Lezione 27

Sommario della Lezione

- Alberi ricoprenti di peso minimo

Immaginiamo di avere n postazioni $V = \{v_1, v_2, \ldots, v_n\}$ e vogliamo costruire una rete di comunicazione su di essi,

Immaginiamo di avere n postazioni $V = \{v_1, v_2, \ldots, v_n\}$ e vogliamo costruire una rete di comunicazione su di essi, con i seguenti requisiti:

1. La rete deve essere connessa (cosicchè sia possibile raggiungere ogni nodo da ogni altro nodo)

Immaginiamo di avere n postazioni $V = \{v_1, v_2, \ldots, v_n\}$ e vogliamo costruire una rete di comunicazione su di essi, con i seguenti requisiti:

1. La rete deve essere connessa (cosicchè sia possibile raggiungere ogni nodo da ogni altro nodo)
2. Vogliamo spendere il meno possibile (assumiamo che stabilire una connessione tra due postazioni ci costi qualcosa)

Immaginiamo di avere n postazioni $V = \{v_1, v_2, \ldots, v_n\}$ e vogliamo costruire una rete di comunicazione su di essi, con i seguenti requisiti:

1. La rete deve essere connessa (cosicchè sia possibile raggiungere ogni nodo da ogni altro nodo)
2. Vogliamo spendere il meno possibile (assumiamo che stabilire una connessione tra due postazioni ci costi qualcosa)

È possibile stabilire una connessione tra alcune coppie (v_i, v_j) di locazioni (non è detto che sia possibile stabilire la connessione tra tutte le coppie), ad un costo pari a $c(v_i, v_j) \geq 0$.

Immaginiamo di avere n postazioni $V = \{v_1, v_2, \ldots, v_n\}$ e vogliamo costruire una rete di comunicazione su di essi, con i seguenti requisiti:

1. La rete deve essere connessa (cosicchè sia possibile raggiungere ogni nodo da ogni altro nodo)
2. Vogliamo spendere il meno possibile (assumiamo che stabilire una connessione tra due postazioni ci costi qualcosa)

È possibile stabilire una connessione tra alcune coppie (v_i, v_j) di locazioni (non è detto che sia possibile stabilire la connessione tra tutte le coppie), ad un costo pari a $c(v_i, v_j) \geq 0$.

Possiamo quindi rappresentare l'insieme delle possibili connessioni che potremmo realizzare mediante un grafo $G = (V, E)$ (dove i vertici rappresentano le locazioni e gli archi le connessioni che si possono stabilire tra connessioni).

Immaginiamo di avere n postazioni $V = \{v_1, v_2, \ldots, v_n\}$ e vogliamo costruire una rete di comunicazione su di essi, con i seguenti requisiti:

1. La rete deve essere connessa (cosicchè sia possibile raggiungere ogni nodo da ogni altro nodo)
2. Vogliamo spendere il meno possibile (assumiamo che stabilire una connessione tra due postazioni ci costi qualcosa)

È possibile stabilire una connessione tra alcune coppie (v_i, v_j) di locazioni (non è detto che sia possibile stabilire la connessione tra tutte le coppie), ad un costo pari a $c(v_i, v_j) \geq 0$.

Possiamo quindi rappresentare l'insieme delle possibili connessioni che potremmo realizzare mediante un grafo $G = (V, E)$ (dove i vertici rappresentano le locazioni e gli archi le connessioni che si possono stabilire tra connessioni). Inoltre, ciascun arco $e = (u, v)$ ha un costo $c(e) \geq 0$ ad esso associato.

Abbiamo quindi il seguente problema algoritmico: Minimo Sottografo Connesso Ricoprente.

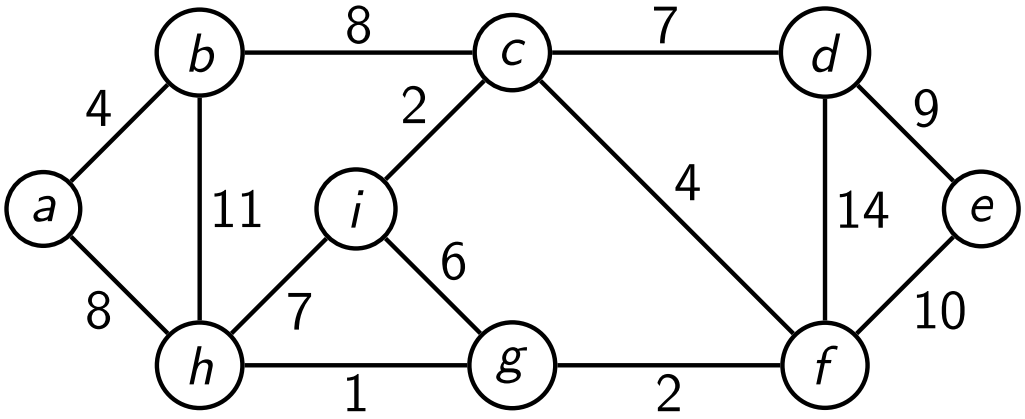
Abbiamo quindi il seguente problema algoritmico: Minimo Sottografo Connesso Ricoprente.

Input al problema: Grafo $G = (V, E)$, costi $c(e) \geq 0$ per ogni arco $e \in E$

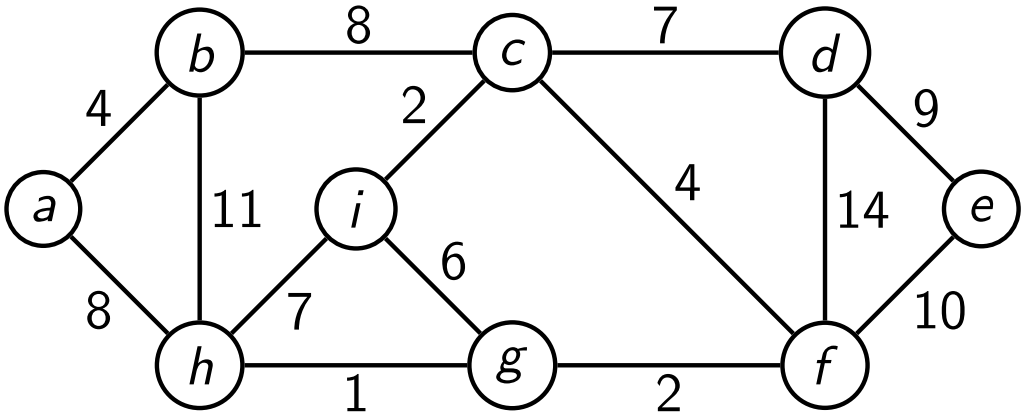
Abbiamo quindi il seguente problema algoritmico: Minimo Sottografo Connesso Ricoprente.

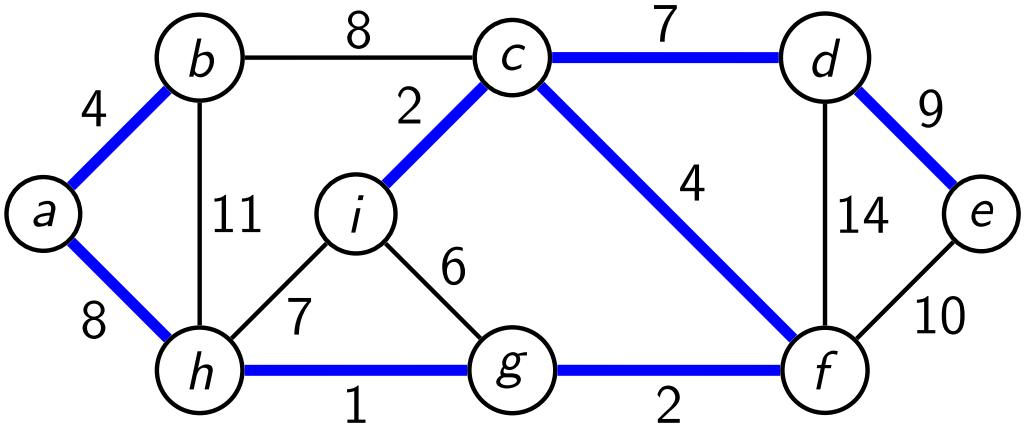
Input al problema: Grafo $G = (V, E)$, costi $c(e) \geq 0$ per ogni arco $e \in E$
Output al problema: Sottoinsieme di archi $T \subseteq E$ tali che il sottografo (V, T) sia connesso ed il costo totale $\sum_{e \in T} c(e)$ sia il più piccolo possibile.

Esempio:

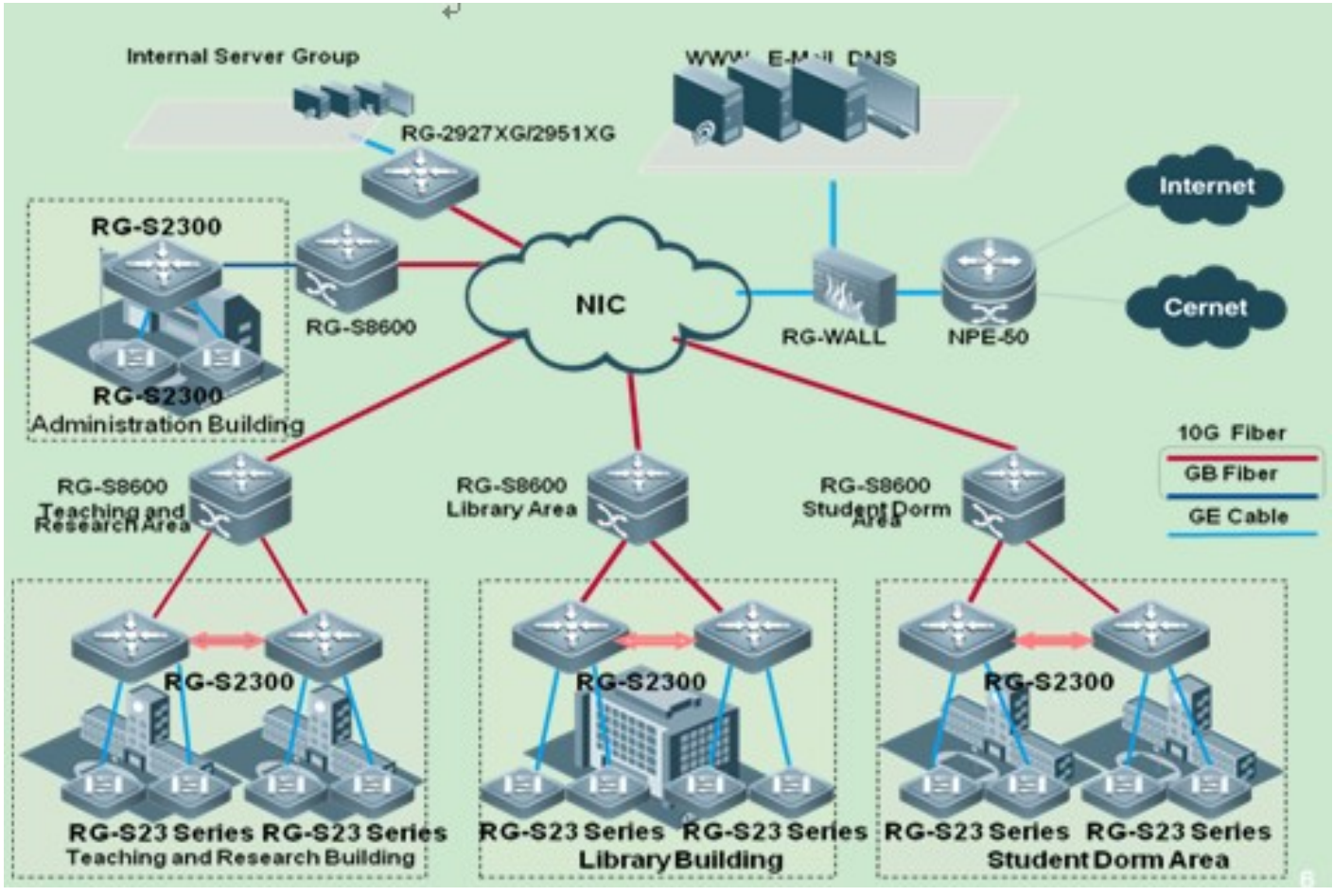


Esempio:

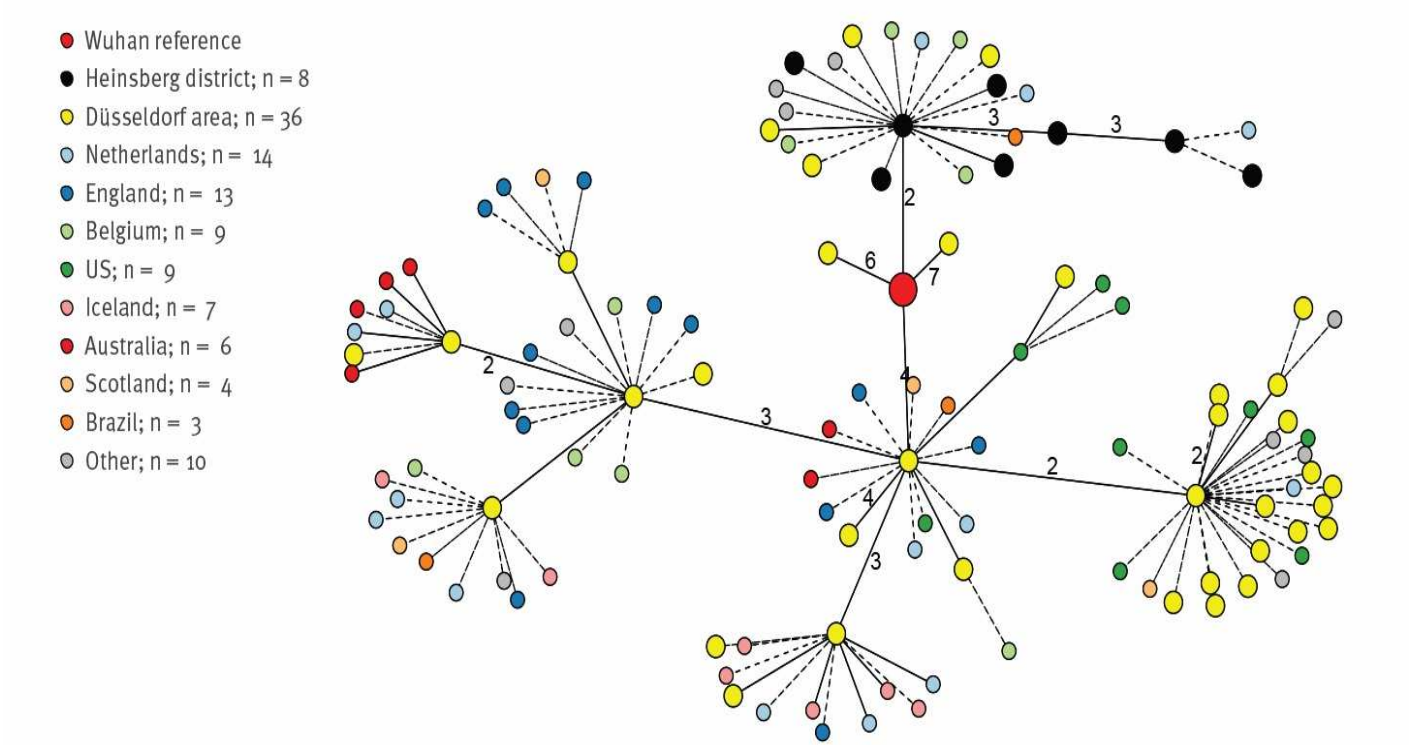

a
b
c
d
e
f
g
h
i
4
8
8
7
2
11
4
9
14
6
7
10
1
2


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

Rete di comunicazione

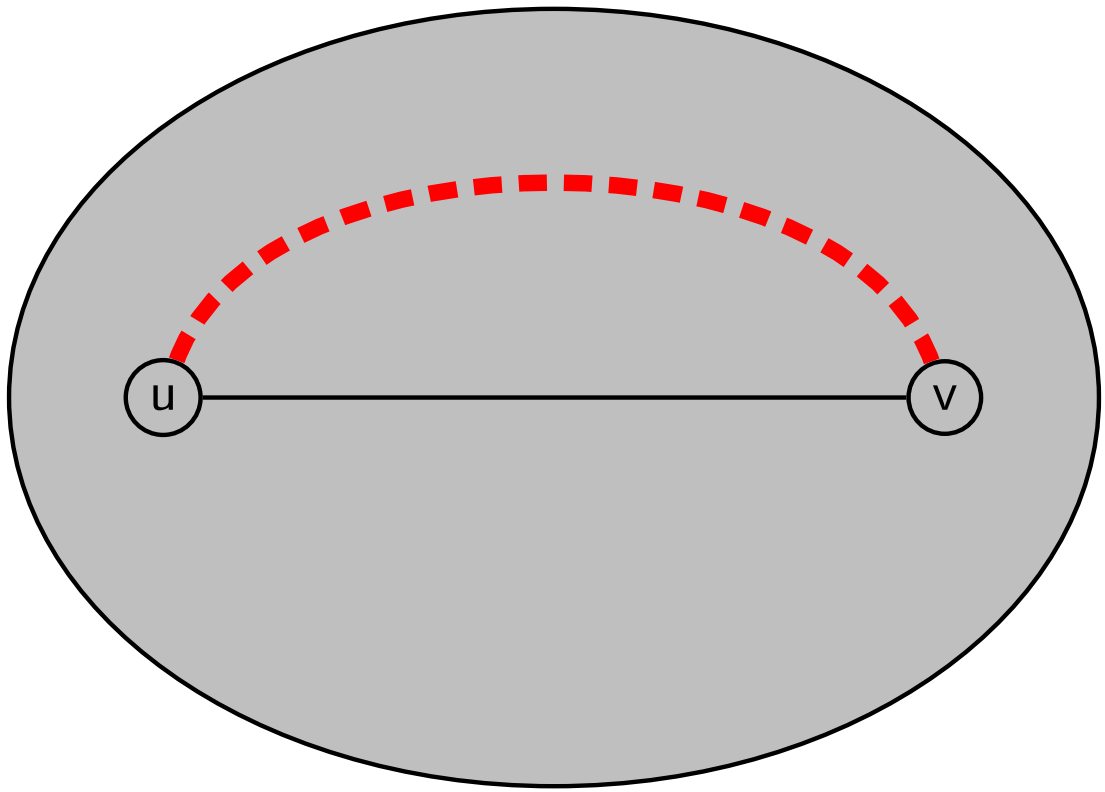


Qui vi è un sottografo connesso costruito sui genotipi (nodi) del SARS-COVID 19 con la lunghezza di ogni arco proporzionale alla differenza genomica tra di essi

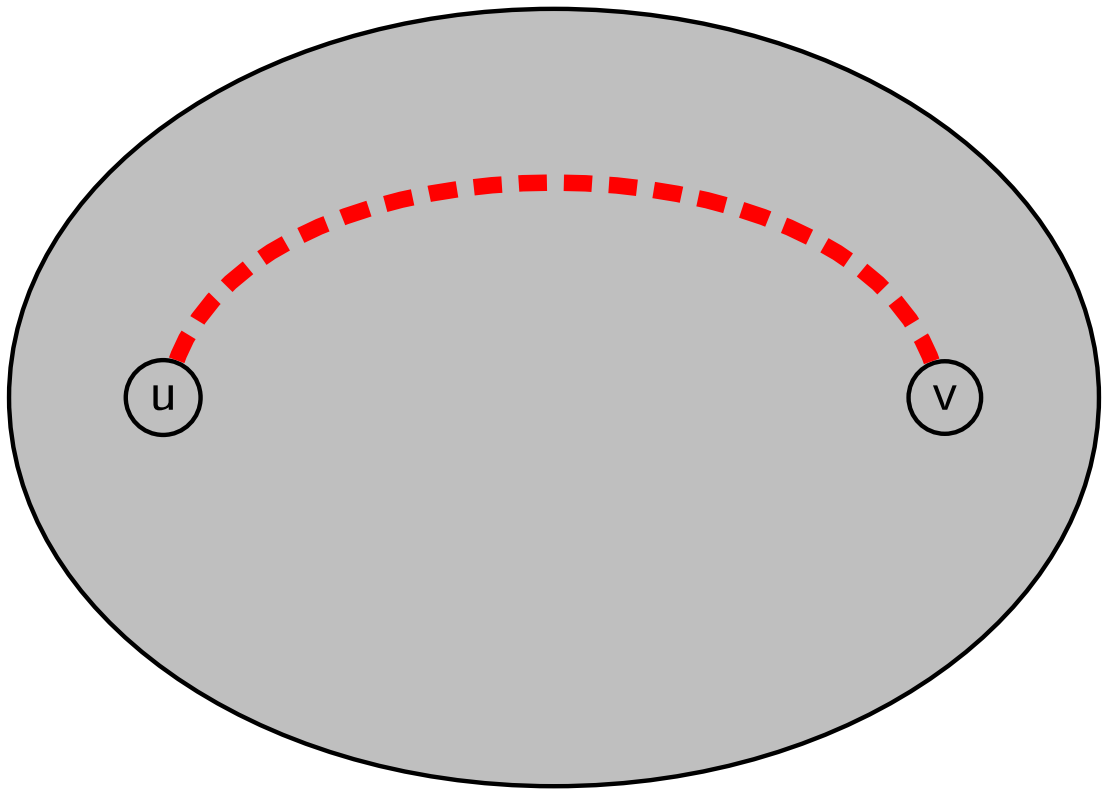


Fatto 1. Esiste una soluzione T di minimo costo al problema sopra esposto in cui (V, T) è un albero.

Fatto 1. Esiste una soluzione T di minimo costo al problema sopra esposto in cui (V, T) è un albero.



Fatto 1. Esiste una soluzione T di minimo costo al problema sopra esposto in cui (V, T) è un albero.



Prova. Sia T una soluzione di minimo costo.

Prova. Sia T una soluzione di minimo costo. Per definizione (V, T) é connesso.

Prova. Sia T una soluzione di minimo costo. Per definizione (V, T) é connesso. Se (V, T) é un albero siamo a posto.

Prova. Sia T una soluzione di minimo costo. Per definizione (V, T) é connesso. Se (V, T) é un albero siamo a posto. Se (V, T) non é un albero, allora contiene un ciclo C.

Prova. Sia T una soluzione di minimo costo. Per definizione (V, T) é connesso. Se (V, T) é un albero siamo a posto. Se (V, T) non é un albero, allora contiene un ciclo C. Eliminando un qualsiasi arco (u, v) dal ciclo C, il sottografo $(V, T' = T - \{(u, v)\})$ rimane connesso in quanto sará sempre possibile andare da u a v seguendo la "via alternativa" che rimane del ciclo C.

Prova. Sia T una soluzione di minimo costo. Per definizione (V, T) é connesso. Se (V, T) é un albero siamo a posto. Se (V, T) non é un albero, allora contiene un ciclo C. Eliminando un qualsiasi arco (u, v) dal ciclo C, il sottografo $(V, T' = T - \{(u, v)\})$ rimane connesso in quanto sará sempre possibile andare da u a v seguendo la "via alternativa" che rimane del ciclo C. La soluzione T' in questo modo ottenuta ha un costo $\leq$ del costo di T.

Prova. Sia T una soluzione di minimo costo. Per definizione (V, T) é connesso. Se (V, T) é un albero siamo a posto. Se (V, T) non é un albero, allora contiene un ciclo C. Eliminando un qualsiasi arco (u, v) dal ciclo C, il sottografo $(V, T' = T - \{(u, v)\})$ rimane connesso in quanto sará sempre possibile andare da u a v seguendo la "via alternativa" che rimane del ciclo C. La soluzione T' in questo modo ottenuta ha un costo $\leq$ del costo di T.Procedendo per via di eliminazione di cicli, otterremo alla fine un albero $(V, \overline{T})$ con costo di $\overline{T} \leq$ costo di T.

Prova. Sia T una soluzione di minimo costo. Per definizione (V, T) é connesso. Se (V, T) é un albero siamo a posto. Se (V, T) non é un albero, allora contiene un ciclo C. Eliminando un qualsiasi arco (u, v) dal ciclo C, il sottografo $(V, T' = T - \{(u, v)\})$ rimane connesso in quanto sará sempre possibile andare da u a v seguendo la "via alternativa" che rimane del ciclo C. La soluzione T' in questo modo ottenuta ha un costo $\leq$ del costo di T.Procedendo per via di eliminazione di cicli, otterremo alla fine un albero $(V, \overline{T})$ con costo di $\overline{T} \leq$ costo di T. Tutto ció implica che anche $\overline{T}$ é di costo minimo

Prova. Sia T una soluzione di minimo costo. Per definizione (V, T) é connesso. Se (V, T) é un albero siamo a posto. Se (V, T) non é un albero, allora contiene un ciclo C. Eliminando un qualsiasi arco (u, v) dal ciclo C, il sottografo $(V, T' = T - \{(u, v)\})$ rimane connesso in quanto sará sempre possibile andare da u a v seguendo la "via alternativa" che rimane del ciclo C. La soluzione T' in questo modo ottenuta ha un costo $\leq$ del costo di T.Procedendo per via di eliminazione di cicli, otterremo alla fine un albero $(V, \overline{T})$ con costo di $\overline{T} \leq$ costo di T. Tutto ció implica che anche $\overline{T}$ é di costo minimo ma questa volta $(V, \overline{T})$ é *finalmente* un albero.

Intuizione per un possibile algoritmo greedy.

Intuizione per un possibile algoritmo greedy.
Siano $e_1, e_2, \ldots, e_m$ gli archi del grafo, ordinati in modo che $c(e_1) \leq c(e_2) \leq \ldots \leq c(e_m)$

Intuizione per un possibile algoritmo greedy.
Siano $e_1, e_2, \ldots, e_m$ gli archi del grafo, ordinati in modo che $c(e_1) \leq c(e_2) \leq \ldots \leq c(e_m)$

- Un'idea potrebbe essere quella di aggiungere all'insieme $T \subseteq E$ che vogliamo costruire (di minimo costo totale, ricordiamolo),

Intuizione per un possibile algoritmo greedy.
Siano $e_1, e_2, \ldots, e_m$ gli archi del grafo, ordinati in modo che $c(e_1) \leq c(e_2) \leq \ldots \leq c(e_m)$

- Un'idea potrebbe essere quella di aggiungere all'insieme $T \subseteq E$ che vogliamo costruire (di minimo costo totale, ricordiamolo), uno ad uno gli archi e_i,

Intuizione per un possibile algoritmo greedy.
Siano $e_1, e_2, \ldots, e_m$ gli archi del grafo, ordinati in modo che $c(e_1) \leq c(e_2) \leq \ldots \leq c(e_m)$

- Un'idea potrebbe essere quella di aggiungere all'insieme $T \subseteq E$ che vogliamo costruire (di minimo costo totale, ricordiamolo), uno ad uno gli archi e_i, iniziando da quello che costa meno

Intuizione per un possibile algoritmo greedy.
Siano $e_1, e_2, \ldots, e_m$ gli archi del grafo, ordinati in modo che $c(e_1) \leq c(e_2) \leq \ldots \leq c(e_m)$

- Un'idea potrebbe essere quella di aggiungere all'insieme $T \subseteq E$ che vogliamo costruire (di minimo costo totale, ricordiamolo), uno ad uno gli archi e_i, iniziando da quello che costa meno e proseguendo via via con quelli di peso maggiore.

Intuizione per un possibile algoritmo greedy.
Siano $e_1, e_2, \ldots, e_m$ gli archi del grafo, ordinati in modo che $c(e_1) \leq c(e_2) \leq \ldots \leq c(e_m)$

- Un'idea potrebbe essere quella di aggiungere all'insieme $T \subseteq E$ che vogliamo costruire (di minimo costo totale, ricordiamolo), uno ad uno gli archi e_i, iniziando da quello che costa meno e proseguendo via via con quelli di peso maggiore.
- Se l'aggiunta di un arco e_i all'insieme attuale T crea un ciclo, non va bene.

Intuizione per un possibile algoritmo greedy.
Siano $e_1, e_2, \ldots, e_m$ gli archi del grafo, ordinati in modo che $c(e_1) \leq c(e_2) \leq \ldots \leq c(e_m)$

- Un'idea potrebbe essere quella di aggiungere all'insieme $T \subseteq E$ che vogliamo costruire (di minimo costo totale, ricordiamolo), uno ad uno gli archi e_i, iniziando da quello che costa meno e proseguendo via via con quelli di peso maggiore.
- Se l'aggiunta di un arco e_i all'insieme attuale T crea un ciclo, non va bene. Scartiamo l'arco e_i e prendiamo in considerazione l'arco e_{i+1}, iterando il processo.

Intuizione per un possibile algoritmo greedy.
Siano $e_1, e_2, \ldots, e_m$ gli archi del grafo, ordinati in modo che $c(e_1) \leq c(e_2) \leq \ldots \leq c(e_m)$

- Un'idea potrebbe essere quella di aggiungere all'insieme $T \subseteq E$ che vogliamo costruire (di minimo costo totale, ricordiamolo), uno ad uno gli archi e_i, iniziando da quello che costa meno e proseguendo via via con quelli di peso maggiore.
- Se l'aggiunta di un arco e_i all'insieme attuale T crea un ciclo, non va bene. Scartiamo l'arco e_i e prendiamo in considerazione l'arco e_{i+1}, iterando il processo.
- Smettiamo di aggiungere archi a T quando abbiamo connesso tutti i vertici di V,

Intuizione per un possibile algoritmo greedy.
Siano $e_1, e_2, \ldots, e_m$ gli archi del grafo, ordinati in modo che $c(e_1) \leq c(e_2) \leq \ldots \leq c(e_m)$

- Un'idea potrebbe essere quella di aggiungere all'insieme $T \subseteq E$ che vogliamo costruire (di minimo costo totale, ricordiamolo), uno ad uno gli archi e_i, iniziando da quello che costa meno e proseguendo via via con quelli di peso maggiore.
- Se l'aggiunta di un arco e_i all'insieme attuale T crea un ciclo, non va bene. Scartiamo l'arco e_i e prendiamo in considerazione l'arco e_{i+1}, iterando il processo.
- Smettiamo di aggiungere archi a T quando abbiamo connesso tutti i vertici di V, ovvero quando (V, T) é un albero.

Vi potrebbe essere, tuttavia, un modo alternativo di procedere, in maniera analoga all'algoritmo di Dijkstra.

Vi potrebbe essere, tuttavia, un modo alternativo di procedere, in maniera analoga all'algoritmo di Dijkstra. Potremmo iniziare con un nodo radice s e tentare di costruire, in modo greedy, un albero con radice in s.

Vi potrebbe essere, tuttavia, un modo alternativo di procedere, in maniera analoga all'algoritmo di Dijkstra. Potremmo iniziare con un nodo radice s e tentare di costruire, in modo greedy, un albero con radice in s. Ad ogni passo attacchiamo all'albero corrente il nodo per cui ci costa meno farlo.

Vi potrebbe essere, tuttavia, un modo alternativo di procedere, in maniera analoga all'algoritmo di Dijkstra. Potremmo iniziare con un nodo radice s e tentare di costruire, in modo greedy, un albero con radice in s. Ad ogni passo attacchiamo all'albero corrente il nodo per cui ci costa meno farlo.

- Per fare ciò manteniamo ad ogni istante un insieme $S \subseteq V$ su cui un Minimo Albero Ricoprente T è stato costruito fin'ora.

Vi potrebbe essere, tuttavia, un modo alternativo di procedere, in maniera analoga all'algoritmo di Dijkstra. Potremmo iniziare con un nodo radice s e tentare di costruire, in modo greedy, un albero con radice in s. Ad ogni passo attacchiamo all'albero corrente il nodo per cui ci costa meno farlo.

- Per fare ciò manteniamo ad ogni istante un insieme $S \subseteq V$ su cui un Minimo Albero Ricoprente T è stato costruito fin'ora. Inizialmente $S = \{s\}$.

Vi potrebbe essere, tuttavia, un modo alternativo di procedere, in maniera analoga all'algoritmo di Dijkstra. Potremmo iniziare con un nodo radice s e tentare di costruire, in modo greedy, un albero con radice in s. Ad ogni passo attacchiamo all'albero corrente il nodo per cui ci costa meno farlo.

- Per fare ciò manteniamo ad ogni istante un insieme $S \subseteq V$ su cui un Minimo Albero Ricoprente T è stato costruito fin'ora. Inizialmente $S = \{s\}$.
- Ad ogni iterazione aumentiamo l'albero di un nodo,

Vi potrebbe essere, tuttavia, un modo alternativo di procedere, in maniera analoga all'algoritmo di Dijkstra. Potremmo iniziare con un nodo radice s e tentare di costruire, in modo greedy, un albero con radice in s. Ad ogni passo attacchiamo all'albero corrente il nodo per cui ci costa meno farlo.

- Per fare ciò manteniamo ad ogni istante un insieme $S \subseteq V$ su cui un Minimo Albero Ricoprente T è stato costruito fin'ora. Inizialmente $S = \{s\}$.
- Ad ogni iterazione aumentiamo l'albero di un nodo, aggiungendo il nodo $v \notin S$ che minimizza il costo di tale aumento,

Vi potrebbe essere, tuttavia, un modo alternativo di procedere, in maniera analoga all'algoritmo di Dijkstra. Potremmo iniziare con un nodo radice s e tentare di costruire, in modo greedy, un albero con radice in s. Ad ogni passo attacchiamo all'albero corrente il nodo per cui ci costa meno farlo.

- Per fare ciò manteniamo ad ogni istante un insieme $S \subseteq V$ su cui un Minimo Albero Ricoprente T è stato costruito fin'ora. Inizialmente $S = \{s\}$.
- Ad ogni iterazione aumentiamo l'albero di un nodo, aggiungendo il nodo $v \notin S$ che minimizza il costo di tale aumento, pari a $\min_{e=(u,v):u \in S} c(e)$,

Vi potrebbe essere, tuttavia, un modo alternativo di procedere, in maniera analoga all'algoritmo di Dijkstra. Potremmo iniziare con un nodo radice s e tentare di costruire, in modo greedy, un albero con radice in s. Ad ogni passo attacchiamo all'albero corrente il nodo per cui ci costa meno farlo.

- Per fare ciò manteniamo ad ogni istante un insieme $S \subseteq V$ su cui un Minimo Albero Ricoprente T è stato costruito fin'ora. Inizialmente $S = \{s\}$.
- Ad ogni iterazione aumentiamo l'albero di un nodo, aggiungendo il nodo $v \notin S$ che minimizza il costo di tale aumento, pari a $\min_{e=(u,v):u \in S} c(e)$, ed includiamo l'arco $e = (u, v)$ che ottiene questo minimo all'albero corrente.

Vi potrebbe essere, tuttavia, un modo alternativo di procedere, in maniera analoga all'algoritmo di Dijkstra. Potremmo iniziare con un nodo radice s e tentare di costruire, in modo greedy, un albero con radice in s. Ad ogni passo attacchiamo all'albero corrente il nodo per cui ci costa meno farlo.

- Per fare ciò manteniamo ad ogni istante un insieme $S \subseteq V$ su cui un Minimo Albero Ricoprente T è stato costruito fin'ora. Inizialmente $S = \{s\}$.
- Ad ogni iterazione aumentiamo l'albero di un nodo, aggiungendo il nodo $v \notin S$ che minimizza il costo di tale aumento, pari a $\min_{e=(u,v):u \in S} c(e)$, ed includiamo l'arco $e = (u, v)$ che ottiene questo minimo all'albero corrente.
- Come prima, smettiamo di aggiungere archi a T quando abbiamo connesso tutti i vertici di V,

Vi potrebbe essere, tuttavia, un modo alternativo di procedere, in maniera analoga all'algoritmo di Dijkstra. Potremmo iniziare con un nodo radice s e tentare di costruire, in modo greedy, un albero con radice in s. Ad ogni passo attacchiamo all'albero corrente il nodo per cui ci costa meno farlo.

- Per fare ciò manteniamo ad ogni istante un insieme $S \subseteq V$ su cui un Minimo Albero Ricoprente T è stato costruito fin'ora. Inizialmente $S = \{s\}$.
- Ad ogni iterazione aumentiamo l'albero di un nodo, aggiungendo il nodo $v \notin S$ che minimizza il costo di tale aumento, pari a $\min_{e=(u,v):u \in S} c(e)$, ed includiamo l'arco $e = (u, v)$ che ottiene questo minimo all'albero corrente.
- Come prima, smettiamo di aggiungere archi a T quando abbiamo connesso tutti i vertici di V, ovvero quando (V, T) è un albero.

Quale dei due metodi funziona (ovvero produce un MST)?

Quale dei due metodi funziona (ovvero produce un MST)? Entrambi, per fortuna.

Quale dei due metodi funziona (ovvero produce un MST)? Entrambi, per fortuna. Il primo metodo porta all'*Algoritmo di Kruskal*,

Quale dei due metodi funziona (ovvero produce un MST)? Entrambi, per fortuna. Il primo metodo porta all'*Algoritmo di Kruskal*,

il secondo all'*Algoritmo di Prim*.

Assumiamo (per il momento) che i costi degli archi siano tutti diversi tra di loro

Assumiamo (per il momento) che i costi degli archi siano tutti diversi tra di loro

Fatto: Sia $\emptyset \neq S \subset V$ un sottoinsieme dei nodi, e sia $e = (u, v)$ l'arco di *costo minimo* con un estremo $u \in S$ e l'altro $v \in V - S$. Allora *ogni* MST contiene l'arco e.

Assumiamo (per il momento) che i costi degli archi siano tutti diversi tra di loro

Fatto: Sia $\emptyset \neq S \subset V$ un sottoinsieme dei nodi, e sia $e = (u, v)$ l'arco di *costo minimo* con un estremo $u \in S$ e l'altro $v \in V - S$. Allora *ogni* MST contiene l'arco e.

Supponiamo che ciò non sia vero, e sia T un MST che **non** contiene l'arco $e = (u, v)$.

Assumiamo (per il momento) che i costi degli archi siano tutti diversi tra di loro

Fatto: Sia $\emptyset \neq S \subset V$ un sottoinsieme dei nodi, e sia $e = (u, v)$ l'arco di *costo minimo* con un estremo $u \in S$ e l'altro $v \in V - S$. Allora *ogni* MST contiene l'arco e.

Supponiamo che ciò non sia vero, e sia T un MST che **non** contiene l'arco $e = (u, v)$. T dovrà contenere almeno un'arco $a = (x, y) \neq (u, v) = e$ con un'estremo $x \in S$ e l'altro in $y \in V - S$

Assumiamo (per il momento) che i costi degli archi siano tutti diversi tra di loro

Fatto: Sia $\emptyset \neq S \subset V$ un sottoinsieme dei nodi, e sia $e = (u, v)$ l'arco di *costo minimo* con un estremo $u \in S$ e l'altro $v \in V - S$. Allora *ogni* MST contiene l'arco e.

Supponiamo che ciò non sia vero, e sia T un MST che **non** contiene l'arco $e = (u, v)$. T dovrà contenere almeno un'arco $a = (x, y) \neq (u, v) = e$ con un'estremo $x \in S$ e l'altro in $y \in V - S$ (altrimenti come farebbe T a connettere tra di loro tutti i nodi di V?)

Assumiamo (per il momento) che i costi degli archi siano tutti diversi tra di loro

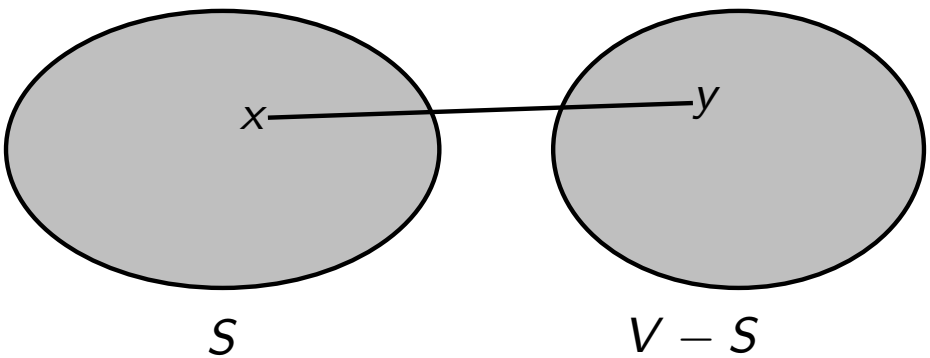
Fatto: Sia $\emptyset \neq S \subset V$ un sottoinsieme dei nodi, e sia $e = (u, v)$ l'arco di *costo minimo* con un estremo $u \in S$ e l'altro $v \in V - S$. Allora *ogni* MST contiene l'arco e.

Supponiamo che ciò non sia vero, e sia T un MST che **non** contiene l'arco $e = (u, v)$. T dovrà contenere almeno un'arco $a = (x, y) \neq (u, v) = e$ con un'estremo $x \in S$ e l'altro in $y \in V - S$ (altrimenti come farebbe T a connettere tra di loro tutti i nodi di V?) con $c(x, y) > c(u, v)$

Assumiamo (per il momento) che i costi degli archi siano tutti diversi tra di loro

Fatto: Sia $\emptyset \neq S \subset V$ un sottoinsieme dei nodi, e sia $e = (u, v)$ l'arco di *costo minimo* con un estremo $u \in S$ e l'altro $v \in V - S$. Allora *ogni* MST contiene l'arco e.

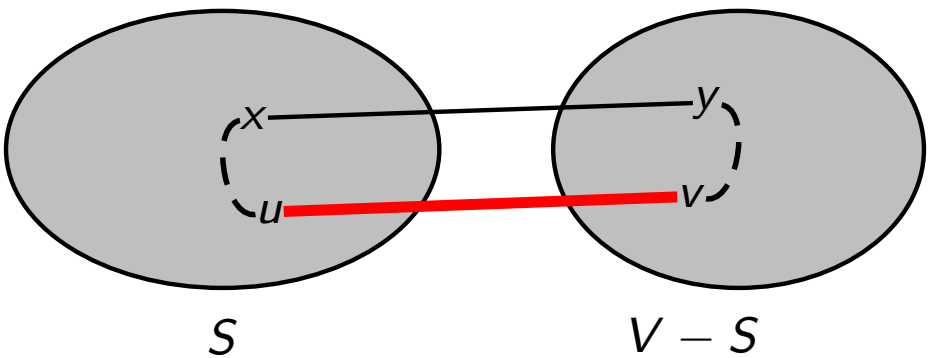
Supponiamo che ciò non sia vero, e sia T un MST che **non** contiene l'arco $e = (u, v)$. T dovrà contenere almeno un'arco $a = (x, y) \neq (u, v) = e$ con un'estremo $x \in S$ e l'altro in $y \in V - S$ (altrimenti come farebbe T a connettere tra di loro tutti i nodi di V?) con $c(x, y) > c(u, v)$



Assumiamo (per il momento) che i costi degli archi siano tutti diversi tra di loro

Fatto: Sia $\emptyset \neq S \subset V$ un sottoinsieme dei nodi, e sia $e = (u, v)$ l'arco di *costo minimo* con un estremo $u \in S$ e l'altro $v \in V - S$. Allora *ogni* MST contiene l'arco e.

Supponiamo che ciò non sia vero, e sia T un MST che **non** contiene l'arco $e = (u, v)$. T dovrà contenere almeno un'arco $a = (x, y) \neq (u, v) = e$ con un'estremo $x \in S$ e l'altro in $y \in V - S$ (altrimenti come farebbe T a connettere tra di loro tutti i nodi di V?) con $c(x, y) > c(u, v)$

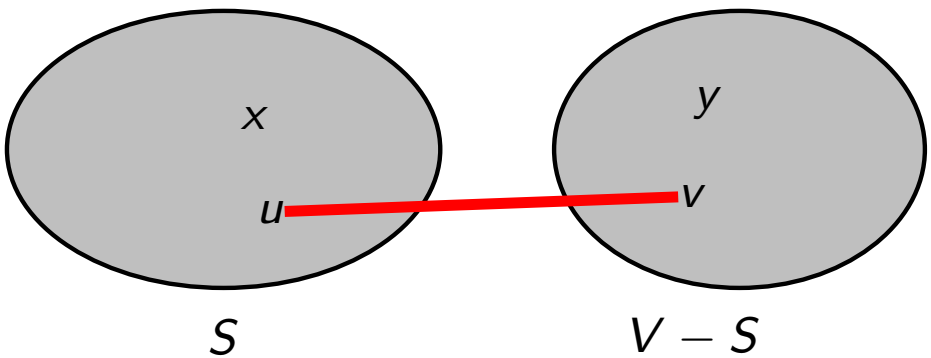


Aggiungiamo a T l'arco $e = (u, v)$, (per ipotesi $c(u, v) < c(x, y)$), si creerà un ciclo!

Assumiamo (per il momento) che i costi degli archi siano tutti diversi tra di loro

Fatto: Sia $\emptyset \neq S \subset V$ un sottoinsieme dei nodi, e sia $e = (u, v)$ l'arco di *costo minimo* con un estremo $u \in S$ e l'altro $v \in V - S$. Allora *ogni* MST contiene l'arco e.

Supponiamo che ciò non sia vero, e sia T un MST che **non** contiene l'arco $e = (u, v)$. T dovrà contenere almeno un'arco $a = (x, y) \neq (u, v) = e$ con un'estremo $x \in S$ e l'altro in $y \in V - S$ (altrimenti come farebbe T a connettere tra di loro tutti i nodi di V?) con $c(x, y) > c(u, v)$

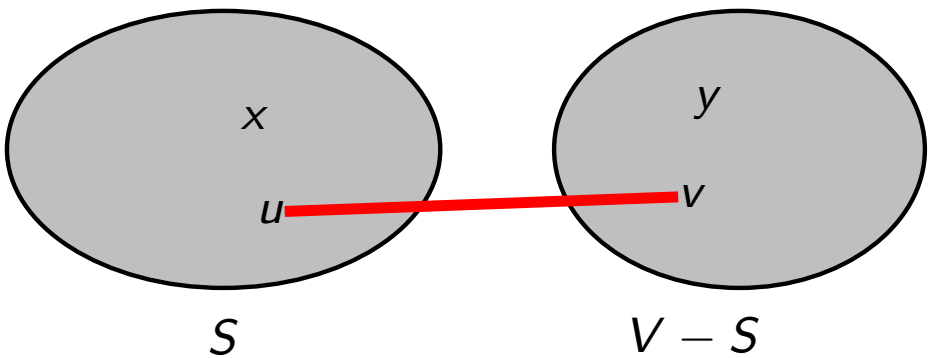


Aggiungiamo a T l'arco $e = (u, v)$, (per ipotesi $c(u, v) < c(x, y)$), si creerà un ciclo! **Eliminiamo** allora l'arco (x, y) per ottenere questa situazione

Assumiamo (per il momento) che i costi degli archi siano tutti diversi tra di loro

Fatto: Sia $\emptyset \neq S \subset V$ un sottoinsieme dei nodi, e sia $e = (u, v)$ l'arco di *costo minimo* con un estremo $u \in S$ e l'altro $v \in V - S$. Allora *ogni* MST contiene l'arco e.

Supponiamo che ciò non sia vero, e sia T un MST che **non** contiene l'arco $e = (u, v)$. T dovrà contenere almeno un'arco $a = (x, y) \neq (u, v) = e$ con un'estremo $x \in S$ e l'altro in $y \in V - S$ (altrimenti come farebbe T a connettere tra di loro tutti i nodi di V?) con $c(x, y) > c(u, v)$



Aggiungiamo a T l'arco $e = (u, v)$, (per ipotesi $c(u, v) < c(x, y)$), si creerà un ciclo! **Eliminiamo** allora l'arco (x, y) per ottenere questa situazione, **cioè un nuovo albero** T' che connette tutti i vertici di V, con costo(T') = costo(T) − $c(x, y) + c(u, v)$ < costo(T), contro l'ipotesi che T avesse costo minimo.

Ricordiamo l'Algoritmo di Kruskal

Siano $e_1, e_2, \ldots, e_m$ gli archi del grafo, ordinati in modo che $c(e_1) \leq c(e_2) \leq \ldots \leq c(e_m)$

- Aggiungiamo all'insieme $T \subseteq E$ che vogliamo costruire uno ad uno gli archi e_i, iniziando da quello che costa meno (ovvero e_1) e proseguendo via via con quelli di peso maggiore.

Ricordiamo l'Algoritmo di Kruskal

Siano $e_1, e_2, \ldots, e_m$ gli archi del grafo, ordinati in modo che $c(e_1) \leq c(e_2) \leq \ldots \leq c(e_m)$

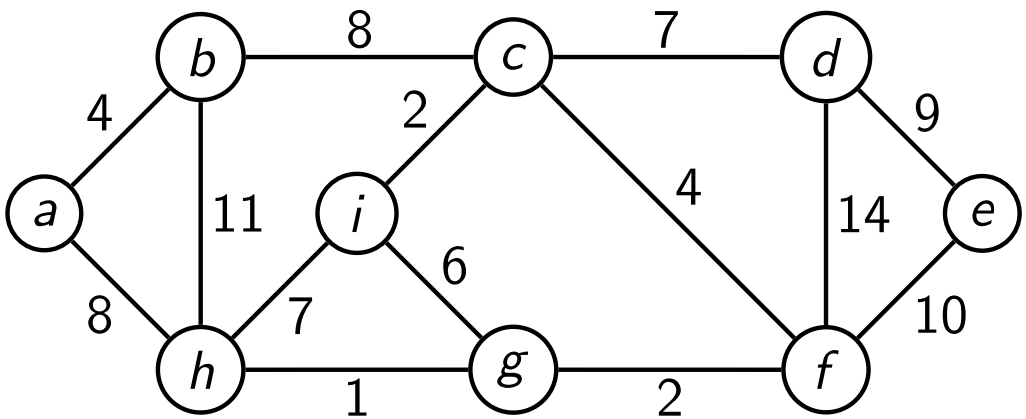
- Aggiungiamo all'insieme $T \subseteq E$ che vogliamo costruire uno ad uno gli archi e_i, iniziando da quello che costa meno (ovvero e_1) e proseguendo via via con quelli di peso maggiore.
- Se l'aggiunta di un arco e_i all'insieme attuale T crea un ciclo, non va bene. Scartiamo l'arco e_i e prendiamo in considerazione l'arco e_{i+1}, iterando il processo.

Ricordiamo l'Algoritmo di Kruskal

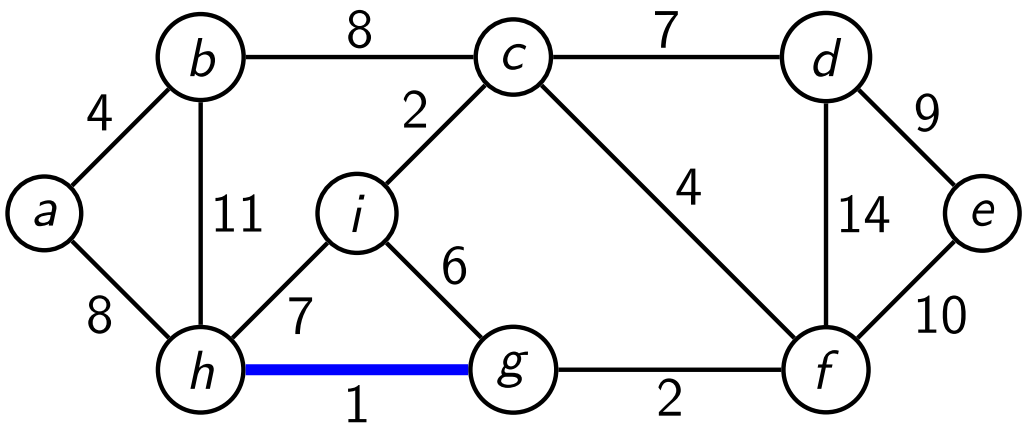
Siano $e_1, e_2, \ldots, e_m$ gli archi del grafo, ordinati in modo che $c(e_1) \leq c(e_2) \leq \ldots \leq c(e_m)$

- Aggiungiamo all'insieme $T \subseteq E$ che vogliamo costruire uno ad uno gli archi e_i, iniziando da quello che costa meno (ovvero e_1) e proseguendo via via con quelli di peso maggiore.
- Se l'aggiunta di un arco e_i all'insieme attuale T crea un ciclo, non va bene. Scartiamo l'arco e_i e prendiamo in considerazione l'arco e_{i+1}, iterando il processo.
- Smettiamo di aggiungere archi a T quando abbiamo connesso tutti i vertici di V, ovvero quando (V, T) é un albero.

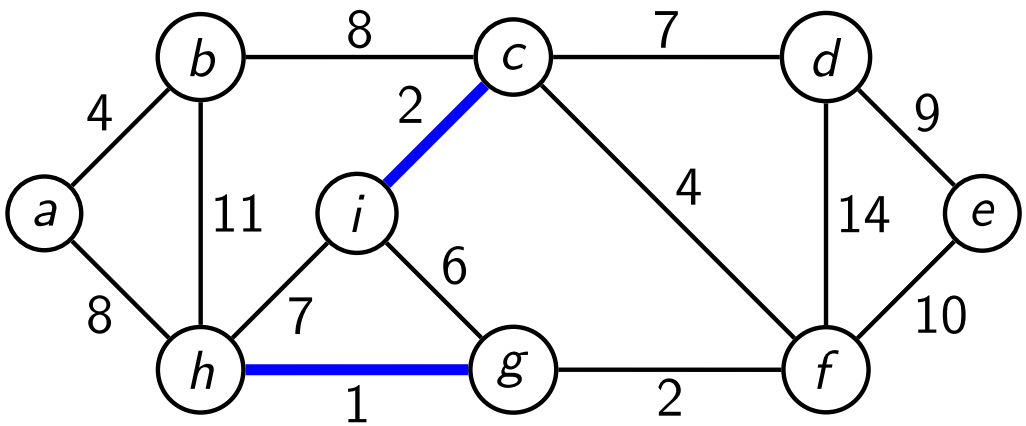
Vediamo un esempio di esecuzione dell'algoritmo di Kruskal sul grafo di seguito, in cui l'ordinamento degli archi, in base al loro costo è:
$(h, g), (i, c), (g, f), (a, b), (c, f), (i, g), (c, d), (i, h), (a, h), (b, c),$
$(d, e), (b, h), (d, f)$



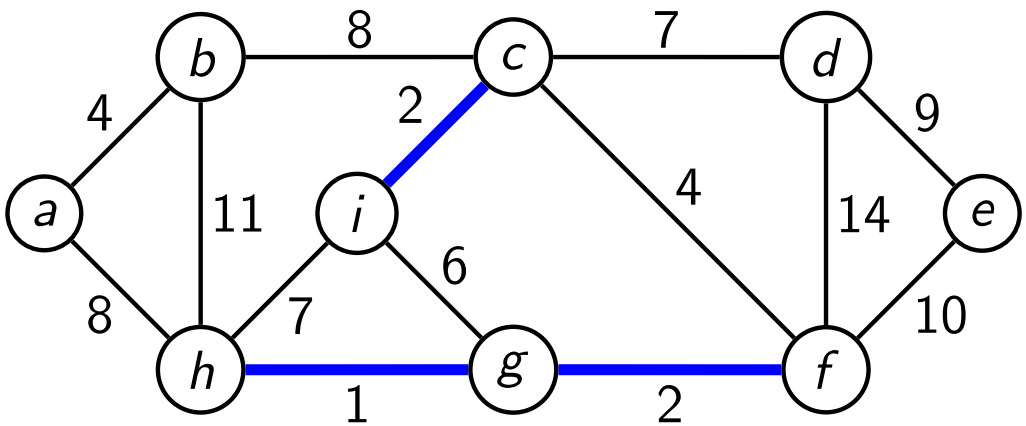
Vediamo un esempio di esecuzione dell'algoritmo di Kruskal sul grafo di seguito, in cui l'ordinamento degli archi, in base al loro costo è:
(h,g), $(i, c), (g, f), (a, b), (c, f), (i, g), (c, d), (i, h), (a, h), (b, c), (d, e), (b, h), (d, f)$



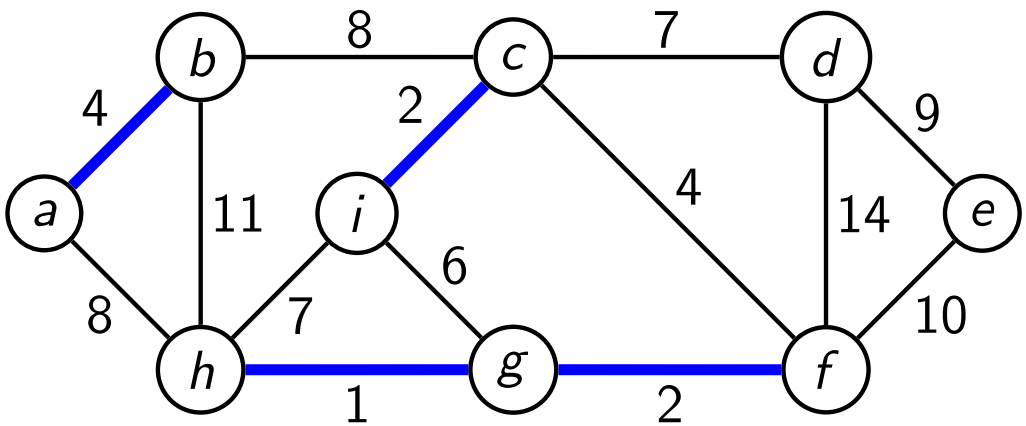
Vediamo un esempio di esecuzione dell'algoritmo di Kruskal sul grafo di seguito, in cui l'ordinamento degli archi, in base al loro costo è:
(h, g), **(i,c)**, (g, f), (a, b), (c, f), (i, g), (c, d), (i, h), (a, h), (b, c), (d, e), (b, h), (d, f)



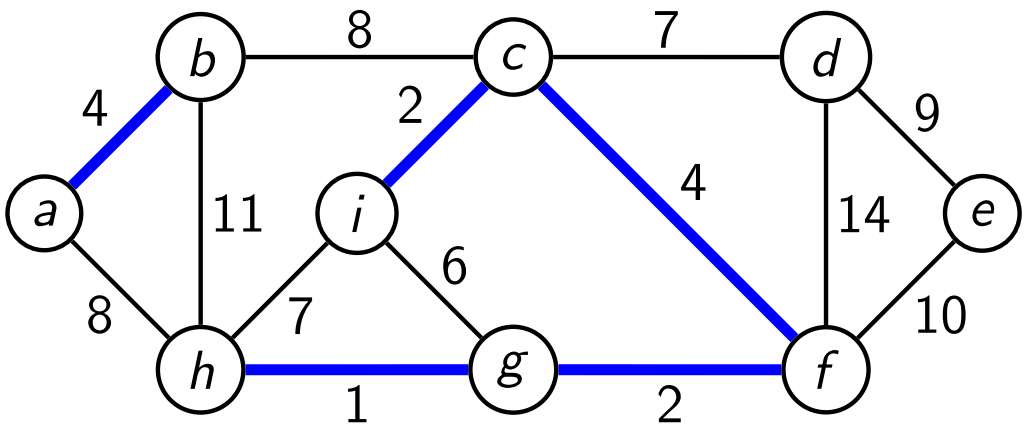
Vediamo un esempio di esecuzione dell'algoritmo di Kruskal sul grafo di seguito, in cui l'ordinamento degli archi, in base al loro costo è:
$(h, g), (i, c), \mathbf{(g,f)}, (a, b), (c, f), (i, g), (c, d), (i, h), (a, h), (b, c), (d, e), (b, h), (d, f)$



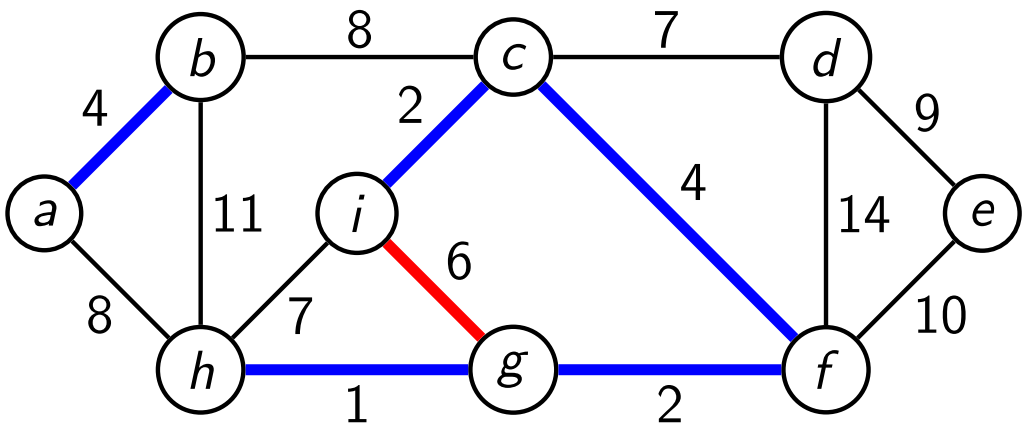
Vediamo un esempio di esecuzione dell'algoritmo di Kruskal sul grafo di seguito, in cui l'ordinamento degli archi, in base al loro costo è:
$(h, g), (i, c), (g, f)$, **(a, b)**, $(c, f), (i, g), (c, d), (i, h), (a, h), (b, c),$
$(d, e), (b, h), (d, f)$



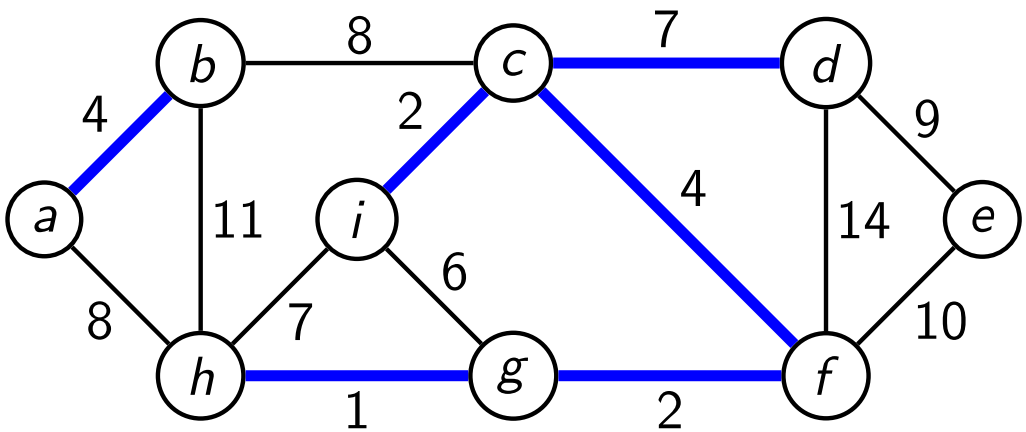
Vediamo un esempio di esecuzione dell'algoritmo di Kruskal sul grafo di seguito, in cui l'ordinamento degli archi, in base al loro costo è:
$(h, g), (i, c), (g, f), (a, b),$ **(c, f)**$, (i, g), (c, d), (i, h), (a, h), (b, c),$
$(d, e), (b, h), (d, f)$



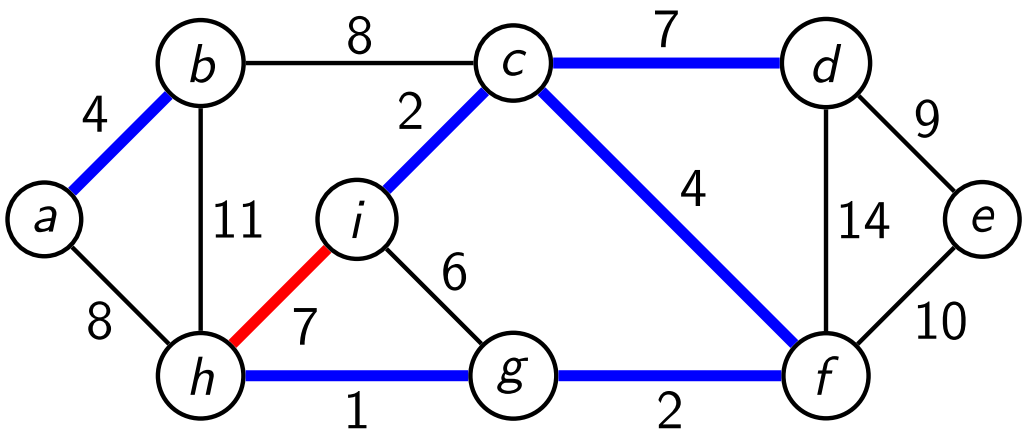
Vediamo un esempio di esecuzione dell'algoritmo di Kruskal sul grafo di seguito, in cui l'ordinamento degli archi, in base al loro costo è:
$(h, g), (i, c), (g, f), (a, b), (c, f)$, **(i, g)**, $(c, d), (i, h), (a, h), (b, c), (d, e), (b, h), (d, f)$



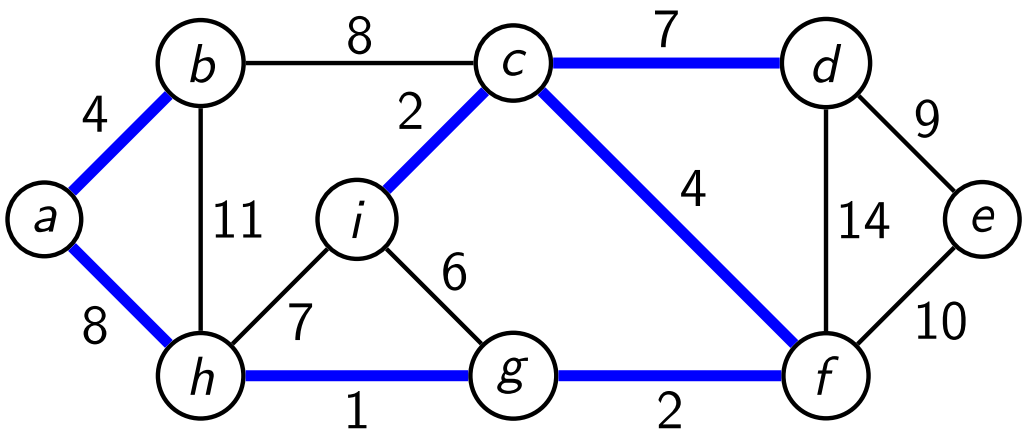
Vediamo un esempio di esecuzione dell'algoritmo di Kruskal sul grafo di seguito, in cui l'ordinamento degli archi, in base al loro costo è:
$(h, g), (i, c), (g, f), (a, b), (c, f), (i, g),$ **(c, d)**$, (i, h), (a, h), (b, c),$
$(d, e), (b, h), (d, f)$



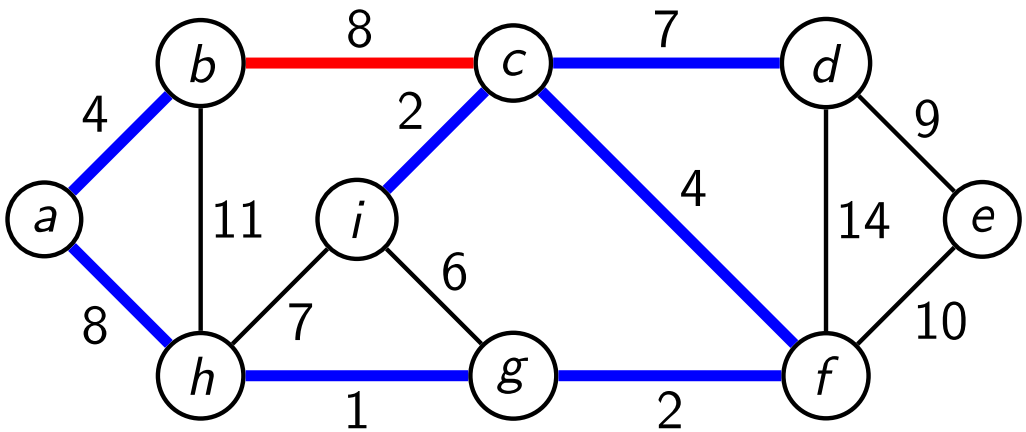
Vediamo un esempio di esecuzione dell'algoritmo di Kruskal sul grafo di seguito, in cui l'ordinamento degli archi, in base al loro costo è:
$(h, g), (i, c), (g, f), (a, b), (c, f), (i, g), (c, d)$, **(i, h)**, $(a, h), (b, c)$, $(d, e), (b, h), (d, f)$



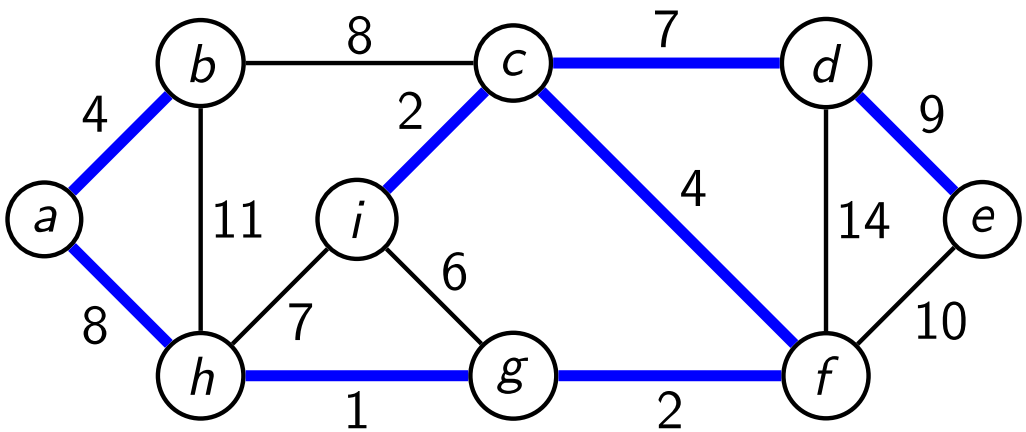
Vediamo un esempio di esecuzione dell'algoritmo di Kruskal sul grafo di seguito, in cui l'ordinamento degli archi, in base al loro costo è:
$(h, g), (i, c), (g, f), (a, b), (c, f), (i, g), (c, d), (i, h),$ **(a, h)**$, (b, c),$
$(d, e), (b, h), (d, f)$



Vediamo un esempio di esecuzione dell'algoritmo di Kruskal sul grafo di seguito, in cui l'ordinamento degli archi, in base al loro costo è:
$(h, g), (i, c), (g, f), (a, b), (c, f), (i, g), (c, d), (i, h), (a, h)$, **(b, c)**,
$(d, e), (b, h), (d, f)$



Vediamo un esempio di esecuzione dell'algoritmo di Kruskal sul grafo di seguito, in cui l'ordinamento degli archi, in base al loro costo è:
$(h, g), (i, c), (g, f), (a, b), (c, f), (i, g), (c, d), (i, h), (a, h), (b, c),$
(d, e)$, (b, h), (d, f)$



Proviamo ora che l'algoritmo di Kruskal

Aggiungi a T uno ad uno gli archi del grafo, in ordine di costo crescente, saltando gli archi che creano cicli con gli archi già aggiunti.

Proviamo ora che l'algoritmo di Kruskal

> Aggiungi a T uno ad uno gli archi del grafo, in ordine di costo crescente, saltando gli archi che creano cicli con gli archi già aggiunti.

produce effettivamente un MST.

Proviamo ora che l'algoritmo di Kruskal

Aggiungi a T uno ad uno gli archi del grafo, in ordine di costo crescente, saltando gli archi che creano cicli con gli archi già aggiunti.

produce effettivamente un MST.
Sia $e = (v, w)$ un generico arco inserito in T dall'algoritmo,

Proviamo ora che l'algoritmo di Kruskal

> Aggiungi a T uno ad uno gli archi del grafo, in ordine di costo crescente, saltando gli archi che creano cicli con gli archi già aggiunti.

produce effettivamente un MST.
Sia $e = (v, w)$ un generico arco inserito in T dall'algoritmo, e sia S l'insieme di tutti i nodi connessi a v attraverso un cammino, al passo immediatamente precedente a quello in cui si aggiunge (v, w) a T.

Proviamo ora che l'algoritmo di Kruskal

> Aggiungi a T uno ad uno gli archi del grafo, in ordine di costo crescente, saltando gli archi che creano cicli con gli archi già aggiunti.

produce effettivamente un MST.
Sia $e = (v, w)$ un generico arco inserito in T dall'algoritmo, e sia S l'insieme di tutti i nodi connessi a v attraverso un cammino, al passo immediatamente precedente a quello in cui si aggiunge (v, w) a T.
Ovviamente vale che $v \in S$, mentre $w \notin S$,

Proviamo ora che l'algoritmo di Kruskal

> Aggiungi a T uno ad uno gli archi del grafo, in ordine di costo crescente, saltando gli archi che creano cicli con gli archi già aggiunti.

produce effettivamente un MST.
Sia $e = (v, w)$ un generico arco inserito in T dall'algoritmo, e sia S l'insieme di tutti i nodi connessi a v attraverso un cammino, al passo immediatamente precedente a quello in cui si aggiunge (v, w) a T.
Ovviamente vale che $v \in S$, mentre $w \notin S$, altrimenti l'arco (v, w) creerebbe un ciclo.

Proviamo ora che l'algoritmo di Kruskal

> Aggiungi a T uno ad uno gli archi del grafo, in ordine di costo crescente, saltando gli archi che creano cicli con gli archi già aggiunti.

produce effettivamente un MST.
Sia $e = (v, w)$ un generico arco inserito in T dall'algoritmo, e sia S l'insieme di tutti i nodi connessi a v attraverso un cammino, al passo immediatamente precedente a quello in cui si aggiunge (v, w) a T. Ovviamente vale che $v \in S$, mentre $w \notin S$, altrimenti l'arco (v, w) creerebbe un ciclo. Inoltre, negli istanti precedenti l'algoritmo non ha incontrato nessun arco da nodi in S a nodi in $V - S$, altrimenti un tale arco sarebbe stato aggiunto, visto che non creava cicli.

Proviamo ora che l'algoritmo di Kruskal

> Aggiungi a T uno ad uno gli archi del grafo, in ordine di costo crescente, saltando gli archi che creano cicli con gli archi già aggiunti.

produce effettivamente un MST.
Sia $e = (v, w)$ un generico arco inserito in T dall'algoritmo, e sia S l'insieme di tutti i nodi connessi a v attraverso un cammino, al passo immediatamente precedente a quello in cui si aggiunge (v, w) a T. Ovviamente vale che $v \in S$, mentre $w \notin S$, altrimenti l'arco (v, w) creerebbe un ciclo. Inoltre, negli istanti precedenti l'algoritmo non ha incontrato nessun arco da nodi in S a nodi in $V - S$, altrimenti un tale arco sarebbe stato aggiunto, visto che non creava cicli. Pertanto l'arco (v, w) è il **primo** arco da S a $V - S$ che l'algoritmo incontra,

Proviamo ora che l'algoritmo di Kruskal

> Aggiungi a T uno ad uno gli archi del grafo, in ordine di costo crescente, saltando gli archi che creano cicli con gli archi già aggiunti.

produce effettivamente un MST.
Sia $e = (v, w)$ un generico arco inserito in T dall'algoritmo, e sia S l'insieme di tutti i nodi connessi a v attraverso un cammino, al passo immediatamente precedente a quello in cui si aggiunge (v, w) a T. Ovviamente vale che $v \in S$, mentre $w \notin S$, altrimenti l'arco (v, w) creerebbe un ciclo. Inoltre, negli istanti precedenti l'algoritmo non ha incontrato nessun arco da nodi in S a nodi in $V - S$, altrimenti un tale arco sarebbe stato aggiunto, visto che non creava cicli. Pertanto l'arco (v, w) è il **primo** arco da S a $V - S$ che l'algoritmo incontra, ovvero è l'arco di minor costo da S a $V - S$ che, abbiamo visto, **appartiene** ad ogni MST.

Ci rimane da mostrare che l'output dell'algoritmo di Kruskal è un albero.

Ci rimane da mostrare che l'output dell'algoritmo di Kruskal è un albero. Sicuramente, per costruzione, l'output (V, T) non contiene cicli.

Ci rimane da mostrare che l'output dell'algoritmo di Kruskal è un albero.
Sicuramente, per costruzione, l'output (V, T) non contiene cicli.
Potrebbe (V, T) non essere connesso?

Ci rimane da mostrare che l'output dell'algoritmo di Kruskal è un albero.
Sicuramente, per costruzione, l'output (V, T) non contiene cicli.
Potrebbe (V, T) non essere connesso? Ovvero potrebbe esistere un $\emptyset \neq S \subset V$ per cui in T non esiste alcun arco da S a $V - S$?

Ci rimane da mostrare che l'output dell'algoritmo di Kruskal è un albero.
Sicuramente, per costruzione, l'output (V, T) non contiene cicli.
Potrebbe (V, T) non essere connesso? Ovvero potrebbe esistere un $\emptyset \neq S \subset V$ per cui in T non esiste alcun arco da S a $V - S$?
Sicuramente no! Infatti, poichè il grafo G è connesso, un tale arco e esiste sicuramente in G

Ci rimane da mostrare che l'output dell'algoritmo di Kruskal è un albero.
Sicuramente, per costruzione, l'output (V, T) non contiene cicli.
Potrebbe (V, T) non essere connesso? Ovvero potrebbe esistere un $\emptyset \neq S \subset V$ per cui in T non esiste alcun arco da S a $V - S$?
Sicuramente no! Infatti, poichè il grafo G è connesso, un tale arco e esiste sicuramente in G e poichè l'algoritmo di Kruskal esamina **tutti** gli archi di G, prima o poi incontrerà tale arco e

Ci rimane da mostrare che l'output dell'algoritmo di Kruskal è un albero. Sicuramente, per costruzione, l'output (V, T) non contiene cicli. Potrebbe (V, T) non essere connesso? Ovvero potrebbe esistere un $\emptyset \neq S \subset V$ per cui in T non esiste alcun arco da S a $V - S$? Sicuramente no! Infatti, poichè il grafo G è connesso, un tale arco e esiste sicuramente in G e poichè l'algoritmo di Kruskal esamina **tutti** gli archi di G, prima o poi incontrerà tale arco e e lo inserirà, visto che non crea cicli.

L'algoritmo di Prim per la costruzione di uno MST procede in maniera differente.

- Inizialmente $S = \{s\}$, $T = \emptyset$.

L'algoritmo di Prim per la costruzione di uno MST procede in maniera differente.

- Inizialmente $S = \{s\}$, $T = \emptyset$.
- Ad ogni iterazione aumentiamo l'albero di un nodo, aggiungendo il nodo $v \notin S$ che *minimizza* il costo di tale aumento, pari a $\min_{e=(u,v):u \in S} c(e)$, ed includiamo l'arco $e = (u, v)$ che ottiene questo minimo all'albero corrente T.

L'algoritmo di Prim per la costruzione di uno MST procede in maniera differente.

- Inizialmente $S = \{s\}$, $T = \emptyset$.
- Ad ogni iterazione aumentiamo l'albero di un nodo, aggiungendo il nodo $v \notin S$ che *minimizza* il costo di tale aumento, pari a $\min_{e=(u,v):u \in S} c(e)$, ed includiamo l'arco $e = (u, v)$ che ottiene questo minimo all'albero corrente T.
- Terminiamo quando (V, T) è un albero (ovvero quando (V, T) è connesso).

Che l'algoritmo di Prim produca un albero è ovvio, visto che aggiunge archi solo da nodi già tra di loro connessi in S a nuovi nodi "fuori" di S (quindi non crea cicli).

Che l'algoritmo di Prim produca un albero è ovvio, visto che aggiunge archi solo da nodi già tra di loro connessi in S a nuovi nodi "fuori" di S (quindi non crea cicli).

Inoltre, sempre per come l'algoritmo opera, ad ogni passo aggiunge a T l'arco di minimo costo che ha un estremo u in S (insieme dei nodi su cui un albero ricoprente parziale è stato già costruito) ad un nodo $v \in V - S$.

Che l'algoritmo di Prim produca un albero è ovvio, visto che aggiunge archi solo da nodi già tra di loro connessi in S a nuovi nodi "fuori" di S (quindi non crea cicli).

Inoltre, sempre per come l'algoritmo opera, ad ogni passo aggiunge a T l'arco di minimo costo che ha un estremo u in S (insieme dei nodi su cui un albero ricoprente parziale è stato già costruito) ad un nodo $v \in V - S$.

Dalla proprietà prima vista, tale arco appartiene ad ogni MST del grafo (cioè, di nuovo l'algoritmo non inserisce mai archi che non appartengono a MST, quindi produce effettivamente un MST).

Implementazione dell'algoritmo di Prim per MST: L'implementazione è simile a quella dell'algoritmo di Dijkstra: occorre decidere quale nodo aggiungere all'albero T con nodi S che stiamo "crescendo".

Implementazione dell'algoritmo di Prim per MST: L'implementazione è simile a quella dell'algoritmo di Dijkstra: occorre decidere quale nodo aggiungere all'albero T con nodi S che stiamo "crescendo".

Per ogni nodo $v \in V - S$, manteniamo un valore

$$a(v) = \min_{e=(u,v):u \in S} c(e)$$

che rappresenta il costo in cui incorriamo per aggiungere il nodo v all'albero, usando l'arco di minimo costo per tale aggiunta.

Implementazione dell'algoritmo di Prim per MST: L'implementazione è simile a quella dell'algoritmo di Dijkstra: occorre decidere quale nodo aggiungere all'albero T con nodi S che stiamo "crescendo".

Per ogni nodo $v \in V - S$, manteniamo un valore

$$a(v) = \min_{e=(u,v):u \in S} c(e)$$

che rappresenta il costo in cui incorriamo per aggiungere il nodo v all'albero, usando l'arco di minimo costo per tale aggiunta.

Manteniamo i nodi in una coda a priorità, organizzata in base ai valori $a(v)$; selezionamo un nodo con l'operazione `ExtractMin`, e effettuiamo l'aggiornamento dei valori $a(v)$ con l'operazione `DecreaseKey`.

Vediamo ora come implementare in maniera efficiente l'algoritmo di Prim per MST.

MST-PRIM$(G = (V, E), c, r)$

Vediamo ora come implementare in maniera efficiente l'algoritmo di Prim per MST.

```
  MST-PRIM(G = (V, E), c, r)
1 Q ← V
```

Vediamo ora come implementare in maniera efficiente l'algoritmo di Prim per MST.

```
  MST-PRIM(G = (V, E), c, r)
1 Q ← V
2 For each u ∈ Q
```

Vediamo ora come implementare in maniera efficiente l'algoritmo di Prim per MST.

```
  MST-PRIM(G = (V, E), c, r)
1 Q ← V
2 For each u ∈ Q
3     a(u) ← ∞
```

Vediamo ora come implementare in maniera efficiente l'algoritmo di Prim per MST.

```
  MST-PRIM(G = (V, E), c, r)
1 Q ← V
2 For each u ∈ Q
3     a(u) ← ∞
4 a(r) ← 0, parent(r) ← NIL
```

Vediamo ora come implementare in maniera efficiente l'algoritmo di Prim per MST.

```
  MST-PRIM(G = (V, E), c, r)
1 Q ← V
2 For each u ∈ Q
3     a(u) ← ∞
4 a(r) ← 0, parent(r) ← NIL
6 While Q ≠ ∅
```

Vediamo ora come implementare in maniera efficiente l'algoritmo di Prim per MST.

```
  MST-PRIM(G = (V, E), c, r)
1 Q ← V
2 For each u ∈ Q
3     a(u) ← ∞
4 a(r) ← 0, parent(r) ← NIL
6 While Q ≠ ∅
7     u ← ExtractMin(Q)
```

Vediamo ora come implementare in maniera efficiente l'algoritmo di Prim per MST.

```
  MST-PRIM(G = (V, E), c, r)
1 Q ← V
2 For each u ∈ Q
3     a(u) ← ∞
4 a(r) ← 0, parent(r) ← NIL
6 While Q ≠ ∅
7     u ← ExtractMin(Q)
8        For each v ∈ Adj[u]
```

Vediamo ora come implementare in maniera efficiente l'algoritmo di Prim per MST.

```
  MST-PRIM(G = (V, E), c, r)
1 Q ← V
2 For each u ∈ Q
3     a(u) ← ∞
4 a(r) ← 0, parent(r) ← NIL
6 While Q ≠ ∅
7     u ← ExtractMin(Q)
8       For each v ∈ Adj[u]
9         If v ∈ Q and c(u, v) < a(v)
```

Vediamo ora come implementare in maniera efficiente l'algoritmo di Prim per MST.

```
  MST-PRIM(G = (V, E), c, r)
1 Q ← V
2 For each u ∈ Q
3     a(u) ← ∞
4 a(r) ← 0, parent(r) ← NIL
6 While Q ≠ ∅
7     u ← ExtractMin(Q)
8       For each v ∈ Adj[u]
9         If v ∈ Q and c(u, v) < a(v)
10          Then parent(v) ← u
```

Vediamo ora come implementare in maniera efficiente l'algoritmo di Prim per MST.

```
  MST-PRIM(G = (V, E), c, r)
1 Q ← V
2 For each u ∈ Q
3     a(u) ← ∞
4 a(r) ← 0, parent(r) ← NIL
6 While Q ≠ ∅
7     u ← ExtractMin(Q)
8       For each v ∈ Adj[u]
9         If v ∈ Q and c(u, v) < a(v)
10          Then parent(v) ← u
11          a(v) ← c(u, v); DecreaseKey(Q, v, a(v))
```

Vediamo ora come implementare in maniera efficiente l'algoritmo di Prim per MST.

```
  MST-PRIM(G = (V, E), c, r)
1 Q ← V
2 For each u ∈ Q
3     a(u) ← ∞
4 a(r) ← 0, parent(r) ← NIL
6 While Q ≠ ∅
7     u ← ExtractMin(Q)
8        For each v ∈ Adj[u]
9          If v ∈ Q and c(u, v) < a(v)
10           Then parent(v) ← u
11           a(v) ← c(u, v); DecreaseKey(Q, v, a(v))
```

Le istruzioni di inizializzazione 1-3 prendono tempo $O(|V|)$.

Vediamo ora come implementare in maniera efficiente l'algoritmo di Prim per MST.

```
  MST-PRIM(G = (V, E), c, r)
1 Q ← V
2 For each u ∈ Q
3     a(u) ← ∞
4 a(r) ← 0, parent(r) ← NIL
6 While Q ≠ ∅
7     u ← ExtractMin(Q)
8        For each v ∈ Adj[u]
9          If v ∈ Q and c(u, v) < a(v)
10           Then parent(v) ← u
11           a(v) ← c(u, v); DecreaseKey(Q, v, a(v))
```

Le istruzioni di inizializzazione 1-3 prendono tempo $O(|V|)$. All'interno del **While** vengono esaminati (una sola volta!) tutti i vertici e gli archi incidenti su di essi.

Vediamo ora come implementare in maniera efficiente l'algoritmo di Prim per MST.

```
  MST-PRIM(G = (V, E), c, r)
1 Q ← V
2 For each u ∈ Q
3     a(u) ← ∞
4 a(r) ← 0, parent(r) ← NIL
6 While Q ≠ ∅
7     u ← ExtractMin(Q)
8        For each v ∈ Adj[u]
9          If v ∈ Q and c(u, v) < a(v)
10           Then parent(v) ← u
11           a(v) ← c(u, v); DecreaseKey(Q, v, a(v))
```

Le istruzioni di inizializzazione 1-3 prendono tempo $O(|V|)$. All'interno del **While** vengono esaminati (una sola volta!) tutti i vertici e gli archi incidenti su di essi. Vengono eseguite $|V|$ operazioni di `ExtractMin`

Vediamo ora come implementare in maniera efficiente l'algoritmo di Prim per MST.

```
  MST-PRIM(G = (V, E), c, r)
1 Q ← V
2 For each u ∈ Q
3     a(u) ← ∞
4 a(r) ← 0, parent(r) ← NIL
6 While Q ≠ ∅
7     u ← ExtractMin(Q)
8        For each v ∈ Adj[u]
9          If v ∈ Q and c(u, v) < a(v)
10           Then parent(v) ← u
11           a(v) ← c(u, v); DecreaseKey(Q, v, a(v))
```

Le istruzioni di inizializzazione 1-3 prendono tempo $O(|V|)$. All'interno del **While** vengono esaminati (una sola volta!) tutti i vertici e gli archi incidenti su di essi. Vengono eseguite $|V|$ operazioni di `ExtractMin` (tempo $O(\log |V|)$ ciascuna se usiamo uno heap per implementare la coda a priorità),

Vediamo ora come implementare in maniera efficiente l'algoritmo di Prim per MST.

```
  MST-PRIM(G = (V, E), c, r)
1 Q ← V
2 For each u ∈ Q
3     a(u) ← ∞
4 a(r) ← 0, parent(r) ← NIL
6 While Q ≠ ∅
7     u ← ExtractMin(Q)
8        For each v ∈ Adj[u]
9          If v ∈ Q and c(u, v) < a(v)
10           Then parent(v) ← u
11           a(v) ← c(u, v); DecreaseKey(Q, v, a(v))
```

Le istruzioni di inizializzazione 1-3 prendono tempo $O(|V|)$. All'interno del **While** vengono esaminati (una sola volta!) tutti i vertici e gli archi incidenti su di essi. Vengono eseguite $|V|$ operazioni di `ExtractMin` (tempo $O(\log |V|)$ ciascuna se usiamo uno heap per implementare la coda a priorità), per un lavoro totale di $O(|V| \log |V|)$.

Vediamo ora come implementare in maniera efficiente l'algoritmo di Prim per MST.

```
  MST-PRIM(G = (V, E), c, r)
1 Q ← V
2 For each u ∈ Q
3     a(u) ← ∞
4 a(r) ← 0, parent(r) ← NIL
6 While Q ≠ ∅
7     u ← ExtractMin(Q)
8        For each v ∈ Adj[u]
9          If v ∈ Q and c(u, v) < a(v)
10           Then parent(v) ← u
11           a(v) ← c(u, v); DecreaseKey(Q, v, a(v))
```

Le istruzioni di inizializzazione 1-3 prendono tempo $O(|V|)$. All'interno del **While** vengono esaminati (una sola volta!) tutti i vertici e gli archi incidenti su di essi. Vengono eseguite $|V|$ operazioni di `ExtractMin` (tempo $O(\log |V|)$ ciascuna se usiamo uno heap per implementare la coda a priorità), per un lavoro totale di $O(|V| \log |V|)$. Vengono eseguite $|E|$ operazioni di `DecreaseKey`

Vediamo ora come implementare in maniera efficiente l'algoritmo di Prim per MST.

```
  MST-PRIM(G = (V, E), c, r)
1 Q ← V
2 For each u ∈ Q
3     a(u) ← ∞
4 a(r) ← 0, parent(r) ← NIL
6 While Q ≠ ∅
7     u ← ExtractMin(Q)
8        For each v ∈ Adj[u]
9          If v ∈ Q and c(u, v) < a(v)
10           Then parent(v) ← u
11           a(v) ← c(u, v); DecreaseKey(Q, v, a(v))
```

Le istruzioni di inizializzazione 1-3 prendono tempo $O(|V|)$. All'interno del **While** vengono esaminati (una sola volta!) tutti i vertici e gli archi incidenti su di essi. Vengono eseguite $|V|$ operazioni di `ExtractMin` (tempo $O(\log |V|)$ ciascuna se usiamo uno heap per implementare la coda a priorità), per un lavoro totale di $O(|V| \log |V|)$. Vengono eseguite $|E|$ operazioni di `DecreaseKey` (tempo $O(\log |V|)$ ciascuna)

Vediamo ora come implementare in maniera efficiente l'algoritmo di Prim per MST.

```
  MST-PRIM(G = (V, E), c, r)
1 Q ← V
2 For each u ∈ Q
3     a(u) ← ∞
4 a(r) ← 0, parent(r) ← NIL
6 While Q ≠ ∅
7     u ← ExtractMin(Q)
8       For each v ∈ Adj[u]
9         If v ∈ Q and c(u, v) < a(v)
10          Then parent(v) ← u
11          a(v) ← c(u, v); DecreaseKey(Q, v, a(v))
```

Le istruzioni di inizializzazione 1-3 prendono tempo $O(|V|)$. All'interno del **While** vengono esaminati (una sola volta!) tutti i vertici e gli archi incidenti su di essi. Vengono eseguite $|V|$ operazioni di `ExtractMin` (tempo $O(\log |V|)$ ciascuna se usiamo uno heap per implementare la coda a priorità), per un lavoro totale di $O(|V| \log |V|)$. Vengono eseguite $|E|$ operazioni di `DecreaseKey` (tempo $O(\log |V|)$ ciascuna) per un lavoro totale di $O(|E| \log |V|)$.

Vediamo ora come implementare in maniera efficiente l'algoritmo di Prim per MST.

```
  MST-PRIM(G = (V, E), c, r)
1 Q ← V
2 For each u ∈ Q
3     a(u) ← ∞
4 a(r) ← 0, parent(r) ← NIL
6 While Q ≠ ∅
7     u ← ExtractMin(Q)
8       For each v ∈ Adj[u]
9         If v ∈ Q and c(u, v) < a(v)
10          Then parent(v) ← u
11          a(v) ← c(u, v); DecreaseKey(Q, v, a(v))
```

Le istruzioni di inizializzazione 1-3 prendono tempo $O(|V|)$. All'interno del **While** vengono esaminati (una sola volta!) tutti i vertici e gli archi incidenti su di essi. Vengono eseguite $|V|$ operazioni di `ExtractMin` (tempo $O(\log |V|)$ ciascuna se usiamo uno heap per implementare la coda a priorità), per un lavoro totale di $O(|V| \log |V|)$. Vengono eseguite $|E|$ operazioni di `DecreaseKey` (tempo $O(\log |V|)$ ciascuna) per un lavoro totale di $O(|E| \log |V|)$. Gran totale$=O(|E| \log |V|)$.

Che succede se esistono archi di costo uguale?

Che succede se esistono archi di costo uguale?

Non cambia nulla.

Che succede se esistono archi di costo uguale?

Non cambia nulla. Innanzitutto possiamo “perturbare” i costi degli archi di una piccola quantità in modo che i nuovi costi siano ora tutti diversi tra di loro,

Che succede se esistono archi di costo uguale?

Non cambia nulla. Innanzitutto possiamo "perturbare" i costi degli archi di una piccola quantità in modo che i nuovi costi siano ora tutti diversi tra di loro, ed in modo che l'ordine relativo tra i costi degli archi rimanga *inalterato* rispetto a prima,

Che succede se esistono archi di costo uguale?

Non cambia nulla. Innanzitutto possiamo "perturbare" i costi degli archi di una piccola quantità in modo che i nuovi costi siano ora tutti diversi tra di loro, ed in modo che l'ordine relativo tra i costi degli archi rimanga *inalterato* rispetto a prima, cosicchè i due algoritmi prima visti effettuino l'esame degli archi nello stesso ordine,

Che succede se esistono archi di costo uguale?

Non cambia nulla. Innanzitutto possiamo "perturbare" i costi degli archi di una piccola quantità in modo che i nuovi costi siano ora tutti diversi tra di loro, ed in modo che l'ordine relativo tra i costi degli archi rimanga *inalterato* rispetto a prima, cosicchè i due algoritmi prima visti effettuino l'esame degli archi nello stesso ordine, (cioè sia nel caso in cui i costi siano uguali che nel nuovo caso in cui sono diversi).

Che succede se esistono archi di costo uguale?

Non cambia nulla. Innanzitutto possiamo "perturbare" i costi degli archi di una piccola quantità in modo che i nuovi costi siano ora tutti diversi tra di loro, ed in modo che l'ordine relativo tra i costi degli archi rimanga *inalterato* rispetto a prima, cosicchè i due algoritmi prima visti effettuino l'esame degli archi nello stesso ordine, (cioè sia nel caso in cui i costi siano uguali che nel nuovo caso in cui sono diversi). Inoltre, ogni albero T che è MST per i nuovi costi è MST anche per i vecchi costi.

Che succede se esistono archi di costo uguale?

Non cambia nulla. Innanzitutto possiamo "perturbare" i costi degli archi di una piccola quantità in modo che i nuovi costi siano ora tutti diversi tra di loro, ed in modo che l'ordine relativo tra i costi degli archi rimanga *inalterato* rispetto a prima, cosicchè i due algoritmi prima visti effettuino l'esame degli archi nello stesso ordine, (cioè sia nel caso in cui i costi siano uguali che nel nuovo caso in cui sono diversi). Inoltre, ogni albero T che è MST per i nuovi costi è MST anche per i vecchi costi. Infatti, se per assurdo esistesse un T^* per i vecchi costi per cui
costo(T^*) <costo(T),

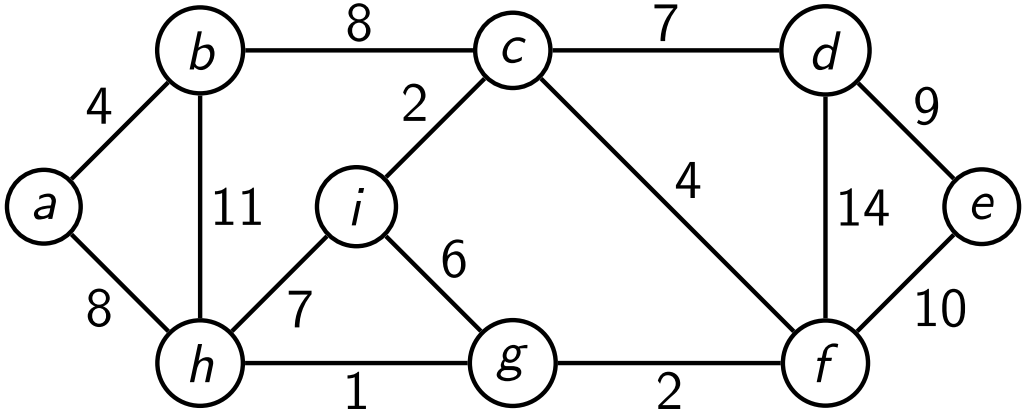
Che succede se esistono archi di costo uguale?

Non cambia nulla. Innanzitutto possiamo "perturbare" i costi degli archi di una piccola quantità in modo che i nuovi costi siano ora tutti diversi tra di loro, ed in modo che l'ordine relativo tra i costi degli archi rimanga *inalterato* rispetto a prima, cosicchè i due algoritmi prima visti effettuino l'esame degli archi nello stesso ordine, (cioè sia nel caso in cui i costi siano uguali che nel nuovo caso in cui sono diversi). Inoltre, ogni albero T che è MST per i nuovi costi è MST anche per i vecchi costi. Infatti, se per assurdo esistesse un T^* per i vecchi costi per cui costo(T^*) <costo(T), allora per una perturbazione sufficientemente piccola che trasforma i costi $c(u, v)$ degli archi in nuovi costi $c'(u, v)$

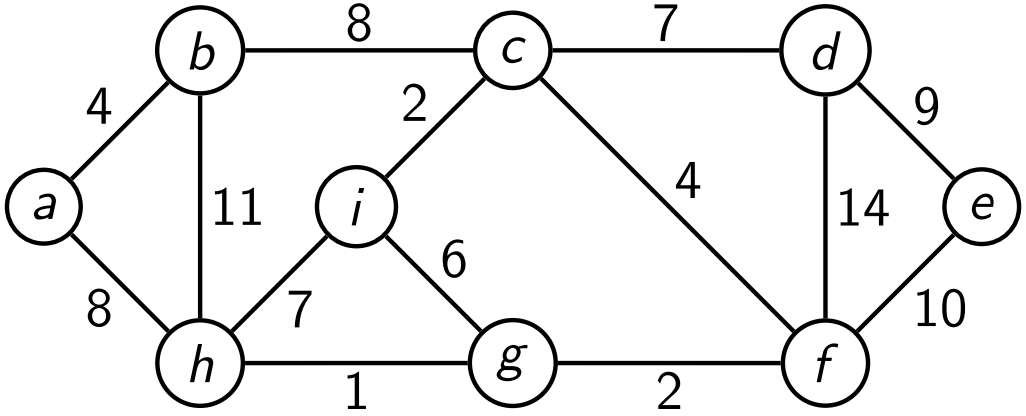
Che succede se esistono archi di costo uguale?

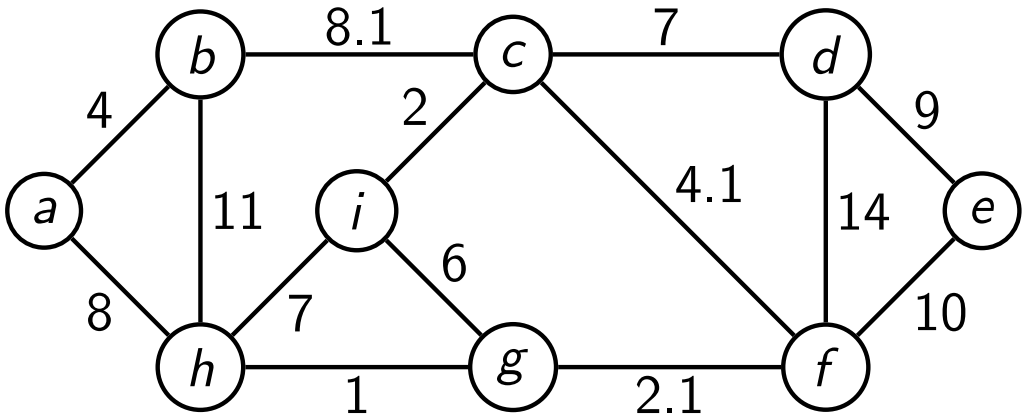
Non cambia nulla. Innanzitutto possiamo "perturbare" i costi degli archi di una piccola quantità in modo che i nuovi costi siano ora tutti diversi tra di loro, ed in modo che l'ordine relativo tra i costi degli archi rimanga *inalterato* rispetto a prima, cosicchè i due algoritmi prima visti effettuino l'esame degli archi nello stesso ordine, (cioè sia nel caso in cui i costi siano uguali che nel nuovo caso in cui sono diversi). Inoltre, ogni albero T che è MST per i nuovi costi è MST anche per i vecchi costi. Infatti, se per assurdo esistesse un T^* per i vecchi costi per cui costo(T^*) <costo(T), allora per una perturbazione sufficientemente piccola che trasforma i costi $c(u, v)$ degli archi in nuovi costi $c'(u, v)$ continuerebbe a valere costo$'(T^*)$ <costo$'(T)$, contro l'ipotesi che T è un MST per i nuovi costi.

$(h, g), (i, c), (g, f), (a, b), (c, f), (i, g), (c, d), (i, h), (a, h), (b, c),$
$(d, e), (b, h), (d, f)$

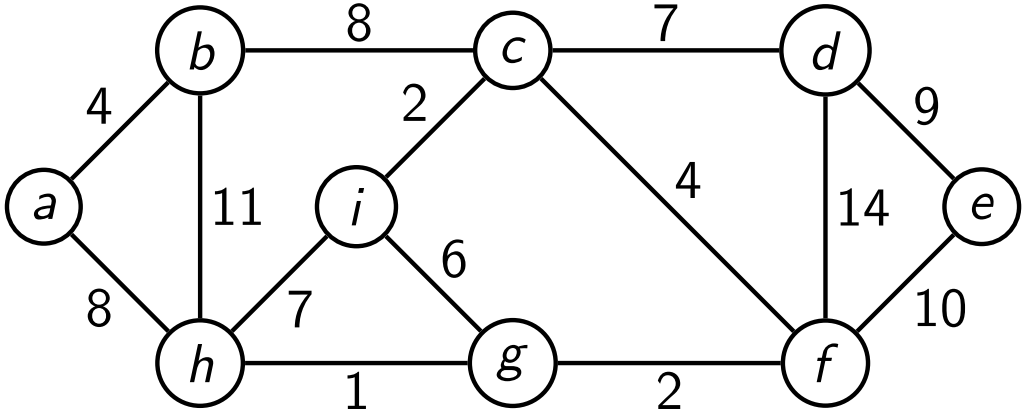


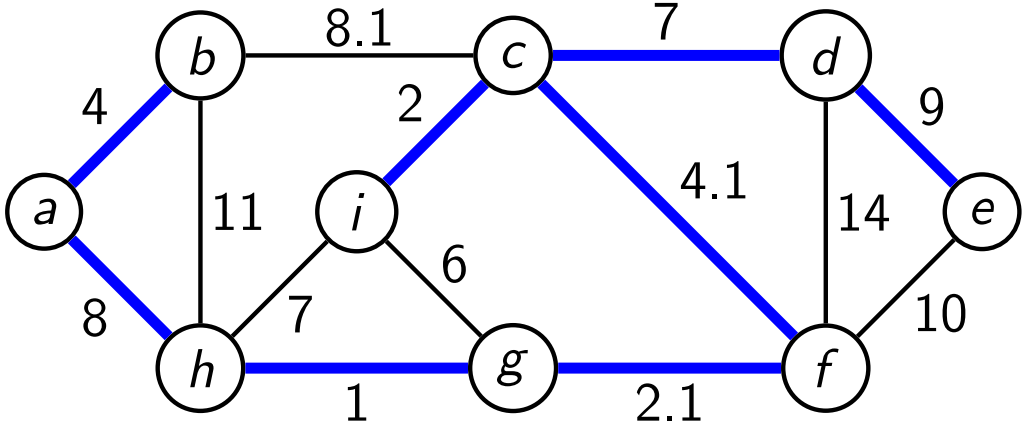
$(h, g), (i, c), (g, f), (a, b), (c, f), (i, g), (c, d), (i, h), (a, h), (b, c),$
$(d, e), (b, h), (d, f)$



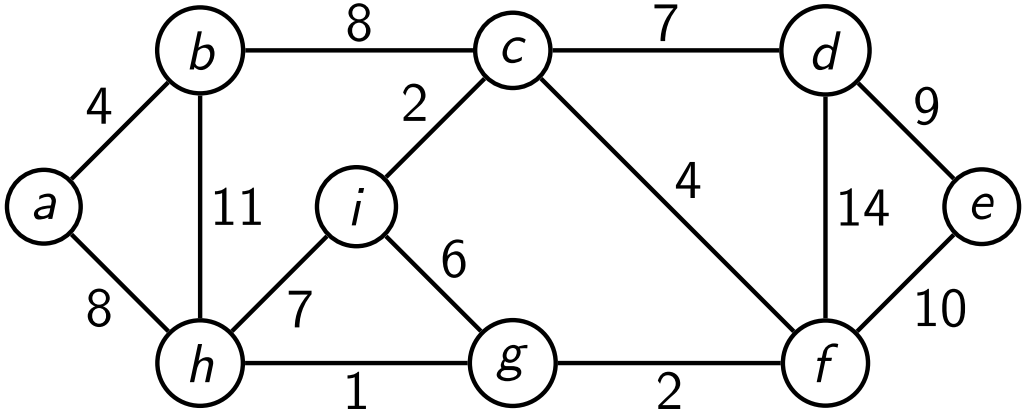


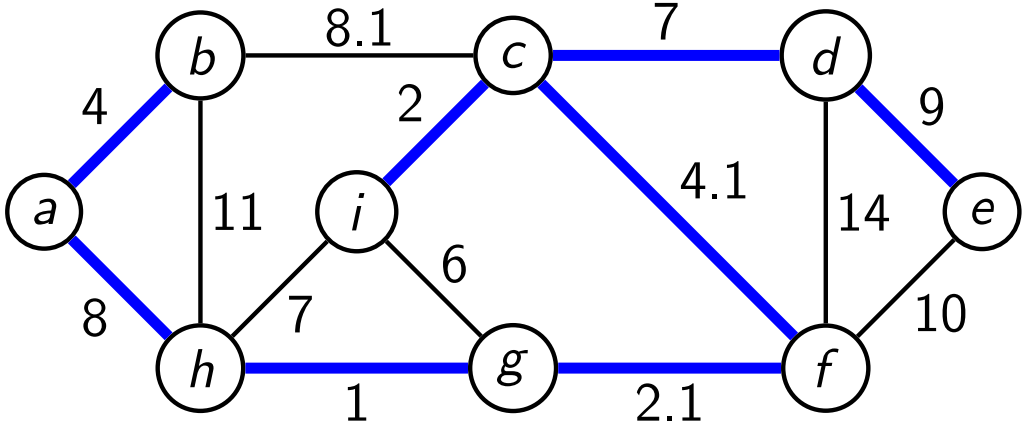
$(h, g), (i, c), (g, f), (a, b), (c, f), (i, g), (c, d), (i, h), (a, h), (b, c),$
$(d, e), (b, h), (d, f)$





$(h,g),(i,c),(g,f),(a,b),(c,f),(i,g),(c,d),(i,h),(a,h),(b,c),$
$(d,e),(b,h),(d,f)$





Peso totale=1+2+2.1+4+4.1+7+9=29.2