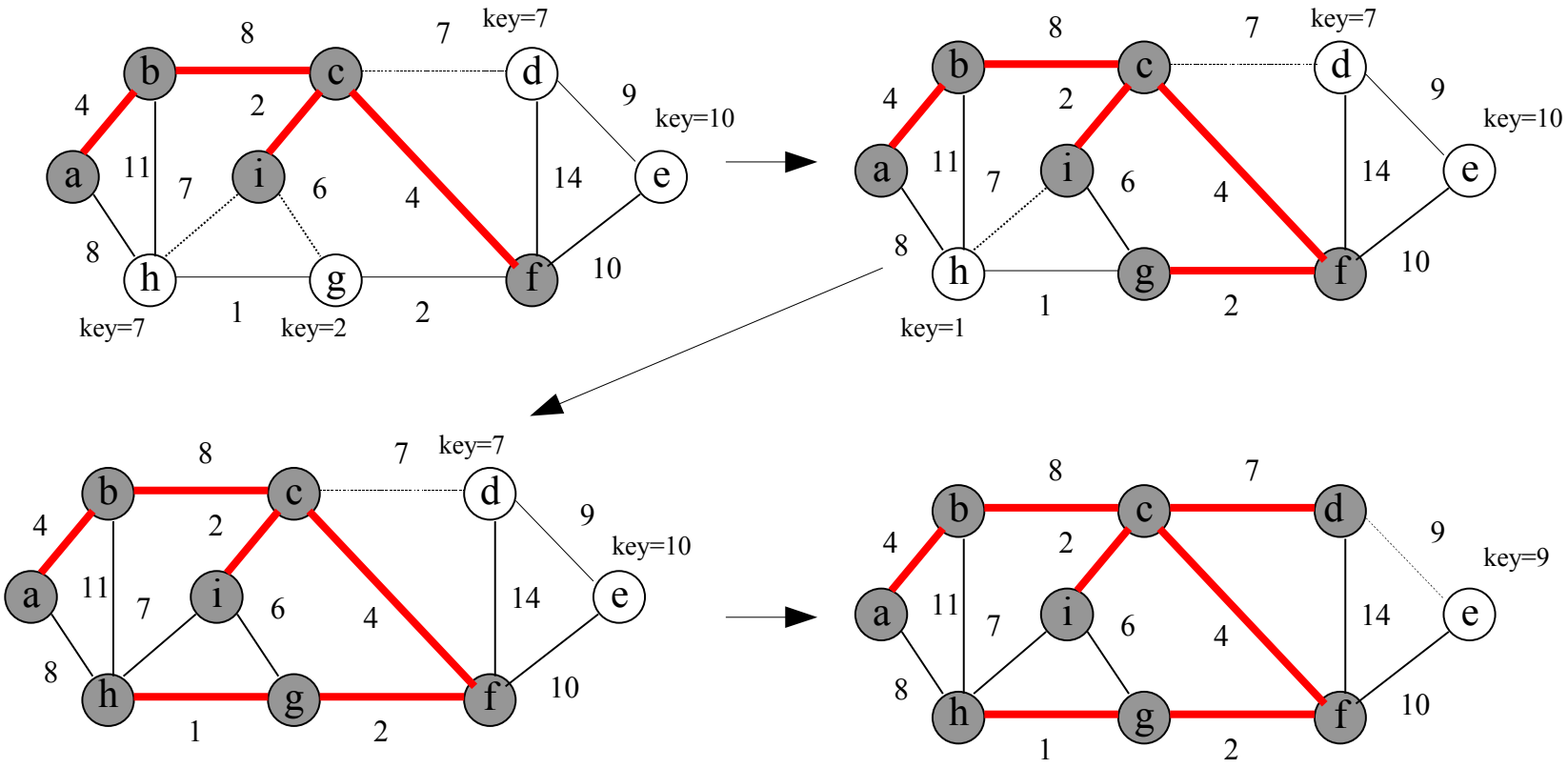
Spiegazione dello pseudocodice

- Al ciclo successivo si esaminerà la coda Q e si troverà che uno dei v esaminati precedentemente è diminuito tanto da essere il vertice con chiave più piccola
- allora si aggiungerà implicitamente v all'albero, fissando la relazione padre-figlio migliore trovata
- e si procederà ad espandere la frontiera dei vertici adiacenti a v, stabilendo nuove potenziali relazioni padre-figlio

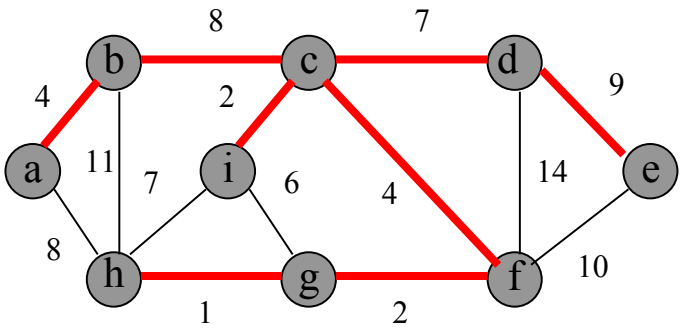
Visualizzazione



Visualizzazione



Visualizzazione



Analisi

- L'efficienza dell'algoritmo di Prim dipende da come viene realizzata la coda con priorità Q
- se Q viene realizzata con uno heap binario:
  - si usa Build-Heap per l'inizializzazione in tempo O(V)
  - il ciclo alla linea 6 viene eseguito |V| volte ed ogni operazione Extract-Min è O(lg V) per un totale di O(V lg V)
  - il ciclo alla linea 8 viene eseguito in tutto O(E) volte perche' la somma delle lunghezze di tutte le liste di adiacenza e' 2|E|
  - il controllo di appartenenza alla linea 9 può essere eseguito in O(1) usando un metodo di indirizzamento diretto
  - l'assegnazione alla linea 11 implica una operazione di Decrease-Key implicita sullo heap che costa O(lg V)
- Il tempo totale è pertanto un O(V+V lg V + E lg V) = O(E lg V) asintoticamente eguale a quello per l'algoritmo di Kruskal