Corso di laurea in Informatica
Progettazione d'algoritmi
Didattica blended

Gli algoritmi greedy: esercizi

Angelo Monti

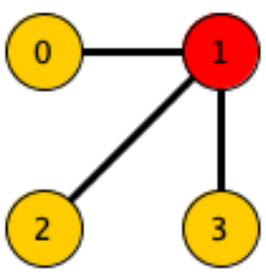
SAPIENZA
UNIVERSITÀ DI ROMA

ESERCIZIO1:

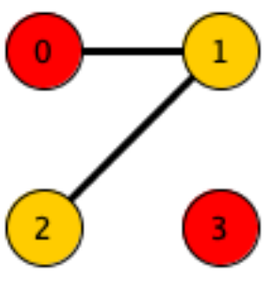
Ci sono n case, indicate con gli interi $i = 0, 1, \ldots, n-1$, ognuna delle quali necessita di una fornitura d'acqua.
La costruzione di un pozzo nella casa i costa $P[i]$ e la costruzione di una tubazione fra le case i e j costa $M[i][j]$ ($M[i][j] = -1$ se non c'è possibilità di stendere una tubazione tra i e j).
Una casa risulta fornita d'acqua se vi è stato costruito un pozzo o l'acqua vi giunge da un pozzo di un altra casa tramite tubazioni.

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.
Il costo totale è dato dalla somma dei costi dei pozzi e delle tubature costruiti.
L'algoritmo deve avere complessità $O(n^2 + m \log n)$.
Motivare bene la correttezza dell'algoritmo.

Per l'istanza con 4 case dove $P = [4,\ 5,\ 6,\ 3]$ e M= $\begin{bmatrix} -1 & 2 & 4 & 6 \\ 2 & -1 & 1 & 7 \\ 4 & 1 & -1 & 8 \\ 6 & 7 & 8 & -1 \end{bmatrix}$



è una soluzione di costo $5 + 2 + 1 + 7 = 15$

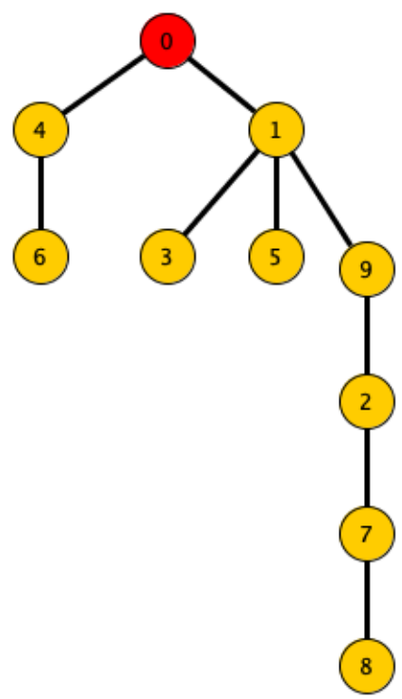


è la soluzione ottima di costo $4 + 3 + 2 + 1 = 10$

Per l'istanza con 10 case dove $P = [1, 8, 6, 5, 2, 3, 3, 2, 8, 4]$ e

$$M=\begin{bmatrix} -1 & 2 & 2 & 2 & 1 & 3 & -1 & 5 & 6 & 7 \\ 2 & -1 & 5 & 1 & -1 & 2 & -1 & 5 & 9 & 1 \\ 2 & 5 & -1 & -1 & -1 & 4 & 5 & 1 & -1 & 1 \\ 2 & 1 & -1 & -1 & 4 & 3 & 4 & 4 & 9 & 1 \\ 1 & -1 & -1 & 4 & -1 & 4 & 1 & 4 & 8 & 4 \\ 3 & 2 & 4 & 3 & 4 & -1 & 3 & 4 & 3 & 4 \\ -1 & -1 & 5 & -1 & 1 & 3 & -1 & 5 & 9 & -1 \\ 5 & 5 & 1 & 4 & 4 & 4 & 5 & -1 & 1 & 1 \\ 6 & 9 & -1 & 9 & 8 & 3 & 9 & 1 & -1 & 9 \\ 7 & 1 & 1 & 1 & 4 & 4 & -1 & 1 & 9 & -1 \end{bmatrix}$$

Una soluzione ottima è

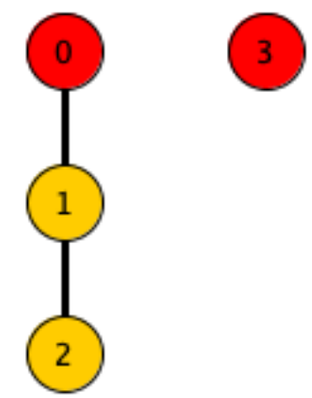


Il costo della soluzione è $1 + 1 + 2 + 1 + 1 + 2 + 1 + 1 + 1 + 1 = 12$

posso vedere una soluzione al problema come una foresta dove ogni albero ha per radice una casa in cui è stato scavato un pozzo mentre gli altri nodi dell'albero sono le case che ricevono acqua da quel pozzo tramite tubature (gli archi dell'albero).

Per l'istanza con 4 case dove $P = [4, 5, 6, 3]$ e M= $\begin{bmatrix} -1 & 2 & 4 & 6 \\ 2 & -1 & 1 & 7 \\ 4 & 1 & -1 & 8 \\ 6 & 7 & 8 & -1 \end{bmatrix}$

La soluzione è data dalla seguente foresta di due alberi:



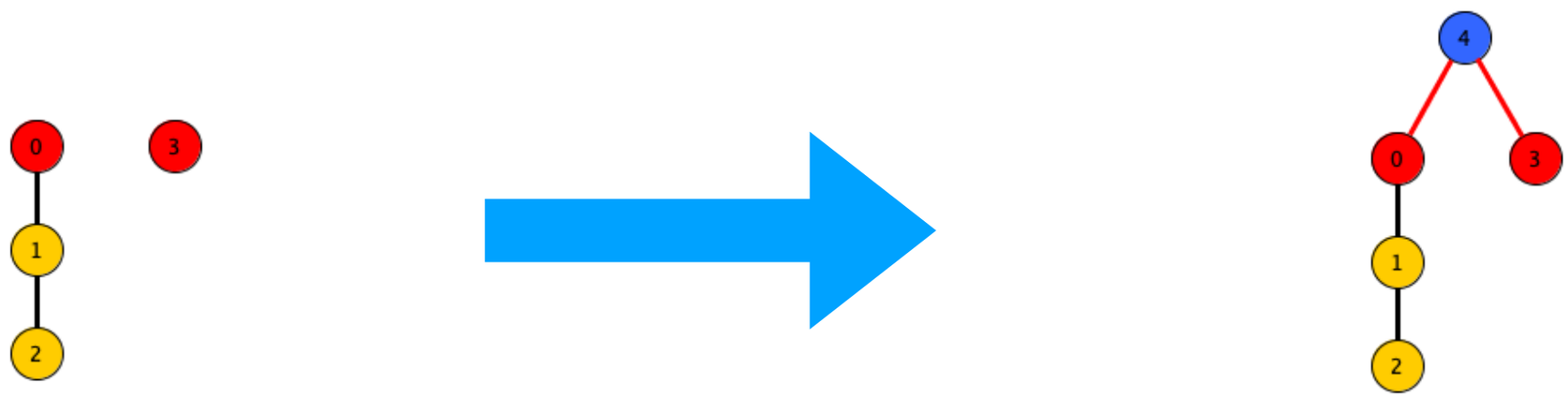
la soluzione come foresta di alberi con alla radice i pozzi deriva dal fatto che:

- le tubature non formano cicli (in caso contrario potrei eliminare una tubatura e risparmiare sui costi)
- non c'è un cammino tra due pozzi (in caso contrario potrei eliminare uno dei due pozzi e l'altro è sufficiente a rifornire le case a loro collegate)

Il costo delle tubature è dato dalla somma dei costi degli alberi della foresta. Per ottenere il costo della soluzione mancano i costi dell'attivazione dei pozzi-radice dei vari alberi.

Il costo delle tubature è dato dalla somma dei costi degli alberi della foresta. Per ottenere il costo della soluzione mancano i costi dell'attivazione dei pozzi-radice dei vari alberi.

Idea: **posso aggiungere una $n+1$-esima casa-fittizia e da questa far partire archi verso i pozzi dove il costo dell'arco verso il pozzo x ha il costo dell'attivazione del pozzo x.**



Ora la foresta è diventato un albero e il costo della soluzione al problema è dato dalla somma degli archi presenti nell'albero.

NOTA: **In questo modo il costo della soluzione al problema è pari al minimo albero di copertura del grafo G di $n+1$ nodi che si ottiene a partire dalla matrice M aggiungendo una riga ed una colonna con i valori di P:**

$$P = [\ 4,\ 5,\ 6,\ 3\ ]$$

$$\mathrm{M}=\begin{bmatrix} -1 & 2 & 4 & 6 \\ 2 & -1 & 1 & 7 \\ 4 & 1 & -1 & 8 \\ 6 & 7 & 8 & -1 \end{bmatrix} \quad \mathrm{G}=\begin{bmatrix} -1 & 2 & 4 & 6 & 4 \\ 2 & -1 & 1 & 7 & 5 \\ 4 & 1 & -1 & 8 & 6 \\ 6 & 7 & 8 & -1 & 3 \\ 4 & 5 & 6 & 3 & -1 \end{bmatrix}$$

NOTA: In questo modo il costo della soluzione al problema è pari al minimo albero di copertura del grafo G di $n+1$ nodi che si ottiene a partire dalla matrice M aggiungendo una riga ed una colonna con i valori di P:

$$P = [\ 4,\ 5,\ 6,\ 3\ ]$$

$$\mathrm{M}=\begin{bmatrix} -1 & 2 & 4 & 6 \\ 2 & -1 & 1 & 7 \\ 4 & 1 & -1 & 8 \\ 6 & 7 & 8 & -1 \end{bmatrix} \longrightarrow \mathrm{G}=\begin{bmatrix} -1 & 2 & 4 & 6 & 4 \\ 2 & -1 & 1 & 7 & 5 \\ 4 & 1 & -1 & 8 & 6 \\ 6 & 7 & 8 & -1 & 3 \\ 4 & 5 & 6 & 3 & -1 \end{bmatrix}$$

Nota: dall'albero di copertura ottimo per G possiamo ricavare:

- i pozzi della soluzione ottima (saranno i vicini del nodo $n+1$ nell'albero di copertura).
- le tubature (saranno gli archi dell'albero di copertura che non hanno un estremo nel nodo $n+1$).

Otteniamo il seguente algoritmo basato sulla tecnica della RIDUZIONE

def *pozzi_acqua*(M, P) :
  a partire da M e P costruisci il grafo G
  ottieni l'albero minimo di copertura T di G con l'algoritmo di Kruskal
  ricava il costo della soluzione ottima sommando i costi degli archi in T
  ricava i pozzi della soluzione ottima cercando i vicini del nodo $n+1$ in T.
  ricava le tubature della soluzione ottima prendendo gli archi in T che non hanno per estremo il nodo $n+1$
  return *costo, pozzi, tubature*

Implementazione:

```
def pozzi_acqua(M,P):
    G=crea_grafo(M,P)
    T=kruskal1(G)
    pozzi=set(T[len(M)])
    tubature=[ (i,j) for i in T for j in T[i] if i<j and j!=len(M)]
    costo=0
    for (i,j) in tubature: costo+=M[i][j]
    for x in pozzi: costo+=P[x]
    return costo, pozzi, tubature
```

```
def crea_grafo(M,P):
    G={}
    for i in range(len(M)):
        G[i]=[(j,c) for j,c in enumerate(M[i]) if c!=-1]
    G[len(M)]=[(j,c) for j,c in enumerate(P)]
    for (j,c) in G[len(M)]: G[j].append((len(M),c))
    return G
```

```
def crea(G):
    return  [(x,1) for x in G]

def find(Comp,a):
    while a != Comp[a][0]: a=Comp[a][0]
    return a

def union(Comp, ca, cb):
    tota, totb= Comp[ca][1], Comp[cb][1]
    if tota<totb:
        Comp[ca]=(cb, tota)
        Comp[cb]=(cb,tota+totb)
    else:
        Comp[cb]=(ca,totb)
        Comp[ca]=(ca,tota+totb)

def kruskal1(G):
    E = [(c, u, v) for u in G for v, c in G[u] if u < v]
    E.sort(reverse = True)
    T ={x:[] for x in G}
    Comp=crea(T)
    while E!=[] :
        c,x,y = E.pop()
        cx=find(Comp,x)
        cy=find(Comp,y)
        if cx!=cy:
            T[x].append(y)
            T[y].append(x)
            union(Comp,cx,cy)
    return T
```

- la funzione *crea_grafo* richiede $\Theta(n^2)$
- la funzione *Kruskal* richiede $O(n^2 + m \log n)$
- generare l'insieme dei pozzi richiede $O(n)$
- generare l'insieme delle tubature richiede $O(n)$
- generare il costo della soluzione richiede $O(n)$

La complessità è $O(n^2 + m \log n)$

Esercizio.

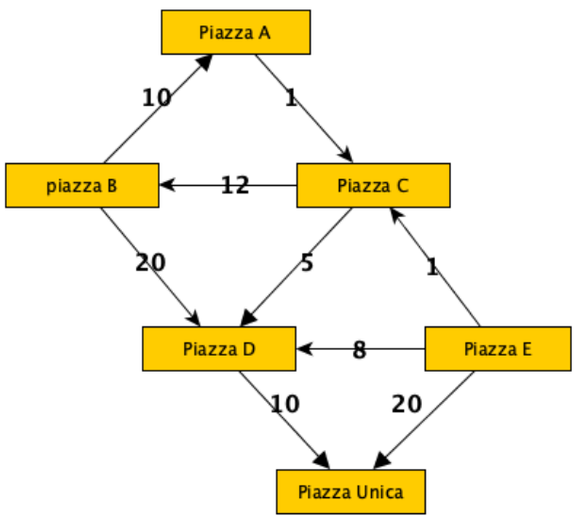
La rete viaria della città $Universa$ consiste di varie piazze collegate tra loro da strade. La particolarità di questa città è che tutte le strade sono a senso unico. La piazza $Unica$ è la più importante della città e si sa che è raggiungibile da qualunque altra piazza.
Il sindaco vuole selezionare un insieme di strade da riasfaltare in modo tale che da ogni piazza sia possibile poi raggiungere la piazza $Unica$ percorrendo solo strade asfaltate. Il costo di riasfaltare le varie strade dipende dalle loro lunghezze ed il sindaco vuole spendere il meno possibile. Gli urbanisti comunali pensano di poter trovare questo insieme di strade tramite il seguente algoritmo greedy che lavora sul grafo pesato G in cui ogni nodo è una piazza di $Universa$ e un arco da x a y di costo c sta a significare che c'è una strada che porta da x a y e riasfaltarla costa c.

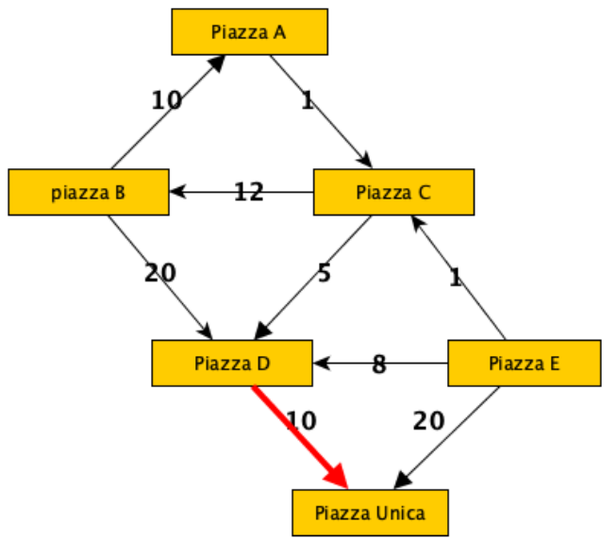
```
def seleziona(G) :
    Sol = set( )
    A = l'insieme con tutti i nodi di G tranne piazza Unica
    E = la lista con tutti gli archi di G
    while A != set( ) :
        prendi in E arco (x,y) con x in A e y non in A di costo minimo
        Sol.add((x,y))
        A.remove(x)
    return Sol
```

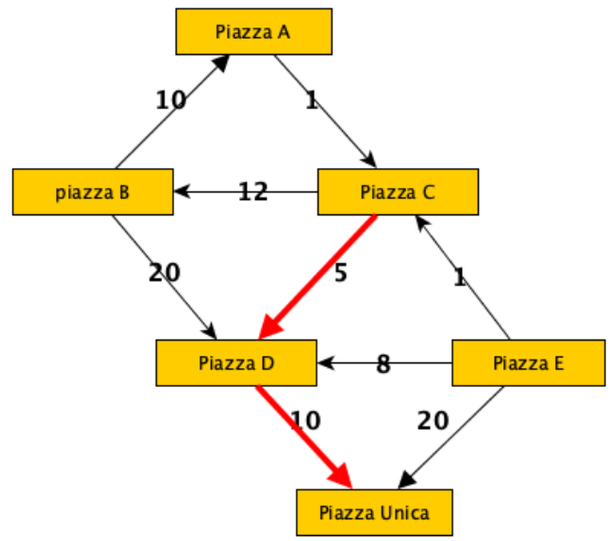
provare o confutare che:

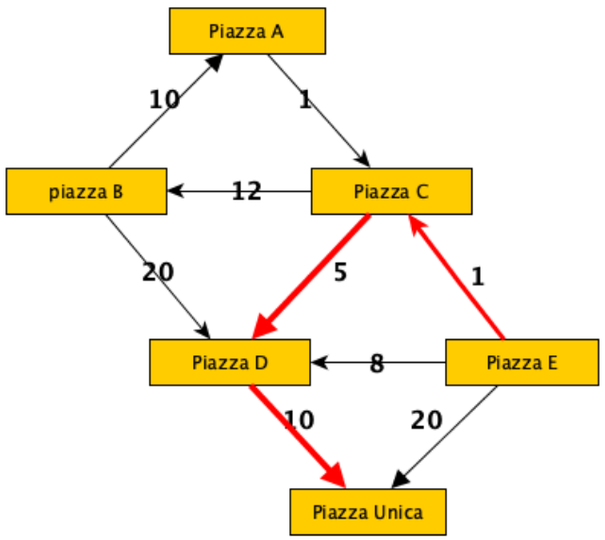
1. le strade selezionate permettono di raggiungere la piazza $Unica$ da qualunque altra piazza
2. la soluzione prodotta è di costo minimo

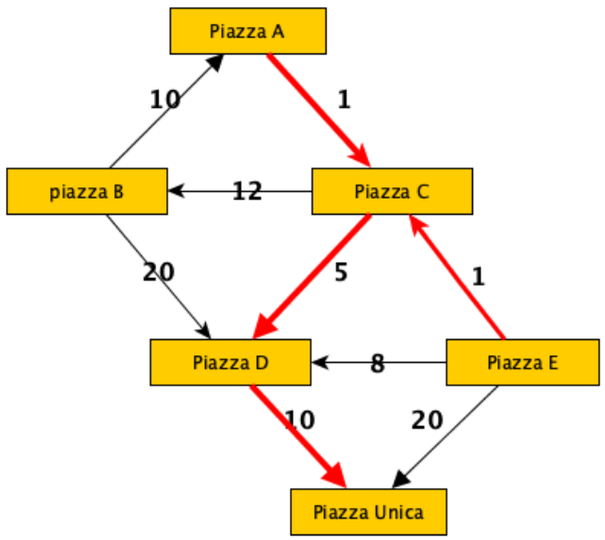
```
def seleziona(G):
    Sol = set( )
    A = l'insieme con tutti i nodi di G tranne Piazza Unica
    E = la lista con tutti gli archi di G
    while A != set( ):
        prendi in E arco (x,y) con x in A e y non in A di costo minimo
        Sol.add((x,y))
        A.remove(x)
    return Sol
```

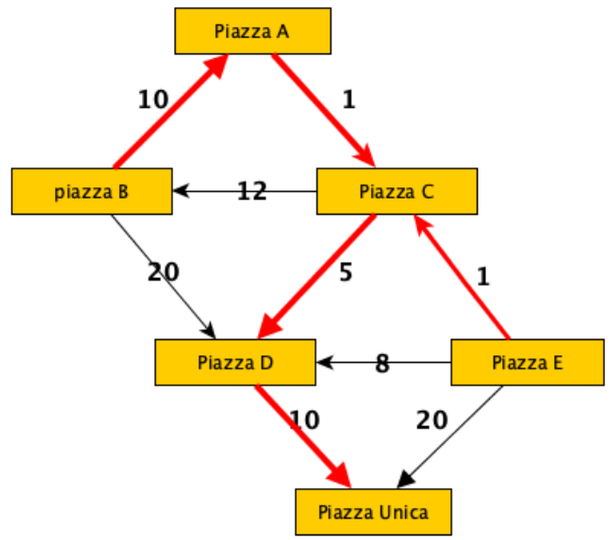










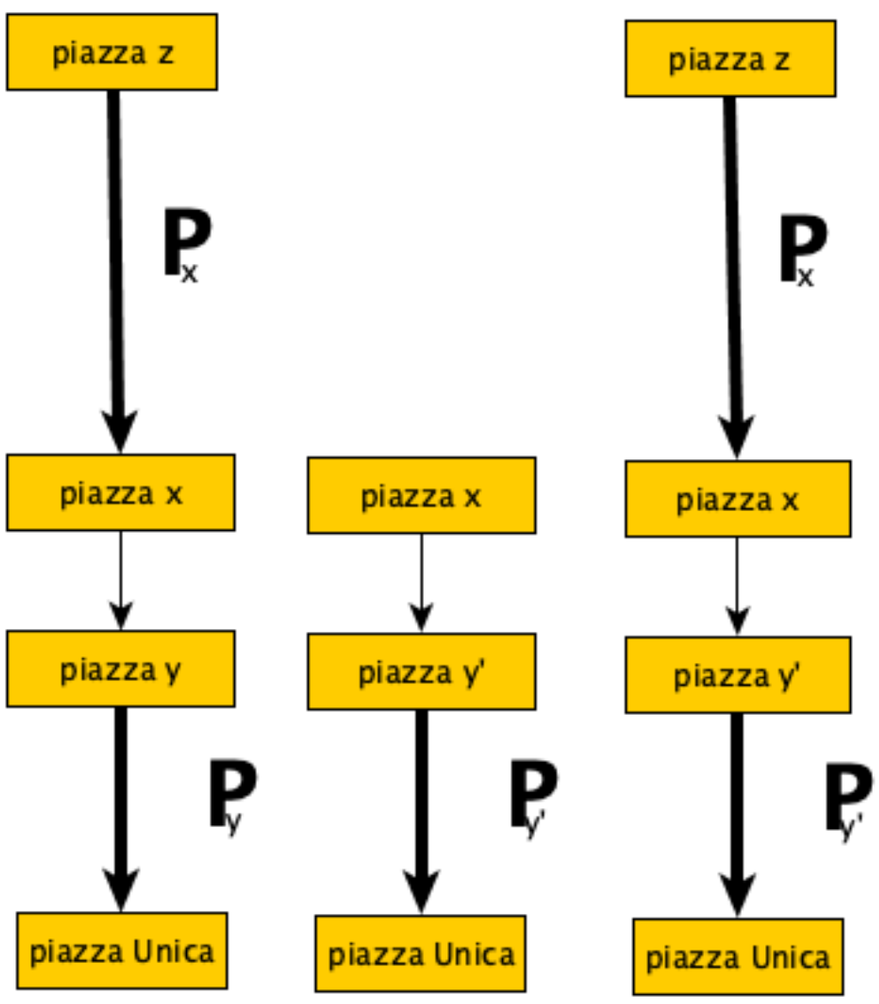


La soluzione prodotta costa 27.

le strade selezionate permettono di raggiungere la piazza Unica da qualunque altra piazza

Prova. assumiamo (per assurdo) che esista una piazza z da cui non è possibile raggiungere piazza $Unica$ con le strade selezionate dall'algoritmo. Tra tutti i percorsi che vanno da z a piazza $Unica$ sia P quello che presenta meno strade non asfaltate. Faremo vedere che ne esiste un'altro P' che ha una strada non asfaltata in meno:

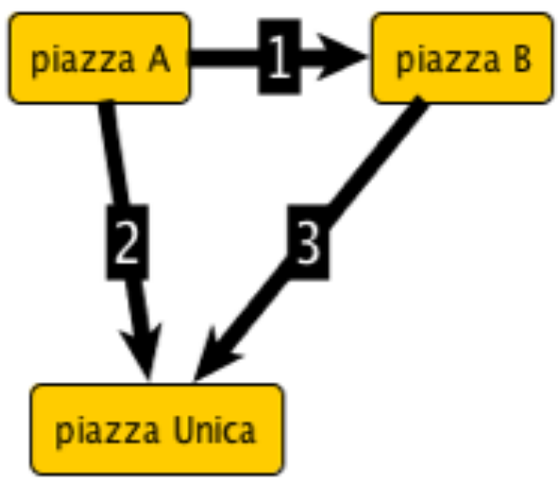
- Sia (x, y) la prima strada non asfaltata che incontriamo percorrendo a ritroso il percorso P (questo arco deve esistere perché il percorso P non è interamente coperto dagli archi selezionati dall'algoritmo). Possiamo vedere il cammino P come composto da tre parti: P_x seguito da (x, y) e infine da Py.
- L'unica ragione per cui la strada (x, y) non è stata selezionata è che la piazza x è stata raggiunta dall'algoritmo selezionando una strada alternativa (x, y') con y' connesso a piazza unica tramite un cammino $P_{y'}$ di strade tutte asfaltate.
- Sia allora P' il percorso composto dal tratto P_x, seguito dalla strada (x, y') e infine da $P_{y'}$. Nota che P' connette z a $Unica$ ed ha una strada non asfaltata in meno.



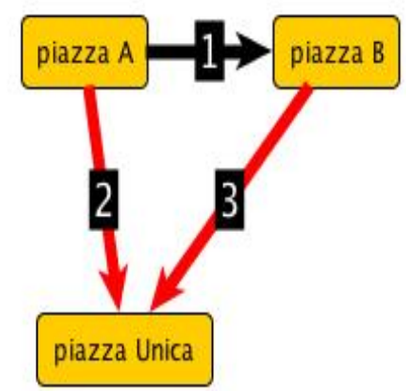
L'algoritmo non è corretto (non produce necessariamente la soluzione di costo minimo)

Si consideri il seguente controesempio:

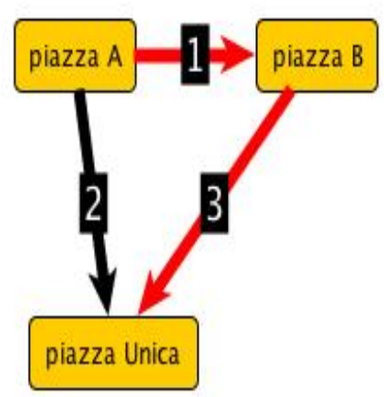
$G =$



Soluzione di costo 5 prodotta dall'algoritmo:



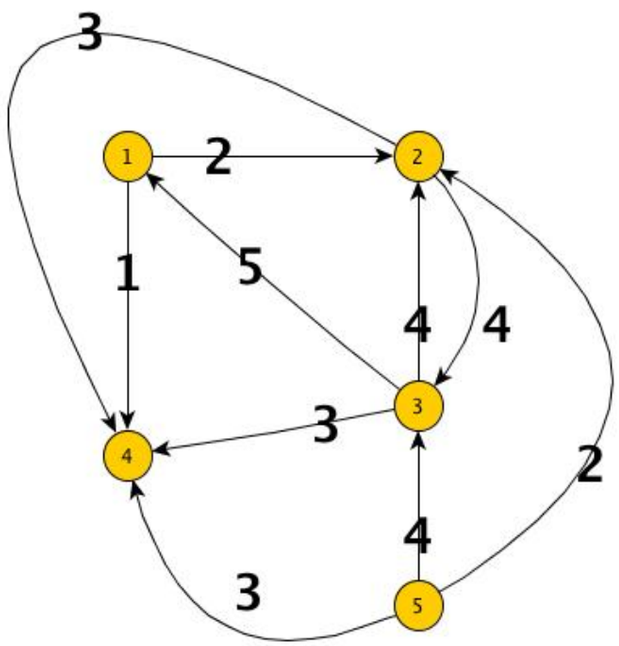
Soluzione ottima di costo 4:

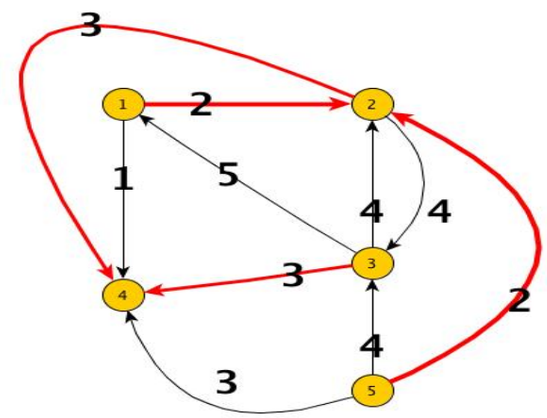


Esercizio.

In un grafo diretto e pesato G un sottoinsieme A degli archi è detto univoco se non contiene 2 o più archi uscenti dallo stesso nodo. Il peso dell'insieme univoco è dato dalla somma dei pesi dei suoi archi.

G=

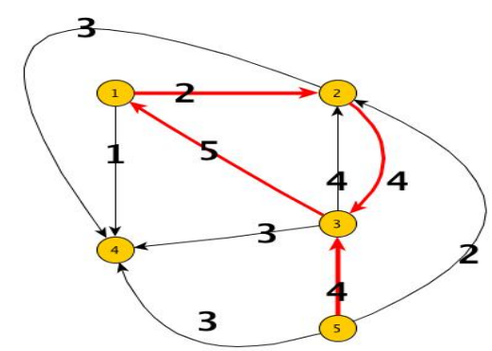




In rosso un insieme univoco di G di peso 10

Problema. Progettare un algoritmo $greedy$ che, dato un grafo G diretto e pesato, in tempo $O(n+m)$ restituisce il peso e gli elementi di un insieme univoco di peso massimo di G. Motivare la correttezza e la complessità dell'algoritmo proposto.

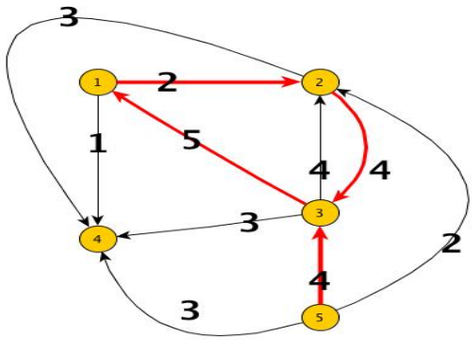
Insieme univoco di peso massimo 15 per G



L'algoritmo *greedy* più naturale sembra essere quello che considera i nodi del grafo in sequenza e per ogni nodo seleziona l'arco di peso massimo che parte da quel nodo.

```
def univoco(G):
    tot, Sol = 0, set()
    for u in G:
        if G[u]!=[]:
            y,c = max(G[u], key= lambda x: x[1])
            Sol.add((u,y))
            tot += c
    return tot, Sol


G={
1:[(2,2),(4,1)],
2:[(4,3),(3,4)],
3:[(1,5),(2,4),(4,3)],
4:[],
5:[(2,2),(3,4),(4,3)]
}
>>> univoco(G)
15, {(1, 2), (3, 1), (2, 3), (5, 3)}
```



- Il costo del for è $O(n)$ se calcolo a parte il costo delle esecuzioni della funzione max.
- Ciascuna esecuzione di max costa $O(m')$ dove m' è il numero di archi presente nella lista su cui max viene applicata. Il costo totale delle esecuzioni di max è dunque $O(m)$

La complessità dell'implementazione è $O(n+m)$.

L'algoritmo *greedy* è corretto

Prova: Assumiamo (per assurdo) che la soluzione Sol prodotta dall'algoritmo *greedy* non sia ottima. Sia Sol^* una soluzione ottima che differisce da Sol nel minor numero di archi. Faremo vedere che esiste una soluzione Sol' ottima che differisce meno da Sol.

- nota che per ogni nodo x che ha archi uscenti l'algoritmo *greedy* seleziona un arco e lo stesso fa la soluzione ottima che, in caso contrario potrebbe essere incrementata. Quindi Sol e Sol^* hanno lo stesso numero di archi.
- esiste un nodo x per cui in Sol^* troviamo un arco (x, y) mentre l'algoritmo *greedy* ha scelto di prendere un altro arco (x, y') (questo nodo deve esistere perché SOL^* e Sol differiscono ma hanno lo stesso numero di archi).
- deve aversi $peso(x, y') \geq peso(x, y)$ (questo è vero per la scelta *greedy* dell'algoritmo
- consideriamo ora l'insieme dominante che si ottiene da Sol^* scambiando l'arco (x, y) con l'arco (x, y'). L'insieme risultante è ancora dominante perché per ogni nodo risulta selezionato al più un arco. Anche il peso della soluzione non è cambiato in quanto avrebbe solo potuto aumentare ma Sol^* ha peso massimo. Ne deriva che Sol' è una nuova soluzione ottima che differisce meno da Sol.

ESERCIZIO: All'acquisto degli n biglietti per un'importante prima teatrale sono interessati m gruppi di persone.
Ciascun gruppo ha intenzione di acquistare i biglietti solo se questi sono a sufficienza per tutti i membri del gruppo.
Vogliamo selezionare i gruppi a cui vendere i biglietti in modo da massimizzare il numero di biglietti venduti.

Per risolvere il problema ci viene proposto il seguente algoritmo che

- prende in input il numero n di posti disponibili e la lista $lista$ di m elementi dove in $lista[i]$ c'è un numero minore o uguale a n che rappresenta i biglietti richiesti dal gruppo i
- restituisce il numero massimo di biglietti che è possibile vendere e l'insieme dei gruppi a cui venderli.

```
def seleziona(n,lista):
    gruppi=[(l,i) for i,l in enumerate(lista)]
    gruppi.sort(reverse=True)
    tot, Sol = 0, set()
    for l,i in gruppi:
        if tot + l <= n:
            Sol.add(i)
            tot+= l
    return tot, Sol
```

1. provare che l'algoritmo non è corretto
2. provare che l'algoritmo ha un rapporto d'approssimazione limitato da 2

ESERCIZIO: All'acquisto degli n biglietti per un'importante prima teatrale sono interessati m gruppi di persone.
Ciascun gruppo ha intenzione di acquistare i biglietti solo se questi sono a sufficienza per tutti i membri del gruppo.
Vogliamo selezionare i gruppi a cui vendere i biglietti in modo da massimizzare il numero di biglietti venduti.

Per risolvere il problema ci viene proposto il seguente algoritmo che:

- **prende in input il numero n di posti disponibili e la $lista$ di m elementi dove in $lista[i]$ c'è il numero di biglietti richiesti dal gruppo i (ovviamente $lista[i] \leq n$).**
- **restituisce il numero massimo di biglietti che è possibile vendere e l'insieme dei gruppi a cui venderli.**

```
def seleziona(n,lista):
    gruppi=[(l,i) for i,l in enumerate(lista)]
    gruppi.sort(reverse=True)
    tot, Sol = 0, set()
    for l,i in gruppi:
        if tot + l <= n:
            Sol.add(i)
            tot+= l
    return tot, Sol
```

1. provare che l'algoritmo non è corretto
2. provare che l'algoritmo ha un rapporto d'approssimazione limitato da 2

```python
def seleziona(n,lista):
    gruppi=[(l,i) for i,l in enumerate(lista)]
    gruppi.sort(reverse=True)
    tot, Sol = 0, set()
    for l,i in gruppi:
        if tot + l <= n:
            Sol.add(i)
            tot+= l
    return tot, Sol
```

1. L'algoritmo non è corretto

per dimostrarlo basta considerare il seguente controesempio dove il teatro ha una capienza di $n = 100$ posti si hanno 3 gruppi di $51,\ 50$ e 50 elementi, rispettivamente.

- L'algoritmo restituisce $tot = 51$ e l'insieme $\{0\}$
- La soluzione ottima è $tot = 100$ e l'insieme $\{1,2\}$

L'esempio mostra anche che il rapporto d'approssimazione è almeno $\frac{100}{51}$>1.9

L'algoritmo ha un rapporto d'approssimazione limitato da 2

Prova:

- dobbiamo dimostrare che per qualunque istanza I vale $\frac{OTT(I)}{A(I)} \leq 2$ dove $OTT(I)$ è il numero massimo di biglietti che è possibile vendere ai gruppi mentre $A(I)$ è il numero di biglietti venduti con la soluzione prodotta dal *greedy*.

- Ordiniamo i gruppi per dimensione decrescente e siano $d_1, d_2, \ldots d_m$ le dimensioni di questi gruppi. Se $\sum_i^n d_i \leq n$ l'agoritmo greedy permetterà di vendere tutti i biglietti, non sbaglia, e quindi si ha $\frac{OTT(I)}{A(I)} = 1 < 2$.

- Assumiamo dunque che $\sum_i^n d_i > n$. Dimostreremo che sotto questa assunzione si ha sempre $A(I) > \frac{n}{2}$.
  - sia $j > 1$ il primo gruppo che con l'algoritmo greedy non riceve i biglietti (questo gruppo deve esistere perchćon $\sum_i^n d_i > n$ non tutti i gruppi possono essere serviti ma il primo gruppo può sempre essere servito) distinguiamo ora due casi:
    * $d_j > \frac{n}{2}$ in questo caso anche $d_1 > \frac{n}{2}$ perché i gruppi sono stati ordinati per dimensione decrescente quindi la vendita dei biglietit al primo gruppo sono sufficienti a garantire $A(I) > \frac{n}{2}$.
    * $d_j \leq \frac{n}{2}$ in questo caso i posti ancora disponibili sono meno di $\frac{n}{2}$ e quindi ancora una volta $A(I) > \frac{n}{2}$

- possiamo ora concludere la prova notando banalmente che $OTT(I) \leq n$ da cui segue $\frac{OTT(I)}{A(I)} < \frac{n}{n/2} = 2$.

ESERCIZIO: **Dato un grafo connesso e pesato G, un suo nodo s ed un intero $k \leq n$ vogliamo trovare un sottografo aciclico di G che comprende il nodo s e contiene k nodi e sia di costo minimo. (nota che quando $n = k$ il problema diviene quello di trovare il minimo albero di copertura di G)**

Viene proposto il seguente algoritmo $greedy$**:**

```
def copertura(G, s, k) :
  A, T = {s}, {x : [ ]}
  for _ in range(k) :
     sia (x, y) un arco di costo minimo con x in A e y not in A
     T[x].add(y)
     T[y] = [x]
     A.add(y)
return T
```

Provare che:

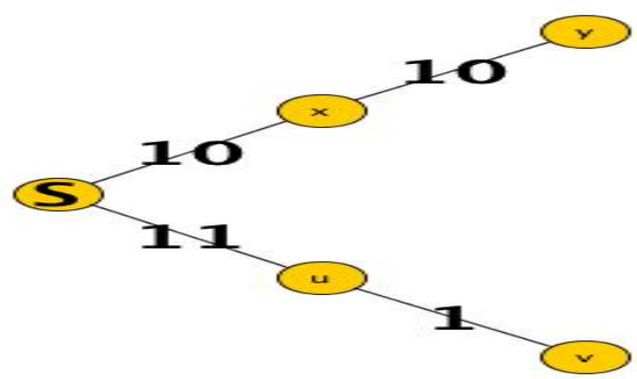
1. **l'algoritmo sbaglia**
2. **l'algoritmo non ha un rapporto d'approssimazione costante**

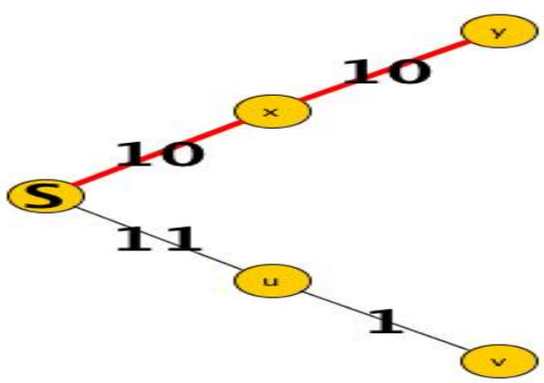
Ad ogni passo l'algoritmo seleziona l'arco di costo minimo che permette di aggiungere un nuovo nodo all'albero che si sta creando a partire dal nodo s

L'algoritmo sbaglia:

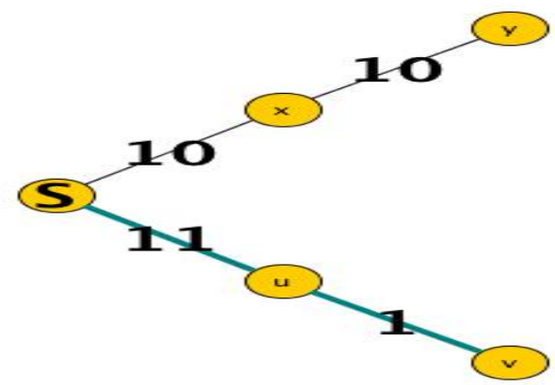
come controesempio considera $k = 3$ ed il seguente grafo G di 5 nodi:

$G =$





Soluzione di costo 20 prodotta dal $greedy$



Soluzione ottima di costo 12

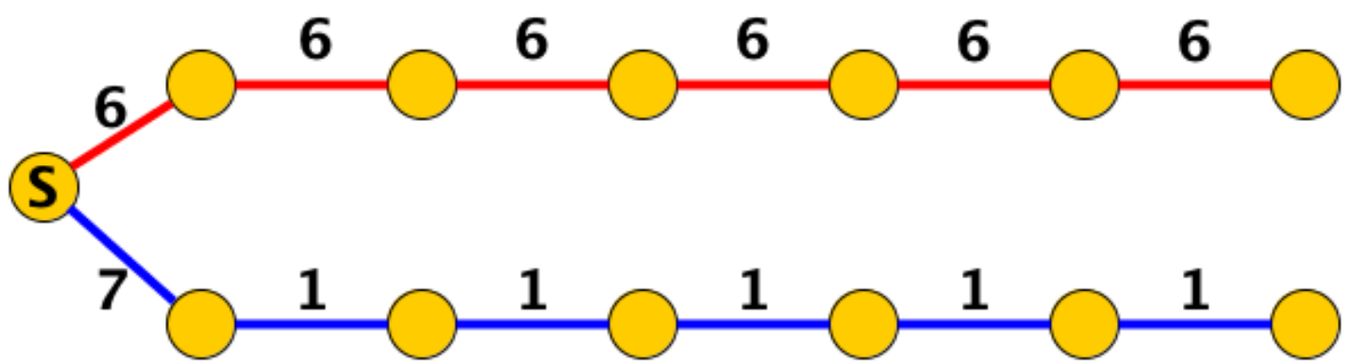
L'algoritmo ha un rapporto d'approssimazione di almeno $\frac{20}{12} > 1.6$

Provare che l'algoritmo non ha un rapporto d'approssimazione costante.

Basterà esibire, per ogni costante l un grafo G_l su cui il rapporto d'approssimazione è l.

Soluzione: **Considera il comportamento dell'algoritmo sul grafo G_l composto da due cammini di $2l$ nodi ciascuno e che si diramano a partire dal nodo s. Ciascun arco del primo cammino ha costo $2l$ mentre dei $2l$ archi del secondo cammino il primo ha costo $2l+1$ e gli altri $2l-1$ archi costo 1. Ad esempio in figura è riportato G_3 con i due cammini in rosso e blu.**

$G_3 =$



Con $k=7$ sul grafo G_3 l'euristica prenderà gli archi del primo cammino di costo 36 mentre la soluzione ottima sceglie gli archi del secondo cammino di costo 12. Il rapporto d'approssimazione in questo caso sarà $\frac{36}{12} = 3$

In generale: con $k = 2l+1$ e grafo G_l l'euristica prenderà gli archi del primo cammino di costo $2l \cdot 2l = 4l^2$ mentre la soluzione ottima sceglie gli archi del secondo cammino di costo $(2l+1)+(2l-1) = 4l$.

Il rapporto d'approssimazione in questo caso sarà: $\frac{A((G_l,\ s,\ 2l+1))}{OTT((G_l,\ s,\ 2l+1))} = \frac{4l^2}{4l} = l$

ESERCIZIO:

Definiamo sottosequenza di una sequenza S una qualunque sequenza che si ottiene da S eliminando un numero arbitrario di elementi.

Ad esempio, per $S = 1011001$

- 01101 e 10001 sono sottosequenze di S (ottenute da _011_01 e 10__001)
- 00110 e 01010 non sono sottosequenze di S

Problema: date due sequenze binarie A e B vogliamo trovare una sottosequenza di lunghezza massima comune ad entrambe.

Ad esempio:

per A=01101 e B=110100 una sottosequenza comune di lunghezza massima è 1101

Viene proposto il seguente algoritmo dove si assume che le sequenze sono rappresentate tramite liste.

```
def comune(A,B):
    za,zb=A.count(0),B.count(0)
    ua,ub=len(A)-za,len(B)-zb
    z=min(za,zb)
    u=min(ua,ub)
    if z>u: return [0]*z
    return [1]*u
```

1. trovare un controesempio che fa sbagliare l'algoritmo con un rapporto d'approssimazione di 2
2. dimostrare che il rapporto d'approssimazione dell'algoritmo è 2.

```python
def comune(A,B):
    za,zb=A.count(0),B.count(0)
    ua,ub=len(A)-za,len(B)-zb
    z=min(za,zb)
    u=min(ua,ub)
    if z>u: return [0]*z
    return [1]*u
```

L'algoritmo sbaglia di almeno 2

L'algoritmo restituisce una sottosequenza comune di tutti 0 o una sottosequenza comune di tutti 1 a seconda di quale delle due può essere più lunga.

IDEA: prendiamo due stringhe uguali contenenti lo stesso numero di 0 e 1

Ad esempio: A=B=111000

- **l'algoritmo restituisce la sottosequenza 111**
- **la sottosequenza comune più lunga è 111000**

per il rapporto d'approssimazione di questa istanza si ha dunque: $\frac{OPT(I)}{A(I)} = \frac{6}{3} = 2$

```python
def comune(A,B):
    za,zb=A.count(0),B.count(0)
    ua,ub=len(A)-za,len(B)-zb
    z=min(za,zb)
    u=min(ua,ub)
    if z>u: return [0]*z
    return [1]*u
```

Il rapporto d'approssimazione dell'algoritmo è limitato da 2

Prova, considera una qualunque istanza I:

- $A(I) = max(u, z)$ (perché viene restituita la sottosequenza comune di massima lunghezza tra quella di tutti 1 e quella di tutti 0)
- $OPT(I) \leq u + z$ (perché la sottosequenza massima comune non può contenere più 1 della sottosequenza massima di soli 1 ne contenere più 0 della sottosequenza massima di soli 0)

Pertanto per il rapporto d'approssimazione si ha:

$$\frac{OPT(I)}{A(I)} \leq \frac{u+z}{max(u,z)} \leq \frac{max(u,z) + max(u,z)}{max(u,z)} = \frac{2 \cdot max(u,z)}{max(u,z)} = 2$$