Corso di laurea in Informatica
Progettazione d'algoritmi

Problemi di ottimizzazione e algoritmi di approssimazione

Angelo Monti

SAPIENZA
UNIVERSITÀ DI ROMA

I problemi di ottimizzazione.

- Un problema di ottimizzazione è un tipo di problema in cui l'obiettivo è trovare la migliore soluzione possibile tra un insieme di soluzioni ammissibili.

- Ogni soluzione ammissibile, cioè una soluzione che soddisfa tutte le condizioni imposte dal problema, ha un valore associato che può essere un "costo" o un "beneficio".

- A seconda del tipo di problema, l'obiettivo può essere minimizzare questo valore (per esempio, ridurre i costi) o massimizzarlo (per esempio, ottenere il massimo guadagno). Abbiamo quindi problemi di minimizzazione e problemi di massimizzazione.

Esempi:

- consideriamo il **problema dello spanning tree** (albero di copertura minimo) su un grafo pesato. Una soluzione ammissibile in questo caso è un albero di copertura, ovvero un sottoinsieme di archi che connette tutti i nodi del grafo senza formare cicli. La soluzione ottima è l'albero di copertura che ha il costo minimo, cioè la somma dei pesi degli archi è la più bassa possibile tra tutti gli alberi di copertura. In questo caso, trovare una soluzione ottima è un problema che può essere risolto in tempo polinomiale (ad esempio, con l'algoritmo di Kruskal).

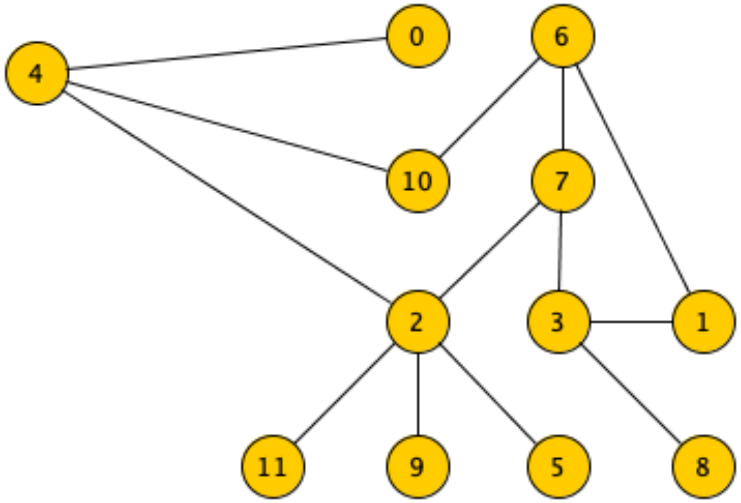
- Un altro esempio di problema di ottimizzazione è la **ricerca del cammino minimo** tra due nodi a e b in un grafo pesato. Una soluzione ammissibile è un cammino che collega a e b attraverso una sequenza di archi del grafo. La soluzione ottima, invece, è il cammino che ha il costo minimo, cioè la somma dei pesi degli archi del cammino è la più bassa possibile. Anche in questo caso, trovare una soluzione ottima è un problema che può essere risolto in tempo polinomiale (ad esempio, con l'algoritmo di Dijkstra).

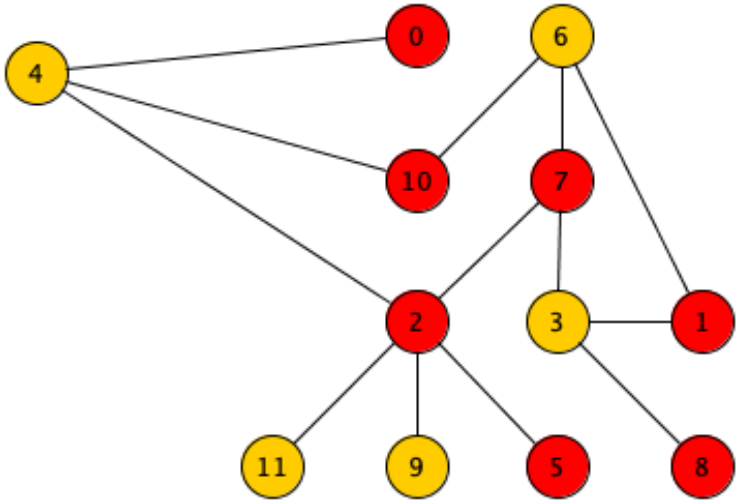
E' importante sottolineare che, sebbene in questi due esempi la complessità di trovare la soluzione ottima sia polinomiale, nella maggior parte dei problemi di ottimizzazione trovare una soluzione ottima risulta essere un compito molto più difficile. In molti casi, infatti, trovare una soluzione ottima può essere un problema NP-difficile, dove la complessità cresce esponenzialmente con la dimensione del problema. In pratica, sebbene determinare se una soluzione è ammissibile possa essere fatto in tempo polinomiale, trovare la soluzione ottima richiede, nella maggior parte dei casi, algoritmi molto più complessi e intensivi in termini di risorse computazionali.

Algoritmi di approssimazione

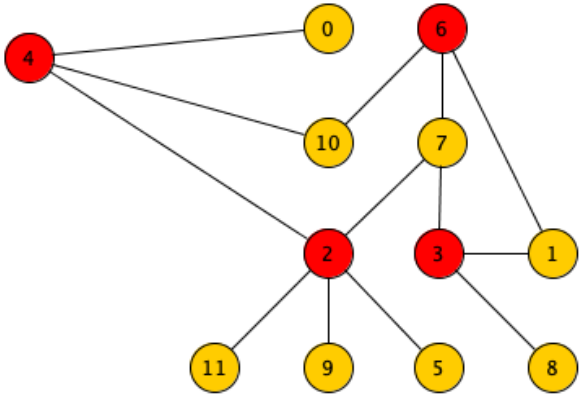
Dato un grafo non diretto G una sua copertura tramite nodi è un sottoinsieme S dei suoi nodi tale che tutti gli archi di G hanno almeno un estremo in S.

$G =$





Il problema di ottimizzazione della copertura tramite nodi:
dato un grafo non diretto G trovare una copertura tramite nodi di minima cardinalità.



Una semplice strategia *greedy* per il problema:

Finché ci sono archi non coperti inserisci in S il nodo che copre il **massimo** numero di archi ancora scoperti.

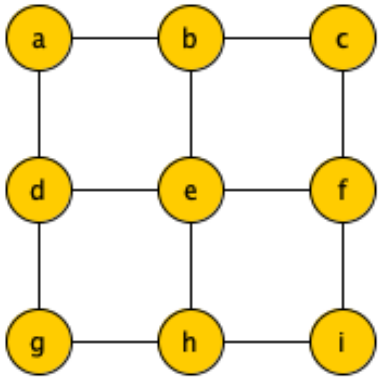
```
def VC(G):
    S = [ ]
    while ci sono in G archi non coperti dai nodi in S:
        prendi il nodo u in G che copre il numero massimo di archi non coperti
        S.append(u)
    return S
```

Una semplice strategia *greedy* per il problema:

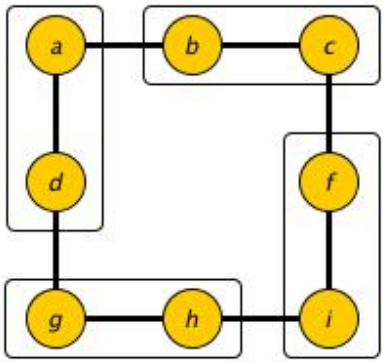
Finché ci sono archi non coperti inserisci in S il nodo che copre il **massimo** numero di archi ancora scoprire.

L'algoritmo non è corretto:

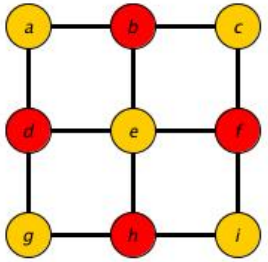
$G =$



Sul grafo G al primo passo verrebbe inserito in S il nodo e (l'unico a coprire 4 archi) dopodiché tutti i nodi restanti sono in grado di coprire lo stesso numero di archi e nei passi successivi almeno un nodo per ogni coppia evidenziata nella figura in basso deve essere scelto.
La soluzione prodotta consiste di 5 nodi.



Soluzione ottima di 4 nodi:



- Ci sono moltissimi problemi di ottimizzazione come Copertura tramite Nodi che sono computazionalmente difficili.

- Per questi problemi non si conoscono algoritmi neanche lontanamente efficienti (sostanzialmente per questi problemi sono noti solo algoritmi esponenziali).

- In questi casi potrebbe essere già soddisfacente ottenere una soluzione **ammissibile** che sia soltanto "vicina" ad una soluzione ottima e, ovviamente, più è vicina e meglio è.

- Fra gli algoritmi che non trovano sempre una soluzione ammissibile ottima, è importante distinguere due categorie piuttosto differenti: Algoritmi di approssimazione ed Euristiche.

- Gli algoritmi di approssimazione sono algoritmi per cui si dimostra che la soluzione ammissibile prodotta ha una certa "vicinanza" alla soluzione ottima. In altre parole, è garantito che la soluzione prodotta approssima entro un certo grado una soluzione ottima.

- Le euristiche sono algoritmi per cui non si riesce a dimostrare che la soluzione ammissibile prodotta ha sempre una certa vicinanza ad una soluzione ottima. Però, sperimentalmente sembrano comportarsi bene. Sono l' ultima spiaggia, quando non si riesce a trovare algoritmi corretti efficienti né algoritmi di approssimazione efficienti che garantiscano un buon grado di approssimazione.

Per una gran parte dei problemi computazionalmente difficili non solo non si conoscono algoritmi corretti efficienti ma neanche buoni algoritmi di approssimazione. Non è quindi sorprendente che fra tutti i tipi di algoritmi, gli algoritmi euristici costituiscano la classe più ampia e che ha dato luogo ad una letteratura sterminata.

Un algoritmo di approssimazione per un dato problema è un algoritmo per cui si dimostra che la soluzione prodotta approssima sempre entro un certo grado una soluzione ottima per il problema. Si tratta quindi di specificare cosa si intende per "approssimazione entro un certo grado".
Iniziamo con problemi di minimizzazione (come ad esempio il problema della copertura tramite vertici) dove ad ogni soluzione ammissibile è associato un costo e cerchiamo quindi la soluzione ammissibile di costo minimo.

- Il modo usuale di misurare il grado di approssimazione è il rapporto *al caso pessimo* tra il costo della soluzione prodotta dall'algoritmo e il costo della soluzione ottima:

Più formalmente:
- Si dice che A approssima il problema di minimizzazione entro un fattore di approssimazione ρ se *per ogni istanza* I del problema vale

$$\frac{A(I)}{OTT(I)} \leq \rho$$

dove, con $OPT(I)$ il costo di una soluzione ottima per l'istanza I e con $A(I)$ il costo della soluzione prodotta dall'algoritmo A per quell'istanza.

Per problemi di massimizzazione dove ad ogni soluzione ammissibile è associato un valore si considera il rapporto inverso vale a dire $\frac{OTT(I)}{A(I)}$

- Si dice che A approssima il problema di minimizzazione entro un fattore di approssimazione ρ se *per ogni istanza* I di P vale $\frac{A(I)}{OTT(I)} \leq \rho$.

- Nota che, trattandosi di un problema di minimizzazione, risulta sempre $A(i) \geq OTT(I)$, di conseguenza il rapporto di approssimazione ρ è sempre un numero maggiore o uguale ad 1.

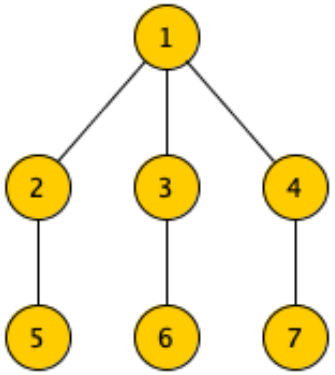
  - Se A approssima P con fattore 1 allora A è corretto per P perché trova sempre una soluzione ottima.

  - Se A approssima P entro un fattore 2 allora A trova sempre una soluzione di costo al più doppio di quello della soluzione ottima.

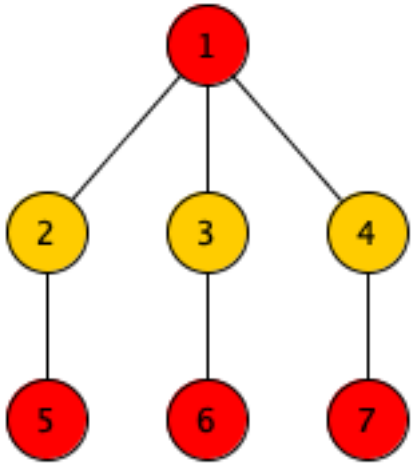
  - Ovviamente quanto più il rapporto d'approssimazione è vicino ad 1 tanto più l'algoritmo d'approssimazione è buono.

Proviamo a valutare il rapporto d'approssimazione dell'algoritmo greedy visto per il problema Ricoprimento tramite Nodi:

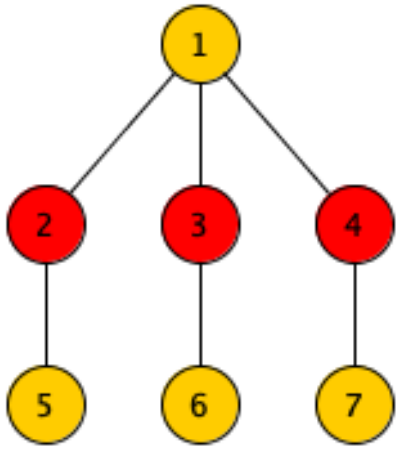
- l'algoritmo greedy che abbiamo visto non è corretto.
- Abbiamo trovato un'istanza per cui l'algoritmo produce una soluzione con 5 nodi mentre la soluzione ottima ha 4 nodi.
- Da ciò possiamo dedurre che il fattore di approssimazione dell'algoritmo è almeno $\frac{5}{4} > 1$.
  Ma potrebbe essere peggiore.

$G =$ 

Il grafo G produce rapporto d'approssimazione di $\frac{4}{3}$ (che è maggiore di $\frac{5}{4}$).



greedy



ottimo

In effetti per ogni costante R si possono esibire grafi per cui l'algoritmo sbaglia di un fattore superiore ad R

Quindi l'algoritmo *greedy* in esame non garantisce nessun fattore di approssimazione costante.

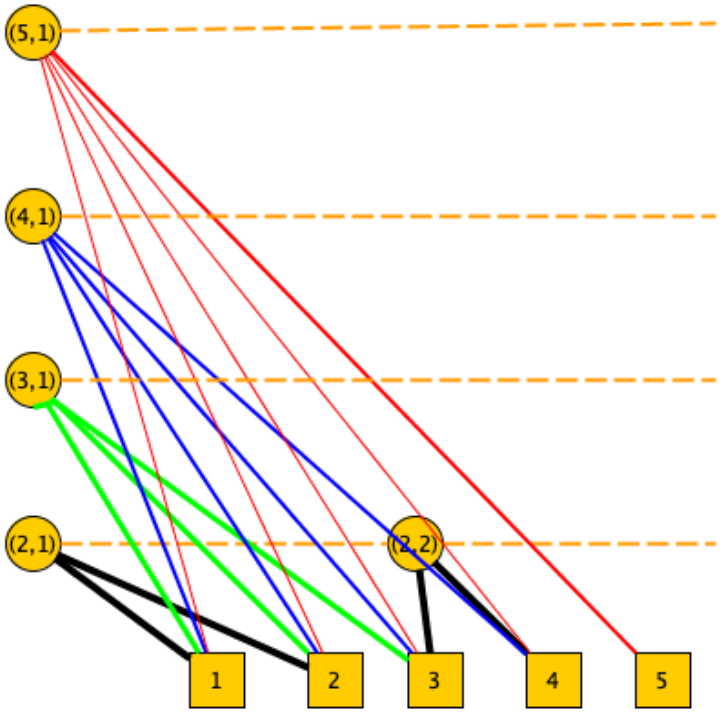
L'algoritmo *greedy* in esame non garantisce nessun fattore di approssimazione costante.

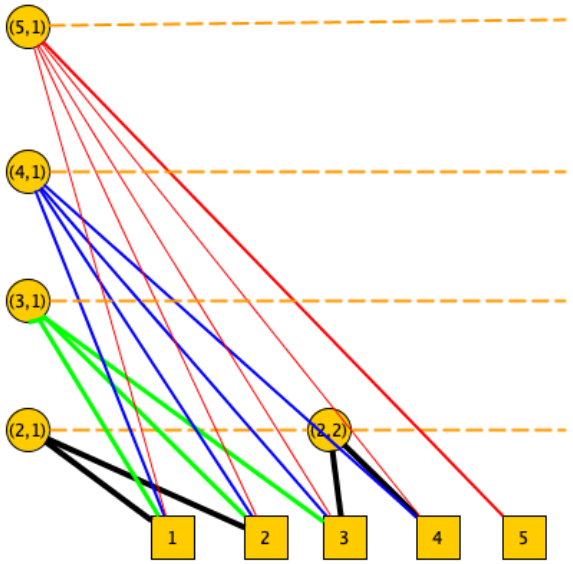
PROVA: Dimostreremo che per ogni intero l possiamo costruire un grafo G_l su cui l'algoritmo *greedy* avrà rapporto di approssimazione $\Omega(\log l)$.

Il grafo G_l è definito come segue:

- il grafo è strutturato in l livelli che vanno da 1 a l e gli archi del grafo collegano nodi a livello $i > 1$ con nodi a livello 1. Più precisamente:
  - a livello i sono presenti $\left\lfloor \frac{l}{i} \right\rfloor$ nodi, ciascuno di questi nodi ha grado i e gli $i \cdot \left\lfloor \frac{l}{i} \right\rfloor \leq l$ archi vanno a nodi distinti del livello 1.

Di seguito viene riportato il grafo G_5





Sul grafo G_l l'euristica funziona come segue:

- il primo nodo ad essere selezionato è il nodo a livello l di grado l (tutti gli altri hanno grado al più $l-1$)
- verranno poi uno dopo l'altro selezionati tutti i nodi a livello $l-1$ di grado $l-1$ (tutti gli altri hanno grado al più $l-2$)
- verranno poi uno dopo l'altro selezionati tutti i nodi a livello $l-2$ di grado $l-2$ (tutti gli altri hanno grado al più $l-3$)
- .....
- verranno infine selezionati tutti i nodi a livello 2 di grado 2 (tutti gli altri, cioè i nodi a livello 1 hanno grado 1)

Per il costo della soluzione prodotta dall'euristica si ha:

$$\begin{aligned} c(SOL) &= \sum_{i=2}^{l} \left\lfloor \frac{l}{i} \right\rfloor \\ &\geq \sum_{i=2}^{l} \left(\frac{l}{i} - 1 \right) \\ &\geq l \int_{2}^{l+1} \frac{1}{x} dx - (l-1) \\ &= l \cdot \log_e(l+1) - l \log_e 2 - l + 1 \\ &\geq \Omega(l \cdot \log l) \end{aligned}$$

Una possibile soluzione al problema è quella di selezionare gli l nodi a livello 1 (infatti per costruzione tutti gli archi incidono su quegli l nodi). Non è difficile dimostrare che questa è anche la soluzione ottima. Possiamo comunque dire $c(SOL^*) = O(l)$.

Abbiamo quindi dimostrato che il rapporto di approssimazione dell'euristica è almeno $\frac{c(SOL)}{c(SOL^*)} = \frac{\Omega(l \cdot \log l)}{O(l)} = \Omega(\log l)$.

Ovviamente il fatto d'aver dimostrato che per un problema un certo algoritmo d'approssimazione ha un cattivo rapporto d'approssimazione non impedisce che per il problema possano esistere altri algoritmi d'approssimazione con un fattore d'approssimazione costante.

Consideriamo il seguente algoritmo *greedy* per la copertura di nodi:
considera i vari archi del grafo uno dopo l'altro e ogni volta che ne trovi uno non coperto (vale a dire nessuno dei suoi estremi è in S) aggiungi entrambi gli estremi dell'arco alla soluzione S.

```
def copertura1(G):
    inizializza la lista E con gli archi di G
    S = [ ]
    while E != [ ]:
        estrai da E un arco (x, y)
        if ne' x ne' y sono in S:
            S.append(x)
            S.append(y)
    return S
```

```
def copertura1(G):
    inizializza la lista E con gli archi di G
    S = [ ]
    while E != [ ]:
        estrai da E un arco (x, y)
        if ne' x ne' y sono in S:
            S.append(x)
            S.append(y)
    return S
```

- L'algoritmo produce ovviamente una copertura (ogni arco verrà infatti esaminato e se risulterà non coperto verrà coperto da entrambi i lati).

- La copertura prodotta non è detto sia minima, l'algoritmo ha rapporto d'approssimazione almeno 2 come dimostra il grafo con due soli nodi ed un arco.

- Dimostreremo che il rapporto d'approssimazione è limitato da 2.

NOTA: non sono noti algoritmi d'approssimazione con rapporto inferiore a 2

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

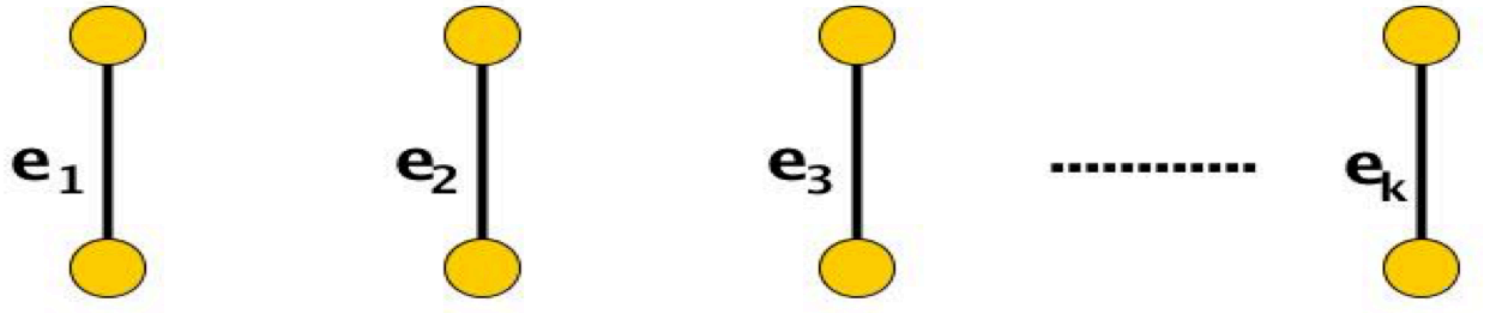
PROVA:
Siano $e_1, e_2, \ldots e_k$ gli archi di G che vengono trovati non coperti durante l'esecuzione dell'algoritmo greedy.

1) per come funziona l'algoritmo deduciamo che $A(I) = 2k$.

i k archi non coperti sono tra loro disgiunti (infatti i due estremi di ciascuno di questi archi vengono incontrati per la prima volta quando viene esaminato l'arco) questo significa che in una qualunque delle soluzioni ottime almeno un estremo di ciascuno di questi k archi deve essere presente.

2) Ne deduciamo che $k \leq OTT(I)$.

dai punti 1) e 2) ricaviamo che $A(I) = 2k \leq 2 \cdot OTT(I)$ da cui segue $\frac{A(I)}{OTT(I)} \leq 2$.



Implementazione dell'algoritmo *greedy*

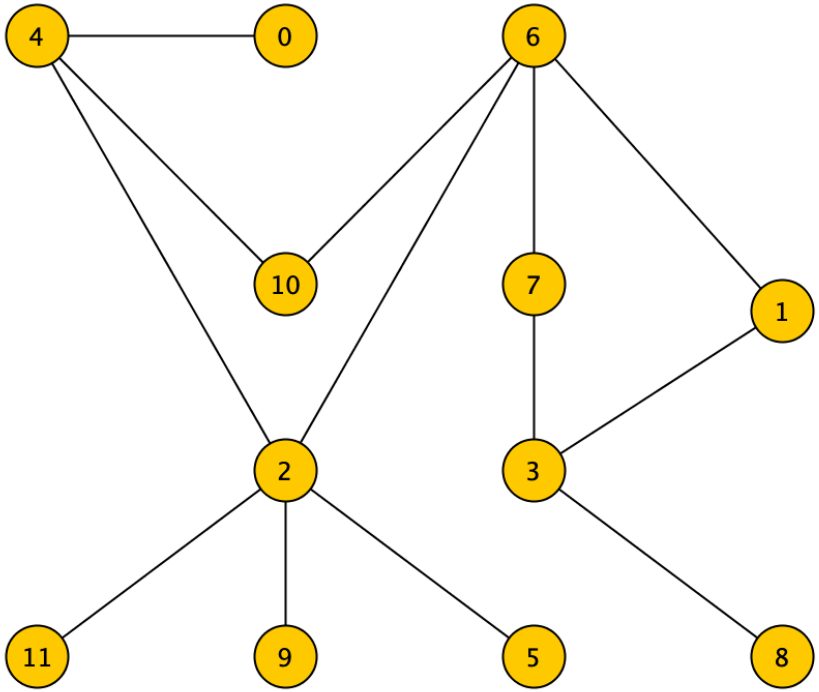
```
def copertura1(G):
    inizializza la lista E con gli archi di G
    S = [ ]
    while E != [ ]:
        estrai da E un arco (x,y)
        if ne' x ne' y sono in S:
            S.append(x)
            S.append(y)
    return S
```

IDEA: per rendere efficiente il test sui nodi in S utilizzo un vettore caratteristico *presi* dove $presi[i] = 1$ se il nodo i è in S, 0 altrimenti.

```
def copertura1(G):
    n = len(G)
    E =[(x,y) for x in range(n) for y in G[x] if x < y]
    presi = [0] * n
    Sol = [ ]
    for a,b in E:
        if presi[a] == presi[b] == 0:
            Sol.append(a)
            Sol.append(b)
            presi[a] = presi[b] = 1
    return Sol
```

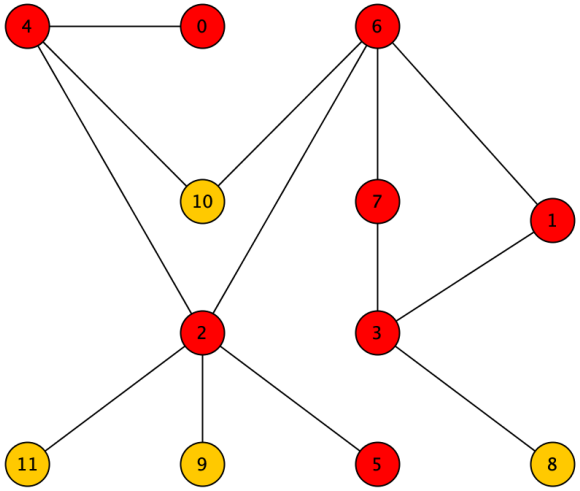
La complessità dell'algoritmo è $O(n+m)$

$G =$




```
>>>G=[
     [4],[3,6],[4,5,7,9,11],
     [1,7,8],[0,2,10],[2],
     [1,7,10],[3,6],[3],
     [2],[4,6],[2]
     ]
>>> copertura1(G)
[0, 4, 1, 3, 2, 5, 6, 7]
```

Soluzione prodotta=



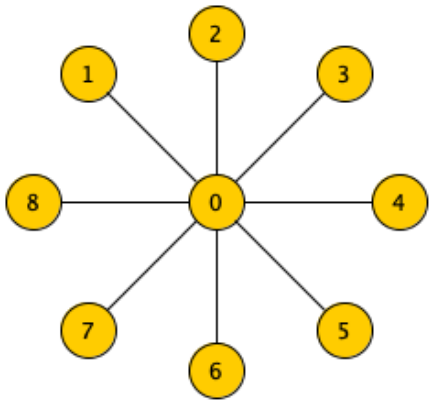
ESERCIZIO: per il problema della copertura tramite nodi viene proposto il seguente algoritmo *greedy*:

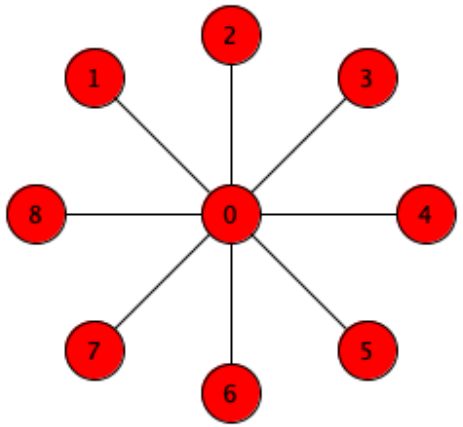
```
def VC(G):
    return [ x for x in range( len(G) ) ]
```

Provare che l'algoritmo non è corretto ed ha un rapporto d'approssimazione $\Omega(n)$.

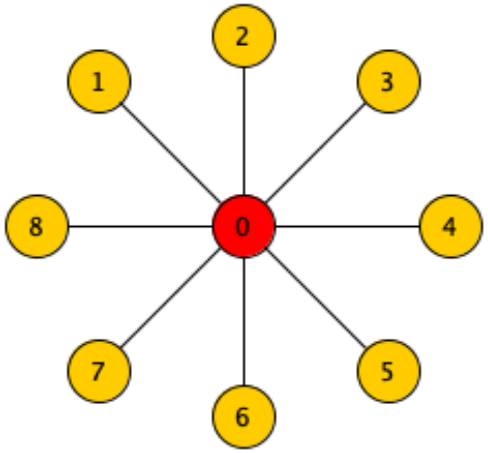
Soluzione: Considera il comportamento dell'algoritmo sul grafo a "stella" G_n di n nodi con al centro il nodo 0 e $n-1$ raggi.
Ad esempio in figura è riportato G_9

$G_9 =$





greedy



Ottima

$$\frac{A(I)}{OTT(I)} = \frac{n}{1} = \Omega(n)$$

Corso di laurea in Informatica
Introduzione agli Algoritmi

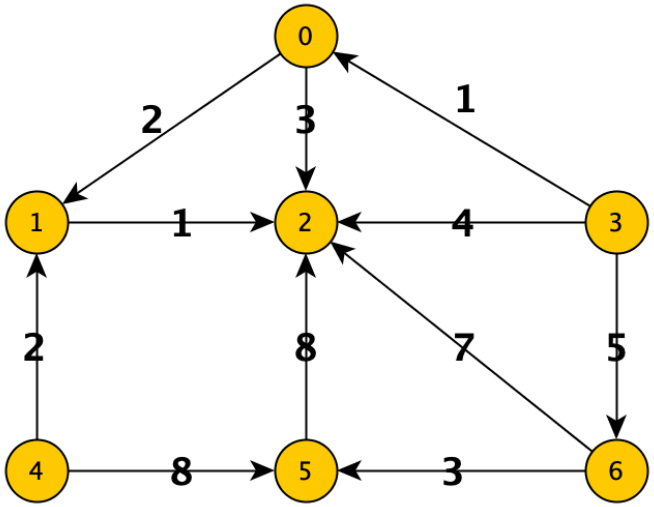
Esercizi per casa

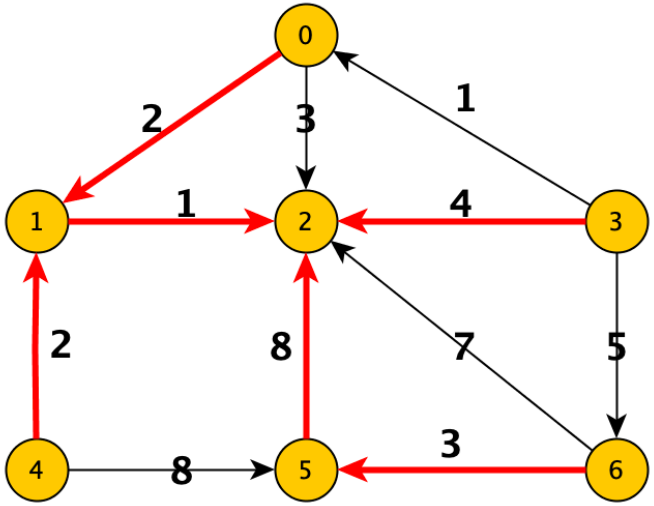
SAPIENZA
UNIVERSITÀ DI ROMA

Esercizio 1.

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=



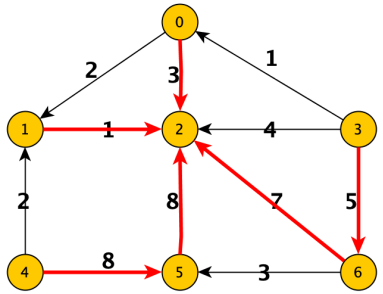


In rosso un insieme univoco di G di peso 20

Progettare un algoritmo *greedy* che, dato un grafo G diretto e pesato, in tempo $O(n+m)$ restituisce il peso e gli archi di un insieme univoco di peso massimo di G.

Motivare la correttezza e la complessità dell'algoritmo proposto.

Insieme univoco di peso massimo 32 per G



Esercizio 2.

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 greedy che

- prende in input il numero n di posti disponibili e la 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 la lista dei gruppi a cui venderli.

```
def selezione(lista, n):
    gruppi=[(lista[i],i) for i in range(len(lista))]
    gruppi.sort(reverse=True)
    tot, Sol=0, []
    for l,i in gruppi:
        if tot+l<=n:
            Sol.append(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 3.

Dato un grafo connesso e pesato G, ed un intero $k \leq n$ vogliamo trovare un sottografo aciclico di G che comprende il nodo 0, 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* che preso il grafo G e l'intero k restituisce i nodi del sottografo aciclico ed il suo costo:

```
def copertura(G,k):
    A,costo= [0],0
    E=[(c,i,y) for i in range(len(G)) for y in G[i] ]
    for _ in range(k):
        E1=[(c,x,y) for c,x,y in E if x in A and y not in A]
        c,x,y= min(E1)
        costo+=c
        A.append(y)
    return A, costo
```

Provare che:

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

ESERCIZIO 4.

Sia S una sequenza binaria, una sottosequenza di S si ottiene eliminando da S un numero arbitrario di caratteri.

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:

```
def comune( A, B):
    z = min( A.count(0), B.count(0))
    u = min( A.count(1), B.count(1))
    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.**

ESERCIZIO 5.
Dato un grafo G=(V,E) vogliamo assegnare ad ogni nodo un colore nell'insieme $\{1,2, \ldots n\}$ in modo tale che nodi adiacenti ricevano colori distinti e che i colori distinti usati sia minimo.

Si propone il seguente algoritmo greedy che restituisce il vettore Sol di n componenti dove $Sol[i]$ riporta il colore assegnato al nodo i.

```
def es(G):
    for i in range(n):
        C=[1]*n
        for j in range (1,i-1):
            if j in G[i]: C[sol[j]]=0
        c=1
        while C[c]==0: c+=1
        sol[i]=c
    return sol
```

Provare o confutare che l'algoritmo risolve il problema.