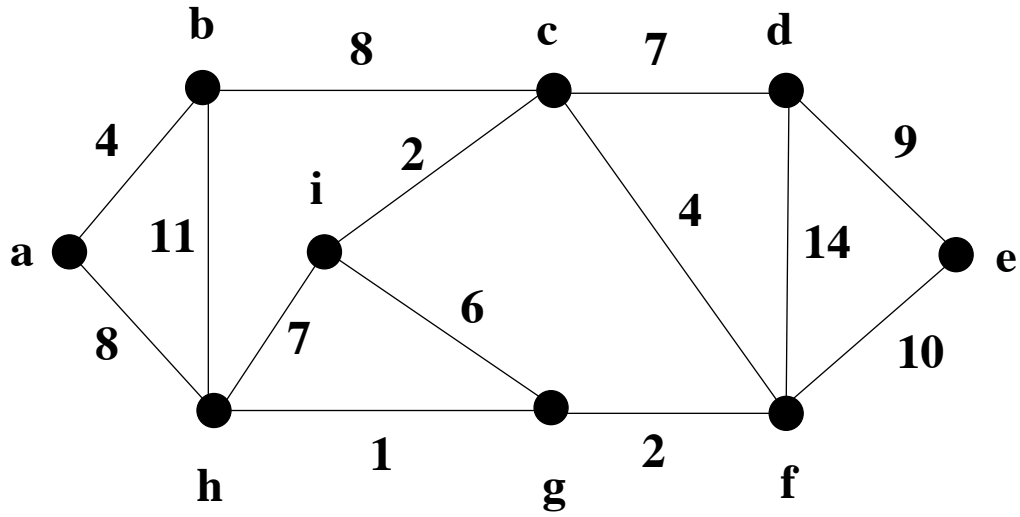
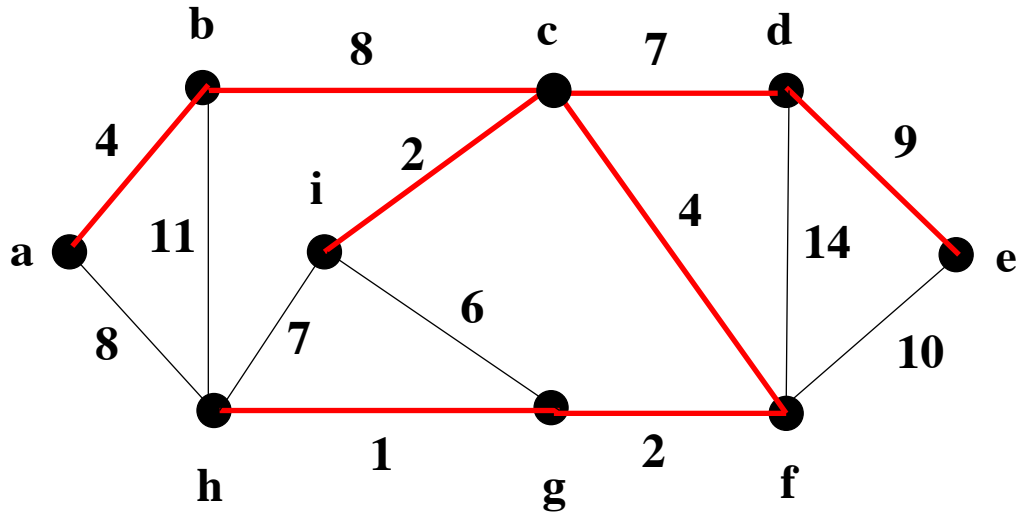
Albero di supporto di costo minimo

Nel problema della distribuzione dell'energia elettrica ci sono diverse case che devono ricevere energia da una centrale elettrica. Per ricevere energia, ogni casa deve essere collegata alla centrale attraverso un cammino fatto di cavi elettrici (che può passare per altre case). L'obiettivo è quello di far arrivare energia a tutte le case minimizzando la quantità di cavi elettrici utilizzati.


b
c
d
8
7
4
2
9
i
4
a
11
14
e
6
8
7
10
1
2
h
g
f


b
c
d
8
7
4
2
9
i
4
a
11
14
e
6
8
7
10
1
2
h
g
f

Grafi pesati

Un grafo pesato è una tripla $G = (V, E, w)$, ove (V, E) è un grafo e $w : E \to \mathbb{R}$ è una funzione peso che associa ad ogni arco un numero reale chiamato **peso** o **costo**.

Possiamo rappresentare un grafo pesato nei seguenti modi:

- liste di adiacenza: l'oggetto v nella lista di adiacenza di u viene esteso con un campo che contiene il costo dell'arco (u, v);
- matrice di adiacenza: l'elemento (i, j) della matrice contiene il costo dell'arco (i, j), se tale arco esiste, oppure NIL, se l'arco non esiste.

Albero di supporto di costo minimo

Sia $G = (V, E, w)$ un grafo pesato **indiretto** e **connesso**.

Un **albero di supporto** (**spanning tree**)

$$T = (V, F)$$

di G è un albero che ha come insieme dei nodi l'insieme V e come insieme degli archi $F \subseteq E$.

Il **costo** di un albero di supporto $T = (V, F)$ è

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

Il problema consiste nel trovare un albero di supporto di G di **costo minimo**. Tale albero non è necessariamente unico.

Schema di algoritmo

L'algoritmo di **Kruskal** e l'algoritmo di **Prim** risolvono entrambi il problema dell'albero di supporto di costo minimo. Entrambi si basano sullo stesso **schema di algoritmo**.

Sia A un insieme di archi tale che esiste almeno un albero di supporto di costo minimo che contiene A. Un arco (u, v) si dice **sicuro** per A se $(u, v) \notin A$ e esiste almeno un albero di supporto di costo minimo che contiene $A \cup \{(u, v)\}$.

Quindi se (u, v) è sicuro per A allora posso aggiungere (u, v) ad A mantenendo invariata la proprietà che esiste un qualche albero di supporto di costo minimo che contiene A.

Lo **schema di algoritmo** è il seguente:

MST(G)

1: $A \leftarrow \emptyset$
2: **while** A non è un albero di supporto **do**
3: trova un arco (u, v) sicuro per A
4: $A \leftarrow A \cup \{(u, v)\}$
5: **end while**
6: **return** A

Gli algoritmi di Kruskal e Prim differiscono per come trovano un arco sicuro (instruzione 3).

Greedy programming

Per trovare un arco sicuro entrambi gli algoritmi di Kruskal e di Prim agiscono in modo **ingordo**.

Un algoritmo ingordo procede facendo in ogni istante la mossa migliore in quel particolare momento.

In altri termini, l'algoritmo ingordo fa tante **scelte ottime locali** sperando in questo modo di ottenere una **soluzione ottima globale**.

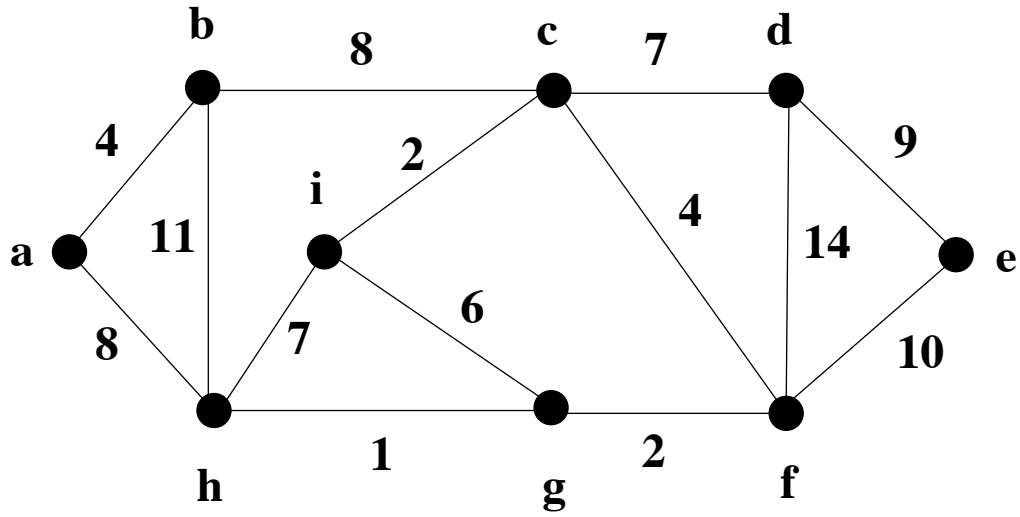
Nel nostro caso questa strategia è ottimale, cioè raggiunge la soluzione ottima. Non sempre una strategia ingorda è ottimale.

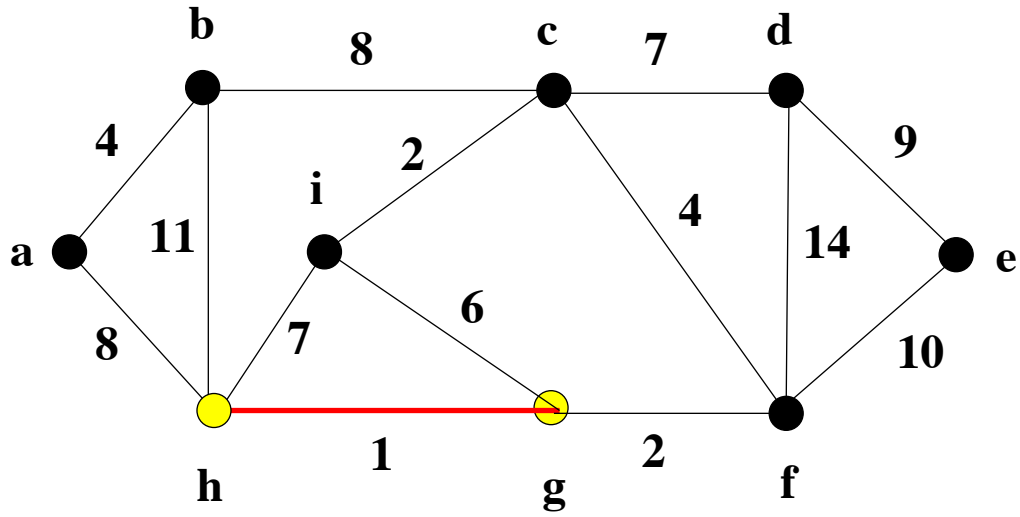
Algoritmo di Kruskal

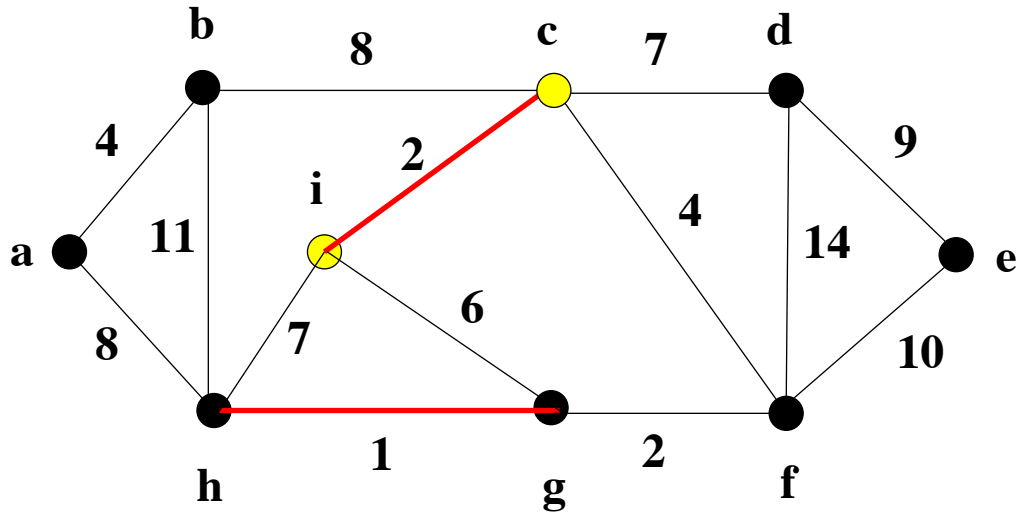
L'algorimo di Kruskal lavora in questo modo:

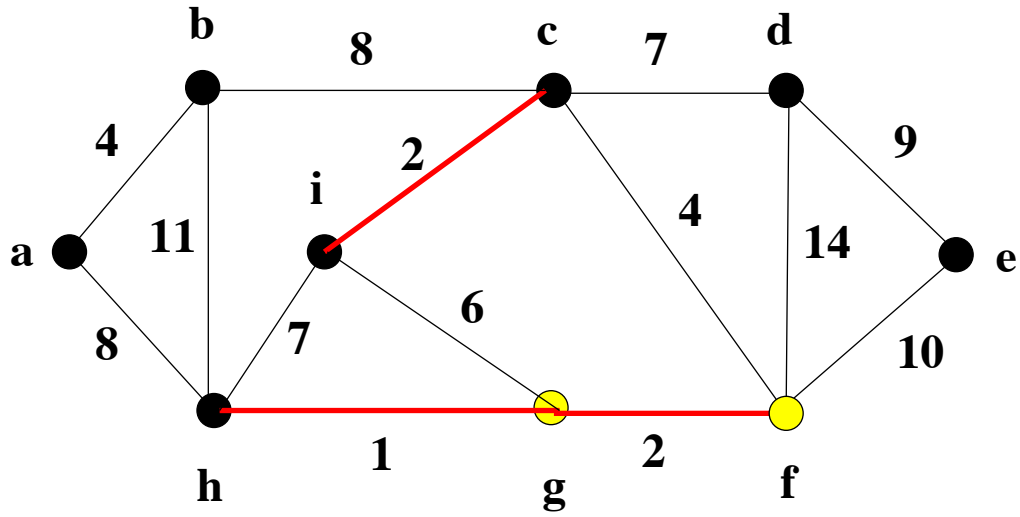
- l'algoritmo mantiene una foresta, cioè un insieme di alberi. Inizialmente ogni albero è un nodo isolato;
- ad ogni passo l'algoritmo sceglie, come arco sicuro, l'arco **più leggero** tra tutti quelli che connettono **alberi differenti**, e lo aggiunge alla foresta. Il numero di alberi della foresta viene così decrementato di uno senza introdurre cicli.
- quando non ci sono più archi che connettono alberi differenti, l'algoritmo termina. La foresta contiene quindi un unico albero. Esso è un **albero di supporto di costo minimo**.

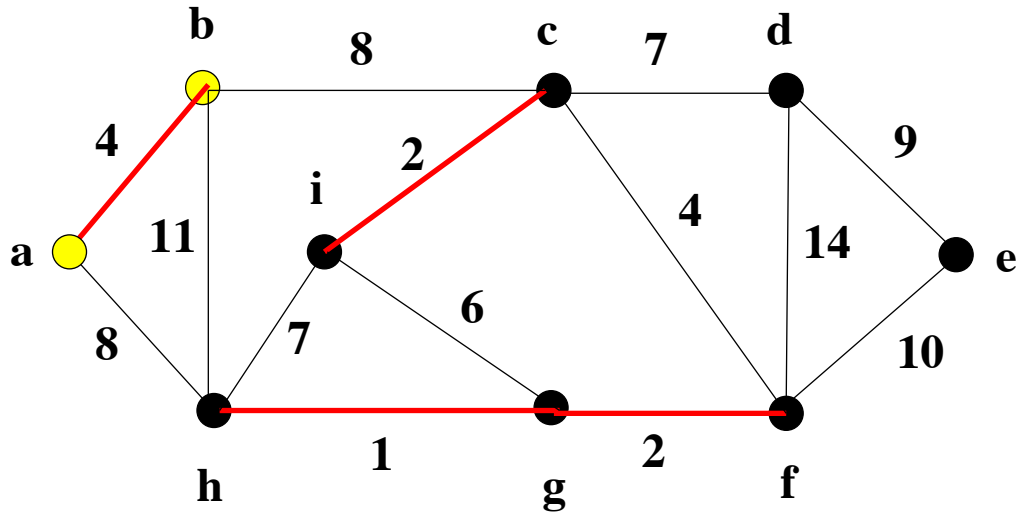
L'algoritmo è **ingordo** in quanto ad ogni passo fa la cosa migliore in quel momento: **aggiungere l'arco più leggero**.

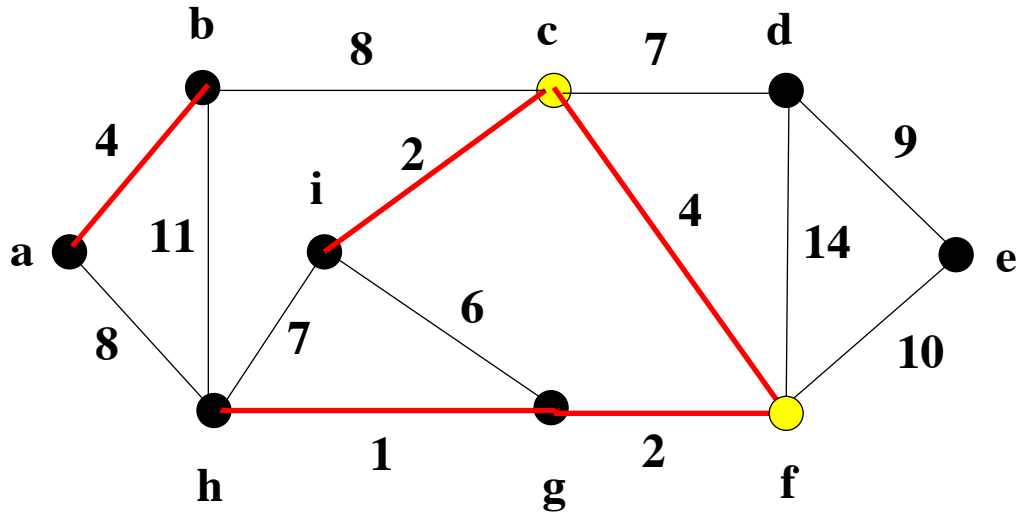

b
c
d
8
7
4
2
9
i
4
a
11
14
e
6
8
7
10
1
2
h
g
f

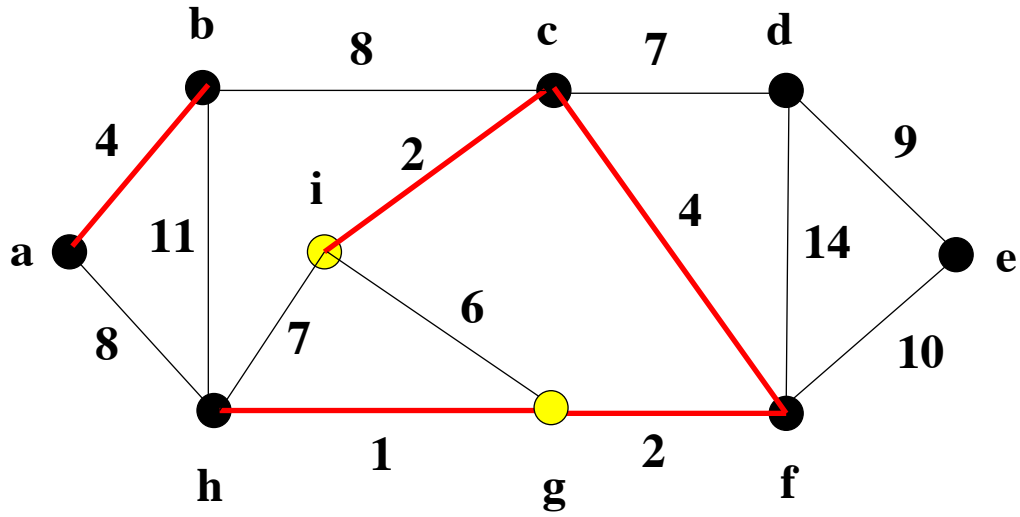

b
8
c
7
d
4
2
9
i
4
a
11
14
e
6
8
7
10
1
2
h
g
f

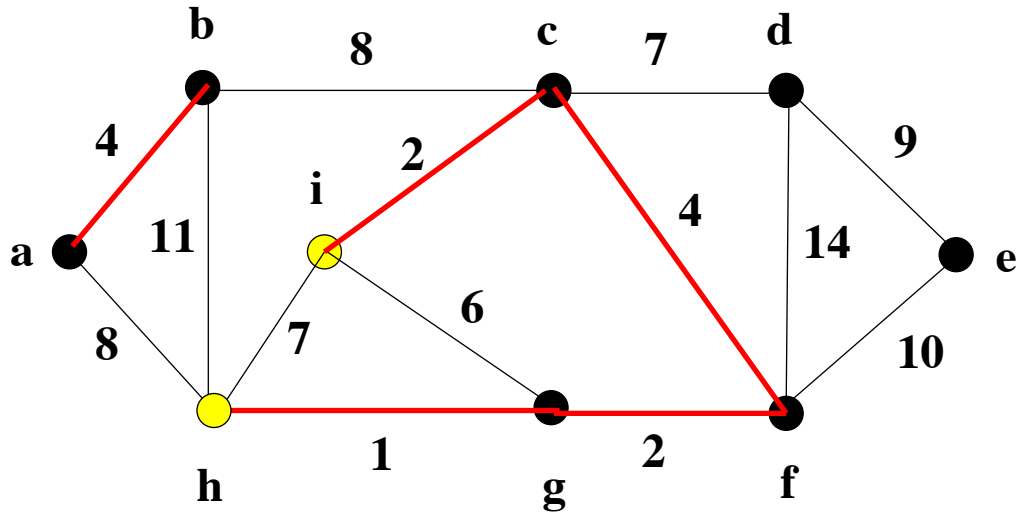

b
8
c
7
d
4
2
9
i
4
a
11
14
e
6
8
7
10
1
2
h
g
f

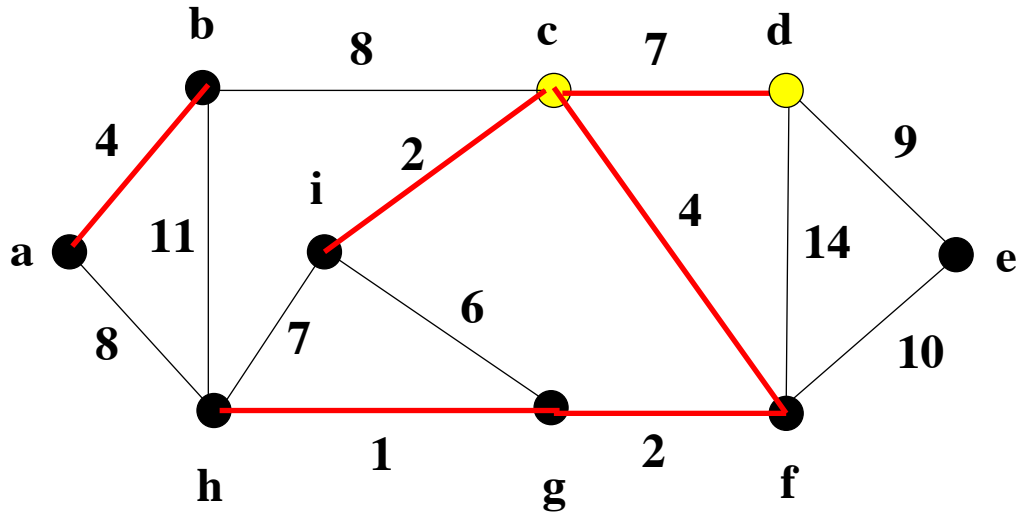

b
8
c
7
d
4
2
9
i
4
a
11
14
e
6
8
7
10
1
2
h
g
f

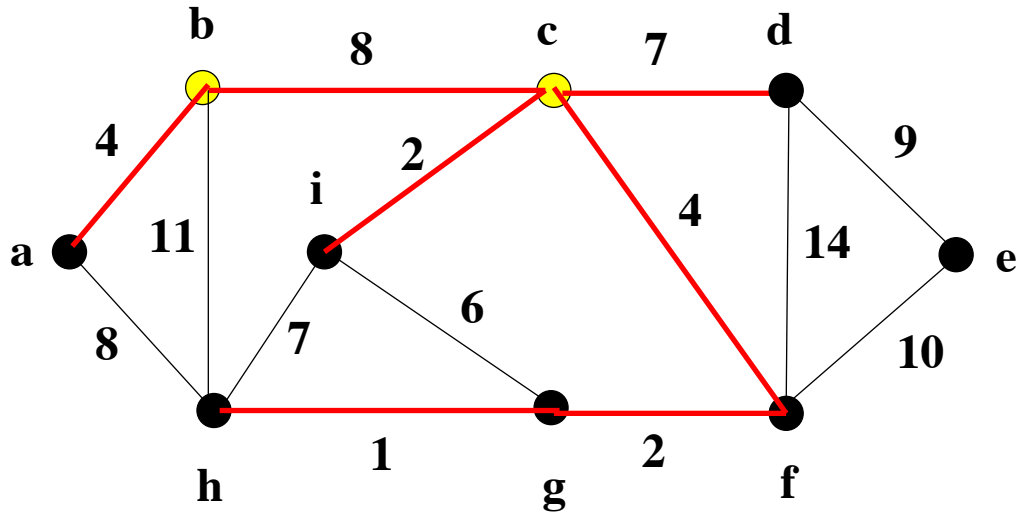

b
c
d
8
7
4
2
9
i
4
a
11
14
e
6
8
7
10
1
2
h
g
f

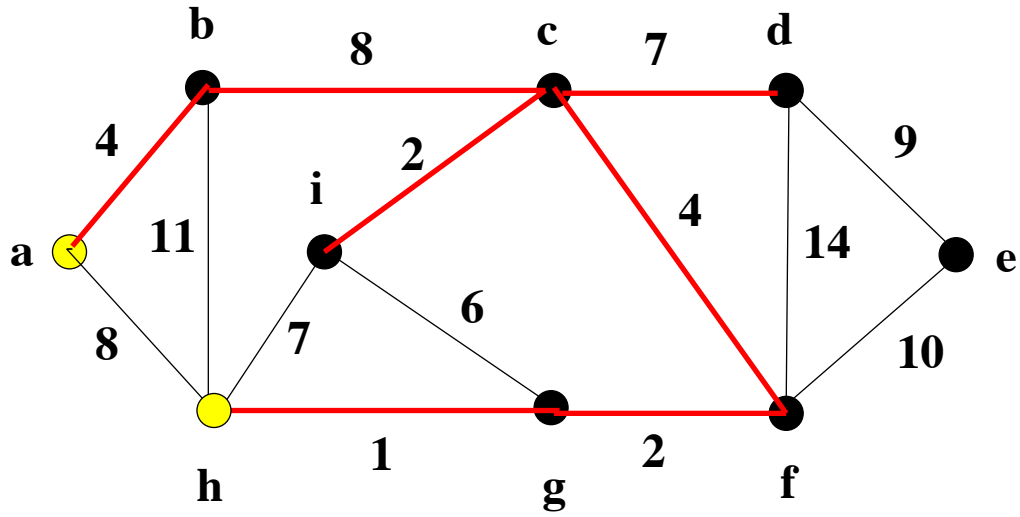

b
8
c
7
d
4
2
9
i
4
a
11
14
e
6
8
7
10
1
2
h
g
f

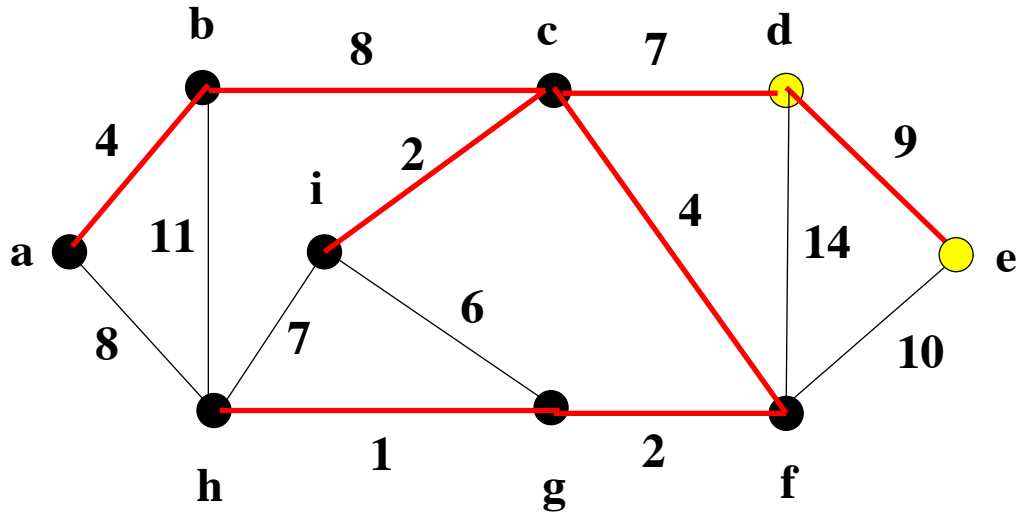

b
8
c
7
d
4
2
9
i
4
a
11
14
e
6
8
7
10
1
2
h
g
f

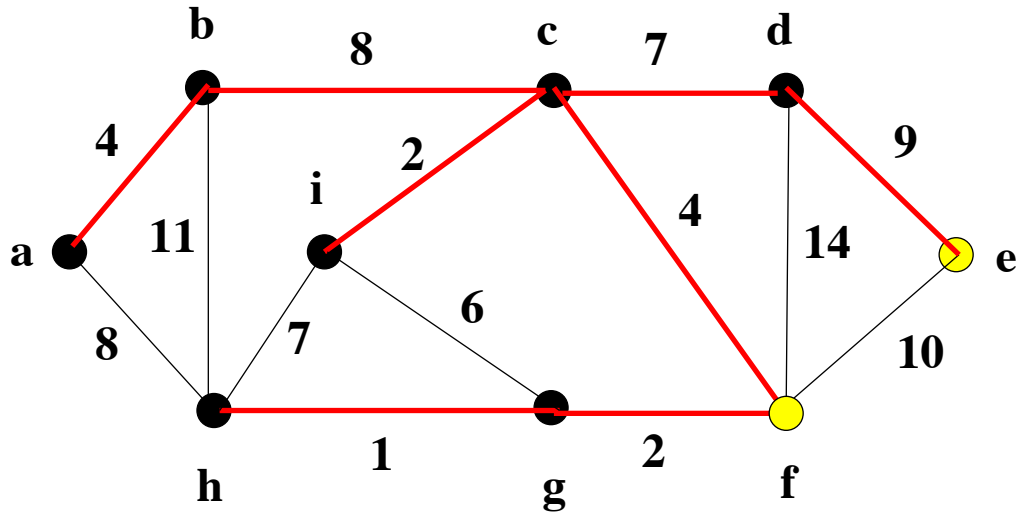

b
c
d
8
7
4
2
9
i
4
a
11
14
e
6
8
7
10
1
2
h
g
f

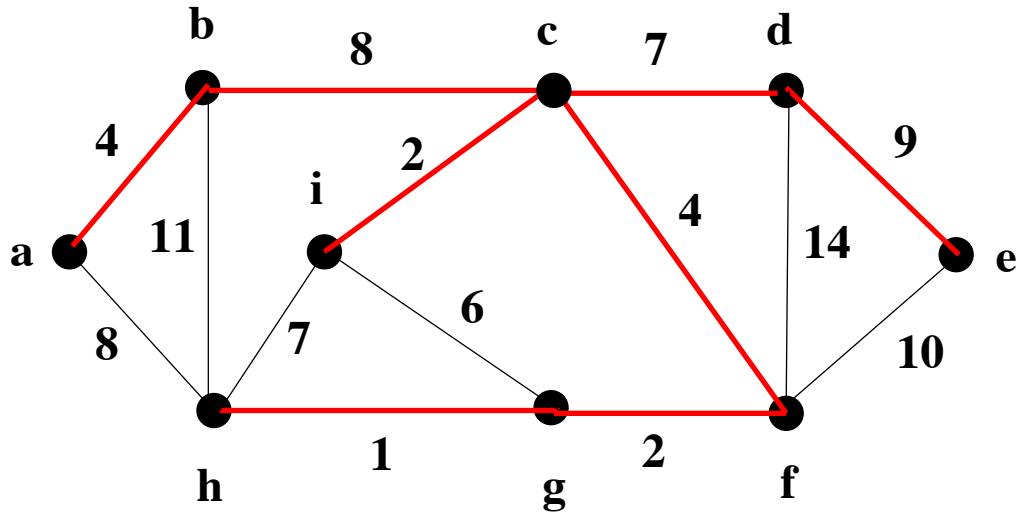

b
8
c
7
d
4
2
9
i
4
a
11
14
e
6
8
7
10
1
2
h
g
f


b
8
c
7
d
4
2
9
i
4
a
11
14
e
6
8
7
10
1
2
h
g
f


b
8
c
7
d
4
2
9
i
4
a
11
14
e
6
8
7
10
1
2
h
g
f


b
8
c
7
d
4
2
9
i
4
a
11
14
e
6
8
7
10
1
2
h
g
f


b
c
d
8
7
4
2
9
i
4
a
11
14
e
6
8
7
10
1
2
h
g
f


b
c
d
8
7
4
2
9
i
4
a
11
14
e
6
8
7
10
1
2
h
g
f

Algoritmo di Kruskal

MST-Kruskal(G)

```
1:  A ← ∅
2:  for each v ∈ V[G] do
3:      MakeSet(v)
4:  end for
5:  HeapSort(E[G], w)
6:  for each (u, v) ∈ E[G] do
7:      if FindSet(u) ≠ FindSet(v) then
8:          A ← A ∪ {(u, v)}
9:          Union(u, v)
10:     end if
11: end for
12: return A
```

Complessità dell'algoritmo di Kruskal

Supponiamo di implementare gli insiemi disgiunti con una rappresentazione ad albero con euristiche di unione per rango e compressione del cammino.

L'algoritmo fa n operazioni di $MakeSet$, $n-1$ operazioni di $Union$, e $2m$ operazioni di $FindSet$.

Dunque il costo totale delle operazioni su insiemi disgiunti è

$$O((n+2m)\cdot\alpha(n)) = O(m\cdot\alpha(n)),$$

in quanto, essendo il grafo connesso, $n-1 \leq m$, cioè $n \leq m+1$.

Il costo dell'ordinamento dell'insieme degli archi è

$$O(m \log m) = O(m \log n),$$

in quanto $m \leq n^2$.

Dunque il costo totale risulta:

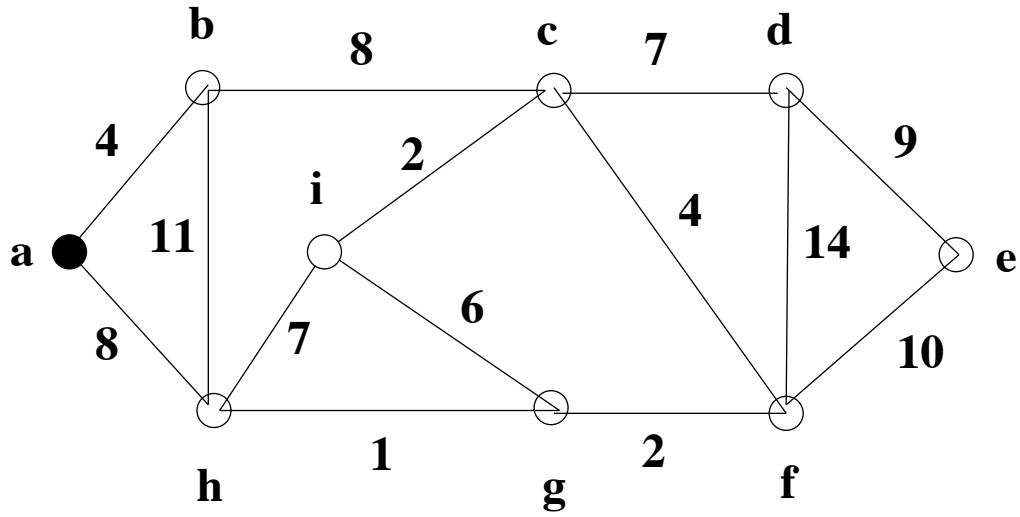
$$O(m \cdot \alpha(n) + m \log n) = O(m \log n),$$

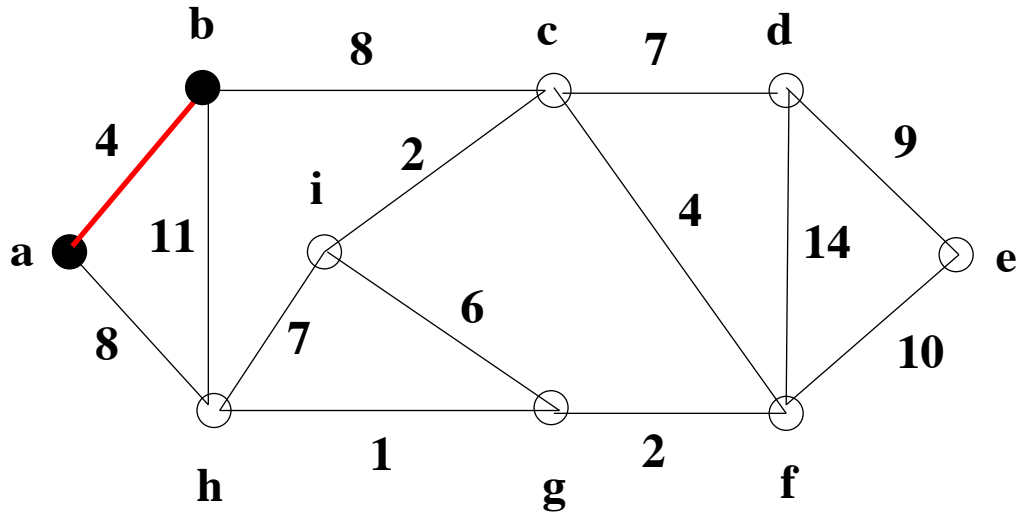
in quanto $\alpha(n) \prec \log n$.

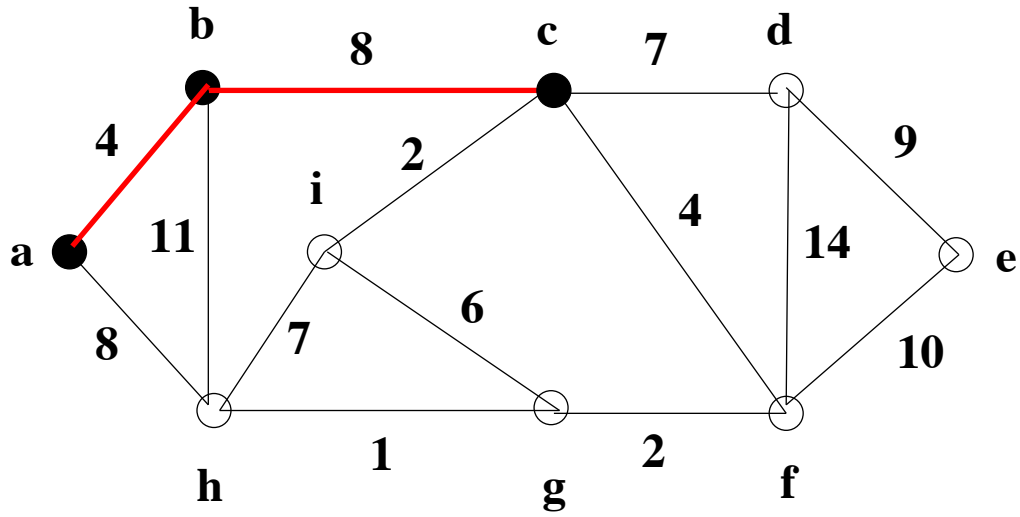
Algoritmo di Prim

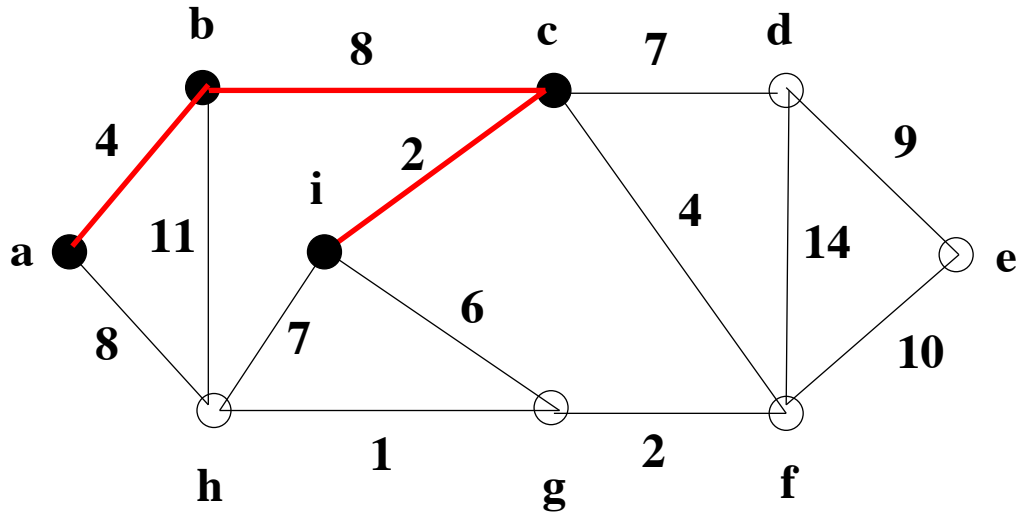
L'algoritmo di Prim lavora in questo modo:

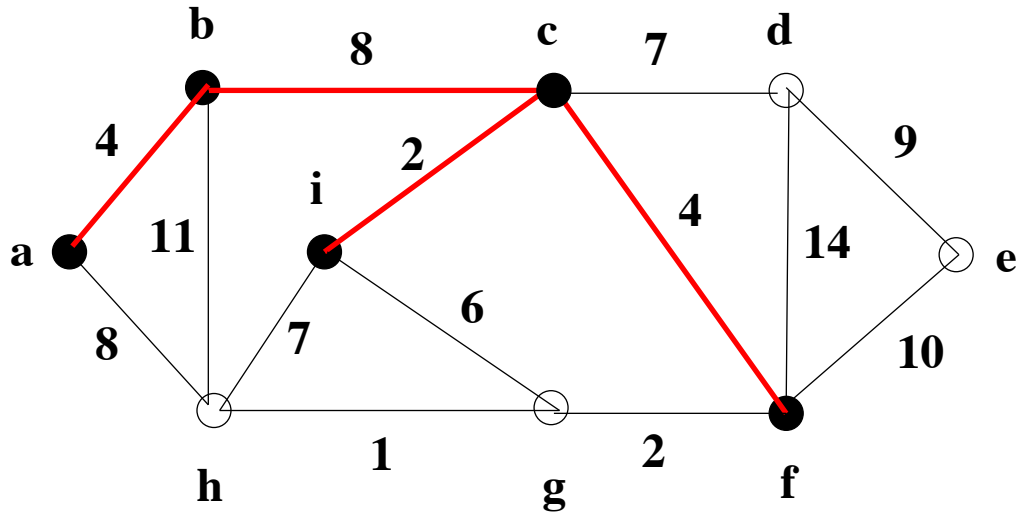
- l'algoritmo parte da un albero che contiene un unico nodo radice;
- ad ogni passo, l'algoritmo aggiunge, come arco sicuro, l'arco **più leggero** che connette un nodo dell'albero costruito con un nodo del grafo non presente nell'albero.
- l'algoritmo termina quando non esistono più nodi del grafo non presenti nell'albero costruito. L'albero così ottenuto è un **albero di supporto di costo minimo**.

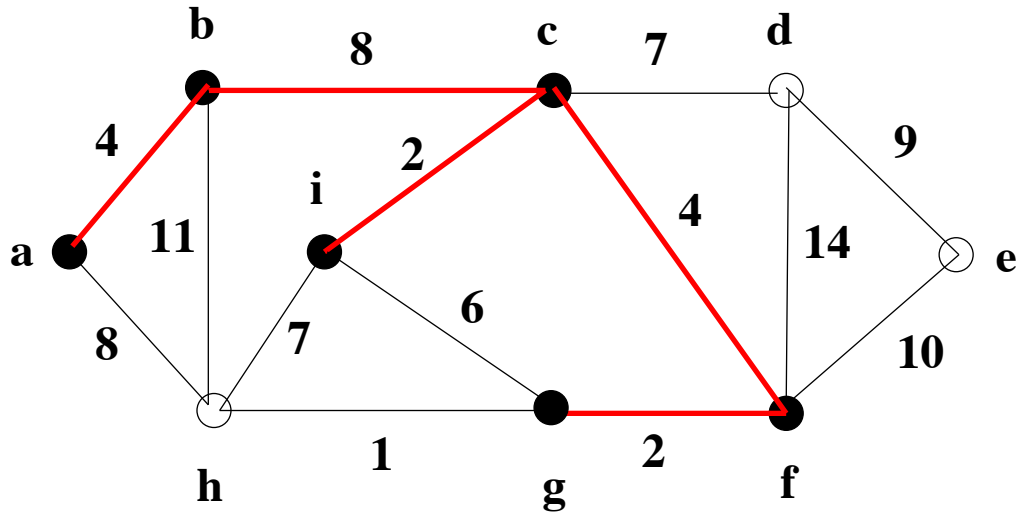

b
8
c
7
d
4
2
9
i
4
a
11
14
e
6
8
7
10
1
2
h
g
f

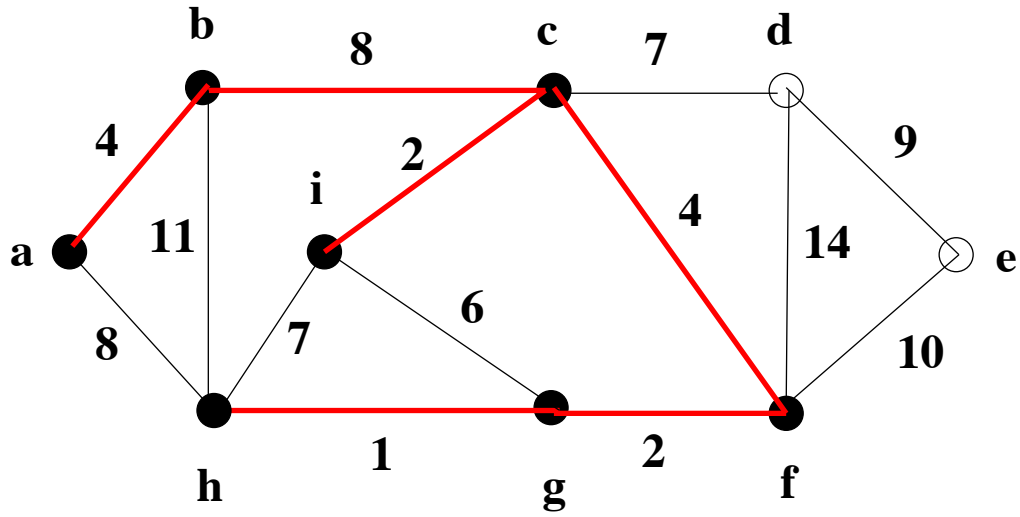

b
8
c
7
d
4
2
9
i
4
a
11
14
e
6
8
7
10
1
2
h
g
f

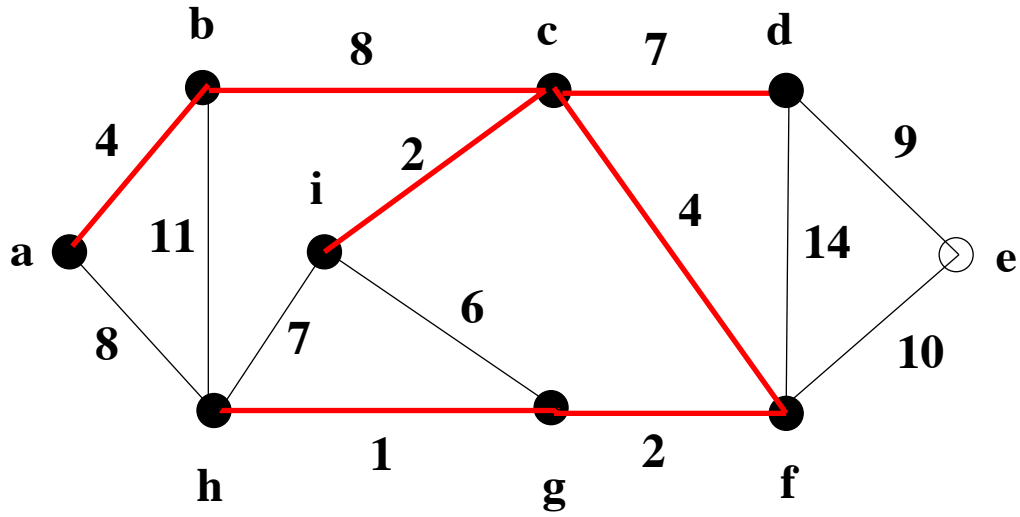

b
8
c
7
d
4
2
9
i
4
a
11
14
e
6
8
7
10
1
2
h
g
f

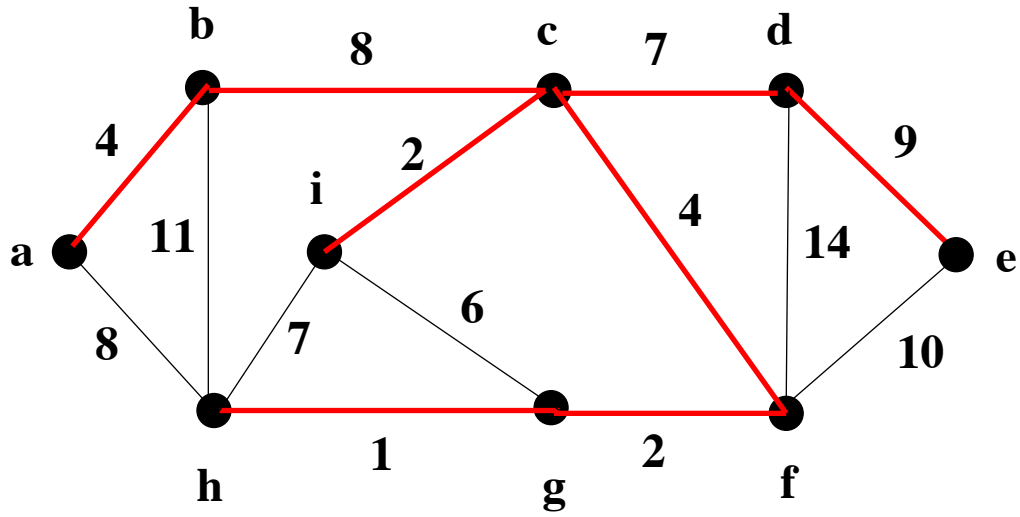

b
8
c
7
d
4
2
9
i
4
a
11
14
e
6
8
7
10
1
2
h
g
f


b
8
c
7
d
4
2
9
i
4
a
11
14
e
6
8
7
10
1
2
h
g
f


b
8
c
7
d
4
2
9
i
4
a
11
14
e
6
8
7
10
1
2
h
g
f


b
8
c
7
d
4
2
9
i
4
a
11
14
e
6
8
7
10
1
2
h
g
f


b
c
d
8
7
4
2
9
i
4
a
11
14
e
6
8
7
10
1
2
h
g
f


b
8
c
7
d
4
2
9
i
4
a
11
14
e
6
8
7
10
1
2
h
g
f

Strutture di dati

Definiamo la **distanza** di un nodo u da un albero T come il peso dell'arco più leggero che connette u ad un nodo nell'albero T. Se non esistono archi che connettono u all'albero, allora la distanza di u vale ∞.

L'algoritmo di Prim usa le seguenti strutture di dati:

- un vettore d tale che $d[u]$ è la distanza del nodo u dall'albero costruito dall'algoritmo;
- un vettore π tale che $\pi[u]$ è il padre di u nell'albero costruito dall'algoritmo, oppure è NIL se tale padre non esiste;
- Una coda con priorità basato sul **minimo** Q che contiene i nodi del grafo che non appartengono all'albero costruito dall'algoritmo. La **priorità** dei nodi è la loro **distanza**.

MST-Prim(G,r)

```
 1: for each u ∈ V[G] do
 2:     d[u] ← ∞
 3:     π[u] ← NIL
 4: end for
 5: d[r] ← 0
 6: Q ← V[G]
 7: while Q ≠ ∅ do
 8:     u ← ExtractMin(Q)
 9:     for all v ∈ Adj[u] do
10:         if v ∈ Q and w(u, v) < d[v] then
11:             d[v] ← w(u, v)
12:             π[v] ← u
13:         end if
14:     end for
15: end while
```

Dettagli implementativi

La coda con priorità basata sul minimo Q può essere implementata usando un **min-heap**, cioè uno heap in cui ogni nodo ha chiave minore di quella dei propri figli.

Ricordiamo che:

- E' possibile costruire un min-heap di n nodi in tempo $\Theta(n)$;
- le operazioni di inserimento, di estrazione del minimo e di modifica della priorità in un min-heap hanno costo $\Theta(\log n)$.

Occorre mantenere una corrispondenza tra i **nodi nel grafo** e i corrispondenti **nodi nella coda** con Q. A tal fine introduciamo le seguenti strutture di dati:

- un vettore pos tale che $pos[u]$ è la posizione del nodo u nella coda Q;
- un vettore $flag$ tale che $flag[u] = \text{TRUE}$ se il nodo $u \in Q$ e $flag[u] = \text{FALSE}$ se il nodo $u \notin Q$;

- l'istruzione $Q \leftarrow V[G]$ costruisce un min-heap Q inserendo ogni nodo u del grafo G, assegnando $flag[u]$ a TRUE e inserendo in $pos[u]$ la posizione del nodo u nella coda Q;
- l'istruzione $u \leftarrow ExtractMin(Q)$ rimuove il minimo u dalla coda Q e assegna $flag[u]$ a FALSE;
- il test $v \in Q$ si riduce a verificare se $flag[v] =$ TRUE;
- l'istruzione $d[v] \leftarrow w(u, v)$ deve modificare la priorità dell'elemento in posizione $pos[v]$ nella coda Q con il valore $w(u, v)$.

MST-Prim(G,r)

```
 1: for each u ∈ V[G] do
 2:     d[u] ← ∞
 3:     π[u] ← NIL
 4: end for
 5: d[r] ← 0
 6: Q ← V[G]
 7: while Q ≠ ∅ do
 8:     u ← ExtractMin(Q)
 9:     for all v ∈ Adj[u] do
10:         if v ∈ Q and w(u, v) < d[v] then
11:             d[v] ← w(u, v)
12:             π[v] ← u
13:         end if
14:     end for
15: end while
```

Complessità dell'algoritmo di Prim

Sia n il numero di nodi e m il numero di archi del grafo.

- L'inizializzazione (linee 1–5) costa $\Theta(n)$;
- L'istruzione $Q \leftarrow V[G]$ costa $\Theta(n)$;
- Il ciclo while viene eseguito n volte. Ogni operazione di estrazione del minimo costa $\Theta(\log n)$. Quindi tutte le estrazioni costano $\Theta(n \log n)$;
- L'operazione di modifica della priorità costa $\Theta(\log n)$ e viene fatto per ogni arco. Quindi tutte le modifiche di priorità costano $\Theta(m \log n)$.

La complessità totale risulta (come per l'algoritmo di Kruskal):

$$\Theta(2n + n \log n + m \log n) = \Theta(m \log n)$$