Strutture Dati, Algoritmi e Complessità

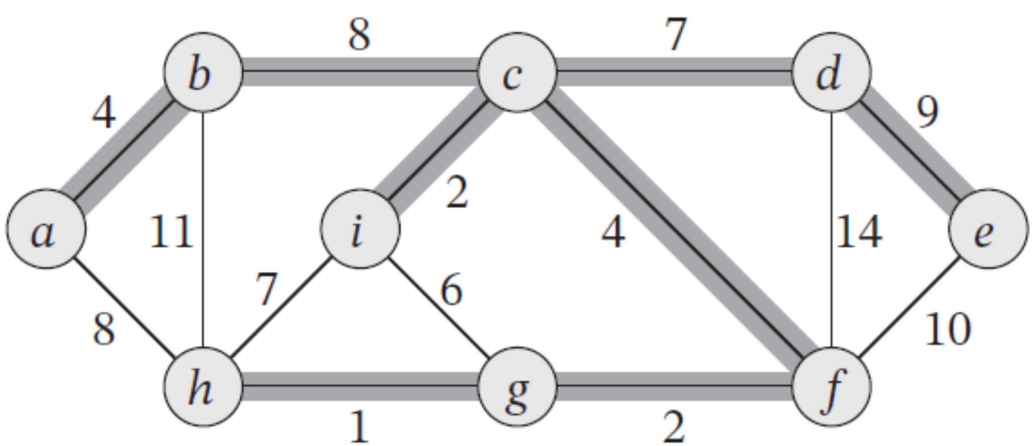
ALBERI DI CONNESSIONE MINIMI

AA 2023-2024

Il problema degli alberi di connessione minimi

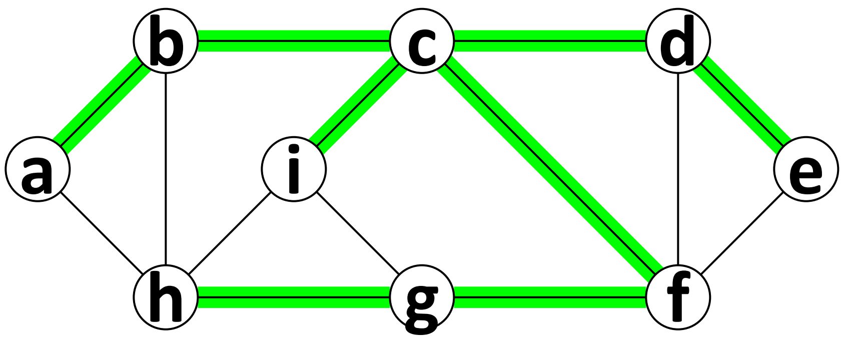
Consideriamo un grafo non orientato ***G* = (*V*,*E*)** connesso e i cui archi hanno associato un peso ω(e). Il problema di definire un *albero di connessione minimo* è quello di trovare un sottoinsieme T di archi tali che

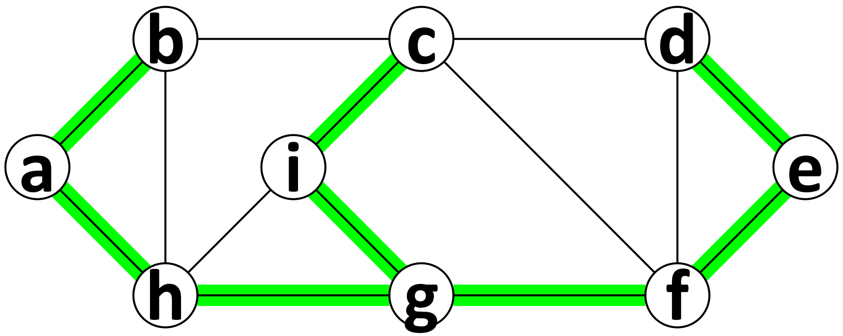
1. T rappresenta un albero
2. il peso $\omega(T) = \sum_{e \in T} \omega(e)$ sia minimo



Osservazione

Un grafo connesso ***G* = (*V*,*E*)** non pesato può avere molti alberi di connessione ma tutti hanno esattamente **|*V* | - 1** archi.





Un Algoritmo Avido per MST

Gli algoritmi avidi (greedy) sono una particolare classe di algoritmi che ci aiutano a risolvere problemi di ottimizzazione adottando, ad ogni passo, la strategia migliore (ossia ragionando su un ottimo locale)

Spesso, con un algoritmo greedy, riusciamo a trovare un ottimo globale ragionando in modo opportuno con gli ottimi locali.

Un Algoritmo Avido per MST

Generic-MST(G, w) // G grafo con funzione peso w

    $A = \emptyset$

    while "*A* non forma un albero di connessione"

        "cerca un *arco sicuro a*"

        "aggiungi *a* all'insieme *A* "

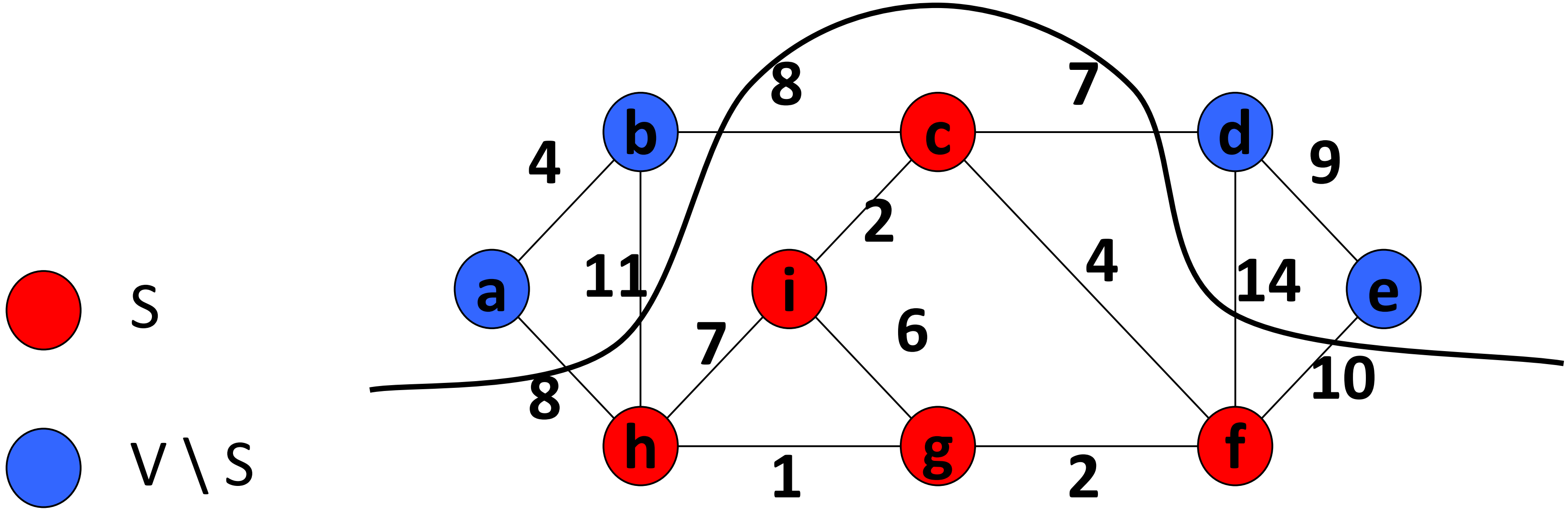
    return A

Cos'è un arco sicuro?

un arco che può far parte della soluzione

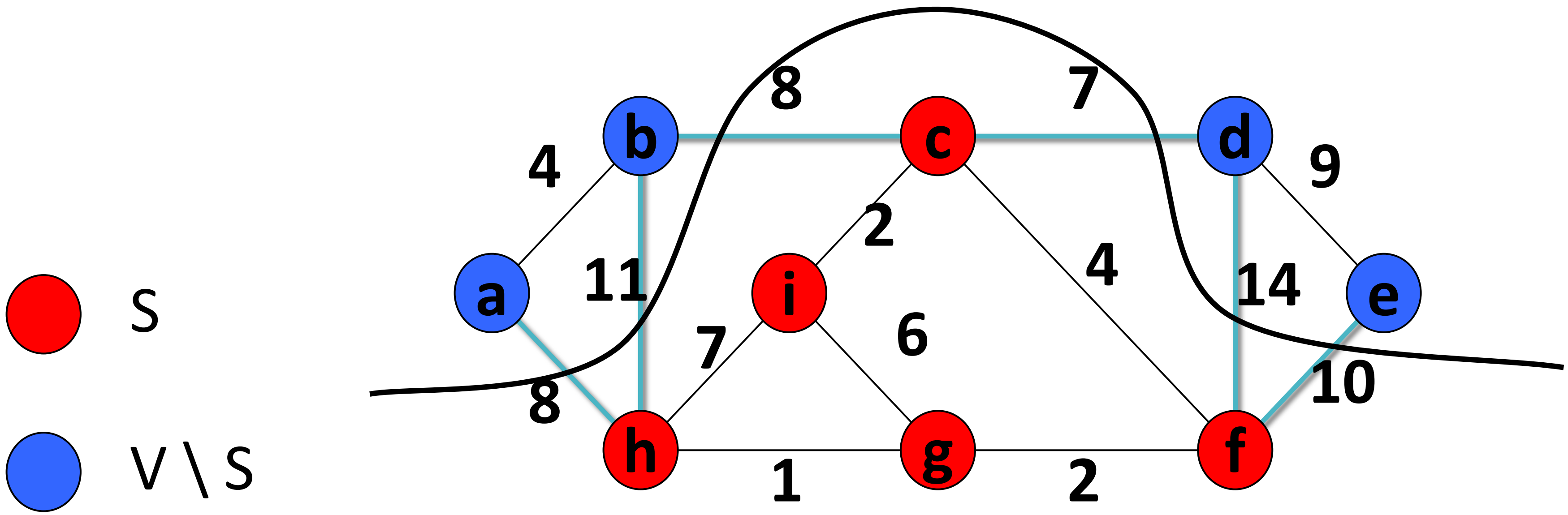
Alcune definizioni preliminari

Un *taglio* in un grafo non orientato ***G* = (*V,E*)** è una partizione dell'insieme di vertici ***V*** in due sottoinsiemi non vuoti **(*S, V \ S*)**.



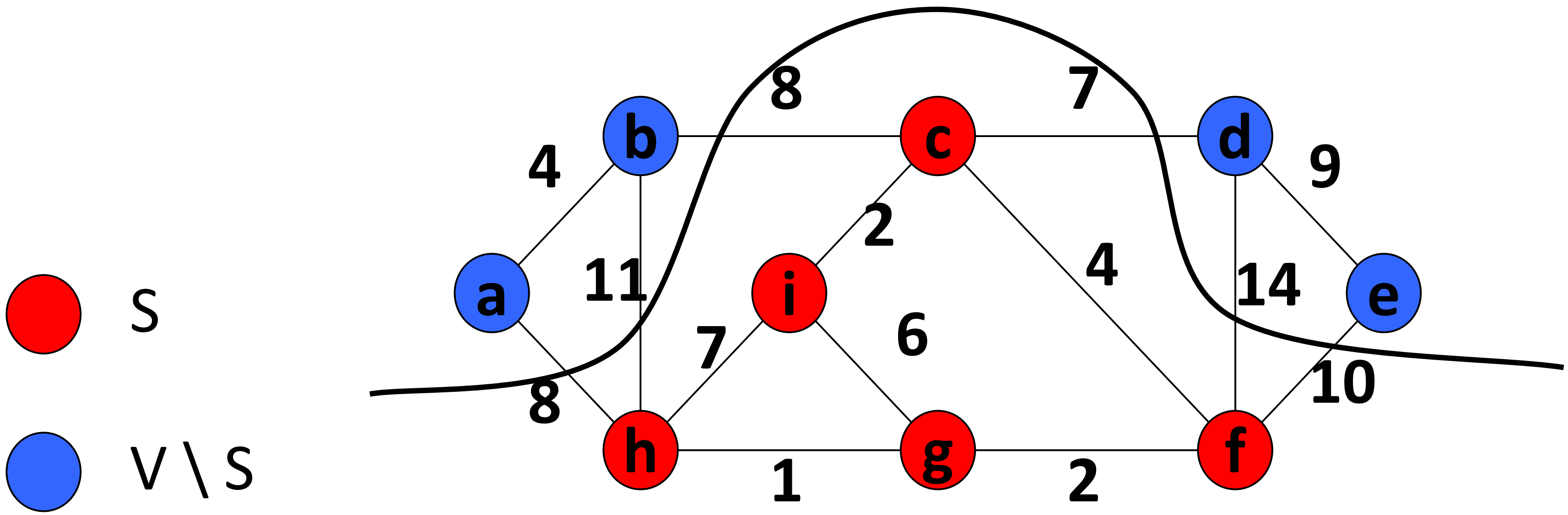
Alcune definizioni preliminari

Diciamo che un arco <u>*interseca*</u> il taglio **(*S*,*V* \ *S*)** se uno dei suoi estremi è in ***S*** e l'altro è in ***V* \ *S***



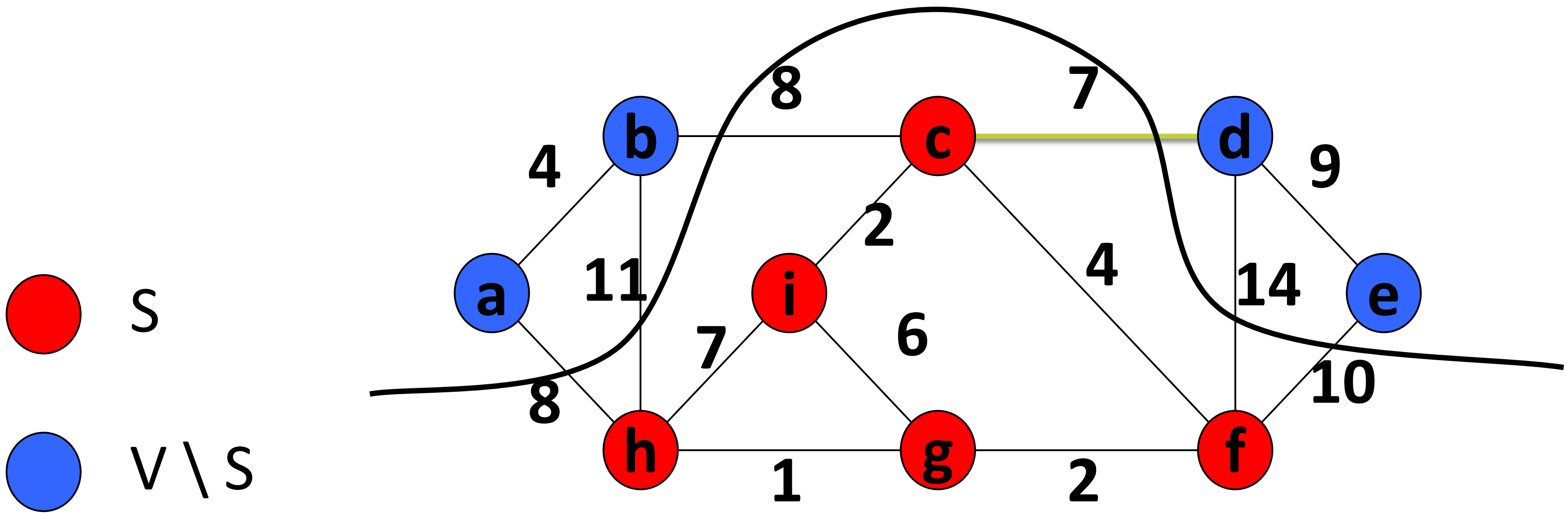
Alcune definizioni preliminari

Diciamo che il taglio **(*S*,*V* \ *S*)** *rispetta* un insieme di archi ***A*** se nessun arco in ***A*** interseca il taglio



Alcune definizioni preliminari

Un *arco leggero* per un taglio è un arco di costo minimo tra tutti quelli che intersecano il taglio



TEOREMA: ***Caratterizzazione degli archi sicuri***

Sia G= (V, E) un grafo connesso non orientato con una funzione **ω** a valori reali definita in E. Sia ***A*** un sottoinsieme di E, che è contenuto in qualche albero di connessione minimo per G, sia **(*S*,*V* \ *S*)** un taglio qualsiasi di G che rispetta ***A*** e sia ***a* = *uv*** un arco leggero per il taglio **(*S*,*V* \ *S*)**. Allora l'arco ***a*** è sicuro.

DIMOSTRAZIONE

Sia ***T*** un albero di connessione minimo che contiene ***A***.

CASO 1: ***T*** contiene anche l'arco ***a*** siamo a posto e il teorema segue banalmente

TEOREMA: *Caratterizzazione degli archi sicuri*

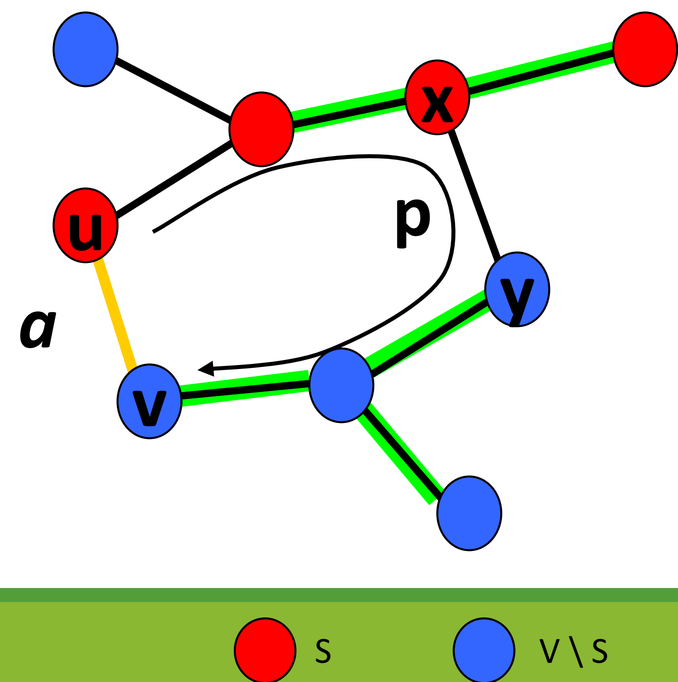
Sia G= (V, E) un grafo connesso non orientato con una funzione ω a valori reali definita in E. Sia ***A*** un sottoinsieme di E, che è contenuto in qualche albero di connessione minimo per G, sia **(*S,V \ S*)** un taglio qualsiasi di G che rispetta ***A*** e sia ***a* = *uv*** un arco leggero per il taglio **(*S,V \ S*)**. Allora l'arco ***a*** è sicuro.

DIMOSTRAZIONE

Sia ***T*** (rappresentato dagli archi neri solidi) un albero di connessione minimo che contiene ***A*** (archi verdi).

CASO 2: ***T*** non contiene anche l'arco ***a.***

Se ***T*** non contiene l'arco ***a* = *uv*** aggiungendo tale arco a ***T*** si forma un ciclo.

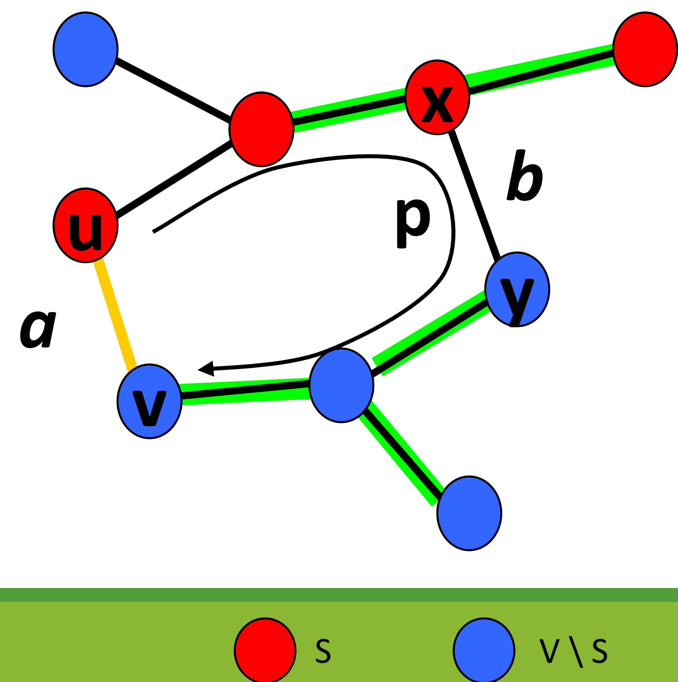


TEOREMA: ***Caratterizzazione degli archi sicuri***

Sia G= (V, E) un grafo connesso non orientato con una funzione **ω** a valori reali definita in E. Sia ***A*** un sottoinsieme di E, che è contenuto in qualche albero di connessione minimo per G, sia **(*S*,*V* \ *S*)** un taglio qualsiasi di G che rispetta ***A*** e sia ***a* = *uv*** un arco leggero per il taglio **(*S*,*V* \ *S*)**. Allora l'arco ***a*** è sicuro.

DIMOSTRAZIONE

Poiché ***u*** e ***v*** stanno da parti opposte rispetto al taglio, deve esserci almeno un arco ***b* = *xy*** nel cammino da ***u*** a ***v*** in ***T*** che interseca il taglio.

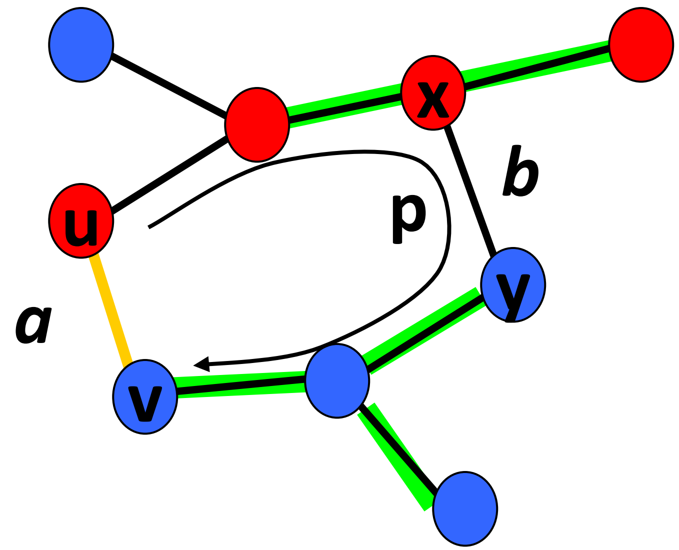


TEOREMA: ***Caratterizzazione degli archi sicuri***

Sia G= (V, E) un grafo connesso non orientato con una funzione **ω** a valori reali definita in E. Sia ***A*** un sottoinsieme di E, che è contenuto in qualche albero di connessione minimo per G, sia **(*S,V \ S*)** un taglio qualsiasi di G che rispetta ***A*** e sia ***a* = *uv*** un arco leggero per il taglio **(*S,V \ S*)**. Allora l'arco ***a*** è sicuro.

DIMOSTRAZIONE

Poiché ***u*** e ***v*** stanno da parti opposte rispetto al taglio, deve esserci almeno un arco ***b* = *xy*** nel cammino da ***u*** a ***v*** in ***T*** che interseca il taglio.



TEOREMA: ***Caratterizzazione degli archi sicuri***

Sia G= (V, E) un grafo connesso non orientato con una funzione **ω** a valori reali definita in E. Sia ***A*** un sottoinsieme di E, che è contenuto in qualche albero di connessione minimo per G, sia **(*S,V* \ *S*)** un taglio qualsiasi di G che rispetta ***A*** e sia ***a* = *uv*** un arco leggero per il taglio **(*S,V* \ *S*)**. Allora l'arco ***a*** è sicuro.

DIMOSTRAZIONE

- Sia ***T'*** l'albero di connessione ottenuto da ***T*** togliendo l'arco ***b*** e aggiungendo l'arco ***a***.
- Poiché sia ***a*** che ***b*** intersecano il taglio ed ***a*** è leggero, il costo di ***T'*** è minore o uguale del costo di ***T***.
- Ma ***T*** è un albero di connessione minimo e quindi anche ***T'*** lo è.
- Poiché il taglio rispetta l'insieme ***A***, l'arco tolto non stava in ***A*** e quindi ***T'*** contiene sia l'arco ***a*** che gli archi in ***A***.
- Pertanto ***a*** è un arco sicuro.

Algoritmi Greedy per MST

KRUSKAL

IDEA

- crea un’unica lista di tutti gli archi in *G*
- ordina la lista degli archi in ordine monotono crescente rispetto al peso
- applica la selezione scegliendo il miglior arco tra quelli dei vicini

Usa una rappresentazione di insiemi disgiunti per liste

PRIM

IDEA

- Costruisce l’albero di connessione minimo partendo da un vertice prescelto come radice ed estendendolo finché non connette tutti i vertici.

Usa una coda di priorità ***Q*** in cui memorizza i vertici non ancora raggiunti dall’albero in costruzione.

Algoritmo di Kruskal

MST-KRUSKAL($G, \boldsymbol{\omega}$) // G grafo con funzione peso $\boldsymbol{\omega}$

$A = \emptyset$

for "ogni $v \in G.V$"

  Make-Set(v)

crea una lista di tutti gli archi in $G.E$

ordina la lista degli archi in ordine monotono crescente rispetto al peso

for "ogni arco $a = uv \in E[G]$ in ordine di costo»

  if *Find-Set*(u) $\neq$ *FindSet*(v)

    $A = A \cup \{uv\}$

    Union(u,v)

return A

Rappresentazione di insiemi disgiunti

Abbiamo bisogno di una struttura dati che ci aiuta a mantenere una collezione

$$\boldsymbol{C = \{S_1, S_2, \ldots , S_k\}}$$

di ***insiemi disgiunti*** (i.e., for each i, j in [1, k], $S_i \cap S_j = \emptyset$).

Ogni insieme della collezione è individuato da un ***rappresentante*** che è uno degli elementi dell’insieme.

Operazioni su Insiemi Disgiunti

- ***Make-Set*(*x*)** : aggiunge alla struttura dati un nuovo insieme contenente solo l'elemento ***x.***
  - Si richiede che ***x*** non compaia in nessun altro insieme della struttura.

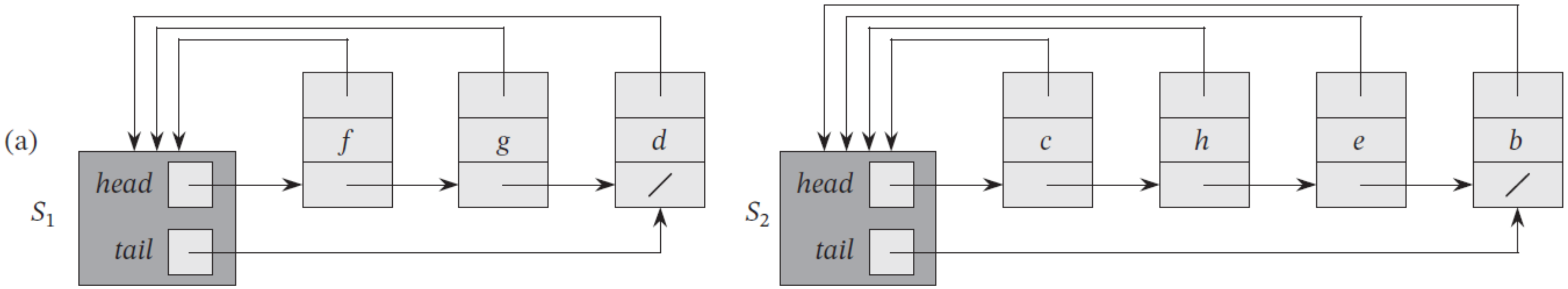
- ***Find-Set*(*x*)** : ritorna il rappresentante dell'insieme che contiene ***x.***

- ***Union*(*x*, *y*)** : riunisce i due insiemi contenenti ***x*** ed ***y*** in un unico insieme.

Rappresentazione di insiemi disgiunti

ESEMPIO

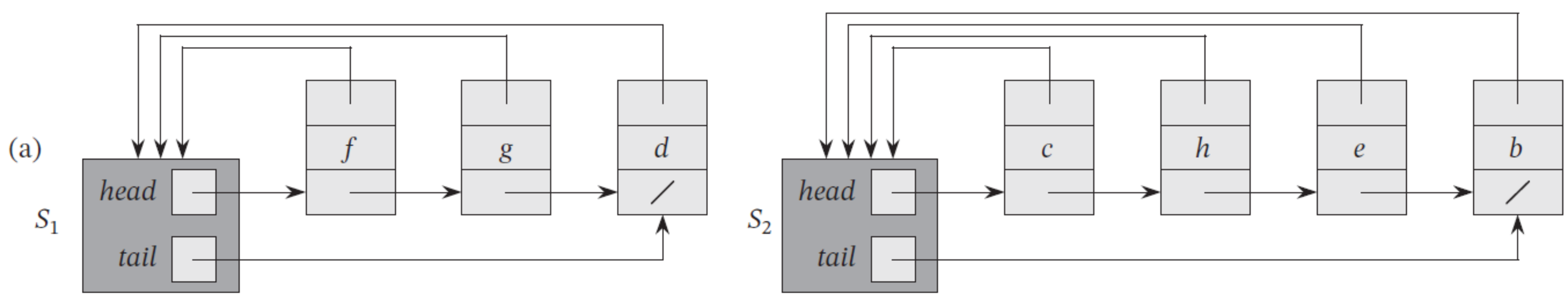
- $C = \{S_1, S_2\}$
  - S1={f, g, d} dove f è il rappresentante
  - S2={c, h, e, b} dove c è il rappresentante



Rappresentazione di insiemi disgiunti

ESEMPIO

- $C = \{S_1, S_2\}$
  - S1={f, g, d} dove f è il rappresentante
  - S2={c, h, e, b} dove c è il rappresentante



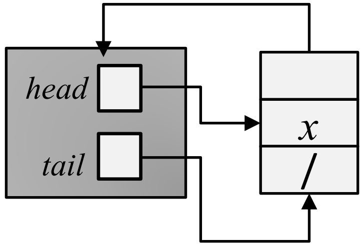
Rappresentazione di insiemi disgiunti

Make-Set(*x*) *%crea un insieme che contiene solo il valore x*

x.head = x

x.tail = x

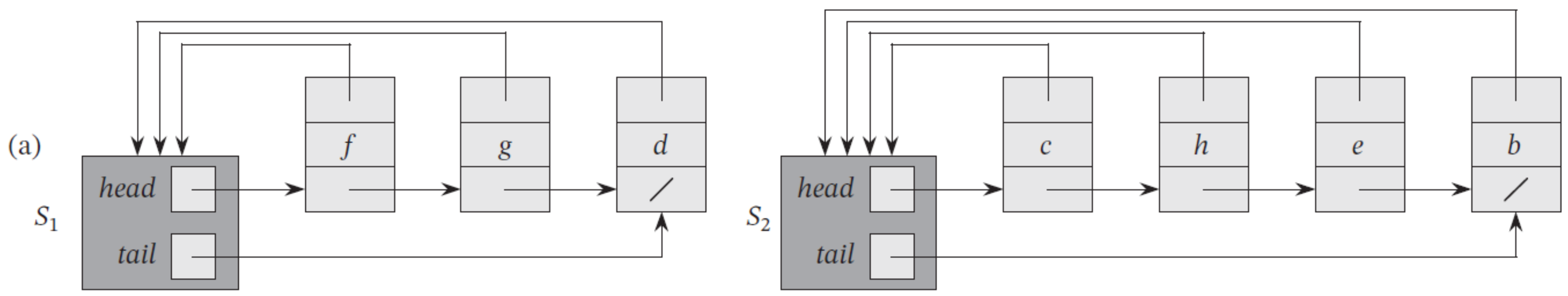
x.succ=/



Rappresentazione di insiemi disgiunti

Find-Set(*x*) *% ritorna il rappresentante dell'insieme che contiene **x***

return *x.head*

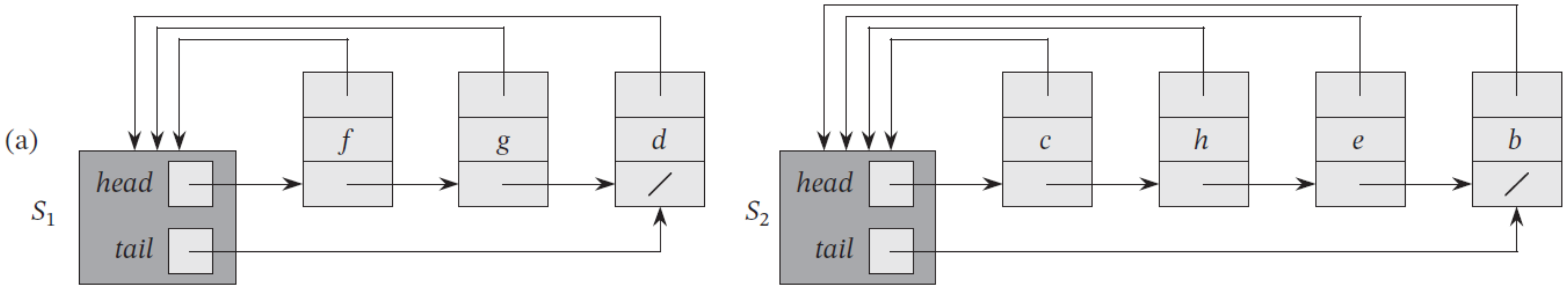


Rappresentazione di insiemi disgiunti

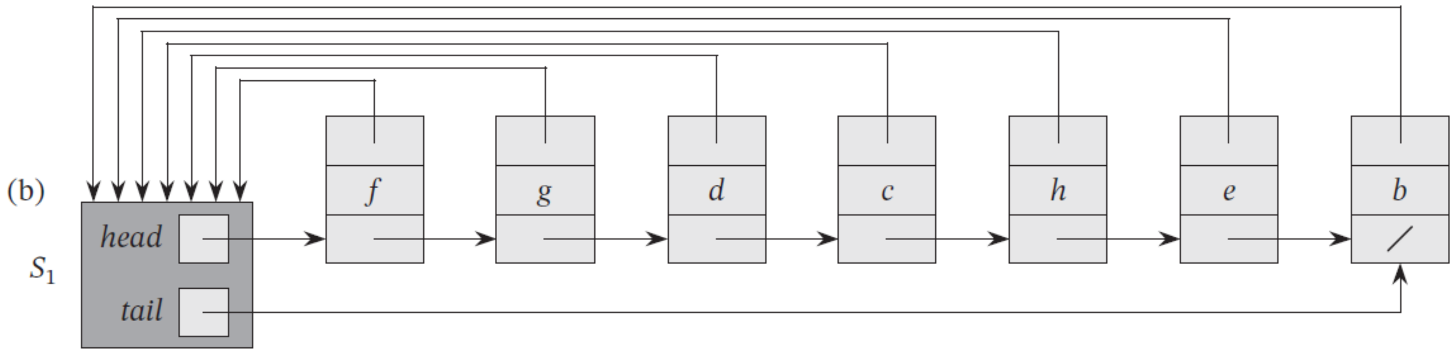
Union(*x*, *y*)

1. Fai puntare l'ultimo elemento di x al primo di y
   - costo costante
2. Cambia il puntatore finale della lista di x e settalo alla fine di y
   - costo costante
3. Per ogni elemento di y, aggiorna il puntatore alla testa di x
   - costo pari alla dimensione di y

Rappresentazione di insiemi disgiunti



Union(S_1, S_2)



Costo della rappresentazione di insiemi disgiunti con liste

Operazione	Descrizione	Stima del costo
Make-Set(x)	aggiunge alla struttura dati un nuovo insieme contenente solo l'elemento ***x***	O(1) per ogni x
Find-Set(x)	ritorna il rappresentante dell'insieme che contiene ***x***	O(1) data la struttura collegata con puntatori alla testa
Union(x, y)	riunisce i due insiemi contenenti ***x*** ed ***y*** in un unico insieme.	Dipende dalla dimensione di y e tende a n

Costo della rappresentazione di insiemi disgiunti con liste

Consideriamo la sequenza di **2*n*-1** operazioni:

Make-Set(x_1) // costo 1
Make-Set(x_2) // costo 1
....
Make-Set(x_n) // costo 1
Union(x_2, x_1) // costo 1
Union(x_3, x_1) // costo 2
Union(x_4, x_1) // costo 3
....
Union(x_n, x_1) // costo n-1

Il costo totale è proporzionale ad $\boldsymbol{n+n(n-1)/2}$ ed è $\boldsymbol{\Theta(n^2)}$
e le operazione hanno costo ammortizzato $\boldsymbol{O(n)}$.

Algoritmo di Kruskal

MST-KRUSKAL(G, ω) // *G* grafo con funzione peso ω

$A = \emptyset$

for "ogni $v \in G.V$"

    Make-Set(v)

crea una lista di tutti gli archi in *G.E*

ordina la lista degli archi in ordine monotono crescente rispetto al peso

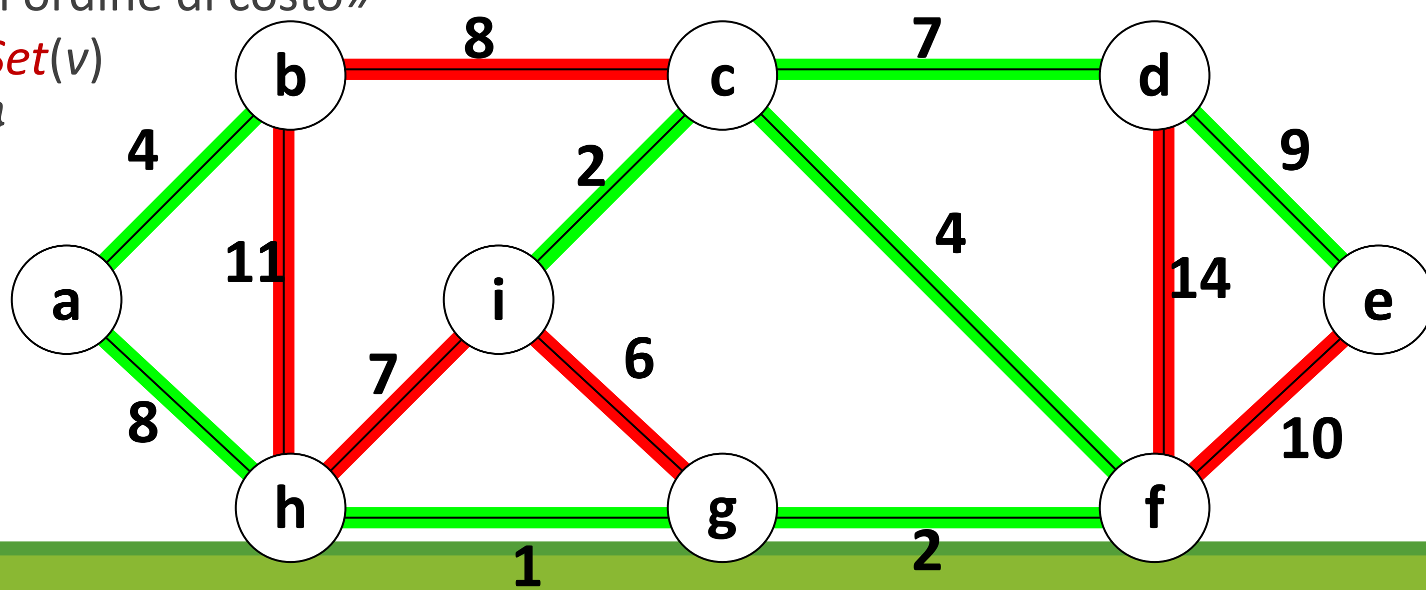
for "ogni arco $a = uv \in E[G]$ in ordine di costo»

    if *Find-Set*(u) $\neq$ *FindSet*(v)

        $A = A \cup \{uv\}$

        Union(u,v)

return *A*



Algoritmo di Prim

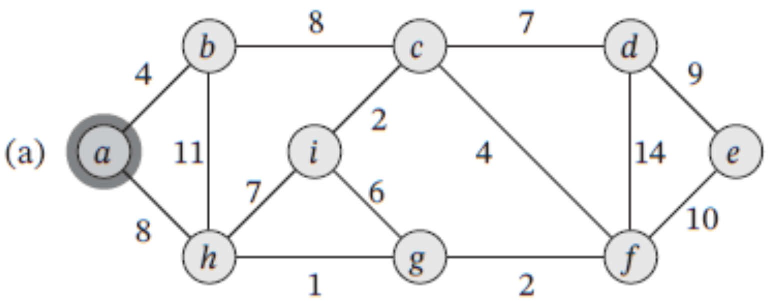
I vertici sono arricchiti con altre due informazioni:

a) un attributo ***p*** che è un puntatore al padre nell'albero in costruzione;

b) un attributo ***key*** che contiene il costo minimo di un arco che connette il vertice ad uno dei vertici già raggiunti dall'albero in costruzione.

Algoritmo di Prim

```
MST-Prim(G, ω, r)      // G grafo con funzione peso ω, r radice
for "ogni u ∈ G.V"
            u.key = ∞, u.p = nil
r.key = 0,  Q = Ø      // Q coda di priorità
for "ogni u ∈ G.V"
      Insert(Q, u)
while Q ≠ Ø
      u = Extract-Min(Q)
      for "ogni v ∈ G.Adj[u]"
            if v ∈ Q and ω(u,v) < v.key
                        v.p = u
                        v.key = ω(u,v)
                        Decrease-Key(Q, v, ω(u,v))
```



Algoritmo di Prim

MST-Prim$(G, \boldsymbol{\omega}, r)$ // G grafo con funzione peso $\boldsymbol{\omega}$, r radice

for "ogni $u \in G.V$"

  $u.key = \infty$, $u.p = nil$

$r.key = 0$, $Q = \emptyset$ // Q coda di priorità

for "ogni $u \in G.V$"

  Insert(Q, u)

while $Q \neq \emptyset$

  u = *Extract-Min*(Q)

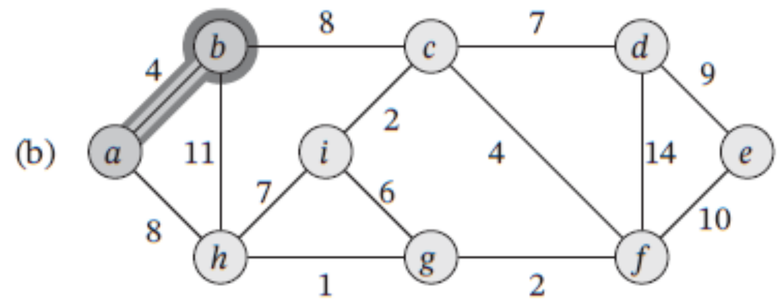
  for "ogni $v \in G.Adj[u]$"

    if $v \in Q$ and $\boldsymbol{\omega}(u,v) < v.key$

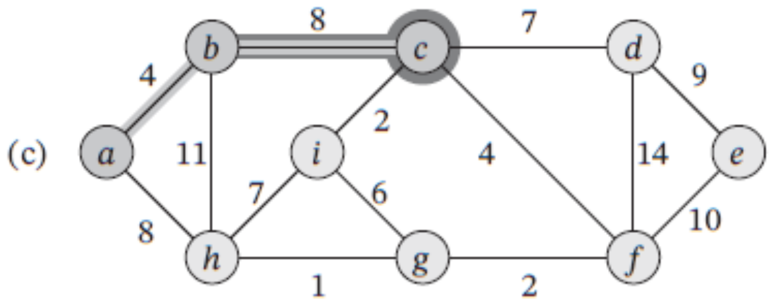
      $v.p = u$

      $v.key = \boldsymbol{\omega}(u,v)$

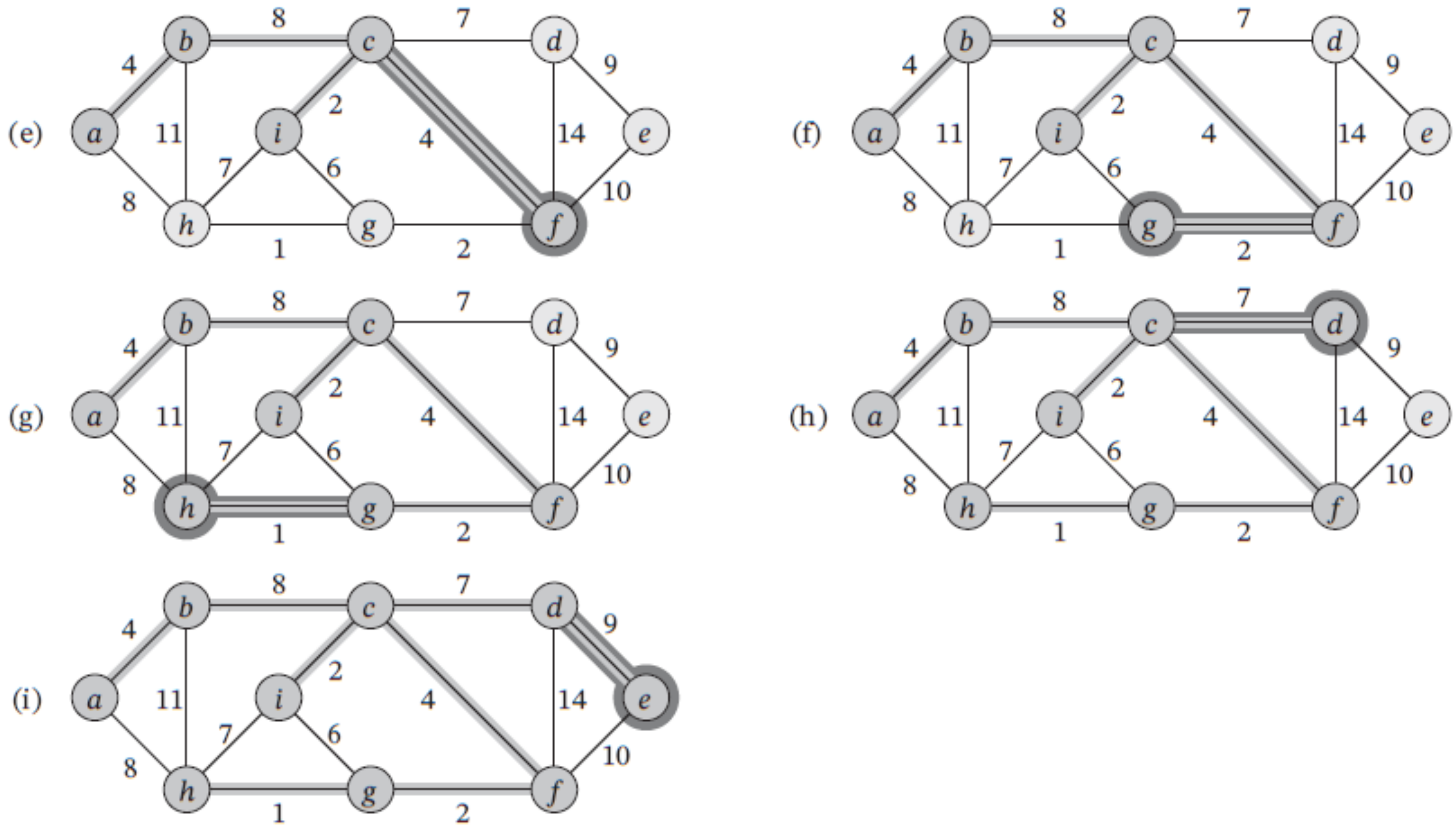
      Decrease-Key$(Q, v, \boldsymbol{\omega}(u,v))$



Algoritmo di Prim



Algoritmo di Prim



Algoritmo di Kruskal: Analisi della complessità

OSSERVAZIONE: La complessità dell’algoritmo di KRUSKAL dipende dal costo delle operazioni MAKE, UNIONI e FIND

Algorithm 3: Algoritmo di Kruskal per alberi di connessione minimi (MST)

MST − Kruskal$(\mathcal{G}, \omega)$
$A \leftarrow \emptyset$;
foreach $v \in \mathcal{G}.V$ **do**
  MAKE − SET(v);
end
$l \leftarrow$ converti_in_lista$(\mathcal{G}.E)$;
sort_decreasing(l, ω);
foreach $(u, v) \in l$ **do**
  if FIND − SET$(u) \neq$ FIND − SET(v) **then**
    $A \leftarrow A \cup \{(u, v)\}$;
    UNION(u, v);
  end
end
return A

Il primo ciclo for richiede un tempo *O*(|*V* | x cost(MAKE-SET))

L’ordinamento degli archi richiede un tempo ***O*(|*E*| log|*E*|)**

L’ultimo ciclo for richiede un tempo *O*(|*E*| x cost(UNION)).

Analisi del costo ammortizzato della struttura di insiemi disgiunti mediante liste

Consideriamo la sequenza di m=**2*n*-1** operazioni:

Make-Set(x_1) // costo 1
Make-Set(x_2) // costo 1
....
Make-Set(x_n) // costo 1
Union(x_2, x_1) // costo 1
Union(x_3, x_1) // costo 2
Union(x_4, x_1) // costo 3
....
Union(x_n, x_1) // costo *n*-1

n operazioni Make-Set ciascuna di costo unitario -> O(n)

n-1 operazioni Union ciascuna di costo $|x_i|$ -> $\sum_{i=1 \text{ to } n-1} i = n(n-1)/2$ -> $O(n^2)$

$O(n) + O(n^2)$ /m

O(n) per ciascuna operazione (Costo ammortizzato)

Euristica per la UNION pesata

OSSERVAZIONE

- La complessità $\Theta(n^2)$ (caso peggiore e O(n) nel caso medio) della realizzazione appena vista dipende dal fatto che, in ogni ***Union***, la seconda lista, quella che viene percorsa per aggiornare i puntatori al rappresentante, è la più lunga delle due.

Per migliorare le performance, modifichiamo UNION attraverso l'euristica dell'***unione pesata***

- sceglie sempre la lista più corta per aggiornare i puntatori al rappresentante.

- basta memorizzare la lunghezza della lista in un nuovo campo ***L*** del rappresentante.

Complessità con l'euristica della UNION pesata

Considerando la rappresentazione mediante liste collegate e l'euristica dell'unione pesata, una sequenza di ***m*** operazioni ***Make-Set***, ***Union*** e ***Find-Set*** delle quali ***n*** sono ***Make-Set***, richiede un tempo $O(m + n \log n)$.

DIMOSTRAZIONE

Consideriamo una sequenza di ***m*** operazioni ***Make-Set***, ***Union*** e ***Find-Set*** delle quali ***n*** sono ***Make-Set***

Tutte le operazioni richiedono un tempo costante eccetto ***Union*** che richiede un tempo costante più un tempo proporzionale al numero di puntatori al rappresentante che vengono modificati (i.e., proporzionale alla lunghezza della lista).

Complessità con l'euristica della UNION pesata

DIMOSTRAZIONE (cont.)

Il tempo richiesto dalla sequenza di ***m*** operazioni è quindi ***O*(*m* + *N*)**

- ***N*** è il numero totale di aggiornamenti dei puntatori al rappresentante eseguiti durante tutta la sequenza di operazioni.

Osserviamo che

- Il numero massimo di oggetti contenuti nella struttura è ***n***, pari al numero di ***Make-Set***.
- Quando un oggetto ***x*** viene creato esso appartiene ad un insieme di cardinalità 1
- Il rappresentante di ***x*** viene aggiornato quando l'insieme contenente ***x*** viene unito ad un insieme di cardinalità maggiore o uguale.

Complessità con l'euristica della UNION pesata

DIMOSTRAZIONE (cont.)

Ogni volta che viene aggiornato il puntatore al rappresentante di $\boldsymbol{x}$ la cardinalità dell'insieme a cui appartiene $\boldsymbol{x}$ viene almeno raddoppiata.

Siccome $\boldsymbol{n}$ è la massima cardinalità di un insieme il puntatore al rappresentante di $\boldsymbol{x}$ può essere aggiornato al più $\boldsymbol{\log_2 n}$ volte.

Quindi $\boldsymbol{N \leq n \log_2 n}$ da cui segue la complessità

Complessità con l'euristica della UNION pesata

COSTO AMMORTIZZATO

La complessità ammortizzata delle operazioni è:

$$O\left(\frac{m + n\log n}{m}\right) = O\left(1 + \frac{n\log n}{m}\right) = O(\log n)$$

Se il numero di ***Make-Set*** è molto minore del numero di ***Union*** e ***Find-Set*** per cui ***n*** **log *n*** = ***O*****(*m*)** allora

$$O\left(1 + \frac{n\log n}{m}\right) = O(1)$$

Foreste di Insiemi Disgiunti

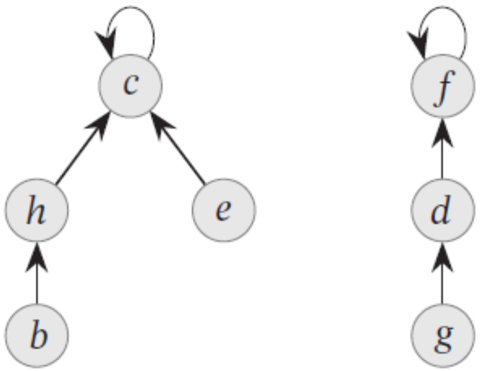
Esaminiamo adesso una struttura dati alternativa per rappresentare insiemi disgiunti: le FORESTE DI INSIEMI DISGIUNTI

Ogni insieme è rappresentato da un albero i cui nodi, oltre al campo ***data*** che contiene l'informazione, hanno soltanto un campo ***p*** che punta al padre.

Foreste di Insiemi Disgiunti

ESEMPIO

- $C = \{S_1, S_2\}$
  - S1={c, h, e, b} dove c è il rappresentante
  - S2={f, d, g} dove f è il rappresentante



(a)

Foreste di Insiemi Disgiunti

Make-Set(x)

  $x.p = x$

% Creo un albero con un solo nodo

Find-Set(x)

  while $x.p \neq x$

    $x = x.p$

  return x

% Segue i puntatori ai padri finchè non trova la radice dell’albero a cui appartiene x

Foreste di Insiemi Disgiunti

Union(x, y)

  x = *Find-Set*(x)

  y = *Find-Set*(y)

  $x.p = y$ // serve controllare se $x \neq y$?

% Aggiorno i puntatori facendo in modo che la radice di un albero punti a quella del secondo

Foreste di Insiemi Disgiunti

La complessità di ***Find-Set***(***x***) è pari alla lunghezza del cammino che congiunge il nodo ***x*** alla radice dell'albero.

La complessità di ***Union*** è essenzialmente quella delle due chiamate ***Find-Set***(***x***) e ***Find-Set***(***y***).

Un esempio analogo a quello usato con le liste mostra che una sequenza di ***n*** operazioni può richiedere un tempo $O(n^2)$.

Possiamo migliorare notevolmente l'efficienza usando due euristiche.

Costo della rappresentazione di insiemi disgiunti con foreste

Operazione	Descrizione	Stima del costo
Make-Set(x)	aggiunge alla struttura dati un nuovo insieme contenente solo l'elemento ***x***	O(1) per ogni x
Find-Set(x)	ritorna il rappresentante dell'insieme che contiene ***x***	O(log n) devo scorrere il cammino da x alla radice dell'albero che la include
Union(x, y)	riunisce i due insiemi contenenti ***x*** ed ***y*** in un unico insieme.	Coincide con il costo di Find-Set(x)

Euristica dell'unione per rango

In ogni nodo ***x*** manteniamo un campo ***rank***

- è un limite superiore all'altezza del sottoalbero di radice ***x*** ed è anche una approssimazione del logaritmo del numero di nodi del sottoalbero

L'operazione ***Union*** mette la radice con rango minore come figlia di quella di rango maggiore.

Make-Set(*x*)
  $x.p = x$
  $x.rank = 0$

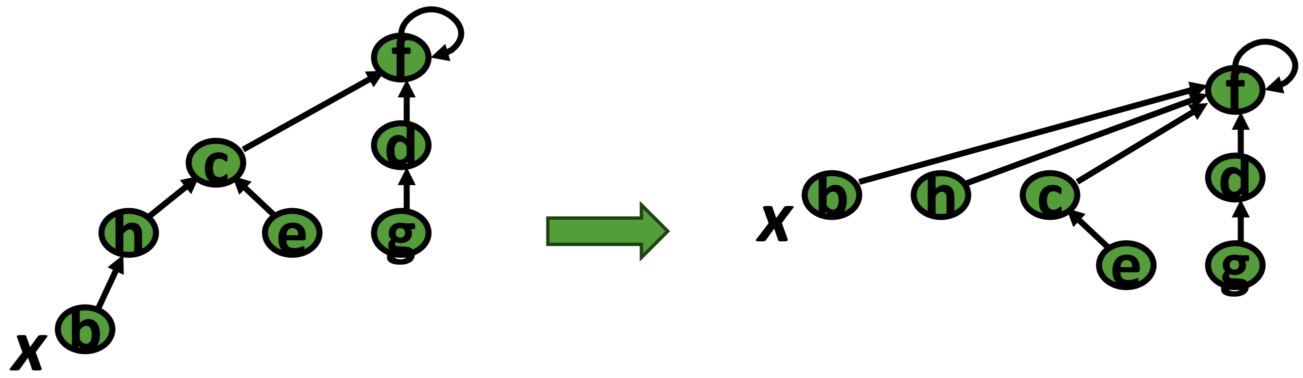
Euristica della compressione dei cammini

Quando effettuiamo una ***Find-Set*****(*****x*****)** attraversiamo il cammino da ***x*** alla radice

- ad ogni passo del cammino possiamo aggiornare i puntatori al padre facendoli puntare direttamente alla radice

Le successive operazioni ***Find-Set*** sui nodi di tale cammino risulteranno molto meno onerose.

```
Find-Set(x)
  if x.p ≠ x
    x.p = Find-Set(x.p)
  return x.p
```



Foreste di Insiemi Disgiunti

```
Union(x, y)
  x = Find-Set(x)
  y = Find-Set(y)
  Link(x, y)
```

```
Link(x, y)
  if x.rank > y.rank
    y.p = x
  else
    x.p = y
    if x.rank == y.rank
      y.rank = y.rank + 1
```

Foreste di Insiemi Disgiunti: complessità

EURISTICA DEL RANGO

Una sequenza di ***m*** operazioni delle quali ***n*** sono ***Make-Set*** richiede un tempo

$$O(m \log n)$$

EURISTICA DI COMPRESSIONE DEI CAMMINI

Una sequenza di ***m*** operazioni delle quali ***n*** sono ***Make-Set*** e ***k*** sono ***Find-Set*** richiede un tempo

$$\Theta(k \log_{(1+k/n)} n) \qquad \text{se } k \geq n$$

Foreste di Insiemi Disgiunti: complessità

Le migliori prestazioni in assoluto si ottengono usando entrambe le euristiche.

Una sequenza di m operazioni delle quali n sono ***Make-Set*** richiede un tempo **O(*m* α(*n*))** ,

- dove $\alpha(n)$ è una funzione che cresce estremamente lentamente:
  - $\alpha(n) \leq 4$ in ogni concepibile uso della struttura dati.

La complessità ammortizzata di una singola operazione risulta quindi **O(α(*n*))**: praticamente costante.

Algoritmo di Prim: Analisi della complessità

OSSERVAZIONE: La complessità dell'algoritmo di Prim dipende da come è implementata la lista di priorità

Algorithm 4: Algoritmo di PRIM per alberi di connessione minimi (MST)

MST − Prim$(\mathcal{G}, \omega, r)$
foreach $v \in \mathcal{G}.V$ **do**
    $u.key \leftarrow \infty$;
    $u.p \leftarrow Null$;
end
$r.key \leftarrow 0$;
$Q \leftarrow \emptyset$;
foreach $u \in \mathcal{G}.V$ **do**
    INSERT(Q, u);
end
while $Q \neq \emptyset$ **do**
    $u \leftarrow$ EXTRACT_MIN(Q);
    foreach $v \in \mathcal{G}.Adj[u]$ **do**
        if $v \in Q$ ***AND*** $\omega(u, v) < v.key$ **then**
            $v.p \leftarrow u$;
            $v.key \leftarrow \omega(u, v)$;
            DECREASE_KEY$(Q, v, \omega(u, v))$;
        end
    end
end

Consideriamo un'implementazione della cosa basata su min-heap

Algoritmo di Prim: Analisi della complessità

Consideriamo un’implementazione della cosa basata su min-heap

Algorithm 4: Algoritmo di PRIM per alberi di connessione minimi (MST)

```
MST − Prim(𝒢, ω, r)
foreach v ∈ 𝒢.V do
    u.key ← ∞;
    u.p ← Null;
end
r.key ← 0;
Q ← ∅;
foreach u ∈ 𝒢.V do
    INSERT(Q, u);
end
while Q ≠ ∅ do
    u ← EXTRACT_MIN(Q);
    foreach v ∈ 𝒢.Adj[u] do
        if v ∈ Q AND ω(u, v) < v.key then
            v.p ← u;
            v.key ← ω(u, v);
            DECREASE_KEY(Q, v, ω(u, v));
        end
    end
end
```

Il primo ciclo for richiede un tempo ***O*(|*V* |)**

l’inserimento di tutti i vertici nella coda richiede un tempo ***O*(|*V* |)**

Algoritmo di Prim: Analisi della complessità

Consideriamo un'implementazione della cosa basata su min-heap

Algorithm 4: Algoritmo di PRIM per alberi di connessione minimi (MST)

MST − Prim$(\mathcal{G}, \omega, r)$
foreach $v \in \mathcal{G}.V$ **do**
    $u.key \leftarrow \infty$;
    $u.p \leftarrow Null$;
end
$r.key \leftarrow 0$;
$Q \leftarrow \emptyset$;
foreach $u \in \mathcal{G}.V$ **do**
    INSERT(Q, u);
end
while $Q \neq \emptyset$ **do**
    $u \leftarrow$ EXTRACT_MIN(Q);
    foreach $v \in \mathcal{G}.Adj[u]$ **do**
        if $v \in Q$ ***AND*** $\omega(u, v) < v.key$ **then**
            $v.p \leftarrow u$;
            $v.key \leftarrow \omega(u, v)$;
            DECREASE_KEY$(Q, v, \omega(u, v))$;
        end
    end
end

Il ciclo while viene eseguito **|*V* |** volte

Extract-Min richiede un tempo ***O*(log|*V* |)**

I cicli for interni visitano tutte le liste delle adiacenze dei vertici (2|E| iterazioni)

Poiché ***Decrease-Key*** richiede un tempo ***O*(log|*V* |)**, richiedono in totale un tempo ***O*(|*E*|log|*V* |)**

Algoritmo di Prim: Analisi della complessità

Consideriamo un'implementazione della cosa basata su min-heap

Algorithm 4: Algoritmo di PRIM per alberi di connessione minimi (MST)

```
MST − Prim(𝒢, ω, r)
foreach v ∈ 𝒢.V do
    u.key ← ∞;
    u.p ← Null;
end
r.key ← 0;
Q ← ∅;
foreach u ∈ 𝒢.V do
    INSERT(Q, u);
end
while Q ≠ ∅ do
    u ← EXTRACT_MIN(Q);
    foreach v ∈ 𝒢.Adj[u] do
        if v ∈ Q AND ω(u, v) < v.key then
            v.p ← u;
            v.key ← ω(u, v);
            DECREASE_KEY(Q, v, ω(u, v));
        end
    end
end
```

Pertanto la complessità è $O(|V| \log |V| + |E| \log |V|) \rightarrow O(|E| \log |V|)$

Algoritmo di Prim: Analisi della complessità

OSSERVAZIONE

- Se usiamo un Heap di Fibonacci ottengo un incremento di performance nel caso medio con una complessità di $O(|E| + |V| \log|V|)$ in quanto
  - EXTRACT-MIN ha un costo ammortizzato di $O(\log |V|)$
  - INSERT e DECREASE-KEY hanno un costo ammortizzato di $O(1)$