Corso di laurea in Informatica
Progettazione d'algoritmi

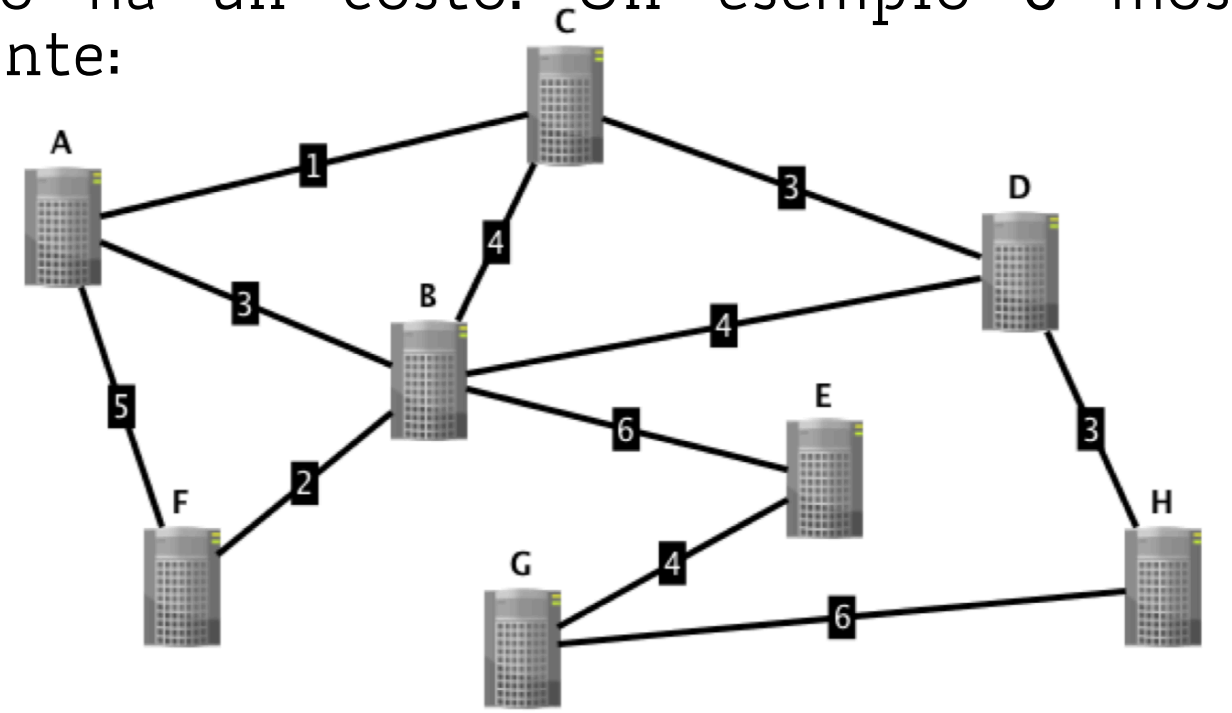
Il minimo albero di copertura

Angelo Monti

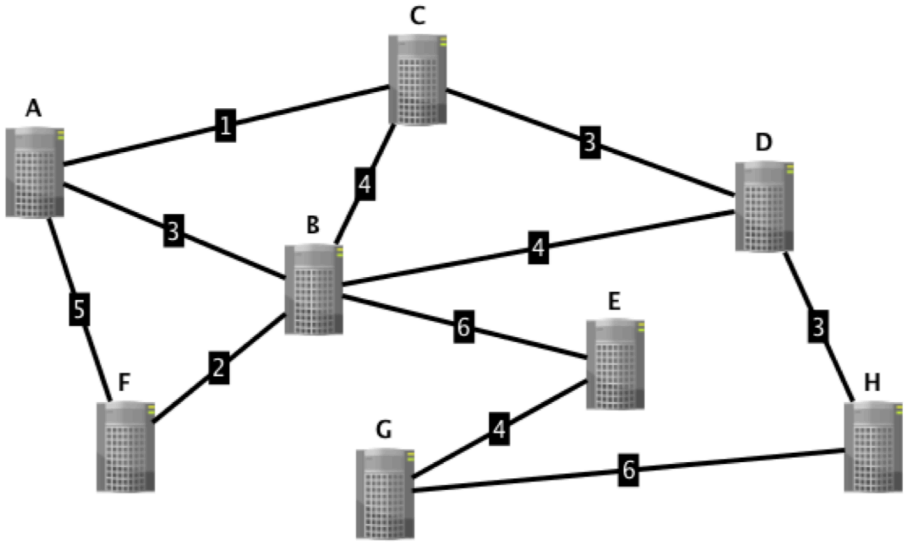
SAPIENZA
UNIVERSITÀ DI ROMA

Il problema del minimo albero di copertura

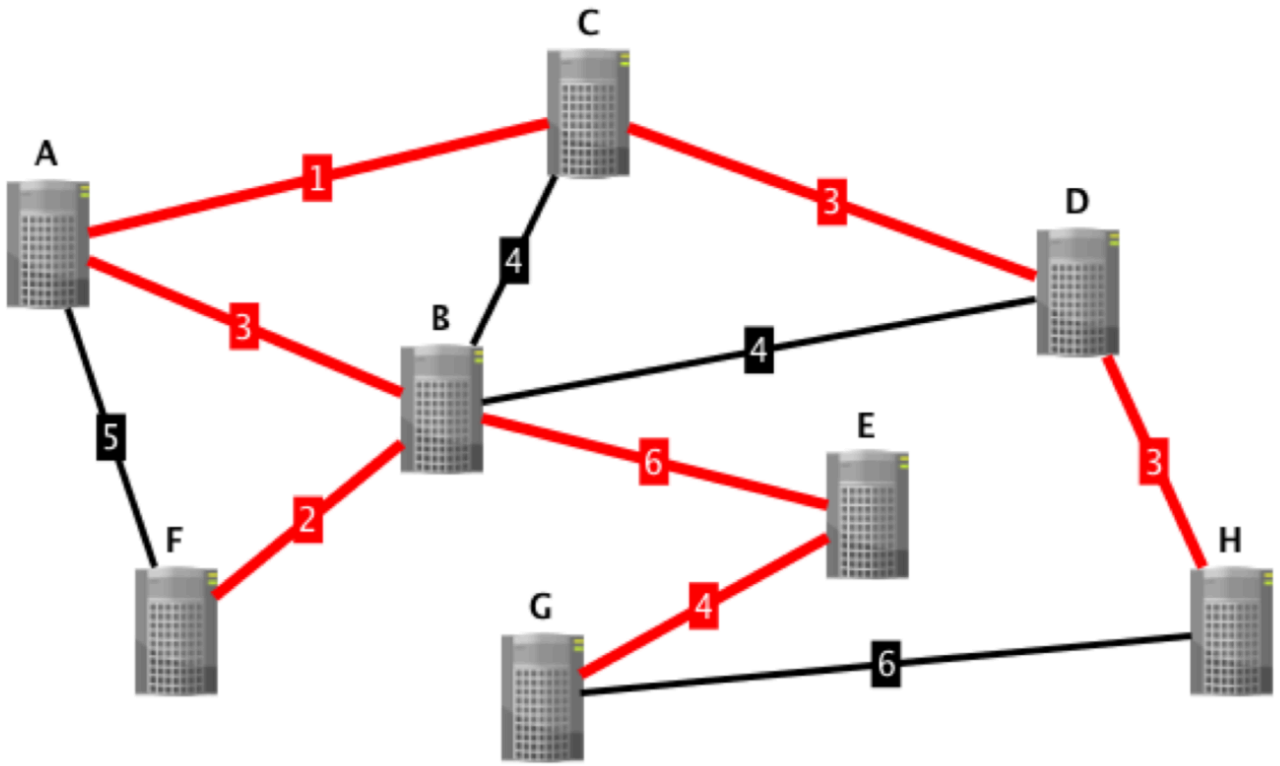
Consideriamo un insieme di computer (server e/o router) che devono essere connessi tramite cavi a formare una rete in modo che ogni computer possa comunicare con ogni altro o tramite un cavo che li collega direttamente o passando per altri computer delle rete. Ogni possibile collegamento tramite cavo ha un costo. Un esempio è mostrato nella figura seguente:

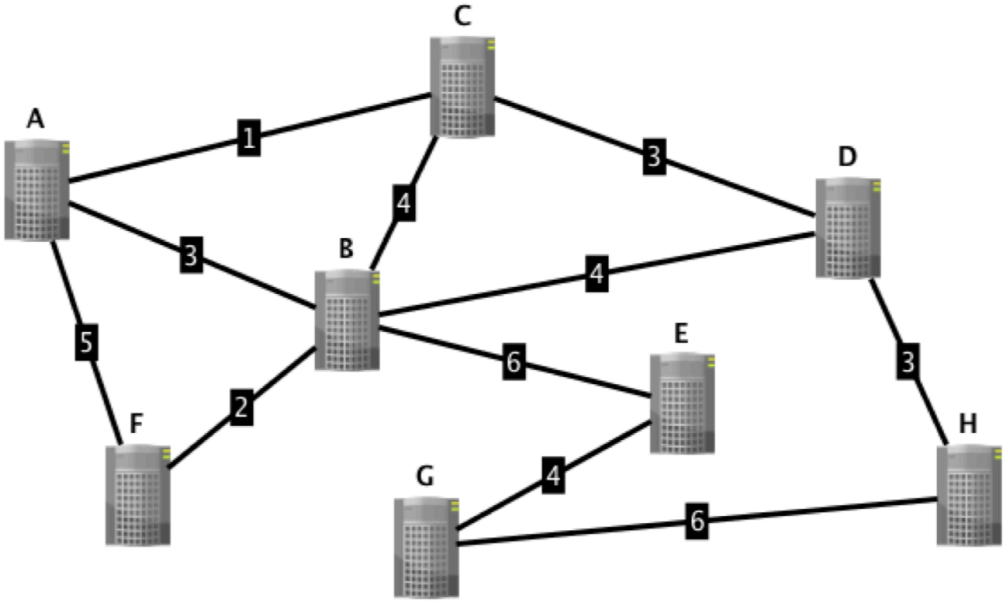


Quindi vogliamo installare alcuni dei collegamenti possibili in modo tale da garantire la connessione della rete e al contempo minimizzare il costo totale.



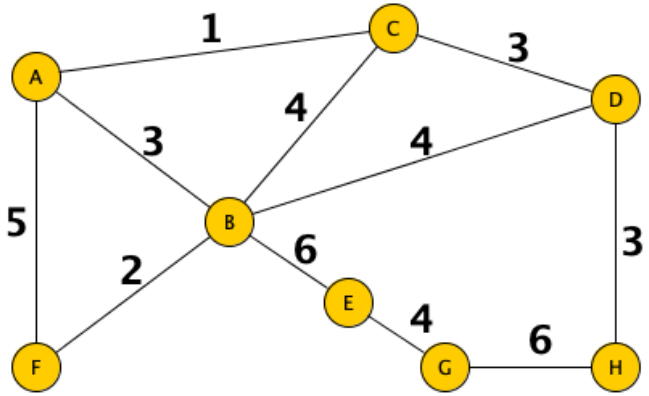
Una possibile soluzione (di costo 22):



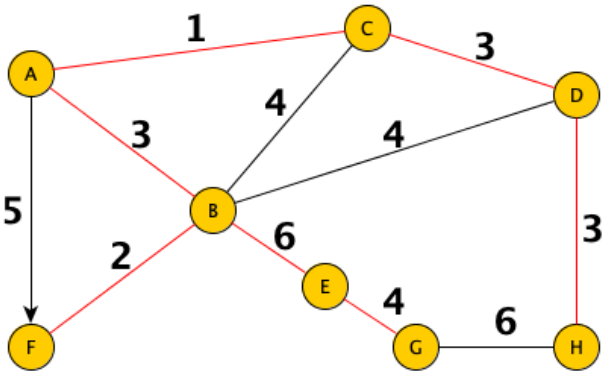


Il problema può essere rappresentato tramite un grafo pesato e connesso G i cui nodi sono i computer, gli archi sono i possibili collegamenti con i loro costi.

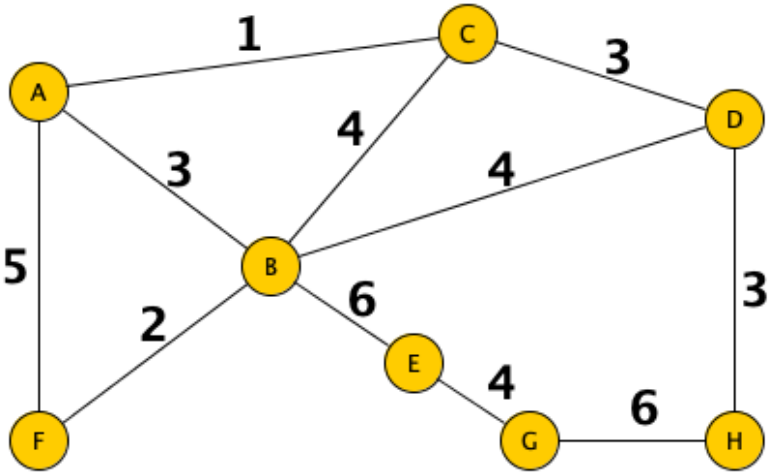
$G =$

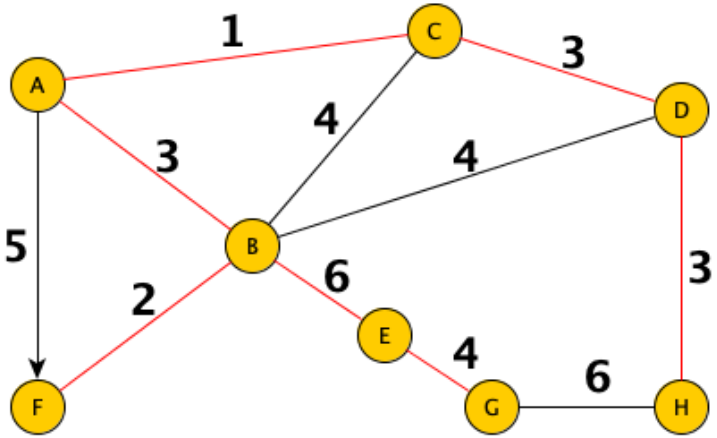


Il requisito della connessione nel modello tramite grafo pesato G è tradotto nel requisito che l'insieme degli archi da selezionare T deve connettere tutti i nodi.



$G =$





Nota che nel grafo soluzione (con gli archi in rosso in figura) non sono mai presenti cicli (l'eliminazione di un qualunque arco del ciclo non farebbe perdere la connessione e diminuirebbe il costo della soluzione).

Il sottoinsieme degli archi del grafo che formano la soluzione è dunque un albero (grafo connesso aciclico). Andiamo quindi alla ricerca in G di un albero che "copre" l'intero grafo e la somma dei costi dei suoi archi sia minima. Questo problema prende il nome di minimo albero di copertura.

Problema: Dato un grafo G connesso e pesato cerchiamo un suo minimo albero di copertura.

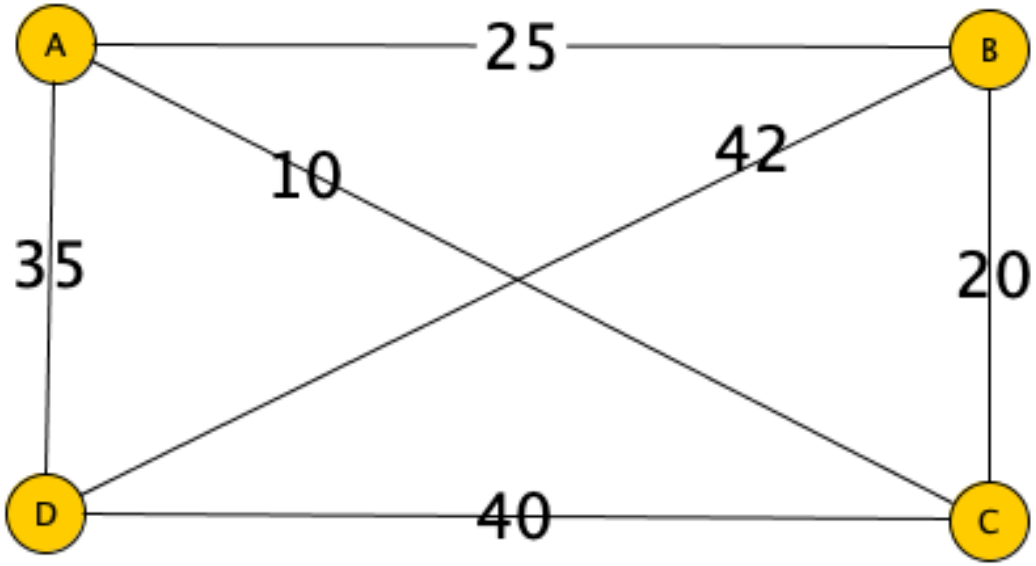
Data la sua importanza esistono diversi algoritmi per risolvere questo problema. Ne vedremo ora uno: l'algoritmo di Kruskal:

- **Parti con il grafo T che contiene tutti i nodi di G e nessun arco di G.**
- **Considera uno dopo l'altro gli archi del grafo G in ordine di costo crescente.**
- **Se l'arco forma ciclo in T con archi già presi allora non prenderlo altrimenti inseriscilo in T.**
- **Al termine restituisci T.**

Nota che l'algoritmo rientra perfettamente nel paradigma della tecnica greedy:

- La sequenza di decisioni irrevocabili: decidi per ciascun arco di G se inserirlo o meno in T. Una volta deciso cosa fare dell'arco non ritornare più su questa decisione.
- le decisioni vengono prese in base ad un criterio "locale": se l'arco crea ciclo non lo prendi, in caso contrario lo prendi in quanto è il meno costoso a non creare cicli tra gli archi che restano da considerare.

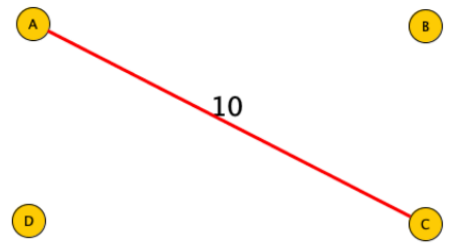
$G =$



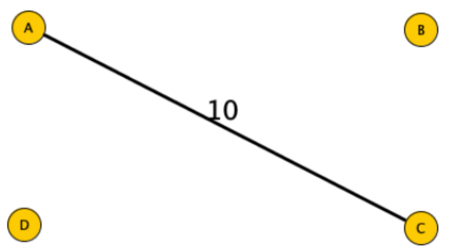
- Parti con il grafo T che contiene tutti i nodi di G e nessun arco di G.
- Considera uno dopo l'altro gli archi del grafo G in ordine di costo crescente.
- Se l'arco forma ciclo in T con archi presi allora non prenderlo altrimenti inseriscilo in T.
- Al termine restituisci T.

Esecuzione…..

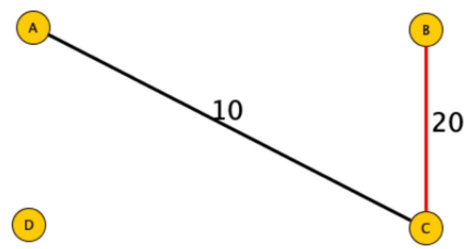
- AC 10
- BC 20
- AB 25
- AD 35
- DC 40
- BD 42



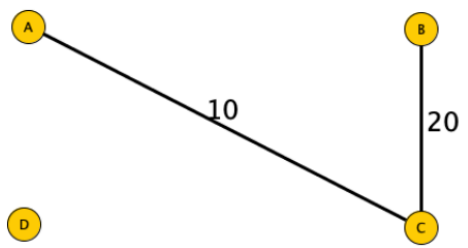
- AC 10
- BC 20
- AB 25
- AD 35
- DC 40
- BD 42



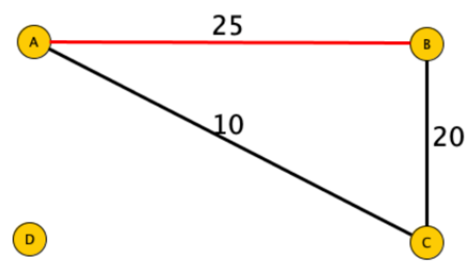
- AC 10
- BC 20
- AB 25
- AD 35
- DC 40
- BD 42



- AC 10
- BC 20
- AB 25
- AD 35
- DC 40
- BD 42

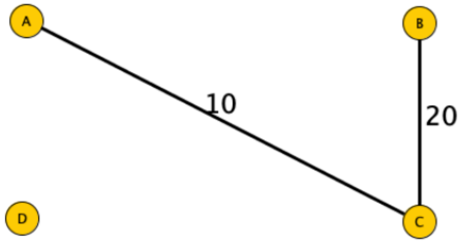


- AC 10
- BC 20
- AB 25
- AD 35
- DC 40
- BD 42

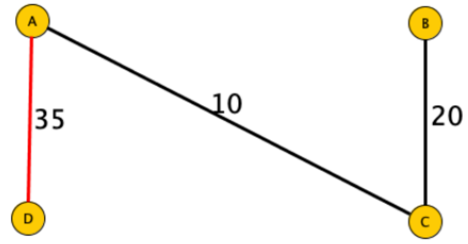


- AC 10
- BC 20
- AB 25
- AD 35
- DC 40
- BD 42

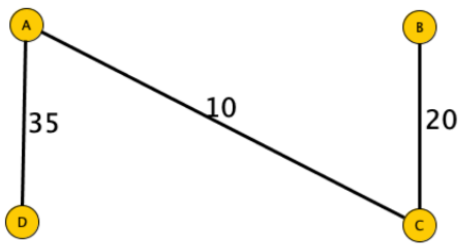
Esecuzione…..



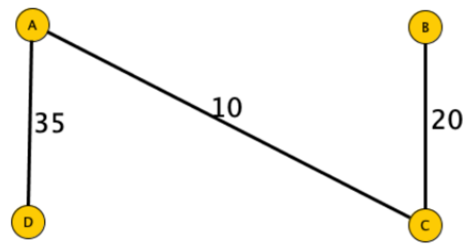
- AC 10
- BC 20
- AB 25
- AD 35
- DC 40
- BD 42



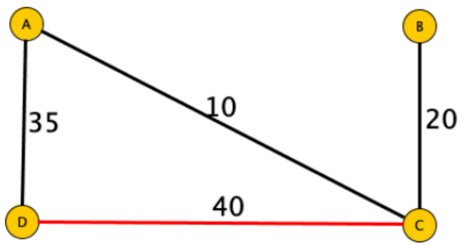
- AC 10
- BC 20
- AB 25
- AD 35
- DC 40
- BD 42



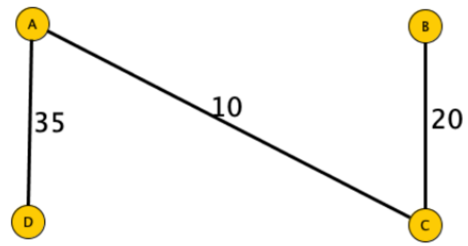
- AC 10
- BC 20
- AB 25
- AD 35
- DC 40
- BD 42



- AC 10
- BC 20
- AB 25
- AD 35
- DC 40
- BD 42



- AC 10
- BC 20
- AB 25
- AD 35
- DC 40
- BD 42



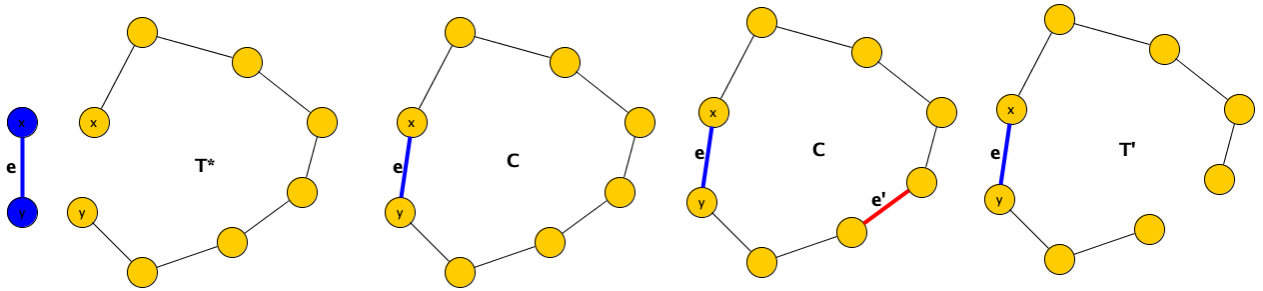
- AC 10
- BC 20
- AB 25
- AD 35
- DC 40
- BD 42

```
kruskal(G):
    T = set()
    inizializza E con gli archi di G
    while E! = []:
        estrai da E un arco (x, y) di peso minimo
        if l'inserimento di (x, y) in T non crea ciclo con gli archi in T:
            inserisci arco (x, y) in T
    return T
```

Correttezza:
Dobbiamo far vedere che al termine dell'algoritmo T è un albero di copertura e che non c'è un altro albero che costa meno.

- *Produce un albero di copertura*. La prova è per assurdo. Supponiamo che al termine in T ci sia più di una componente. Consideriamone una A, poiché G è connesso nel grafo c'è un arco (x, y) da un nodo x di A ad un nodo y di un'altra componente B ma l'arco (x, y) ad un certo punto è stato estratto da E e se non crea ciclo in T ora che l'algoritmo è terminato non lo creava neanche al momento in cui è stato estratto e quindi sarebbe stato aggiunto a T e non scartato.

- *Non c'e' un albero di copertura per G che costa meno dell'albero T ottenuto da Kruskal*.
  La prova è per assurdo. Tra tutti gli alberi di copertura di costo minimo per G prendiamo quello che differisce nel minor numero di archi da T. Sia T^*. Supponiamo per assurdo che T differisca da T^*. Faremo vedere che questa assunzione porterebbe all'assurdo perché avrebbe come conseguenza l'esistenza di un altro albero di copertura di costo minimo per G che differisce da T in meno archi di T^*.
  Considera l'ordine $e_1, e_2, \ldots$ con cui gli archi sono stati presi in considerazione nel corso dell'algoritmo e sia e il primo arco preso che non compare in T^*. Se inserisco e in T^* si forma di certo un ciclo C. Il ciclo C contiene almeno un arco e' che non appartiene a T (infatti non tutti gli archi del ciclo C sono in T altrimenti e non sarebbe stato preso dall'algoritmo di Kruskal) . Considera ora l'albero T' che ottengo da T^* inserendo l'arco e ed eliminando e'. Il costo del nuovo albero T' (che è $costo(T^*) - costo(e') + costo(e)$) non può aumentare rispetto a quello di T^*(perché $costo(e) \leq costo(e')$, nota infatti che tra i due archi e ed e' che non creavano ciclo Kruskal ha considerato prima l'arco e) ma allora T' è un altro albero di copertura ottimo che differisce da T in meno archi di quanto faccia T^* il che contraddice l'ipotesi che T^* differisce da T nel minor numero di archi.



Implementazione:

```
kruskal(G):
    T = set()
    inizializza una lista E con gli archi di G
    while E != []:
        estrai da E un arco (x, y) di peso minimo
        if l'inserimento di (x, y) in T non crea ciclo con archi in T:
            inserisci arco (x, y) in T
    return T
```

Idee:

- Con un pre-processing ordino gli archi nella lista E cosicché scorrendo la lista ottengo di volta in volta l'arco di costo minimo in tempo $O(1)$.

- Verifico che l'arco (x, y) non formi ciclo in T controllando se y è raggiungibile da x in T.

Implementazione:

```
def kruskal(G):
    # crea la lista con gli archi pesati di G
    E=[(c,u,v) for u in range(len(G)) for v,c in G[u] if u<v]
    # ordina gli archi in ordine crescente di peso
    E.sort()
    T=[[] for _ in G]
    for c,u,v in E:
        if not connessi(u,v,T):
            T[u].append(v)
            T[v].append(u)
    return T
```

```
def connessi(u, v, T):
    ''' esegue una visita nella grafo T a partire da u e restituisce
    True se nel corso della visita si raggiunge v, False altrimenti'''
    ###
    def DFSr(a, b, T, visitati):
        visitati[a] = 1
        for z in T[a]:
            if z == b:
                return True
            if not visitati[z]:
                if  DFSr(z,b,T,visitati):
                    return True
        return False
    ###
    visitati=[0]*len(T)
    return DFSr(u, v, T, visitati)
```

Complessità dell'implementazione:

```
def kruskal(G):
    # crea la lista con gli archi pesati di G
    E=[(c,u,v) for u in range(len(G)) for v,c in G[u] if u<v]
    # ordina gli archi in ordine crescente di peso
    E.sort()
    T=[[] for _ in G]
    for c,u,v in E:
        if not connessi(u,v,T):
            T[u].append(v)
            T[v].append(u)
    return T
```

- L'ordinamento esterno al *for* costa $O(m \log m) = O(m \log n^2) = O(m \log n)$.

- Il *for* viene iterato m volte
  - Controllare che l'arco (a, b) non crei ciclo in T con la procedura $connessi(a, b, T)$ richiede il costo di una visita di un grafo aciclico quindi $O(n)$.

  Il for richiede tempo $O(m \cdot n)$

La complessità di questa implementazione è $O(m \cdot n)$.

Per migliorare l'implementazione non possiamo permetterci di pagare tempo $O(n)$ ad ogni iterazione del *for* a causa della visita per testare la raggiungibilità.

Ricorreremo alla struttura dati UNION e FIND che permette di testare efficientemente se due nodi appartengano o meno alla stessa componente connessa.

la UNION-FIND, è una struttura dati per la collezione C delle componenti connesse di un grafo di n nodi in modo tale che sia possibile "efficientemente" effettuare le due seguenti operazioni:

- -UNION(a, b, C) fonde due componenti connesse a e b in C in tempo $O(1)$
- -FIND(x, C): trova in C la componente connessa in cui si trova il nodo x. in tempo $O(\log n)$.

```
def kruskal(G):
    E = [ (c, u, v) for u in G for v, c in G[u]  if u < v]
    E.sort( )
    T =[ [  ]  for  _  in G]
    C=crea(T)
    for c, u, v in E:
        cu = find(u, C)
        cv = find(v, C)
        if cu != cv:
            T[x].append(y)
            T[y].append(x)
            union(cu, cv, C)
    return T
```

Implementazione finale dell'algoritmo di Kruskal ha complessità $O(m \log n)$.

```
def kruskal1(G):
    E=[ (c, u, v) for u in range(len(G)) for v, c in G[u] if u < v]
    E.sort()
    T = [ [ ] for _ in G ]
    C = Crea(T)
    for c,u,v in E:
        cu = Find(u, C)
        cv = Find(v, C)
        if cu != cv:
            T[u].append(v)
            T[v].append(u)
            Union(cu,cv, C)
    return T
```

```
def Crea(G):
    C=[ (i,1) for i in range(len(G))]
    return C

def Find(u,C):
    while u != C[u]:
        u = C[u]
    return u


def Union (a,b,C):
    tota, totb = C[a][1], C[b][1]
    if tota >= totb:
        C[a]=(a, tota + totb)
        C[b]=(a, totb)
    else:
        C[b]=(b, tota + totb)
        C[b]=(a, totb)
```

- L'ordinamento costa $O(m \log n)$
- il *for* viene iterato m volte:
  - L'estrazione dell'arco (a, b) di costo minimo da E richiede $\Theta(1)$
  - eseguire il FIND costa $O(\log n)$
  - eseguire l'UNION costa $\Theta(1)$ (e all'interno del *for* viene eseguito esattamente $n - 1$ volte).
- il costo del *for* è $O(m \log n)$

La complessità di questa implementazione è $O(m \log n)$.

Corso di laurea in Informatica
Introduzione agli Algoritmi

Esercizi per casa

SAPIENZA
UNIVERSITÀ DI ROMA

- Sia G un grafo connesso e pesato con pesi positivi. Sia T un suo minimo albero di copertura e T_s il suo albero dei cammini minimi a partire da un suo nodo s. Provare o confutare che la somma dei pesi degli archi di T è uguale alla somma dei pesi degli archi di T_s.

- Sia G un grafo non diretto, connesso e pesato dove i pesi sono tutti diversi tra loro. Sia T un minimo albero di copertura di G, sia s un nodo e T_s l'albero dei cammini minimi da s verso tutti gli altri nodi di G. Dimostrare oppure fornire un controesempio che T_s e T condividono almeno un arco.

- Sia G un grafo non diretto, connesso e pesato, e G' il grafo non diretto,connesso e pesato che si ottiene da G moltiplicando il peso di ciascun arco per un a fissata costante $c > 0$. Sia T un albero di copertura di costo minimo per G e T_s l'albero dei cammini costruito a partire dal nodo s per G. Dimostrare o confutare che i due alberi sono di copertura minima e dei cammini minimi anche per il grafo G'.

Ci sono n case, indicate con gli interi $i = 1, 2, \ldots, n$, ognuna delle quali necessita di una fornitura d'acqua. La costruzione di un pozzo nella casa i costa $p[i]$ e la costruzione di una tubazione fra le case i e j costa $c[i, j]$. La fornitura d'acqua per una casa o è un pozzo costruito nella casa o è un cammino di tubazioni dalla casa a qualche pozzo.
Dare lo pseudo-codice di un algoritmo che determina le case in cui costruire i pozzi e le tubazioni da costruire per dare una fornitura d'acqua a tutte le case minimizzando il costo totale.
L'algoritmo deve avere complessità $O(n^2)$. Discutere la correttezza dell'algoritmo.
Suggerimento: rappresentare il problema tramite un opportuno grafo pesato con $n + 1$ *nodi, il nodo in più rappresenta i pozzi* ...