Algoritmi di approssimazione

A.A. 2017-2018

Programmazione lineare
Programmazione dinamica

Nell‘algebra lineare un sistema di equazioni ha la forma

$$A\,x = b$$

Dove A e` una matrice data, b un vettore dato e x il vettore incognito di numeri reali che risolve l'equazione.

Nel caso della programmazione lineare questo sistema viene sostituito da un sistema di disequazioni; si tratta quindi di determinare un vettore x che soddisfi il sistema di disequazioni

$$A\,x \geq b$$

Un tale sistema di disuguaglianze individua nello spazio, delle regioni, che costituiscono l‘insieme delle soluzioni.
La programmazione lineare cerca di minimizzare una combinazione lineare $c^t\,x$ (dove c^t e` un vettore di *coefficienti* e $c^t\,x$ denota il prodotto scalare dei due vettori) delle coordinate di x su tutti gli x che appartengono alla regione.

Possiamo allora scrivere la forma standard della programmazione lineare come problema di ottimizzazione:

Data una matrice A: m $\times$ n e due vettori, $b \in R^m$ e $c \in R^n$, trovare un vettore $x \in R^n$ che risolva il seguente problema di ottimizzazione:

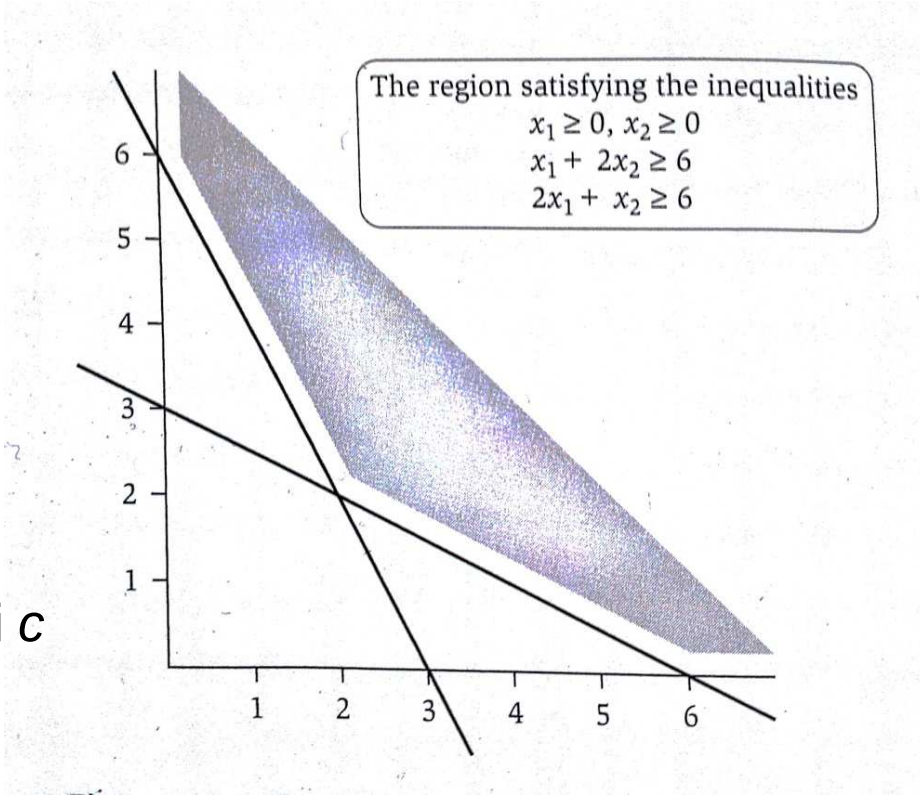
$$min(c^t x \mid x \geq 0 \text{ and } Ax \geq b)$$

$c^t x$ viene chiamata funzione obiettivo e $A x \geq b$ costituisce l'insieme dei vincoli.

Prendiamo ad esempio $x = (x_1,x_2)$ un vettore a due dimensioni e i seguenti vincoli:

$$x_1 \geq 0 \ , \ x_2 \geq 0$$
$$x_1 + 2x_2 \geq 6$$
$$2x_1 + x_2 \geq 6$$

La regione delle soluzioni nel piano cartesiano è identificata nella figura.



Si deve trovare un vettore di coefficenti c tale che la combinazione lineare $c^t x$ soddisfi i vincoli dati.

Nel nostro esempio, supponendo di avere il vettore $c = (1.5, 1)$, dobbiamo minimizzare la quantità $1.5x_1 + x_2$
La soluzione risultante e` il punto $x = (2,2)$ che fornisce un valore di $c^t x = 5$, che si puo` facilmente verificare minimo.

Possiamo rifrasare la programmazione lineare nella forma di problema di decisione:

Data una matrice A, due vettori b e c, e un limite γ
Esiste $x \geq 0$ *tale che* $Ax \geq b$ *e* $c^t x \leq \gamma$?

Per evitare problemi di rappresentazione dei dati, supponiamo che *gli elementi di vettori e matrici siano interi.*

Complessità

Per molto tempo la programmazione lineare è stata il più famoso esempio di problema in ***NP*** e *co-**NP**,* ma nel 1981 Khachiyan trovò un algoritmo polinomiale per la sua risoluzione.
Nonostante ciò, l'algoritmo più utilizzato è ancora il simplesso, che ha complessita` polinomiale nel caso medio e complessità esponenziale nel caso pessimo, in quanto il caso peggiore è difficile che si presenti e nel caso medio il simplesso risulta migliore dell'algoritmo proposto da Khachiyan.

Per i nostri scopi è comunque sufficiente sapere che esistono algoritmi in grado di risolvere il problema in tempo polinomiale

Con il metodo pricing abbiamo costruito un algoritmo approssimato per il problema **copertura di vertici pesata**, con rapporto di approssimazione uguale a 2.

Riprendiamo lo stesso problema per costruire un diverso algoritmo di approssimazione, definito come problema di programmazione lineare, anche questo con rapporto 2.

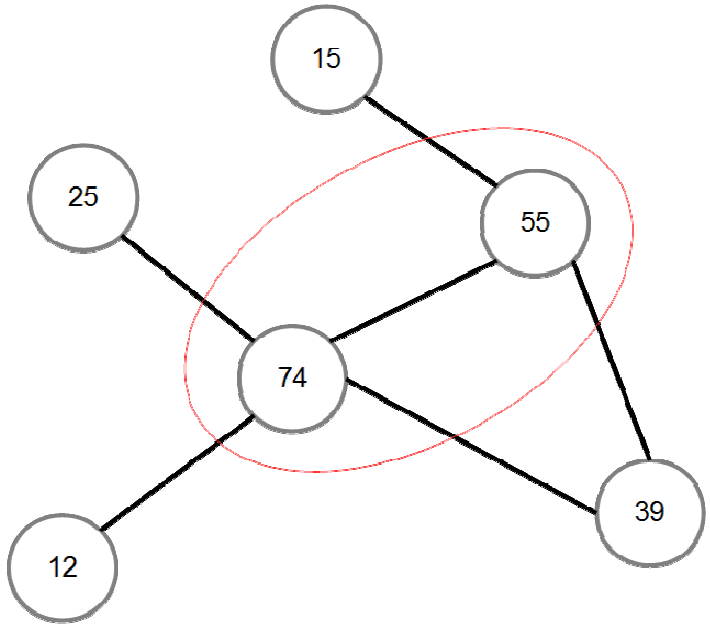
Il problema

Una copertura di vertici S sul grafo G = (V, E) è un sottoinsieme di V, tale che ogni arco in E ha almeno un estremo in S.

A ogni vertice è associato un peso $w_i \geq 0$.

Il peso della copertura S è dato da

$$W(S) = \sum_{i \in S} w_i$$



Per usare la programmazione lineare abbiamo bisogno di codificare in disuguaglianze lineari la richiesta che i nodi formino una copertura e nella funzione obiettivo lo scopo di minimizzare il peso totale.
Per esprimere il problema di minimizzazione scriviamo l'insieme dei pesi dei nodi come vettore w, con componenti w_i.
Si deve pertanto minimizzare il prodotto $w_i\, x$.

$$\begin{aligned} \text{Min} \quad & \sum_{i \in V} w_i x_i \\ \text{s.t.} \quad & x_i + x_j \geq 1 \quad (i,j) \in E \\ & x_i \in \{0,1\} \quad i \in V. \end{aligned} \tag{1}$$

Le coperture di vertici del grafo G sono in corrispondenza uno ad uno con le soluzioni del sistema (1), cioe`:

S è una copertura di vertici in G se e solo se il vettore x, definito come $x_i = 1$ per i in S e $x_i = 0$ altrimenti, soddisfa i vincoli (1). Inoltre $w(S) = w_i\, x$.

Trasformiamo il sistema nella forma matriciale usata nella programmazione lineare.

Definiamo una matrice A: *archi di G* $\times$ *vertici di G , d*ove A[*e*, *i*] = 1 se l'arco e *è* incidente nel nodo *i*, 0 altrimenti. Allora abbiamo

$$A\,x \geq \vec{1}$$

$$\vec{1} \geq x \geq \vec{0}$$

Otteniamo la funzione obiettivo:

$$min(\vec{w_i}\,\vec{x_i} \mid 1 \geq x \geq 0,\ A\,x \geq 1,\ x \text{ ha componenti intere})$$

Questa è un'istanza del problema in cui richiediamo che le coordinate di *x* assumano valori interi (solo valori in {0,1}); abbiamo allora a che fare con un problema di "programmazione intera„.

La definizione del problema copertura di vertici in termini di programmazione intera definisce una riduzione polinomiale:

$$\text{Vertex Cover} \propto \text{Integer Programming}$$

La programmazione intera e` un problema **NP**-arduo, quindi non possiamo sperare di risolverlo con un algoritmo polinomiale, ma la programmazione lineare non e` “difficile„ quanto la programmazione intera.
Il problema sarebbe molto più facile se potessimo *rilassare* il vincolo che *x* assuma valori interi e utilizzare anche valori reali.

Risolveremo il problema più semplice, di programmazione lineare, in modo esatto e trasformeremo la soluzione reale in una intera approssimata con un rapporto di approssimazione 2.

Definiamo quindi un nuovo problema rilassando il vincolo $x_i \in \{0,1\}$ per trovare un insieme $\{x_i^*\}$ di valori reali tra 0 e 1:

$$\min \sum_i w_i x_i^*$$

$$x_i^* + x_j^* \geq 1 \qquad \forall (i, j) \text{ in } E$$

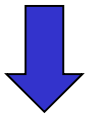
$$0 \leq x_i^* \leq 1 \qquad \forall i \text{ in } V$$

Il nuovo problema è risolubile in tempo polinomiale.

La soluzione trovata w_{LP} non può essere maggiore del peso di una soluzione ottima S* per copertura di vertici.
Infatti , dal momento che nel problema formulato come programmazione lineare *x* puo` assumere qualunque valore reale tra 0 e 1, la minimizzazione e` fatta con molte piu` scelte per x. La soluzione w_{LP} non potrà essere maggiore di quella ottenuta per la programmazione a valori interi.

$$w_{LP} \leq w(S^*)$$

Quanto ci aiuta la programmazione lineare? Può dare soluzioni con peso molto minore di *w*(S*)?



Arrotondamento alla soluzione intera

Data una soluzione reale al problema di programmazione lineare, definiamo i valori di x_i nel modo seguente:

$$x_i = 0 \quad \text{se} \quad x_i^* < 1/2$$
$$x_i = 1 \quad \text{se} \quad x_i^* \geq 1/2$$

Sia $S = \{i \in V \mid x_i^* \geq ½\}$

1) S è una copertura di vertici
2) $w(S) \leq 2\, w_{LP}$

1) Poichè $x^*_i + x^*_j \geq 1$ in qualunque soluzione reale ammissibile, x_i^* oppure x_j^* dovranno essere maggiori o uguali di ½ , in questo modo l'algoritmo inserirà almeno uno tra il nodo i e il nodo j in S.

Analisi del rapporto di approssimazione

2) Consideriamo il peso *w*(S), l'insieme S ha solo vertici con valore $x_i^* \geq ½$ nella soluzione reale. Così almeno uno tra gli estremi di un arco x_i^* e x_j^* sarà $\geq ½$, quindi arrotondato per eccesso e inserito in S.
La soluzione di programmazione lineare *paga* un costo almeno $½w_i$ per ogni nodo *i* in S, mentre la soluzione intera *paga* al massimo il doppio, cioè w_i.

$$w_{LP} = w^t x^* = \sum_i w_i x_i^* \geq \sum_{i \in S} w_i x_i^* \geq \frac{1}{2} \sum_{i \in S} w_i = \frac{1}{2} w(S). \qquad (S \subseteq V)$$

Dalle disuguaglianze $w_{LP} \leq w(S^*)$ e $w_{LP} \geq 1/2 w(S)$ otteniamo:

$$w(S) \leq 2w_{LP} \leq 2w(S^*)$$

Per il problema copertura pesata di vertici, l'insieme S ottenuto da una soluzione esatta di programmazione lineare per arrotondamento e` una soluzione approssimata con rapporto di approssimazione 2.

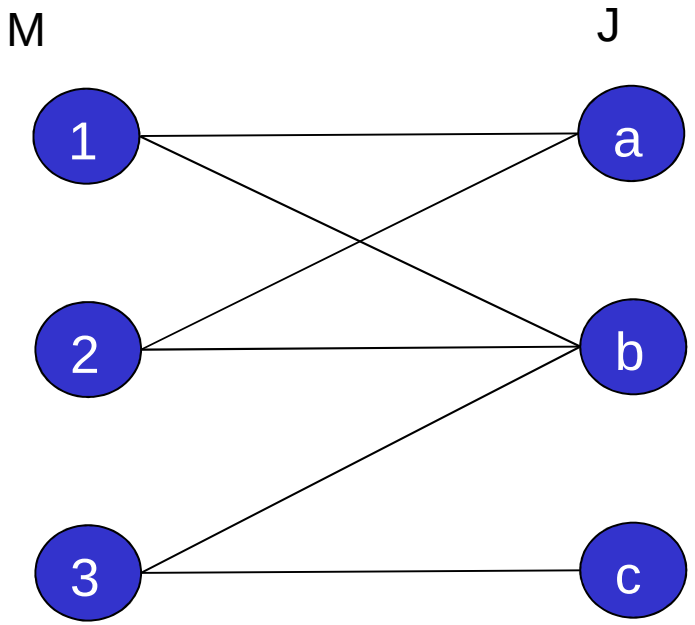
Come per il problema della copertura di vertici, sviluppiamo un algoritmo di 2-approssimazione per il problema del **load balancing**. Per farlo risolveremo il problema come uno equivalente di programmazione lineare, in cui le variabili assumono valori discreti.
Modificheremo poi il problema rilassando questo requisito, arrotondando la soluzione risultante a valori interi in modo che sia il più possibile vicina all'ottimo.

Il problema

Nell classico problema del load balancing si hanno:

- Un insieme J di *n* lavori, ognuno con peso t_j
- Un insieme M di *m* macchine

Consideriamo ora il problema nella sua variante più generale in cui ogni processo j non può essere assegnato a qualunque macchina ma solo ad un suo sottoinsieme arbitrario M_j.
L'obiettivo è assegnare ogni lavoro ad una macchina in modo da minimizzare il massimo carico su ogni macchina.



$M_a = \{1, 2\}$
$M_b = \{1, 2, 3\}$
$M_c = \{3\}$

Chiamiamo *ammissibile* un assegnamento se ogni lavoro è assegnato ad una macchina i in M_j

Il problema di programmazione intera:

- x_{ij} indica il carico del lavoro j assegnato alla macchina i
- $x_{ij} = 0$ indica che il lavoro j non è assegnato alla macchina i
- $x_{ij} = t_j$ indica che l'intero carico di j è assegnato alla macchina i

possiamo quindi considerare *x* come il vettore soluzione con *mn* coordinate.

<u>sistema di diseguaglianze</u>:

$$\min L$$

$$\sum_i x_{ij} = t_j \quad \text{for all } j \in J$$

$$\sum_j x_{ij} \leq L \quad \text{for all } i \in M \qquad (2)$$

$$x_{ij} \in \{0, t_j\} \quad \text{for all } j \in J, i \in M_j.$$

$$x_{ij} = 0 \quad \text{for all } j \in J, i \notin M_j.$$

Funzione obiettivo da minimizzare, indica il carico massimo

Il carico di ogni lavoro deve essere eseguito completamente

Il carico di ogni macchina non deve superare *L*

Una assegnazione di lavori alle macchine ha carico al più L se e solo se il vettore x soddisfa i vincoli (2).

Come nel caso precedente, il costo per ottenere la soluzione intera sarebbe troppo elevato. Affrontiamo quindi il problema con la tecnica della programmazione lineare, rilassando il vincolo $x_{ij} \in \{0, t_j\}$, e sostituendolo con la richiesta più debole $x_{ij} \geq 0$, sfruttando anche il vincolo $x_{ij} \leq t_j$ implicitamente espresso dal vincolo

$$\sum_i x_{ij} = t_j \quad \text{per tutti i lavori in J}$$

In questo modo si esprime il fatto che x_{ij} può assumere qualunque valore compreso tra 0 e t_j.

L'algoritmo di programmazione lineare restituisce in tempo polinomiale una soluzione (x, L).

Come ottenere una assegnazione ammissibile a partire da quella ottenuta con il rilassamento dei vincoli?

Per farlo dovremo arrotondare i valori calcolati per x_{ij} a 0 oppure a t_j.
Se si assegna ogni lavoro j ad una macchina i con $x_{ij} > 0$ l'assegnazione sarà ammissibile.
Il problema è che potremmo trovarci nel caso in cui la soluzione ha assegnato frazioni molto piccole di t_j a macchine diverse, per cui nessuna di queste verrebbe arrotondata per eccesso e il lavoro j non sarebbe assegnato a nessuna macchina.

Dobbiamo quindi adottare una strategia di arrotondamento diversa da quella usata per l'esempio precedente, basata sulla struttura della soluzione.

A tale scopo costruiamo un grafo bipartito G = (V, E) i cui nodi rappresentano sia le macchine che i lavori: V = M ∪ J, e in E è presente un arco <i, j> se e solo se $x_{ij} > 0$.

Partendo dalla soluzione del problema in programmazione lineare costruiamo un grafo bipartito

$$G = (V, E)$$

i cui nodi rappresentano sia le macchine che i lavori: $V = M \cup J$,

in E è presente un arco <i, j> se e solo se $x_{ij} > 0$.

Per esempio nel caso precedente una soluzione potrebbe essere:

$x_{1a} = 3$, $x_{1b} = 1$, $x_{1c} = 0$ c non è compatibile con 1

$x_{2a} = 0$, $x_{2b} = 1$, $x_{2c} = 0$ c non sono compatibile con 2

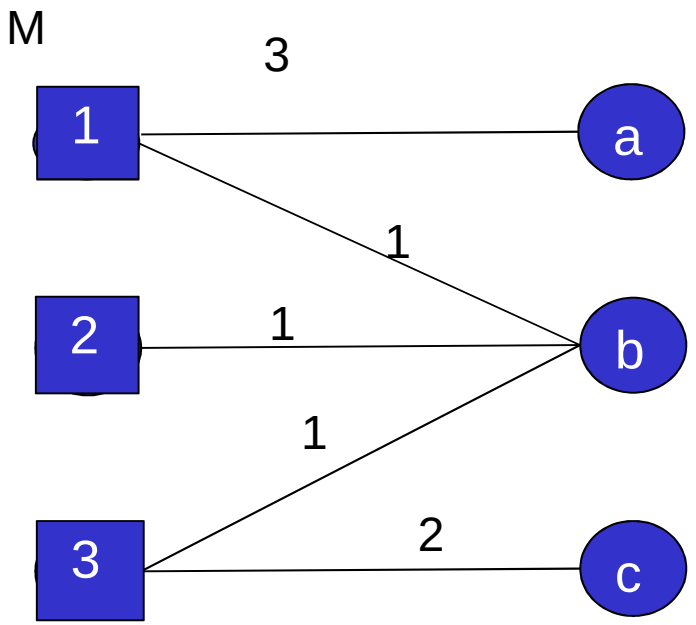
$x_{3a} = 0$, $x_{3b} = 1$, $x_{3c} = 2$ a non è compatibile con 3

$t_a = 3$

$t_b = 3$

$t_c = 2$

NOTA: i valori numerici sono indicativi. *Non* sono quelli che si otterrebero dalla soluzione in programmazione lineare

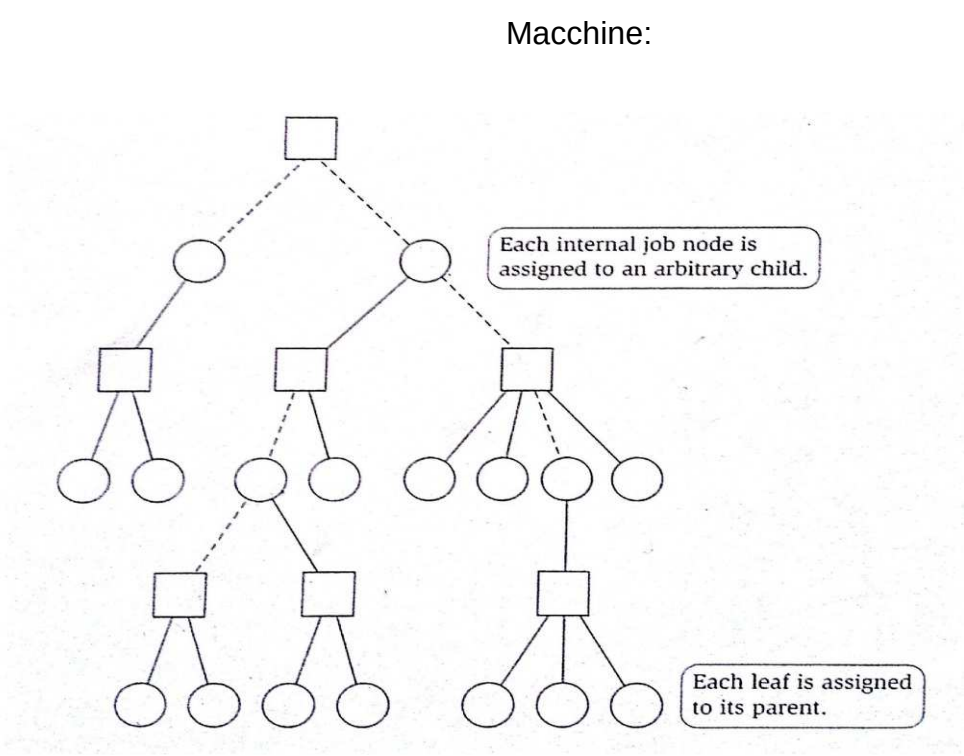


Se il grafo ottenuto *non ha cicli*, ognuna delle sue componenti connesse è un albero. Possiamo costruire l'assegnazione considerando ogni componente separatamente.

L'assegnazione può essere fatta nel modo seguente:

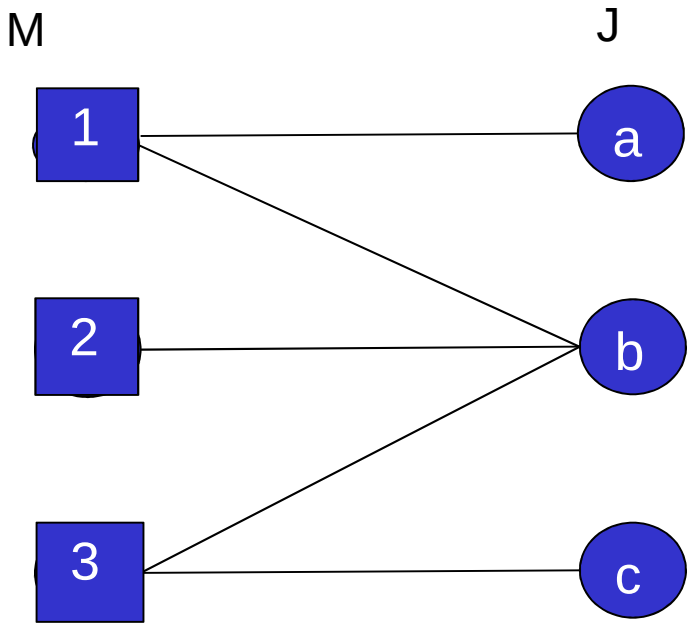
- Scegliere un nodo come radice
- Se il job j è una foglia assegnare j al suo unico padre i
- Se j non è un foglia, assegnare j ad uno (e uno solo) dei suoi figli

Quando si assegna j a i x_{ij} e` arrotondato a t_j e gli eventuali x_{kj} ($k \neq j$) vengono azzerati.



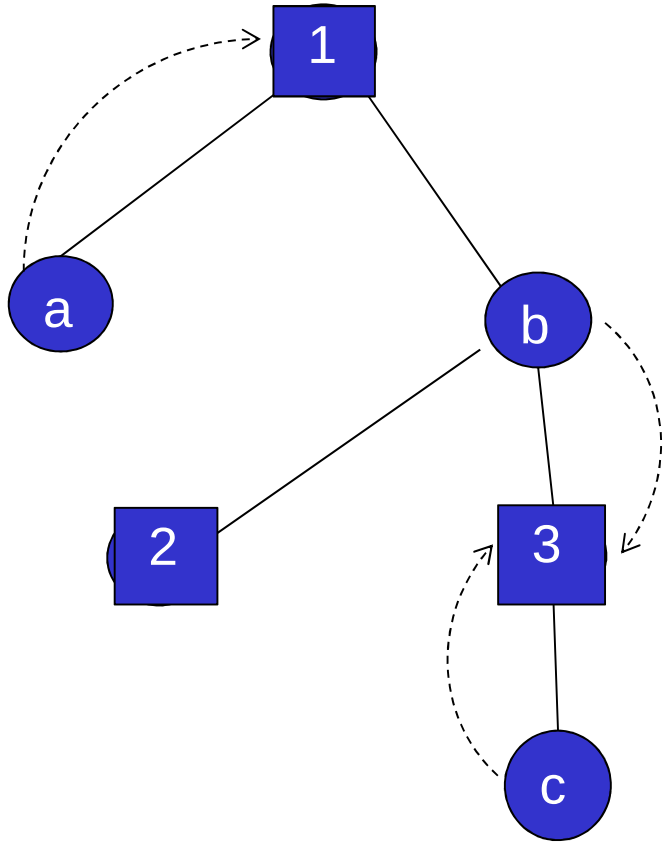
La costruzione del grafo e l'assegnazione dei job alle macchine possono essere eseguite in tempo $\boldsymbol{O}(mn)$.

Esempio: prendiamo il caso precedente ma togliamo il ciclo



$M_1 = \{a\}$
$M_2 = \emptyset$
$M_3 = \{b, c\}$

$t_a = 3$
$t_b = 3$
$t_c = 2$



- Il job a viene assegnato alla macchina 1, i job b e c alla macchina 3.

- Notare che il job b poteva anche essere assegnato alla macchina 2.

Analisi del rapporto di approssimazione

1. come già notato nel caso dell'algoritmo ,"greedy", il carico ottimale L^* non può essere minore del massimo tempo d'esecuzione dei lavori:

$$L^* \geq max_j\, t_j$$

2. l'insieme dei lavori assegnati alla macchina i, J_i, è composto dall'insieme dei suoi figli che sono foglie, più eventualmente il nodo padre *p*(i). Per trovare una limitazione al carico, consideriamo le due componenti separatamente:

   Per i *nodi foglia* si ha $x_{ij} = t_j$ e un vincolo del problema, mantenuto dall'arrotondamento, richiede che il carico massimo per ogni macchina sia ≤ *L*.
   Otteniamo cosi`: $\sum_{j \in J_i, j \neq p(i)} t_j \leq \sum_{j \in J} x_{ij} \leq L$

   Sappiamo inoltre che per il padre j = *p*(i) del nodo i vale $t_j \leq L^*$ (per il punto 1.)

   Unendo le due disuguaglianze otteniamo una limitazione superiore per il carico calcolato dall'algoritmo approssimato: $\sum_{j \in J_i} t_{ij} \leq L + L^*$

3. Come già evidenziato per il problema copertura di vertici, la soluzione ottima L^* non può essere inferiore alla soluzione ottima L della programmazione lineare: $L^* \geq L$.

Possiamo allora concludere che il carico massimo è limitato da 2L*, il doppio dell'ottimo.

$$\sum_{j \in J_i} t_{ij} \leq L + L^* \leq 2\, L^*$$

Data una soluzione (x, L) del problema programmazione lineare, se il grafo G non ha cicli, si può usare la soluzione x per ottenere un assegnamento ammissibile di job alle macchine con carico massimo al più doppio dell'ottimo in tempo polinomiale $\mathbf{O}(mn)$.

L'algoritmo proposto funziona quando il grafo è privo di cicli.

Spesso non è così.

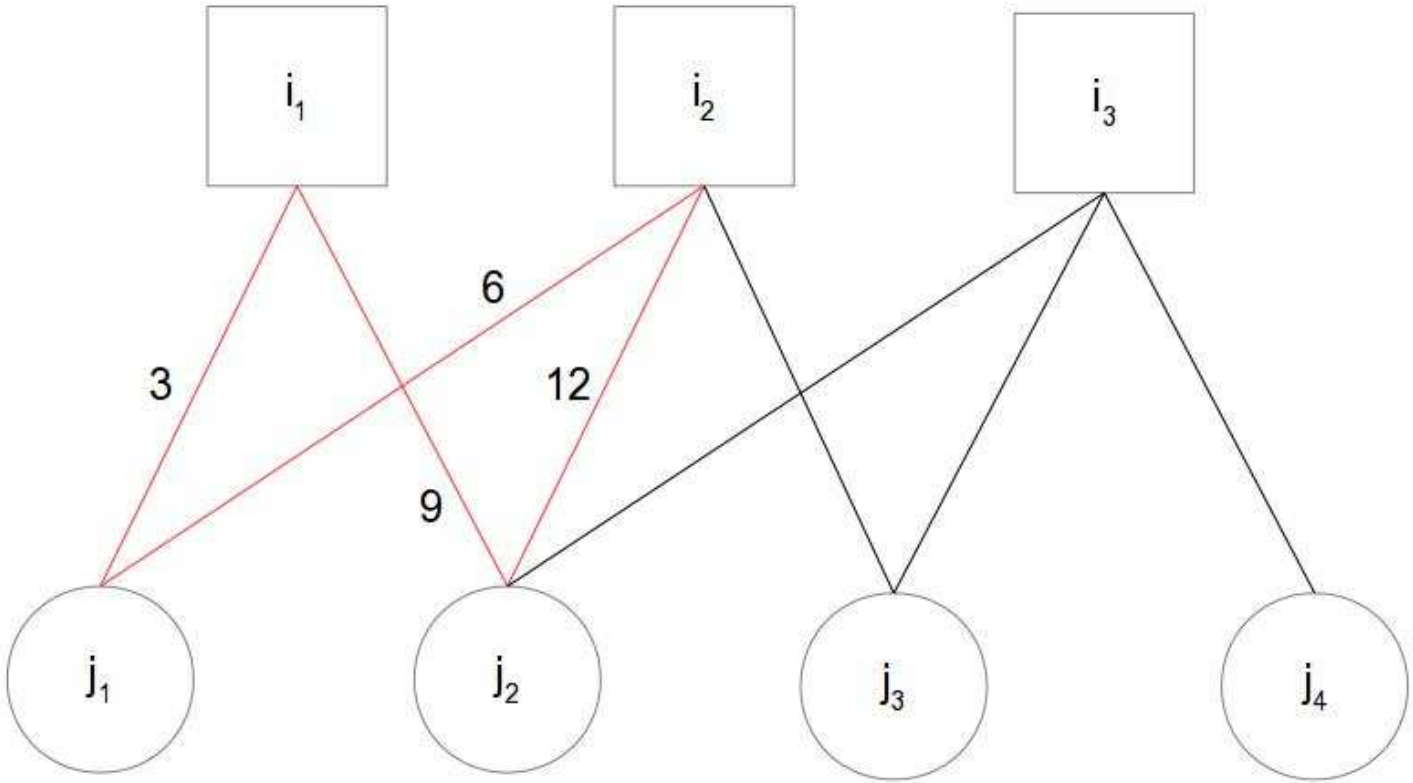
Occorre quindi trovare un metodo per eliminare i cicli in modo da poter applicare l'algoritmo e fornire una soluzione in tempo polinomiale.

Supponiamo che i nodi che formano un ciclo siano $i_1, j_1, i_2, j_2, \ldots\ldots, i_k, j_k, i_1$ dove i_h e` una macchina e j_h un job (h = 1, …, k).
Modifichiamo il grafo decrementando il carico su tutti gli archi (j_h, i_h) e incrementandolo su quelli (j_h, i_{h+1}), per tutti gli h = 1,..., k, di una quantità δ.

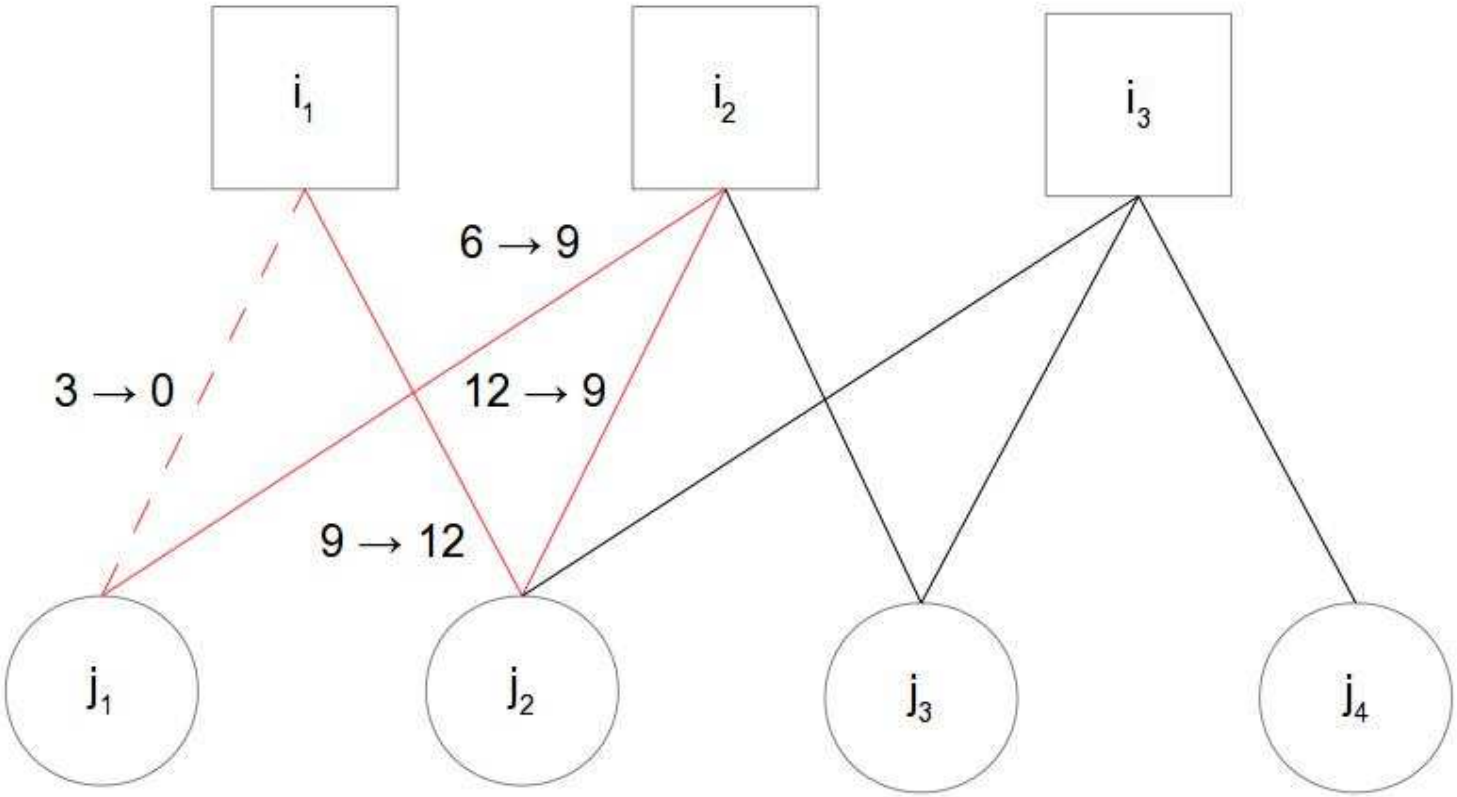
Questa modifica garantisce il mantenimento dei carichi sulle macchine e dei tempi di esecuzione per i job.
Scegliendo $\delta = min_{h=1\ldots k}\, x_{ih\,,\,jh}$, cioè il minimo carico fornito da un arco, assicuriamo che l'assegnamento rimanga ammissibile e che l'arco con peso minimo possa essere eliminato, in quanto il suo peso diventa 0.

Sul grafo ottenuto si puo` ripetere il procedimento fino a quando tutti i cicli sono stati eliminati, e quindi applicare l'algoritmo visto per la costruzione della soluzione approssimata.

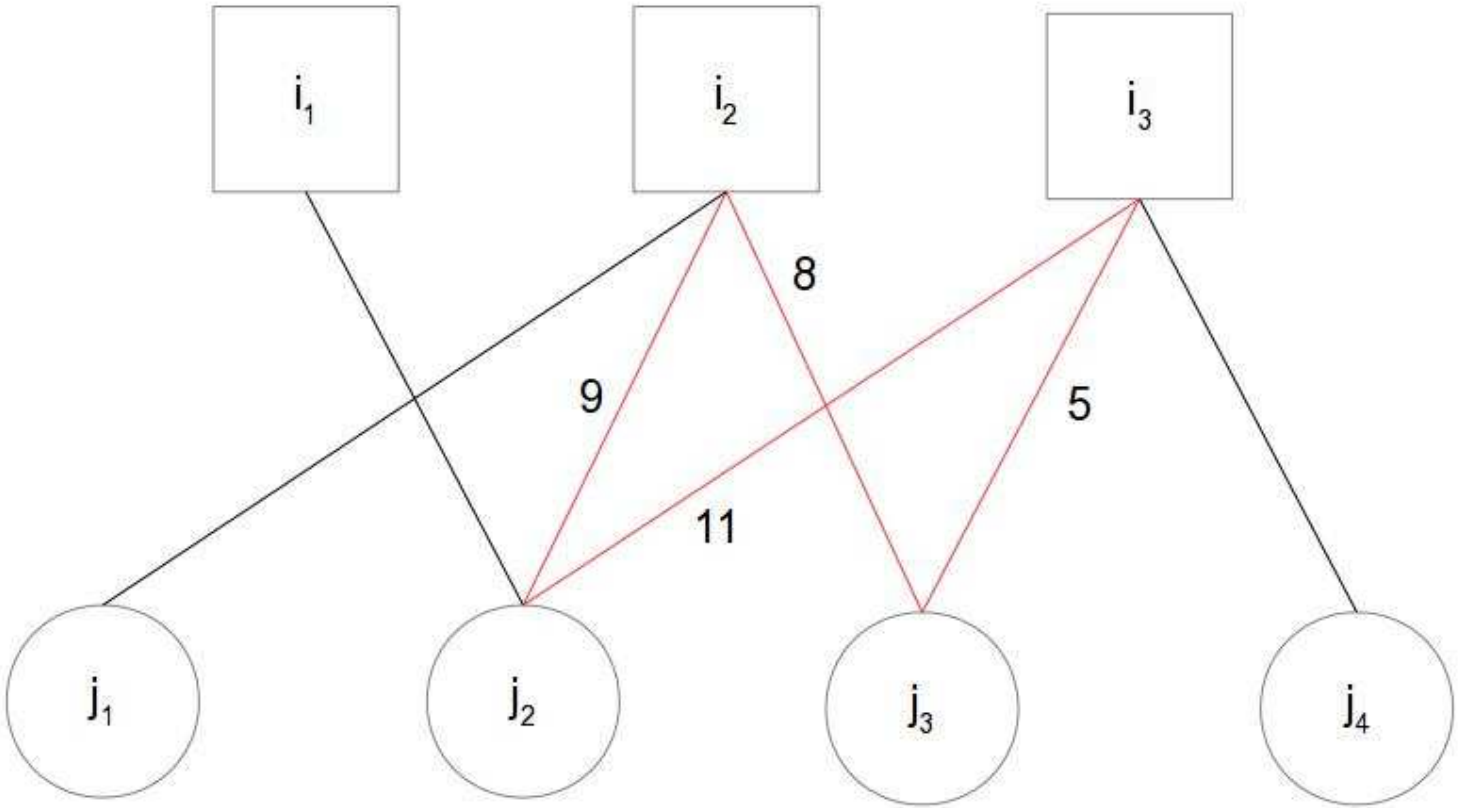
ESEMPIO - 1



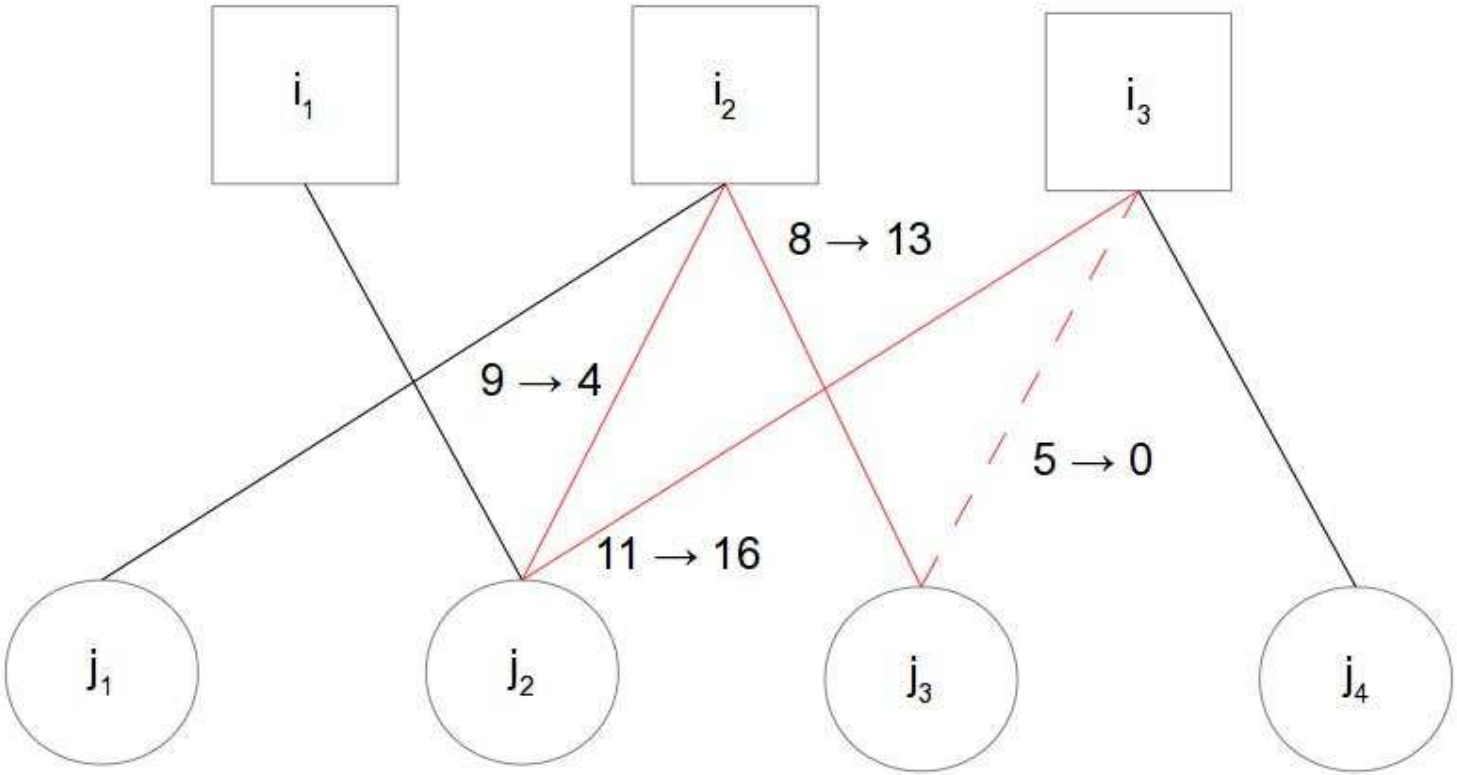
ESEMPIO - 2


i_1
i_2
i_3
$6 \rightarrow 9$
$3 \rightarrow 0$
$12 \rightarrow 9$
$9 \rightarrow 12$
j_1
j_2
j_3
j_4

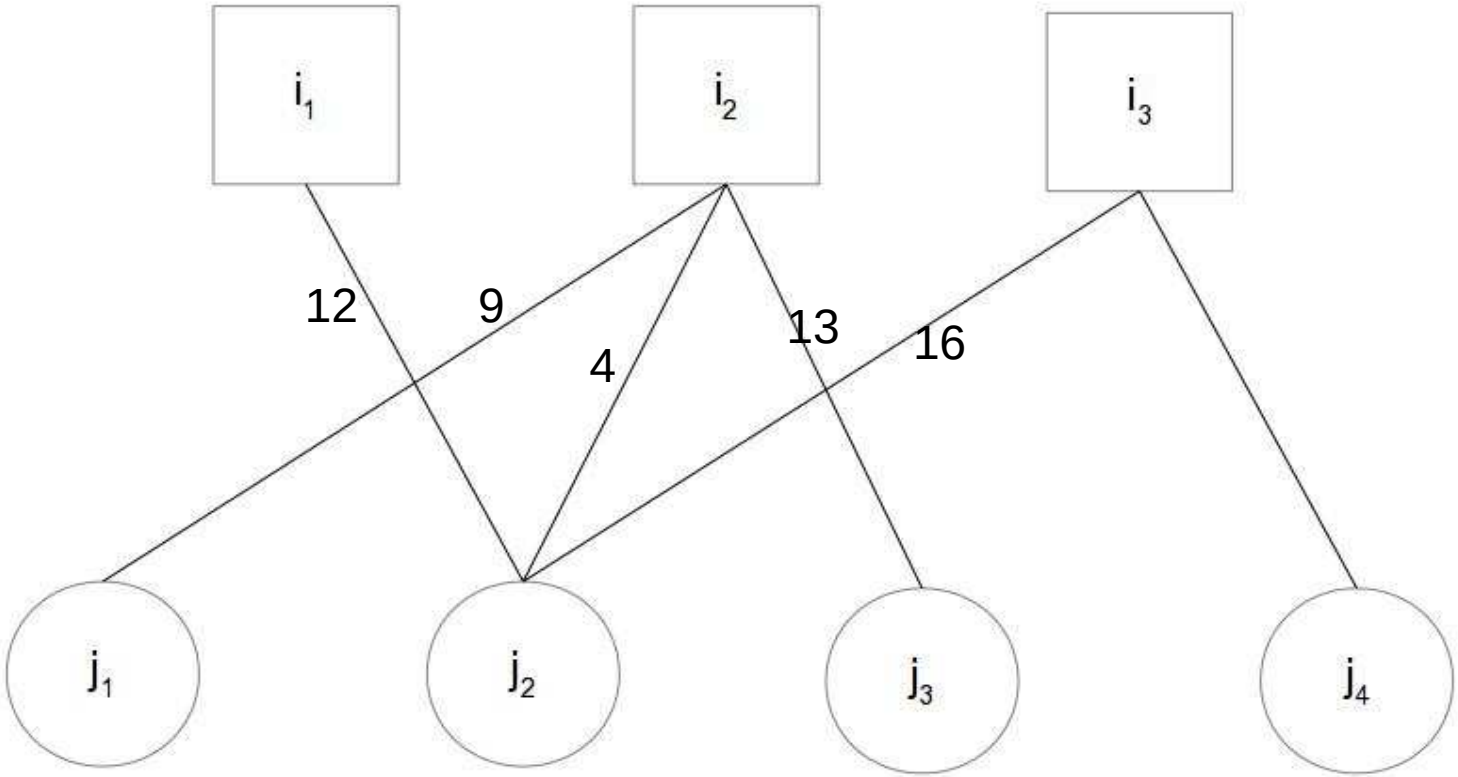
ESEMPIO - 3



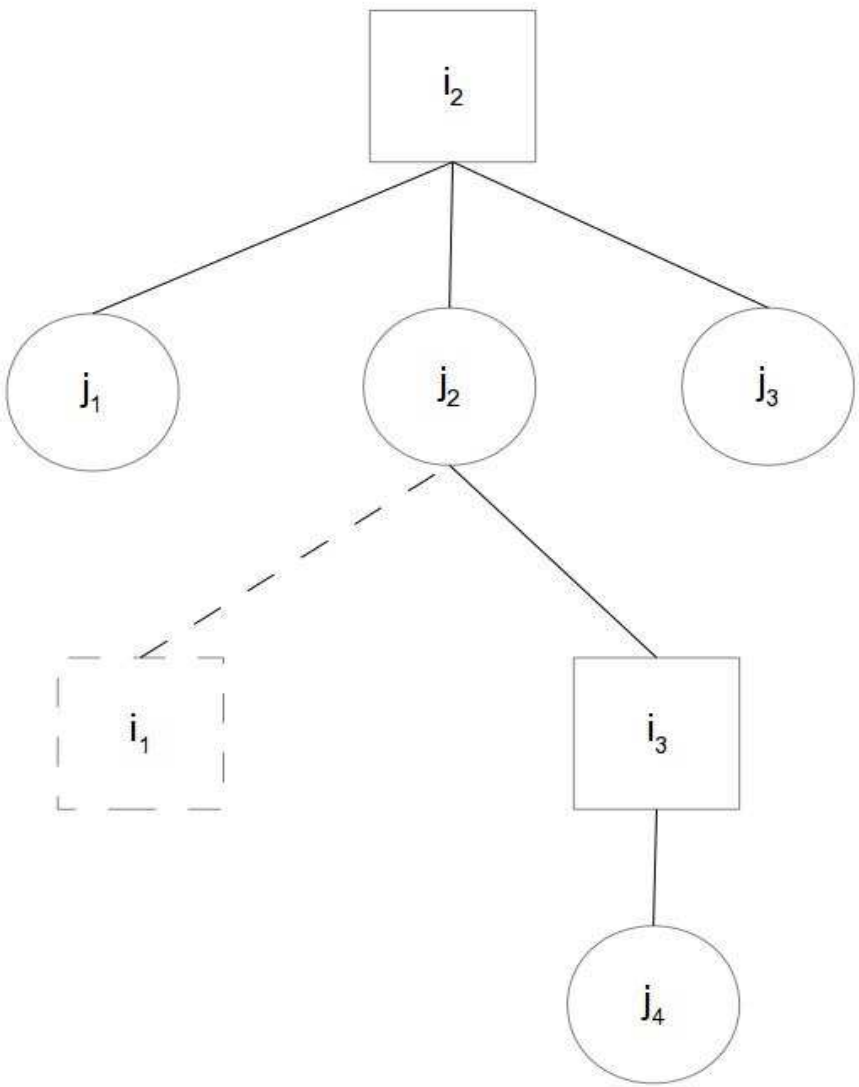
ESEMPIO - 4

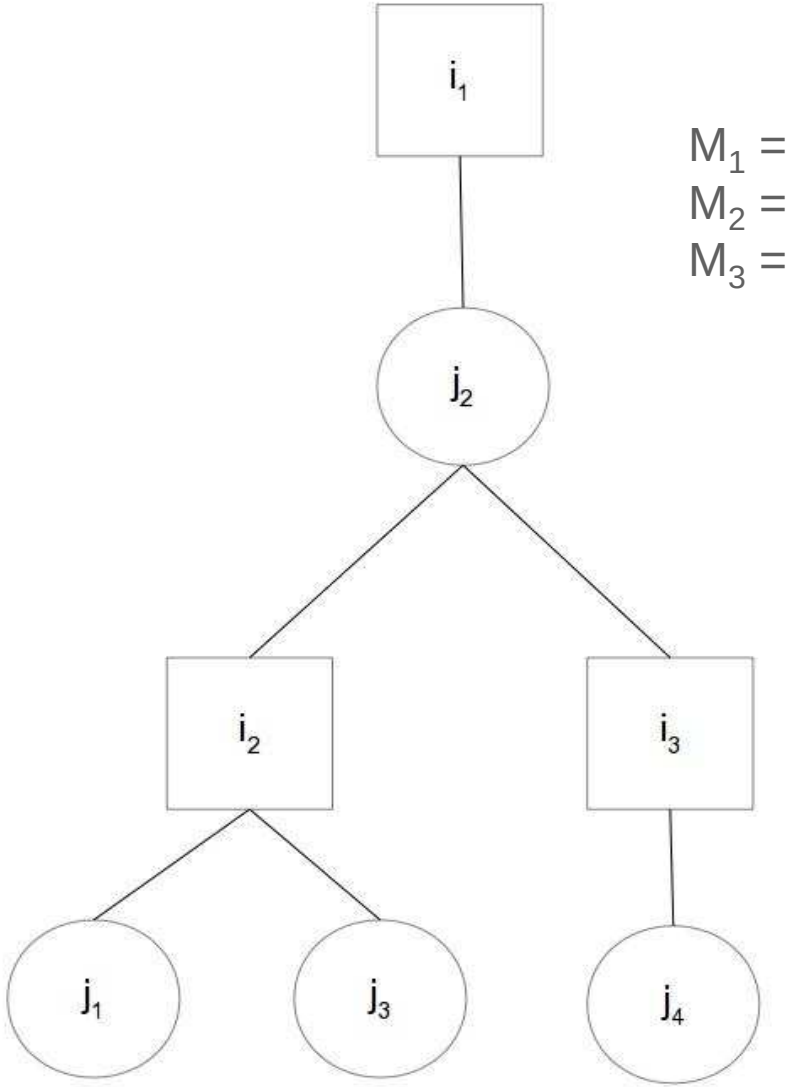


ESEMPIO – 5: grafo aciclico



ESEMPIO – due possibili alberi per l'assegnazione del carico ottenibili dal grafo in cui sono stati eliminati i cicli.





$M_1 = \emptyset$
$M_2 = \{J_1, j_3\}$
$M_3 = \{j_2, j_4\}$

Se c è un ciclo nel grafo G si può eliminare almeno un arco da G senza aumentare il carico e senza introdurre nuovi archi in tempo lineare nella lunghezza del ciclo.

Data una istanza del problema Load Balancing Generalizzato
si puo` trovare, in tempo polinomiale,
un assegnamento ammissibile
con carico massimo al piu` doppio del minimo possibile.

La soluzione nota di programmazione dinamica del problema *zaino* definisce i sottoproblemi da risolvere in funzione degli oggetti e della capacità dello zaino.

oggetti

$$Val(i,k) = \begin{cases} 0 & \text{se } i = 0 \text{ or } k = 0 \\ Val(i-1,k) & \text{se } i, k \neq 0 \text{ and } k < w_i \\ \max \{Val(i-1,k),\ Val(i-1,\ k - w_i) + v_i\} & \text{se } i, k \neq 0 \text{ and } k \geq w_i \end{cases}$$

capacità

Per capacità molto grandi si genera un gran numero di sottoproblemi.

Esempio

n = 4 W = 8

$v_1 = 3$ $v_2 = 8$ $v_3 = 1$ $v_4 = 4$

$p_1 = 3$ $p_2 = 5$ $p_3 = 2$ $p_4 = 4$

i ↓ k →	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	0	0	3	3	3	3	3	3
2	0	0	0	3	3	8	8	8	11
3	0	0	1	3	3	8	8	9	11
4	0	0	1	3	3	8	8	9	11

Il valore massimo è dato dagli oggetti 1 e 2.

Zaino è un problema non fortemente **NP**-completo: se i valori sono piccoli il problema può essere risolto in tempo polinomiale.

Però

Se i valori sono grandi possiamo «scalarli» e risolvere il problema sui valori scalati; non otterremo la soluzione migliore possibile, ma possiamo sperare in una buona approssimazione. In particolare possiamo ottenere un'approssimazione nell'ordine di $(1+\epsilon)$ per ogni ϵ.

Gli algoritmi che dipendono dai valori in modo pseudo-polinomiale possono spesso essere usati per disegnare *schemi di approssimazione* polinomiali in tempo.

Organizziamo allora l'algoritmo *in funzione dei valori anziché dei pesi*.

Definiamo il problema come *zaino(i, V)* che rappresenta il minimo peso w dello zaino che serve per raggiungere un valore almeno V considerando il sottoinsieme {1 … i} degli oggetti. Avremo quindi un sottoproblema per ogni i = 0 … n e per ogni valore V = 1, ... ,$\sum_{j = 1 \ldots n} v_j$.

Dal momento che la complessità dell'algoritmo dipende dalla dimensione dei valori, abbiamo bisogno di una strategia per diminuirla.

Usiamo un parametro di *arrotondamento b* e facciamo in modo che i valori risultanti siano suoi multipli.
Calcoliamo i valori $v'_i = \lceil v_i / b \rceil \cdot b$ in modo tale da arrotondarli (per eccesso) al multiplo di *b* più vicino.

Per ogni oggetto i si ottiene: $v_i \leq v'_i \leq v_i + b$

Si tratta ancora di valori grandi, per diminuirne la dimensione li dividiamo per *b* ottenendo:

$$v''_i = v'_i / b = \lceil v_i / b \rceil$$

I valori v' e i valori scalati v'' danno lo *stesso* insieme di soluzioni ottime poiché essi differiscono esattamente per un fattore *b*.

Usiamo allora nell'algoritmo i valori v'' (discreti) invece dei v.

Sia zaino(I, V) la *minima* capacità dello zaino per trasportare un valore di *almeno* V o comunque il valore il più possibile vicino a V

Sia O l'insieme di oggetti che definiscono la soluzione ottima.
Consideriamo solo oggetti di peso minore della capacità dello zaino.
Otteniamo la seguente relazione di ricorrenza:

$$zaino(i,V) = \begin{cases} 0 & \text{se } V = 0 \\ zaino(i-1,\ V - v_i) + w_i & \text{se } V > \sum_{j=1\ldots i-1} v_j \text{ (i è necessario } \mathbf{i} \text{ per raggiungere V)} \\ \min\{zaino(i-1,V),\ zaino(i-1,\ V - v_i) + w_i\} & \text{altrimenti} \end{cases}$$

Se chiamiamo v* il massimo valore dei v'', il più grande valore V ottenibile in un sottoproblema e' $\sum_{j\,=\,1\ldots n} v''_j \leq nv^*$; otteniamo così $\mathbf{O}(n^2v^*)$ sottoproblemi.

$$\text{N.B. } A \dot{-} B = \begin{cases} A - B & \text{se } A \geq B \\ 0 & \text{altrimenti} \end{cases}$$

Knapsack-a (n)

Array W [0 … n, 1 … V]

For $i = 0, \ldots, n$
  $M[i, 0] = 0$
Endfor
For $i = 1, 2, \ldots, n$
  For $V = 1, \ldots, \sum_{j=1}^{i} v_j$
    If $V > \sum_{j=1}^{i-1} v_j$ then
      $M[i, V] = w_i + M[i-1, V]$
    Else
      $M[i, V] = \min(M[i-1, V], w_i + M[i-1, \max(0, V - v_i)])$
    Endif
  Endfor
Endfor
Return the maximum value V such that $M[n, V] \leq W$

Ma come scegliere b?
Usiamo un valore che dipende dalla approssimazione voluta ($\varepsilon > 0$), dal numero di oggetti e dal massimo valore degli oggetti:

$$b = (\varepsilon/n)\ \max_i v_i$$

Possiamo ora scrivere il seguente algoritmo approssimato:

Knapsack-Approx (ε)
  $b \leftarrow (\varepsilon/n)\ \max_i v_i$
  calcola i valori v_i"
  risolvi il problema dello zaino con valori v_i"
  restituisci l'insieme S degli oggetti trovati

Poiché i pesi non sono stati modificati si può asserire che:

L'insieme di oggetti S restituito dall'algoritmo ha peso totale al più W: $\sum_{i \in S} w_i \leq W$

Calcolare *b* e i valori arrotondati può ovviamente essere fatto in tempo polinomiale.
Poiché l'algoritmo Knapsack-a ha una complessità che dipende da $v^* = \max_i v''_i$ (nell'algoritmo usiamo i valori v''), dobbiamo determinare tale valore per poter calcolare la complessità di Knapsack-Approx.

Sia j l'oggetto con il valore massimo: $v_j = max_i\, v_i$ allora

$$max_i\, v''_i = v''_j = \lceil v_j / b \rceil = n\varepsilon^{-1}$$

Quindi il tempo di esecuzione dell'algoritmo è $\mathbf{O}(n^3\varepsilon^{-1})$, polinomiale per ogni ε fissato.

L'algoritmo Knapsack-Approx ha complessità *in tempo polinomiale per tutti gli* $\varepsilon > 0$ *fissati.*

L'algoritmo è polinomiale per ogni valore di ε, ma per ε molto piccoli diventa esponenziale perché le operazioni su operandi dell'ordine di ε^{-1} (che diventa molto grande) non si possono più considerare a tempo costante e bisogna considerare nella dimensione dell'input la codifica di ε che è nell'ordine di log ε.

Utilizzando l'algoritmo di programmazione dinamica per ottenere un algoritmo di approssimazione polinomiale, si ottiene in effetti uno *schema di approssimazione*, cioè un algoritmo di approssimazione con rapporto che può diminuire arbitrariamente, ovviamente a spese del tempo di calcolo.

Analisi del rapporto di approssimazione.

L'algoritmo Knapsack trova una soluzione ottima con i valori v_i', quindi per ogni altra soluzione S* (compresa quella ottima):

$$\sum_{i\in S} v'_i \;\geq\; \sum_{i\in S^*} v'_i$$

Otteniamo quindi la seguente catena di disuguaglianze:

$$\sum_{i\in S^*} v_i \;\leq\; \sum_{i\in S^*} v'_i \;\leq\; \sum_{i\in S} v'_i \;\leq\; \sum_{i\in S}(v_i + b) \;\leq\; nb + \sum_{i\in S} v_i$$

Poiché ogni oggetto può stare nello zaino, v_j da solo non può superare il valore una soluzione per cui

$$v_j = \max_i v_i \leq \sum_{i\in S} v_i$$

Quindi $n\cdot b = n\,(\varepsilon/n)\max_i v_i \leq \varepsilon \sum_{i\in S} v_i$ e $\sum_{i\in S} v_i + nb \leq (1+\varepsilon)\sum_{i\in S} v_i$

Se S è la soluzione trovata dall'algoritmo Knapsack-Approx, ed S* qualunque altra soluzione che soddisfa $\sum_{i\in S^*} w_i \leq W$, allora $\sum_{i\in S^*} v_i \leq (1+\varepsilon)\sum_{i\in S} v_i$

Esempio

Applichiamo il procedimento visto all'esempio precedente.

n = 4 W = 8

$v_1 = 3$ $v_2 = 8$ $v_3 = 1$ $v_4 = 4$

$p_1 = 3$ $p_2 = 5$ $p_3 = 2$ $p_4 = 4$

Prendiamo $\varepsilon = 0{,}8$

Abbiamo allora $b = \varepsilon/n \times v^* = 0{,}8/4 \times 8 = 1{,}6$

$v_1 = 3$	$v_1' = \lceil 3/1.6 \rceil \times 1.6 = 2 \times 1{,}6 = 3.2$	$3 \le 3.2 \le 4.6$	$v_1'' = 2$
$v_2 = 8$	$v_2' = \lceil 8/1.6 \rceil \times 1.6 = 5 \times 1{,}6 = 8$	$8 \le 8 \le 9.6$	$v_2'' = 5$
$v_3 = 1$	$v_3' = \lceil 1/1.6 \rceil \times 1.6 = 1 \times 1{,}6 = 1.6$	$1 \le 1.6 \le 2.6$	$v_3'' = 1$
$v_4 = 4$	$v_4' = \lceil 4/1.6 \rceil \times 1.6 = 3 \times 1{,}6 = 4.8$	$4 \le 4.8 \le 5.6$	$v_4'' = 3$

Esempio

n = 4 W = 8

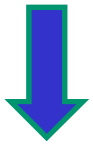
$v_1'' = 2$ $v_2'' = 5$ $v_3'' = 1$ $v_4'' = 3$

$p_1 = 3$ $p_2 = 5$ $p_3 = 2$ $p_4 = 4$

i ↓ v →	0	1	2	3	4	5	6	7	8	9	10	11
0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	3	3	3	3	3	3	3	3	3	3	3
2	0	3	3	8	8	8	8	8	8	8	8	8
3	0	2	3	5	5	5	7	8	10	10	10	10
4	0	2	3	4	5	5	7	8	10	14	14	14

Soluzione ottima per i valori v_i'' e v_i':

il massimo valore trasportabile con lo zaino di capacità W = 8 è 7, ed è formato dagli oggetti 1 e 2.



$v_1 + v_2 = 11$

la soluzione «approssimata» coincide con la soluzione ottima

Altro esempio

Se, restando inalterati i pesi e il valore di ε, i valori degli oggetti fossero i seguenti:

$v_1 = 8 \quad v_2 = 10 \quad v_3 = 7 \quad v_4 = 9$

$p_1 = 3 \quad p_2 = 5 \quad p_3 = 2 \quad p_4 = 4$

Avremmo $b = \varepsilon/n \cdot v^* = 0{,}8/4 \times 10 = 2$ e quindi i seguenti valori di v_i''

$v_1'' = 4 \quad v_2'' = 5 \quad v_3'' = 4 \quad v_4'' = 5$

Soluzione esatta per i valori v_i'':
9 ottenibile con peso 6 dello zaino usando gli oggetti 3 e 4.

Soluzione approssimata per i valori v:
16 valore massimo trasportabile

Soluzione esatta per i valori v: 18

$18 < (1 + 0.8)\ 16$

Mentre i problemi **NP**-completi sono tutti equivalenti rispetto alla risolubilità in tempo polinomiale, se **P** ≠ **NP**, essi differiscono considerevolmente nel fatto che le loro soluzioni possano essere efficientemente approssimate.

In alcuni casi è possibile dimostrare dei limiti sull'approssimabilità (algoritmo per "selezione dei centri", algoritmo per "copertura di insiemi").

Quando si ha a che fare con problemi di ottimizzazione **NP**-ardui si vorrebbero trovare soluzioni approssimate che non si discostino troppo dalla soluzione ottima (ad es. 1%).

Sono noti algoritmi di ε-approssimazione, che producono una soluzione ε-approssimata in tempo polinomiale sia nella dimensione del problema sia in 1/ε, per pochissimi problemi di ottimizzazione che coinvolgono numeri, come "Zaino" e "Somma di Sottinsieme", per i quali peraltro esistono già algoritmi che forniscono la soluzione ottima in tempo pseudo-polinomiale (problemi non fortemente **NP**-completi).

Non per tutti i problemi di ottimizzazione si dispone di un algoritmo di approssimazione.

Infatti per alcuni problemi famosi come "**Cricca**", "**Colorabilità**" e "**Commesso viaggiatore**" trovare una soluzione approssimata è difficile, in termini di complessità, tanto quanto trovarne una ottima.

Teorema:
Sia *P* un problema di ottimizzazione tale che :

- la versione decisionale di *P* è fortemente **NP**-completa
- per ogni insieme I di dati in input per *P* il valore c(x*) della soluzione ottima è limitato superiormente da un polinomio *p* funzione del più grande numero che compare in I.

Allora, a meno che **P** non coincida con **NP**, non esiste alcun algoritmo di ε-approssimazione per *P* che richieda tempo polinomiale sia nella dimensione del problema che in 1/ε.