Algoritmi greedy
III parte

Progettazione di Algoritmi 2017-18

Matricole congrue a 1

Docente: Annalisa De Bonis

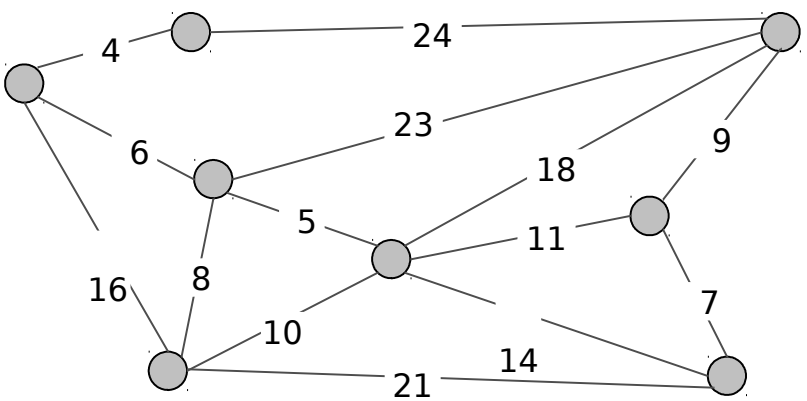
Minimo albero ricoprente (Minimum spanning tree)

- Supponiamo di avere un insieme di pin $V = \{v_1, v_2, \ldots, v_n\}$ che decono essere interconnessi in modo che per ogni coppia di pin esista un percorso che li collega.
- Alcune coppie di pin possono essere collegate direttamente.
- Stabilire un collegamento diretto tra una coppia di pin ha un costo che dipende dalla posizione dei due pin collegati
- L'obiettivo è di utilizzare esattamente n-1 di questi collegamenti diretti tra coppie di pin in modo da connettere l'intera rete e da minimizzare la somma dei costi degli n-1 collegamenti stabiliti
- Altri esempi di applicazione alla progettazione di una rete sono.
  - Reti telefoniche, idriche, televisive, stradali, di computer

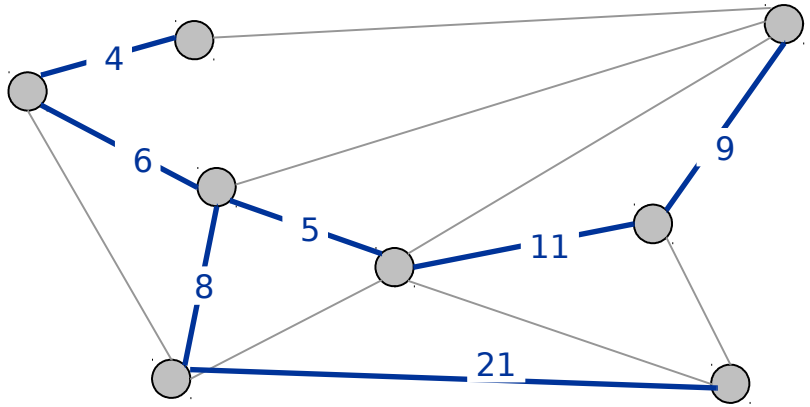
Minimo albero ricoprente (Minimum spanning tree o in breve MST)

- Grafo non direzionato connesso G = (V, E).
- Per ogni arco e, c_e = costo dell'arco e (c_e numero reale).

- Def. Albero ricoprente (spanning tree). Sia dato un grafo non direzionato connesso G = (V, E). Uno spanning tree di G è un sottoinsieme di archi $T \subseteq E$ tale che |T|=n-1 e gli archi in T non formano cicli (in altre parole T forma un albero che ha come nodi tutti i nodi di G).
- Def. Sia dato un grafo non direzionato connesso G = (V, E) tale che ad ogni arco e di G è associato un costo c_e .Per ogni albero ricoprente T di G , definiamo il costo di T come $\sum_{e \in T} c_e$



G = (V, E)

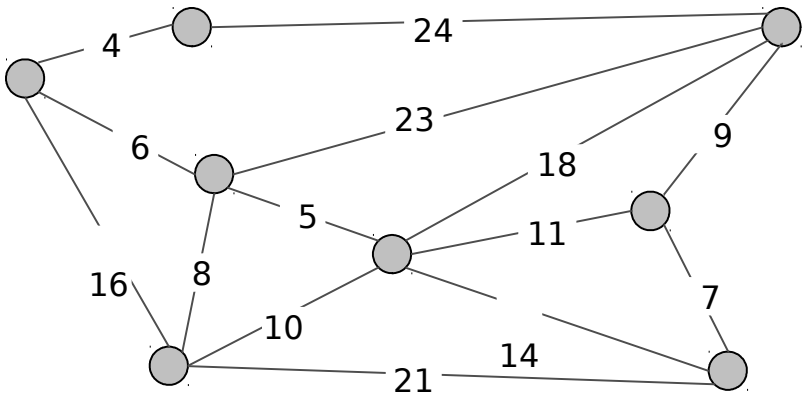


T, $\Sigma_{e \in T} c_e = 64$

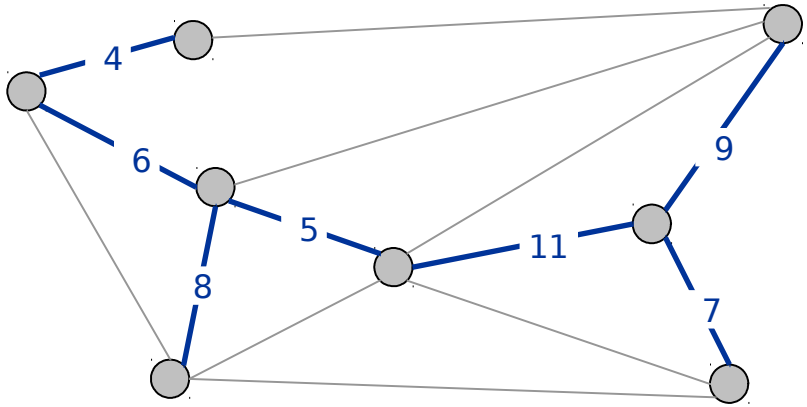
Minimo albero ricoprente (Minimum spanning tree o in breve MST)

- Input:
  - Grafo non direzionato connesso G = (V, E).
  - Per ogni arco e, c_e = costo dell'arco e.

✂ Minimo albero ricoprente. Sia dato un grafo non direzionato connesso G = (V, E) con costi c_e degli archi a valori reali. Un minimo albero ricoprente è un sottoinsieme di archi $T \subseteq E$ tale che T è un albero ricoprente di costo minimo.



G = (V, E)



T, $\Sigma_{e \in T} c_e = 50$

Minimo albero ricoprente (Minimum spanning tree o in breve MST)

- Il problema di trovare un minimo albero ricoprente non può essere risolto con un algoritmo di forza bruta
- Teorema di Cayley. Ci sono n^{n-2} alberi ricoprenti del grafo completo K_n.

Algoritmi greedy per MST

- Kruskal. Comincia con T = ϕ. Considera gli archi in ordine non decrescente di costo. Inserisce un arco e in T se e solo il suo inserimento non determina la creazione di un ciclo in T

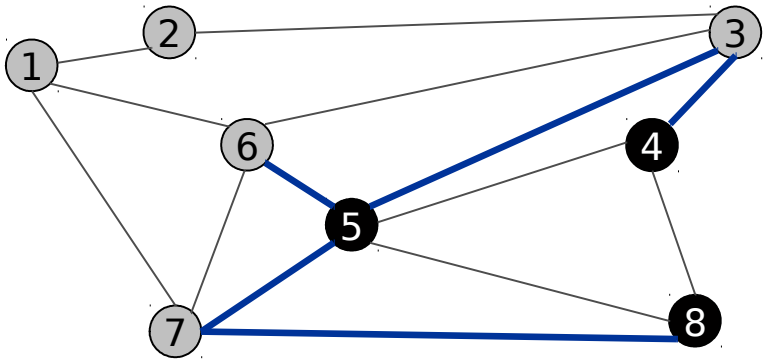
- Inverti-Cancella. Comincia con T = E. Considera gli archi in ordine non crescente dei costi. Cancella e da T se e solo se la sua cancellazione non rende T disconnesso.

- Prim. Comincia con un certo nodo s e costruisce un albero T avente s come radice. Ad ogni passo aggiunge a T l'arco di peso più basso tra quelli che hanno esattamente una delle due estremità in T (se un arco avesse entrambe le estremità in T, la sua introduzione in T creerebbe un ciclo)

Taglio

- Taglio. Un taglio è una partizione [S,V-S] dell'insieme dei vertici del grafo.

- Insieme di archi che attraversano il taglio [S,V-S]. Sottoinsieme D di archi che hanno un'estremità in S e una in V-S.



Taglio [S,V-S] = ({ 4, 5, 8 }, { 1,2,3,6,7 })
Archi che attraversano [S,V-S] D = 5-6, 5-7, 3-4, 3-5, 7-8

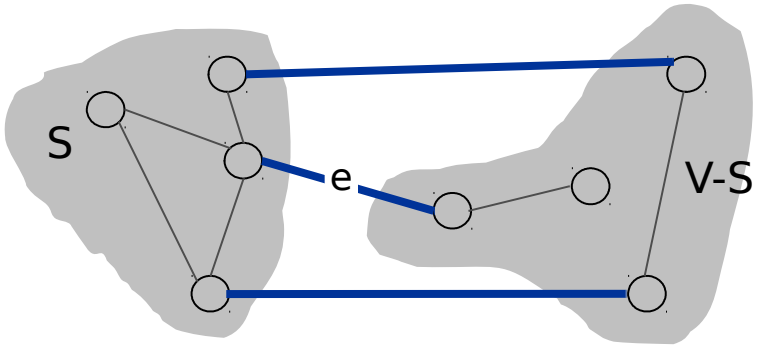
Algoritmi Greedy

Per il momento assumiamo per semplicità che i pesi degli archi siano a due a due distinti

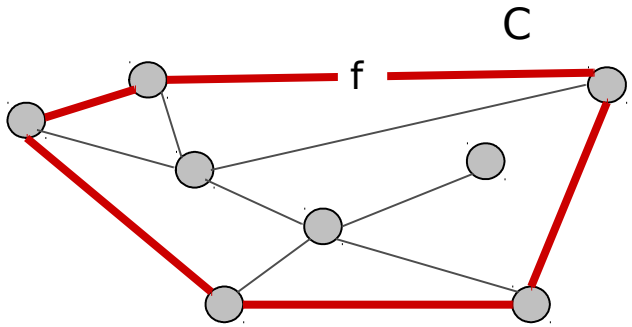
- Vedremo che questa assunzione implica che il minimo albero ricoprente è unico.

Proprietà del taglio. Sia S un qualsiasi sottoinsieme di nodi e sia e l'arco di costo minimo che attraversa il taglio [S,V-S]. Ogni minimo albero ricoprente contiene e.

Proprietà del ciclo. Sia C un qualsiasi ciclo e sia f l'arco di costo massimo in C. Nessun minimo albero ricoprente contiene f.



e è nello MST



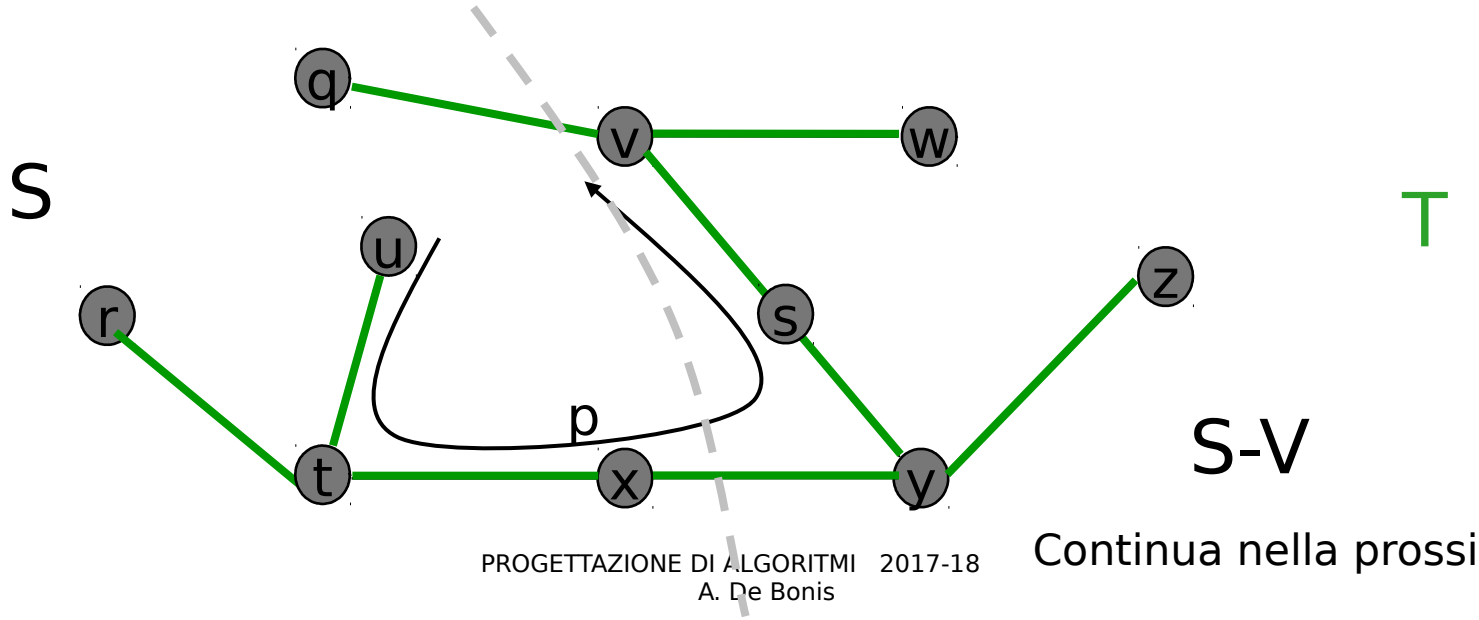
f non è nello MST

Proprietà del taglio

- Stiamo assumendo (per il momento) che i costi degli archi siano a due a due distinti
- Proprietà del taglio. Sia S un qualsiasi sottoinsieme di nodi e sia e=(u,v) l'arco di costo minimo che attraversa il taglio [S,V-S]. Ogni minimo albero ricoprente contiene e.

Dim. (nel caso in cui i costi degli archi sono a due a due distinti)

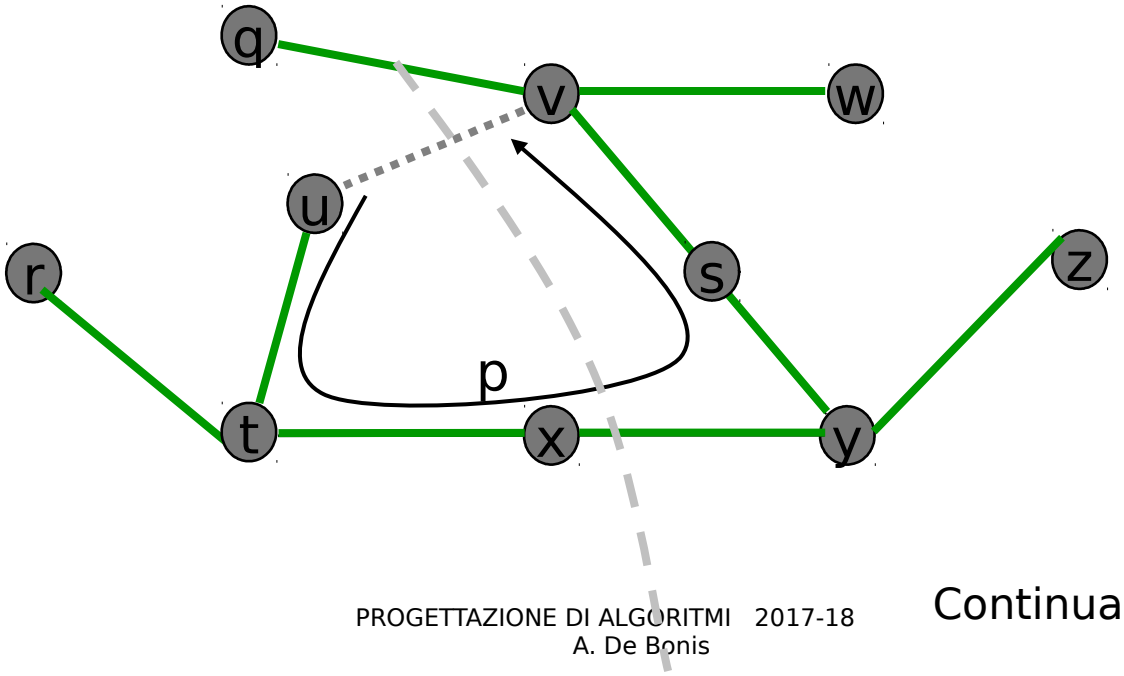
- Sia T un albero ricoprente tale che e=(u,v) $\notin$ T. Dimostriamo che T non può essere un minimo albero ricoprente.
- Sia p il percorso da u a v in T. In T non ci sono altri percorsi da u a v altrimenti ci sarebbe un ciclo



Continua nella prossima slide

Proprietà del taglio

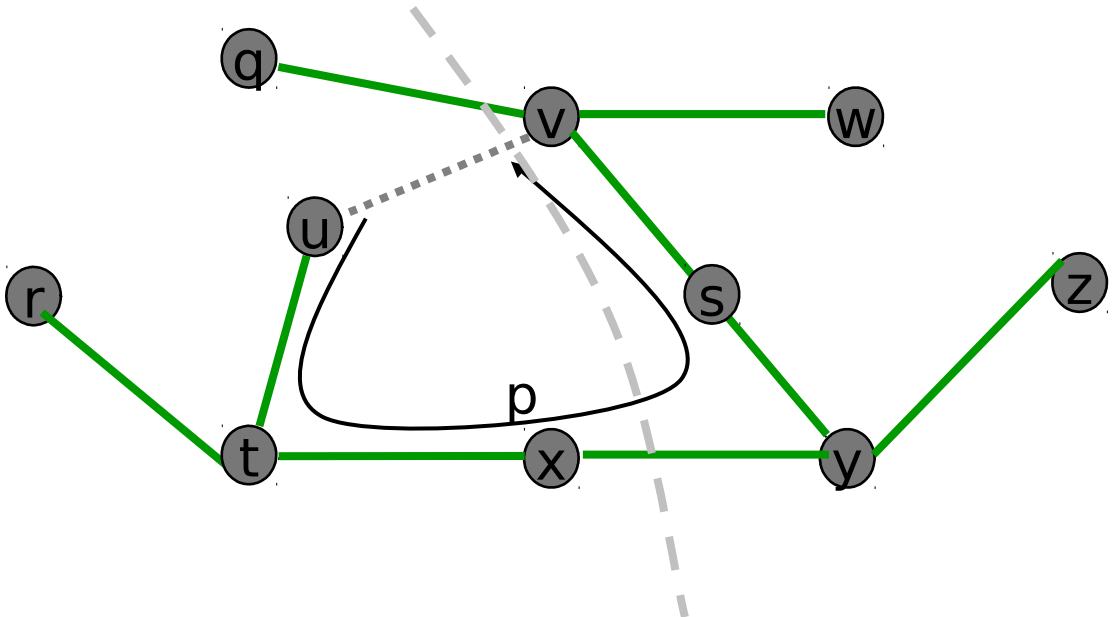
- Osserviamo che se aggiungiamo e=(u,v) a T generiamo un ciclo
- Siccome u e v sono ai lati opposti del taglio [S,V-S] allora il ciclo deve comprendere un arco f=(x,y) di T che attraversa il taglio [S,V-S]



Continua nella prossima slide

Proprietà del taglio

Poichè $(x,y) \neq (u,v)$ e poichè entrambi attraversano il taglio e (u,v) è l'arco di peso minimo tra quelli che attraversano il taglio allora si ha $c_e < c_f$

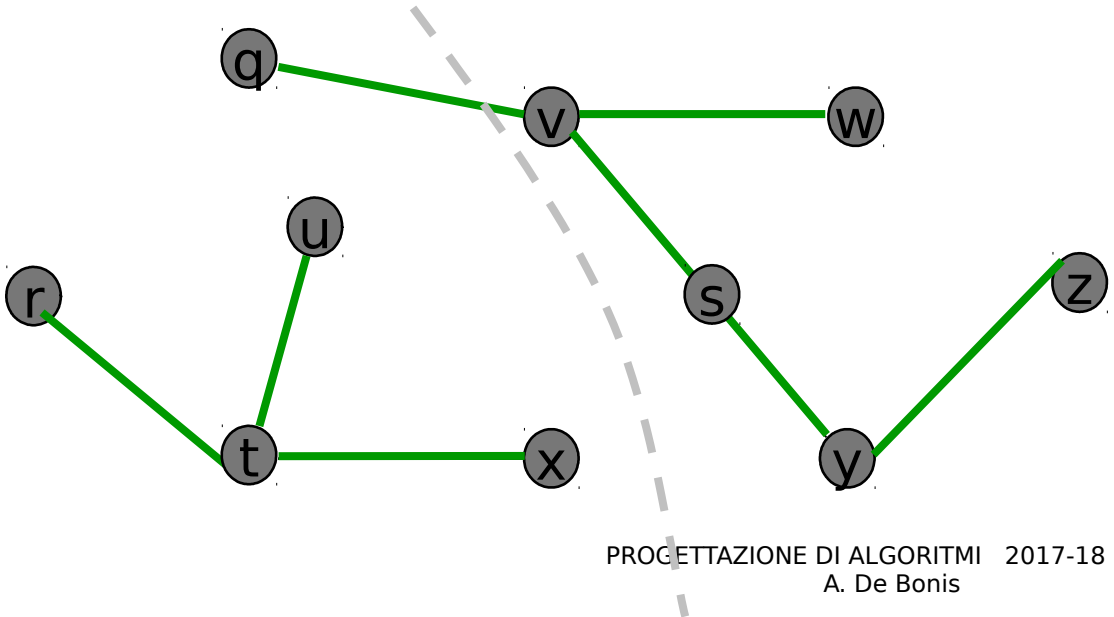


Proprietà del taglio

- Poichè (x,y) ≠ (u,v) e poichè entrambi attraversano il taglio e (u,v) è l'arco di peso minimo tra quelli che attraversano il taglio allora si ha $c_e < c_f$

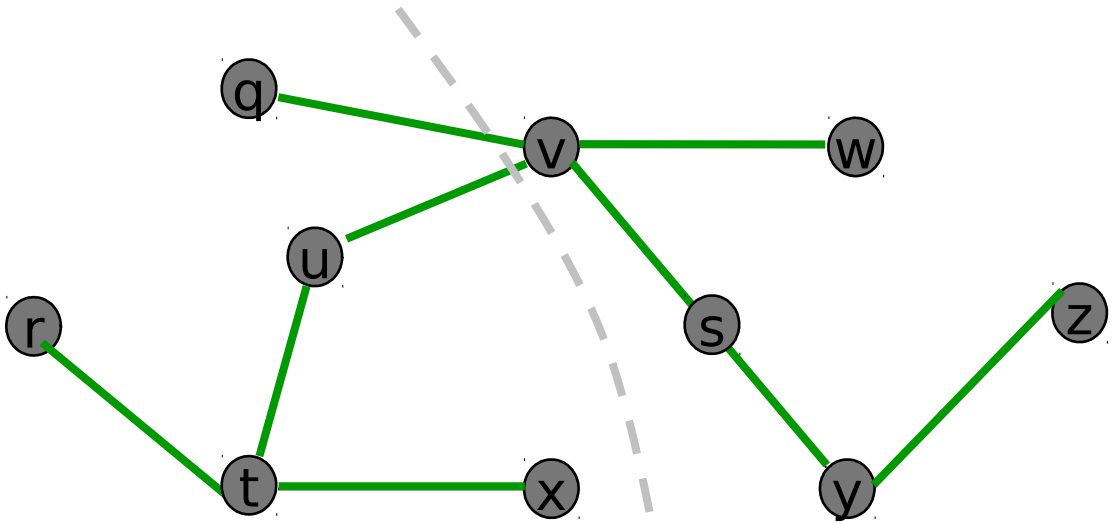
  - f=(x,y) si trova sull'unico percorso che connette u a v in T

  - Se togliamo f=(x,y) da T dividiamo T in due alberi, uno contente u e l'altro contenente v



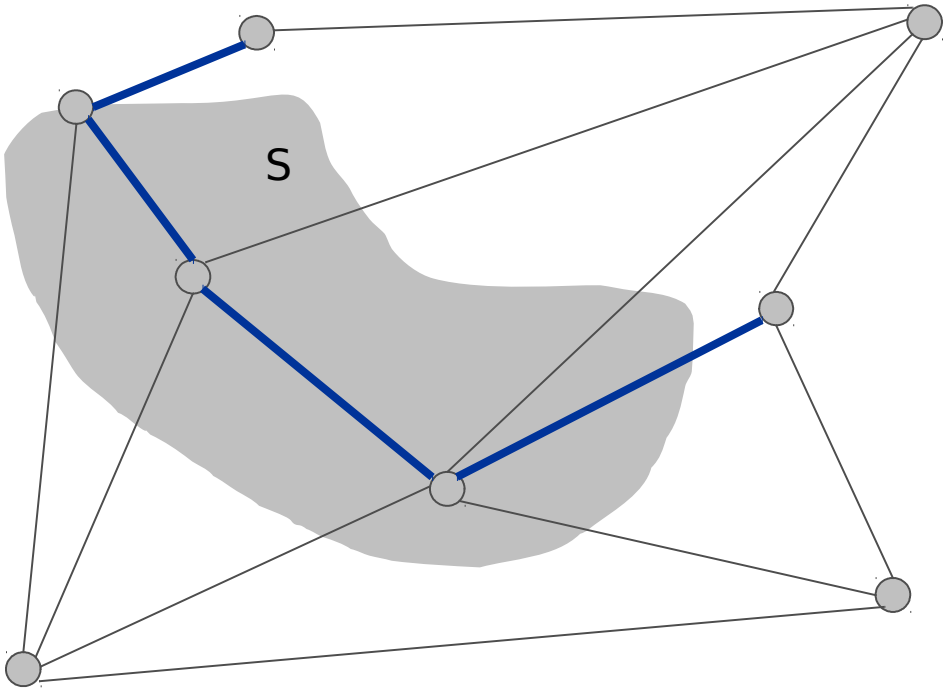
Proprietà del taglio

- Poichè (x,y) ≠ (u,v) e poichè entrambi attraversano il taglio e (u,v) è l'arco di peso minimo tra quelli che attraversano il taglio allora si ha $c_e < c_f$
- f=(x,y) si trova sull'unico percorso che connette u a v in T
- Se togliamo f=(x,y) da T dividiamo T in due alberi, uno contente u e l'altro contenente v
- Se introduciamo e=(u,v) riconnettiamo i due alberi ottenendo un nuovo albero T' = T-{f}∪{e} di costo $c(T')=c(T)-c_f+c_e<c(T)$. Ciò vuol dire che T non è un minimo albero ricoprente.



Algoritmo di Prim

- Algoritmo di Prim. [Jarník 1930, Prim 1957, Dijkstra 1959,]
- Ad ogni passo T è un sottoinsieme di archi dello MST
- S= insieme di nodi di T
  - Inizializzazione: Pone in S un qualsiasi nodo u. Il nodo u sarà la radice dello MST
  - Ad ogni passo aggiunge a T l'arco (x,y) di costo minimo tra tutti quelli che congiungono un nodo x in S ad un nodo y in V-S (scelta greedy)
  - Termina quando S=V

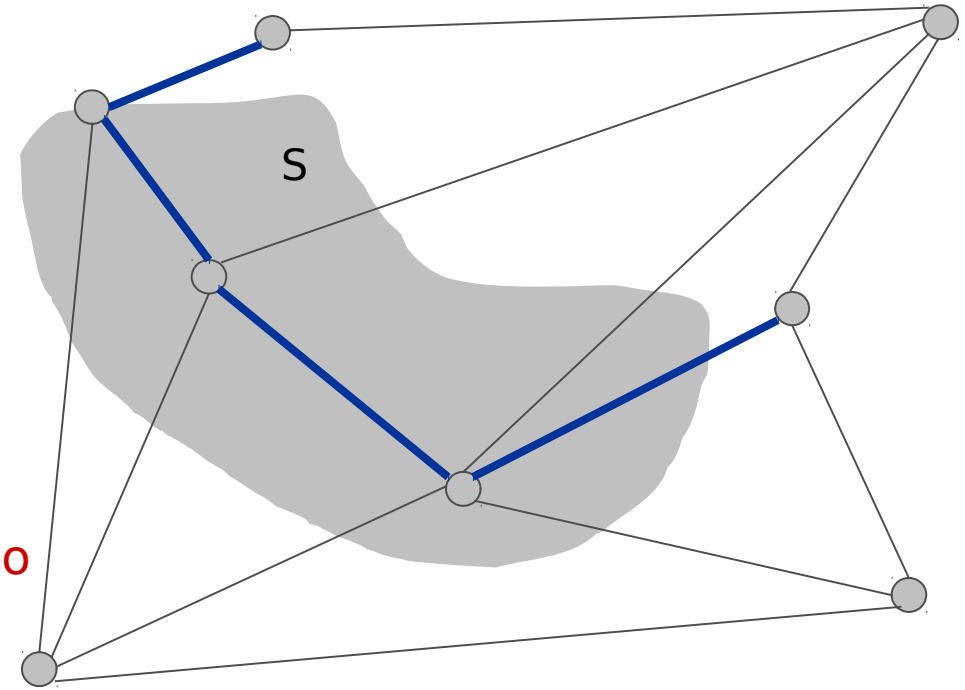


Correttezza dell'algoritmo di Prim

- L'algoritmo di Prim ottiene il minimo albero ricoprente in quanto ad ogni passo seleziona l'arco di costo minimo tra quelli che attraversano il taglio [S,V-S].
- La proprietà del taglio implica che ogni albero ricoprente deve contenere ciascuno degli archi selezionati dall'algoritmo → T sottoinsieme di MST
- Ci resta da dimostrare che T è un albero ricoprente. Ovviamente T non contiene cicli in quanto un arco viene inserito solo se una delle sue estremità non è ancora attaccata all'albero. Poiché inoltre l'algoritmo si ferma solo quando S=V cioè quando ha attaccato tutti i vertici all'albero, si ha che T è un albero ricoprente.

In conclusione T è un albero ricoprente che contiene esclusivamente archi che fanno parte dello MST e quindi T è lo MST.

NB: quando i costi sono a due a due distinti c'è un unico MST in quanto per ogni taglio c'è un unico arco di costo minimo che lo attraversa



Implementazione dell'algoritmo di Prim con coda a priorità

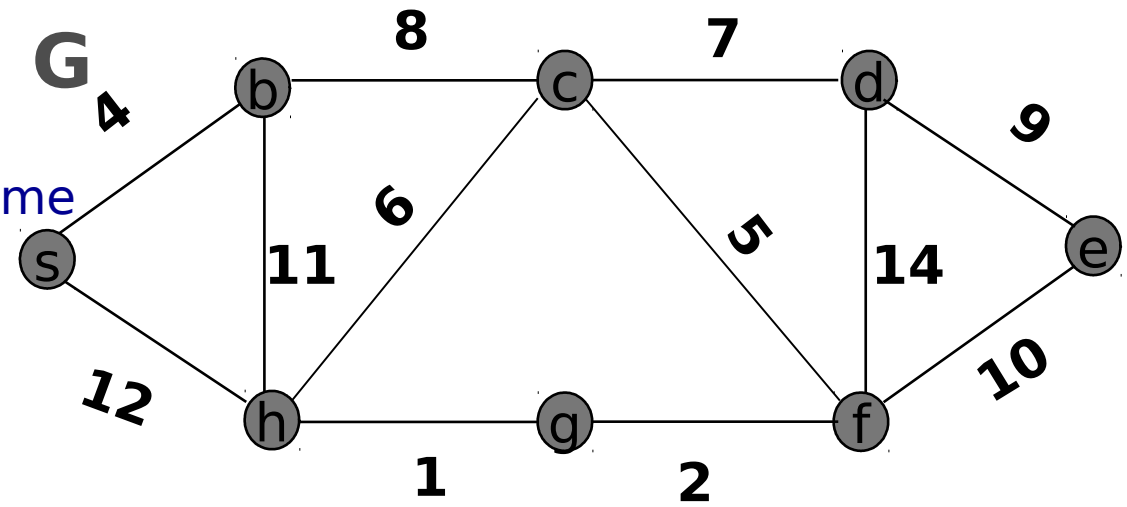
- Mantiene un insieme di vertici esplorati S.
- Per ogni nodo non esplorato v, mantiene a[v] = costo dell'arco di costo più basso tra quelli che uniscono v ad un nodo in S
- Mantiene coda a priorità Q delle coppie (a[v],v)
- Stessa analisi dell'algoritmo di Prim con coda a priorità:
- $O(n^2)$ con array o lista non ordinati;
- O(m log n+nlog n) con heap. Siccome nel problema dello MST il grafo è connesso allora m≥n-1 e O(m log n+nlog n) =O(mlog n)

```
Prim's Algorithm (G,c)
Let S be the set of explored nodes
For each u not in S, we store the cost a[u]
Let Q be a priority queue of nodes u with keys a[u] s.t. u is not
in S
S ← Ø
For each u in V insert (Q, ∞,u)  in Q EndFor
While (Q is not empty)
  u ← ExtractMin(Q)
  Add u to S
  For each edge e=(u,v)
   If  ((v not in S) && (c_e < a[v]))
     ChangeKey(Q,v, c_e)
EndWhile
```

Un esempio

In questo esempio, per ciascun nodo u manteniamo anche un campo **π[u]** che alla fine è uguale al padre di u nello MST

s = nodo scelto come radice



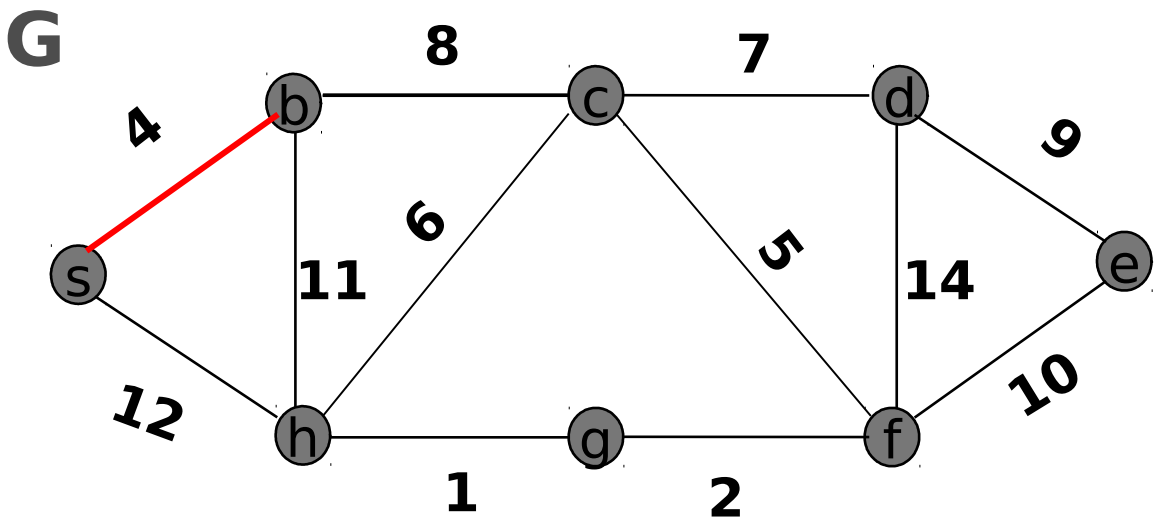
[u,a[u]]

Q = {[s, ∞],[b ,∞],[c, ∞],[d, ∞],[e, ∞],[f, ∞],[g, ∞],[h, ∞]}

Si estrae s da Q e si aggiornano i campi a e π dei nodi adiacenti ad s

Q = {[b ,4],[c, ∞],[d, ∞],[e, ∞],[f, ∞],[g, ∞],[h, 12]}

Un esempio



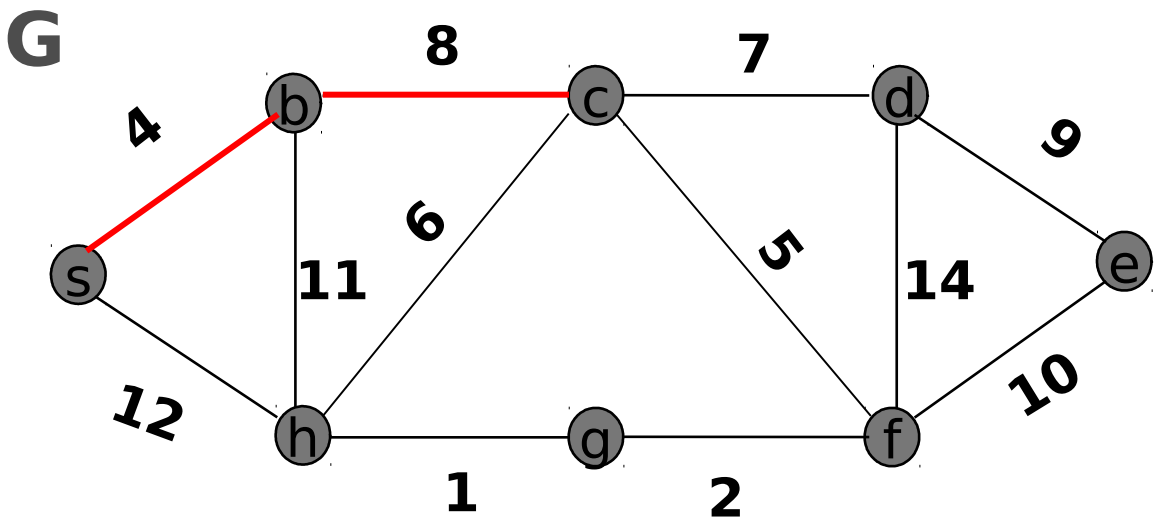
$Q = \{[b, 4],[c, \infty],[d, \infty],[e, \infty],[f, \infty],[g, \infty],[h, 12]\}$

- Si estrae b da Q.

- Si aggiornano i campi a e π dei nodi adiacenti a b che si trovano in Q

$$Q = \{[c,8],[d, \infty],[e, \infty],[f, 5],[g, \infty],[h, 11]\}$$

Un esempio



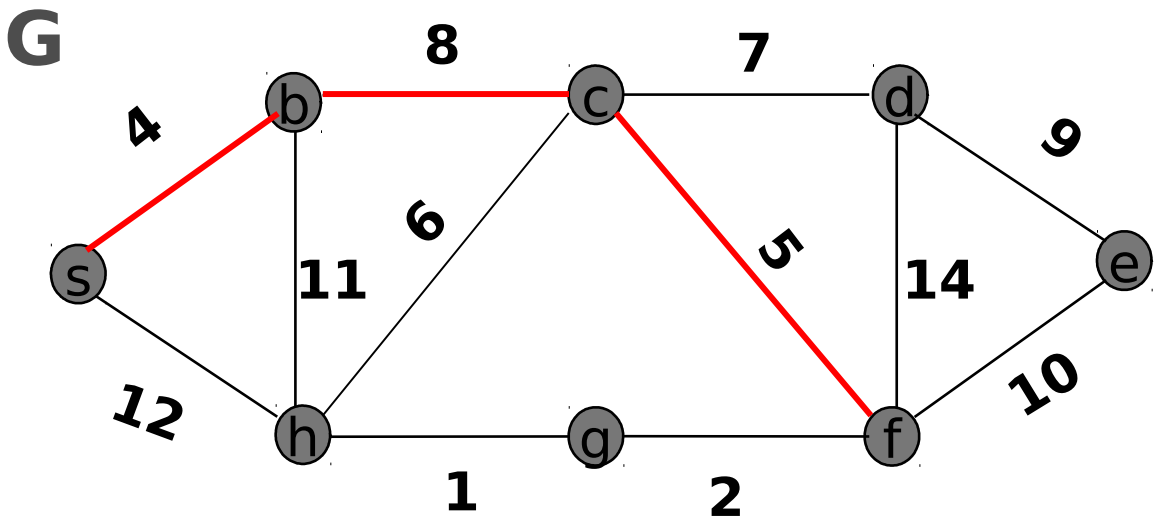
$$Q = \{[c, 8],[d, \infty],[e, \infty],[f, \infty],[g, \infty],[h, 12]\}$$

- Si estrae c da Q.

- Si aggiornano i campi a e π dei nodi adiacenti a c che si trovano in Q

$$Q = \{[d, 7],[e, \infty],[f, 5],[g, \infty],[h, 6]\}$$

Un esempio

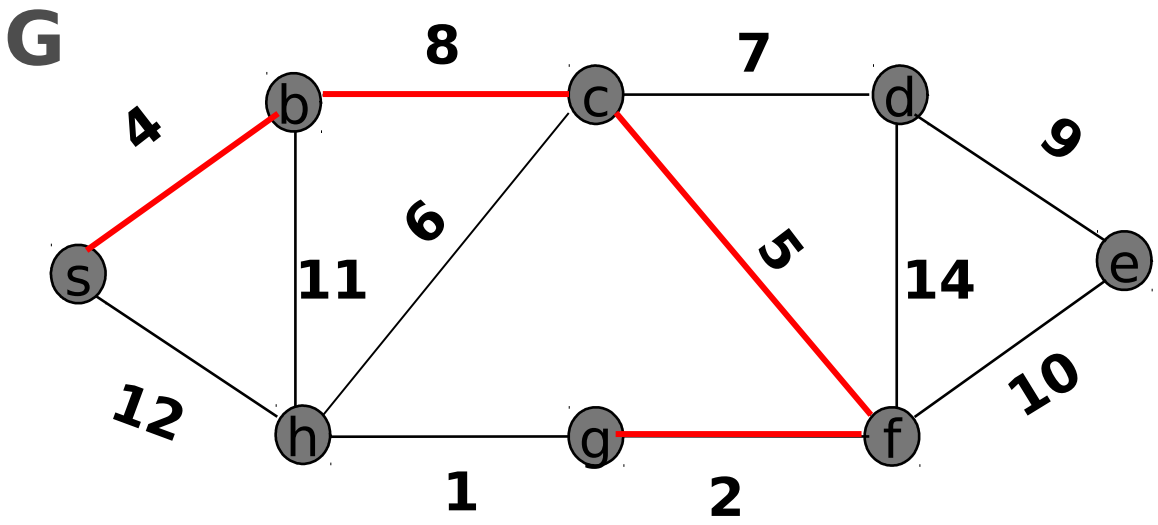


$$Q = \{[d, 7],[e, \infty],[f, 5],[g, \infty],[h, 6]\}$$

Si estrae f da Q e si aggiornano i campi a e π dei nodi adiacenti a f che si trovano in Q

$$Q = \{[d, 7],[e, 10],[g, 2],[h, 6]\}$$

Un esempio

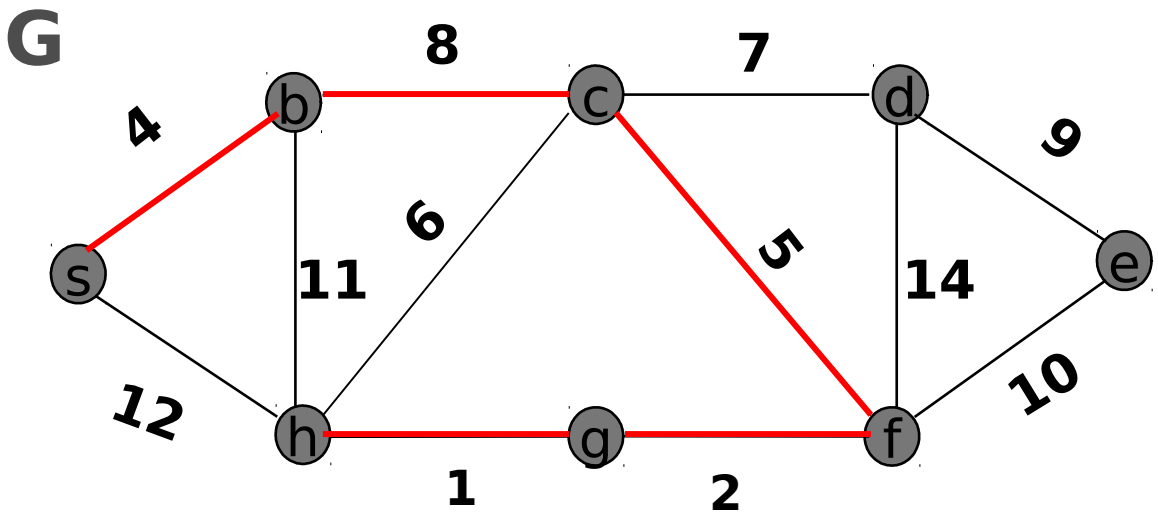


$$Q = \{[d, 7],[e, 10],[g, 2],[h, 6]\}$$

Si estrae g da Q e si aggiornano i campi a e π dei nodi adiacenti a g che si trovano in Q

$$Q = \{[d, 7],[e, 10],[h, 1]\}$$

Un esempio

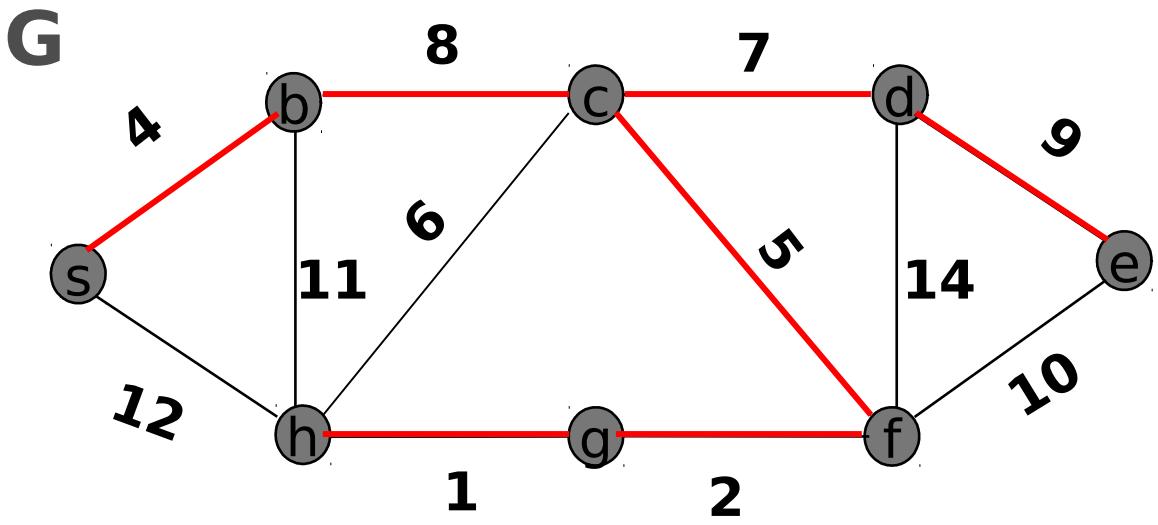


$$Q = \{[d, 7],[e, 10],[h, 1]\}$$

Si estrae h da Q e si aggiornano i campi a e π dei nodi adiacenti a h che si trovano in Q

$$Q = \{[d, 7],[e, 10]\}$$

Un esempio



$Q = \{[d, 7],[e, 10]\}$

Si estrae d da Q e si aggiorna il campo a e π di e

$Q = \{[e, 9]\}$

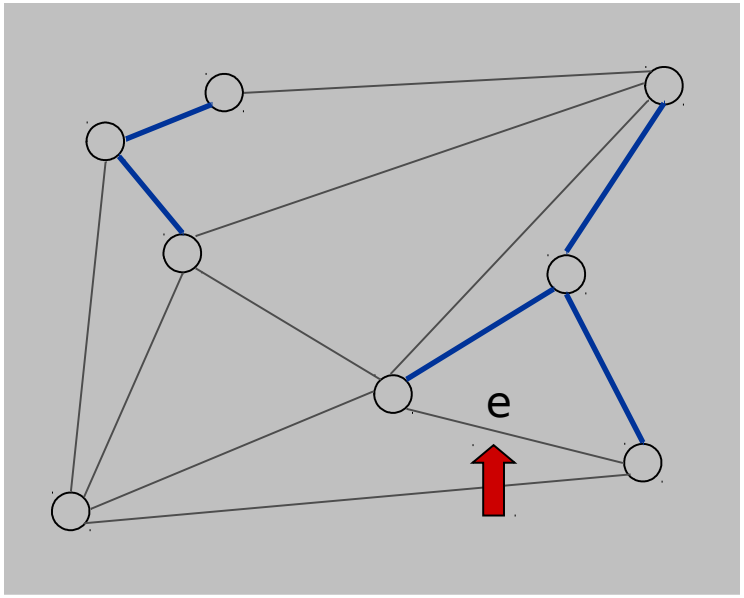
Si estrae e da Q

Algoritmo di Kruskal

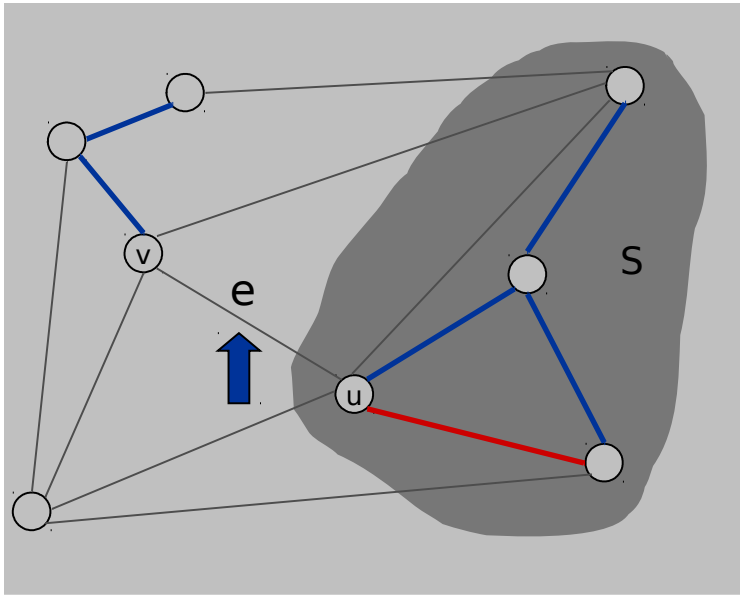
Algoritmo di Kruskal. [Kruskal, 1956]

- Considera ciascun arco in ordine non decrescente di peso
- Caso 1: Se e crea un cliclo allora scarta e
- Case 2: Altrimenti inserisce e in T

- NB: durante l'esecuzione T è una foresta composta da uno o più alberi



Caso 1



Caso 2

Esempio

Archi in MST : rossi (già selezionati) e verdi (non ancora selezionati)

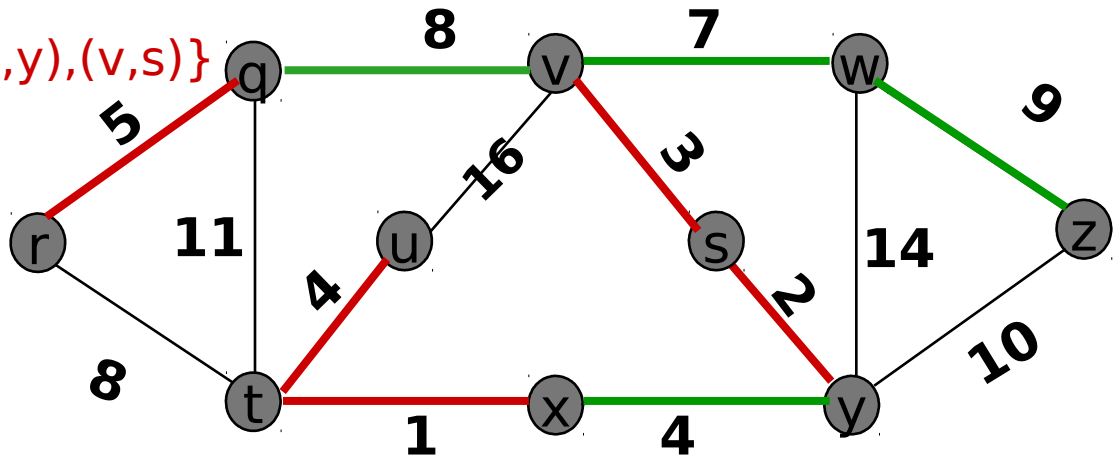
T={(r,q),(t,u),(t,x),(v,s),(s,y)} archi dell'MST già inseriti

Componenti connesse (alberi) in $G_T=(V,T)$:

$C_1=\{(r,q)\}$

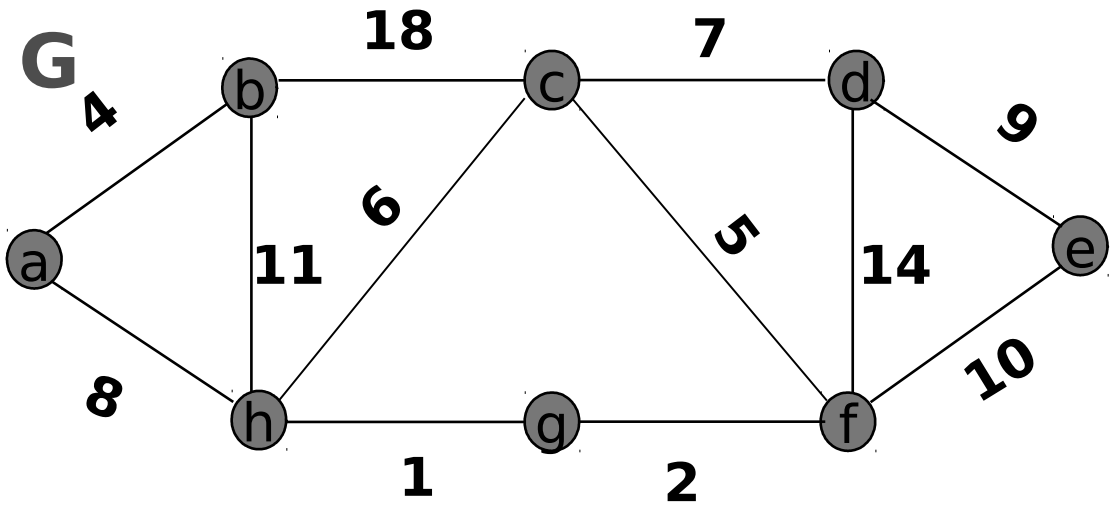
$C_2=\{(t,x),(t,u)\}$

$C_3=\{(s,y),(v,s)\}$



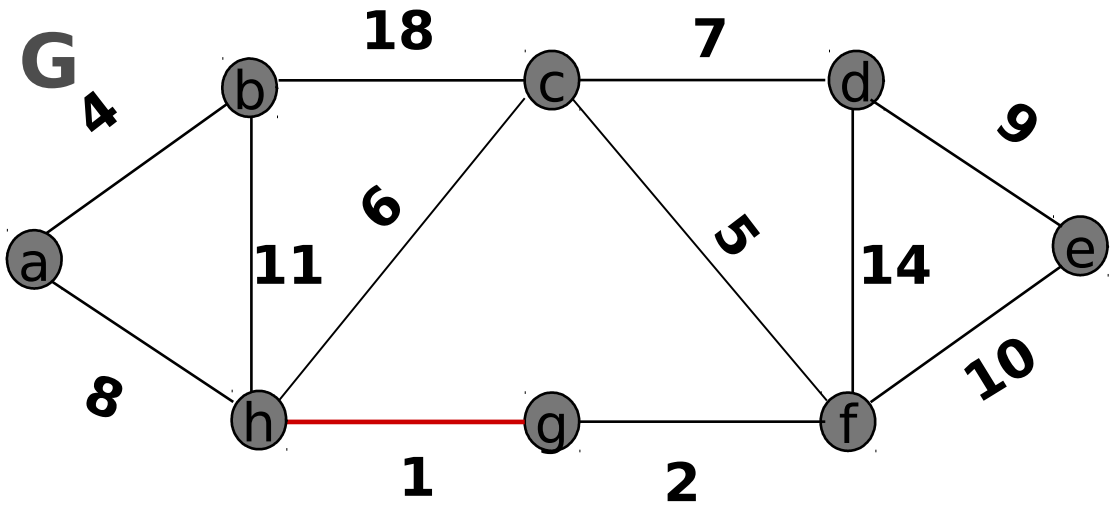
Il prossimo arco selezionato è (x,y)

Un esempio



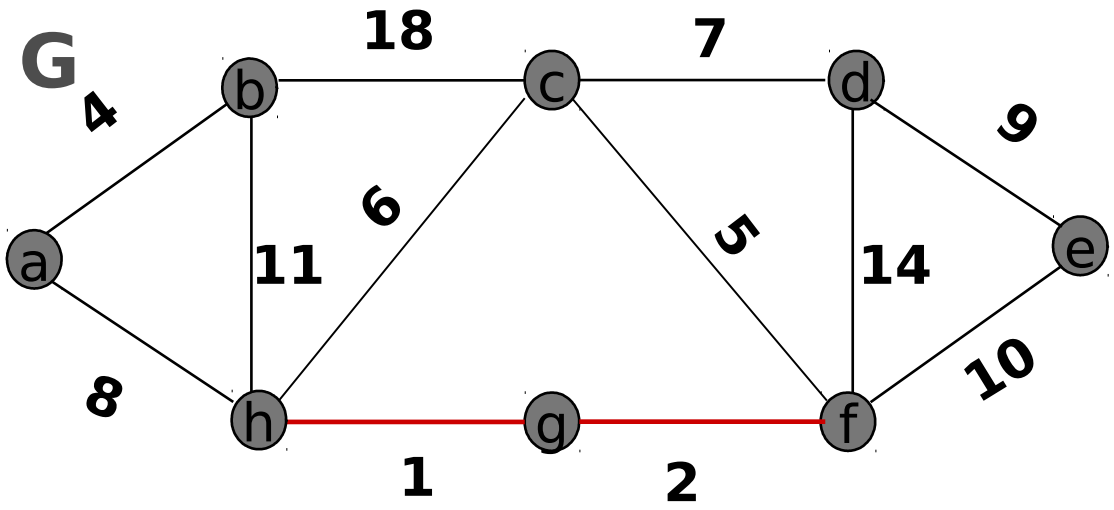
T = { }

Un esempio



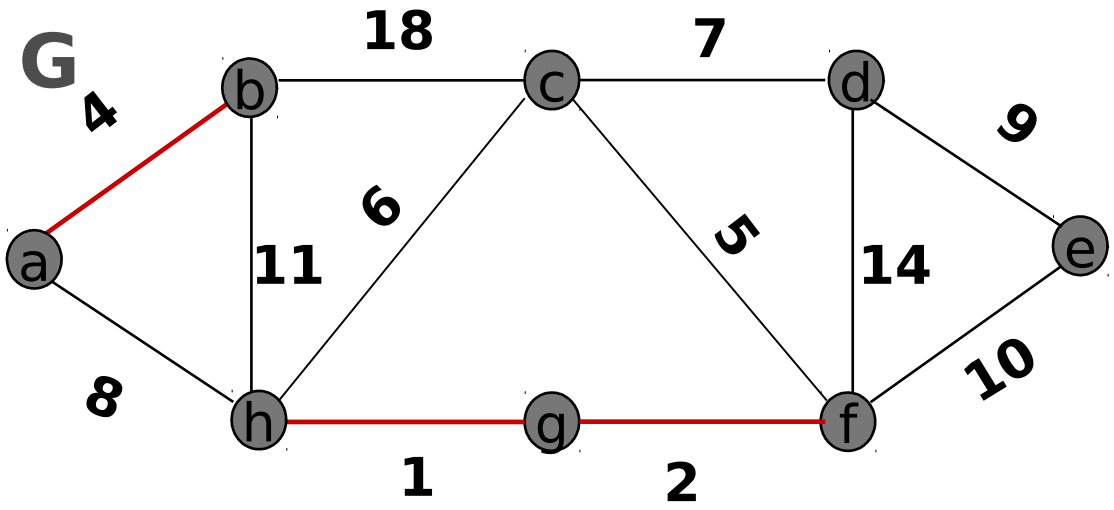
T = {(h,g) }

Un esempio



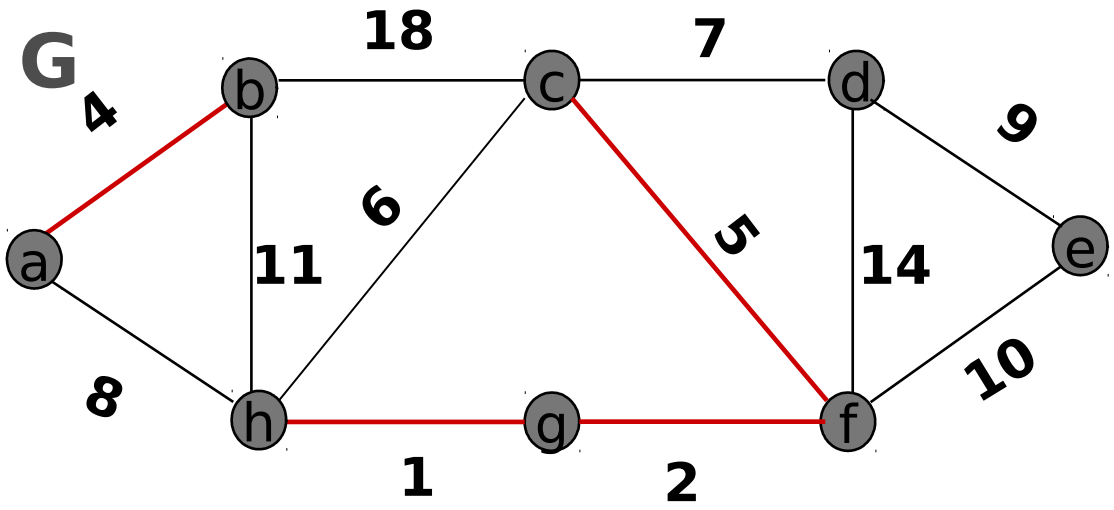
T = {(h,g), (g,f) }

Un esempio



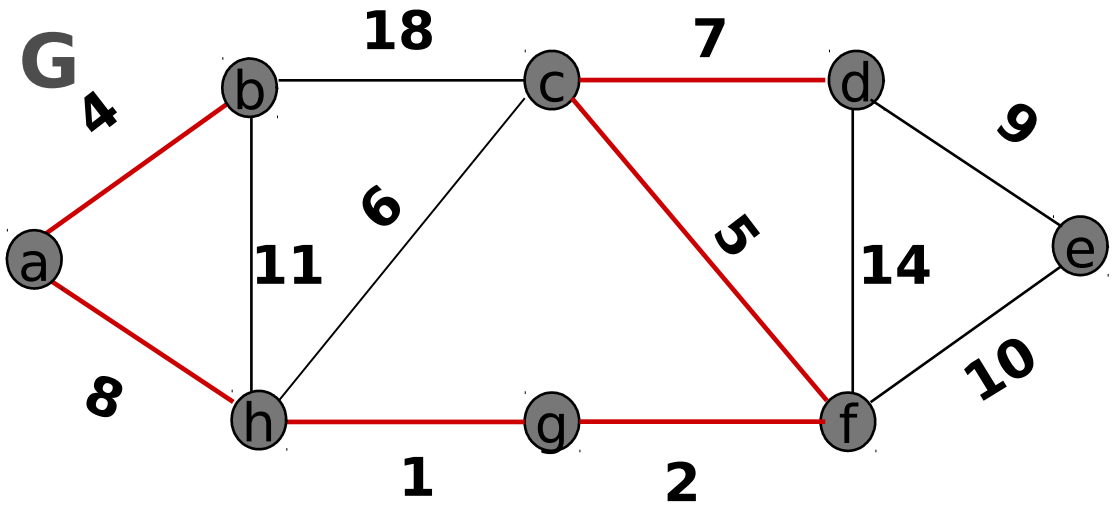
T = {(h,g), (g,f), (a,b) }

Un esempio



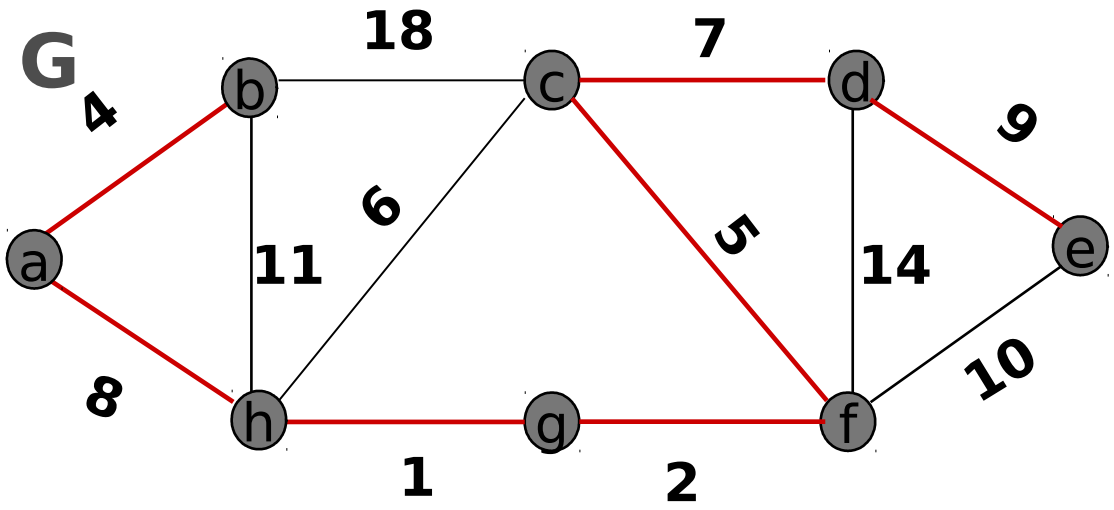
T = {(h,q), (g,f), (a,b),(c,f) }

Un esempio



T = {(h,g), (g,f), (a,b),(c,f),(c,d), (a,h) }

Un esempio



T = {(h,g), (g,f), (a,b),(c,f),(c,d), (a,h), (e,d) }

Correttezza dell'algoritmo di Kruskal

- L'insieme di archi T prodotto dall'algoritmo di Kruskal è un MST

  Dim. (per il caso in cui gli archi hanno costi a due a due distinti)

- Prima dimostriamo che ogni arco di T è anche un arco di MST
- Ad ogni passo l'algoritmo inserisce in T l'arco **e=(u,v)** di **peso minimo** tra quelli **non ancora esaminati** e **che non creano cicli in T**.
- Il fatto che l'arco **e** non crea cicli in T vuol dire che T fino a quel momento non contiene un percorso che collega **u** a **v**.
- Ricordiamo che gli archi di T formano uno o più alberi. Consideriamo l'albero contenente **u** e chiamiamo S l'insieme dei suoi vertici. Ovviamente **v** non è in S altrimenti esisterebbe un percorso da **u** a **v**.

- L'arco **e=(u,v)** è l'arco di peso minimo tra quelli che attraversano il taglio [S, V-S] (perché?) e quindi per la proprietà del taglio l'arco **e=(u,v)** è nel minimo albero ricoprente.

Continua nella prossima slide

Correttezza dell'algoritmo di Kruskal

Ora dimostriamo che T è un albero ricoprente.

T è un albero ricoprente perchè

- l'algoritmo non introduce mai cicli in T (ovvio!)
- connette tutti i vertici
  - Se così non fosse esisterebbe un insieme W non vuoto e di al più n-1 vertici tale che non c'è alcun arco di T che connette un vertice di W ad uno di V-W.
  - Siccome il grafo input G è connesso devono esistere uno o più archi in G che connettono vertici di W a vertici di V-W
  - Dal momento che l'algoritmo di Kruskal esamina tutti gli archi avrebbe selezionato sicuramente uno degli archi che connettono un vertice di W ad uno di V-W
  - Quindi non può esistere alcun insieme W non vuoto e di al più n-1 vertici tale che in T nessun vertice di W è connesso ad un vertice di V-W.

Implementazione dell'algoritmo di Kruskal

- Abbiamo bisogno di rappresentare le componenti connesse (alberi della foresta)
- Ciascuna componente connessa è un insieme di vertici disgiunto da ogni altro insieme.

```
Kruskal(G, c) {
   Sort edges weights so that c_1 ≤ c_2 ≤ ... ≤ c_m.
   T ← φ

   foreach (u ∈ V) make a set containing singleton u

   for i = 1 to m
      (u,v) = e_i
      if (u and v are in different sets) {
         T ← T ∪ {e_i}
         merge the sets containing u and v
      }
   return T
}
```

are u and v in different connected components?

merge two components

Implementazione dell'algoritmo di Kruskal

- Ciascun albero della foresta è rappresentato dal suo insieme di vertici
- Per rappresentare questi insiemi di vertici, si utilizza la struttura dati Union-Find per la rappresentazione di insiemi disgiunti
- Operazioni supportate dalla struttura dati Union-Find
- MakeUnionFind(S): crea una collezione di insiemi ognuno dei quali contiene un elemento di S
  - Nella fase di inizializzazione dell'algoritmo di Kruskal viene invocato MakeUnionFind(V): ciascun insieme creato corrisponde ad un albero con un solo vertice.
- Find (x): restituisce l'insieme che contiene x
  - Per ciascun arco esaminato (u,v), l'algoritmo di Kruskal invoca find(u) e find(v). Se entrambe le chiamate restituiscono lo stesso insieme allora vuol dire che u e v sono nello stesso albero e quindi (u,v) crea un ciclio in T.
- Union(X,Y): unisce l'insieme X a Y
  - Se l'arco (u,v) non crea un ciclo in T allora l'algoritmo di Kruskal invoca Union(Find(u),Find(v)) per unire le componenti connesse di u e v in un'unica componente connessa

Implementazione dell'algoritmo di Kruskal con Union-Find

```
Kruskal(G, c) {
   Sort edges weights so that c_1 ≤ c_2 ≤ ... ≤ c_m.
   T ← ϕ

   MakeUnionFind(V)   //create n singletons for the n vertices

   for i = 1 to m
      (u,v) = e_i
      if (Find(u)≠Find(v)) {
         T ← T ∪ {e_i}
         Union(Find(u),Find(v))
      }
   return T
}
```

Implementazione di Union-Find con array

- La struttura dati Union-Find può essere implementata in vari modi
- Implementazione di Union-Find con array
  - Gli elementi sono etichettati con interi consecutivi da 1 ad n e ad ogni elemento è associata una cella dell'array S che contiene il nome del suo insieme di appartenenza.
  - Find: O(1). Basta accedere alla cella di indice x dell'array S
  - Union: O(n). Occorre aggiornare le celle associate agli elementi dei due insiemi uniti.
  - MakeUnionFind: O(n). Occorre inizializzare tutte le celle.

Analisi dell'algoritmo di Kruskal in questo caso:

Inizializzazione: $O(n)+O(m \log m)=O(m \log n^2)=O(m \log n)$.

- O(n) per creare la struttura Union-Find e O(m log m) per ordinare gli archi

In totale, 2m find --> O(m)

- 2 find per ogni arco esaminato; O(1) per le 2 find.

In totale n-1 union (perché?) --> $O(n^2)$

- 1 union per ogni arco aggiunto a T; O(n) per la union

Algoritmo: **$O(m \log n)+ O(n^2)$**

Implementazione di Union-Find con array ed union-by-size

- Stessa implementazione della slide precedente ma si usa anche un altro array A per mantenere traccia della cardinalità di ciascun insieme. L'array ha n celle perché inizialmente ci sono n insiemi.
- Find(x) è identica a prima
- MakeUnionFind O(n): occorre inizializzare tutte le celle S e tutte le celle di A. Inizialmente le celle di A sono tutte uguali ad 1.
- Union: si guarda quali dei due insiemi è più piccolo e si aggiornano solo le celle di S corrispondenti agli elementi di questo insieme. In queste celle viene messo il nome dell'insieme più grande. La cella dell'array A corrispondente all'insieme più piccolo viene posta a 0 mentre quella corrispondente all'insieme più grande viene posta uguale alla somma delle cardinalità dei due insiemi uniti.
  - Corrisponde ad inserire gli elementi dell'insieme più piccolo in quello più grande.

Implementazione di Union-Find con array e union-by-size

- Nell'implementazione con union-by-size la singola operazione di unione richiede ancora O(n) nel caso pessimo perché i due insiemi potrebbero avere entrambi dimensione pari ad n diviso una certa costante.
- Vediamo però cosa accade quando consideriamo una sequenza di unioni.
- Inizialmente tutti gli insiemi hanno dimensione 1.

Implementazione di Union-Find con array e union-by-size

Affermazione. Una qualsiasi sequenza di unioni richiede al più tempo O(n log n).

Dim. Il tempo di esecuzione di una sequenza di Union dipende dal numero di aggiornamenti che vengono effettuati nell'array S.

- Calcoliamo quanto lavoro viene fatto per un qualsiasi elemento x. Questo lavoro dipende dal numero di volte in cui viene aggiornata la cella S[x] e cioè dal numero di volte in cui x viene spostato da un insieme ad un altro per effetto di una Union.
- Ogni volta che facciamo un'unione, x cambia insieme di appartenenza solo se proviene dall'insieme che ha dimensione minore o uguale dell'altro. Ciò vuol dire che l'insieme risultante dall'unione ha dimensione pari almeno al doppio dell'insieme da cui proviene x.
- Dopo un certo numero k di unioni che richiedono lo spostamento di x, l'elemento x si troverà in un insieme di taglia almeno 2^k → x viene spostato al più log(n) volte in quanto, al termine della sequenza di unioni, x si troverà in un insieme B di cardinalità al più n.
- Quindi per ogni elemento viene fatto un lavoro che richiede tempo O(log n). Siccome ci sono n elementi, in totale il tempo richiesto dalla sequenza di Union è O(n log n).

Implementazione di Union-Find con array e union-by-size

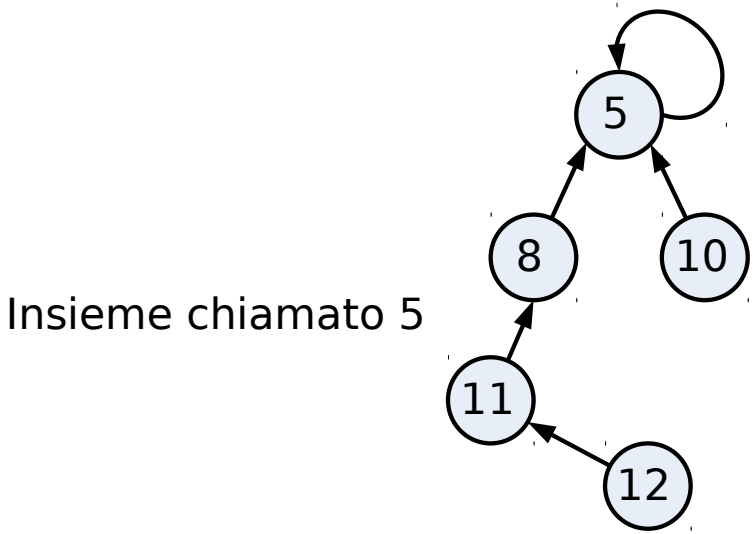
Analisi algoritmo di Kruskal in questo caso:

- Inizializzazione O(n)+O(m log m)=O(m log m) (come nella versione precedente).
- In totale le find richiedono O(m) (come nella versione precedente)
- Le n-1 union, per l'affermazione nella slide precedente, richiedono O(nlog n)

Algoritmo: $O(m \log m + n \log n) = O(m \log m) = O(m \log n^2)$
$= O(m \log n)$

Implementazione basata su struttura a puntatori

- Insiemi rappresentati da strutture a puntatori
- Ogni nodo contiene un campo per l'elemento ed un campo con un puntatore ad un altro nodo dello stesso insieme.
- In ogni insieme vi è un nodo il cui campo puntatore punta a sé stesso. L'elemento in quel nodo dà nome all'insieme
- Inizialmente ogni insieme è costituito da un unico nodo il cui campo puntatore punta al nodo stesso.

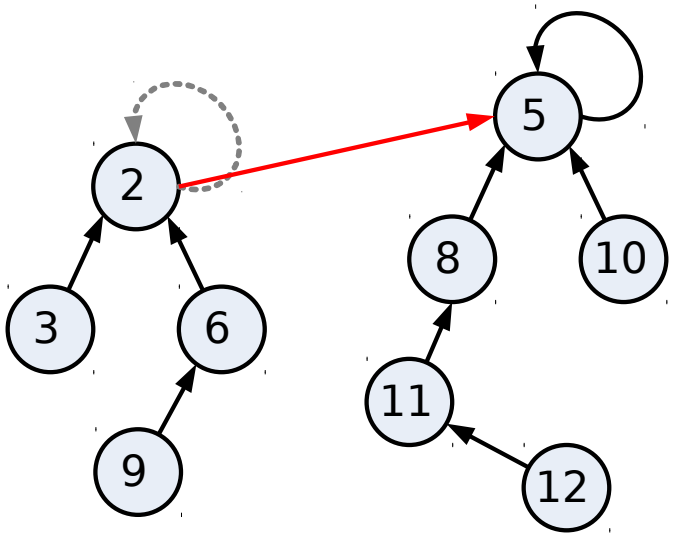


Union

- Per eseguire la union di due insiemi A e B, dove A è indicato con il nome dell' elemento x mentre B con il nome dell'elemento y, si pone nel campo puntatore del nodo contenente x un puntatore al nodo contenente y. In questo modo y diventa il nome dell'insieme unione. Si può fare anche viceversa, cioè porre nel campo puntatore del nodo contenente y un puntatore al nodo contenente x. In questo caso, il nome dell'insieme unione è x.

- Tempo: O(1)

Unione dell'insieme di nome 2 con quello di nome 5. L'insieme unione viene indicato con 5.

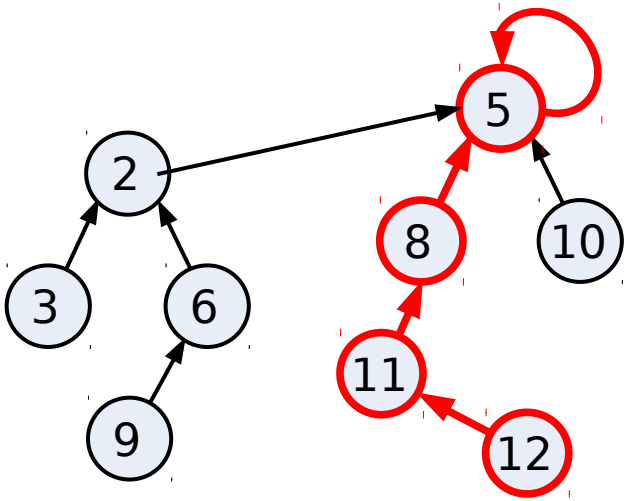


Find

- Per eseguire una find, si segue il percorso che va dal nodo che contiene l'elemento passato in input alla find fino al nodo che contiene l'elemento che dà nome all'insieme (nodo il cui campo puntatore punta a se stesso)

- Tempo: O(n) dove n è il numero di elementi nella partizione.
- Il tempo dipende dal numero di puntatori attraversati per arrivare al nodo contenente l'elemento che dà nome all'insieme.
- Il caso pessimo si ha quando la partizione è costituita da un unico insieme ed i nodi di questo insieme sono disposti uno sopra all'altro e ciascun nodo ha il campo puntatore che punta al nodo immediatamente sopra di esso

Find(12)

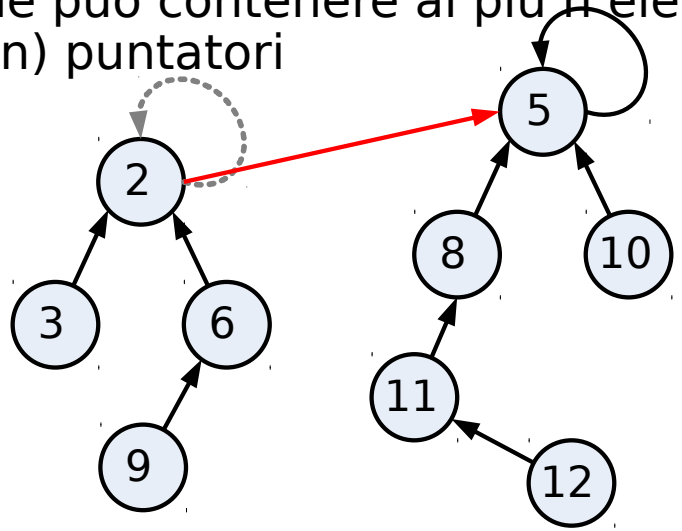


Euristica per migliorare l'efficienza

- **Union-by-size:** Diamo all'insieme unione il nome dell'insieme più grande. In questo modo find richiede tempo O(log n)

Dim.

- Contiamo il numero massimo di puntatori che possono essere attraversati durante l'esecuzione di un'operazione di find
- Osserviamo che un puntatore viene creato solo se viene effettuata una unione. Quindi attraversando il puntatore da un nodo contenente x ad uno contenente y passiamo da quello che prima era l'insieme x all'insieme unione degli insiemi x e y. Poiché usiamo la union-by-size, abbiamo che l'unione di questi due insiemi ha dimensione pari almeno al doppio della dimensione dell'insieme x
- Di conseguenza, ogni volta che attraversiamo un puntatore da un nodo ad un altro, passiamo in un insieme che contiene almeno il doppio degli elementi contenuti nell'insieme dal quale proveniamo.
- Dal momento che un insieme può contenere al più n elementi, in totale si attraversano al più O(log n) puntatori



Euristica per migliorare l'efficienza

Union-by-size

- Consideriamo la struttura dati Union-Find creata invocando MakeUnionFind su un insieme S di dimensione n. Se si usa l'implementazione della struttura dati Union-Find basata sulla struttura a puntatori che fa uso dell'euristica union-by-size allora si ha
- Tempo Union : O(1) (manteniamo per ogni nodo un ulteriore campo che tiene traccia della dimensione dell'insieme corrispondente)
- Tempo MakeUnionFind: O(n) occorre creare un nodo per ogni elemento.
- Tempo Find: O(log n) per quanto visto nella slide precedente

Kruskal con questa implementazione di Union-Find richiede
$O(m \log m) = O(m \log n^2) = O(m \log n)$ per l'ordinamento
O(n) per l'inizializzazione degli insiemi della partizione
O(m log n) per le O(m) find
O(n) per le n-1 Union.

In totale O(m log n)
come nel caso in cui si usa l'implementazione di Union-Find basata sull'array con uso dell'euristica union-by-size

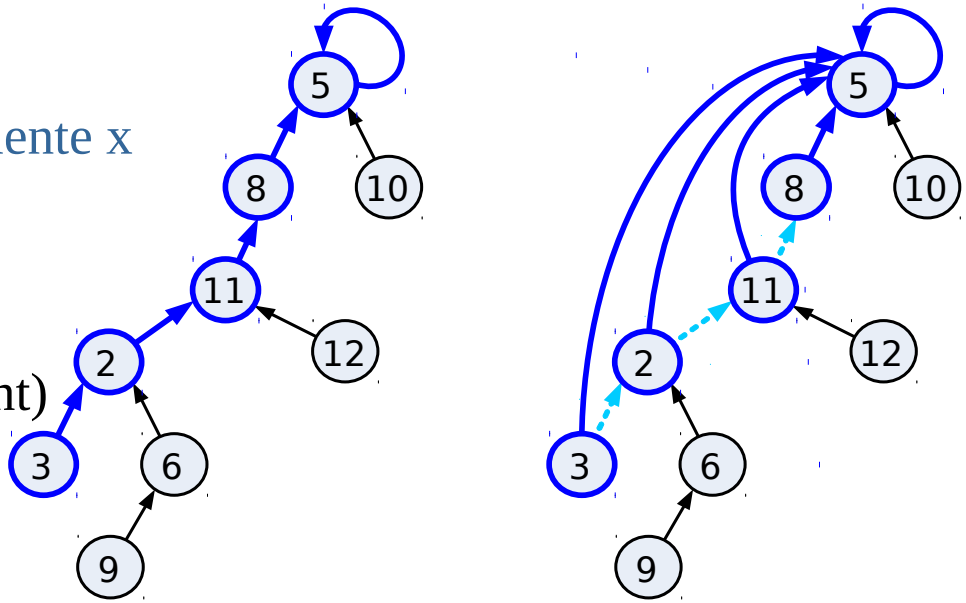
Un’altra euristica per l’efficienza

Path Compression (non ci serve per migliorare il costo dell’alg. Di Kruskal)

- Dopo aver eseguito un’operazione di Find, tutti i nodi attraversati nella ricerca avranno come il campo puntatore che punta al nodo contenente l’elemento che dà nome all’insieme
- Intuizione: ogni volta che eseguiamo la Find con in input un elemento x di un certo insieme facciamo del lavoro in più che ci fa risparmiare sulle successive operazioni di Find effettuate su elementi incontrati durante l’esecuzione di Find(x). Questo lavoro in più non fa comunque aumentare il tempo di esecuzione asintotico della singola Find.

Indichiamo con q(x) il nodo contenente x

```
Find(x)
 if q(x) ≠q(x).pointer
  then  p← q(x).pointer
        q(x).pointer ← Find(p.element)
return q(x).pointer
```



Union-by-size e path-compression

- Se si utilizza l'euristica path-compression allora una sequenza di n operazioni di find richiede tempo $O(n\,\alpha(n))$
- $\alpha(n)$ è l'inversa della funzione di Ackermann
- $\alpha(n))\leq 4$ per tutti i valori pratici di n

Proprietà del ciclo

- **Per semplicità assumiamo che tutti i costi c_e siano distinti.**

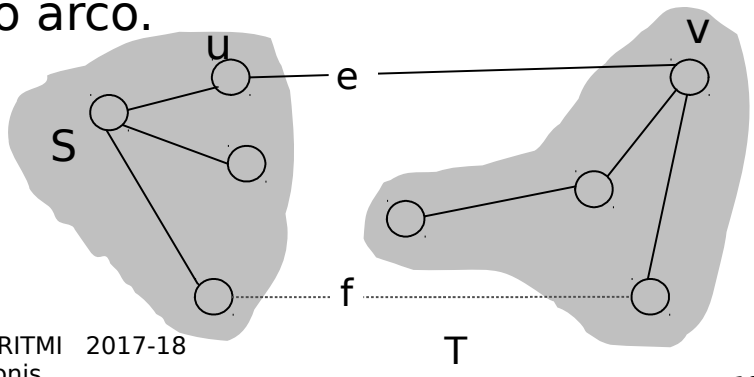
- Proprietà del ciclo. Sia C un ciclo e sia e=(u,v) l'arco di costo massimo tra quelli appartenenti a C. Ogni minimo albero ricoprente non contiene l'arco e.

  Dim. (tecnica dello scambio)

  - Sia T un albero ricoprente che contiene l'arco e. Dimostriamo che T non può essere un MST.
  - Se rimuoviamo l'arco e da T disconnettiamo T e otteniamo due alberi disgiunti, uno contenete u ed uno contenente v. Chiamiamo S l'insieme dei nodi dell'albero contenente u. Ovviamente v sarà in V-S.
  - Il ciclo C contiene due percorsi per andare da u a v. Uno è costituito dall'arco e=(u,v) mentre l'altro percorso va da u a v attraversando gli archi diversi da (u,v). Tra questi archi deve essercene uno che attraversa il taglio [S,V-S] altrimenti non sarebbe possibile andare da u che sta in S a v che sta in V-S. Sia f questo arco.

    Se al posto dell'arco e inseriamo in T l'arco f, otteniamo un albero ricoprente T' di costo $c(T')=c(T)- c_e+c_f$

    Siccome $c_f < c_e$, $c(T') < c(T)$.



Correttezza dell'algoritmo Inverti-Cancella

- L'algoritmo Inverti-Cancella produce un MST.

Dim. (nel caso in cui i costi sono a due a due distinti)

- Sia T il grafo prodotto da Inverti-Cancella.
- Prima dimostriamo che gli archi che non sono in T non sono neanche nello MST.
  - Sia e un qualsiasi arco che non appartiene a T.
  - Se e=(u,v) non appartiene a T vuol dire che quando l'arco e=(u,v) è stato esaminato l'arco si trovava su un ciclo C (altrimenti la sua rimozione avrebbe disconnesso u e v).
  - Dal momento che gli archi vengono esaminati in ordine decrescente di costo, l'arco e=(u,v) ha costo massimo tra gli archi sul ciclo C.
  - La proprietà del ciclo implica allora che e=(u,v) non fa parte dello MST.
- Abbiamo dimostrato che ogni arco dello MST appartiene anche a T. Ora dimostriamo che T non contiene altri archi oltre a quelli dello MST.
  - Sia T^* lo MST. Ovviamente (V,T^*) è un grafo connesso.
  - Supponiamo **per assurdo** che esista un arco (u,v) di T che non è in T^*.
  - Se agli archi di T^* aggiungiamo l'arco (u,v), si viene a creare un ciclo. Poiché T contiene tutti gli archi di T^* e contiene anche (u,v) allora T contiene un ciclo C. Ciò è impossibile perché l'algoritmo rimuoverebbe C rimuovendo l'arco di costo più alto su C. Abbiamo quindi ottenuto una contraddizione.

Correttezza degli algoritmi quando i costi non sono distinti

- In questo caso la correttezza si dimostra perturbando di poco i costi c_e degli archi, cioè aumentando i costi degli archi in modo che valgano le seguenti tre condizioni
  - i nuovi costi $\hat{c}_e$ risultino a due a due distinti
  - se $c_e < c_{e'}$ allora $\hat{c}_e < \hat{c}_{e'}$
  - la somma dei valori aggiunti ai costi degli archi sia minore del minimo delle quantità $|c(T_1)-c(T_2)|$, dove il min è calcolato su tutte le coppie di alberi ricoprenti T_1 e T_2 tali che $\mathbf{c(T_1) \neq c(T_2)}$ (Questo non è un algoritmo per cui non ci importa quanto tempo ci vuole a calcolare il minimo)

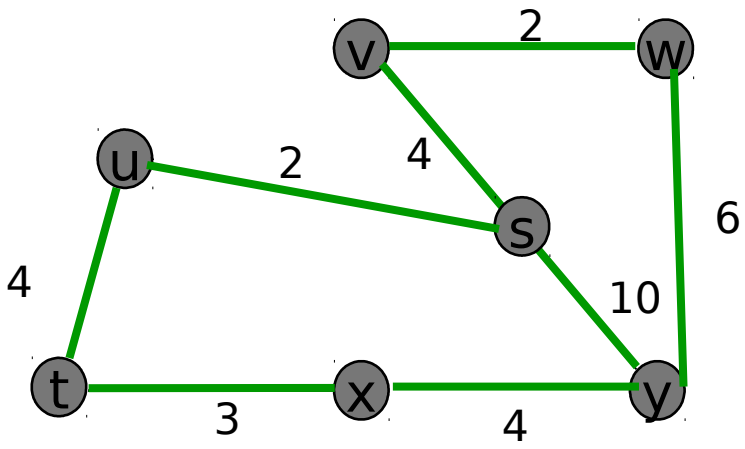
Continua nella prossima slide

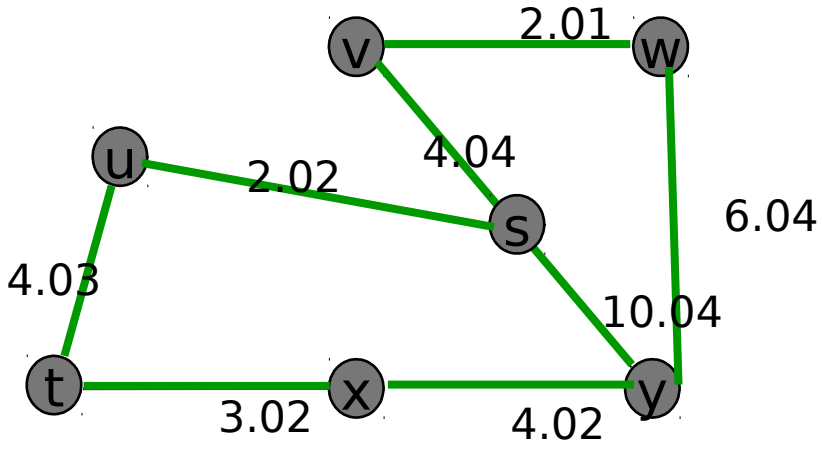
Correttezza degli algoritmi quando i costi non sono distinti

- Chiamiamo G il grafo di partenza (con i costi non pertubati) e $\hat{G}$ quello con i costi perturbati.
- Sia T un minimo albero ricoprente del grafo $\hat{G}$. Dimostriamo che T è un minimo albero ricoprente anche per G.
- Se così non fosse esisterebbe un albero T^* che in G ha costo minore di T.
- Siano c(T) e c(T^*) i costi di T e T^* in G. Per come abbiamo perturbato i costi, si ha che c(T)-c(T^*) > somma dei valori aggiunti ai costi degli archi di G
- Vediamo di quanto può essere cambiato il costo di T^* dopo aver perturbato gli archi.
- Il costo di T^* non può essere aumentato più della somma totale dei valori aggiunti ai costi degli archi dell'intero grafo G
- Poiché la somma dei valori aggiunti ai costi degli archi è minore di c(T)-c(T^*) allora il costo di T^* è aumentato di un valore minore di c(T)-c(T^*). Di conseguenza, dopo aver perturbato i costi, la differenza tra il costo di T e quello di T^* è diminuita di un valore inferiore a c(T)-c(T^*) per cui è ancora maggiore di 0. Ne consegue che T non può essere lo MST di $\hat{G}$ perché T^* ha costo più piccolo di T anche in $\hat{G}$.

Correttezza degli algoritmi quando i costi non sono distinti

- In questo esempio i costi sono interi quindi è chiaro che i costi di due alberi ricoprenti di costo diverso differiscono almeno di 1.
- Se perturbiamo i costi come nella seconda figura, si ha che
  - I nuovi costi sono a due a due distinti
  - Se e ha costo minore di e' all'inizio allora e ha costo minore di e' anche dopo aver modificato i costi.
  - La somma dei valori aggiunti ai costi è 0.01+0.02+0.02+0.02+0.03+00.4+0.04+0.04 < 1





Correttezza degli algoritmi quando i costi non sono distinti

- Proprietà del taglio (senza alcun vincolo sui costi degli archi) Sia S un qualsiasi sottoinsieme di nodi e sia e **un** arco di costo minimo che attraversa il taglio [S,V-S]. Esiste un minimo albero ricoprente che contiene e.

  Dim.

- Siano $e_1,e_2,\ldots,e_p$ gli archi di G che attraversano il taglio ordinati in modo che $c(e_1) \leq c(e_2) \leq \ldots \leq c(e_p)$ con $e_1=e$.
- Perturbiamo i costi degli archi di G come mostrato nelle slide precedenti e facendo in modo che $\hat{c}(e_1) < \hat{c}(e_2) < \ldots < \hat{c}(e_p)$. Per fare questo basta perturbare i costi c di G nel modo già descritto e stando attenti che se $c(e_i)=c(e_{i+1})$, per un certo $1 \leq i \leq p-1$, allora deve essere $\hat{c}(e_i) < \hat{c}(e_{i+1})$.
- Consideriamo lo MST T di $\hat{G}$.
- La proprietà del taglio per grafi con costi degli archi a due a due distinti implica che lo MST di $\hat{G}$ contiene l'arco e (in quanto e è l'arco di peso minimo che attraversa [S,V-S] in $\hat{G}$) → T contiene e.
- Per quanto dimostrato nelle slide precedenti, T è anche un MST di G.
- Abbiamo quindi dimostrato che esiste un MST di G che contiene e.
- NB: MST distinti di G possono essere ottenuti permutando tra di loro archi di costo uguale nell'ordinamento $c(e_1) \leq c(e_{2)} \leq \ldots \leq c(e_p)$

Correttezza degli algoritmi quando i costi non sono distinti

- Proprietà del ciclo (senza alcun vincolo sui costi degli archi) Sia C un ciclo e sia e **un** arco di costo massimo in C. Esiste un minimo albero ricoprente che non contiene e.
- Dim.
- Siano $e_1,e_2,\ldots,e_p$ gli archi del ciclo C, ordinati in modo che $c(e_1) \leq c(e_{2)} \leq \ldots \leq c(e_p)$ con $e_p=e$.
- Perturbiamo i costi degli archi di G come mostrato nelle slide precedenti e facendo in modo che $\hat{c}(e_1) < \hat{c}(e_2) < \ldots < \hat{c}(e_p)$. Per fare questo basta perturbare i costi c di G nel modo già descritto e stando attenti che se $c(e_i)=c(e_{i+1})$, per un certo $1 \leq i \leq p-1$, allora deve essere $\hat{c}(e_i) < \hat{c}(e_{i+1})$.
- Consideriamo lo MST T di $\hat{G}$.
- La proprietà del ciclo per grafi con costi degli archi a due a due distinti implica che lo MST di $\hat{G}$ non contiene l'arco e (in quanto e è l'arco di peso massimo nel ciclo C in $\hat{G}$)→ T NON deve contenere e.
- Per quanto dimostrato nelle slide precedenti T è anche un MST di G.
- Abbiamo quindi dimostrato che esiste un MST di G che non contiene e.
- NB: MST distinti di G possono essere ottenuti permutando tra di loro archi di costo uguale nell'ordinamento $c(e_1) \leq c(e_{2)} \leq \ldots \leq c(e_p)$.

Correttezza degli algoritmi quando i costi non sono distinti

- Si è visto che la proprietà del taglio può essere estesa al caso in cui i costi degli archi **non** sono a due a due distinti
- Possiamo quindi dimostrare la correttezza degli algoritmi di Kruskal e di Prim nello stesso modo in cui abbiamo dimostrato la correttezza di questi algoritmi nel caso in cui gli archi hanno costi a due a due distinti.

- Si è visto che la proprietà del ciclo può essere estesa al caso in cui i costi degli archi **non** sono a due a due distinti
- Possiamo quindi dimostrare la correttezza dell'algoritmo Inverti-Cancella nello stesso modo in cui abbiamo dimostrato la correttezza dell'algoritmo nel caso in cui gli archi hanno costi a due a due distinti.