Elementi di Teoria dei Grafi

Argomenti lezione:

- Il problema dell'albero ricoprente a costo minimo ("*minimum spanning tree*")
- Condizioni di ottimalità
- Algoritmo di Kruskal
- Algoritmi di Prim–Dijkstra
- Esercizi

dariano,pacciarelli@dia.uniroma3

Introduzione

Un **albero ricoprente** di un grafo connesso è un albero che collega tutti i vertici del grafo.

Questo problema insorge in numerose applicazioni.

Tali applicazioni pratiche possono esistere nel data mining (cluster analysis), nella costruzione di circuiti elettronici, reti di computer, autostrade o reti elettriche, solo per citare alcuni esempi.

Albero ricoprente a costo minimo

Si consideri un **grafo (non orientato) $G=(V, E)$ pesato e connesso**.
Obiettivo: individuare un albero ricoprente tale che la somma dei pesi dei lati appartenenti all'albero sia minimizzata.

Se alcuni pesi dei lati possono assumere valori uguali tra di loro, sarà sufficiente trovare una sola soluzione ottima tra le molte che potrebbero esistere con lo stesso valore della funzione obiettivo.

Indicheremo con T_G^* un albero ricoprente a costo minimo.

Dato un albero ricoprente generico T_G, chiameremo :
$E(T_G)$ l'insieme di lati appartenenti all'albero ($e \in T_G$)
$E / E(T_G)$ l'insieme di lati non appartenenti ad esso ($f \notin T_G$)

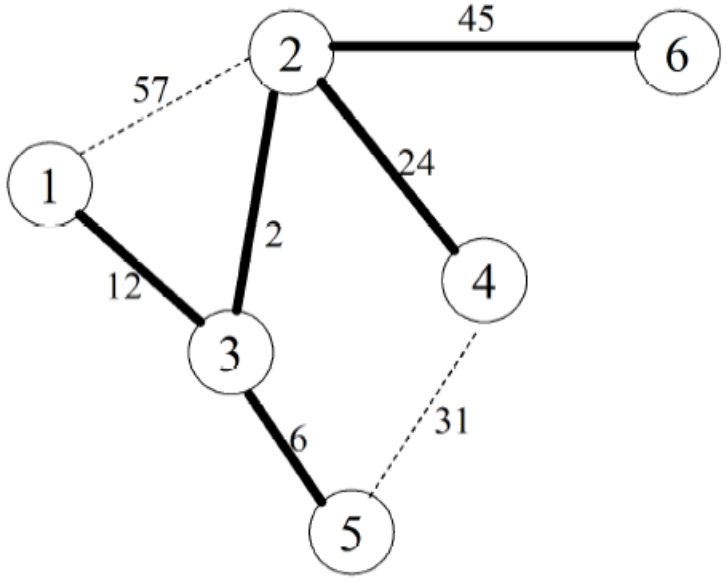
dariano,pacciarelli@dia.uniroma3

Ciclo fondamentale

- Dato un albero ricoprente si può notare come l'aggiunta di un qualunque lato ad esso crea esattamente un *ciclo fondamentale*
- Un grafo contiene $m - n + 1$ ($= |E| - |V| + 1$) lati che appartengono al grafo ma non appartengono al suo albero ricoprente

$\Rightarrow$ $m - n + 1$ cicli fondamentali

- L'eliminazione di un qualunque lato dal ciclo fondamentale genera un altro albero ricoprente



$m = 7$

$n = 6$

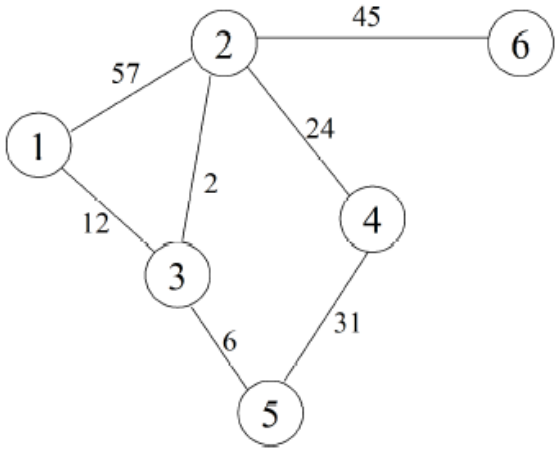
$\Rightarrow$ 2 lati non appartengono all'albero ricoprente T_G ma appartengono al grafo G

$E / E(T_G) = \{(1,2), (4,5)\}$

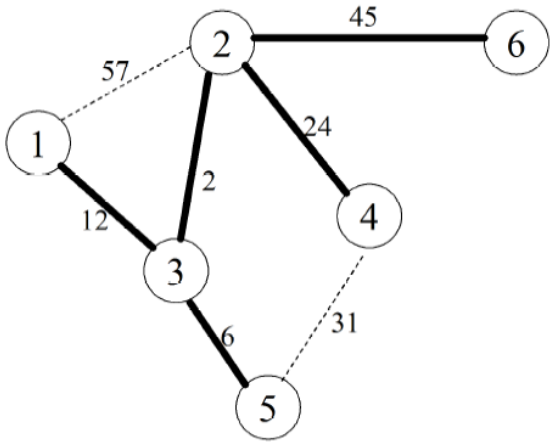
dariano,pacciarelli@dia.uniroma3

Taglio fondamentale

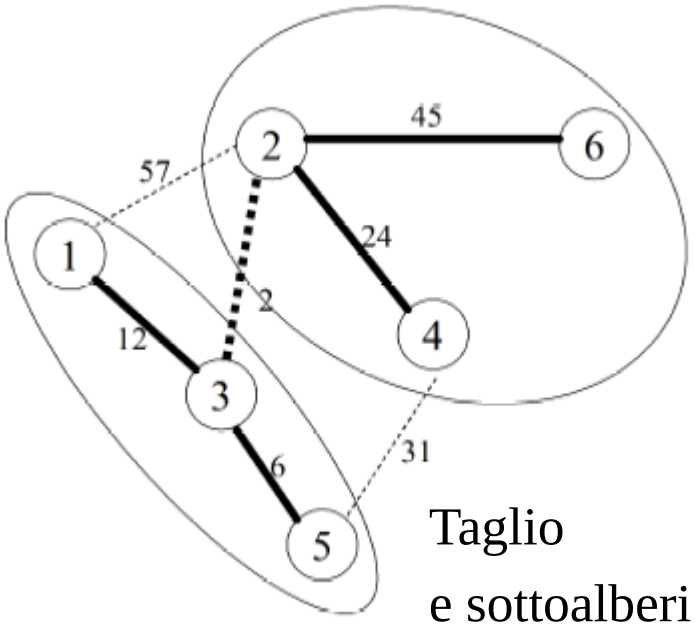
- Dati un grafo connesso e un suo albero ricoprente, si può notare che la rimozione di un qualunque lato dall'albero ricoprente genera due sottoalberi, che definiscono un *taglio fondamentale*.
- Visto che un albero contiene $n - 1$ lati sarà possibile definire esattamente $n - 1$ tagli fondamentali.
- Aggiungendo un lato appartenente al taglio si ottiene nuovamente un albero ricoprente.



Grafo connesso



Albero ricoprente



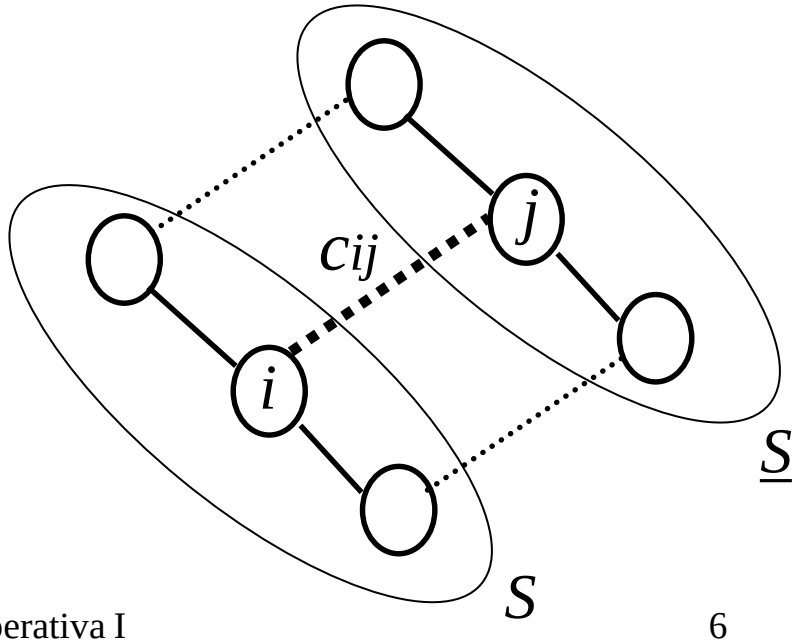
Taglio
e sottoalberi

dariano,pacciarelli@dia.uniroma3

Condizioni di ottimalità sui tagli (1)

Teorema: Dato un grafo $G = (V, E)$, con costo c_{ij} associato al lato $(i, j) \in E$, un albero ricoprente T^*_G di G è minimo se e solo se per ogni lato $(i, j) \in E(T^*_G)$, detto $[S, \underline{S}]$ il taglio di G definito dagli insiemi dei vertici delle due componenti connesse ottenute da T^*_G rimuovendo il lato (i, j), risulta valida la condizione :

$c_{ij} \leq c_{pq}$ per ogni $(p, q) \in (S, \underline{S})$.



dariano,pacciarelli@dia.uniroma3

Condizioni di ottimalità sui tagli (2)

Dimostrazione Necessità:

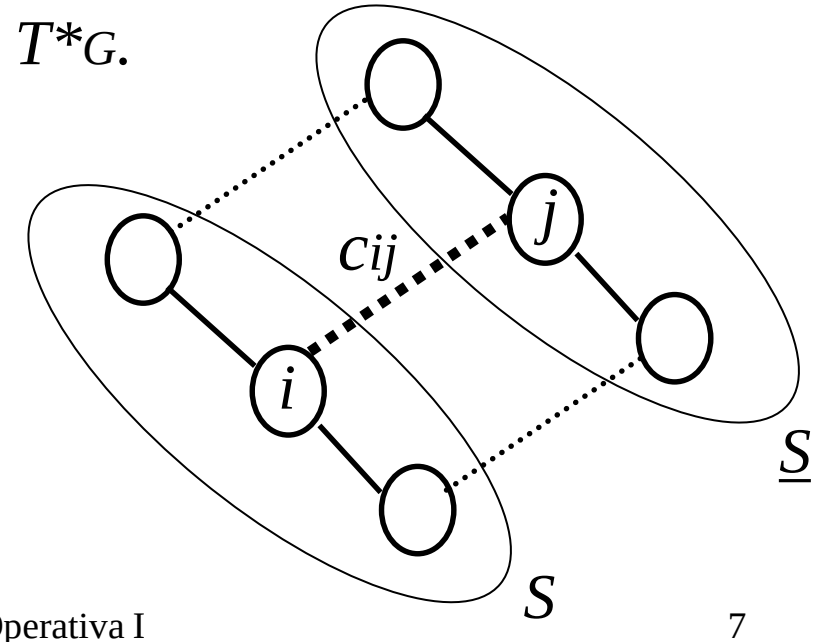
Sia T^*_G un minimo albero ricoprente di G e sia $[S, \underline{S}]$ il taglio di G definito a seguito della rimozione del lato $(i, j) \in E(T^*_G)$ da T^*_G.

Se (per assurdo) esistesse un lato $(p,q) \in (S, \underline{S})$ con $c_{ij} > c_{pq}$,

sostituendo (i, j) con (p, q) in T^*_G

si otterrebbe un nuovo albero ricoprente di G di costo inferiore a T^*_G,

ma ciò è impossibile data l'ipotesi su T^*_G.



dariano,pacciarelli@dia.uniroma3

Condizioni di ottimalità sui tagli (3)

Dimostrazione Sufficienza (I):

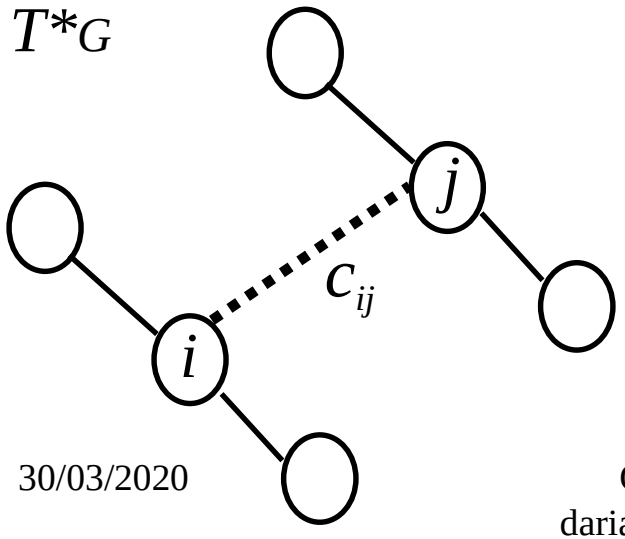
Sia T^*_G un albero ricoprente che soddisfa la condizione di ottimalità, sia T'_G un minimo albero ricoprente di G. Domanda: T^*_G è ottimo?

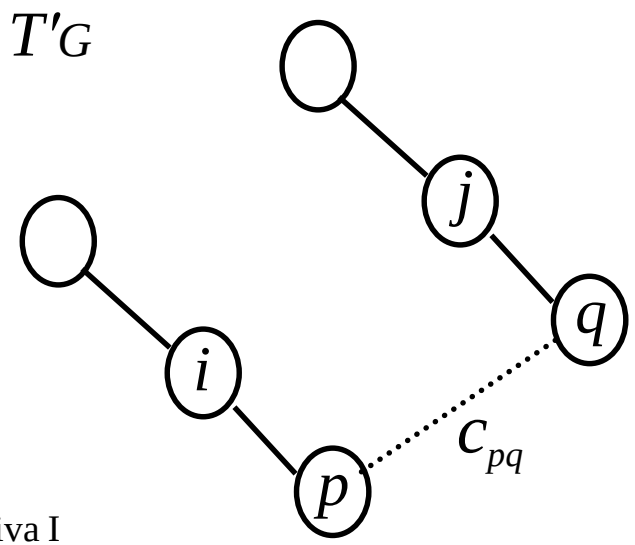
Se $T'_G \neq T^*_G$ allora esiste almeno un lato $(i, j) \in E \setminus E(T'_G)$ in T^*_G .

I lati di T^*_G soddisfano la condizione di ottimalità, e quindi $c_{ij} \leq c_{pq}$;

poichè T'_G è un minimo albero ricoprente, si ha che $c_{pq} \leq c_{ij}$;

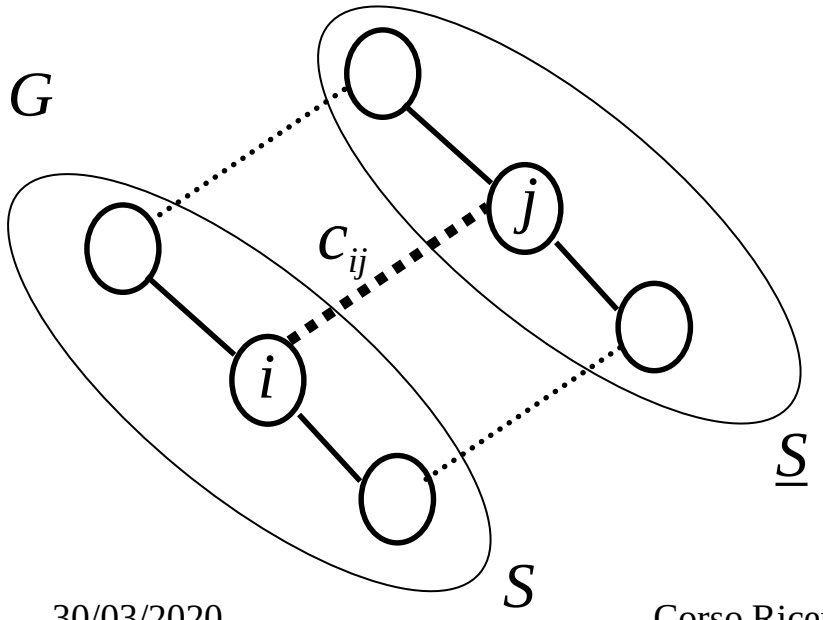
di conseguenza, $c_{ij} = c_{pq}$.

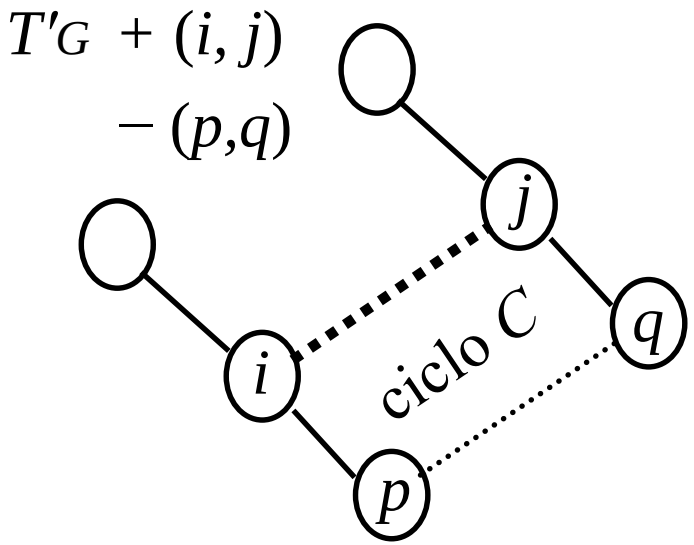




Condizioni di ottimalità sui tagli (4)

Dimostrazione Sufficienza (II): Sia [S, $\underline{S}$] il taglio di G definito dalla rimozione del lato (i, j) da T^*_G. Se si aggiunge il lato (i, j) a T'_G si crea un ciclo C che deve contenere almeno un (p, q) $\in$ (S, $\underline{S}$). Dato che $c_{ij} = c_{pq}$, sostituire (p, q) con (i, j) in T'_G non comporta alcun aumento del costo di tale albero, ma si ottiene un nuovo minimo albero ricoprente di G che ha (i, j) in comune con T^*_G.



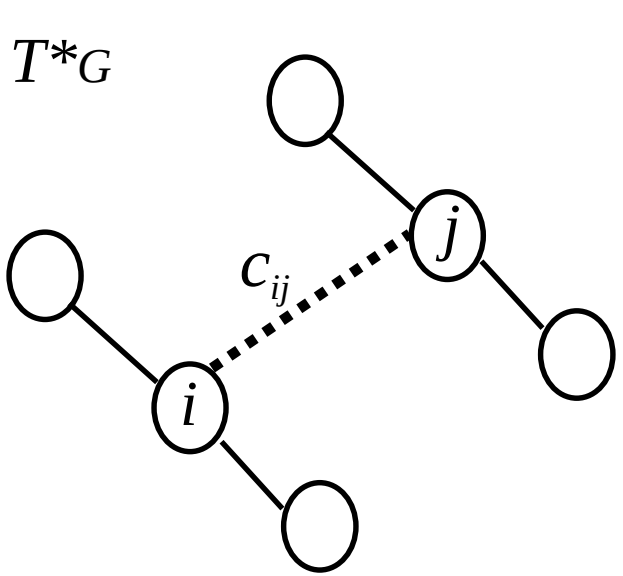


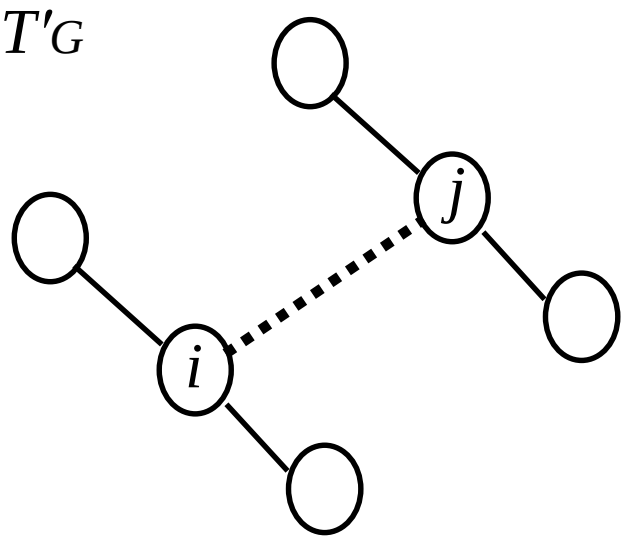
dariano,pacciarelli@dia.uniroma3

Condizioni di ottimalità sui tagli (5)

Dimostrazione Sufficienza (III):

Ripetendo tale procedura per i lati di T^*_G non appartenenti a T'_G,
si riesce a trasformare l'albero T'_G nell'albero T^*_G
conservando a ogni sostituzione la proprietà di ottimalità,
il che dimostra che anche T^*_G è un minimo albero ricoprente.



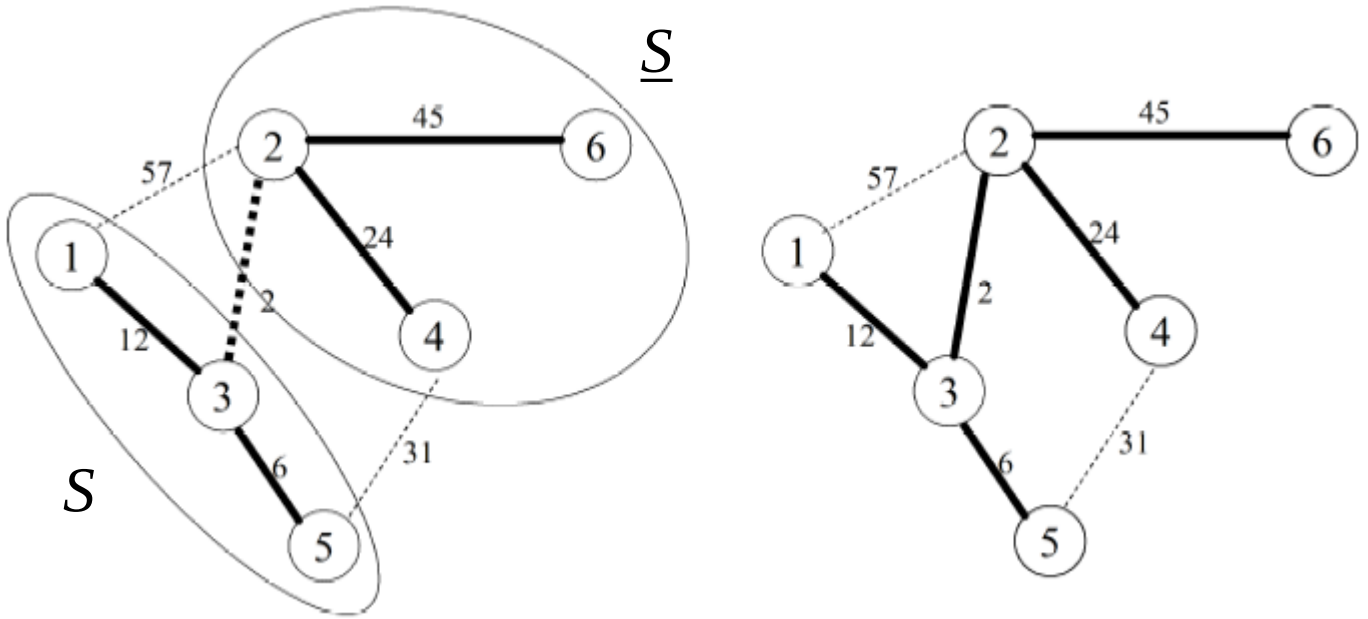


dariano,pacciarelli@dia.uniroma3

Condizioni di ottimalità sui tagli (6)

Il teorema ci suggerisce un algoritmo che ad ogni passo considera un taglio differente e aggiunge all'albero il lato a costo minimo.

Esempio: $(2,3) \in E(T_G)$ con peso 2 è il lato che minimizza il taglio, mentre i lati $\in E / E(T_G)$ nel taglio $[S, \underline{S}]$ hanno pesi 31 e 57.

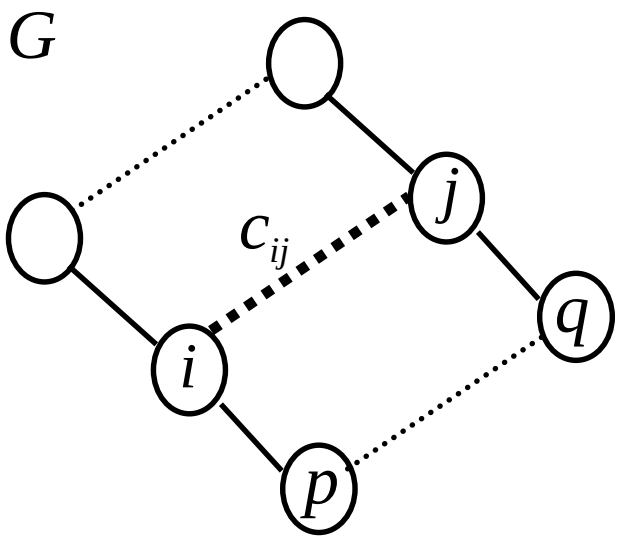


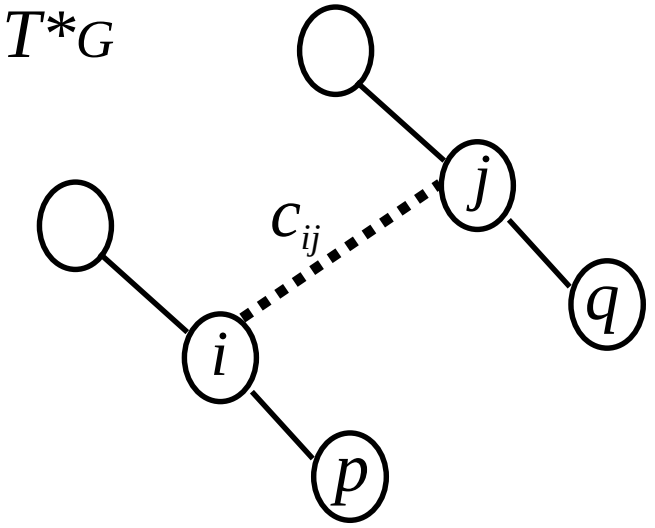
dariano,pacciarelli@dia.uniroma3

Condizioni di ottimalità sui cammini (1)

Teorema:

Dato un grafo $G = (V, E)$, con costo c_{ij} associato a $(i, j) \in E$, un albero ricoprente T^*_G del grafo G è minimo se e solo se per ogni $(p,q) \in E \setminus E(T^*_G)$ e detto P_{pq} il cammino tra p e q in T^*_G risulta valida la condizione $c_{ij} \leq c_{pq}$ per ogni lato $\in P_{pq}$.





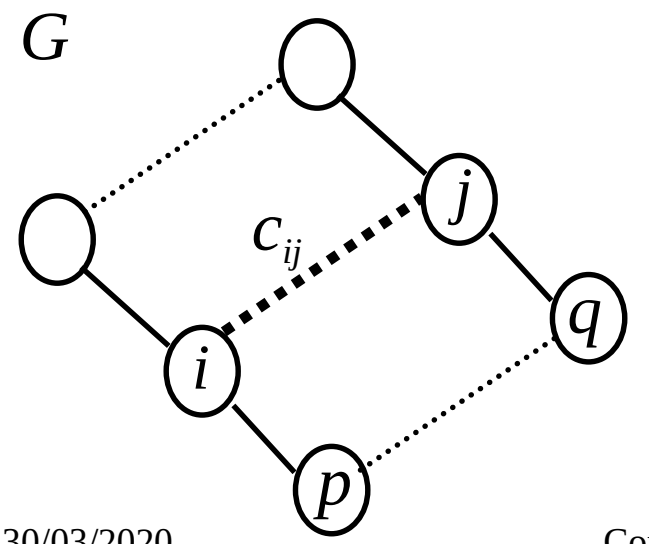
Condizioni di ottimalità sui cammini (2)

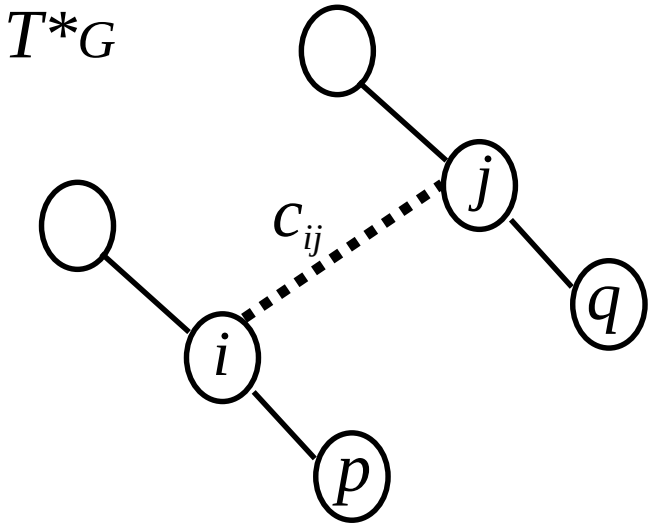
Dimostrazione Necessità:

Ipotizziamo che T^*_G è un minimo albero ricoprente di G.

Per assurdo, si ipotizza l'esistenza di un lato $(p, q) \in E \setminus E(T^*_G)$ e di un lato (i, j) del cammino tra p e q in T^*_G tali che $c_{ij} > c_{pq}$.

Sostituendo (i, j) con (p, q) in T^*_G si ottiene un nuovo albero ricoprente di costo inferiore a T^*_G, contraddicendo l'ipotesi su T^*_G.



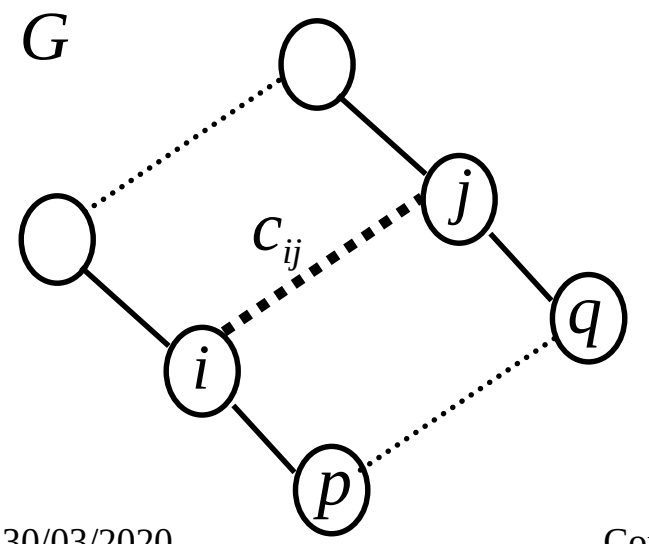


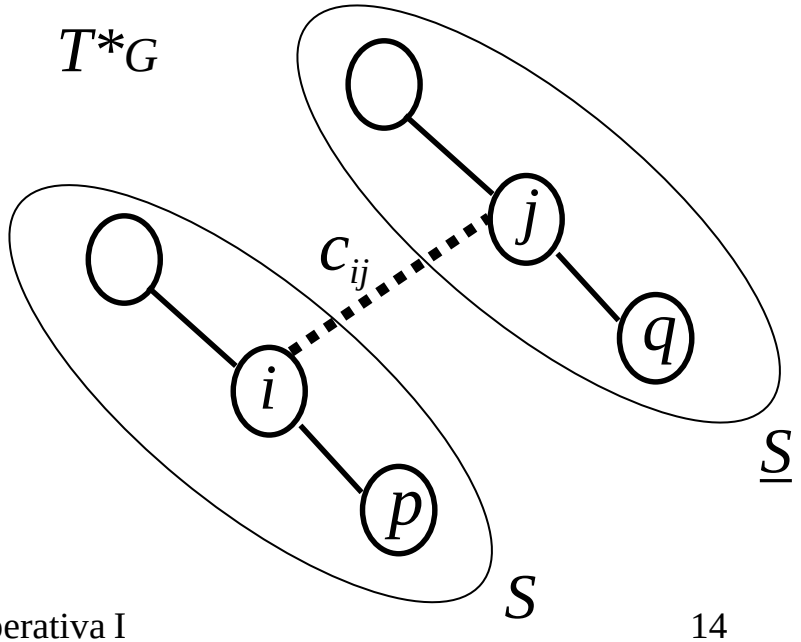
Condizioni di ottimalità sui cammini (3)

Dimostrazione Sufficienza (I):

Per ipotesi su T^*_G abbiamo $c_{ij} \leq c_{pq}$ per ogni lato $\in P_{pq}$

Date le componenti connesse S *e* $\underline{S}$ ottenute rimuovendo (i, j) da T^*_G vogliamo dimostrare che $c_{ij} \leq c_{pq}$ per ogni $(p, q) \in (S, \underline{S})$





dariano,pacciarelli@dia.uniroma3

Condizioni di ottimalità sui cammini (4)

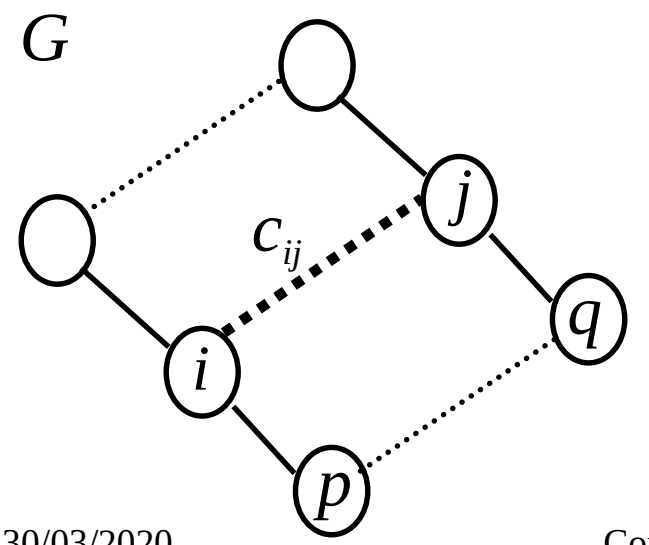
Dimostrazione Sufficienza (II):

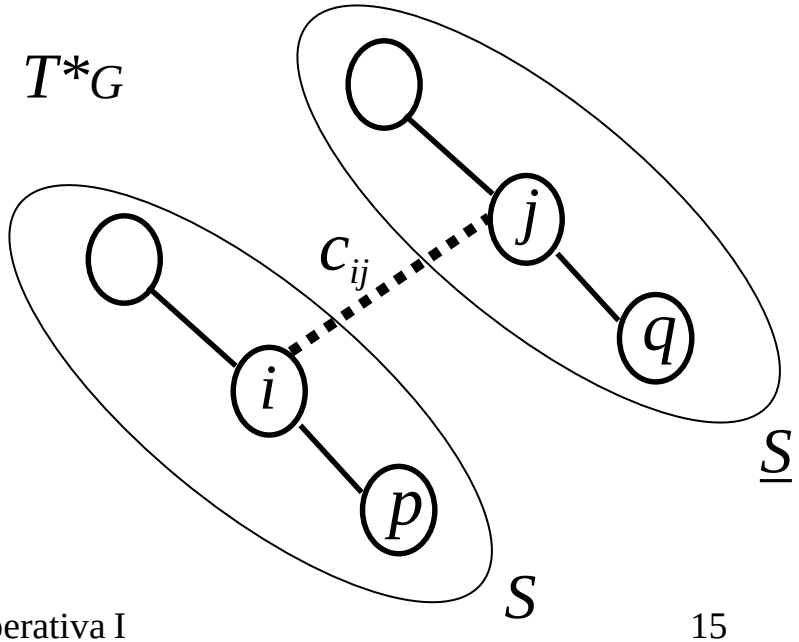
T^*_G contiene un unico cammino P_{pq} tra p e q. Tale cammino passa per lo spigolo (i, j), che connette in T^*_G vertici di S con quelli di $\underline{S}$.

Per ipotesi su T^*_G abbiamo $c_{ij} \leq c_{pq}$ per ogni lato $\in P_{pq}$

segue che $c_{ij} \leq c_{pq}$ per ogni $(p, q) \in (S, \underline{S})$

da cui T^*_G è un minimo albero ricoprente.

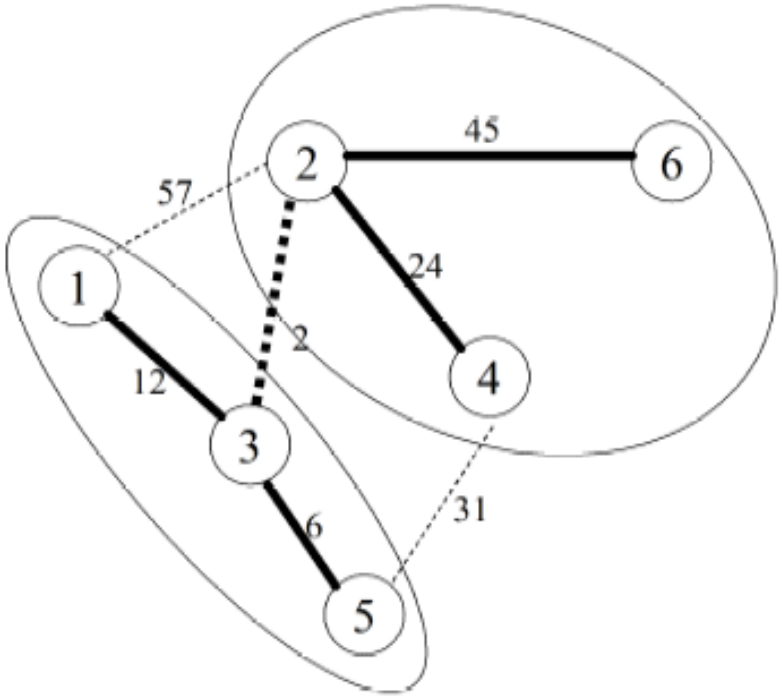




dariano,pacciarelli@dia.uniroma3

Condizioni di ottimalità sui cammini (5)

Esempio: Si noti come il lato (4,5) $\in E / E(T_G)$ genera un cammino P_{45} tra 4 e 5 composto solamente da lati $\in E(T_G)$ di peso inferiore. Infatti il peso del lato (4,5) detto $c_{45} = 31$, mentre i pesi dei lati $\in E(T_G)$ sono $c_{42} = 24$, $c_{23} = 2$ e $c_{35} = 6$



dariano,pacciarelli@dia.uniroma3

Algoritmo di Kruskal (1)

Sfruttando le condizioni di ottimalità sui cammini è possibile costruire un algoritmo noto come *Algoritmo di Kruskal*:

0) L'algoritmo parte con un albero ricoprente vuoto
(lista di lati T = insieme vuoto)

1) Si ordinano i lati in ordine crescente di peso in una lista

2) Si scorre la lista *prendendo* ad ogni passo il lato con peso minore
[Ad ogni passo si deve decidere se il lato considerato va aggiunto all'albero ricoprente di costo minimo o meno.]

3) Alla fine si ottiene un albero ricoprente a costo minimo.

Algoritmo di Kruskal (2)

L'aggiunta di un lato all'albero in costruzione può portare a due casi:

A. si genera un ciclo

B. vengono connesse 2 componenti dell'albero

Se il lato appena aggiunto ha entrambe le estremità nella stessa componente allora viene generato un ciclo, e il lato aggiunto sarà quello di costo maggiore all'interno del ciclo, quindi in una soluzione ottima tale lato non apparterrà all'albero ricoprente di costo minimo.

Altrimenti il lato considerato congiunge due componenti a costo minimo e necessariamente apparterrà all'albero di costo minimo.

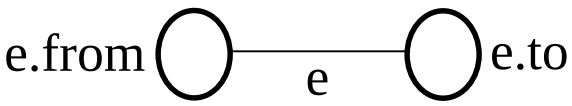
Algoritmo di Kruskal (3)

Una possibile implementazione:

vettore LATI:
contiene i lati ordinati

vettore COMP:
stabilisce per ogni vertice la sua componente

lista *T*:
albero ricoprente T_G contenente *n* -1 lati



Algoritmo di Kruskal

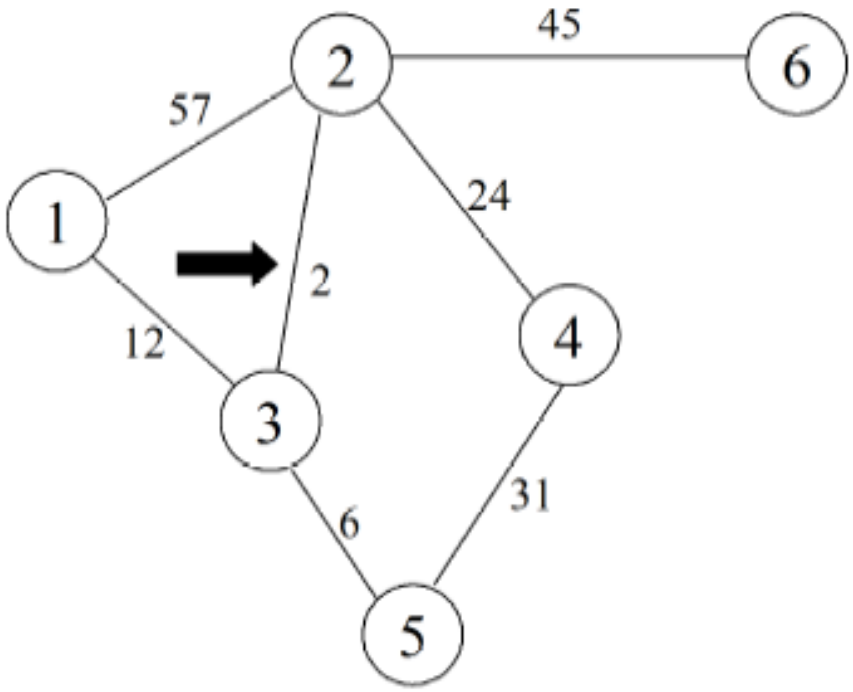
```
//Inizializzazione
COMP[i] = i, ∀i ∈ V;  (una COMP per ogni vertice)
Ordina tutti i lati in LATI ;  (per costi crescenti)
i = 0;
T = ∅;
// Ciclo Principale
// Fino a che non ho aggiunto n − 1 lati
while ( |T| < n − 1 )
{
    // Considera il primo lato della lista
    e = LATI [i];
    // Se il lato unisce due componenti separate...
    if ( COMP[e.from] != COMP[e.to] )
    then
    // ...aggiorna i dati
    {
        fondi i due elementi di COMP;
        T = T ∪ e;
    }
    i++;
}
```

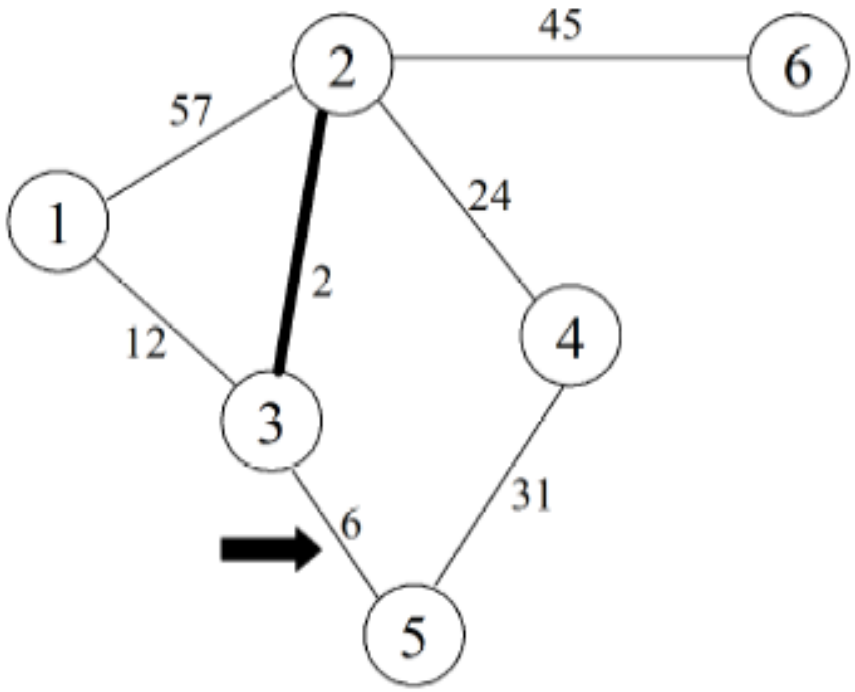
Algoritmo di Kruskal (4)

Esempio

- Ordino i lati per costi (pesi) crescenti ottenendo la seguente sequenza (2,3), (3,5), (1,3), (2,4), (4,5), (2,6), (1,2).
- I iterazione: aggiungo il lato (2,3)

COMP = [1, 2, 3, 4, 5, 6] => COMP = [1, 2, 2, 4, 5, 6]





dariano,pacciarelli@dia.uniroma3

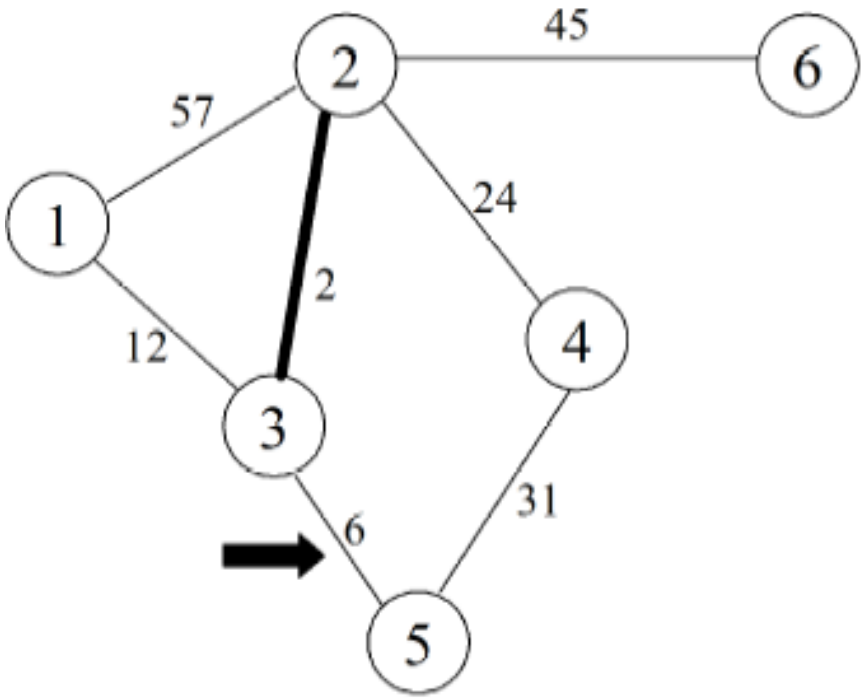
Algoritmo di Kruskal (5)

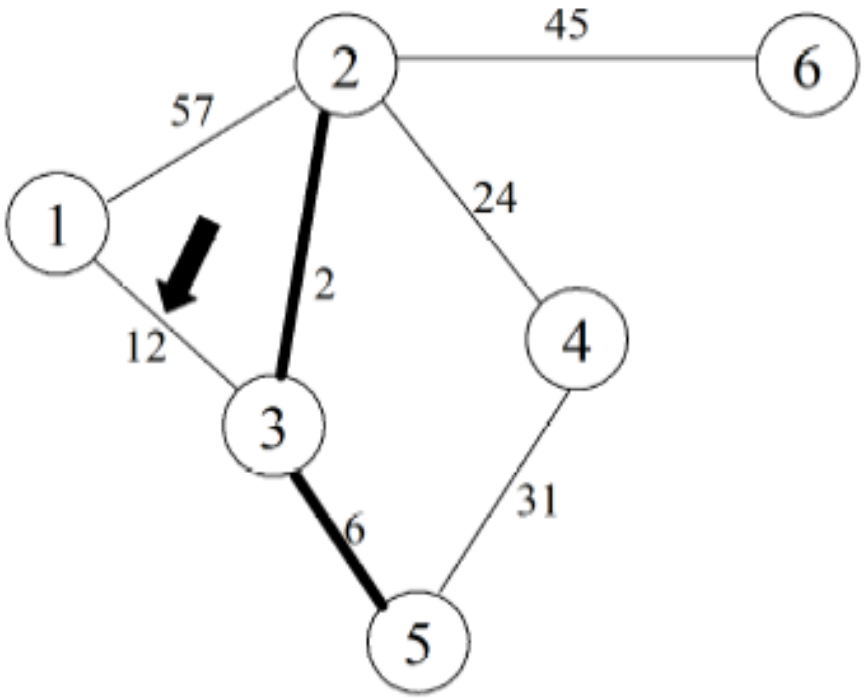
(2,3), (3,5), (1,3), (2,4), (4,5), (2,6), (1,2)

Esempio

- II iterazione: aggiungo il lato (3,5)

COMP = [1, 2, 2, 4, 5, 6] => COMP = [1, 2, 2, 4, 2, 6]





dariano,pacciarelli@dia.uniroma3

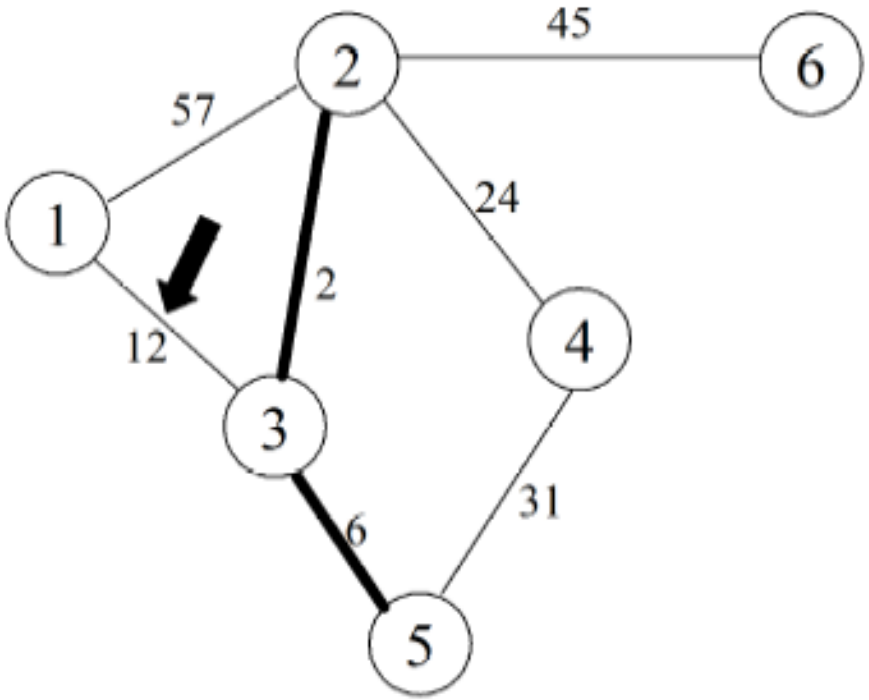
Algoritmo di Kruskal (6)

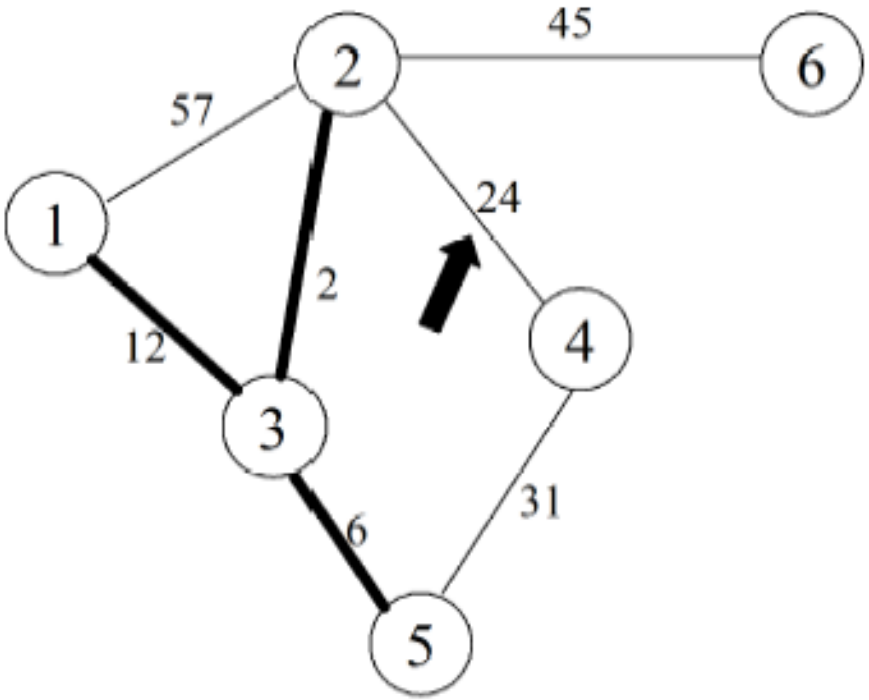
(2,3), (3,5), (1,3), (2,4), (4,5), (2,6), (1,2)

Esempio

- III iterazione: aggiungo il lato (1,3)

COMP = [1, 2, 2, 4, 2, 6] => COMP = [1, 1, 1, 4, 1, 6]





dariano,pacciarelli@dia.uniroma3

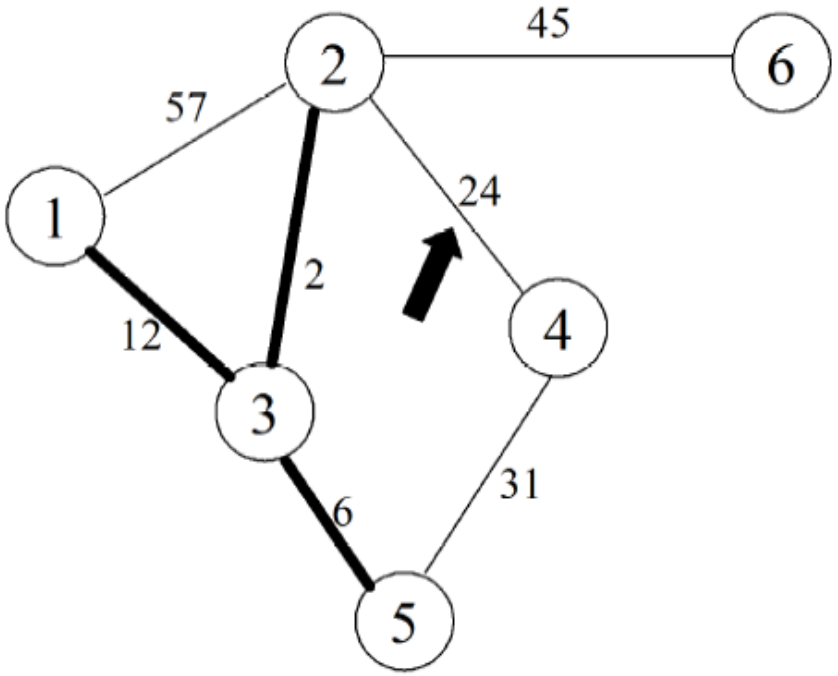
Algoritmo di Kruskal (7)

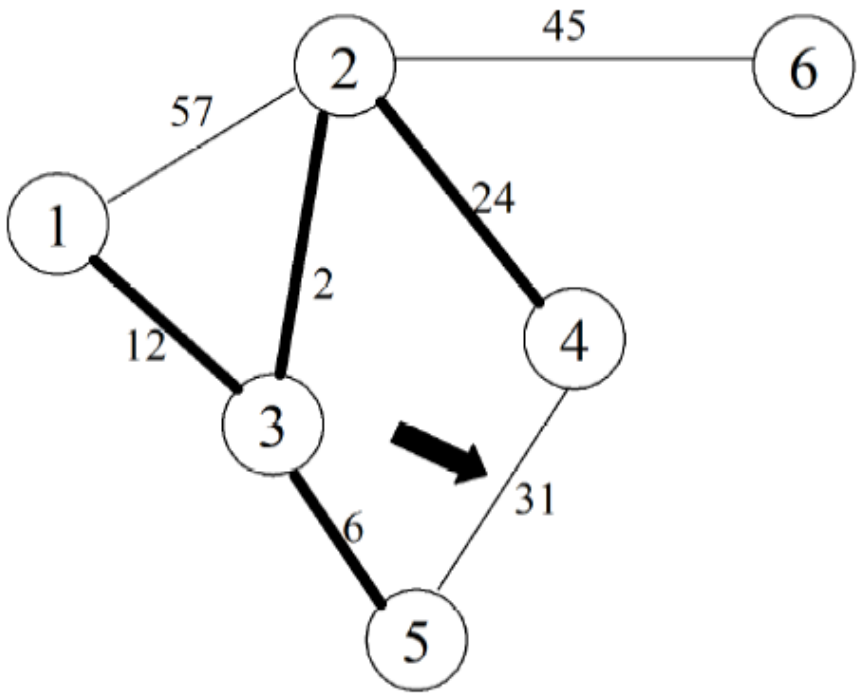
(2,3), (3,5), (1,3), (2,4), (4,5), (2,6), (1,2)

Esempio

- IV iterazione: aggiungo il lato (2,4)

COMP = [1, 1, 1, 4, 1, 6] => COMP = [1, 1, 1, 1, 1, 6]





dariano,pacciarelli@dia.uniroma3

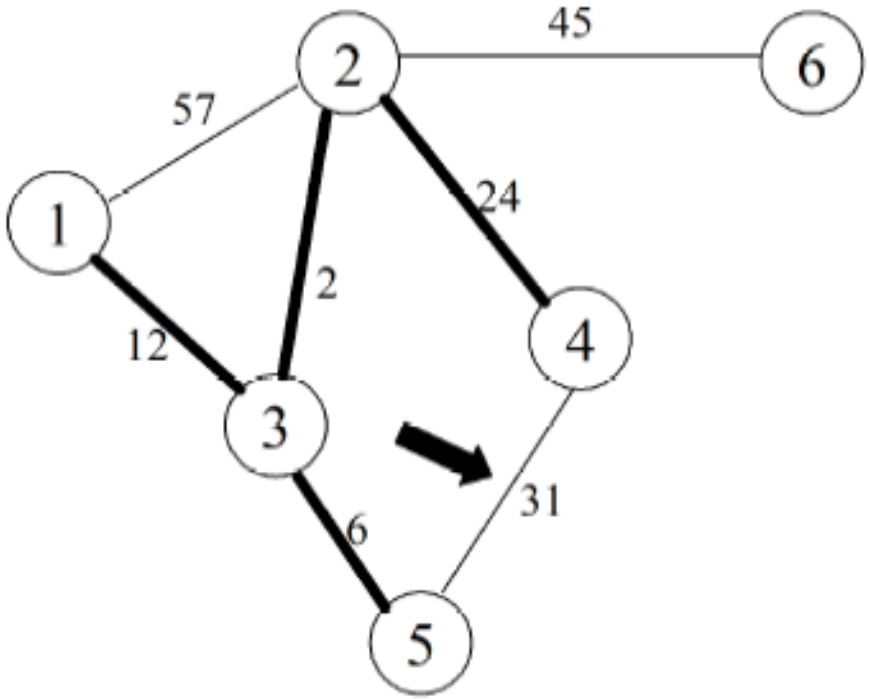
Algoritmo di Kruskal (8)

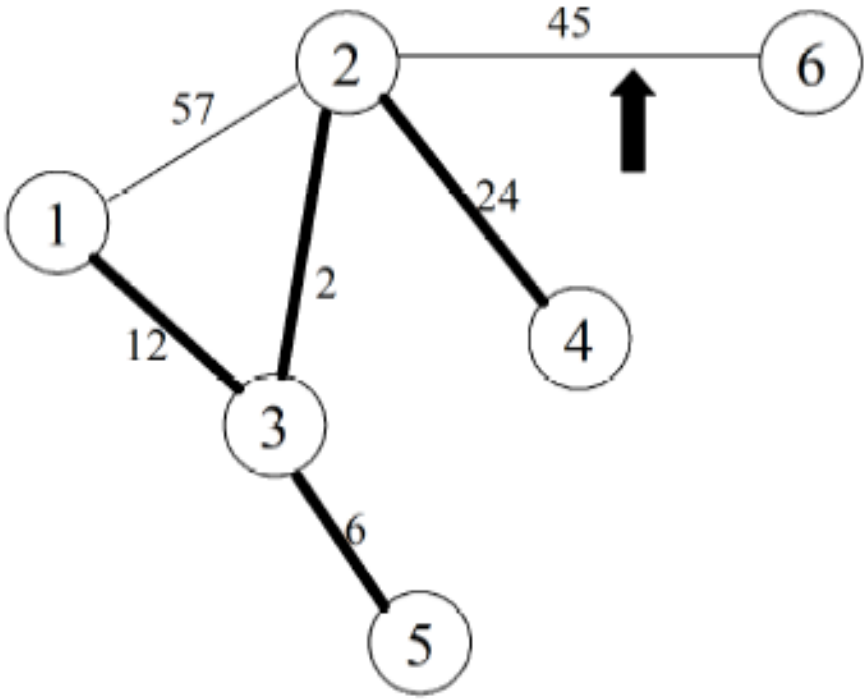
(2,3), (3,5), (1,3), (2,4), (4,5), (2,6), (1,2)

Esempio

- V iterazione: scarto il lato (4,5) perché genera il ciclo 24532

COMP = [1, 1, 1, 1, 1, 6]





dariano,pacciarelli@dia.uniroma3

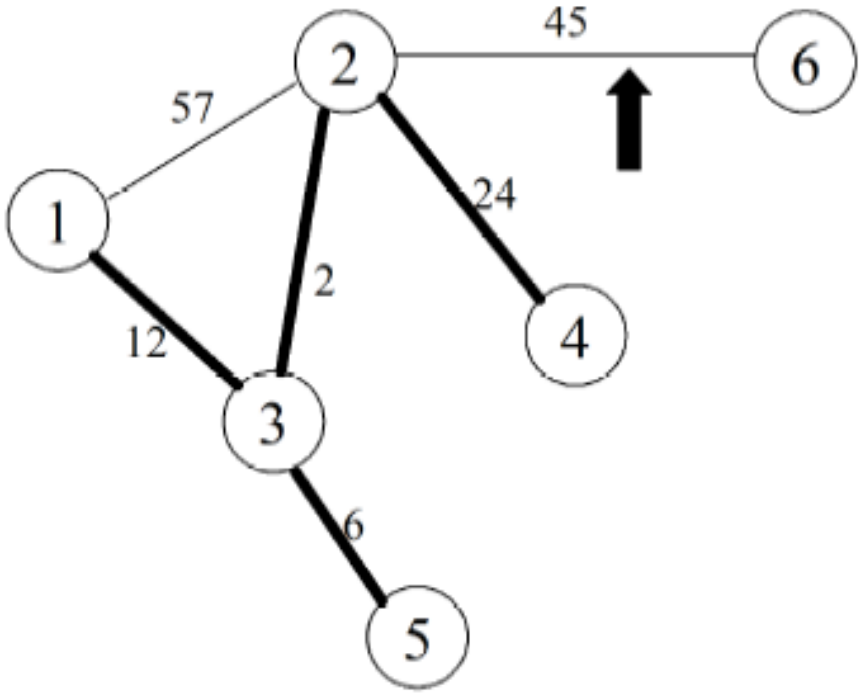
Algoritmo di Kruskal (9)

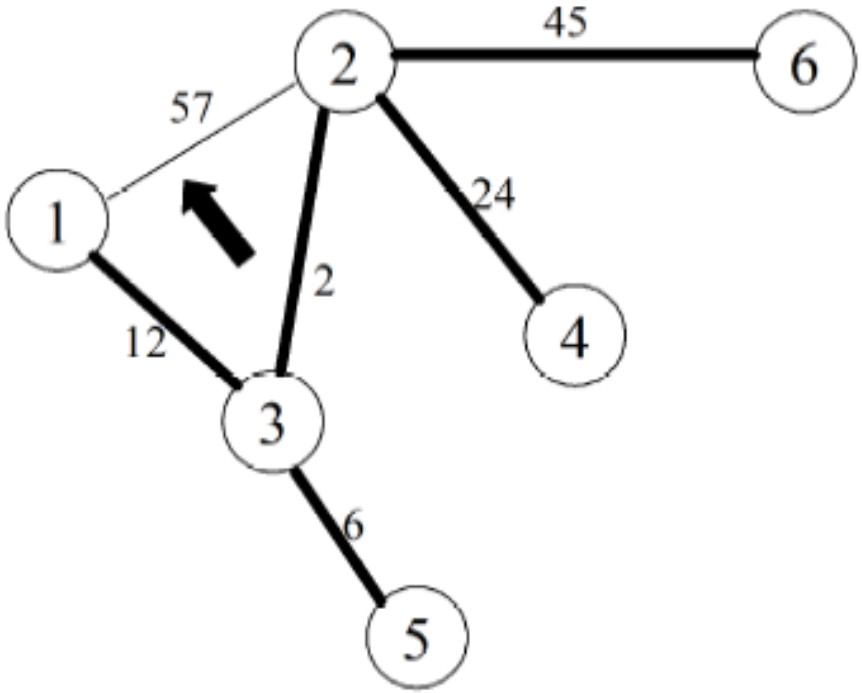
(2,3), (3,5), (1,3), (2,4), (4,5), (2,6), (1,2)

Esempio

- VI iterazione: aggiungo il lato (2,6)

COMP = [1, 1, 1, 1, 1, 6] => COMP = [1, 1, 1, 1, 1, 1]





dariano,pacciarelli@dia.uniroma3

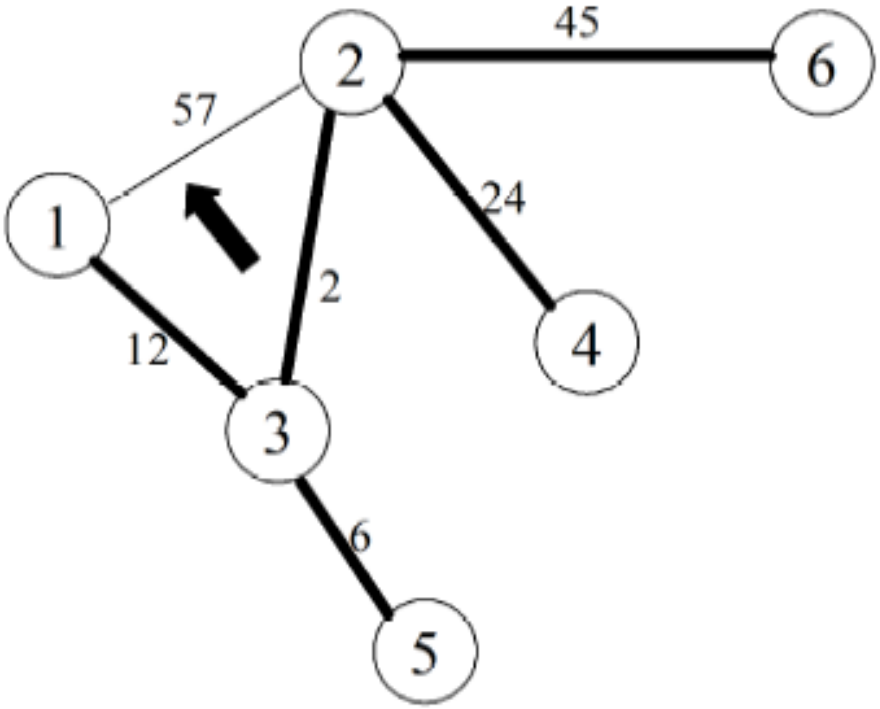
Algoritmo di Kruskal (10)

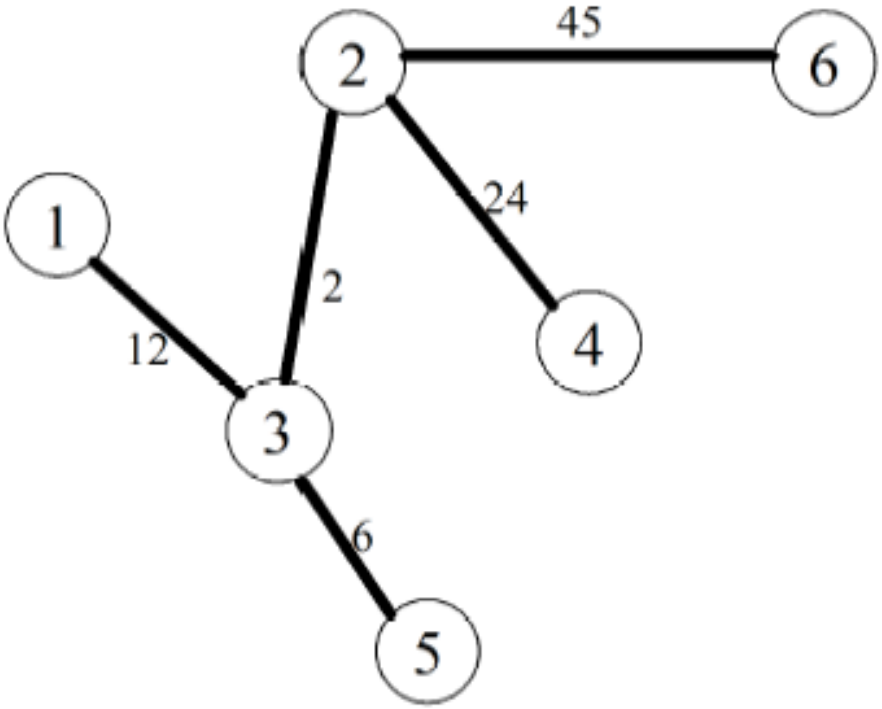
(2,3), (3,5), (1,3), (2,4), (4,5), (2,6), (1,2)

Esempio

- VII iterazione: scarto il lato (1,2) perché genera il ciclo 2312

COMP = [1, 1, 1, 1, 1, 1]





dariano,pacciarelli@dia.uniroma3

Algoritmo di Kruskal (11)

Il numero di operazioni per un'esecuzione dell'algoritmo =
numero di operazioni per ordinare i lati +
numero di operazioni di aggiornamento * num. aggiornamenti

- Ogni fase di aggiornamento viene eseguita al più $n-1$ volte
- Ogni esecuzione richiede al più n aggiornamenti del vettore COMP
- Ogni fase di aggiornamento del vettore LATI richiede:
  $O(m \log m) <= O(m \log n^2) = O(2m \log n) = O(m \log n)$

Da cui il numero di operazioni per una esecuzione dell'algoritmo di Kruskal è limitato superiormente da $\mathbf{O(m \log n + n^2)}$ operazioni

Con strutture dati specializzate è possibile ridurre la complessità teorica dell'algoritmo di Kruskal fino a $O(m \log n)$ operazioni

Algoritmo di Kruskal (12)

Analisi della complessità:

Sort degli *m* elementi nel vettore LATI: O(*m* log *n*)

Fase di aggiornamento (ciclo while) viene eseguita al più *n*-1 volte

Fase di esecuzione richiede al più *n* aggiornamenti del vettore COMP

O(*m* log *n* + n^2)

Algoritmo di Kruskal

```
//Inizializzazione
COMP[i] = i, ∀i ∈ V;  (una COMP per ogni vertice)
Ordina tutti i lati in LATI ;  (per costi crescenti)
i = 0;
T = ∅;
// Ciclo Principale
// Fino a che non ho aggiunto n − 1 lati
while ( |T| < n − 1 )
{
    // Considera il primo lato della lista
    e = LATI [i];
    // Se il lato unisce due componenti separate...
    if ( COMP[e.from] != COMP[e.to] )
    then
    // ...aggiorna i dati
    {
        fondi i due elementi di COMP;
        T = T ∪ e;
    }
    i++;
}
```

Algoritmo di Prim - Dijkstra (1)

L'algoritmo di Prim–Dijkstra, invece, sfrutta le condizioni di ottimalità sui tagli per generare l'albero ricoprente a costo minimo

Passo iniziale: Albero ricoprente vuoto

Ad ogni iterazione:

1) si considera un taglio fondamentale differente
2) si cerca il lato che minimizza il peso tra i lati appartenenti al taglio corrente
3) si aggiorna il taglio corrente tramite il lato selezionato
4) si inserisce il lato selezionato nell'albero ricoprente

L'iterazione viene ripetuta finché non si è costruito l'intero albero

Algoritmo di Prim - Dijkstra (2)

Una possibile Implementazione:

L'algoritmo fa uso di una struttura *S* che contiene i vertici che definiscono il taglio corrente

A fine esecuzione, una lista di archi *T* conterrà un albero ricoprente minimo

Algoritmo di Prim-Dijkstra (Ver. $O(n^3)$)

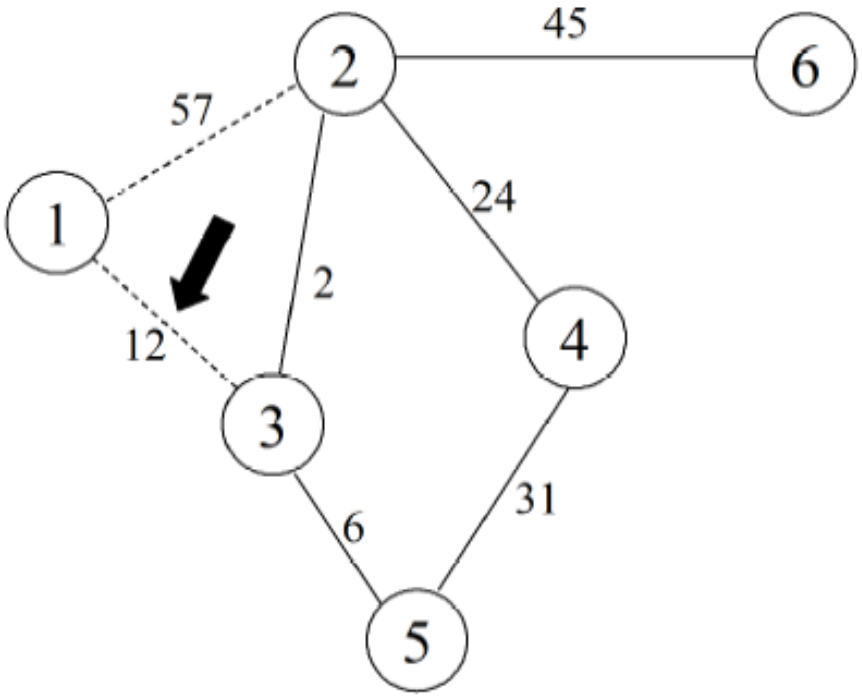
// Inzializzazione
$T = \emptyset$;
$S = \{1\}$; (diversi alberi con lo stesso peso per diverse *S*)
// Ciclo principale
// Fino a che non ho aggiunto $n-1$ lati
while $(|T| < n-1)$
{
    // Trova il lato di costo minimo nel taglio
    $(i \in S, j \in \bar{S}) = \arg\min\{c_h, h \in [S, \bar{S}]\}$;
    // Aggiunge (i, j) all'albero e j al taglio
    $T = T \cup (i, j)$;
    $S = S \cup j$;
}

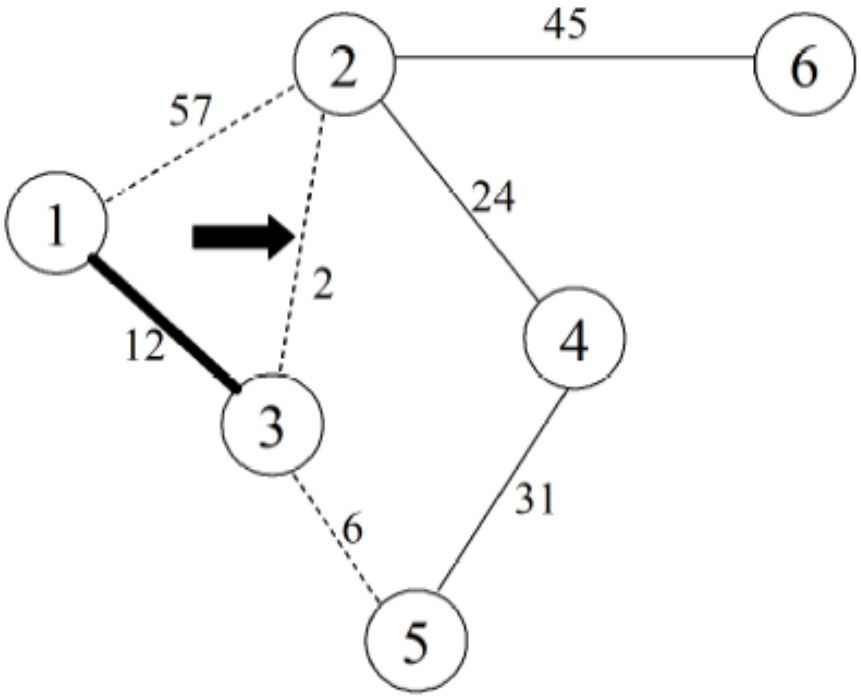
dariano,pacciarelli@dia.uniroma3

Algoritmo di Prim - Dijkstra (3)

Esempio

- Si sceglie il vertice iniziale a caso, ad esempio il vertice 1
- Si considera quindi il taglio generato dall'insieme $S = \{1\}$, ovvero $[S, \underline{S}] = \{(1, 2), (1, 3)\}$
- Il lato (1, 3) con $c_{13} = 12$ minimizza il costo dei lati del taglio corrente, per cui il vertice 3 viene aggiunto all'insieme S



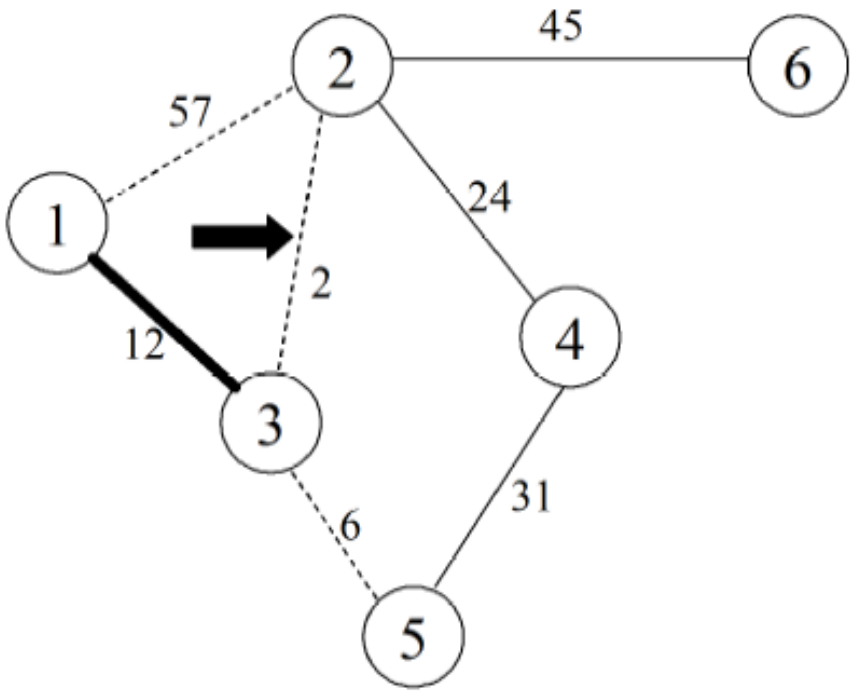


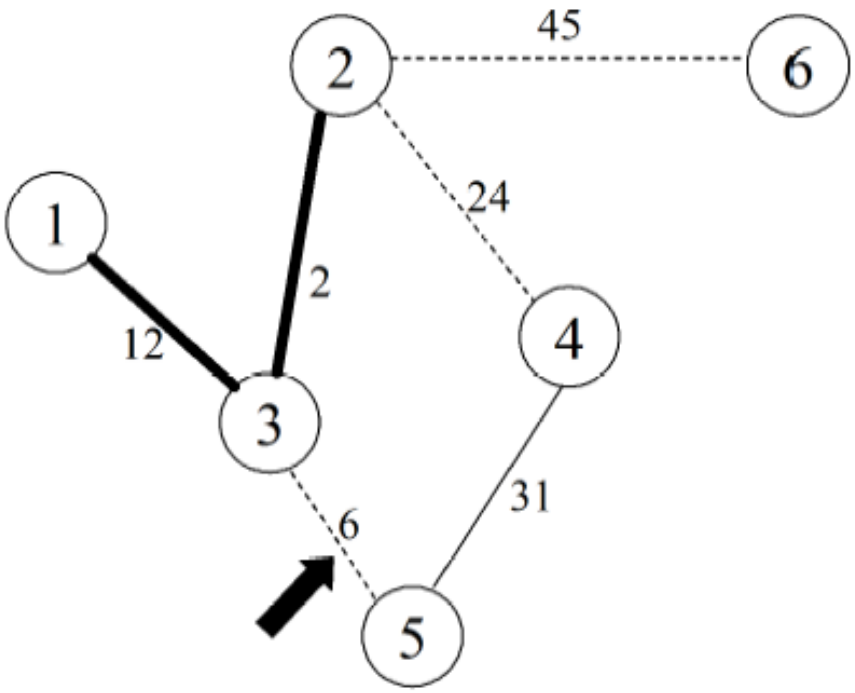
dariano,pacciarelli@dia.uniroma3

Algoritmo di Prim - Dijkstra (4)

Esempio

- Alla seconda iterazione $S = \{1\} \cup \{3\}$
- Il lato che minimizza il costo del taglio è (2,3), e $S = \{1,3\} \cup \{2\}$
- Il lato (1,2) non appartiene più al taglio $[S, V - S]$



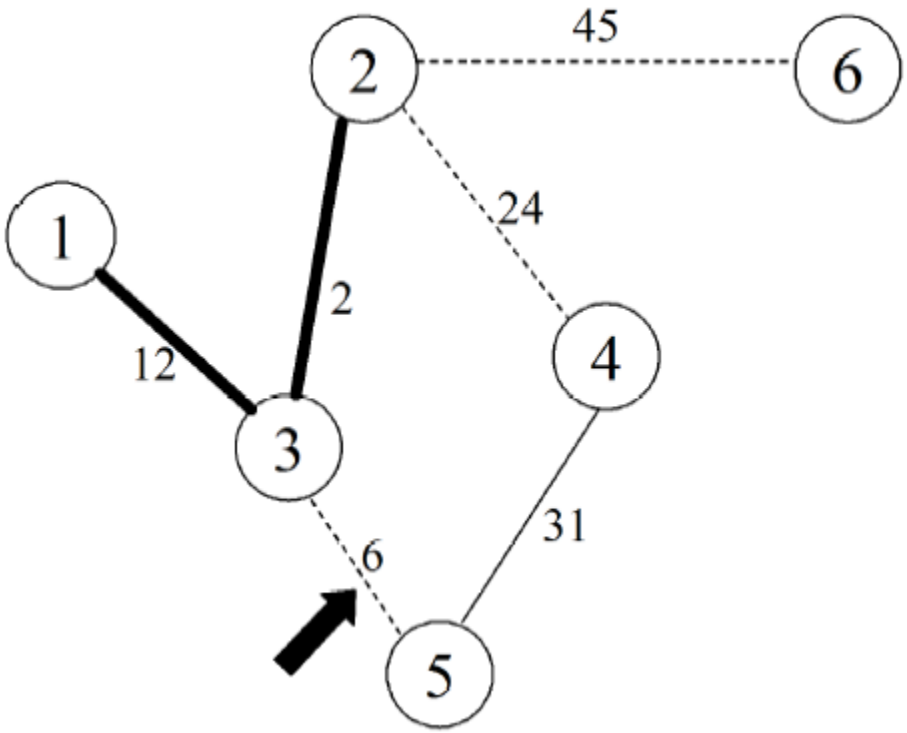


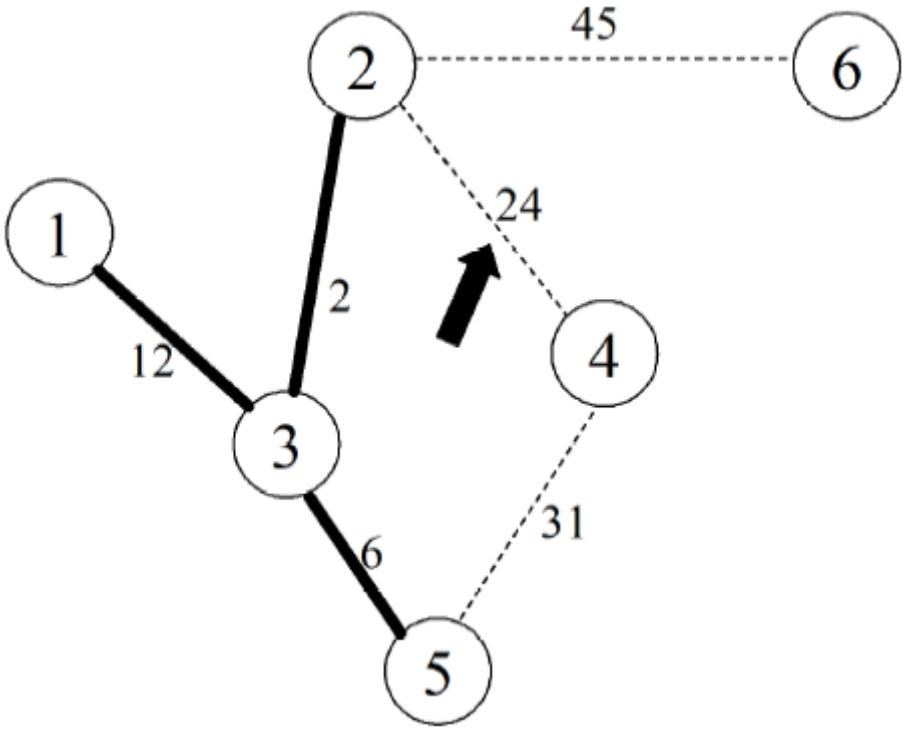
dariano,pacciarelli@dia.uniroma3

Algoritmo di Prim - Dijkstra (5)

Esempio

- Alla terza iterazione $S = \{1,3,2\}$
- Il lato che minimizza il costo del taglio è (3, 5),
  e $S = \{1,3,2\} \cup \{5\}$



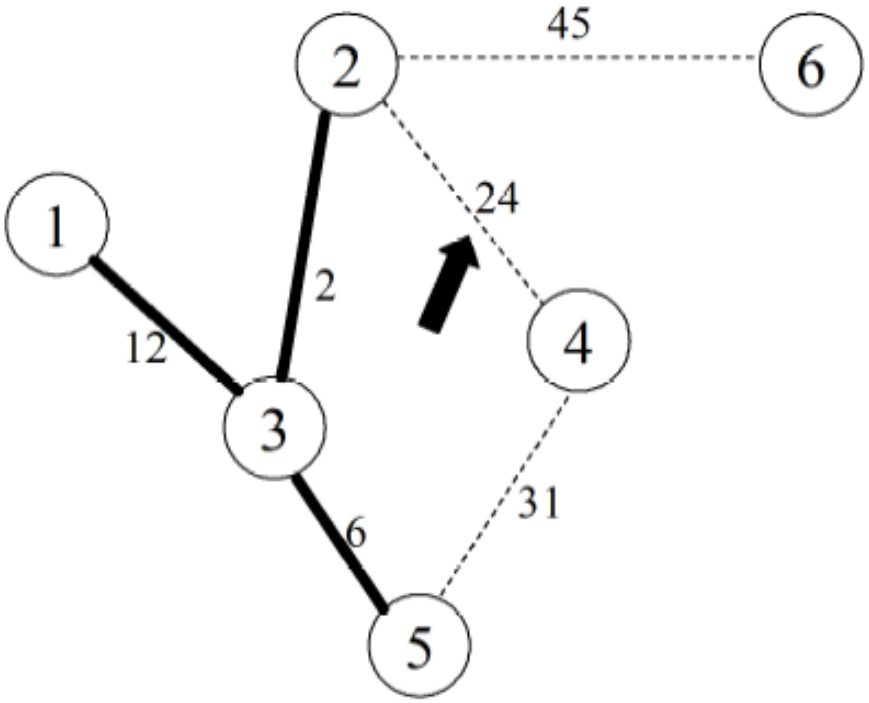


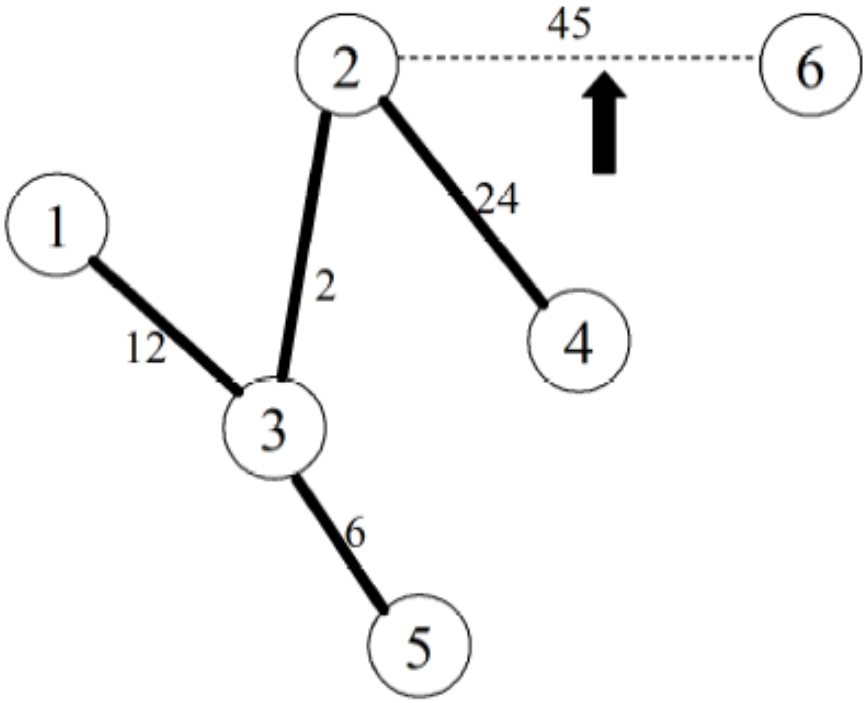
dariano,pacciarelli@dia.uniroma3

Algoritmo di Prim - Dijkstra (6)

Esempio

- Alla quarta iterazione $S = \{1,3,2,5\}$
- Il lato che minimizza il costo del taglio è (2, 4), e $S = S \cup \{4\}$
- Il lato (4,5) non appartiene più al taglio $[S, V - S]$

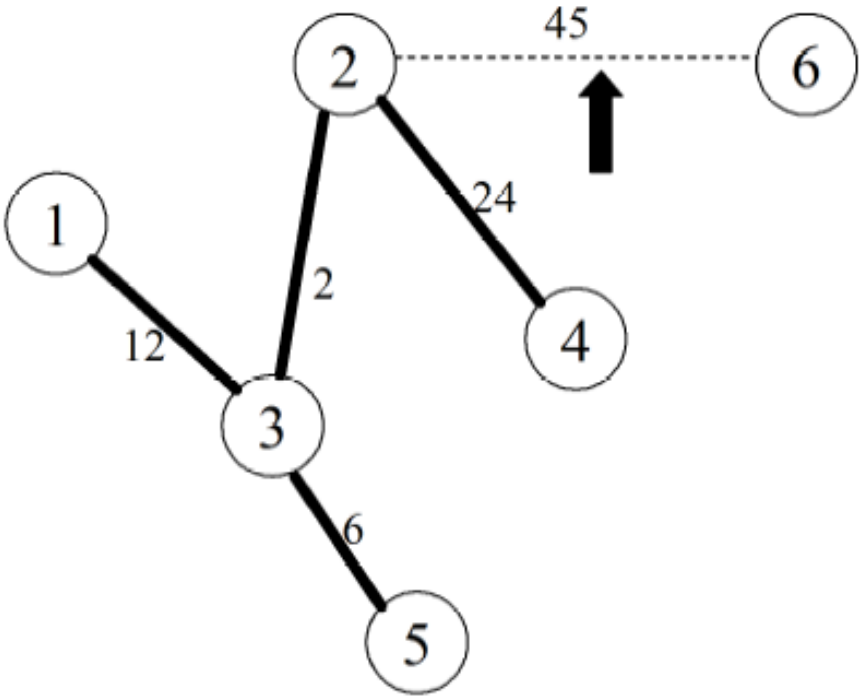


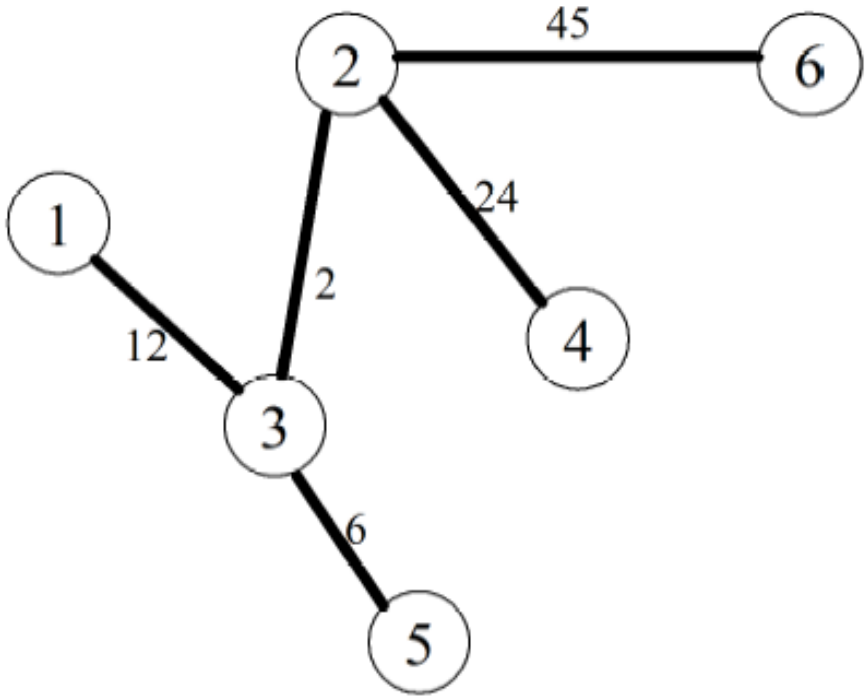


Algoritmo di Prim - Dijkstra (7)

Esempio

- Alla quinta iterazione S = {1,3,2,5,4}
- Il lato che minimizza il costo del taglio è (2, 6), e $S = S$ U {6}





Algoritmo di Prim - Dijkstra (8)

- Il numero di operazioni svolte dall'algoritmo dipende da come ad ogni passo si individua il lato di peso minimo dal taglio [S, $\underline{S}$]
- L'algoritmo esegue al più $n-1$ iterazioni, infatti ad ogni iterazione l'algoritmo aggiunge un vertice e un lato all'albero ricoprente
- La versione vista dell'algoritmo richiede m operazioni di confronto, infatti è necessario scorrere tutti i lati del grafo (m) per stabilire se ciascun lato appartiene o meno all'insieme [S, $\underline{S}$], e se vi appartiene stabilire se è quello di costo minimo o meno.
- La complessità, intesa come il maggior numero di operazioni per una esecuzione dell'algoritmo di Prim–Dijkstra, sarà dunque non superiore a $O(mn)$ operazioni. Dato che $m << n^2$ si ha $\boldsymbol{O(n^3)}$

dariano,pacciarelli@dia.uniroma3

Algoritmo di Prim - Dijkstra (9)

Analisi della complessità:

Fase di aggiornamento (ciclo while) viene eseguita al più *n*-1 volte

Fase di esecuzione richiede di scorrere tutti i lati del grafo (*m*)

Dato che $m << n^2$ (un vertice ha al più *n*-1 lati adiacenti, non ci sono anelli o lati multipli) si ha $\boldsymbol{O(n^3)}$

Algoritmo di Prim-Dijkstra (Ver. $O(n^3)$)

```
// Inzializzazione
T = ∅;
S = {1};(diversi alberi con lo stesso peso per diverse S)
// Ciclo principale
// Fino a che non ho aggiunto n − 1  lati
while (|T| < n − 1)
{
    //  Trova il lato di  costo minimo nel taglio
    (i ∈ S, j ∈ S) = arg min{c_h, h ∈ [S, S̄]};
    // Aggiunge (i, j) all'albero e j al taglio
    T = T ∪ (i, j);
    S = S ∪ j;
}
```


dariano,pacciarelli@dia.uniroma3

Algoritmo di Prim - Dijkstra (10)

Concetto chiave: L'operazione collo di bottiglia nell'algoritmo è quella di individuare il lato a minor costo nel taglio [S, $\underline{S}$]

Osservazione rilevante: Tra una iterazione e la successiva, [S, $\underline{S}$] non deve essere necessariamente ricalcolato per intero, infatti molti lati rimangono gli stessi

Speed-up dell'algoritmo: Struttura dati specializzata che mantiene memorizzati i dati di [S, $\underline{S}$] per riusarli nell'iterazione successiva

Algoritmo di Prim - Dijkstra (11)

Nuova struttura dati:

- un vettore booleano di flag (V) che conserva l'informazione se il vertice i appartiene o meno all'insieme S
- un vettore dei costi (C) che mantiene aggiornato, per ogni vertice in $\underline{S}$, il costo minimo necessario per congiungere il vertice all'insieme S

  { Il vettore dei costi C sarà inizializzato con i valori dei lati della stella del vertice radice }
- la matrice H di adiacenza ($n \times n$) che per ogni coppia di vertici i e j rappresenta il costo per congiungerli $h[i, j]$
- { Nel caso in cui il lato da i a j non sia presente si assume $h[i, j]$ = + infinito, altrimenti si avrà $h[i, j] = c_{ij}$ }

Algoritmo di Prim - Dijkstra (12)

Considerazioni sul vettore dei costi:

Il vettore *C* rappresenta una stima del costo minimo necessario per congiungere ogni vertice in *S* all'insieme *S*

Questo costo viene inizializzato con + infinito
se non esiste un lato tra il vertice radice e il generico vertice,
altrimenti viene inizializzato con il peso del lato

Durante l'esecuzione dell'algoritmo questi costi vengono diminuiti non appena il vertice diviene raggiungibile, ovvero il vertice appartiene al taglio corrente

Se il costo rimane + infinito, il vertice non è ancora raggiungibile

Algoritmo di Prim - Dijkstra (13)

Pseudo-codice per l'impl. efficiente:

V = vettore booleano di flag

C = vettore dei costi

T = lista dei lati dell'albero ricoprente

H = matrice di adiacenza

Algoritmo di Prim–Dijkstra (Ver. $O(n^2)$)

```
// Inizializzazione
T = ∅;
V[i] = 0, ∀i;
V[1] = 1:
C[i] = H[1, i], ∀i;
// Ciclo principale
while (|T| < n − 1)
{
    Trova il vertice k che minimizza i costi in C∩!V;
    // aggiornamento...
    V[k] = 1;
    T = T ∪ (x, k):
    // Aggiorna C con i lati adiacenti k
    for ( j = 1; j ≤ n; j++ )
    {
        if ( V[j] == 0 )&&( H[k, j] < C[j] )
        {
            C[j] = H[k, j];
        }
    }
}
```

Algoritmo di Prim - Dijkstra (14)

Analisi della complessità:

Fase di aggiornamento (ciclo while) viene eseguita al più *n*-1 volte

Fase di esecuzione richiede al più *n*-1 operazioni per trovare il vertice *k*

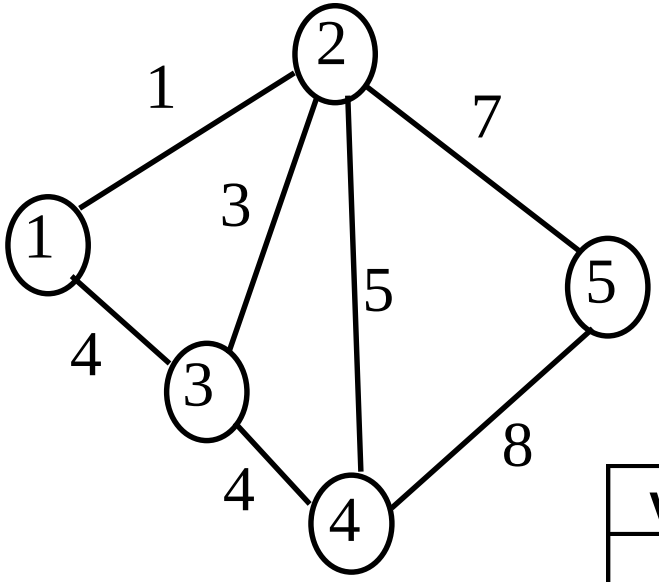
(nella versione precedente dell'algoritmo servivano *m* operazioni per trovare il vertice di costo minimo nel taglio corrente)

Algoritmo di Prim–Dijkstra (Ver. $O(n^2)$)

```
// Inizializzazione
T = ∅;
V[i] = 0, ∀i;
V[1] = 1:
C[i] = H[1, i], ∀i;
// Ciclo principale
while (|T| < n − 1)
{
    Trova il vertice k che minimizza i costi in C∩!V;
    // aggiornamento...
    V[k] = 1;
    T = T ∪ (x, k):
    // Aggiorna C con i lati adiacenti k
    for ( j = 1; j ≤ n; j++ )
    {
        if ( V[j] == 0 )&&( H[k, j] < C[j] )
        {
            C[j] = H[k, j];
        }
    }
}
```

Algoritmo di Prim - Dijkstra (15)

Esempio con implementazione efficiente:

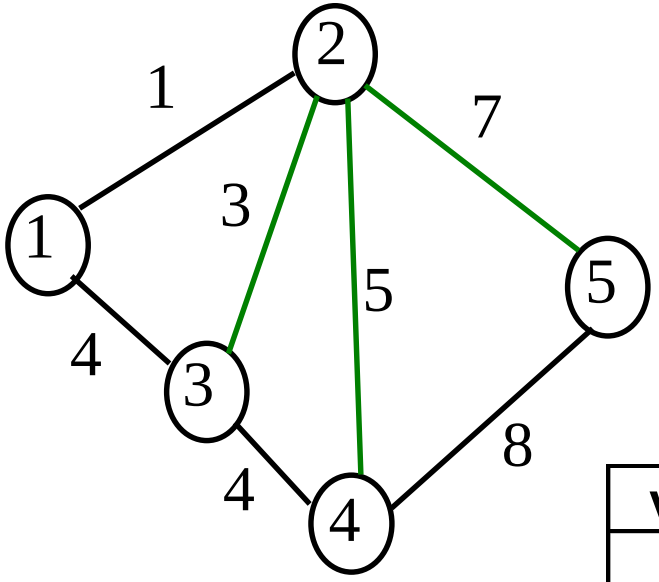


S = {1}

vertice	flag (V)	costo (C)	predecessore
1	1	0	nessuno
2	0	1	1
3	0	4	1
4	0	inf	1
5	0	inf	1

Algoritmo di Prim - Dijkstra (16)

Esempio con implementazione efficiente:

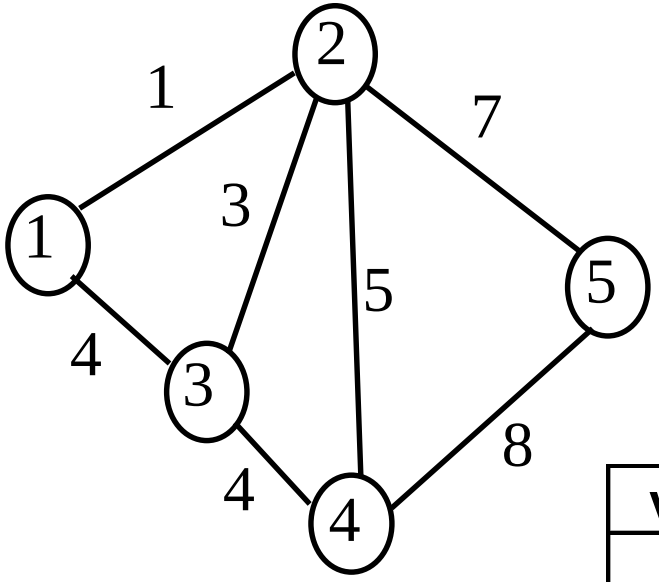


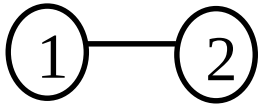
S = {1}

vertice	flag (V)	costo (C)	predecessore
1	1	0	nessuno
2	0	1	1
3	0	4	1
4	0	inf	1
5	0	inf	1

Algoritmo di Prim - Dijkstra (17)

Esempio con implementazione efficiente:





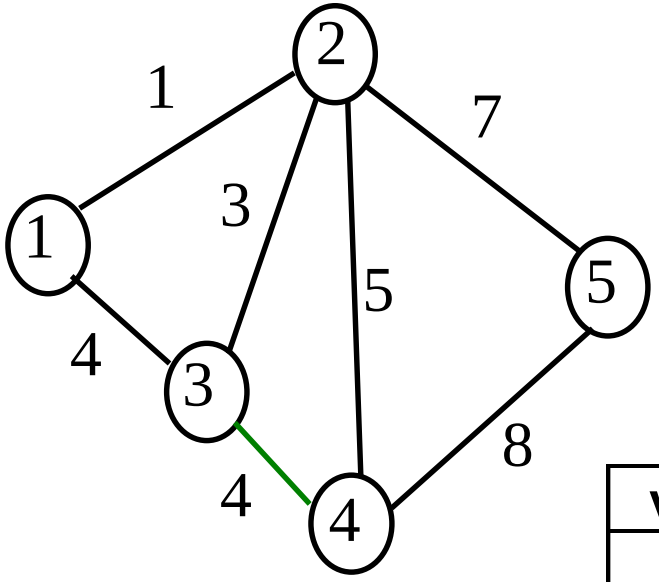
S = {1,2}

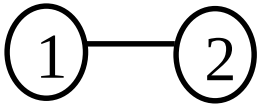
vertice	flag (V)	costo (C)	predecessore
1	1	0	nessuno
2	1	1	1
3	0	3	2
4	0	5	2
5	0	7	2

dariano,pacciarelli@dia.uniroma3

Algoritmo di Prim - Dijkstra (18)

Esempio con implementazione efficiente:



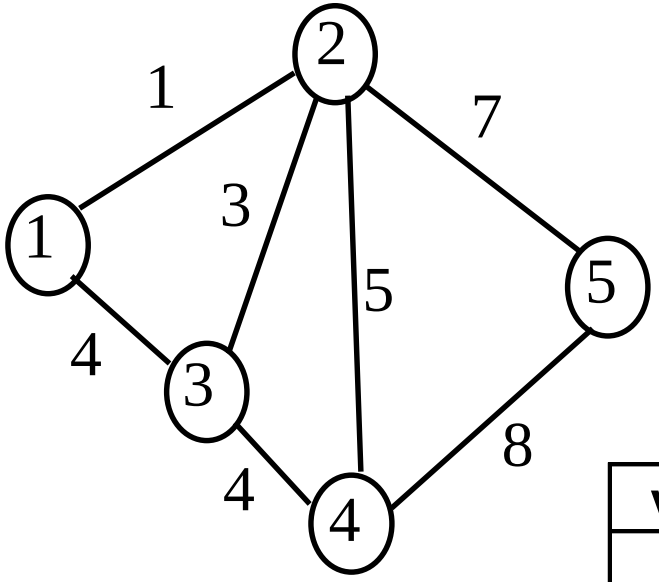


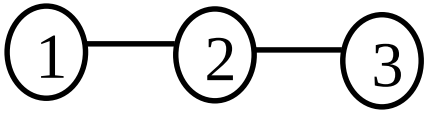
S = {1,2}

vertice	flag (V)	costo (C)	predecessore
1	1	0	nessuno
2	1	1	1
3	0	3	2
4	0	5	2
5	0	7	2

Algoritmo di Prim - Dijkstra (19)

Esempio con implementazione efficiente:



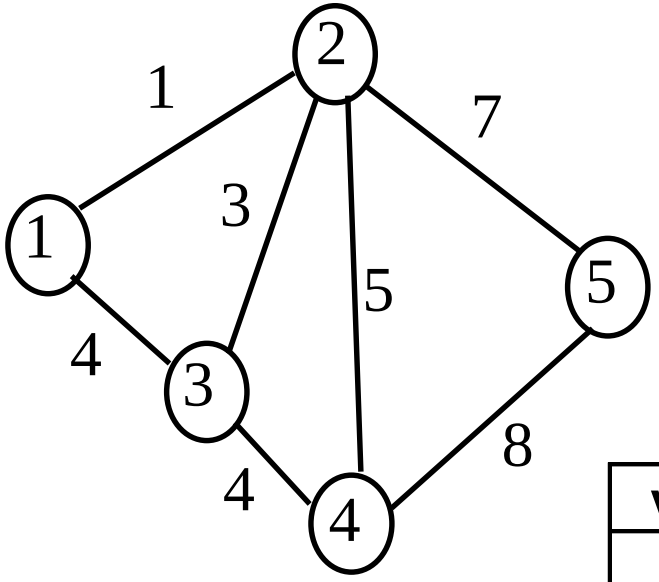


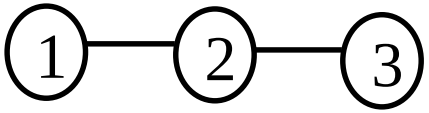
S = {1,2,3}

vertice	flag (V)	costo (C)	predecessore
1	1	0	nessuno
2	1	1	1
3	1	3	2
4	0	4	3
5	0	7	2

Algoritmo di Prim - Dijkstra (20)

Esempio con implementazione efficiente:





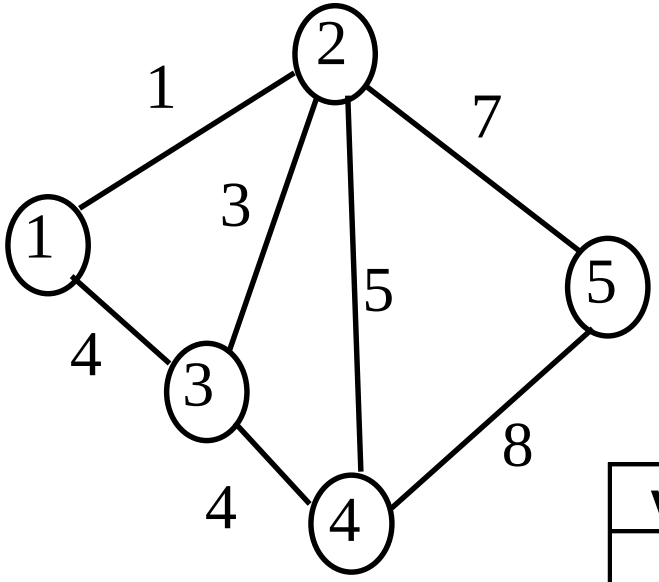
S = {1,2,3}

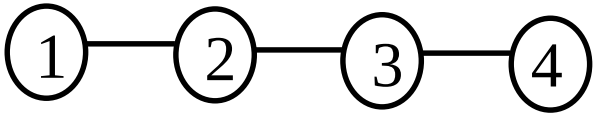
vertice	flag (V)	costo (C)	predecessore
1	1	0	nessuno
2	1	1	1
3	1	3	2
4	0	4	3
5	0	7	2

dariano,pacciarelli@dia.uniroma3

Algoritmo di Prim - Dijkstra (21)

Esempio con implementazione efficiente:





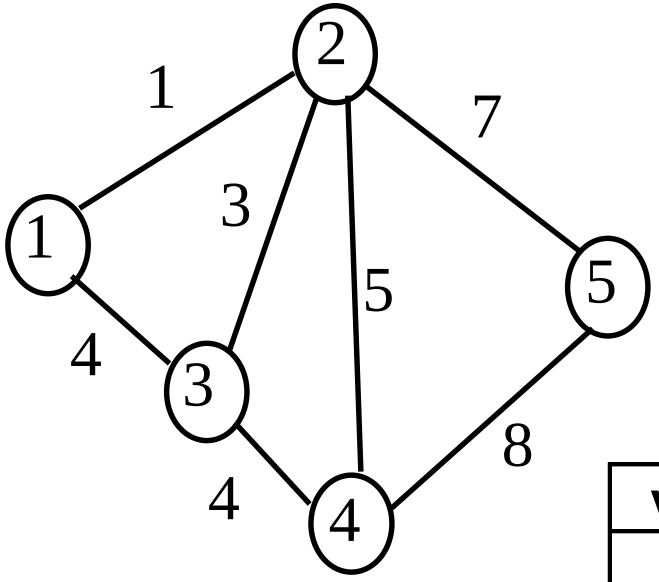
S = {1,2,3,4}

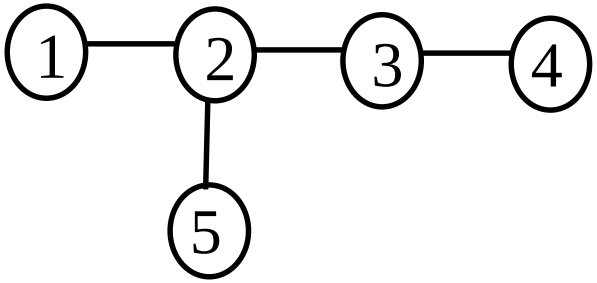
vertice	flag (V)	costo (C)	predecessore
1	1	0	nessuno
2	1	1	1
3	1	3	2
4	1	4	3
5	0	7	2

dariano,pacciarelli@dia.uniroma3

Algoritmo di Prim - Dijkstra (22)

Esempio con implementazione efficiente:



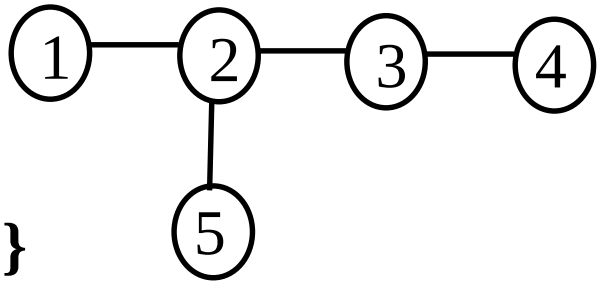


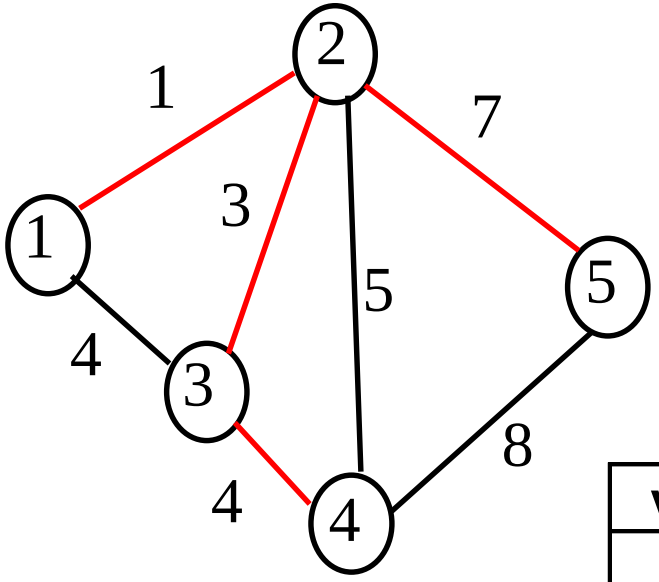
S = {1,2,3,4,5}

vertice	flag (V)	costo (C)	predecessore
1	1	0	nessuno
2	1	1	1
3	1	3	2
4	1	4	3
5	1	7	2

Algoritmo di Prim - Dijkstra (23)

Esempio con implementazione efficiente:





S = {1, 2, 3, 4, 5}

Costo albero = 15

I vertici del grafo vengono fissati ad 1 nell'ordine 1, 2, 3, 4, 5.

vertice	flag (V)	costo (C)	predecessore
1	1	0	nessuno
2	1	1	1
3	1	3	2
4	1	4	3
5	1	7	2

Esercizio su Algo Prim – Dijkstra (1)

La tabella mostra la matrice di incidenza vertici/lati (righe/colonne) di un grafo (non orientato). La prima e l'ultima riga indicano, rispettivamente, i nomi dei lati e i loro pesi.

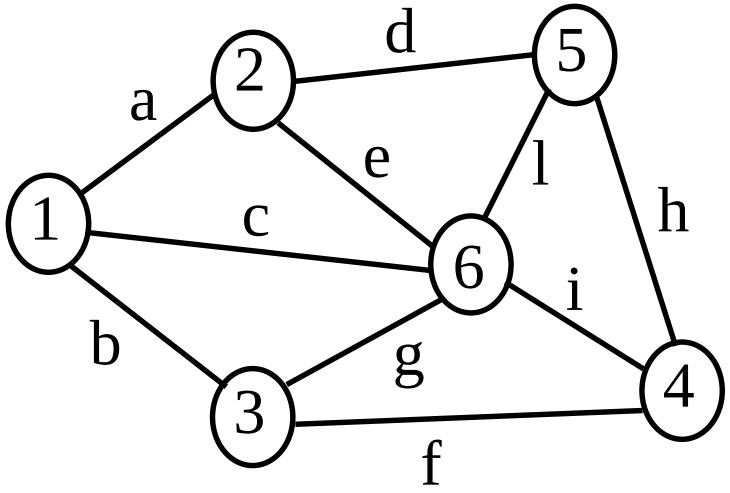
	a	*b*	*c*	*d*	*e*	*f*	*g*	*h*	*i*	*l*
1	1	1	1	0	0	0	0	0	0	0
2	1	0	0	1	1	0	0	0	0	0
3	0	1	0	0	0	1	1	0	0	0
4	0	0	0	0	0	1	0	1	1	0
5	0	0	0	1	0	0	0	1	0	1
6	0	0	1	0	1	0	1	0	1	1
Costi	7	5	10	5	3	4	6	10	12	2

Trovare l'albero ricoprente di peso minimo, a partire dal vertice **1.**

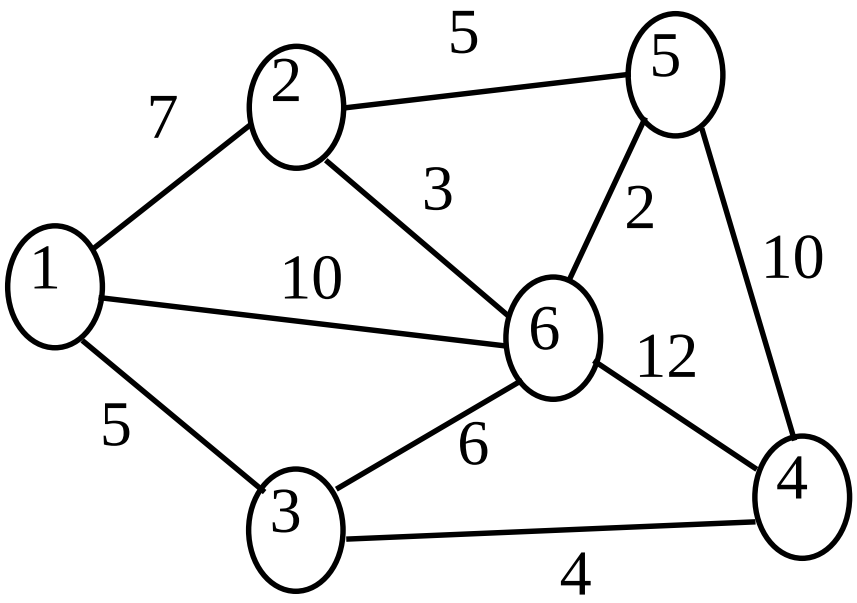
Indicare in quale ordine vengono fissati ad 1 i flag dei vertici del grafo

Esercizio su Algo Prim – Dijkstra (2)

	a	*b*	*c*	*d*	*e*	*f*	*g*	*h*	*i*	*l*
1	1	1	1	0	0	0	0	0	0	0
2	1	0	0	1	1	0	0	0	0	0
3	0	1	0	0	0	1	1	0	0	0
4	0	0	0	0	0	1	0	1	1	0
5	0	0	0	1	0	0	0	1	0	1
6	0	0	1	0	1	0	1	0	1	1
Costi	7	5	10	5	3	4	6	10	12	2

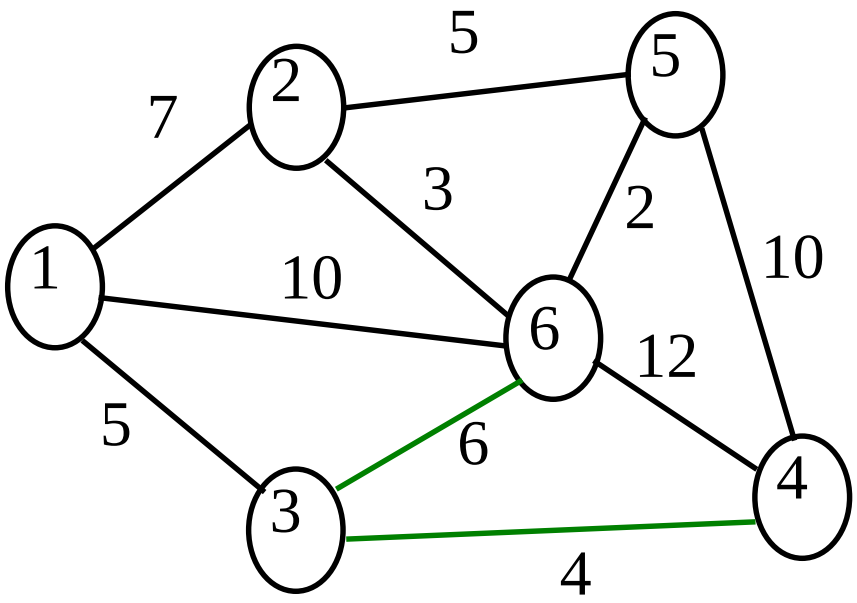


Esercizio su Algo Prim – Dijkstra (3)



vertice	flag (V)	costo (C)	predecessore
1	1	0	nessuno
2	0	7	1
3	0	5	1
4	0	inf	1
5	0	inf	1
6	0	10	1

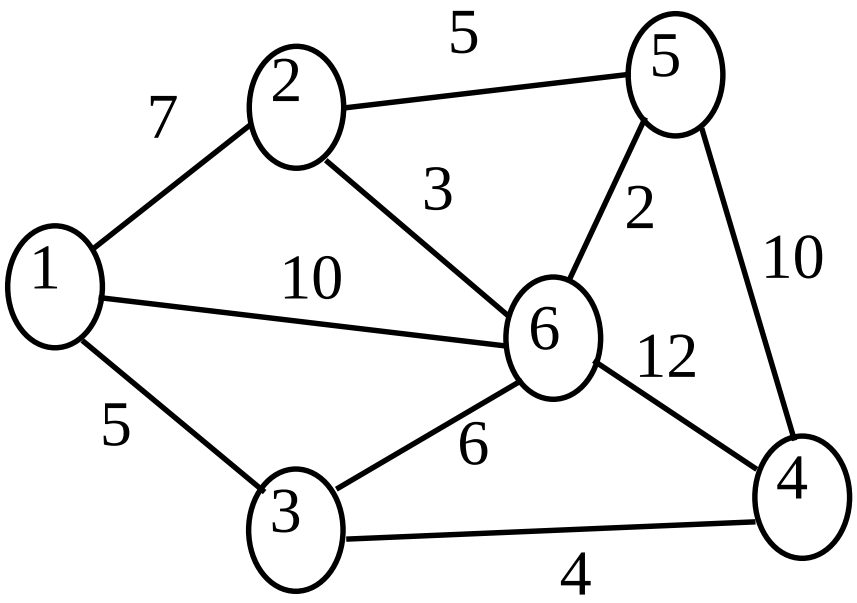
Esercizio su Algo Prim – Dijkstra (4)

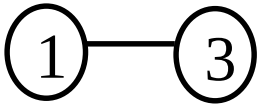


vertice	flag (V)	costo (C)	predecessore
1	1	0	nessuno
2	0	7	1
3	0	5	1
4	0	inf	1
5	0	inf	1
6	0	10	1

dariano,pacciarelli@dia.uniroma3

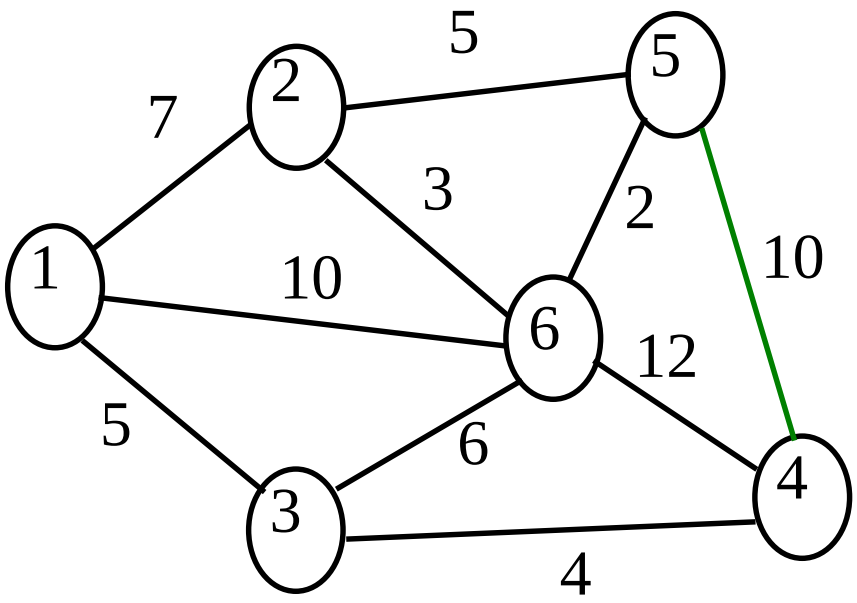
Esercizio su Algo Prim – Dijkstra (5)

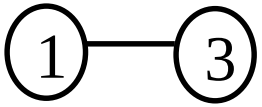




vertice	flag (V)	costo (C)	predecessore
1	1	0	nessuno
2	0	7	1
3	1	5	1
4	0	4	3
5	0	inf	1
6	0	6	3

Esercizio su Algo Prim – Dijkstra (6)

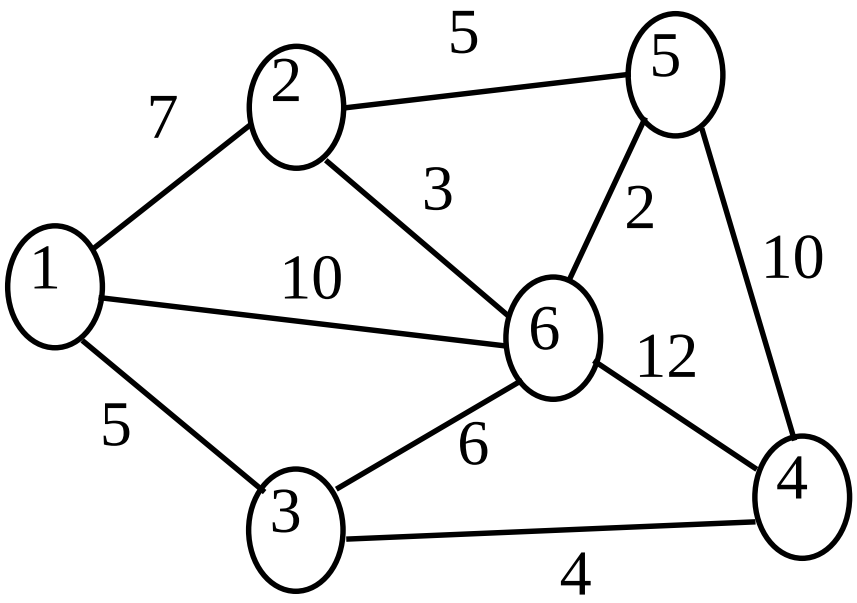


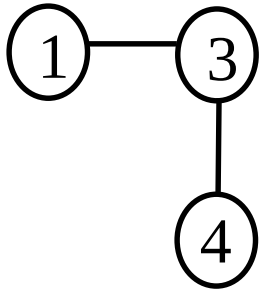


vertice	flag (V)	costo (C)	predecessore
1	1	0	nessuno
2	0	7	1
3	1	5	1
4	0	4	3
5	0	inf	1
6	0	6	3

dariano,pacciarelli@dia.uniroma3

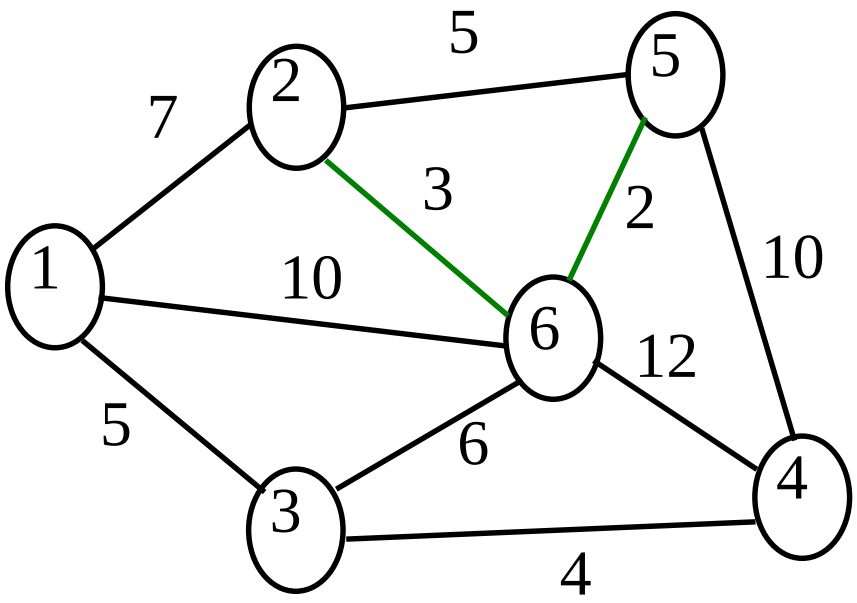
Esercizio su Algo Prim – Dijkstra (7)

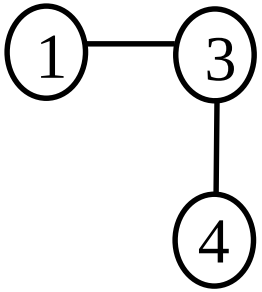




vertice	flag (V)	costo (C)	predecessore
1	1	0	nessuno
2	0	7	1
3	1	5	1
4	1	4	3
5	0	10	4
6	0	6	3

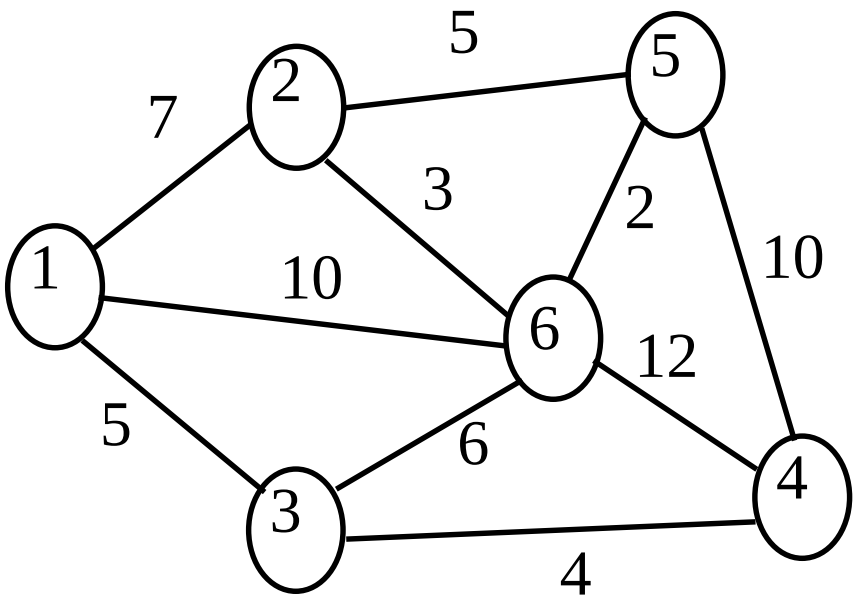
Esercizio su Algo Prim – Dijkstra (8)

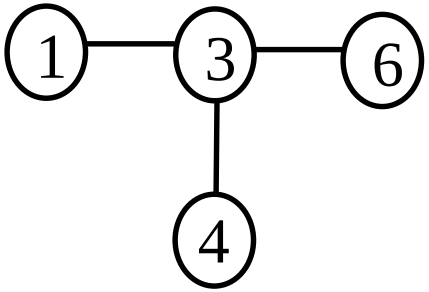




vertice	flag (V)	costo (C)	predecessore
1	1	0	nessuno
2	0	7	1
3	1	5	1
4	1	4	3
5	0	10	4
6	0	6	3

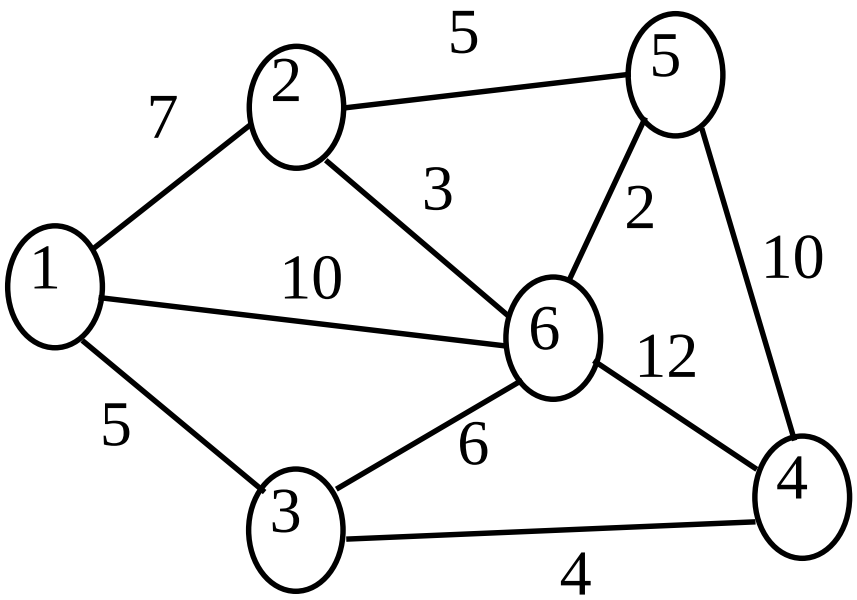
Esercizio su Algo Prim – Dijkstra (9)

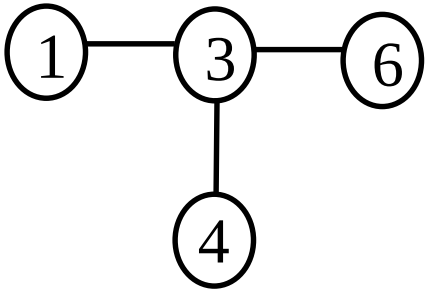




vertice	flag (V)	costo (C)	predecessore
1	1	0	nessuno
2	0	3	6
3	1	5	1
4	1	4	3
5	0	2	6
6	1	6	3

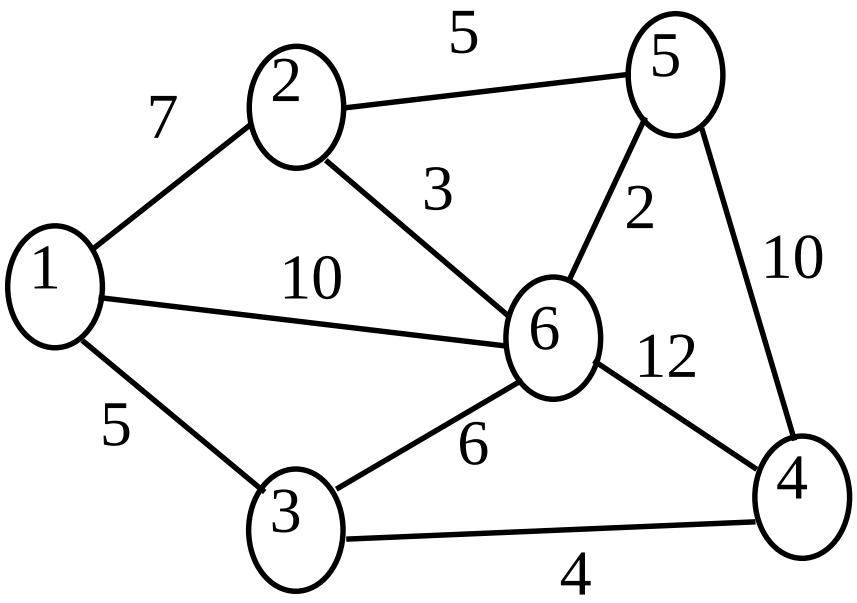
Esercizio su Algo Prim – Dijkstra (10)

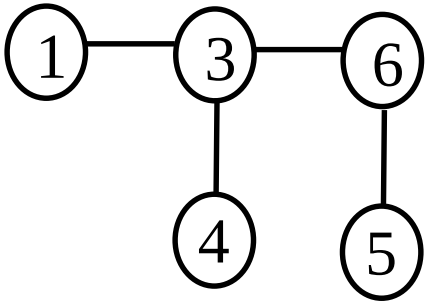




vertice	flag (V)	costo (C)	predecessore
1	1	0	nessuno
2	0	3	6
3	1	5	1
4	1	4	3
5	0	2	6
6	1	6	3

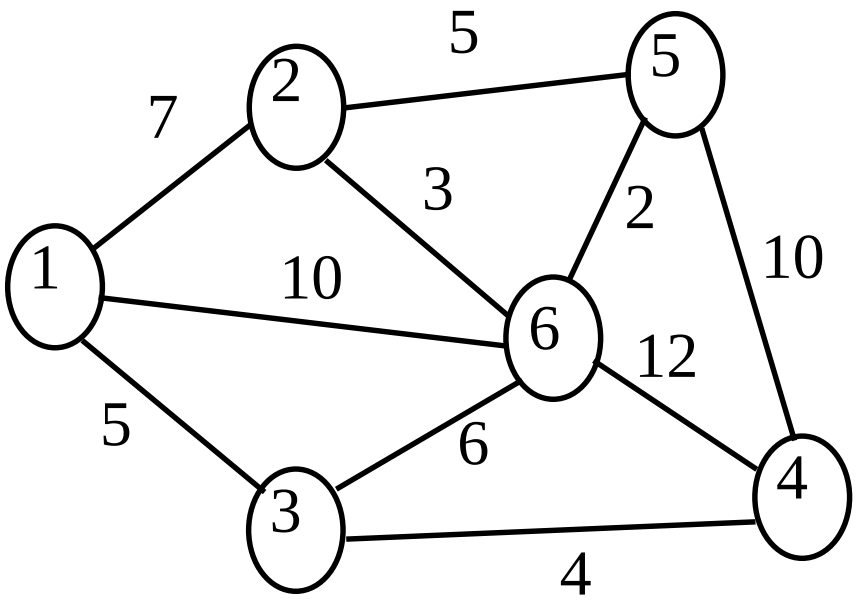
Esercizio su Algo Prim – Dijkstra (11)

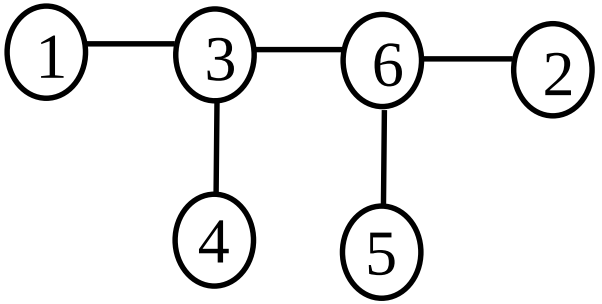




vertice	flag (V)	costo (C)	predecessore
1	1	0	nessuno
2	0	3	6
3	1	5	1
4	1	4	3
5	1	2	6
6	1	6	3

Esercizio su Algo Prim – Dijkstra (12)



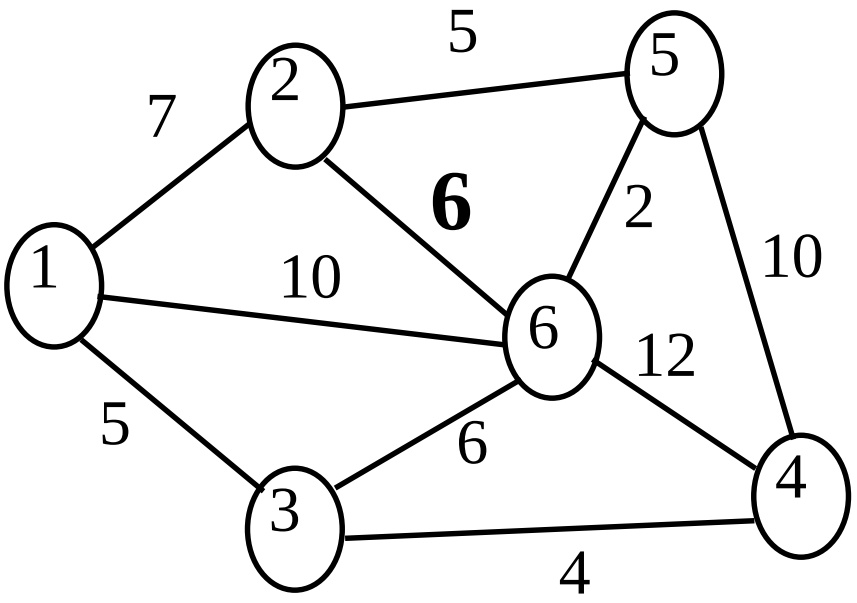


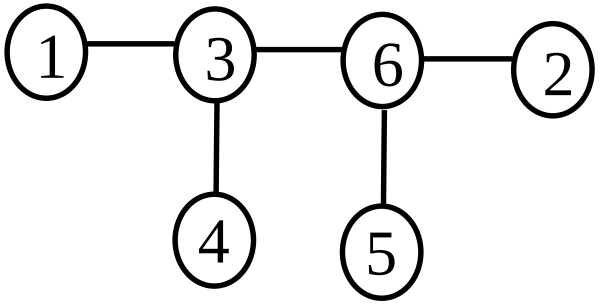
I vertici del grafo vengono fissati ad 1 nell'ordine 1, 3, 4, 6, 5, 2.

Il costo dell'albero è 20

vertice	flag (V)	costo (C)	predecessore
1	1	0	nessuno
2	1	3	6
3	1	5	1
4	1	4	3
5	1	2	6
6	1	6	3

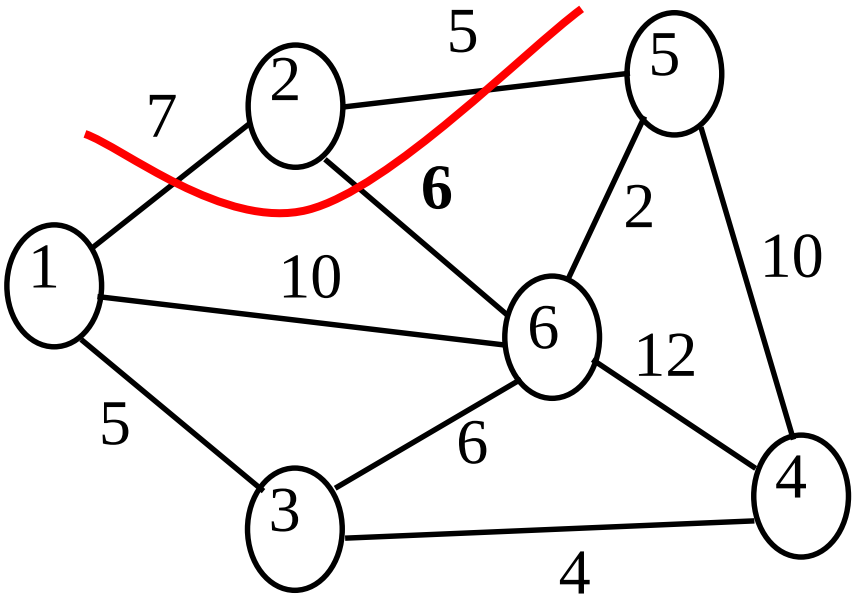
Esercizio su Algo Prim – Dijkstra (13)

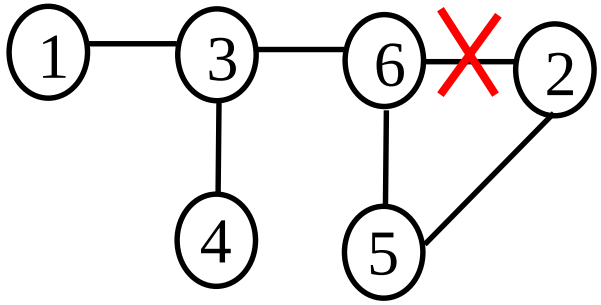




Domanda: Supponendo che il peso del lato c_{26} diventi 6 si discuta che cosa cambia nella soluzione ottima. Usare le condizioni di ottimalità sui tagli per ricalcolare la nuova soluzione ottima senza eseguire nuovamente l'algoritmo.

Esercizio su Algo Prim – Dijkstra (14)





Il costo del nuovo albero è 22

E' sufficiente individuare il taglio composto dai lati (1,2), (2,6), (2,5) e osservare come il lato c_{26} con il nuovo peso (6) non è il lato del taglio con il peso minimo. Quindi per le condizioni di ottimalità sui tagli, il lato c_{26} non appartiene più all'albero ricoprente di costo minimo, ed è sostituito dal lato c_{25} con peso 5.

Altro esercizio su Algo Prim - Dijkstra (1)

In tabella è riportato il peso di lati di un grafo (non orientato) con 8 vertici : 1…8. Trovare l'albero ricoprente di peso minimo, a partire dal vertice **1**, utilizzando l'algo di Prim-Dijkstra (efficiente). Indicare in quale ordine si aggiungono i lati all'albero ricoprente (in quale ordine vengono fissati a 1 i flag dei vertici del grafo).

Lati	(1,2)	(1,3)	(1,5)	(1,6)	(1,8)	(2,3)	(2,4)	(2,7)	(3,4)	(3,6)	(4,8)	(5,6)	(5,7)	(6,7)	(6,8)	(7,8)
Costi	3	2	12	4	6	19	2	11	5	4	6	10	1	21	4	6

dariano,pacciarelli@dia.uniroma3

Altro esercizio su Algo Prim - Dijkstra (2)

Lati	(1,2)	(1,3)	(1,5)	(1,6)	(1,8)	(2,3)	(2,4)	(2,7)	(3,4)	(3,6)	(4,8)	(5,6)	(5,7)	(6,7)	(6,8)	(7,8)
Costi	3	2	12	4	6	19	2	11	5	4	6	10	1	21	4	6

Soluzione:

- L'ordine in cui vengono fissati i flag dei vertici è

  1, 3, 2, 4, 6, 8, 7, 5
- L'albero ricoprente di costo minimo è composto dai seguenti lati (1,3), (1,2), (2,4), (1,6), (4,8), (8,7), (7,5)
- Il costo dell'albero è 24 (2+3+2+4+6+6+1)
- In particolare il costo del cammino da 1 a 5 è 18 (3+2+6+6+1)
- Si osservi come al posto del lato (1,6) si possa selezionare il lato (3,6) ottenendo comunque una soluzione ottima che segue esattamente lo stesso ordinamento dei flag

  (ovvero in questo caso esistono più soluzioni ottime)

Esercizio su Algo Kruskal (1)

In tabella è riportata la matrice di incidenza vertici/lati di un grafo (non orientato). La prima e l'ultima riga indicano, rispettivamente, i nomi dei lati e i loro pesi.

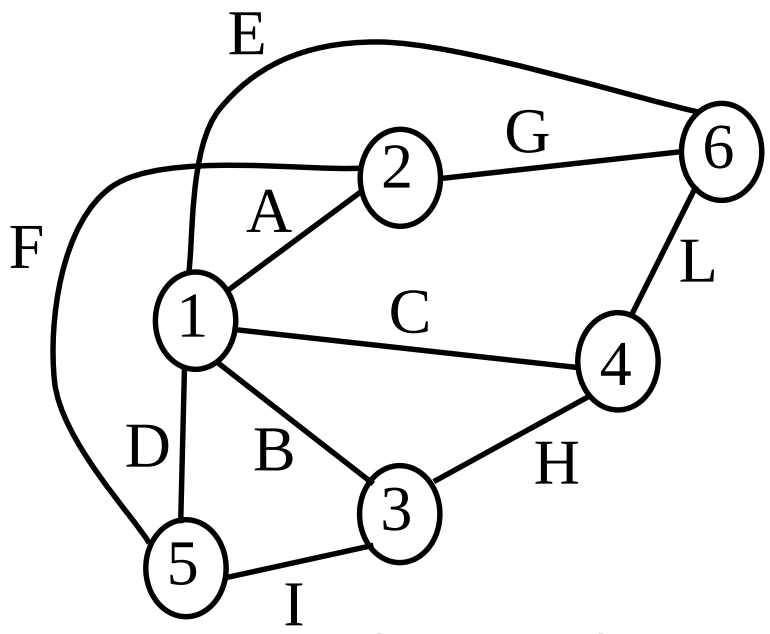
	A	*B*	*C*	*D*	*E*	*F*	*G*	*H*	*I*	*L*
1	1	1	1	1	1	0	0	0	0	0
2	1	0	0	0	0	1	1	0	0	0
3	0	1	0	0	0	0	0	1	1	0
4	0	0	1	0	0	0	0	1	0	1
5	0	0	0	1	0	1	0	0	1	0
6	0	0	0	0	1	0	1	0	0	1
Costi	7	17	12	3	17	8	11	9	13	9

Trovare l'albero ricoprente di peso minimo, con l'algo di Kruskal.

Indicare in quale ordine vengono aggiunti i lati all'albero ricoprente.

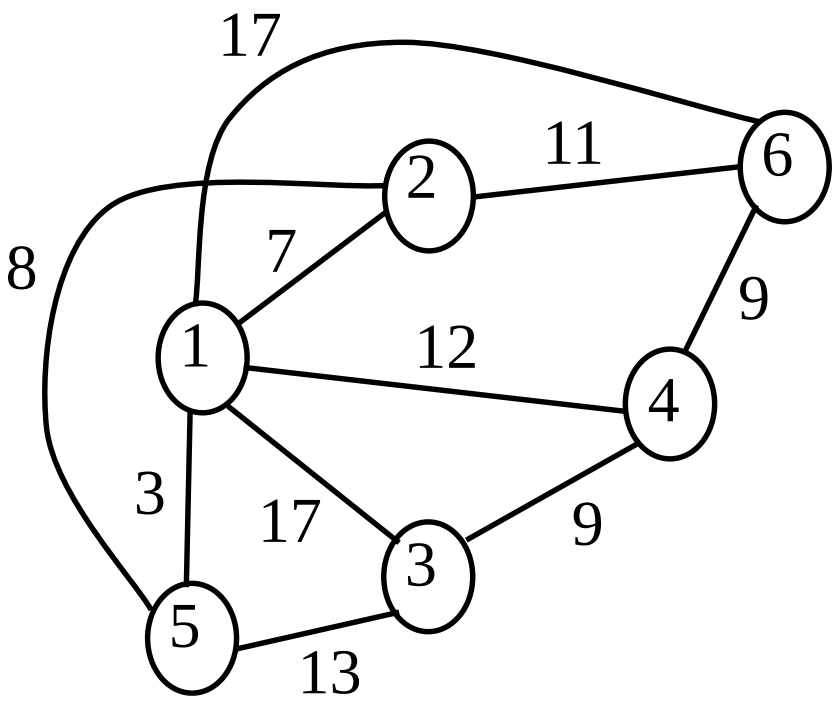
Esercizio su Algo Kruskal (2)

	A	*B*	*C*	*D*	*E*	*F*	*G*	*H*	*I*	*L*
1	1	1	1	1	1	0	0	0	0	0
2	1	0	0	0	0	1	1	0	0	0
3	0	1	0	0	0	0	0	1	1	0
4	0	0	1	0	0	0	0	1	0	1
5	0	0	0	1	0	1	0	0	1	0
6	0	0	0	0	1	0	1	0	0	1
Costi	7	17	12	3	17	8	11	9	13	9



dariano,pacciarelli@dia.uniroma3

Esercizio su Algo Kruskal (3)



Ordino i lati per peso crescente:

(1,5) (1,2) (5,2) (4,6) (3,4) (2,6)

(1,4) (3,5) (1,6) (1,3)

COMP[1,2,3,4,5,6]

Scelgo (1,5)

COMP[1,2,3,4,1,6]

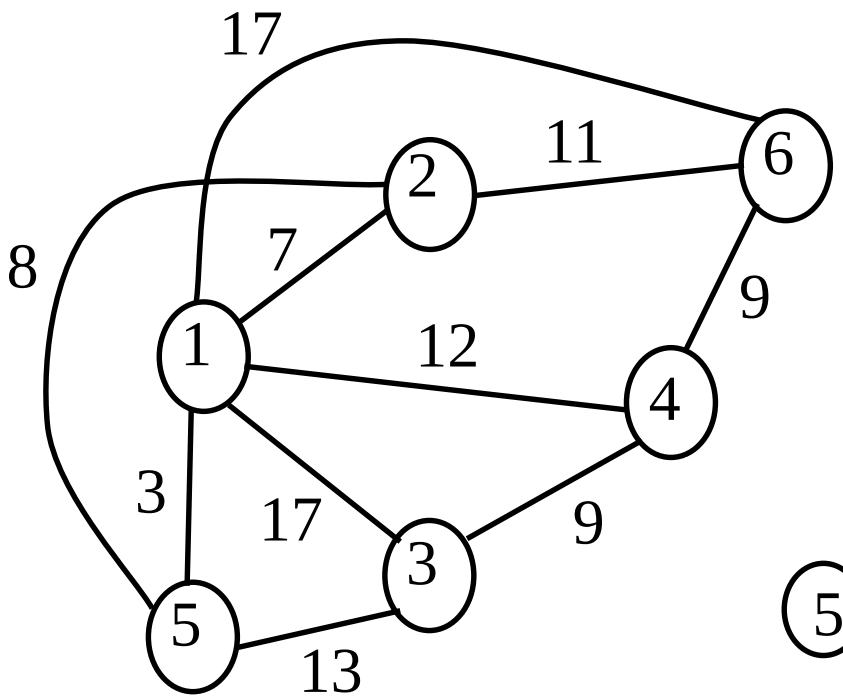
Scelgo (1,2)

COMP[1,1,3,4,1,6]

Scarto (5,2) => ciclo 1521

Esercizio su Algo Kruskal (4)

(1,5) (1,2) (5,2) (4,6) (3,4) (2,6) (1,4) (3,5) (1,6) (1,3)



COMP[1,1,3,4,1,6]

Scelgo (4,6)

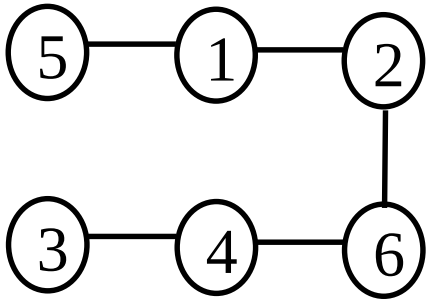
COMP[1,2,3,4,1,4]

Scelgo (3,4)

COMP[1,1,3,3,1,3]

Scelgo (2,6)

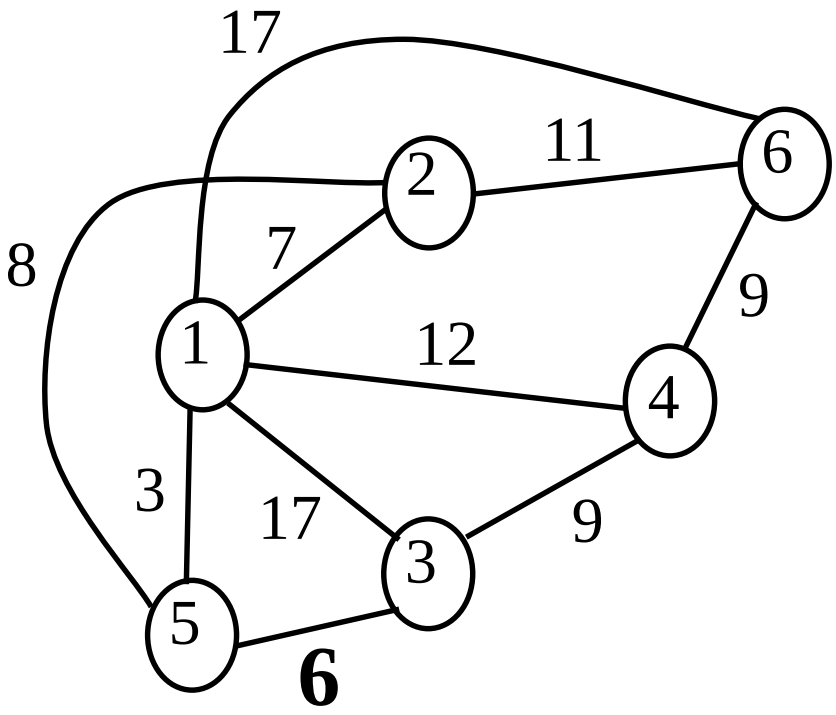
COMP[1,1,1,1,1,1]

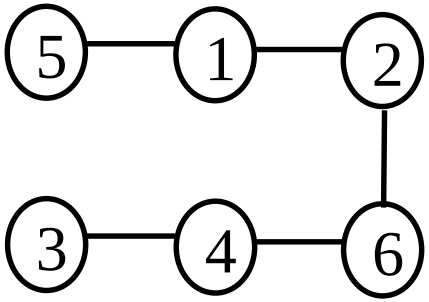


Albero ricoprente
di costo minimo
Costo dell'albero 39

I lati vengono aggiunti all'albero ricoprente nel seguente ordine:
(1,5), (1,2), (4,6), (3,4), (2,6) oppure (1,5), (1,2), (3,4), (4,6), (2,6).

Esercizio su Algo Kruskal (5)

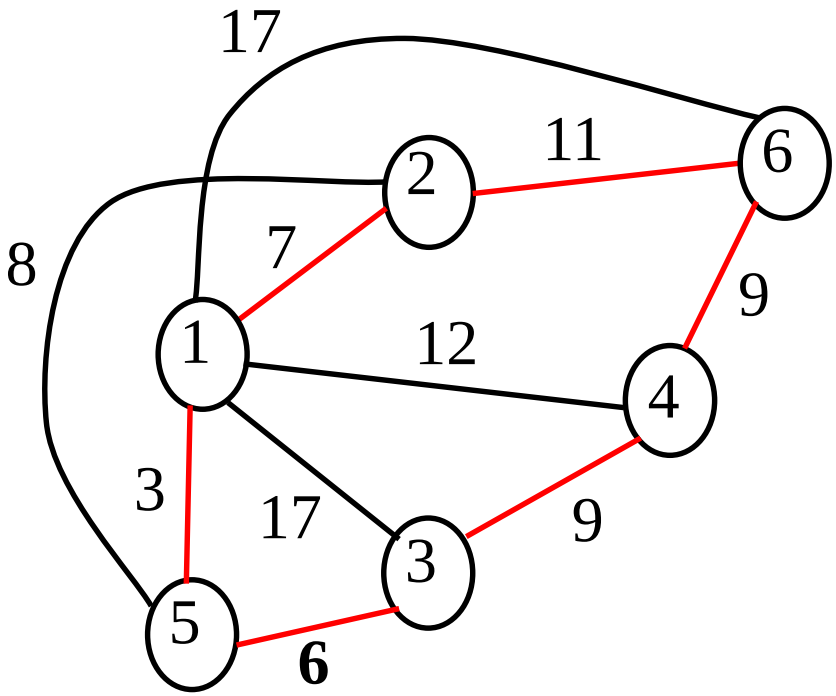


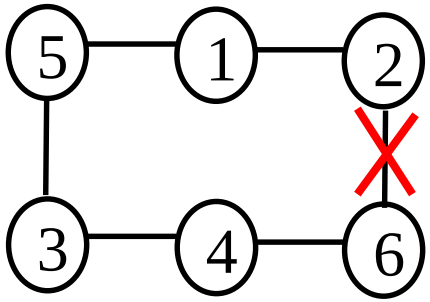


Albero ricoprente
di costo minimo
Costo dell'albero 39

Domanda: Supponendo che il peso del lato c_{53} diventi 6, si discuta che cosa cambia nella soluzione ottima. Usare le condizioni di ottimalità sui cammini per ricalcolare la nuova soluzione ottima senza eseguire di nuovo l'algoritmo.

Esercizio su Algo Kruskal (6)





Nuovo albero ricoprente
di costo minimo
Costo del nuovo albero 34

E' sufficiente individuare il cammino tra 3 e 5 composto dai lati (3,4), (4,6), (6,2), (2,1), (1,5) e osservare come il lato c_{53} è di peso (6) inferiore al lato con il **peso massimo del cammino** (ovvero il lato c_{26} di peso 11). Da cui per le condizioni di ottimalità sui cammini, il lato c_{53} apparterrà al nuovo albero ricoprente di costo minimo, sostituendo il lato c_{26} appartenente al vecchio albero.