Algoritmi d'Approssimazione

Algoritmi 2 – Lezione 11

A. Monti
Sapienza Università di Roma

Corso: Algoritmi 2

Indice

1. Problemi di Ottimizzazione
2. Euristiche ed Algoritmi di Approssimazione
   - Differenza tra euristiche e approssimazione
3. Il Problema della Copertura Minima Tramite Nodi
   - Definizione del problema
   - Approccio Greedy al problema
   - Un'euristica Greedy Basata sul Grado dei Nodi
   - Un Algoritmo d'Approssimazione con Rapporto 2

Problemi di Ottimizzazione

- **Obiettivo:** Trovare la soluzione **migliore** tra un insieme di soluzioni ammissibili.
- **Componenti chiave:**
  - Un insieme di soluzioni ammissibili.
  - Una **funzione obiettivo** (da massimizzare o minimizzare).
  - Una serie di **vincoli**.
- **Esempio:** Il problema del **minimo albero di copertura** (MST).
  - Obiettivo: trovare un albero con la somma dei pesi degli archi più bassa.
  - Esistono algoritmi polinomiali ottimi (es. Kruskal).

Perché servono soluzioni approssimate?

- Per molti problemi, non sono noti algoritmi efficienti che trovino la soluzione ottima.
- La ricerca esaustiva è computazionalmente intrattabile.
- **L'alternativa:** Sviluppare algoritmi che, pur non garantendo la soluzione ottima, la **approssimano** in modo efficiente.
  - **Euristiche** (senza garanzia).
  - **Algoritmi di Approssimazione** (con garanzia matematica).

Euristiche vs. Algoritmi di Approssimazione

Euristiche	**Algoritmi di Approssimazione**
"Regole pratiche" per trovare una buona soluzione.	Algoritmi con una **garanzia matematica** sulla qualità.
Nessuna garanzia sul risultato.	Il risultato è al massimo ρ volte peggiore dell'ottimo ($\rho =$ fattore di approssimazione).

- Le euristiche sono utili nella pratica, ma non offrono garanzie formali.
- Gli algoritmi di approssimazione sono progettati per offrire **garanzie di qualità** della soluzione.

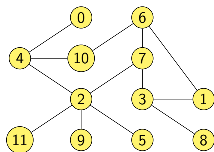
Rapporto di Approssimazione

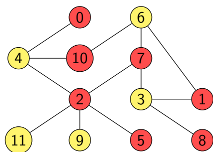
- Per un problema di minimizzazione, il **rapporto di approssimazione** $\rho \geq 1$ è il rapporto tra il costo della soluzione trovata e quello della soluzione ottima nel caso peggiore.
- Per un problema di massimizzazione, ρ è il rapporto tra il costo della soluzione ottima e quello della soluzione trovata.
- Se un algoritmo restituisce sempre la soluzione ottima, allora $\rho = 1$.
- Più ρ si allontana da 1, minore è la qualità della soluzione.
- Se ρ non è limitato da una costante, lalgoritmo si comporta come una euristica.

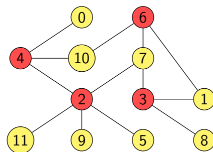
Definizione del problema di Copertura tramite Nodi

Dato un grafo G, una **copertura tramite nodi** è un sottoinsieme dei suoi nodi S tale che ogni arco di G abbia almeno uno dei suoi estremi all'interno di S.
In altre parole S è tale che ogni arco del grafo ha almeno un estremo in S.
L'obiettivo del problema è trovare una copertura tramite nodi con il minor numero di nodi possibile.

In figura, a sinistra abbiamo un grafo G, al centro sono evidenziati in rosso i 7 nodi di una copertura di G e sulla destra sono evidenziati i 4 nodi di una copertura minima.







Strategia Greedy: idea generale

- Questo problema, pur essendo facile da definire, sembra essere intrattabile. Non si conoscono, infatti, algoritmi efficienti in grado di trovare la soluzione ottima per qualsiasi tipo di grafo. Questo motiva l'uso di soluzioni approssimate.

- Un approccio naturale e semplice per risolvere il problema della copertura minima tramite nodi è quello di adottare una strategia greedy. L'idea è, a ogni passo, prendere la decisione che sembra migliore a livello locale, sperando che questa porti a una soluzione ottima a livello globale.

Un'euristica Greedy basata sul grado dei nodi

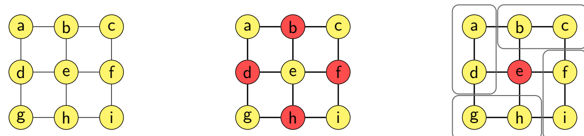
Euristica Greedy (basata sul grado): Aggiungi il nodo che copre il maggior numero di archi.

1. Si parte con un insieme di copertura vuoto.
2. Finché ci sono archi non coperti nel grafo:
    1. Scegli il nodo che copre il maggior numero di archi non ancora coperti.
    2. Aggiungi questo nodo all'insieme di copertura.
    3. Rimuovi tutti gli archi coperti da questo nodo.
3. Quando tutti gli archi sono coperti, l'algoritmo termina.

Perché l'euristica Greedy può fallire

- **Perché fallisce?** Non sempre l'ottimo locale porta a un ottimo globale.

Controesempio:

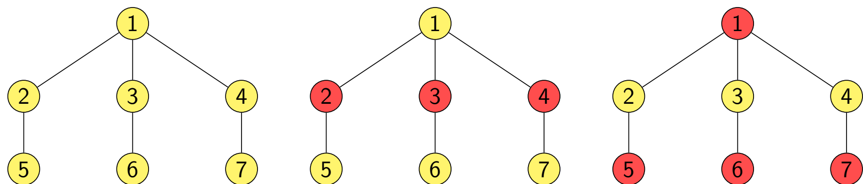


In figura a sinistra un grafo G, al centro la copertura ottima mentre a destra in rosso il primo nodo scelto dall'algoritmo greedy, questa scelta porterà ad una copertura sub-ottimale con 5 nodi.

- L'algoritmo ha un rapporto d'approssimazione $\rho \geq \frac{5}{4}$

Un altro esempio di fallimento: rapporto $\geq \frac{4}{3}$

Un secondo controesempio che accresce il rapporto d'approssimazione portandolo ad almeno $\frac{4}{3} > \frac{5}{4}$ è il seguente:



Per il grafo G di sinistra la copertura ottima è al centro mentre l'algoritmo greedy produce una soluzione con 4 nodi come quella a destra.

- In realtà, si dimostra che il rapporto dapprossimazione può crescere arbitrariamente!

Algoritmo di Approssimazione con Rapporto 2

- Fortunatamente, esistono diversi algoritmi in grado di trovare una soluzione approssimata con un rapporto limitato.
- Il miglior rapporto di approssimazione garantito finora è 2, e ora vedremo uno degli algoritmi che lo ottiene.
- L'algoritmo è un approccio greedy che, a differenza del precedente basato sul grado dei nodi, opera scegliendo direttamente gli archi. L'idea è di selezionare iterativamente un arco non coperto e aggiungere *entrambi* i suoi estremi alla copertura, garantendo così che quell'arco (e potenzialmente altri) venga coperto.

Implementazione in Python

```
def copertura(G):
    ''''
    Restituisce una lista C contenente
    i nodi di una copertura minima per G
    '''
    C = [  ]
    n = len(G)
    E =[(x,y) for x in range(n) for y in G[x] if x < y]
    presi = [0] * n
    for a,b in E:
        if presi[a] == presi[b] == 0:
            C.append(a)
            C.append(b)
            presi[a] = presi[b] = 1
    return C
```

- **Complessità temporale:** $O(n + m)$.

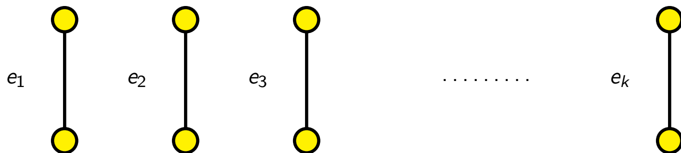
Dimostrazione del Rapporto di Approssimazione 2

Teorema L'algoritmo ha un rapporto di approssimazione $\rho \leq 2$.

- L'algoritmo seleziona k archi disgiunti a coppie.
- La soluzione C ha $2k$ nodi.
- Una soluzione ottima C^* deve coprire tutti i k archi disgiunti. Ogni nodo può coprire al massimo uno di questi archi disgiunti, quindi

$$|C^*| \geq k.$$

- **Conclusione:** $|C| = 2k \leq 2|C^*|$.



Conclusioni

- I problemi di ottimizzazione difficili possono essere risolti con algoritmi di approssimazione.
- Le euristiche non offrono garanzie.
- Per il problema della copertura minima tramite nodi l'algoritmo basato sugli archi è efficiente e garantisce una soluzione al massimo due volte peggiore dell'ottimo.
- Non si conoscono algoritmi con rapporto inferiore a 2; si congettura che non esistano.

ESERCIZI

Suggerimento: Prima di cercare le soluzioni in rete, prova a ragionarci da solo: gli esercizi che seguono sono pensati per aiutarti a mettere alla prova la tua comprensione, la tua capacità di sviluppare una prova di correttezza e costruire un algoritmo corretto.
Solo dopo averci riflettuto, confronta la tua soluzione con altre possibili versioni.

Esercizi

1 Per il problema della **copertura minima tramite nodi**. Viene proposta la seguente euristica greedy.

```
def Copertura(G):
    '''
    Calcola una copertura approssimata del grafo G
    Restituisce la lista dei nodi della copertura
    '''
    n = len(G)
    Copertura = [ ]
    InCopertura = [False]*n
    for u in range(1,n):
        if not InCopertura[u]:
            for v in G[u]:
                if not InCopertura[v]:
                    Copertura.append(u)
                    InCopertura[u] = True
                    break
    return Copertura
```

Sulla base di questa implementazione, svolgete i seguenti compiti:

1. **Dimostrazione di ammissibilità:** Dimostrate che l'algoritmo produce sempre una soluzione ammissibile, ovvero un insieme di nodi che copre tutti gli archi del grafo.
2. **Rapporto di approssimazione:** Dimostrate che l'algoritmo non ha un rapporto di approssimazione limitato, ovvero per ogni costante c posso trovare un grafo per cui il rapporto tra soluzione trovata e soluzione ottima sia maggiore di c.

Esercizi

2 Dato un grafo connesso e pesato G, ed un intero k, con $k \leq n$, vogliamo trovare un sottografo aciclico di G che comprenda il nodo 0, sia connesso, contenga esattamente k nodi e sia di costo minimo (nota che per $k = n$ il problema diviene quello di trovare un minimo albero di copertura di G).
Per ottenere il sottografo cercato viene proposto il seguente algoritmo:

```
def copertura(G, k):
    A = {0}
    EA = []
    if k <= 1:
        return list(A), EA
    E = [(c, u, v) for u in range(len(G)) for v, c in G[u]]
    for _ in range(k - 1):
        E1=[(c,x,y)for c,x,y in E if x in A and y not in A]
        costo_minimo, x, y = min(E1)
        # Aggiungi l'arco alla copertura e il  nodo al set
        # Rappresenta l'arco non orientato in modo univoco
        if x<y:
            EA.append((x,y))
        else:
            EA.append((y,x))
        A.add(y)
    return list(A), EA
```

Dimostrare che Il sottografo formato dai nodi A e dagli archi EA restituiti dall'algoritmo non necessariamente è quello di costo minimo.

Esercizi

3 Siano date due stringhe binarie S_1 e S_2, entrambe di lunghezza n. L'obiettivo è trovare la lunghezza della sottostringa comune più lunga. Per risolvere il problema, viene proposto il seguente algoritmo:

```
def sottostringa_approssimata(s1, s2):
    # 1. Conta il numero minimo di uni ("1") in S1 e S2.
    count_1_s1 = s1.count('1')
    count_1_s2 = s2.count('1')
    min_1 = min(count_1_s1, count_1_s2)

    # 2. Conta il numero minimo di zeri ("0") in S1 e S2.
    count_0_s1 = s1.count('0')
    count_0_s2 = s2.count('0')
    min_0 = min(count_0_s1, count_0_s2)

    # 3. Restituisce il massimo tra i due valori.
    return max(min_1, min_0)
```

1. Mostra, tramite un controesempio, che l'algoritmo proposto non sempre trova la lunghezza della sottostringa comune più lunga.
2. Dimostra che il rapporto di approssimazione dell'algoritmo proposto è limitato da 2.

Esercizi

4 All'acquisto degli n biglietti per unimportante 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 teatro(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.

Suggerimento: comincia col provare che il numero di biglietti venduti è sempre maggiore di $\frac{n}{2}$.