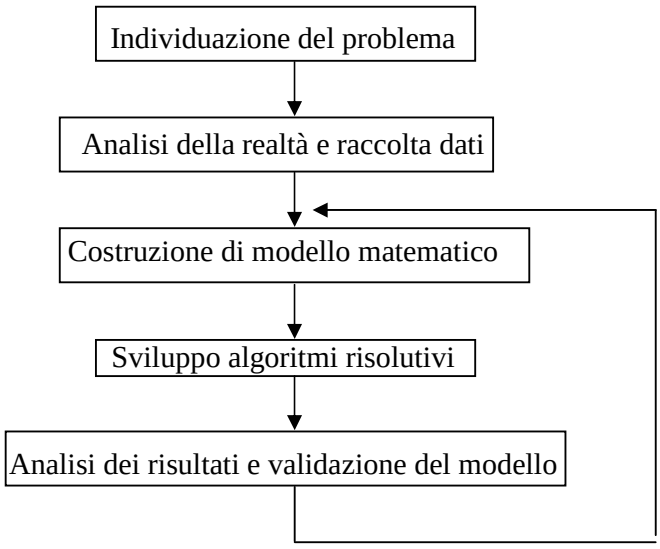
METODI MATEMATICI PER LA GESTIONE DELLE AZIENDE

Daniela Favaretto, Elena Moretti

Capitolo 1: Problema, modello, algoritmo. Complessità computazionale

La Ricerca Operativa si occupa della formulazione di problemi decisionali e dello studio e messa a punto di metodologie per la soluzione di tali problemi; si tratta di scegliere quali decisioni prendere per controllare in modo efficiente un sistema reale, studiando il problema nel suo complesso e utilizzando strumenti matematici che ne sfruttano le proprietà. Un *processo decisionale* può essere decomposto nelle seguenti fasi:

1) individuazione del problema,
2) analisi della realtà e raccolta dei dati,
3) costruzione di un modello matematico che descriva gli aspetti essenziali del problema,
4) sviluppo di metodologie matematiche efficienti (algoritmi risolutivi) per determinare una soluzione,
5) analisi dei risultati ottenuti, confronto con la realtà e validazione del modello.



Sono tutte fasi sequenziali anche se a volte può capitare che una di queste fasi richieda modifiche dei risultati ottenuti in fasi precedenti. In particolare, dopo aver analizzato i risultati ottenuti (fase 5), può succedere che ci sia la necessità di ritoccare il modello costruito nella fase 3. I punti 3) e 4) sono quelli più interessanti dal punto di vista matematico

1. Costruzione di un modello matematico

Il *modello* è una descrizione logico-matematica della porzione di realtà interessante ai fini del processo decisionale. Una volta descritto il sistema attraverso relazioni matematiche tra variabili che rappresentano gli elementi del sistema stesso, si tratta di cercare valori per tali variabili che soddisfino i vincoli imposti e che ottimizzino (massimizzino o minimizzino) una funzione obiettivo opportunamente definita. Non va mai dimenticata la diversità esistente tra problema reale e rappresentazione dello stesso tramite il modello: la soluzione di un problema è sempre la soluzione della rappresentazione del problema reale che è stata costruita, cioè del modello. Il modello è dunque in generale una descrizione molto limitata della realtà; nel formularlo si dovrà cercare di rappresentare in modo accurato gli aspetti che interessano ai fini della soluzione del problema decisionale che si sta affrontando.

Per *problema* si intende una domanda espressa in termini generali, la cui risposta dipende da un certo numero di parametri e variabili. Esso viene definito per mezzo di:

- una descrizione dei suoi parametri,
- una descrizione delle proprietà che devono caratterizzare la risposta o soluzione che si desidera.

(ES.: trovare il percorso più breve tra due città)

Un'*istanza* del problema *P* è una domanda specifica che si ottiene attribuendo un valore assegnato a ciascun parametro di *P*.

(ES.: trovare il percorso più breve tra Venezia e Padova)

Spesso un problema viene definito fornendo l'insieme *F* delle possibili risposte o soluzioni. Tale insieme viene detto *regione ammissibile* e i suoi elementi *soluzioni ammissibili*. Sulla regione ammissibile viene definita una *funzione obiettivo*

$$c : F \to \Re ,$$

che fornisce il costo o il beneficio associato ad ogni soluzione.

Un *problema di ottimizzazione* è un problema in cui, date la regione ammissibile *F* e la funzione obiettivo *c,* si cerca una soluzione ammissibile (elemento di *F*) che renda minima oppure massima la funzione obiettivo. Un problema di ottimizzazione verrà indicato usando la seguente notazione:

a) $(F, c; min)$ $\quad [min\{c(x): x \in F\}]$,

b) $(F, c; max)$ $\quad [max\{c(x): x \in F\}]$.

Un *problema di esistenza* (o *problema decisionale*) è un problema in cui si richiede di determinare una soluzione ammissibile e quindi di verificare se la regione ammissibile è vuota oppure no. Un problema di esistenza verrà indicato usando la seguente notazione:

c) $(F; \exists)$ [*determinare, se esiste,* $x \in F$].

I problemi a), b) e c) sono equivalenti nel senso che possono essere considerati rappresentazioni alternative di uno stesso problema. In seguito si dimostra infatti che ciascuna rappresentazione può essere trasformata nelle altre.

a) $\leftrightarrow$ b):

$$min\{c(x): x \in F\} \equiv - max\{-c(x): x \in F\}.$$

a) $\leftrightarrow$ c):

a) $\rightarrow$ c) sia $F^* = \{x \in F : c(x) \leq c(y), \forall y \in F\}$,

allora $(F, c; min) \equiv (F^*; \exists)$,

cioè determinare un punto di minimo per la funzione c nell'insieme F equivale a determinare un elemento di F^*;

a) $\leftarrow$ c) sia $c' : F' \rightarrow [0,1]$, $F' \supseteq F$, così definita

$$c'(x) = \begin{cases} 0, & \text{se } x \in F, \\ 1, & \text{altrimenti;} \end{cases}$$

allora $(F; \exists) \equiv (F', c'; min)$,

cioè determinare (se esiste) un elemento di F equivale a determinare un punto di minimo per la funzione c' in F': se il valore minimo di c' è 0 significa che $\exists x \in F$, se invece il valore minimo di c' è 1 significa che non $\exists x \in F$.

b) $\leftrightarrow$ c):

segue da a) $\leftrightarrow$ c) e da a) $\leftrightarrow$ b).

Dato un problema di ottimo del tipo $(F, c; min)$ [$(F, c; max)$], si dice problema decisionale ad esso associato, o sua versione decisionale, il problema $(F_k; \exists)$, dove $F_k = \{x \in F : c(x) \leq k\}$ [$F_k = \{x \in F : c(x) \geq k\}$], con k intero prefissato. Facendo variare il parametro k e risolvendo ogni volta un problema di esistenza è possibile determinare il valore ottimo della funzione obiettivo.

Dato un problema decisionale (F; $\exists$) si dice *problema di certificato* ad esso associato il problema seguente:

dato $\bar{x}$ *, determinare se esso è soluzione di (F; ∃), cioè se* $\bar{x} \in F$.

2. Esempi

Il problema dello zaino

Sia $E = \{1,...,n\}$ un insieme di oggetti che si vorrebbero caricare nello zaino. Siano a_i e c_i rispettivamente il peso e l'utilità dell'elemento i, $i \in \{1,...,n\}$. Si tratta di determinare un sottoinsieme di elementi di E che abbia utilità totale massima e il cui peso totale non superi la capacità dello zaino, b. Ovviamente, affinchè il problema non sia banale deve necessariamente valere la relazione seguente: $0 < b < \sum_{i=1}^{n} a_i$.

Il problema dello zaino si può formulare come problema di programmazione matematica come segue:

$$\begin{aligned} \max \ & \sum_{i=1}^{n} c_i x_i, \\ & \sum_{i=1}^{n} a_i x_i \le b, \\ & x_i \in \{0,1\}, \quad i = 1,...,n, \end{aligned}$$

dove $x_i = \begin{cases} 1, & \text{se l'elemento } i \text{ viene messo nello zaino,} \\ 0, & \text{altrimenti.} \end{cases}$

Il problema si può formulare come (F, c; *max*), dove $F = \left\{ S \subseteq E : \sum_{i \in S} a_i \le b \right\}$ e $c(S) = \sum_{i \in S} c_i$.

La versione decisionale del problema è la seguente:

$$\exists S \subseteq E : \sum_{i \in S} a_i \le b \ \text{ e } \ c(S) \ge k \, ?$$

Il problema di certificato associato è il seguente:

$$\text{dato } \bar{S} \subseteq E \text{, verificare che } \sum_{i \in \bar{S}} a_i \le b \ \text{ e che } \ c(\bar{S}) \ge k \, .$$

Ordinamento di lavori su macchine: minimizzazione del tempo di completamento

Siano dati *n* lavori e *m* macchine. Il lavoro *i, i*=1,…,*n,* può essere svolto su ciascuna macchina e richiede un tempo di lavorazione d_i, indipendente dalla macchina su cui viene svolto. Si tratta di assegnare i lavori alle macchine in modo da completare i lavori nel minimo tempo possibile.

Il problema si può formulare come (*F*, *c*; *min*), dove, dato $E = \{J \subseteq \{1,...,n\}\}$,

$$F = \left\{ I \subseteq E : J_1, J_2 \in I \Rightarrow J_1 \cap J_2 = \varnothing,\ \bigcup_{J \in I} J = \{1,...n\},\ |I| \le m \right\},$$

$$c(I) = \max_{J \in I} \left\{ \sum_{i \in J} d_i \right\}.$$

Il problema del commesso viaggiatore (TSP)

Dato un grafo completo, simmetrico e ad archi pesati, (N, E, w), con

N insieme dei nodi, $|N| = n$,

E insieme degli archi, $|E| = \dfrac{n(n-1)}{2}$,

$w : E \to \Re_+$ funzione che associa ad ogni arco un peso,

si tratta di determinare un circuito hamiltoniano, cioè un circuito che passa per tutti i nodi del grafo una e una sola volta, di costo minimo.

Il problema si formula come (*H*, *w*; *min*), dove

$$H = \{I \subseteq E : I \text{ individua un circuito hamilton.}\} \text{ e } w(I) = \sum_{e \in I} w(e).$$

Soddisfacibilità (SAT)

Siano $P_1, P_2, ..., P_n$ delle proposizioni logiche. Si indichi con $\pm P_j$ la proposizione P_j o la sua negazione $\neg P_j$, per *j*=1,2,…,*n*. Siano $C_i = \pm P_1 \vee \pm P_2 \vee ... \vee \pm P_r$, per *i*=1,…,*m* e sia $A = C_1 \wedge C_2 \wedge ... \wedge C_m$. Si tratta di determinare, se esiste, una valore di verità per le proposizioni $P_1, P_2, ..., P_n$ che renda vera la formula *A*.

Come problema di esistenza:

$(F, \exists)$, dove

F è l'insieme degli assegnamenti di verità che rendono vera la *A*.

Come problema di ottimo:

$(F', \nu, \min)$, dove

$F' = \{falso, vero\}^n$ e $\{falso, vero\}^n$ indica il prodotto cartesiano di $\{falso, vero\}$ per *n* volte;

ν è la funzione che conta il numero di clausole non soddisfatte.

Ovviamente una formula è soddisfacibile se e solo se il valore ottimo del problema di ottimo associato $(F', \nu, \min)$ è zero.

3. Algoritmi e complessità

Una volta che è stato formulato un problema, si tratta di risolverlo e quindi di mettere a punto strumenti di calcolo che, data una qualsiasi istanza, siano in grado di fornire la soluzione in un tempo finito. Tali strumenti si dicono algoritmi. Un *algoritmo* che risolve un dato problema *P* è una sequenza finita di istruzioni che, applicata ad una qualsiasi istanza *p* di *P*, si arresta dopo un numero finito di passi fornendo una soluzione dell'istanza *p* o indicando che l'istanza *p* non ha soluzioni ammissibili.

Dato un problema *P*, una sua istanza *p* e un algoritmo *A* che risolve *P*, si dice *costo* (o *complessità*) di *A* applicato a *p* una misura delle risorse utilizzate dall'esecuzione di *A* su una macchina per determinare la soluzione di *p*. Le risorse sono di due tipi: la memoria occupata e il tempo di calcolo. Poiché generalmente è il tempo di calcolo la risorsa più critica, è usuale utilizzare il tempo di calcolo come misura della complessità degli algoritmi. Supponendo che tutte le operazioni elementari (operazioni algebriche e confronti) abbiano la stessa durata, si esprime il tempo di calcolo come numero di operazioni elementari effettuate dall'algoritmo.

Dato un algoritmo è utile disporre di una misura di complessità che consenta di valutare la bontà dell'algoritmo stesso ed eventualmente di confrontare questo con algoritmi alternativi. Ovviamente non è possibile conoscere la complessità di un algoritmo *A* per ognuna delle istanze di *P*, poiché queste in generale sono in numero non limitato. Si cerca allora di esprimere la complessità come una funzione *g*(*n*) che rappresenta il costo necessario per risolvere la più difficile tra le istanze di dimensione *n*. Si parla in tal caso di *complessità nel caso peggiore*. Per *dimensione* di un'istanza *p* si intende una misura del numero di bit necessari per rappresentare su calcolatore i dati che definiscono *p*, cioè una

misura della lunghezza del suo input. In realtà, più che la conoscenza esatta della funzione $g(n)$ interessa il suo ordine di grandezza; si parla in tal caso di *complessità asintotica*. Data una funzione $g(x)$, si dice che $g(x)$ è $O(f(x))$ se esistono due costanti c_1 e c_2 tali che $g(x) \leq c_1 f(x) + c_2$. Sia $g(x)$ il numero di operazioni elementari che vengono effettuate da un algoritmo A applicato alla più difficile istanza, tra tutte quelle che hanno lunghezza di input x, di un dato problema P. Si dice che la complessità di A è $O(f(x))$ se $g(x)$ è $O(f(x))$.

Per avere un esempio dell'utilizzo della teoria della complessità, si descrivono di seguito due diversi algoritmi per l'ordinamento di un insieme di numeri.

Esempio

Dato un insieme di numeri $I = \{a_1, a_2, ..., a_n\}$ a due a due diversi (non è un'ipotesi restrittiva), si vogliono ordinare gli elementi di I in senso crescente. Si presentano di seguito due diverse tecniche: l'algoritmo degli scambi e l'algoritmo di ordinamento per sottoinsiemi.

Algoritmo degli scambi

Si confrontano a_1 e a_2. Se $a_1 < a_2$, si lascia invariata la loro posizione, altrimenti si scambiano tra di loro. Si confronta poi a_3 con l'elemento che si trova in seconda posizione e se serve con l'elemento che si trova in prima posizione, per trovare la giusta collocazione di a_3 rispetto ad a_1 e a_2. In generale, al k-esimo passo si tratterà di trovare la giusta collocazione per a_{k+1} rispetto ai primi k elementi già ordinati; siano essi $b_1, \ldots, b_k$. Si tratta quindi di confrontare a_{k+1} con b_k, poi se occorre con b_{k-1}, e così via fino a che si individua j tale che $b_j < a_{k+1} < b_{j+1}$. Ma quanti confronti occorrono complessivamente? Ovviamente questo numero dipende da come gli n numeri sono ordinati inizialmente. Volendo, come è usuale fare, valutare il numero di operazioni elementari che occorrono nel caso peggiore, bisogna supporre che gli n numeri da ordinare dal più piccolo al più grande siano inizialmente ordinati dal più grande al più piccolo. In questo modo ogni elemento sarà costretto a cambiare posizione con tutti gli elementi alla sua sinistra prima di trovare la giusta collocazione rispetto a questi: al k-esimo passo a_{k+1} subirà k confronti e altrettanti scambi. In totale i confronti sono (l'uguaglianza si dimostra per induzione):

$$1 + 2 + \ldots + (n-1) = \frac{n(n-1)}{2}.$$

Quindi il numero di operazioni elementari da fare nel caso peggiore è un polinomio di grado 2 in *n*, dove *n* è il numero di elementi da ordinare e quindi la dimensione dell'istanza. Si dice allora che questo algoritmo ha complessità $O(n^2)$.

Algoritmo di ordinamento per sottoinsiemi (Merge)

Si intende illustrare questa seconda procedura mediante un esempio. Si consideri l'insieme *I* costituito da $16=2^4$ elementi da ordinare:

I={20, 8, 11, 3, 15, 7, 30, 2, 18, 31, 4, 25, 9, 40, 10, 35}.

Sia I_1 l'insieme che ha come elementi le 8 coppie consecutive ordinate di elementi consecutivi di *I*:

I_1={8, 20, 3, 11, 7, 15, 2, 30, 18, 31, 4, 25, 9, 40, 10, 35}.

Sia I_2 l'insieme che ha come elementi i 4 insiemi consecutivi ordinati formati ciascuno da 4 elementi consecutivi di I_1:

I_2={3, 8, 11, 20, 2, 7, 15, 30, 4, 18, 25, 31, 9, 10, 35, 40}.

Sia I_3 l'insieme che ha come elementi i 2 insiemi consecutivi ordinati formati ciascuno da 8 elementi di I_2:

I_3={2, 3, 7, 8, 11, 15, 20, 30, 4, 9, 10, 18, 25, 31, 35, 40}.

Infine sia I_4 l'insieme che si ottiene da I_3 ordinando tutti i 16 elementi:

I_4={2, 3, 4, 7, 8, 9, 10, 11, 15, 18, 20, 25, 30, 31, 35, 40}.

E' evidente che se *I* contiene 2^k elementi (nell'esempio *I* contiene $2^4=16$ elementi) si dovranno costruire *k* insiemi, I_1, I_2, …, I_k (nell'esempio I_1, I_2, I_3, I_4). Quindi con $n=2^k$ elementi si costruiscono $k=\log_2 n$ insiemi (nell'esempio *n*=16, *k*=4). Per costruire I_{h+1} a partire da I_h si fondono a due a due dei sottoinsiemi adiacenti formati da 2^h elementi, ciascuno già ordinato in precedenza. Questa fase richiede al più tante operazioni quanti sono gli elementi da fondere ($\leq n$), infatti per fondere a_1, a_2, …, a_t con b_1, b_2, …, b_t si costruisce un elenco di 2*t* elementi e si procede come segue: si confrontano a_1 e b_1 e se $a_1<b_1$ si trascrive a_1 e si confrontano a_2 e b_1, altrimenti, se $a_1>b_1$ si trascrive b_1 e si confrontano a_1 e b_2. In generale, quando si confrontano a_i e b_j, se $a_i<b_j$ si trascrive a_i e si confrontano a_{i+1} e b_j, altrimenti, se $a_i>b_j$ si trascrive b_j e si confrontano a_i e

b_{j+1}. Poiché ad ogni confronto si trascrive sempre almeno un elemento, I_{h+1} si ottiene da I_h dopo non più di *n* confronti. Di conseguenza l'algoritmo richiede non più di $n\log_2 n$ operazioni. Si noti che la tecnica non cambia se il numero di elementi da ordinare *n* è un intero qualunque e non una potenza di 2; si tratterà eventualmente di fondere gruppi aventi numerosità diversa.

Tra le due tecniche descritte è generalmente preferita la seconda in quanto presenta una complessità inferiore.

Si dicono *trattabili* i problemi per i quali esistono algoritmi la cui complessità è O(*p*(*x*)), con *p*(*x*) un polinomio in *x*, e *intrattabili* i problemi per i quali un tale algoritmo non si conosce. La tabella 1 mostra, per diverse funzioni di complessità, come aumentano i tempi di calcolo all'aumentare della dimensione dell'istanza, nell'ipotesi che il tempo richiesto per effettuare una operazione elementare mediante calcolatore sia 10^{-6} *sec*. Si noti che le funzioni di complessità esponenziali, pur avendo tempi di calcolo confrontabili con quelli polinomiali per dimensione bassa dell'istanza, presentano tempi di calcolo molto elevato all'aumentare della dimensione dell'istanza.

	n=10	***n=20***	***n=40***	***n=60***
G(n)=n	10^{-5}*sec*	$2\cdot10^{-5}$*sec*	$4\cdot10^{-5}$*sec*	$6\cdot10^{-5}$*sec*
G(n)=n³	10^{-3}*sec*	$8\cdot10^{-3}$*sec*	$6{,}4\cdot10^{-2}$*sec*	$2{,}16\cdot10^{-1}$*sec*
G(n)=n⁵	10^{-1}*sec*	*3,2sec*	*102,4sec*	*777,6sec*
			(1,7min)	*(13min)*
G(n)=2ⁿ	$1{,}024\cdot10^{-3}$*sec*	*1,049sec*	*1099512sec*	$1{,}153\cdot10^{12}$*sec*
			(12,7gg)	*(36500anni)*
G(n)=3ⁿ	$5{,}9\cdot10^{-2}$*sec*	*3486,8sec*	$1{,}22\cdot10^{13}$*sec*	$4{,}24\cdot10^{22}$*sec*
		(0,97ore)	$(3{,}9\cdot10^{5}$*anni)*	$(1{,}3\cdot10^{15}$*anni)*

Tabella 1: tempi di calcolo con diverse funzioni di complessità e diverse dimensioni dell'istanza, nell'ipotesi che il tempo richiesto per una operazione elementare sia 10^{-6}sec

Si definisce ***classe NP*** l'insieme di tutti i problemi decisionali il cui associato problema di certificato può essere risolto in tempo polinomiale. Ad esempio il problema

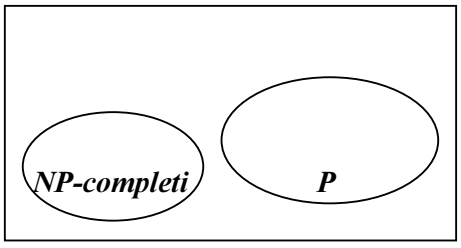
del commesso viaggiatore è un problema della classe *NP* in quanto il problema di certificato associato ad esso può essere risolto con un algoritmo che ha complessità $O(n)$ ovvero in tempo lineare. Nella classe *NP* è contenuta la ***classe P***, cioè l'insieme di tutti i problemi decisionali per i quali esistono algoritmi risolutivi di complessità polinomiale. I problemi per i quali esistono algoritmi risolutivi di complessità polinomiale si dicono *problemi facili*.

Dati due problemi decisionali *P* e *Q*, si dice che *P* si riduce in tempo polinomiale a *Q*, e si scrive $P \propto Q$, se, supponendo che esista un algoritmo A_Q che risolve *Q* in tempo costante (indipendente dalla lunghezza dell'input), allora esiste un algoritmo che risolve in tempo polinomiale *P* utilizzando come sottoprogramma A_Q. Si parla in tal caso di *riduzione polinomiale* di *P* a *Q*. La relazione $\propto$ ha le seguenti proprietà:

i) è riflessiva, cioè $P \propto P$;

ii) è transitiva, cioè $P \propto Q$ e $Q \propto R \Rightarrow P \propto R$.

Un problema *Q* è detto ***NP-completo*** (o *difficile*) se $Q \in NP$ e se per ogni $P \in NP$ si ha che $P \propto Q$. E' possibile dimostrare che (teorema di Cook, 1971) ogni problema $P \in NP$ si riduce polinomialmente al problema della soddisfacibilità (*SAT*), quindi la classe dei problemi *NP-completi* è non vuota, poiché *SAT* è un problema *NP-completo*. Se esistesse un algoritmo polinomiale che risolve un problema *NP-completo*, allora tutti i problemi in *NP* sarebbero risolubili in tempo polinomiale e si avrebbe $P = NP$. A tutt'oggi non si sa se esistono problemi in *NP* che non appartengano anche a *P*, cioè se $P \neq NP$. Si ritiene che sia molto probabile che $P \neq NP$.



Classe NP

Un problema che abbia come caso particolare un problema *NP-completo* ha la proprietà di essere almeno tanto difficile quanto i problemi *NP-completi*. Un problema di questo tipo si dice ***NP-hard***. Ad esempio sono *NP-hard* i problemi di ottimo la cui versione decisionale è un problema *NP-completo*.

In molti casi, a causa della complessità del problema, può risultare troppo oneroso (se non addirittura impossibile) determinare una soluzione ottima del problema. Ci si accontenta allora di cercare delle “buone” soluzioni ammissibili. Un algoritmo che risolve un problema senza garantirne l’ottimalità della soluzione trovata è detto *algoritmo euristico*. E’ evidente che un buon algoritmo euristico deve avere bassa complessità e deve essere in grado di fornire una soluzione che approssima bene la soluzione ottima.

Capitolo 2: Programmazione Lineare Intera (PLI)

Un problema di programmazione lineare intera (*PLI*) è un problema di programmazione lineare con l'ulteriore restrizione che le variabili possono assumere solo valori interi. Il generale *PLI* può essere scritto come segue:

$$\begin{aligned} \min\ & \underline{c} \cdot \underline{x}, \\ & A\underline{x} = \underline{b}, \\ & \underline{x} \geq \underline{0},\ \ x_i \text{ intero},\ i \in \{1,...,n\}, \end{aligned}$$

dove $\underline{x}, \underline{c} \in \Re^n$, $\underline{b} \in \Re^m$ e A è una matrice $m \times n$.

Altre forme si possono avere per esempio con vincoli di disuguaglianza. L'introduzione dei vincoli di interezza avviene quando non è consentito avere soluzioni con componenti non intere, per il significato delle variabili del problema. Per esempio in alcuni problemi di produzione i livelli di produzione sono caratterizzati da unità di prodotti.

Si noti che la Regione Ammissibile (RA) di un problema PLI è un insieme discreto di punti e quindi ***non*** *è un insieme convesso. Di conseguenza la teoria sviluppata per i problemi PL non può essere applicata ai problemi PLI.*

In certi casi solo alcune variabili sono vincolate ad essere intere, mentre le altre possono assumere valori reali non negativi qualsiasi: si ha allora un problema di programmazione lineare mista intera (*PLMI*) la cui forma generale è la seguente

$$\begin{aligned} \min\ & (\underline{c} \cdot \underline{x} + \underline{d} \cdot \underline{y}), \\ & A\underline{x} + B\underline{y} = \underline{b}, \\ & \underline{x} \geq \underline{0},\ \ x_i \text{ intero},\ i \in \{1,...,n\}, \\ & \underline{y} \geq \underline{0}, \end{aligned}$$

dove $\underline{x}, \underline{c} \in \Re^n$, $\underline{y}, \underline{d} \in \Re^k$, $\underline{b} \in \Re^m$, A è una matrice $m \times n$ e B è una matrice $m \times k$. Si dimostra che i problemi *PLI* e *PLMI* sono problemi difficili (*NP-hard*).

In seguito si vedrà che per risolvere un problema *PLI* non è possibile ricorrere alla tecnica intuitiva di risolvere il problema *PL* ad esso associato e poi approssimare all'intero superiore o inferiore le eventuali componenti non intere del vettore soluzione: sono necessari metodi di soluzione specifici (ad hoc).

1. Esempi di problemi PLI

Problema dello zaino (knapsack)

Il problema dello zaino già presentato in precedenza può essere riformulato come *PLI* nelle seguenti due varianti.

- *È possibile mettere nello zaino più oggetti dello stesso tipo*:

  Sia x_i il numero di oggetti di tipo *i* caricati nello zaino. Il problema si formula come segue:

$$\begin{aligned} \max_{x} \ & \sum_{i=1}^{n} c_i x_i, \\ & \sum_{i=1}^{n} a_i x_i \le b, \\ & x_i \text{ intero}, \ x_i \ge 0, \ i = 1,...,n. \end{aligned}$$

  Le varibili decisionali sono x_i, cioè il numero di oggetti di tipo *i* da caricare nello zaino.

- *Non è possibile mettere nello zaino più oggetti dello stesso tipo*:

$$\underline{Sia}\ x_i = \begin{cases} 1, & \text{se l'oggetto } i\text{-esimo viene caricato}, \\ 0, & \text{altrimenti.} \end{cases}$$

Il problema si formula come segue:

$$\begin{aligned} \max_{x} \ & \sum_{i=1}^{n} c_i x_i, \\ & \sum_{i=1}^{n} a_i x_i \le b, \\ & x_i \in \{0,1\}, \ i = 1,...,n. \end{aligned}$$

  Le varibili decisionali sono x_i, cioè si tratta di decidere se caricare o no l'oggetto *i*-esimo nello zaino. Questo è un problema *PLI*-{0,1}; le variabili decisionali x_i sono dette variabili booleane.

Sequencing di lavori su una macchina

Una unica macchina deve eseguire n lavori. Si tratta di decidere quando iniziare ogni lavoro, in modo da minimizzare il tempo di completamento, rispettando alcuni vincoli che riguardano:

1. eventuali precedenze tra lavori;
2. il fatto che la macchina può svolgere un solo lavoro alla volta;
3. il rispetto dei termini di tempo imposti.

Siano

x_j l'istante di inizio del lavoro j-esimo,

p_j la durata del lavoro j-esimo,

d_j l'istante entro il quale dovrebbe essere completato il lavoro j-esimo (*due date*).

I vincoli 1, 2 e 3 si possono esprimere come segue:

Vincoli 1:

se il lavoro i deve precedere il lavoro j si scrive

$$x_i + p_i \le x_j .$$

Vincoli 2:

$$\begin{cases} x_i - x_j \ge p_j, & \text{se } j \text{ precede } i, \\ x_j - x_i \ge p_i, & \text{se } i \text{ precede } j. \end{cases} \tag{1}$$

Si tratta di formalizzare "se j precede i" e quindi "se i precede j". Siano

M un numero positivo sufficientemente grande,

$$y_{ij} = \begin{cases} 0, & \text{se } i \text{ precede } j, \\ 1, & \text{altrimenti.} \end{cases}$$

I vincoli (1) possono essere sostituiti da

$$\begin{cases} M(1 - y_{ij}) + (x_i - x_j) \ge p_j, \\ My_{ij} + (x_j - x_i) \ge p_i, \end{cases} \tag{2}$$

infatti

se $y_{ij} = 0$ (cioè *i* precede *j*), il primo vincolo di (2) è banalmente soddisfatto, mentre il secondo è efficace e coincide con il secondo vincolo di (1);

se $y_{ij} = 1$ (cioè *j* precede *i*), il secondo vincolo di (2) è banalmente soddisfatto, mentre il primo è efficace e coincide con il primo vincolo di (1).

Vincoli 3:

deve essere:

$$x_j + p_j \le d_j, \quad j \in \{1,...,n\}.$$

Indicato con *t* il tempo di completamento di tutti i lavori, il problema si formula come segue:

$$\begin{aligned}
&\min\ t,\\
&\quad x_j + p_j \le t, \quad j \in \{1,...,n\},\\
&\quad My_{ij} + (x_j - x_i) \ge p_i, \quad i, j \in \{1,...,n\},\\
&\quad M(1 - y_{ij}) + (x_i - x_j) \ge p_j, \quad i, j \in \{1,...,n\},\\
&\quad x_j + p_j \le d_j, \quad j \in \{1,...,n\},\\
&\quad x_j \ge 0, \quad j \in \{1,...,n\},\\
&\quad y_{ij} \in \{0,1\}, \ i, j \in \{1,...,n\}.
\end{aligned}$$

Job-shop scheduling

E' una versione più complicata del problema precedente in quanto ci sono a disposizione *m* macchine (anziché 1) per eseguire gli *n* lavori.

Siano

p_j la durata del lavoro *j*-esimo (processing time), *j*=1,…,*n*;

$a_{ij} = \begin{cases} 1, & \text{se il lavoro } j \text{ può essere eseguito sulla macchina } i, \\ 0, & \text{altrimenti,} \end{cases}$ *i*=1,.. ,*m*, *j*=1,…,*n*;

d_j l'istante data entro il quale il lavoro *j* deve essere completato (due date),

j=1,…,*n*.

Si supponga che ci sia un ordinamento di precedenza tra i lavori.

L'obiettivo è quello di minimizzare il tempo di completamento *t* (min *t*).

Le variabili decisionali sono le seguenti:

s_j, tempo di inizio del lavoro j, j=1,…,n;

$$x_{ij} = \begin{cases} 1, & \text{se il lavoro } j \text{ è eseguito sulla macchina } i, \\ 0, & \text{altrimenti,} \end{cases} \quad i=1,\ldots,m,\ j=1,\ldots,n.$$

I vincoli sono i seguenti:

1. Ogni lavoro deve essere eseguito da una e una sola macchina:

$$\sum_{i=1}^{m} x_{ij} = 1, \quad j = 1,...,n\,.$$

2. Ogni lavoro deve essere concluso entro t:

$$s_j + p_j \le t, \quad j = 1,...,n\,.$$

3. x_{ij} può essere diverso da 0 (cioè il lavoro j è eseguito sulla macchina i) solo se a_{ij} è diverso da 0 (cioè il lavoro j può essere eseguito sulla macchina i):

$$x_{ij} \le a_{ij}, \quad i=1,\ldots,m,\ j=1,\ldots,n.$$

Ciò significa che se $a_{ij} = 0$, allora $x_{ij} = 0$, mentre se $a_{ij} = 1$, allora può essere $x_{ij} = 0$ oppure $x_{ij} = 1$.

4. Due operazioni non possono essere eseguite simultaneamente sulla stessa macchina. Se $x_{ij} = x_{ik} = 1$, cioè sia il lavoro j che il lavoro k sono eseguiti sulla stessa macchina i, e il lavoro j precede il lavoro k, allora deve essere

$$s_j + p_j \le s_k\,,$$

cioè il lavoro k può cominciare solo dopo che si è concluso il lavoro j.

Si tratta di formalizzare "se j precede k". Siano

$$\delta_{jk} = \begin{cases} 1, & \text{se } j \text{ deve essere eseguito prima di } k, \\ 0, & \text{altrimenti,} \end{cases}$$

$$T = \sum_{j=1}^{n} p_j\,.$$

Ovviamente

$$\delta_{jk} + \delta_{kj} = 1\,,\ j,k=1,\ldots,n.$$

Il seguente vincolo esprime il fatto che due lavori non possono essere eseguiti contemporaneamente sulla stessa macchina:

$$s_j + p_j - s_k \le T(1-\delta_{jk} + 2 - x_{ij} - x_{ik}),\ j \ne k,\ i = 1,...,m, j,\ k = 1,...,n.$$

Il vincolo è efficace, e diventa $s_j + p_j \le s_k$, solo se

$$\delta_{jk} = 1 \quad \text{e} \quad x_{ij} = x_{ik} = 1,$$

cioè solo se j precede k ($\delta_{jk} = 1$) e sia j che k sono eseguiti sulla macchina i ($x_{ij} = x_{ik} = 1$). In tutti gli altri casi il vincolo è banalmente soddisfatto.

5. Si deve tener conto delle *due date*:

$$s_j + p_j \le d_j,\ j=1,\ldots,n.$$

Riassumendo il problema si formula come segue:

$$\begin{aligned}
&\min t \\
&s_j + p_j \le t, \quad j \in \{1,...,n\}, \\
&\sum_{i=1}^{m} x_{ij} = 1, \quad j \in \{1,...,n\}, \\
&x_{ij} \le a_{ij}, \quad i \in \{1,...,m\}, j \in \{1,...,n\}, \\
&s_j + p_j - s_k \le T(1-\delta_{jk} + 2 - x_{ij} - x_{ik}), \quad i \in \{1,...,m\}, j,k \in \{1,...,n\}, \ j \ne k, \\
&s_j + p_j \le d_j, \quad j \in \{1,...,n\}, \\
&s_j \ge 0, \quad j \in \{1,...,n\}, \\
&x_{ij} \in \{0,1\}, \quad i \in \{1,...,m\}, j \in \{1,...,n\}.
\end{aligned}$$

Il problema di assegnazione

Si suppone che n lavori debbano essere eseguiti da un insieme di n macchine, in modo che

- tutti i lavori risultino eseguiti,
- tutte le macchine risultino impiegate.

Si tratta di assegnare ogni lavoro ad una ed una sola macchina ed ogni macchina ad uno ed un solo lavoro in modo da minimizzare i costi. Siano

c_{ij} il costo che si deve sopportare se la macchina i esegue il lavoro j,

$$x_{ij} = \begin{cases} 1, & \text{se il lavoro } j \text{ è eseguito sulla macchina } i, \\ 0, & \text{altrimenti.} \end{cases}$$

→ *Obiettivo*: minimizzare i costi. Di conseguenza la funzione obiettivo risulta:

$$\min \sum_{i=1}^{n} \sum_{j=1}^{n} c_{ij} x_{ij} \ .$$

→ *Vincoli*:

- ogni lavoro *j* deve essere eseguito su una e una sola macchina *i*:

$$\sum_{i=1}^{n} x_{ij} = 1, \quad j \in \{1,...,n\};$$

- ogni macchina *i* deve eseguire uno ed un solo lavoro *j*:

$$\sum_{j=1}^{n} x_{ij} = 1, \quad i \in \{1,...,n\}.$$

Il problema si formula come segue:

$$\begin{aligned} &\min \sum_{i=1}^{n} \sum_{j=1}^{n} c_{ij} x_{ij} \\ &\sum_{i=1}^{n} x_{ij} = 1, \quad j \in \{1,...,n\}, \\ &\sum_{j=1}^{n} x_{ij} = 1, \quad i \in \{1,...,n\}, \\ &x_{ij} \in \{0,1\}, \quad i, j \in \{1,...,n\}. \end{aligned}$$

In forma compatta il problema si formula come segue:

$$\begin{aligned} &\min \underline{c} \cdot \underline{x} \\ &A\underline{x} = \underline{1}, \\ &x_{ij} \in \{0,1\}, \end{aligned}$$

con

$\underline{x} = (x_{11}, x_{12},..., x_{1n}, x_{21}, x_{22},..., x_{2n},..., x_{n1}, x_{n2},..., x_{nn})$, $\underline{x} \in \Re^{n \times n}$,

$\underline{c} = (c_{11}, c_{12},..., c_{1n}, c_{21}, c_{22},..., c_{2n},..., c_{n1}, c_{n2},..., c_{nn})$, $\underline{c} \in \Re^{n \times n}$,

$\underline{1} \in \Re^{2n}$ con tutte componenti uguali a 1,

A matrice $2n \times n^2$ così strutturata:

$$\left|\begin{array}{cccccccccccc}
1 & 1 & ... & 1 & 0 & 0 & ... & 0 & ... & 0 & 0 & ... & 0 \\
0 & 0 & ... & 0 & 1 & 1 & ... & 1 & ... & 0 & 0 & ... & 0 \\
 & & ... & & & & ... & & ... & & & ... & \\
0 & 0 & ... & 0 & 0 & 0 & ... & 0 & ... & 1 & 1 & ... & 1 \\
1 & 0 & ... & 0 & 1 & 0 & ... & 0 & ... & 1 & 0 & ... & 0 \\
0 & 1 & ... & 0 & 0 & 1 & ... & 0 & ... & 0 & 1 & ... & 0 \\
 & & ... & & & & ... & & ... & & & ... & \\
0 & 0 & ... & 1 & 0 & 0 & ... & 1 & ... & 0 & 0 & ... & 1
\end{array}\right|$$

Il problema di Set Covering

Siano

$T = \{t_1, t_2, ..., t_n\}$ $\quad |T| = n$,

$F = \{J_1, J_2, ..., J_m\}$, $J_j \subseteq T$, $j = 1, ..., m$, una famiglia di sottoinsiemi di *T*.

Definizione $F' \subseteq F$ si dice *copertura* di *T* se

$$\bigcup_{J_j \in F'} J_j = T .$$

Il problema di *set covering* consiste nel determinare una copertura di *T* di cardinalità minima.

Siano

$$x_j = \begin{cases} 1, & \text{se } J_j \in F', \\ 0, & \text{se } J_j \notin F', \end{cases}$$ (variabili decisionali)

e

$$e_{ij} = \begin{cases} 1, & \text{se } t_i \in J_j, \\ 0, & \text{se } t_i \notin J_j. \end{cases}$$

E' ovvio che *F'* è una copertura se e solo se

$$\sum_{j=1}^{m} e_{ij} x_j \geq 1, \quad i = 1, ..., n,$$

cioè se e solo se ogni elemento t_i appartiene ad almeno un insieme J_j ($e_{ij} = 1$) che appartiene a F' ($x_j = 1$).

Il problema si formula come problema di programmazione lineare booleana nel modo seguente:

$$\min \sum_{j=1}^{m} x_j$$
$$\sum_{j=1}^{m} e_{ij} x_j \geq 1, \; i = 1,...,n,$$
$$x_j \in \{0,1\}, \; j = 1,...,m.$$

Una variante del Set Covering è il *Set Covering pesato*. Si supponga che ad ogni $J_j \in F$ sia associato un peso (costo) w_j. Si tratta di trovare una copertura di F che abbia peso (costo) minimo. Il problema si formula come problema di programmazione lineare booleana nel modo seguente:

$$\min \sum_{j=1}^{m} w_j x_j$$
$$\sum_{j=1}^{m} e_{ij} x_j \geq 1, \; i = 1,...,n,$$
$$x_j \in \{0,1\}, \; j = 1,...,m.$$

***Un'altra variante del Set Covering è il* Set Partitioning.**

Definizione $F' \subseteq F$ ***si dice partizione di T se***

$$\bigcup_{J_j \in F'} J_j = T, \text{ con } J_j \cap J_k = \varnothing, \text{ per ogni } J_j, J_k \in F', \; j \neq k .$$

Si tratta di determinare una partizione di T di cardinalità minima.

Il problema si formula come problema di programmazione lineare intera nel modo seguente:

$$\min \sum_{j=1}^{m} x_j$$
$$\sum_{j=1}^{m} e_{ij} x_j = 1, \; i = 1,...,n,$$
$$x_j \in \{0,1\}, \; j = 1,...,m.$$

Esempio

Siano

$$T = \{1,2,3,4,5\},\ |T| = n = 5,$$

$$F = \{\{1,2\},\{1,3,5\},\{2,4,5\},\{3\},\{1\},\{4,5\}\},\ |\mathrm{F}| = m = 6.$$

La famiglia $F' = \{\{1,2\},\{1,3,5\},\{2,4,5\}\} \subseteq F$ è una copertura di *T*, infatti

$$\bigcup_{J_j \in F'} J_j = T.$$

In questo caso la matrice degli elementi e_{ij} è la seguente

$$E = [e_{ij}] = \begin{vmatrix} 1 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 \end{vmatrix}$$

F' non è la copertura di cardinalità minima infatti $F'' = \{\{1,3,5\},\{2,4,5\}\} \subseteq F$ è anch'essa una copertura e $|F''| = 2 < |F'| = 3$.

La famiglia $F''' = \{\{1,2\},\{3\},\{4,5\}\} \subseteq F$ è una partizione di *T*, infatti

$$\bigcup_{J_j \in F'''} J_j = T,\ \text{con } J_j \cap J_k = \varnothing, \text{per ogni } J_j, J_k \in F''',\ j \neq k.$$

Problema del carico fisso

Si consideri il seguente problema (non lineare):

$$\begin{aligned} \min_{\underline{x}}\ & \sum_{i=1}^{n} \tilde{c}_i(x_i), \\ & A\underline{x} = \underline{b}, \\ & \underline{x} \geq \underline{0},\ x_i \text{ intero},\ i \in \{1,\ldots,n\}, \end{aligned} \tag{3}$$

con

$$\tilde{c}_i(x_i) = \begin{cases} 0, & \text{se } x_i = 0, \\ k_i + c_i x_i, & \text{se } x_i \neq 0. \end{cases}$$

Esso ha un'interpretazione in termini di programmazione della produzione:

Si devono produrre oggetti di *n* tipi. Siano:

k_i il costo fisso che interviene quando si attiva la produzione dell'*i*-esimo tipo e non interviene invece se la *i*-esima produzione non è attivata;

c_i il costo di produzione di una unità dell'*i*-esimo tipo;

x_i il numero di unità di prodotto *i*-esimo che si intendono produrre.

Si tratta di minimizzare il costo complessivo di produzione rispettando prefissati vincoli, cioè i vincoli che compaiono in (3).

Tale problema non lineare può essere trasformato in un problema *PLI* nel modo seguente.

Sia

$$y_i = \begin{cases} 1, & \text{se } x_i \neq 0, \\ 0, & \text{se } x_i = 0. \end{cases}$$

Sia *M* un numero intero positivo sufficientemente grande (uguale per esempio alla massima capacità produttiva) e si considerino i vincoli

$$x_i \leq M y_i .$$

Se $y_i = 0$, cioè non è attivata la produzione dell'*i*-esimo tipo, allora $x_i = 0$.

Se $y_i = 1$, cioè è attivata la produzione dell'*i*-esimo tipo, allora $x_i \leq M$ è banalmente soddisfatto.

Il problema originale è quindi equivalente al problema *PLI*:

$$\begin{aligned} \min \ & \sum_{i=1}^{n} (c_i x_i + k_i y_i), \\ & A\underline{x} = \underline{b}, \\ & 0 \leq x_i \leq M y_i, \ i \in \{1,...,n\}, \\ & x_i \text{ intero}, \ i \in \{1,...,n\}, \\ & y_i \in \{0,1\}, \ i \in \{1,...,n\}. \end{aligned}$$

Esempio

In una centrale elettrica sono a disposizione tre generatori e ogni giorno si deve decidere quali usare di giorno e quali di notte per assicurare una produzione di almeno 4000 megawatt di giorno e di almeno 2800 megawatt di notte. L'uso di un generatore comporta la presenza di personale tecnico che sorvegli il suo funzionamento. Tale personale viene retribuito in maniera diversa tra il giorno e la notte a seconda del tipo di generatore. Nella tabella che segue sono riportati tali costi

di attivazione (in migliaia di lire), il costo per ogni megawatt prodotto e la capacità massima di produzione in megawatt.

	Costo di att. giorno	**Costo di att. notte**	**Costo per MegaW.**	**Cap. max**
Generatore A	750	1000	3	2000
Generatore B	600	900	5	1700
Generatore C	800	1100	6	2500

Si vuole formulare un modello *PLI* che permetta di rappresentare il problema.

Variabili:

→ $x_{A_i}, x_{B_i}, x_{C_i}$, megawatt generati dai generatori *A*, *B*, *C* nel periodo *i*, *i*=1,2;

→ $\delta_{A_i} = \begin{cases} 1, & \text{se il generatore } A \text{ è attivato nel periodo } i, \\ 0, & \text{altrimenti;} \end{cases}$

→ $\delta_{B_i}, \delta_{C_i}$ analoghi, *i*=1,2.

Funzione obiettivo:

$$\min(3x_{A_1} + 3x_{A_2} + 5x_{B_1} + 5x_{B_2} + 6x_{C_1} + 6x_{C_2} + 750\delta_{A_1} + 1000\delta_{A_2} + \\ + 600\delta_{B_1} + 900\delta_{B_2} + 800\delta_{C_1} + 1100\delta_{C_2})$$

Si osservi che $3x_{A_1}$ rappresenta il costo per produrre x_{A_1} megawatt nel generatore *A* nel periodo 1 e $750\delta_{A_1}$ rappresenta il costo di attivazione *A* nel periodo 1, ….

Vincoli:

→ vincoli sulla domanda:

$$x_{A_1} + x_{B_1} + x_{C_1} \geq 4000,$$
$$x_{A_2} + x_{B_2} + x_{C_2} \geq 2800;$$

→ i costi di produzione ci sono se il generatore corrispondente è attivo:

$$x_{A_i} \leq 2000\delta_{A_i}, \ i = 1,2,$$
$$x_{B_i} \leq 1700\delta_{B_i}, \ i = 1,2,$$
$$x_{C_i} \leq 2500\delta_{C_i}, \ i = 1,2;$$

$$\rightarrow \quad x_{A_i}, x_{B_i}, x_{C_i} \geq 0,\ i=1,2; \quad \delta_{A_i}, \delta_{B_i}, \delta_{C_i} \in \{0,1\},\ i=1,2.$$

Plant location

Siano dati

m impianti che producono un tipo di prodotto,

n clienti,

d_j domanda del cliente *j*-esimo, *j*=1,…,*n*,

k_i costo fisso dell'*i*-esimo impianto, se è in funzione, *i*=1,…,*m*,

M_i massima capacità produttiva dell'impianto *i*-esimo, *i*=1,…,*m*,

c_{ij} costo per trasportare una unità di prodotto dall'impianto *i* al cliente *j*.

Si tratta di localizzare la produzione al minimo costo (costo di attivazione degli impianti e costo di trasporto). Siano

z_{ij} la quantità di materiale spedito da *i* a *j*, $z_{ij} \geq 0$,

$$y_i = \begin{cases} 1, & \text{se l'impianto } i \text{ è attivato,} \\ 0, & \text{se l'impianto } i \text{ non è attivato} \end{cases}$$

le variabili decisionali.

Il problema si formula come segue:

$$\min \left(\sum_{i=1}^{m} \sum_{j=1}^{n} c_{ij} z_{ij} + \sum_{i=1}^{m} k_i y_i \right),$$

$$\sum_{i=1}^{m} z_{ij} = d_j, \quad j \in \{1,...,n\},$$

$$\sum_{j=1}^{n} z_{ij} \leq M_i y_i, \quad i \in \{1,...,m\},$$

$$z_{ij} \geq 0, \quad y_i \in \{0,1\}, \quad i \in \{1,...,m\}, \ j \in \{1,...,n\}.$$

Il primo termine della funzione obiettivo rappresenta il costo di trasporto, mentre il secondo termine rappresenta il costo di attivazione. Il primo gruppo di vincoli rappresenta i vincoli di domanda e il secondo gruppo di vincoli rappresenta il vincolo che si può sfruttare l'impianto *i* solo se questo è attivato.

2. Considerazioni sull'arrotondamento della soluzione del problema PL associato

I problemi *PLI* hanno caratteristiche molto diverse dai problemi *PL*. A volte, dato un problema *PLI*, il problema *PL* ad esso associato (detto anche *problema rilassato*), cioè il problema che si ottiene da *PLI* rilassando i vincoli di interezza delle variabili, ha soluzione ottima a componenti intere. In questi casi la soluzione ottima del problema rilassato è anche soluzione ottima del problema *PLI*. In generale però il valore ottimo del problema rilassato costituisce solo un'approssimazione del valore ottimo del problema *PLI* (inferiore se se si stanno risolvendo problemi di minimo, superiore se si stanno risolvendo problemi di massimo).

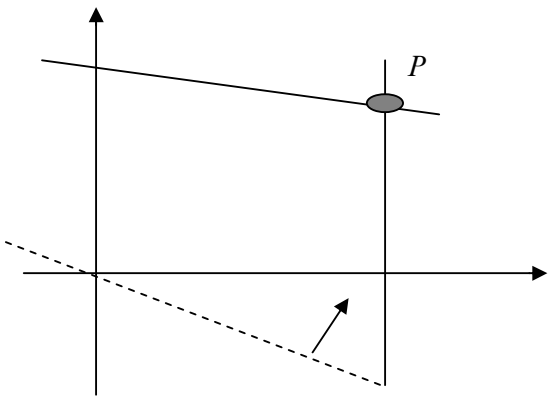
Si vuole far vedere, con dei controesempi, che la soluzione ottima di un problema *PLI* non coincide in generale con l'arrotondamento all'intero più vicino oppure con il troncamento della soluzione ottima del problema *PL* che si ottiene dal precedente rilassando i vincoli di interezza. Spesso infatti succede che il processo di approssimazione genera una soluzione non ammissibile per *PLI*. Tuttavia, se tale procedimento porta ad una soluzione ammissibile per *PLI*, si può pensare ad essa come ad una "buona" soluzione.

Esempio 1

Si consideri il problema *PLI*

$$\begin{aligned} &\max(x_1 + 5x_2), \\ &\quad x_1 + 10x_2 \le 20, \\ &\quad x_1 \le 2, \\ &\quad x_1, x_2 \ge 0, \\ &\quad x_1, x_2 \text{ interi.} \end{aligned}$$

Si risolva il problema *PL* che si ottiene dal precedente rilassando i vincoli di interezza.



La soluzione ottima di *PL* è il punto $P\left(2,\frac{9}{5}\right)$ avente seconda coordinata non intera. Arrotondando all'intero più vicino si ottiene il punto $(2,2)$ che non appartiene alla Regione Ammissibile di *PLI*. Troncando si ottiene il punto $(2,1)$ che appartiene alla Regione Ammissibile di *PLI* e in cui la funzione obiettivo vale 7. Esso non è comunque la soluzione ottima di *PLI*, infatti, calcolando il valore della funzione obiettivo in corrispondenza di tutte le soluzioni ammissibili, si trova:

$(0,0)$ z=0,

$(1,0)$ z=1,

$(2,0)$ z=2,

$(0,1)$ z=5,

$(0,2)$ z=10,

$(1,1)$ z=6,

$(2,1)$ z=7.

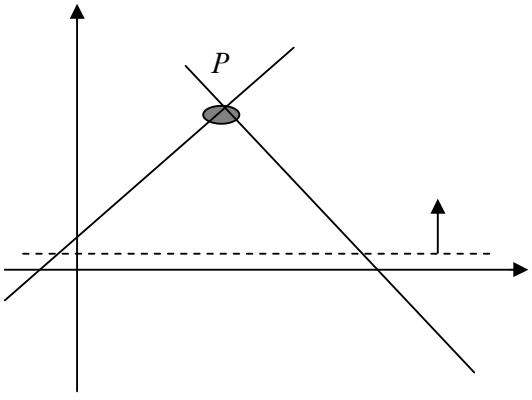
Da ciò si deduce che la soluzione ottima di *PLI* è $(0,2)$.

Esempio 2

Si consideri il problema *PLI*

$$\max(x_2),$$
$$-2x_1 + 2x_2 \le 7,$$
$$2x_1 + 2x_2 \le 33,$$
$$x_1, x_2 \ge 0,$$
$$x_1, x_2 \text{ interi.}$$

Si risolva il problema *PL* che si ottiene dal precedente rilassando i vincoli di interezza.



La soluzione ottima di *PL* è il punto $P\left(\frac{13}{2},10\right)$ avente prima coordinata non intera. Arrotondando all'intero più vicino si ottiene il punto $(7,10)$ che non appartiene alla Regione Ammissibile di *PLI*. Troncando si ottiene il punto $(6,10)$ che non appartiene alla Regione Ammissibile di *PLI*. Come si procede? Calcolare la funzione obiettivo in tutte le soluzioni ammissibili è troppo lungo! In seguito saranno presentate alcune metodologia risolutive per i problemi di programmazione lineare intera.

3. Totale unimodularità

Dato un problema di programmazione lineare intera, se la soluzione ottima del problema di programmazione lineare ad esso associato ha tutte le componenti intere, allora essa è anche soluzione ottima del problema di programmazione lineare intera originario. Si vedrà in seguito una condizione sufficiente affinchè la soluzione ottima di un problema di programmazione lineare abbia tutte le componenti intere. Questo succede in particolare se i vertici della regione ammissibile del problema *PL* hanno tutti coordinate intere.

Definizione Una matrice quadrata ad elementi interi *B* è detta *unimodulare (UM)* se $\det(B) = \pm 1$. Una matrice ad elementi interi *A* è detta *totalmente unimodulare (TUM)* se ogni sua sottomatrice quadrata non singolare è *UM*.

Si dimostra il seguente teorema:

Teorema Se *A* è una matrice $m \times n$ di rango *m* ad elementi interi *TUM* e $\underline{b}$ è un vettore di $\Re^m$ a componenti intere, allora tutti i vertici di

$$R_1(A) = \left\{ \underline{x} \in \Re^n \middle| A\underline{x} = \underline{b}, \underline{x} \geq \underline{0} \right\}$$

hanno coordinate intere.

Dim.:

Per ipotesi A è una matrice $m \times n$, $m<n$, a rango pieno. Ogni matrice quadrata B formata da m colonne di A linearmente indipendenti (ne esiste almeno una per ipotesi!) individua una soluzione di base:

$$\underline{x} = \left(\underline{x}_B, \underline{0} \right), \text{ con } \underline{x}_B = B^{-1}\underline{b} = \frac{B_{adj}\,\underline{b}}{\det(B)}.$$

Il vettore $\underline{x}_B$ ha componenti intere perchè il numeratore è un numero intero (essendo la matrice B_{adj} e il vettore $\underline{b}$ a componenti intere) e il denominatore è ± 1 (poichè A è *TUM*). Quindi tutte le soluzioni di base di un problema di programmazione lineare che ha $R_1(A)$ come regione ammissibile sono intere.■

Analogamente si dimostra il seguente teorema:

Teorema Se A è una matrice $m \times n$ di rango m ad elementi interi *TUM* e $\underline{b}$ è un vettore di $\Re^m$ a componenti intere, allora tutti i vertici di

$$R_2(A) = \left\{ \underline{x} \in \Re^n \middle| A\underline{x} \le \underline{b}, \underline{x} \ge \underline{0} \right\}$$

hanno coordinate intere.

Il seguente teorema fornisce delle condizioni sufficienti (attenzione: non anche necessarie) per la totale unimodularità di una matrice.

Teorema Una matrice ad elementi interi A con $a_{ij} \in \{0,+1,-1\}$ è *TUM* se in ogni colonna ci sono non più di due elementi non nulli e se le righe di A possono essere suddivise in due sottoinsiemi I_1 e I_2 tali che

a) se una colonna ha due elementi non nulli dello stesso segno, allora le due righe corrispondenti stanno in sottoinsiemi diversi;
b) se una colonna ha due elementi non nulli di segno diverso, allora le righe corrispondenti stanno nello stesso sottoinsieme.

Dim.:

Vogliamo dimostrare che A è *TUM*, cioè che ogni sua sottomatrice quadrata non singolare è *UM*, cioè ogni sua sottomatrice quadrata ha determinante che vale 0, +1 oppure –1. Sia C una sottomatrice quadrata di A. Si dimostra il risultato per induzione sulla dimensione della sottomatrice quadrata di A considerata, C.

Per *k=1* è verificata la tesi, infatti $C = a_{ij} \in \{0,+1,-1\}$.

Si supponga vera la tesi per *k-1* e la si dimostri per *k*.

Sia *C* sottomatrice quadrata di *A* di ordine *k*.

Tesi: $\det(C) \in \{0,+1,-1\}$.

- Se *C* ha una colonna con tutti 0, *C* è singolare e quindi $\det(C) = 0$.
- Se *C* ha una colonna con un unico elemento non nullo $\hat{c}$, si ha

  $$\det(C) = \hat{c}\det(C'),$$

  con *C′* sottomatrice quadrata di *C* e quindi di *A* di ordine *k-1*. Per ipotesi induttiva vale $\det(C') \in \{0,+1,-1\}$ e, poichè $\hat{c}$ è un elemento di *A*, vale $\hat{c} \in \{0,+1,-1\}$. Da ciò segue la tesi.
- Se *C* ha due elementi non nulli in ogni colonna, le condizioni a) e b) del teorema implicano che

  $$\sum_{i \in I_1} a_{ij} = \sum_{i \in I_2} a_{ij}, \text{ per ogni } j$$

  e quindi esiste una combinazione lineare delle righe nulla. Ciò significa che le righe di *C* sono linearmente dipendenti, da cui segue che $\det(C) = 0$.■

Corollario La matrice dei vincoli del problema di assegnazione è *TUM.*

Dim.:

Si applica il teorema precedente considerando $I_1 = \{\text{prime } n \text{ righe}\}$ e $I_2 = \{\text{ultime } n \text{ righe}\}$.■

Questo corollario è importante perchè assicura che la soluzione ottima del problema di programmazione lineare associato al problema di assegnazione è a coordinate intere. Ciò significa che una soluzione ottima del problema di assegnazione è una soluzione ottima del problema di programmazione lineare ad esso associato.

ESERCIZIO

Si dica, dimostrando l'affermazione, se le seguenti matrici sono totalmente unimodulari:

$$A = \begin{vmatrix} 0 & 1 & 0 & 0 & -1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & -1 & 0 & -1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & -1 & 1 \end{vmatrix}, \quad B = \begin{vmatrix} 1 & 1 & 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{vmatrix}$$

(Soluz.: per le matrici sono verificate le condizioni sufficienti del teorema, quindi esse sono totalmente unimodulari.)

4. Metodo di soluzione "cutting planes"

Si consideri il problema *PLI*

$$\begin{aligned} &\min \ \underline{c} \cdot \underline{x}, \\ &A\underline{x} = \underline{b}, \\ &\underline{x} \geq \underline{0}, \ x_i \text{ intero per ogni } i = 1,...,n. \end{aligned}$$

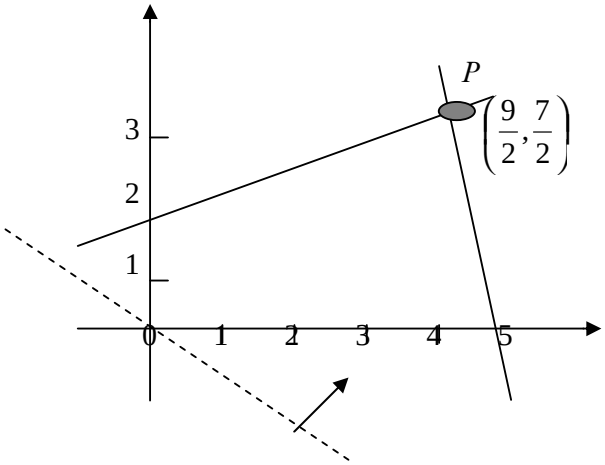
Sia $\underline{x}^*$ soluzione ottima del problema *PL* ad esso associato; in generale il vettore $\underline{x}^*$ non ha componenti intere. L'idea del metodo *cutting planes* prevede di aggiungere al problema *PL* un vincolo che non escluda alcun punto della regione ammissibile di *PL* a coordinate intere, escludendo però il punto di ottimo trovato, qualora esso non sia a coordinate intere. Si procede in questo modo fino a che il punto di ottimo risulta a coordinate intere. Poiché così facendo non si esclude alcun punto ammissibile a coordinate intere, la soluzione ottima che si trova (a coordinate intere!) è anche soluzione ottima per il problema *PLI* di partenza. Il vincolo che si aggiunge è detto *piano secante* in quanto esclude una porzione della regione ammissibile del problema *PL*.

Esempio

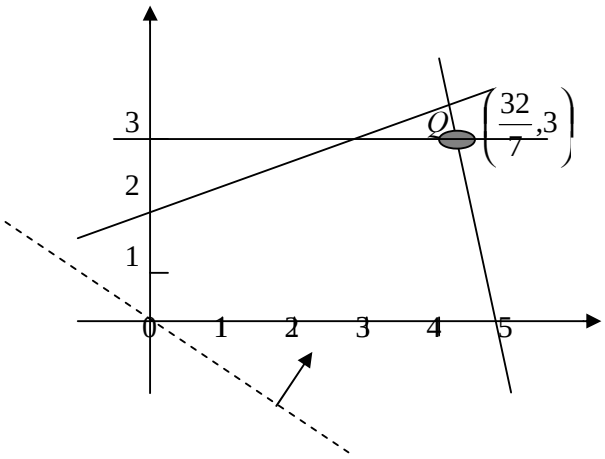
Si consideri il problema *PLI*

$$\begin{aligned} &\max(7x_1 + 9x_2), \\ &-x_1 + 3x_2 \leq 6, \\ &7x_1 + x_2 \leq 35, \\ &x_1, x_2 \geq 0, \\ &x_1, x_2 \text{ interi.} \end{aligned}$$

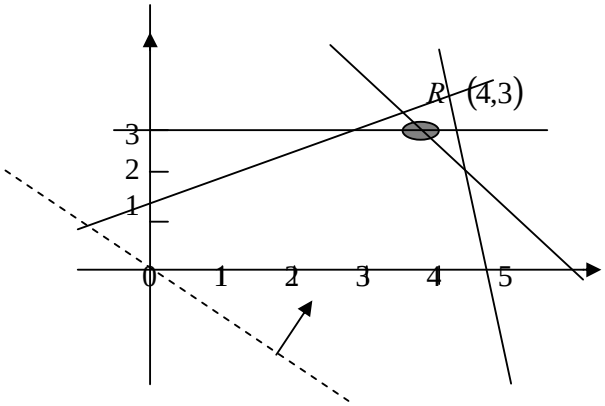
La soluzione ottima del problema *PL* ad esso associato è $P\left(\frac{9}{2},\frac{7}{2}\right)$ a coordinate non intere.



Aggiungendo il vincolo $x_2 \le 3$ non si esclude alcuna soluzione ammissibile di *PLI*:



La soluzione ottima del nuovo problema *PL* è $Q\left(\frac{32}{7},3\right)$, con ancora una componente non intera. Aggiungendo il vincolo $x_1 + x_2 \le 7$ non si esclude alcuna soluzione ammissibile a coordinate intere:



La soluzione ottima del nuovo problema *PL* è $R(4,3)$, a componenti intere. Il procedimento si arresta fornendo la soluzione ottima del problema *PLI* di partenza.

Taglio di Gomory

Def.: Dato $a \in \Re$, si dice *parte intera* di *a* ($\lfloor a \rfloor$) il massimo intero minore o uguale ad *a*

$$a = \lfloor a \rfloor + f, \quad 0 \le f < 1;$$

f è detta *parte frazionaria* di *a.*

Per esempio,

a=5.6, $\lfloor a \rfloor = 5$, *f*=0.6;

a=-5.6, $\lfloor a \rfloor = -6$, *f*=0.4;

a=5, $\lfloor a \rfloor$=5, *f*=0.

Si consideri un generico vincolo del problema *PLI*:

$$a_1 x_1 + a_2 x_2 + ... + a_n x_n = a_0, \quad x_j \ge 0 \text{, intero } \forall j .$$

Il vincolo può essere scritto:

$$(\lfloor a_1 \rfloor + f_1)x_1 + (\lfloor a_2 \rfloor + f_2)x_2 + ... + (\lfloor a_n \rfloor + f_n)x_n = (\lfloor a_0 \rfloor + f_0),\ x_j \ge 0 \text{, int., } 0 \le f_j < 1 \ \forall j ,$$

e di conseguenza anche, separando parti intere e parti frazionarie:

$$\lfloor a_1 \rfloor x_1 + \lfloor a_2 \rfloor x_2 + ... + \lfloor a_n \rfloor x_n - \lfloor a_0 \rfloor = f_0 - f_1 x_1 - ... - f_n x_n .$$

Il primo membro è intero e quindi, data l'uguaglianza, anche il secondo lo è. Inoltre, dato il segno positivo di x_j e f_j, vale anche

$$f_0 - f_1 x_1 - ... - f_n x_n \le f_0 .$$

Poiché $f_0 < 1$, allora

$$f_0 - f_1 x_1 - \ldots - f_n x_n \leq f_0 < 1$$

e quindi, data l'interezza di $f_0 - f_1 x_1 - \ldots - f_n x_n$, si ha

$$f_0 - f_1 x_1 - \ldots - f_n x_n \leq 0 .$$

Questo vincolo è detto *taglio di Gomory*.

Si può quindi schematizzare il procedimento nel seguente *algoritmo di Gomory*:

Passo 1: Risolvere il problema *PL* associato al problema *PLI*.

Passo 2: Se la soluzione ottima di *PL* è intera, allora *STOP*,
altrimenti andare al Passo 3.

Passo 3: Determinare l'equazione con termine noto a parte frazionaria maggiore nella tabella ottimale del simplesso (compreso l'obiettivo).

Passo 4: Calcolare il taglio di Gomory e includerlo nel problema da risolvere.

Passo 5: Risolvere il problema modificato e tornare al Passo 2.

Teorema: Se si aggiunge un taglio di Gomory ad una tabella ottimale di un problema *PL*, nessun punto ammissibile a coordinate intere viene escluso.

Esempio

Si consideri il seguente problema *PLI*:

$$\begin{aligned} &\min(-x_2), \\ &\quad -3x_1 + 2x_2 \leq 0, \\ &\quad 3x_1 + 2x_2 \leq 6, \\ &\quad x_1, x_2 \geq 0, \\ &\quad x_1, x_2 \text{ interi.} \end{aligned}$$

Consideriamo il problema *PL* ad esso associato e trasformiamolo in un problema con vincoli di uguaglianza:

$$\begin{aligned} &\min(-x_2), \\ &\quad -3x_1 + 2x_2 + x_3 = 0, \\ &\quad 3x_1 + 2x_2 + x_4 = 6, \\ &\quad x_1, x_2, x_3, x_4 \geq 0. \end{aligned}$$

Risolviamolo utilizzando il metodo del simplesso:

			*	*	
	x_1	x_2	x_3	x_4	
	-3	**2**	1	0	0
	3	2	0	1	6
	0	-1	0	0	0

		*		*	
	-3/2	1	½	0	0
	6	0	-1	1	6
	-3/2	0	½	0	0

	*	*			
	0	1	¼	¼	3/2
	1	0	-1/6	1/6	1
	0	0	¼	¼	3/2

Poiché nell'ultima riga non ci sono termini negativi ci si ferma:

$\left(1,\frac{3}{2}\right)$ è la soluzione ottima del problema *PL* associato.

Non essendo le sue componenti intere, essa non è la soluzione ottima del problema *PLI.* Applichiamo il taglio di Gomory generato dalla prima riga (avente termine noto con parte frazionaria maggiore o uguale a quella di tutti gli altri termini noti):

$$\frac{1}{2}-\frac{1}{4}x_3-\frac{1}{4}x_4 \le 0\,.$$

Volendo esprimere il taglio in termini di x_1 e x_2, sfruttando i vincoli originari espressi in forma di uguaglianza

$$x_3 = 3x_1 - 2x_2,$$
$$x_4 = 6 - 3x_1 - 2x_2,$$

il taglio di Gomory può anche essere scritto:

$$\frac{1}{4}(3x_1 - 2x_2)+\frac{1}{4}(6 - 3x_1 - 2x_2)\ge \frac{1}{2}$$

e cioè

$$x_2 \leq 1.$$

Dunque sia PL_1 il problema che si ottiene aggiungendo all'ultima tabella del simplesso il vincolo

$$\frac{1}{4}x_3 + \frac{1}{4}x_4 \geq \frac{1}{2},$$

che, scritto in forma di uguaglianza diventa

$$\frac{1}{4}x_3 + \frac{1}{4}x_4 - s_1 = \frac{1}{2}.$$

Si applica il metodo delle due fasi per trovare una soluzione iniziale ammissibile:

$$\frac{1}{4}x_3 + \frac{1}{4}x_4 - s_1 + y_1 = \frac{1}{2}.$$

	*	*				*	
	x_1	x_2	x_3	x_4	s_1	y_1	
	0	1	¼	1/4	0	0	3/2
	1	0	-1/6	1/6	0	0	1
	0	0	¼	**1/4**	-1	1	½
	0	0	0	0	0	1	0
	0	0	-1/4	-1/4	1	0	-1/2
	*	*		*			
	0	1	0	0	1	-1	1
	1	0	-1/3	0	2/3	-2/3	2/3
	0	0	1	1	-4	4	2
	0	0	0	0	0	1	0

Da ciò si ricava che $\left(\frac{2}{3},1,0,2,0\right)$ è soluzione ammissibile di base per PL_1.

	*	*		*		
	0	1	0	0	1	1
	1	0	-1/3	0	2/3	2/3
	0	0	1	1	-4	2
	0	0	¼	1/4	0	3/2
	0	0	0	0	1	1

Poiché nell'ultima riga non ci sono termini non negativi ci si ferma:

$\left(\frac{2}{3},1,0,2,0\right)$ è soluzione ottima per PL_1 (non unica).

Non essendo a coordinate intere, essa non è soluzione ottima per *PLI.*

Applichiamo il taglio di Gomory generato dalla seconda riga (avente termine noto con parte frazionaria maggiore o uguale a quella di tutti gli altri termini noti):

$$\frac{2}{3}-\frac{2}{3}x_3-\frac{2}{3}s_1 \le 0\,.$$

Volendo esprimere il taglio in termini di x_1 e x_2, si ottiene $x_1 - x_2 \ge 0$.

Dunque sia PL_2 il problema che si ottiene aggiungendo all'ultima tabella del simplesso il vincolo

$$\frac{2}{3}x_3+\frac{2}{3}s_1 \ge \frac{2}{3},$$

che, scritto in forma di uguaglianza diventa

$$\frac{2}{3}x_3+\frac{2}{3}s_1-s_2=\frac{2}{3}.$$

Si applica il metodo delle due fasi per trovare una soluzione iniziale ammissibile:

$$\frac{2}{3}x_3+\frac{2}{3}s_1-s_2+y_2=\frac{2}{3}.$$

	*	*		*			*	
	x_1	x_2	x_3	x_4	s_1	s_2	y_2	
	0	1	0	0	**1**	0	0	1
	1	0	-1/3	0	2/3	0	0	2/3
	0	0	1	1	-4	0	0	2
	0	0	2/3	0	2/3	-1	1	2/3
	0	0	0	0	0	0	1	0
	0	0	-2/3	0	-2/3	1	0	-2/3

	*			*	*		*	
	0	1	0	0	1	0	0	1
	1	-2/3	-1/3	0	0	0	0	0
	0	4	1	1	0	0	0	6
	0	-2/3	**2/3**	0	0	-1	1	0
	0	2/3	-2/3	0	0	1	0	0

	*		*	*	*			
	0	1	0	0	1	0	0	1
	1	-1	0	0	0	-1/2	1/2	0
	0	5	0	1	0	3/2	-3/2	6
	0	-1	1	0	0	-3/2	3/2	0
	0	0	0	0	0	0	1	0

Da ciò si ricava che $(0,0,0,6,1,0)$ è soluzione ammissibile per PL_2.

	*		*	*	*		
	0	**1**	0	0	1	0	1
	1	-1	0	0	0	-1/2	0
	0	5	0	1	0	3/2	6
	0	-1	1	0	0	-3/2	0
	0	0	0	0	1	0	1
	0	-1	0	0	0	0	0

	*	*	*	*		
	0	1	0	0	1	0	1
	1	0	0	0	1	-1/2	1
	0	0	0	1	-5	3/2	1
	0	0	1	0	1	-3/2	1
	0	0	0	0	1	0	1

Poiché nell'ultima riga non ci sono termini non negativi ci si ferma:

$(1,1,1,1,0,0)$ è soluzione ottima per PL_2 (non unica).

Poiché essa ha coordinate intere, $(1,1)$ è soluzione ottima anche per il problema *PLI* di partenza.

ESERCIZIO: si verifichi graficamente che la soluzione ottima del problema *PLI* è $(1,1)$.

Teorema: L'algoritmo di Gomory termina dopo un numero finito di passi e fornisce una soluzione ottima di *PLI* oppure trova che *PLI* non ha soluzione ottima.

5. Metodo di soluzione "Branch and Bound"

Si consideri il problema *PLI*

$$\min\{z(x) = \underline{c} \cdot \underline{x} : \underline{x} \in S\},$$

con $S = Q \cap Z^n$ finito e Q un poliedro di $\Re^n$. Il metodo *Branch and Bound* consiste di due fasi: una fase di *Branching*, che letteralmente significa ramificazione, in cui si generano sottoproblemi a partire da un problema assegnato, e una fase *Bounding*, in cui si cercano delle limitazioni (in questo caso superiori) per il valore della funzione obiettivo e

si eliminano casi non significativi (si ricordi sempre che, se si sta risolvendo un problema di minimo, "restringendo" la regione ammissibile si può solo aumentare il valore della funzione obiettivo).

Sia $\underline{\hat{x}} \in Q$ *soluzione ottima del problema rilassato* $\min\{z(x) = \underline{c} \cdot \underline{x} : \underline{x} \in Q\}$. *Se le sue componenti sono intere allora è anche soluzione ottima del problema* PLI. *Si supponga che* $\underline{\hat{x}}$ *abbia componenti non intere e sia* $\underline{x}^*$ *la soluzione ottima del problema* PLI *(da determinare). Ovviamente* $z(\underline{\hat{x}})$ *rappresenta una limitazione inferiore (lower bound) al valore ottimo intero, cioè* $z(\underline{\hat{x}}) \le z(\underline{x}^*)$. *La strategia di Branching prevede di partizionare la regione ammissibile del problema* PLI, *cioè l'insieme* S, *in una famiglia di sottoinsiemi* $S_1,...,S_r$ *tale che*

$$S_i \cap S_j = \varnothing,\ i, j = 1,...,n,\ i \neq j,\ \text{e}\ \bigcup_{i=1}^{r} S_i = S;$$

nel costruire la partizione si deve fare in modo di eliminare una parte della regione ammissibile del problema *PL* associato, cioè dell'insieme *Q*, tra cui anche la soluzione $\underline{\hat{x}}$. Si cerca sostanzialmente di eliminare da Q la soluzione $\underline{\hat{x}}$ senza escludere nessuna soluzione di *S*. Si ottiene così un insieme $Q' \subseteq Q$ tale che $S = Q \cap Z^n = Q' \cap Z^n$ e $\underline{\hat{x}} \notin Q'$. Si considerino i problemi con stessa funzione obiettivo del problema di partenza e regione ammissibile S_i:

$$(P_i) \qquad \min\{z(x) = \underline{c} \cdot \underline{x} : \underline{x} \in S_i\}$$

Sia $\underline{x}^i$ soluzione ottima del problema *PL* associato a P_i. Ovviamente $z(\underline{x}^i)$ rappresenta una limitazione inferiore per il valore ottimo di P_i. Sia $\tilde{z} = z(\underline{\tilde{x}})$ un valore *ottimo corrente* per *PLI* di partenza (*upper bound*); cioè il valore della funzione obiettivo calcolato in una soluzione ammissibile di *PLI*; se non si conoscono soluzioni ammissibili per *PLI* si pone $\tilde{z} = +\infty$. Se $z(\underline{x}^i) \ge \tilde{z}$, cioè se il valore ottimo del problema *PL* associato a P_i è non minore dell'ottimo corrente, allora il problema P_i può essere abbandonato perché non può produrre un miglioramento della funzione obiettivo. Se invece $z(\underline{x}^i) < \tilde{z}$, allora il problema P_i è un candidato per individuare la soluzione ottima e la sua regione ammissibile verrà "eventualmente" ripartita ripetendo la fase di Branching.

I criteri di arresto sono i seguenti:

- Se un problema *PL* ha soluzione ottima intera ci si ferma, il ramo viene dichiarato secco (nel senso che non si riparte più da quel problema) e il valore ottimo è un upper bound per il valore ottimo della funzione obiettivo del problema *PLI* di partenza.
- Se un problema *PL* ha regione ammissibile vuota il ramo è secco.
- Se un problema *PL* ha soluzione ottima non intera e il valore ottimo è non inferiore all'upper bound allora il ramo è secco (procedendo da là non si riesce sicuramente a fare meglio).

Ci si ferma quando tutti i rami sono secchi: il valore ottimo di *PLI* è l'ultimo upper bound determinato.

Nel riportare una descrizione schematica della strategia risolutiva, si considera la versione dell'algoritmo che prevede che l'operazione di branching partizioni in due soli sottoinsiemi la regione ammissibile a cui si applica il Branching. Se risulta che una componente della soluzione ottima di un problema è non intera, $x_i = \alpha$, allora si partiziona la regione ammissibile di tale problema nei due sottoinsiemi che si ottengono da esso aggiungendo rispettivamente i vincoli

$$x_i < \alpha \quad \text{e} \quad x_i > \alpha .$$

Poiché interessano solo le souzioni a componenti intere, i due vincoli precedenti sono equivalenti rispettivamente a

$$x_i \le \lfloor \alpha \rfloor \quad \text{e} \quad x_i \ge \lfloor \alpha \rfloor + 1.$$

Algoritmo:

Passo 1: Considerare il problema *PL* associato al problema *PLI* di partenza, P_0. Porre $\tilde{z} = +\infty$.

Passo 2: Branching:

se tutti i rami sono secchi, andare al Passo 7;

se ci sono rami non secchi, scegliere un vertice P_j (un problema *PLI*) e considerare il problema *PL* ad esso associato: se è stato risolto andare al Passo 3, altrimenti andare al Passo 4.

Passo 3: detta $\underline{x}^j$ una soluzione ottima di P_j, scegliere una sua componente x_i^j con parte frazionaria positiva e fare Branching ponendo:

$$x_i \le \lfloor x_i^j \rfloor \quad \text{e} \quad x_i \ge \lfloor x_i^j \rfloor + 1 .$$

Andare al Passo 2.

Passo 4: Risolvere il problema *PL* associato a P_j.

Se non esistono soluzioni ammissibili, dichiarare secco il ramo P_j e andare al Passo 2.

Se esiste una soluzione ottima, sia $\tilde{z}^j$ il valore ottimo della funzione obiettivo di P_j.

Andare al Passo 5.

Passo 5: Se la soluzione ottima di P_j è non intera andare al Passo 6.

Se la soluzione ottima di P_j è intera, dichiarare secco il ramo P_j e porre $\tilde{z} = \min\{\tilde{z}, \tilde{z}^j\}$. Andare al Passo 2.

Passo 6: Dichiarare secchi tutti i rami P_j tali che $\tilde{z}^j \geq \tilde{z}$. Andare al Passo 2.

Passo 7: Terminazione:

se $\tilde{z} = +\infty$, non esiste soluzione ammissibile;

se $\tilde{z} < +\infty$, la soluzione che ha $\tilde{z}$ come valore ottimo della funzione obiettivo è soluzione ottima.

Esempio

Si consideri il problema *PLI*

$$\begin{aligned} &\min(-7x_1 - 9x_2), \\ &-x_1 + 3x_2 \leq 6, \\ &7x_1 + x_2 \leq 35, \\ &x_1, x_2 \geq 0, \text{ interi.} \end{aligned}$$

Si consideri il problema *PL* ad esso associato e sia esso P_0.

Il problema P_0 ha soluzione ottima $x_1^0 = \frac{9}{2}, \quad x_2^0 = \frac{7}{2}$ e valore ottimo $z^0 = -63$.

Ovviamente $\left(\frac{9}{2}, \frac{7}{2}\right)$ non è soluzione ammissibile per *PLI*. Sia $\tilde{z} = +\infty$ un *upper bound* per il problema *PLI*.

Si considerano allora i problemi P_1 e P_2 così costruiti:

P_1: P_0

con vincolo $x_1 < \frac{9}{2}$;

P_2: P_0

con vincolo $x_1 > \frac{9}{2}$.

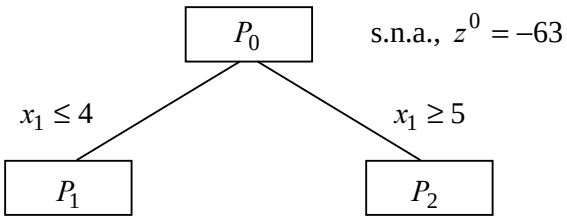
Poiché si stanno cercando soluzioni a componenti intere, non si esclude alcuna soluzione ammissibile per *PLI* se si considerano:

P_1: P_0

con vincolo $x_1 \leq 4$;

P_2: P_0

con vincolo $x_1 \geq 5$.



Il problema P_1 ha soluzione ottima $x_1^1 = 4, \quad x_2^1 = \frac{10}{3}$ e valore ottimo $z^1 = -58$.

Il problema P_2 ha soluzione ottima $x_1^2 = 5, \quad x_2^2 = 0$ e valore ottimo $z^2 = -35$.

<u>Osservazione</u>: scendendo verso il basso, cioè "ramificando", il valore della funzione obiettivo aumenta (essendo il problema un problema di minimo).

La soluzione ottima di P_2, $(5, 0)$, è soluzione ammissibile per *PLI*. Poiché $z^2 = -35 < \tilde{z}$, allora $\tilde{z} := z^2 = -35$ rappresenta il nuovo *upper bound* per il valore ottimo della funzione obiettivo di *PLI*.

Poiché $z^1 = -58 < \tilde{z} = -35$ bisogna continuare il Branching a partire da P_1

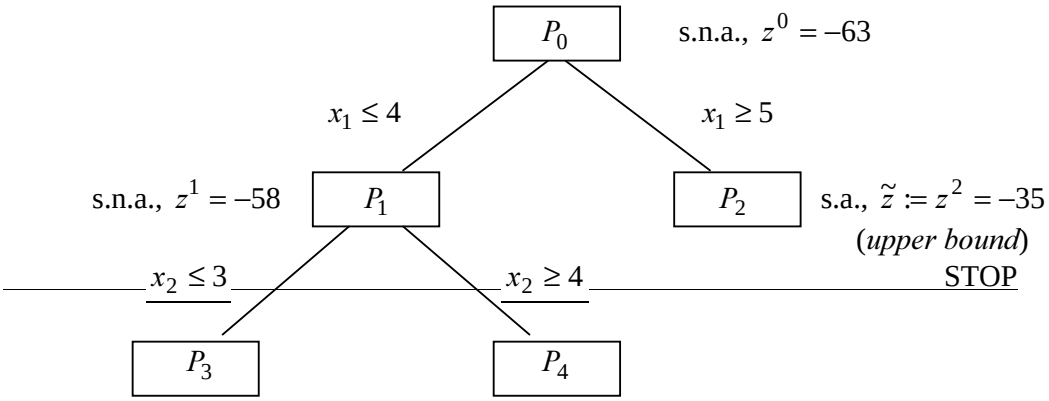
Si considerano allora i problemi P_3 e P_4 così costruiti:

P_3: P_1

con vincolo $x_2 \leq 3$;

P_4: P_1

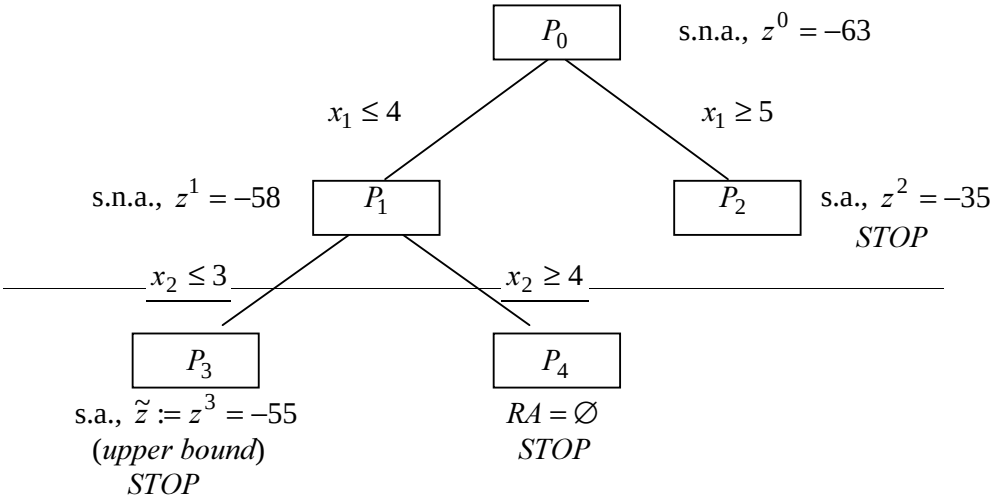
con vincolo $x_2 \geq 4$.



Il problema P_3 ha soluzione ottima $x_1^3 = 4, \quad x_2^3 = 3$ e valore ottimo $z^3 = -55$.

Il problema P_4 ha regione ammissibile vuota.

La soluzione ottima di P_3, $(4,3)$, è soluzione ammissibile per *PLI*. Il ramo P_3 è secco e poiché $z^3 = -55 < \tilde{z}$, allora $\tilde{z} := z^3 = -55$ rappresenta il nuovo *upper bound* per il valore ottimo della funzione obiettivo di *PLI*.



Ci si ferma perché tutti i rami sono secchi. L'*upper bound* $\tilde{z} = -55$ rappresenta il valore ottimo della funzione obiettivo di *PLI* e quindi $(4,3)$ è la soluzione ottima di *PLI*.

Esempio

Si consideri il problema *PLI*

$$\begin{aligned} &\min(2x+3y),\\ &\quad x+3y \geq 5,\\ &\quad 2x+y \geq 6,\\ &\quad x, y \geq 0, \text{ interi.} \end{aligned}$$

Si consideri il problema *PL* ad esso associato e sia esso P_0.

Il problema P_0 ha soluzione ottima $x^0 = 2.6, \quad y^0 = 0.8$ e valore ottimo $z^0 = 7.6$.

Ovviamente $(2.6,0.8)$ non è soluzione ammissibile per *PLI*. Sia $\tilde{z} = +\infty$ un *upper bound* per il problema *PLI*.

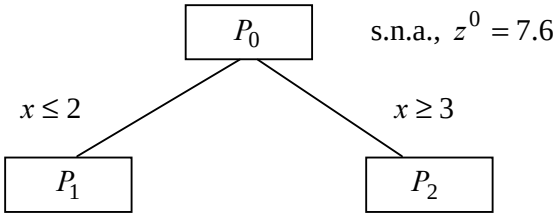
Si considerano allora i problemi P_1 e P_2 così costruiti:

P_1: P_0

con vincolo $x \leq 2$;

P_2: P_0

con vincolo $x \geq 3$.



Il problema P_1 ha soluzione ottima $x^1 = 2, \quad y^1 = 2$ e valore ottimo $z^1 = 10$.

Il problema P_2 ha soluzione ottima $x^2 = 3, \quad y^2 = \frac{2}{3}$ e valore ottimo $z^2 = 8$.

La soluzione ottima di P_1, $(2,2)$, è soluzione ammissibile per *PLI*. Poiché $z^1 = 10 < \tilde{z}$, allora $\tilde{z} := z^1 = 10$ rappresenta il nuovo *upper bound* per il valore ottimo della funzione obiettivo di *PLI*.

Poiché $z^2 = 8 < \tilde{z} = 10$ bisogna continuare il Branching a partire da P_2

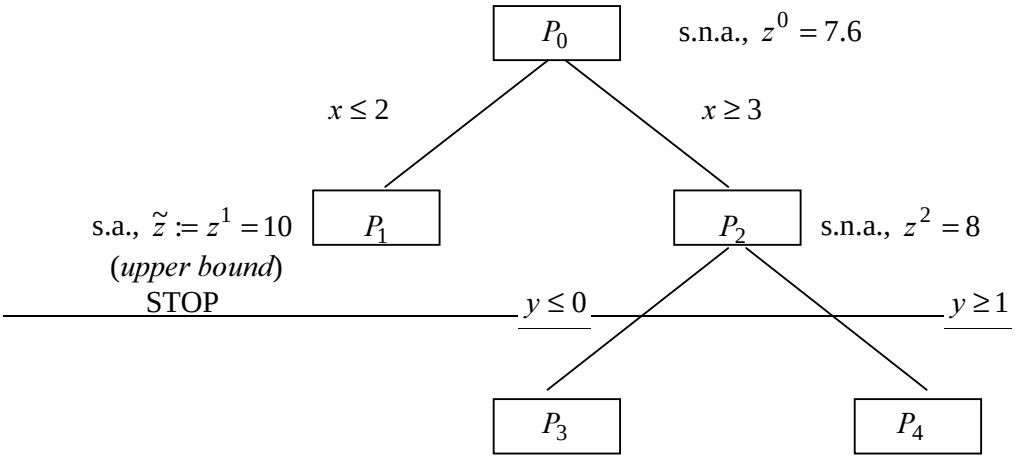
Si considerano allora i problemi P_3 e P_4 così costruiti:

P_3: P_2

con vincolo $y \leq 0$;

P_4: P_2

con vincolo $y \geq 1$.

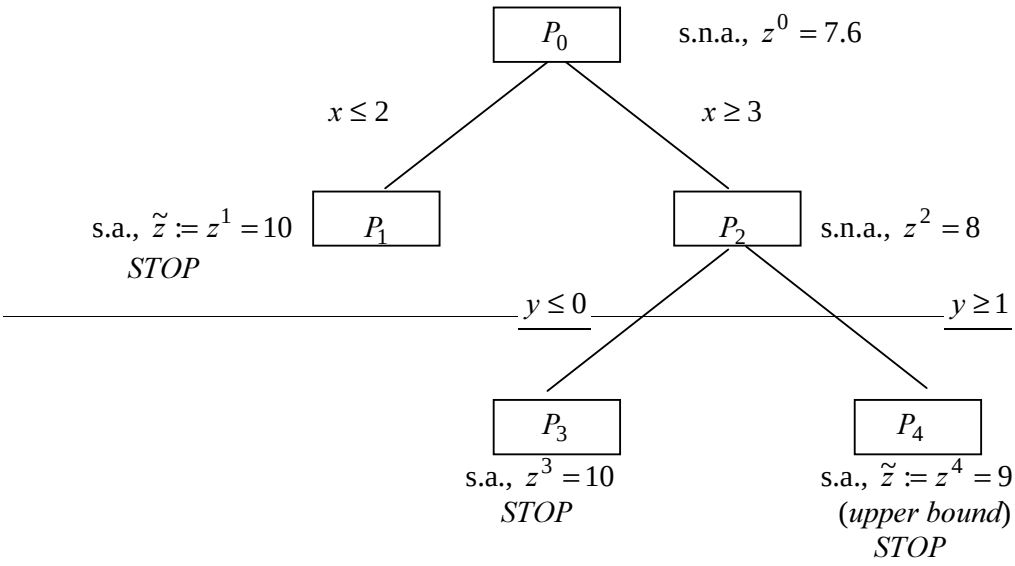


Il problema P_3 ha soluzione ottima $x^3 = 5, \quad y^3 = 0$ e valore ottimo $z^3 = 10$.

Il problema P_4 ha soluzione ottima $x^4 = 3, \quad y^4 = 1$ e valore ottimo $z^4 = 9$.

La soluzione ottima di P_3, $(5, 0)$, è soluzione ammissibile per *PLI*. Il ramo P_3 è secco e poiché $z^3 = 10 = \tilde{z}$, allora l'*upper bound* resta il precedente.

La soluzione ottima di P_4, $(3,1)$, è soluzione ammissibile per *PLI*. Il ramo P_4 è secco e poiché $z^4 = 9 < \tilde{z}$, allora $\tilde{z} := z^4 = 9$ rappresenta il nuovo *upper bound* per il valore ottimo della funzione obiettivo di *PLI*.



Poiché tutti i rami sono secchi ci si ferma. L'*upper bound* $\tilde{z} = 9$ rappresenta il valore ottimo della funzione obiettivo di *PLI* e quindi $(3,1)$ è la soluzione ottima di *PLI*.

6. Rilassamenti

Sia Π una classe di problemi contenente istanze del tipo π=(F,c,min). Una coppia $[\Pi', \sigma]$, con

i) Π' una classe di problemi del tipo $\pi' = (F', c', \min)$,

ii) $\sigma : \Pi \to \Pi'$ funzione che fa corrispondere ad ogni problema $\pi \in \Pi$ un problema $\pi' \in \Pi'$, $\pi' = \sigma(\pi) = (F', c', \min)$,

è detta *schema di rilassamento* per Π se per ogni $\pi \in \Pi$ e $(F', c', \min) = \sigma(\pi)$ si ha:

$$F \subseteq F', \qquad x \in F \Rightarrow c'(x) \le c(x).$$

Dati $\pi \in \Pi$ e lo schema di rilassamento $[\Pi', \sigma]$, si dice che $\pi' = \sigma(\pi)$ è un *rilassamento* di π. Sia $\pi' = (F', c', \min)$ un rilassamento di π=(F,c,min); allora valgono i seguenti risultati:

Teorema $z(\pi) = \min\{c(x) : x \in F\} \ge z(\pi') = \min\{c'(x) : x \in F'\}$.

Teorema Sia $x^* \in F'$ una soluzione ottima per π', cioè $z(\pi') = c'(x^*)$, e sia inoltre $x^* \in F$ e $c(x^*) = c'(x^*)$. Allora x^* è anche soluzione ottima per π.

Nel seguito saranno descritte in breve due tecniche di rilassamento.

Eliminazione di vincoli

Si consideri il problema

$$(P) \qquad \min\{f(x) : x \in X_1 \cap X_2\}.$$

Un suo rilassamento è il problema

$$(PR) \qquad \min\{f(x) : x \in X_1\}$$

che si ottiene da P eliminando il vincolo che x debba appartenere a X_2. Un rilassamento di questo tipo ha senso quando i vincoli che definiscono l'insieme $X_1 \cap X_2$ sono di difficile trattazione, mentre sono trattabili quelli che definiscono l'insieme X_1. Evidentemente se la soluzione ottima di PR appartiene anche all'insieme X_2, allora essa è anche soluzione ottima di P.

Rilassamento continuo

Si consideri il problema

$$(P) \qquad \min\{f(x) : x \in X,\ x \in \{0,1\}^n\},$$

con $X \subseteq \Re^n$ insieme convesso. Si dice *rilassamento continuo* di (P) il seguente problema:

$$(Pc) \qquad \min\{f(x) : x \in X,\ x \in [0,1]^n\}.$$

Capitolo 3: Problemi di scheduling deterministico

I problemi di scheduling sono problemi che studiano l'assegnamento nel tempo di lavori a macchine, in modo da rispettare dei vincoli strutturali e temporali ed ottimizzare una prefissata funzione obiettivo. A seconda del numero e delle caratteristiche delle macchine, delle caratteristiche dei lavori da eseguire (*job*) e della funzione obiettivo, si hanno diversi problemi di scheduling.

Qui si suppone che una macchina non possa eseguire più di un job contemporaneamente e che un job non possa essere lavorato su più macchine contemporaneamente. Si suppone inoltre che il problema sia deterministico, cioè che i dati del problema siano noti in anticipo e con certezza, e che le macchine non siano soggette a guasti. Evidentemente la modellizzazione che si sta cercando è molto semplificata rispetto alla realtà, in cui, per esempio, le macchine sono soggette a guasti e non è detto che la domanda dei clienti sia nota in anticipo.

Si definisce *schedule ammissibile* un qualunque sequenziamento dei job che rispetti i vincoli del problema; si definisce invece *schedule ottimo* uno schedule ammissibile che ottimizzi la funzione obiettivo prescelta.

Siano

$M = \{M_i, i = 1,2,..., m\}$ l'insieme delle macchine a disposizione,

$J = \{J_j, j = 1,2,..., n\}$ l'insieme dei job che devono essere lavorati.

Per ogni job J_j vanno specificati i seguenti dati:

- il *numero* m_j *di operazioni* che compongono il job (può succedere che un job debba essere lavorato su più macchine per essere completato, quindi è ragionevole pensarlo scomposto in più operazioni, ognuna delle quali va eseguita su una macchina);
- il *tempo di lavorazione* (*processing time*) p_j del job J_j, nel caso in cui il job sia composto da una singola operazione, oppure i *tempi di lavorazione* p_{ij} dell'operazione *i*-esima del job J_j, nel caso in cui il job sia composto da operazioni multiple;
- la *data di disponibilità* r_j (*release time*) del job J_j, cioè l'istante in cui il job è disponibile per essere lavorato; non è detto che l'istante in cui il job è pronto per essere lavorato coincida con l'istante in cui inizia la sua lavorazione;

- una *funzione costo* non decrescente, f_j, che valuta il costo $f_j(t)$ dovuto al completamento della lavorazione del job J_j all'istante *t*;
- la *due date*, d_j, istante di tempo entro il quale la lavorazione del job J_j deve essere completata (o per lo meno "dovrebbe", nel senso che un job può eventualmente essere consegnato in ritardo), e un peso, w_j, che possono essere utilizzati nel definire la funzione costo.

Un problema di scheduling è completamente formulato se sono specificate le caratteristiche dei job, le caratteristiche delle macchine e la funzione obiettivo. Tali informazioni sono raccolte in una lista a tre campi $\alpha/\beta/\gamma$.

1. L'ambiente delle macchine

L'ambiente delle macchine è descritto da $\alpha = \alpha_1, \alpha_2$.

$$\alpha_1 \in \{o, P, Q, R, O, F, J\}$$

♦ Se $\alpha_1 \in \{o, P, Q, R\}$, allora ogni job J_j consiste di un'unica operazione che può essere eseguita da ogni macchina M_i; il tempo di lavorazione di J_j su M_i è p_{ij}.

- Se $\alpha_1 = o$, si ha macchina singola; $p_{1j} = p_j$;
- se $\alpha_1 = P$, si hanno macchine parallele identiche, cioè macchine che hanno la stessa velocità; il tempo di lavorazione dei job non dipende dalla particolare macchina, $p_{ij} = p_j$, per ogni $i = 1,...,m$;
- se $\alpha_1 = Q$, si hanno macchine parallele uniformi, cioè macchine che hanno velocità diverse indipendenti dal job in lavorazione; $p_{ij} = \frac{p_j}{s_i}$, con s_i velocità di M_i, s_i data. Un esempio è fornito dalle macchine da cucire: una macchina da cucire vecchia è generalmente più lenta di una macchina da cucire nuova, qualunque sia il lavoro da eseguire;
- se $\alpha_1 = R$, si hanno macchine parallele indipendenti, cioè macchine aventi velocità dipendenti dal job in lavorazione; $p_{ij} = \frac{p_j}{s_{ij}}$, con s_{ij} velocità di esecuzione del job J_j su M_i, s_j data. Un esempio è fornito dalle sarte in un

laboratorio: alcune sarte sono più brave a fare certe cose, altre sono più brave a farne altre.

- Se $\alpha_1 = O$, si ha un *open shop*: il job J_j consiste di un insieme di *m* operazioni $\{O_{1j},...,O_{mj}\}$. L'operazione O_{ij} deve essere eseguita sulla macchina M_i e ha tempo di lavorazione p_{ij}. L'ordine di esecuzione delle operazioni sulle macchine può essere qualsiasi.
- Se $\alpha_1 \in \{F, J\}$ è imposto un ordine di esecuzione delle operazioni corrispondenti ad ogni job sulle macchine.
  - Se $\alpha_1 = F$ si ha un *flow shop*, in cui ogni job J_j consiste delle operazioni $\{O_{1j},...,O_{mj}\}$. L'operazione O_{ij} deve essere eseguita sulla macchina M_i e ha tempo di lavorazione p_{ij};
  - se $\alpha_1 = J$ si ha un *job shop*, in cui ogni job consiste delle operazioni $\{O_{1j},...,O_{m_j j}\}$. L'operazione O_{ij} deve essere eseguita su una data macchina μ_{ij} e ha tempo di lavorazione p_{ij}. Si ha $\mu_{ij} \neq \mu_{i+1\,j}$, per ogni $i = 1,..., m_j - 1$.

Si osservi che $\alpha_1 \in \{O, F, J\}$ significa che il generico job deve essere lavorato su più macchine. Se $\alpha_1 = O$ non è fissato alcun ordine di esecuzione delle operazioni sulle macchine, mentre nel caso $\alpha_1 = F$ è fissato un ordine di esecuzione delle operazioni sulle macchine e questo è lo stesso per ogni job. Nel caso $\alpha_1 = J$ è fissato un ordine di esecuzione delle operazioni sulle macchine che può essere diverso da job a job.

$$\alpha_2 \in \mathrm{N} \cup \{\mathrm{o}\}$$

- Se $\alpha_2 = m \in \mathrm{N}$, allora si ha un problema con un numero costante di macchine, *m*.
- Se $\alpha_2 = o$, allora il numero di macchine è variabile.

Si osservi che $\alpha_1 = o \Leftrightarrow \alpha_2 = 1$.

Se si scrive $\alpha_1 = P$ e $\alpha_2 = 2$ significa che si sta affrontando un problema con 2 macchine parallele identiche.

2. Le caratteristiche dei job

Le caratteristiche dei job sono specificate da $\beta = \beta_1, \beta_2, \beta_3, \beta_4$.

- ♦ $\beta_1 \in \{pmtn, o\}$;
  - se $\beta_1 = pmtn$, la *preemtion* dei job è permessa, nel senso che la lavorazione delle operazioni può essere interrotta e ripresa successivamente;
  - se $\beta_1 = o$, la preemtion dei job non è permessa.
- ♦ $\beta_2 \in \{prec, tree, o\}$;
  - se $\beta_2 = prec$, è specificata una relazione di precedenza, $\rightarrow$, tra i job: $J_j \rightarrow J_k$, significa che J_j deve essere completato prima che J_k possa iniziare. Le precedenze tra job vengono di solito schematizzate mediante grafi i cui nodi rappresentano i job e i cui archi orientati rappresentano le precedenze;
  - se $\beta_2 = tree$, è specificata una relazione di precedenza ad albero tra i vari job, cioè il grafo che rappresenta le precedenze è un albero (grafo connesso e privo di cicli). Per descrivere le precedenze ad albero si utilizzano di solito alberi caratterizzati dal fatto che ogni nodo ha un solo arco entrante oppure un solo arco uscente;
  - se $\beta_2 = o$, non è specificata alcuna relazione di precedenza.
- ♦ $\beta_3 \in \{r_j, o\}$;
  - se $\beta_3 = r_j$, sono specificate le date di disponibilità, r_j, per ogni job J_j;
  - se $\beta_3 = o$, $r_j = 0$ per ogni *j*.
- ♦ $\beta_4 \in \{p_j = 1, p_{ij} = 1, o\}$;
  - se $\beta_4 = (p_j = 1)$, ogni job ha tempo di esecuzione unitario (si può verificare solo se $\alpha_1 \in \{o, P, Q, R\}$);
  - se $\beta_4 = (p_{ij} = 1)$, ogni operazione ha tempo di esecuzione unitario (si può verificare solo se $\alpha_1 \in \{O, F, J\}$);
  - se $\beta_4 = o$, i p_j o i p_{ij} sono tutti numeri interi non negativi arbitrari.

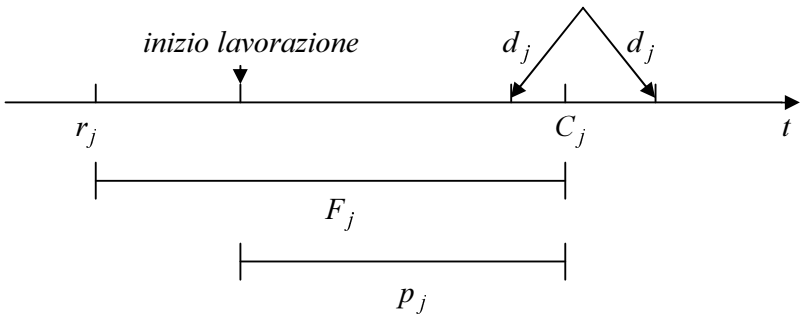
Per esempio il problema $P, o / pmtn, tree, r_j, p_j = 1/*$ è un problema a macchine parallele (non è specificato il numero delle macchine), in cui è ammessa la preemption, ci sono precedenze ad albero tra i job, sono specificate le date di disponibilità dei vari job e tutti i job hanno tempo di esecuzione unitario. Il terzo campo deve ancora essere specificato.

3. Criterio di ottimalità

Il criterio di ottimalità è specificato dal terzo campo, $\gamma \in \{f_{\max}, \overline{f}\}$. Specificheremo in seguito il significato di $f_{\max}$ e di $\overline{f}$.

Per ogni job J_j, siano

- C_j l'istante di completamento della lavorazione del job J_j.
- F_j il *tempo di flusso* del job J_j, cioè il tempo che trascorre tra l'istante in cui il job J_j è disponibile per essere lavorato e l'istante di completamento della sua lavorazione, $F_j = C_j - r_j$. Si osservi che il tempo di flusso coincide con il tempo di completamento se e solo se la data di disponibilità del job coincide con l'istante di inizio della sua lavorazione.
- $L_j = C_j - d_j$ la *lateness* del job J_j, cioè la differenza tra l'istante di completamento e la due date. La *lateness* può essere sia positiva, che negativa, che nulla: positiva se il job viene terminato in ritardo, negativa se il job viene terminato in anticipo, nulla se il job viene terminato esattamente alla *due date*. Quindi la *lateness* tiene conto non solo dell'eventuale ritardo, ma anche dell'eventuale anticipo nella consegna.
- $T_j = \max\{0, L_j\}$ la *tardiness* del job J_j. La *tardiness* misura l'effettivo ritardo nella consegna, nel senso che essa è positiva se e solo se $\max\{0, L_j\} > 0$, cioè $C_j - d_j > 0$; se il job viene terminato in tempo la *tardiness* è nulla.
- $E_j = \max\{0, -L_j\}$ la *earliness* del job J_j. La *earliness* tiene conto solo dell'eventuale anticipo nella consegna.
- I_i il tempo di inattività della macchina M_i, $I_i = C_{\max} - \sum_{j=1}^{n} p_{ij}$, dove $C_{\max} = \max_{1 \le j \le n} C_j$ è l'istante di completamento dell'ultimo job lavorato. Il tempo di inattività di una macchina viene calcolato come la differenza tra l'istante di completamento dell'ultimo job lavorato e il tempo totale in cui effettivamente la macchina risulta impiegata.



Siano poi

- $N_W(t)$ il numero di job in attesa di lavorazione tra le macchine o non pronti per la lavorazione all'istante *t*;
- $N_p(t)$ il numero di job che sono in lavorazione all'istante *t*;
- $N_c(t)$ il numero di job la cui lavorazione è stata completata all'istante *t*;
- $N_u(t)$ il numero di job la cui lavorazione deve ancora essere completata all'istante *t*.

Tra tali variabili valgono le seguenti relazioni:

$$N_W(t) + N_p(t) + N_c(t) = n, \text{ per ogni } t,$$
$$N_W(t) + N_p(t) = N_u(t), \text{ per ogni } t,$$
$$N_u(0) = n,$$
$$N_u(C_{\max}) = 0.$$

Si definisce poi una media per queste quantità:

$$\overline{N}_W = \frac{1}{C_{\max}} \int_0^{C_{\max}} N_W(t)dt, \quad \overline{N}_p = \frac{1}{C_{\max}} \int_0^{C_{\max}} N_p(t)dt, \quad \overline{N}_c = \frac{1}{C_{\max}} \int_0^{C_{\max}} N_c(t)dt,$$

$$\overline{N}_u = \frac{1}{C_{\max}} \int_0^{C_{\max}} N_u(t)dt.$$

3.1. Criteri basati sui tempi di completamento

Tali criteri di ottimalità prevedono di minimizzare il massimo tempo di flusso, $F_{\max}$, o il massimo tempo di completamento, $C_{\max}$, o *make-span*,

- $f_{\max} \in \{C_{\max}, F_{\max}\}$, dove $C_{\max} = \max_{1 \le j \le n} C_j$ e $F_{\max} = \max_{1 \le j \le n} F_j$,

oppure il tempo di flusso medio $(\overline{F})$ o il tempo di completamento medio $(\overline{C})$,

- $\overline{f} \in \{\overline{C}, \overline{F}\}$, dove $\overline{C} = \frac{1}{n}\sum_{j=1}^{n} C_j$ e $\overline{F} = \frac{1}{n}\sum_{j=1}^{n} F_j$, se si tratta di media aritmetica, oppure $\overline{C} = \frac{1}{n}\sum_{j=1}^{n} w_j C_j$ e $\overline{F} = \frac{1}{n}\sum_{j=1}^{n} w_j F_j$, se si tratta di media pesata, con w_j fattori di peso tali che $\sum w_j = 1$.

3.2. Criteri basati sulle due date

Tali criteri si pongono l'obiettivo di minimizzare la lateness massima, $L_{\max}$, la tardiness massima, $T_{\max}$, oppure la lateness media, $\overline{L}$, la tardiness media, $\overline{T}$. Si tratta quindi di minimizzare

- $f_{\max} \in \{L_{\max}, T_{\max}\}$,

oppure

- $\overline{f} \in \{\overline{L}, \overline{T}\}$.

E' opportuno minimizzare la lateness quando c'è convenienza a concludere la lavorazione dei job il prima possibile, anche se comunque in tempo; si considera la tardiness quando tale convenienza non c'è. Talvolta c'è penalità solo per il fatto che la lavorazione di un job si conclude in ritardo, indipendentemente dalla quantità del ritardo. In questo caso si tratta di minimizzare il numero di job in ritardo, n_T, cioè il numero di job la cui lavorazione viene completata dopo la due date.

3.3. Criteri basati sui costi di inventario e di utilizzo

Sono criteri in cui si tratta per esempio di minimizzare il numero medio di job in attesa $(\overline{N}_W)$, oppure il numero medio di job non ancora completati $(\overline{N}_u)$: ciò porta a minimizzare i costi di magazzino del processo di lavorazione. Oppure si tratta di minimizzare il numero medio di job completati $(\overline{N}_c)$, e ciò significa cercare di ridurre i costi di magazzino dei beni finiti. Oppure ancora si tratta di massimizzare il numero medio di job che sono in lavorazione contemporaneamente $(\overline{N}_p)$, volendo assicurare la massima

efficienza nell'uso delle macchine. Talvolta l'obiettivo è quello di minimizzare il tempo massimo di inattività $(I_{\max})$, oppure il tempo medio di inattività $(\bar{I})$ delle macchine.

3.4. Misure di performance regolari

Le funzioni obiettivo fino ad ora introdotte si dicono anche *misure di performance*. Esse si distinguono in misure di performance regolari e non regolari.

Def.: Una misura di performance, *R*, si dice *regolare* se è non decrescente negli istanti di completamento, cioè $R(C_1, C_2, ..., C_n)$ si dice misura di performance regolare se dati $(C_1, C_2, ..., C_n)$ e $(C_1', C_2', ..., C_n')$ tali che

$$C_1 \le C_1', C_2 \le C_2', ..., C_n \le C_n',$$

si ha che

$$R(C_1, C_2, ..., C_n) \le R(C_1', C_2', ..., C_n').$$

Ciò significa che dati due schedule tali che nel primo tutti i job sono completati non dopo rispetto al secondo, allora, rispetto ad una misura di performance regolare, il primo schedule è almeno buono quanto il secondo.

Teor.: Le misure di performance $C_{\max}, \bar{C}, F_{\max}, \bar{F}, L_{\max}, \bar{L}, T_{\max}, \bar{T}, n_T$ sono regolari.

Dim.: Si vuole dimostrare che $C_{\max}$ è misura di performance regolare.

$$R(C_1, C_2, ..., C_n) = C_{\max} = \max\{C_1, C_2, ..., C_n\}.$$

Siano $C_1 \le C_1', C_2 \le C_2', ..., C_n \le C_n'$; allora

$$\begin{aligned} R(C_1, C_2, ..., C_n) &= \max\{C_1, C_2, ..., C_n\} \\ &\le \max\{C_1', C_2', ..., C_n'\} \\ &= R\{C_1', C_2', ..., C_n'\}. \end{aligned}$$

Quindi $C_{\max}$ è misura di performance regolare.■

4. Esempi

- **Atterraggio in un aereoporto**

  E' un problema di scheduling con *n* job (gli aerei che arrivano all'aereoporto per atterrare in un giorno) e 1 macchina (la pista di atterraggio). La data di disponibilità di ogni job è l'istante in cui l'aereo arriva nello spazio aereo dell'aereoporto ed è

pronto per atterrare. L'obiettivo può essere quello di minimizzare il tempo medio d'attesa degli aerei prima di atterrare e ci possono essere dei vincoli di precedenza che tengono del numero di passeggeri di ogni aereo. La funzione obiettivo è allora

$$\overline{W} = \frac{1}{n}\sum_{j=1}^{n} W_j = \frac{1}{n}\sum_{j=1}^{n}(C_j - r_j - p_j) = \\ = \overline{C} - \bar{r} - \sum_{j=1}^{n} p_j$$

e i parametri che descrivono il problema sono

$$\alpha_1 = o, \alpha_2 = 1;$$
$$\beta_1 = o, \beta_2 = prec \text{ oppure } \beta_2 = o, \beta_3 = r_j, \beta_4 = o;$$
$$\gamma = \overline{W}.$$

Il problema si classifica nel modo seguente:

$$1/\ prec, r_j\ /\overline{W} \text{ oppure } 1/\ r_j\ /\overline{W}\ .$$

♦ **Operazioni di pazienti in un ospedale**

Si suppone che ci siano *n* pazienti che devono essere operati dal chirurgo in una certa sala operatoria. Questi pazienti devono

- essere portati nell'anticamera della sala (macchina M_1);
- fatti entrare nella camera operatoria e preparati per l'operazione (macchina M_2);
- operati (macchina M_3);
- fatti uscire dalla camera operatoria (macchina M_4).

Si tratta di un problema di scheduling con *n* job (i pazienti) e 4 macchine, in cui ogni job deve essere lavorato su tutte e quattro le macchine e seguendo un ordine ben preciso: M_1, M_2, M_3, M_4. Risulta pertanto un *flow-shop*. L'obiettivo può essere quello di concludere tutte le operazioni il più velocemente possibile, dando ovviamente precedenza ai pazienti più urgenti. Il problema è allora del tipo $F,4/\ prec/C_{\max}$.

♦ **Organizzazione di un ciclo di seminari**

Si vuole organizzare un ciclo di 14 seminari da tenersi in un'unica giornata a Venezia. L'Università mette a disposizione 5 aule e dispone che le conferenze possano iniziare alle 8.00 e finire alle 17.00, prevedendo un'ora di pausa dalle 12.00 alle 13.00. Si prevedono moduli di un'ora e ogni conferenza può richiedere uno, due

o tre moduli. Inoltre ogni conferenziere può decidere quando collocare la propria conferenza nell'arco della giornata, cioè scegliere quali moduli ricoprire:

seminari	*A*	*B*	*C*	*D*	*E*	*F*	*G*	*H*	*I*	*J*	*K*	*L*	*M*	*N*
periodi	2	8	4,5	1,2	3,4,5	6	7,8	2,3	1,2	5	6,7	3,4	8	2,3,4

Si tratta di costruire uno schedule (se ne esiste uno) che preveda di usare non più di 5 aule, e di durare non più di 8 ore. Si tratta cioè di assegnare i seminari (14 job) alle aule (5 macchine), rispettando i vincoli di tempo e accontentando le richieste dei dei conferenzieri. Non è detto che un tale schedule esista: se per esempio più di 5 conferenzieri chiedessero di tenere la loro conferenza nella seconda ora, evidentemente non si riuscirebbe ad organizzare 5 sessioni parallele rispettando le richieste dei conferenzieri. I dati riportati non presentano questo problema, infatti sono al più cinque i conferenzieri che chiedono di parlare contemporaneamente. Uno schedule ammissibile è il seguente:

moduli	1	2	3	4	5	6	7	8
aula 1	*D*	*D*	*E*	*E*	*E*			*B*
aula 2	*I*	*I*	*L*	*L*		*F*		*M*
aula 3		*A*		*C*	*C*			
aula 4		*H*	*H*		*J*		*G*	*G*
aula 5		*N*	*N*	*N*		*K*	*K*	

Si supponga ora che alcune conferenze richiedano particolari supporti disponibili solo in alcune aule e che in particolare le conferenze *A*, *D*, *E* e *I* debbano essere tenute nelle aule 2,3,1 e 4 rispettivamente.

moduli	1	2	3	4	5	6	7	8
aula 1	*I*	*I*			*J*			*B*
aula 2		*A*	*L*	*L*		*F*		*M*
aula 3	*D*	*D*		*C*	*C*			
aula 4		*H*	***H,E***	*E*	*E*		*G*	*G*
aula 5		*N*	*N*	*N*		*K*	*K*	

Si osservi che con tali vincoli aggiuntivi uno schedule ammissibile non esiste. Si potrebbe ovviare al problema convincendo il relatore della conferenza *E* ad utilizzare l'aula 1, oppure richiedere all'Università un'aula in più.

5. Riducibilità tra problemi di scheduling

Def.: Due misure di performance si dicono *equivalenti* se uno schedule ottimo rispetto all'una è ottimo anche rispetto all'altra.

Siano R_1 e R_2 due misure di performance. Si dice che R_1 e R_2 sono equivalenti se lo schedule S^*, ottimo rispetto a R_1, è ottimo anche rispetto a R_2 e viceversa:

$$S^* \text{ ottimo rispetto } R_1 \Leftrightarrow S^* \text{ ottimo rispetto } R_2 .$$

Si sta dicendo una cosa molto importante, infatti si sta dicendo che due problemi con funzioni obiettivo formalmente diverse hanno la stesso soluzione ottima. In seguito si mostra che alcune misure di performance introdotte nel paragrafo 3 sono equivalenti.

Proposizione: Le misure di performance $\overline{C}, \overline{F}, \overline{L}$ sono equivalenti.

Dim.: Dalle definizioni date segue immediatamente che per ogni job J_j si ha

$$C_j = F_j + r_j = L_j + d_j .$$

Sommando su tutti i job (eventualmente pesati) e dividendo per *n*, si ottiene:

$$\frac{1}{n}\sum_{j=1}^{n} C_j = \frac{1}{n}\sum_{j=1}^{n}(F_j + r_j) = \frac{1}{n}\sum_{j=1}^{n}(L_j + d_j)$$

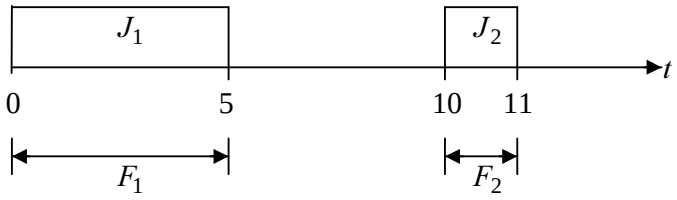
e quindi

$$\overline{C} = \overline{F} + \overline{r} = \overline{L} + \overline{d} .$$

Ora, le quantità $\overline{r}, \overline{d}$ sono costanti e indipendenti dallo schedule. Quindi, scegliere uno schedule che minimizza $\overline{C}$ significa scegliere uno schedule che minimizza $\overline{F}, \overline{L}$, e così via.■

Non vale un risultato analogo per le misure di performance $C_{\max}, F_{\max}, L_{\max}$. Per esempio non è in generale vero che $C_{\max} = F_{\max} + r_{\max}$.

Esempio: Si consideri un problema con m=1 e n=2, tale che $r_1 = 0, p_{11} = 5$, $r_2 = 10, p_{12} = 1$. Eseguendo prima il job J_1 e poi il job J_2 si ha che $C_1 = 5$, $C_2 = 11$, $F_1 = 5$ e $F_2 = 1$. Infatti si inizia all'istante 0 la lavorazione del job 1 che richiede un tempo uguale a 5 e, dato che la data di disponibilità del job 2 è 10, anche se la macchina sarebbe libera all'istante 5 si deve necessariamente attendere l'istante 10 per iniziare la nuova lavorazione, che si concluderà all'istante 11.



È evidente che la soluzione proposta è ottima rispetto a $C_{\max}$ ma non rispetto a $F_{\max}$. Infatti $C_{\max}$ è il massimo tra i due tempi di completamento: il job 1 viene completato all'istante 5 e il job 2 viene completato all'istante 11, quindi $C_{\max} = 11$. $F_{\max}$ è invece il massimo tra i due tempi di flusso: $F_1 = 5 - 0 = 5$ e $F_2 = 11 - 10 = 1$, quindi $F_{\max} = 5$. Ora si osservi che non vale la relazione $C_{\max} = F_{\max} + r_{\max}$, infatti $r_{\max} = 10$ e quindi $C_{\max} = 11 \neq F_{\max} + r_{\max} = 5 + 10$. Dunque minimizzare il massimo tempo di completamento non è equivalente a minimizzare il massimo tempo di flusso.

Osservazione: Si osservi che, nel caso particolare in cui tutte le *release time* siano nulle, le misure di performance $C_{\max}$ e $F_{\max}$ sono equivalenti.

Si dimostra il seguente risultato:

Proposizione: Ogni schedule che minimizza $L_{\max}$ minimizza anche $T_{\max}$.

Dim.: Si deduce immediatamente da

$$
\begin{aligned}
T_{\max} &= \max\{T_1, T_2, ..., T_n\} \\
&= \max\{\max\{L_1, 0\}, \max\{L_2, 0\}, ..., \max\{L_n, 0\}\} = \\
&= \max\{L_1, L_2, ..., L_n, 0\} = \\
&= \max\left\{\max_{1 \le j \le n}\{L_j\}, 0\right\} = \\
&= \max\{L_{\max}, 0\}.
\end{aligned}
$$

Infatti sia S^* uno schedule che minimizza $L_{\max}$; il valore ottimo di $L_{\max}$ potrà essere positivo, nullo o negativo. Se $L_{\max}$ è positivo o nullo, allora $T_{\max} = L_{\max}$; se $L_{\max}$ è invece negativo, cioè tutti i job vengono conclusi in anticipo, allora $T_{\max} = 0$; in entrambi i casi uno schedule che minimizza $L_{\max}$ è uno schedule che minimizza anche $T_{\max}$.■

Non è vero il viceversa, infatti siano *S* e *S'* due schedule per cui $L_{\max}(S) < L_{\max}(S') \le 0$, cioè due schedule i cui job si concludono tutti entro la *due date*. La misura di performance $T_{\max}$ vale 0 sia in *S* che in *S'*, quindi entrambi gli schedule sono soluzione ottima $T_{\max}$. Tuttavia lo schedule *S'* non è sicuramente soluzione ottima per la misura di performance $L_{\max}$. Quindi è stato trovato uno schedule che minimizza $T_{\max}$ ma non minimizza $L_{\max}$.

Osservazione: Si osservi che, nel caso particolare in cui $L_{\max}$ sia positivo in tutti gli schedule ammissibili, uno schedule ottimo per $T_{\max}$ è anche ottimo per $L_{\max}$.

Si dimostra che è equivalente minimizzare il massimo istante di completamento, massimizzare il numero di job lavorati contemporaneamente o minimizzare il tempo medio di inattività delle macchine:

Proposizione: Le misure $C_{\max}, \overline{N}_p, \overline{I}$ sono equivalenti.

Dim.: Si vuole dimostrare che uno schedule ottimo per $C_{\max}$ è anche ottimo per $\overline{N}_p$ (numero medio di lavori processati contemporaneamente) e per $\overline{I}$ (tempo medio di inattività delle macchine) e viceversa.

(i) Equivalenza di $C_{\max}, \overline{N}_p$:

per definizione

$$\overline{N}_p = \frac{1}{C_{\max}} \int_0^{C_{\max}} N_p(t)dt ,$$

dove $N_p(t)$ è il numero di job che sono contemporaneamente in lavorazione all'istante *t*. Per ogni job J_j si introduce una variabile binaria $\delta_j(t)$ che assume valore 1 se il job J_j è in lavorazione all'istante *t*, 0 altrimenti:

$$\delta_j(t) = \begin{cases} 1, & \text{se } J_j \text{ è in lavorazione all'istante } t, \\ 0, & \text{altrimenti.} \end{cases}$$

Si osservi che $N_p(t) = \sum_{j=1}^{n} \delta_j(t)$. Ora, nell'intervallo $[0, C_{\max}]$, $\delta_j(t) = 1$ per un periodo di tempo complessivo uguale a $\sum_{i=1}^{m} p_{ij}$, cioè si tiene conto del tempo di lavorazione del job J_j su ognuna delle *m* macchine. Così

$$\int_0^{C_{\max}} \delta_j(t)dt = \sum_{i=1}^{m} p_{ij}$$

e quindi

$$\int_0^{C_{\max}} N_p(t)dt = \sum_{j=1}^{n} \int_0^{C_{\max}} \delta_j(t)dt = \sum_{j=1}^{n}\sum_{i=1}^{m} p_{ij} ,$$

da cui

$$\overline{N}_p = \frac{\sum_{j=1}^{n}\sum_{i=1}^{m} p_{ij}}{C_{\max}} .$$

Si deduce allora che, essendo $\sum_{j=1}^{n}\sum_{i=1}^{m} p_{ij}$ una costante indipendente dallo schedule, $\overline{N}_p$ è inversamente proporzionale a $C_{\max}$ e quindi massimizzare $\overline{N}_p$ è equivalente a minimizzare $C_{\max}$.

(ii) Equivalenza di $C_{\max}$ e $\bar{I}$:

Dalla relazione

$$I_i = C_{\max} - \sum_{j=1}^{n} p_{ij} ,$$

sommando su *i* e dividendo per *m* si ottiene

$$\bar{I} = \frac{1}{m}\sum_{i=1}^{m} I_i = \frac{1}{m}(mC_{\max} - \sum_{i=1}^{m}\sum_{j=1}^{n} p_{ij}) = C_{\max} - \frac{1}{m}\sum_{i=1}^{m}\sum_{j=1}^{n} p_{ij} .$$

Si deduce che $C_{\max}$ e $\bar{I}$ differiscono per una costante e quindi minimizzare l'uno equivale a minimizzare l'altro.■

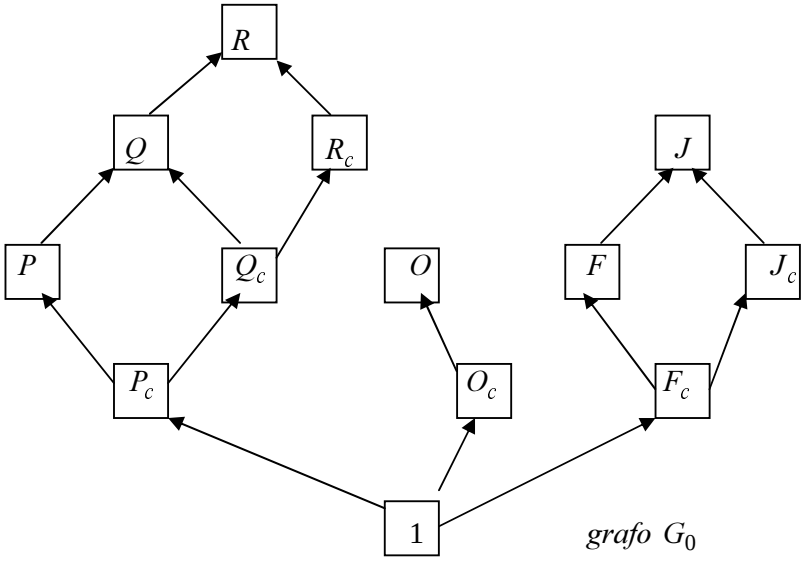
Tutto quanto è stato detto sull'equivalenza tra misure di performance riveste una grossa importanza. Quando si affronta un problema di scheduling e le caratteristiche delle macchine e dei job sono dati, chi risolve il problema può scegliere l'obiettivo, a meno che questo non sia stato fissato esternamente. Il fatto che ci sia equivalenza tra diversi obiettivi permette di evitare di risolvere alcuni problemi quando se ne siano già risolti altri con obiettivi equivalenti.

Nel seguito si vedrà una rappresentazione dei problemi di scheduling mediante grafi. Questa rappresentazione è molto più significativa della classificazione $\alpha / \beta / \gamma$ perché da essa si può dedurre molto sulla complessità dei problemi trattati. La complessità per i problemi di scheduling è molto importante, in quanto si dimostra che la maggior parte di essi sono problemi difficili.

Ogni problema di scheduling viene individuato da una lista di sei elementi, $(u_0, u_1, u_2, u_3, u_4, u_5)$, in cui u_i è un nodo del grafo G_i riportato nelle seguenti figure. In seguito verranno descritti i 6 grafi $(G_0, G_1, G_2, G_3, G_4, G_5)$. Ogni nodo dei 6 grafi rappresenta una caratteristica di un problema e gli archi orientati indicano che il nodo secondo estremo dell'arco rappresenta un problema più complesso del predecessore.

Il grafo G_0 riassume le informazioni contenute nel primo campo della classificazione $\alpha / \beta / \gamma$, cioè la parte che descriveva la struttura delle macchine del problema di scheduling considerato. Per quanto riguarda le informazioni contenute in tale campo, il problema più semplice è quello in cui c'è una sola macchina, 1. Complicando via via, se i job sono costituiti da un'unica operazione, si trovano il caso di un numero prefissato, c, *di macchine parallele identiche, P_c, il caso di un numero prefissato,* c, *di macchine parallele uniformi, Q_c, e il caso di un numero prefissato,* c, *di macchine parallele indipendenti, R_c. Ad ognuno di questi ultimi tre problemi corrisponde la versione più complicata in cui il numero di macchine non sia prefissato:* P, Q *e* R. *Ovviamente* R *risulta più complicato di* Q *che risulta più complicato di* P. *Se invece i job sono costituiti da più operazioni e l'ordine di lavorazione delle operazioni sulle diverse macchine non è fissato, si ha il*

problema open shop*, con un numero prefissato,* c*, di macchine,* O_c*, la cui immediata complicazione è l'*open shop *con un numero non fissato di macchine,* O*. Se infine i job sono costituiti da più operazioni e l'ordine di lavorazione delle operazioni sulle diverse macchine è fissato, si trova il* flow shop*, con un numero prefissato,* c*, di macchine,* F_c*, in cui l'ordine di esecuzione è lo stesso per tutti i job, e, immediatamente più complicato, il* job shop*, con un numero fissato,* c*, di macchine,* J_c*, in cui l'ordine di esecuzione può variare da job a job. Ovviamente versioni più complicate di questi sono quelle in cui il numero delle macchine non è fissato,* F *e* J.

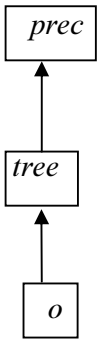


grafo G_0

Il grafo G_1 ha una struttura molto più semplice, infatti è costituito da due soli nodi isolati che dicono se è permessa la preemption dei job oppure no.

grafo G_1

Il grafo G_2 rappresenta le precedenze tra i job: il caso più semplice è quello in cui non ci sono precedenze, seguito dal caso in cui ci siano precedenze ad albero, seguito a sua volta dal caso in cui ci siano precedenze con struttura più complicata.

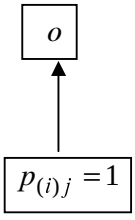


grafo G_2

Il grafo G_3 rappresenta le *release date* dei job. I casi possibili sono due: il più semplice è quello in cui i job siano tutti disponibili da subito; il più complicato è quello in cui i job abbiano date di disponibilità diverse, r_j .

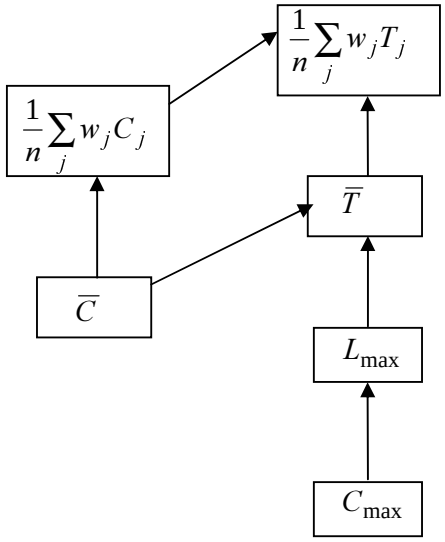
grafo G_3

Il grafo G_4 rappresenta i tempi di esecuzione: il caso in cui ciascun job (o ciascuna operazione) abbia tempo di esecuzione unitario è sicuramente più semplice del caso in cui i tempi di esecuzione siano tutti diversi tra loro.



grafo G_4

Il grafo G_5 rappresenta gli obiettivi e cioè le misure di performance. Il caso più semplice è quello con misura di performance $C_{\max}$, seguito nell'ordine da $L_{\max}$, $\overline{T}$ e da $\frac{1}{n}\sum_j w_j T_j$. Anche $\overline{C}$ risulta più semplice di $\frac{1}{n}\sum_j w_j C_j$ e, rispettivamente, $\overline{C}$ e $\frac{1}{n}\sum_j w_j C_j$ risultano più semplici di $\overline{T}$ e $\frac{1}{n}\sum_j w_j T_j$.



grafo G_5

Dati due problemi $P = (u_0,...,u_5)$ e $Q = (v_0,...,v_5)$, si scrive $P \to Q$, e si dice che il problema *P* è riducibile polinomialmente al problema *Q*, se $u_i = v_i$ oppure se G_i contiene un cammino orientato da u_i a v_i, con $i = 0,...,5$. Si dimostra che se $P \to Q$ e *Q* è risolubile in tempo polinomiale, allora anche *P* è risolubile in tempo polinomiale; se $P \to Q$ e *P* è *NP-hard*, allora anche *Q* è *NP-hard.*

6. Complessità dei problemi di scheduling

La complessità degli algoritmi che risolvono in modo ottimo i problemi di scheduling permette di suddividere tali problemi in classi per le quali sono necessari diversi approcci. La distinzione principale è ovviamente tra problemi facili e difficili.

Se il problema è facile è naturale richiedere la soluzione ottima e concentrarsi sulla riduzione della complessità dell'algoritmo. Per i problemi difficili (*NP*-completi o *NP*-hard), non potendo ottenere in tempi ragionevoli la soluzione ottima, si possono

effettuare dei rilassamenti, rendendo più deboli alcuni vincoli ed ottenendo problemi meglio trattabili, le cui soluzioni sono "vicine" alla soluzione del problema di partenza, oppure ricorrere ad algoritmi approssimati, in gradi di determinare soluzioni buone, senza richiedere l'ottimalità. In questo caso si cerca contemporaneamente di ridurre la complessità dell'algoritmo e di migliorare il grado di approssimazione della soluzione fornita rispetto alla soluzione ottima.

7. Macchina singola

Nel seguito si considerano problemi in cui è disponibile un'unica macchina per la lavorazione dei job e i job sono disponibili per essere lavorati all'inizio del periodo considerato.

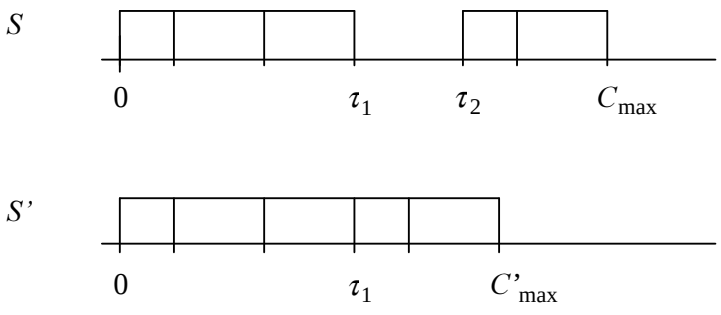
Ipotesi : i) $m = 1$;

ii) $r_j = 0$, per ogni J_j, $j = 1,2,...,n$.

Nel seguente teorema si dimostra che non è necessario considerare i tempi di inattività in un problema a macchina singola con misura di performance regolare.

Teor.: Per un problema 1*/o/B*, con *B* misura di performance regolare, esiste uno schedule ottimo in cui i tempi di inattività sono nulli, cioè la macchina comincia a lavorare all'istante *t*=0 e continua senza sosta fino a $t = C_{\max}$.

Dim.: Sicuramente esiste uno schedule ottimo, poichè si tratta di individuarlo in un insieme finito di schedule ammissibili. Sia *S* uno schedule ottimo con intervallo di inattività $[\tau_1, \tau_2]$. Sia *S'* lo schedule che si ottiene da *S* anticipando di $\tau_2 - \tau_1$ tutte le operazioni che cominciano dopo τ_1.

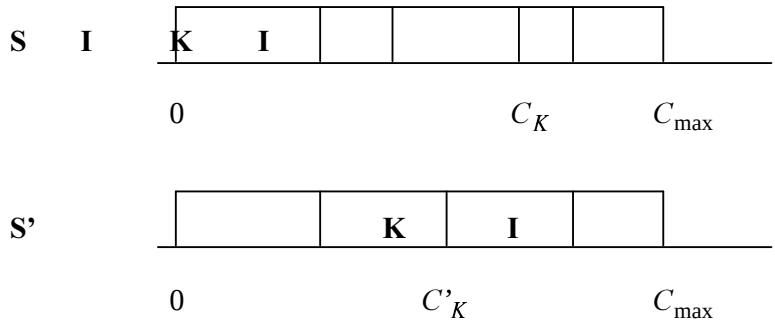


Si noti che *S′* è ammissibile perchè c'è un'unica macchina. Chiaramente, detti C_j e C_j' gli istanti di completamento del job J_j nello schedule *S* e *S′* rispettivamente, vale la relazione $C_j' \leq C_j$, per ogni $j = 1,2,...,n$. Di conseguenza, essendo *B* misura di performance regolare, si ha $R(C_1', C_2', ..., C_n') \leq R(C_1, C_2, ..., C_n)$. Quindi anche *S′* è ottimo, se vale l'uguaglianza, mentre *S* non era schedule ottimo, ma lo è *S'*, se vale la disuguaglianza stretta.■

Altro risultato interessante riguarda il fatto che non è mai vantaggioso in questo tipo di problemi considerare la preemption.

Teor.: In un problema 1*/o/B*, con *B* misura di performance regolare, non è mai vantaggioso ammettere la preemption.

Dim.: Sia *S* uno schedule ottimo in cui la lavorazione del job *I* viene interrotta per permettere la lavorazione del job *K* e poi ripresa; si supponga che la lavorazione del job *K* non subisca interruzioni. Sia *S′* lo schedule che si ottiene da *S* scambiando la prima parte di *I* con *K*.



Gli istanti di completamento non subiscono variazioni a parte l'istante di completamento del job *K* per cui $C_K' < C_K$. Quindi il valore di una misura regolare non può diminuire nel passaggio da *S* a *S′*. Anche *S′* è ottimo.■

Allora nei problemi a macchina singola non conviene lasciare la macchina ferma, quindi è preferibile farla lavorare ininterrottamente dall'istante 0 all'istante $C_{\max}$, e non conviene permettere la preemption dei job. In problemi di questo tipo si tratta solo di trovare quella particolare permutazione dei job che minimizza il valore di *B*.

7.1. Problemi senza vincoli di precedenza ($\beta_2 = o$).

♦ **Shortest processing time (SPT),** $1/o/\overline{F}$

Si considera il problema $1/o/\overline{F}$, in cui è disponibile una macchina e si vuole minimizzare il flusso medio, $\overline{F}$. Si osservi che, essendo tutte nulle le date di disponibilità dei job, il tempo di flusso del job J_j coincide con il suo tempo di completamento: $F_j = C_j - r_j = C_j$. Si dimostra il seguente risultato.

Teor.: Una soluzione ottima di un problema $1/o/\overline{F}$ è uno schedule

$$(J_{j(1)}, J_{j(2)}, ..., J_{j(n)}),$$

detto *schedule SPT* (da Shortest Processing Time), tale che

$$p_{j(1)} \le p_{j(2)} \le ... \le p_{j(n)},$$

con $p_{j(k)}$ tempo di esecuzione del job che viene eseguito per *k*-esimo.

Dim.: Si tratta sostanzialmente di schedulare prima il job con tempo di esecuzione più corto e continuare così fino alla fine. Si suppone per assurdo che *S* sia uno schedule non-*SPT* ottimo. Allora per qualche *k* si ha che $p_{j(k)} > p_{j(k+1)}$. Sia *S′* lo schedule che si ottiene da *S* scambiando tra loro i job $I = J_{j(k)}$ e $K = J_{j(k+1)}$. Nello schedule *S* e nello schedule *S′* tutti i job diversi da *I* e *K* hanno lo stesso tempo di flusso, quindi la differenza tra $\overline{F}$ e $\overline{F}$' dipende dai tempi di flusso di *I* e *K*. I tempi di flusso dei job *I* e *K* sono, in *S*,

$$F_I = \sum_{l=1}^{k-1} p_{j(l)} + p_I \quad \text{e} \quad F_K = \sum_{l=1}^{k-1} p_{j(l)} + p_I + p_K$$

e, in *S′*,

$$F_I' = \sum_{l=1}^{k-1} p_{j(l)} + p_K + p_I \quad \text{e} \quad F_K' = \sum_{l=1}^{k-1} p_{j(l)} + p_K .$$

Quindi il contributo di *I* e *K* a $\overline{F}$ è, in *S*,

$$\frac{1}{n}(F_I + F_K) = \frac{1}{n}(2\sum_{j=1}^{k-1} p_{j(l)} + 2p_I + p_K)$$

e, in *S′*,

$$\frac{1}{n}(F_I' + F_K') = \frac{1}{n}(2\sum_{j=1}^{k-1} p_{j(l)} + p_I + 2p_K).$$

Poichè $p_I > p_K$, è evidente che il contributo di I e K a $\overline{F}$ in S è maggiore del contributo di I e K a $\overline{F}$ in S' e quindi, dato uno schedule non-*SPT*, se ne può sempre trovare uno a cui corrisponde un flusso medio inferiore; di conseguenza uno schedule non-*SPT* non può essere ottimo.■

Dunque la regola *SPT* fornisce uno schedule ottimo per il problema $1/o/\overline{F}$. Si può quindi formulare un algoritmo risolutivo molto semplice.

Algoritmo SPT

Passo 1: Ordina i job per tempo di esecuzione non decrescente;

Passo 2: Schedula i job seguendo quest'ordine.

Le operazioni che questo algoritmo deve compiere sono sostanzialmente l'ordinamento del vettore dei tempi di esecuzione dei job e la costruzione della lista dei lavori individuata dall'ordinamento. Come si è visto l'algoritmo più veloce per ordinare n numeri ha complessità $n\log n$; inoltre la trascrizione dei job richiede esattamente n operazioni. Complessivamente $n\log n+n$ operazioni, nel caso peggiore. Dunque la complessità di questo algoritmo è $n\log n$. Essendo l'algoritmo *SPT* polinomiale, il problema $1/o/\overline{F}$ è un problema facile.

- **Earliest due date scheduling (EDD),** $1/o/L_{\max}$

Si considera il problema $1/o/L_{\max}$, in cui è disponibile una macchina e si tratta di minimizzare la massima lateness, $L_{\max}$. Si dimostra il seguente risultato.

Teor.: Una soluzione ottima di un problema $1/o/L_{\max}$ è lo schedule

$$(J_{j(1)}, J_{j(2)}, ..., J_{j(n)}),$$

detto *schedule EDD* (da Earliest Due Date), per cui

$$d_{j(1)} \le d_{j(2)} \le ... \le d_{j(n)},$$

dove $d_{j(k)}$ è la *due date* del job che viene eseguito per k-esimo.

Dim.: Sia S uno schedule non-*EDD*, cioè uno schedule in cui i job non sono ordinati per *due date* non decrescente. Quindi per qualche k si ha che $d_{j(k)} > d_{j(k+1)}$. Posto

$I = J_{j(k)}$ e $K = J_{j(k+1)}$, si ha quindi $d_I > d_K$. Sia S' lo schedule che si ottiene da S scambiando tra loro I e K e lasciando il resto invariato. Siano poi L e L' le massime lateness degli *n-2* job diversi da I e K nello schedule S e nello schedule S' rispettivamente. Ovviamente $L=L'$. Siano L_I e L_K le *lateness* di I e K nello schedule S e L_I' e L_K' le *lateness* di I e K nello schedule S'. Allora le *lateness* massime nello schedule S e nello schedule S' sono rispettivamente

$$L_{\max} = \max(L, L_I, L_K),$$

$$L_{\max}\text{'} = \max(L', L_I\text{'}, L_K\text{'}) = \\ = \max(L, L_I\text{'}, L_K\text{'}).$$

Ora, nello schedule S si ha

$$L_I = a + p_I - d_I,$$
$$L_K = a + p_I + p_K - d_K,$$

dove $a = \sum_{l=1}^{k-1} p_{j(l)}$, e nello schedule S' si ha

$$L_K\text{'} = a + p_K - d_K,$$
$$L_I\text{'} = a + p_K + p_I - d_I.$$

Si osservi che

$$L_K > L_K', \quad \text{poichè } p_I > 0,$$

e

$$L_K > L_I', \quad \text{poichè } d_I > d_K.$$

Quindi

$$L_K > \max(L_I\text{'}, L_K\text{'}),$$

da cui

$$L_{\max} = \max(L, L_I, L_K) \geq \max(L, L_K) \\ \geq \max(L, L_I\text{'}, L_K\text{'}) = \max(L', L_I\text{'}, L_K\text{'}) = L_{\max}\text{'}.$$

Ciò dimostra che, dato uno schedule non-*EDD*, o questo non può essere ottimo (se vale la disuguaglianza stretta), oppure questo è ottimo ma esiste un altro schedule S' ottimo (se vale l'uguaglianza). Se S' è uno schedule *EDD*, allora si è dimostrato il teorema, altrimenti si applica il procedimento fino a che si trova uno schedule *EDD*.■

Si può sviluppare un algoritmo risolutivo simile a quello visto per *SPT*, avente la stessa complessità.

Algoritmo EDD

Passo 1: Ordina i job per *due date* non decrescente;

Passo 2: Schedula i job seguendo quest'ordine.

Si osservi inoltre che una soluzione del problema $1/o/L_{\max}$ è anche soluzione del problema $1/o/T_{\max}$, dal momento che, come è stato dimostrato, uno schedule che minimizza $L_{\max}$ minimizza anche $T_{\max}$.

Esempio

Si risolva il problema $1/o/T_{\max}$ con i seguenti dati:

job	A	B	C	D	E	F
due date	7	3	8	12	9	3
tempo di esec.	1	1	2	4	1	3

Per il teorema precedente la soluzione ottima di questo problema è (F, B, A, C, E, D). Volendo calcolare $T_{\max}$,

job	F	B	A	C	E	D
tempo di complet. $C_{j(k)} = \sum_{l=1}^{k} p_{j(k)}$	3	4	5	7	8	12
lateness $L_{j(k)} = C_{j(k)} - d_{j(k)}$	0	1	-2	-1	-1	0
tardiness $T_{j(k)} = \max(0, L_{j(k)})$	0	1	0	0	0	0

da cui si ricava che $T_{\max} = 1$.

7.2. Problemi con vincoli di precedenza ($\beta_2 = prec$)

♦ **Algoritmo di Lawler per i problemi** $1/prec/f_{\max}$

Si considera un problema a macchina singola in cui ci sono dei vincoli di precedenza e si tratta di minimizzare $f_{\max} = \max_{1 \le j \le n} f_j$, con f_j funzione di costo associata al job J_j, misura di performance regolare. Per risolvere un problema di questo tipo si usa un algoritmo, algoritmo di Lawler, che si basa sul seguente teorema:

Teor.: Detto $N = \{1,2,...,n\}$ l'insieme degli indici dei job, sia $L \subseteq N$ l'insieme degli indici dei job che non devono necessariamente avere successori. Posto

$$p(S) = \sum_{j \in S} p_j, \text{ per ogni } S \subseteq N,$$

sia $J_l, l \in L$, un job tale che

$$f_l(p(N)) = \min_{j \in L} f_j(p(N)),$$

dove la quantità $f_j(p(N))$ rappresenta il costo dovuto al fatto che il job J_j viene completato all'istante $p(N)$. Allora c'è uno schedule ottimo in cui J_l è eseguito per ultimo.

Dim.: Si indica con $f^*_{\max}(S)$ il costo di uno schedule ottimo per *S*, *S* sottoinsieme di *N*. $f^*_{\max}(N)$ rappresenta il costo di uno schedule ottimo per *N*. Per $f^*_{\max}(N)$ valgono le seguenti disuguaglianze:

$$f^*_{\max}(N) = \min f_{\max}(N) = \min \max_{j \in N} f_j(p(N)) \ge \min \max_{j \in L} f_j(p(N)) \ge$$
$$\ge \min_{j \in L} f_j(p(N)) = f_l(p(N)),$$

$$f^*_{\max}(N) \ge f^*_{\max}(N - \{j\}), \text{ per ogni } j \in N.$$

La prima disuguaglianza dipende dal fatto che il costo minimo che si ha nello schedulare tutti i job è maggiore o uguale del costo minimo che si ha nello schedulare tutti i job scegliendo come ultimo un job che non debba necessariamente avere successori. Infatti se all'istante $p(N)$ si concludesse un lavoro che non sta in *L*, allora $C_{\max}$ aumenterebbe e quindi anche $f_{\max}$, in quanto funzione crescente di $C_{\max}$. La seconda disuguaglianza dipende dal fatto che il costo per schedulare *n* job è sempre maggiore o uguale del costo per schedulare *n*-1 job. Allora

$$f^*_{\max}(N) \geq \max\{f_l(p(N)), f^*_{\max}(N-\{l\})\}.$$

La quantità a secondo membro nella precedente disuguaglianza rappresenta il costo di uno schedule ottimo in cui J_l viene eseguito per ultimo. Da ciò si deduce che esiste uno schedule ottimo in cui J_l viene eseguito per ultimo.■

Sia J_l il job che deve essere eseguito per ultimo sulla base del precedente teorema. Quindi c'è uno schedule ottimo del tipo (A, J_l), dove A è una permutazione degli altri *n-1* job. Poichè il massimo costo di questa sequenza è il più grande tra il costo di completamento del job J_l e il massimo costo di completamento dei job in A, una sequenza ottima si può individuare cercando di minimizzare il costo di completamento dei job in A, riproponendo il problema originale su un insieme di *n-1* job. Iterando questa procedura si arriva ad individuare una sequenza ottima avente per ultimo il job J_l. A questo punto è possibile formalizzare l'*algoritmo di Lawler*

Algoritmo di Lawler

Passo 1: Costruire l'insieme L iniziale, cioè l'insieme degli indici dei job che non devono necessariamente avere successori. Andare al passo 2.

Passo 2: Determinare un job J_l per cui $f_l(p(N)) = \min_{j \in L} f_j(p(N))$. Andare al passo 3.

Passo 3: Schedulare il job J_l per ultimo. Porre $N := N - \{l\}$. Andare al passo 4.

Passo 4: Aggiornare l'insieme L: togliere l'indice l e tenere l'insieme differenza (L:=L-{l}) ed eventualmente inserire il job che precede J_l, se questo non deve necessariamente avere successori. Andare al passo 5.

Passo 5: Se $L = \varnothing$ allora STOP.
Se $L \neq \varnothing$ allora andare al passo 2.

Si vuole ora determinare la complessità dell'algoritmo sopra descritto. Si può dimostrare che la complessità di questo algoritmo è dell'ordine $O(n^2)$, tenendo conto che per determinare lo schedule che deve essere eseguito per ultimo sono richieste al più n operazioni e che ciò deve essere ripetuto tante volte quanti sono i job e quindi

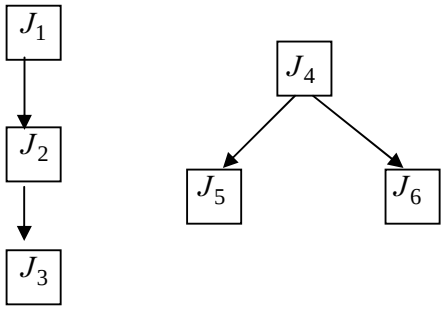
n. Complessivamente saranno richieste, nel caso peggiore, $n \cdot n$ operazioni. Allora l'algoritmo ha complessità $O(n^2)$.

Esempio

Si consideri il problema $1/prec/L_{\max}$, con *n*=6, in cui i vincoli di precedenza sono i seguenti: J_1 deve precedere J_2;

J_2 deve precedere J_3;

J_4 deve precedere J_5 e J_6.



I tempi di esecuzione e le due date siano:

job	1	2	3	4	5	6
tempi di esecuz.	2	3	4	3	2	1
due date	3	6	9	7	11	7

Per trovare il job che è opportuno sia eseguito per ultimo:

$L = \{3,5,6\}$;

poichè $\sum_{j=1}^{6} p_j = 15$, $L_3 = 15-9 = 6, L_5 = 15-11 = 4, L_6 = 15-7 = 8$;

essendo $L_5 < L_3, L_6$, l'ultimo job dello schedule è J_5.

Per trovare il job che è opportuno sia eseguito per quinto:

$L = \{3,6\}$;

poichè $\sum_{\substack{j=1 \\ j\neq 5}}^{6} p_j = 13$, $L_3 = 13-9 = 4$, $L_6 = 13-7 = 6$;

essendo $L_3 < L_6$, il quinto job dello schedule è J_3.

Per trovare il job che è opportuno sia eseguito per quarto:

$L = \{2,6\}$;

poichè $\sum_{\substack{j=1 \\ j\neq 5,3}}^{6} p_j = 9$, $L_2 = 9-6 = 3$, $L_6 = 9-7 = 2$;

essendo $L_6 < L_2$, il quarto job dello schedule è J_6.

Per trovare il job che è opportuno sia eseguito per terzo:

$L = \{2,4\}$;

poichè $\sum_{\substack{j=1 \\ j\neq 5,3,6}}^{6} p_j = 8$, $L_2 = 8-6 = 2$, $L_4 = 8-7 = 1$;

essendo $L_1 < L_2$, il terzo job dello schedule è J_4.

Per trovare il job che è opportuno sia eseguito per secondo:

$L = \{2\}$;

quindi il secondo job dello schedule è J_2.

Il primo job dello schedule è inevitabilmente J_1.

Lo schedule ottimo risulta allora $(J_1, J_2, J_4, J_6, J_3, J_5)$.

8. Macchine parallele

Nel seguito si considerano problemi caratterizzati da due o più macchine che possono essere identiche oppure avere caratteristiche diverse. Si tratta di assegnare tutti i job da lavorare alle macchine esistenti ottimizzando prefissate misure di performance.

8.1. Macchine parallele identiche

Si consideri il caso in cui si devono assegnare n lavori, formati da un'unica operazione ciascuno, senza vincoli di precedenza, ad m macchine parallele identiche.

8.1.1. Il problema $P,m/o/\frac{1}{n}\sum_{j=1}^{n}C_j$

L'obiettivo è quello di minimizzare il tempo medio di completamento. Il problema si risolve in modo esatto in tempo polinomiale ($O(n\ln n)$).

Teor.: Per il problema $P,m/o/\frac{1}{n}\sum_{j=1}^{n}C_j$ uno schedule ottimo è dato dalla successione di job $(J_{j(1)},J_{j(2)},...,J_{j(n)})$ tale che

i) $p_{j(1)} \le p_{j(2)} \le ... \le p_{j(n)}$;

ii) ogni job viene assegnato alla macchina che è libera per prima.

Dim.: Si ordinino i job secondo la *i*). Si consideri uno schedule parziale dei primi m job. Assegnando ogni job ad una macchina si ottiene uno schedule che è necessariamente una soluzione parziale di una soluzione ottima. Infatti i tempi di esecuzione dei job che vengono caricati per primi sono quelli che vengono contati più volte, quindi è opportuno che siano caricati prima i job che hanno tempi di esecuzione più brevi. Uno schedule ottimale per il job m+1-esimo sarà quello che prevede di assegnare il job alla macchina che per prima termina la lavorazione del job precedente. Seguendo questa regola di assegnazione si è certi che i primi m job assegnati sono quelli con tempo di esecuzione minore, i secondi m sono quelli con tempo di esecuzione minore tra quelli rimasti, … Ripetendo fino ad esaurire la lista dei job da assegnare si ottiene una soluzione ottima.■

L'algoritmo che si basa sul precedente teorema è il seguente:

Passo 1: Ordinare i job per tempo di esecuzione non decrescente. Andare al passo 2.

Passo 2: Schedulare i primi m lavori sulle m macchine disponibili (ovviamente generalmente sarà $m<n$). Porre k:=m+1 e andare al passo 3.

Passo 3: Schedulare il lavoro k-esimo sulla macchina che si libera per prima. Andare al passo 4.

Passo 4: Porre k:=k+1. Andare al passo 5.

Passo 5: Se $k>n$, allora STOP.

Se $k \leq n$, allora andare al passo 3.

La complessità dell'algoritmo precedente è $O(n \ln n)$, in quanto la fase più costosa è quella del passo 1 (ordinamento di n job) che richiede proprio $n \ln n$ operazioni nel caso peggiore.

8.1.2. Il problema $P, m / o / C_{\max}$

L'obiettivo è quello di minimizzare il massimo tempo di completamento. Il problema è *NP*-hard, ma esistono algoritmi semplici e con un buon grado di approssimazione.

- **L'algoritmo List-Scheduling (LS)**

  Tale algoritmo prevede di considerare una lista dei job da eseguire ed assegnarli alle macchine nell'ordine in cui si presentano nella lista.

Algoritmo LS

Passo 1: Scegliere un job qualsiasi fra quelli non ancora considerati e assegnarlo alla macchina attualmente meno carica (quella in cui la somma dei tempi di lavorazione dei job fino ad ora assegnati è minore).

Passo 2: Se vi sono ancora job da assegnare tornare al passo 1, altrimenti restituire lo schedule così ottenuto.

Si dimostra il seguente

Teor.: L'algoritmo LS fornisce soluzioni approssimate con errore relativo rispetto alla soluzione ottima $R_{LS} = \dfrac{Z_{LS} - Z_{opt}}{Z_{opt}}$ limitato da $R_{LS} \leq \dfrac{m-1}{m}$. Nella definizione di R_{LS}, Z_{opt} e Z_{LS} sono il valore ottimo della funzione obiettivo e il valore della funzione obiettivo in corrispondenza della soluzione fornita dall'algoritmo LS.

Dim.: Sia *Lw* il rapporto tra la somma dei tempi di esecuzione dei job e il numero delle macchine:

$$Lw = \frac{\sum_{j=1}^{n} p_j}{m};$$

Lw rappresenta un lower bound per il valore ottimo della funzione obiettivo Z_{opt}.

Siano J^* il job che termina per ultimo e T^* l'istante in cui inizia la sua lavorazione nello schedule fornito dall'algoritmo *LS*. Sicuramente valgono le seguenti relazioni

$$Lw \le Z_{opt} \le Z_{LS},$$

$$T^* \le \frac{\sum_{j=1}^{n} p_j - p(J^*)}{m} = Lw - \frac{p(J^*)}{m}.$$

La seconda delle due è giustificata dal fatto che all'istante T^* tutte le macchine stanno lavorando, altrimenti J^* verrebbe assegnato prima ad una macchina libera. Vale come uguaglianza solo nel caso in cui le macchine terminino di lavorare tutti i job tranne J^* contemporaneamente. In generale ci sono macchine che terminano prima e macchine che terminano dopo: J^* va caricato nella macchina che termina per prima. Allora,

$$Z_{LS} = T^* + p(J^*) \le Lw - \frac{p(J^*)}{m} + p(J^*) \le Z_{opt} + \frac{m-1}{m} p(J^*),$$

da cui si ricava che l'errore relativo R_{LS} è

$$R_{LS} = \frac{Z_{LS} - Z_{opt}}{Z_{opt}} \le \frac{Z_{opt} + \frac{m-1}{m} p(J^*) - Z_{opt}}{Z_{opt}} = \frac{m-1}{m} \frac{p(J^*)}{Z_{opt}} \le \frac{m-1}{m}.$$

L'ultima disuguaglianza deriva dal fatto che, essendo $p(J^*) \le Z_{opt}$, si ha $\frac{p(J^*)}{Z_{opt}} \le 1$.■

Si dimostra che l'algoritmo LS ha complessità ($O(n)$).

- **L'algoritmo Longest Processing Time (LPT)**

Rappresenta una variante dell'algoritmo LS leggermente più complessa ma con grado di approssimazione migliore. Prevede di ordinare preliminarmente i job per tempi di esecuzione decrescenti e di scegliere, ad ogni passo, il job più lungo tra quelli rimasti.

Algoritmo LPT

Passo 1: Ordinare i job per tempi di esecuzione decrescenti.

Passo 2: Scegliere il job con tempo di esecuzione più lungo tra quelli non ancora considerati e assegnarlo alla macchina attualmente meno carica. Ripetere il procedimento fino ad esaurimento dei job.

Si dimostra il seguente

Teor.: L'algoritmo LPT fornisce soluzioni approssimate con errore relativo rispetto alla soluzione ottima $R_{LPT} = \frac{Z_{LPT} - Z_{opt}}{Z_{opt}}$ limitato da $R_{LPT} \leq \frac{m-1}{3m}$.

Si dimostra che l'algoritmo LPT ha complessità ($O(n\ln n)$).

8.2. Macchine parallele non identiche, $R,m/o/\frac{1}{n}\sum_{j=1}^{n} C_j$

Si considera un problema in cui ci sono *m* macchine parallele non identiche, in cui cioè il tempo di lavorazione del job J_j (formato da un'unica operazione) sulla macchina M_i è $p_{ij} = \frac{p_j}{s_{ij}}$, se s_{ij} è la velocità della macchina M_i nell'eseguire il job J_j. Come misura di performance si considera il tempo medio di completamento $\frac{1}{n}\sum_{j=1}^{n} C_j$. Il problema può essere formulato come un problema di programmazione lineare intera e la sua struttura è tale per cui esso può essere risolto in tempo polinomiale.

Se $J_1, J_2, ..., J_l$ sono i job che sono lavorati dalla macchina M_i, nell'ordine indicato, la somma dei tempi di completamento per questi job è

$$\sum_{j=1}^{l} C_j = lp_{i1} + (l-1)p_{i2} + ... + p_{il}.$$

Infatti $C_1 = p_{i1}$, $C_2 = p_{i1} + p_{i2}, \ldots$, $C_l = p_{i1} + p_{i2} + ... + p_{il}$, cioè il tempo di esecuzione del job che viene lavorato per ultimo viene contato una sola volta, il tempo di esecuzione del job che viene lavorato per penultimo viene contato 2 volte,… il tempo di esecuzione del job che viene lavorato per primo viene contato tante volte quanti sono in totale i job lavorati sulla macchina M_i. Si ponga

$$x_{ikj} = \begin{cases} 1, \text{ se } J_j \text{ è il } k\text{ - ultimo job che viene lavorato dalla macchina } M_i, \\ 0, \text{ altrimenti,} \end{cases}$$

per $i \in \{1,...,m\}$, $j \in \{1,...,n\}$ e $k \in \{0,...,n\}$, dove per k=1 si trova il job J_j che viene lavorato per ultimo sulla macchina M_i, per k=2 si trova il job J_j che viene lavorato per penultimo sulla macchina M_i,… Il problema si può formulare come un problema di programmazione intera 0-1 nel modo seguente:

$$\min \sum_{i,k} \sum_{j} kp_{ij} x_{ikj},$$

$$\text{soggetto a : } \sum_{i,k} x_{ikj} = 1, \ j \in \{1,2,...,n\}, \qquad (1)$$

$$\sum_{j} x_{ikj} \le 1, \ i \in \{1,2,...,m\}; k \in \{0,1,,...,n\}, \qquad (2)$$

$$x_{ikj} \in \{0,1\}, \ i \in \{1,2,...,m\}, \ j \in \{1,2,...,n\}, k \in \{0,1,,...,n\}. \qquad (3)$$

I vincoli (1) assicurano che ogni job sia lavorato esattamente una volta, mentre i vincoli (2) si accertano che ogni posizione su ogni macchina sia occupata da al più un job. La matrice dei coefficienti dei vincoli è totalmente unimodulare, quindi i vincoli di interezza (3) possono essere sostituiti dai vincoli di non negatività $x_{ikj} \ge 0$ senza modificare la regione ammissibile. Si dimostra che il problema può essere risolto con un algoritmo di complessità $O(n^3)$.

9. Flow-shop

I problemi di *flow shop* sono problemi difficili e la complessità aumenta all'aumentare del numero delle macchine.

Il problema di *flow shop* più studiato è il problema $F,m/o/C_{\max}$. Si tratta di determinare una permutazione dei job su ogni macchina in modo da creare uno schedule avente tempo di completamento massimo (makespan) minimo, tenendo conto che è prefissato un ordine di esecuzione delle operazioni sulle macchine e questo è lo stesso per ogni job.

Esempio

Si consideri il seguente problema di *flow shop*:

m=3 macchine,

n=4 job, composti ognuno da 3 operazioni da eseguire una su ogni macchina nell'ordine indicato, $J_j = \{O_{1j}, O_{2j}, O_{3j}\}$, $j = 1,2,3,4$.

I tempi di esecuzione delle operazioni sulle macchine siano

$$p_{11} = 2,\ p_{21} = 1,\ p_{31} = 3,$$
$$p_{12} = 4,\ p_{22} = 5,\ p_{32} = 5,$$
$$p_{13} = 3,\ p_{23} = 4,\ p_{33} = 4,$$
$$p_{14} = 2,\ p_{24} = 6,\ p_{34} = 2.$$

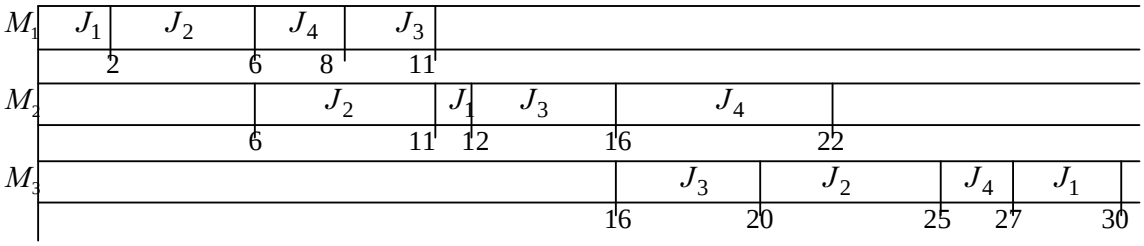
Essendo un *flow shop* l'ordine di esecuzione delle operazioni di ogni job sulle macchine deve essere lo stesso. Tuttavia l'ordine di esecuzione dei job sulla stessa macchina può variare da macchina a macchina. Si supponga che

sulla macchina M_1 l'ordine di esecuzione dei job sia J_1, J_2, J_4, J_3,

sulla macchina M_2 l'ordine di esecuzione dei job sia J_2, J_1, J_3, J_4,

sulla macchina M_3 l'ordine di esecuzione dei job sia J_3, J_2, J_4, J_1.

Uno schedule ammissibile, avente makespan pari a 30, è il seguente:



Ovviamente non è detto che questo sia uno schedule ottimo. D'altra parte, essendo il problema un problema difficile, per trovare una soluzione ottima bisognerebbe provare tutte le soluzioni ammissibili.

Si suppone da ora che ogni job sia composto da esattamente *m* operazioni, una per ogni macchina. Si dimostra il seguente teorema che semplifica le cose nel caso in cui le macchine siano due.

Teor.: Per il problema $F,2/o/C_{\max}$ esiste una soluzione ottima tale che la permutazione degli *n* job sulla prima macchina coincide con quella sulla seconda.

Questo aiuta molto perché una volta scelto come ordinare i job sulla prima macchina si ha già l'ordinamento dei job sulla seconda.
Si consideri il problema $F,2/o/C_{\max}$; una delle proprietà più importanti è illustrata dal *teorema di Johnson*:

Teor.: Sia *S* uno schedule per il problema $F,2/o/C_{\max}$ nel quale il job J_k è eseguito prima del job J_j, con J_k e J_j job adiacenti. Se

$$\min\{p_{1j}, p_{2k}\} \leq \min\{p_{1k}, p_{2j}\},$$

allora lo schedule *S′* che si ottiene da *S* scambiando J_k e J_j sarà tale che

$$C_{\max,S} \geq C_{\max,S'}.$$

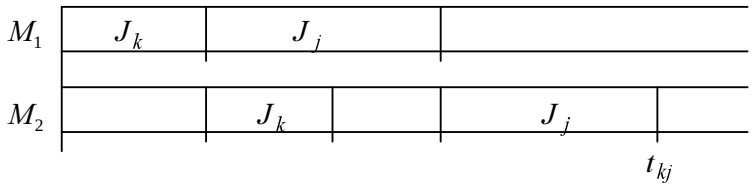
Dim.:

Si suppone che J_k e J_j siano i job che vengono eseguiti per primi e, per semplicità di notazione si dimostra il teorema solo in questo caso. Siano

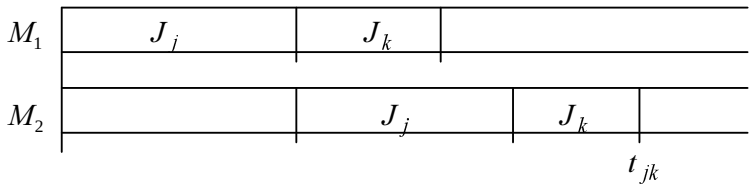
t_{kj} il tempo di completamento dei job J_k e J_j nello schedule *S*,

t_{jk} il tempo di completamento dei job J_j e J_k nello schedule *S′*.

Nello schedule *S*:



Nello schedule S':



È evidente dalla prima figura che

$$t_{kj} = p_{1k} + p_{2j} + \max\{p_{1j}, p_{2k}\}.$$

È evidente dalla seconda figura che

$$t_{jk} = p_{1j} + p_{2k} + \max\{p_{1k}, p_{2j}\}.$$

Per ipotesi

$$\min\{p_{1j}, p_{2k}\} \leq \min\{p_{1k}, p_{2j}\},$$

ovvero

$$\max\{-p_{1j}, -p_{2k}\} \geq \max\{-p_{1k}, -p_{2j}\}.$$

Aggiungendo ad entrambe i membri della disequazione la quantità

$$p_{1k} + p_{2k} + p_{1j} + p_{2j}$$

si ottiene

$$\max\{p_{1k} + p_{2k} + p_{2j}, p_{1k} + p_{1j} + p_{2j}\} \geq \max\{p_{2k} + p_{1j} + p_{2j}, p_{1k} + p_{2k} + p_{1j}\},$$

ovvero

$$p_{1k} + p_{2j} + \max\{p_{2k}, p_{1j}\} \geq p_{2k} + p_{1j} + \max\{p_{2j}, p_{1k}\}.$$

Quindi

$$t_{kj} \geq t_{jk},$$

da cui la tesi.■

Regola di Johnson

Per il problema $F,2/o/C_{\max}$ uno schedule è ottimo se per ogni coppia di lavori J_k e J_j, dove J_k è lavorato prima di J_j, sussiste la seguente relazione:

$$\min\{p_{1j}, p_{2k}\} \geq \min\{p_{1k}, p_{2j}\}.$$

È una condizione sufficiente ma non necessaria, cioè uno schedule che soddisfa queste condizioni è una soluzione ottima, ma non è detto che tutte le soluzioni ottime debbano avere soddisfare queste condizioni.

Algoritmo AJ per $F,2/o/C_{\max}$

Passo 1 Sia $N=\{J_1,J_2,...,J_n\}$ l'insieme dei job.

Costruire gli insiemi

$$X=\{J_j\in N: p_{1j}<p_{2j}\},$$

$$Y=\{J_j\in N: p_{1j}\geq p_{2j}\}.$$

Passo 2 Ordinare i job in X in ordine non decrescente di p_{1j};

ordinare i job in Y in ordine non crescente di p_{2j}.

Siano $\hat{X}$ e $\hat{Y}$ gli insiemi X e Y ordinati.

Passo 3 Scedulare i job appartenenti a $\hat{X}$ e successivamente i job appartenenti a $\hat{Y}$.

La complessità di tale algoritmo è $O(n\ln n)$, infatti l'operazione più costosa è quella di ordinare gli elementi degli insiemi X e Y.
Il seguente teorema dimostra che la soluzione individuata dall'algoritmo AJ è ottima.

Teor.: L'algoritmo AJ costruisce una soluzione che soddisfa la regola di Johnson e quindi determina una soluzione ottima.

Dim.:

i) Se $J_k,J_j\in\hat{X}$ (in quest'ordine),

$$p_{1k}\leq p_{1j}<p_{2j}$$

Quindi

$$\min\{p_{1k},p_{2j}\}=p_{1k}.$$

Poichè

$$p_{1k}<p_{2k}, \text{dal momento che } J_k\in\hat{X},$$

si ha

$$\min\{p_{1k},p_{2j}\}\leq\min\{p_{2k},p_{1j}\},$$

come richiesto.

ii) Se $J_k,J_j\in\hat{Y}$ (in quest'ordine),

$$p_{1k}\geq p_{2k} \quad \text{e} \quad p_{1j}\geq p_{2j}$$

e

$$p_{2k} \geq p_{2j}.$$

Di conseguenza

$$p_{1k} \geq p_{2j}.$$

Quindi

$$\min\{p_{1k}, p_{2j}\} = p_{2j}.$$

Poichè

$$p_{2j} \leq p_{2k} \quad \text{e} \quad p_{2j} \leq p_{1j},$$

allora

$$p_{2j} \leq \min\{p_{2k}, p_{1j}\}$$

e quindi

$$\min\{p_{1k}, p_{2j}\} \leq \min\{p_{2k}, p_{1j}\},$$

come richiesto.

iii) Se $J_k \in \hat{X}, J_j \in \hat{Y}$, allora

$$p_{1k} < p_{2k} \quad \text{e} \quad p_{2j} \leq p_{1j}.$$

Di conseguenza

$$\min\{p_{1k}, p_{2j}\} \leq \min\{p_{2k}, p_{1j}\},$$

come richiesto.■

Esempio

Sia n=10. Si supponga che i tempi di esecuzione delle operazioni dei 10 job sulle 2 macchine siano i seguenti:

	p_{1j}	p_{2j}
J_1	1	2
J_2	4	3
J_3	3	1
J_4	3	4
J_5	5	2
J_6	1	9
J_7	2	2
J_8	3	1
J_9	2	3
J_{10}	10	1

Sia X l'insieme dei job che hanno tempo di esecuzione sulla prima macchina inferiore al tempo di esecuzione sulla seconda macchina e sia Y l'insieme dei job che hanno tempo di esecuzione sulla prima macchina non inferiore al tempo di esecuzione sulla seconda macchina:

$$X = \{J_1, J_4, J_6, J_9\},$$

$$Y = \{J_2, J_3, J_5, J_7, J_8, J_{10}\}$$

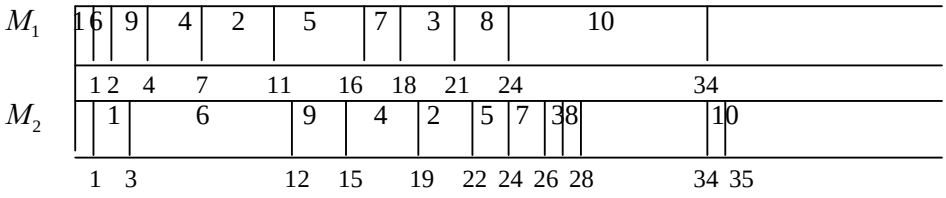
Si devono ora ordinare gli elementi di X per tempi di esecuzione sulla prima macchina crescenti:

$$\hat{X} = \{J_1, J_6, J_9, J_4\}$$

e si devono ordinare gli elementi di Y per tempi di esecuzione sulla seconda macchina decrescenti:

$$\hat{Y} = \{J_2, J_5, J_7, J_3, J_8, J_{10}\}.$$

Si tratta infine di schedulare su ciascuna macchina prima i job di $\hat{X}$ e poi i job di $\hat{Y}$.



Si osservi che nell'applicare l'algoritmo bisogna fare attenzione a non sovrapporre l'esecuzione dello stesso job su macchine diverse.

10. Open shop

Nei problemi di *open shop* ogni job è composto da più operazioni e non è prefissato un ordine di lavorazione delle operazioni sulle macchine. Per il problema $O,2/o/C_{\max}$ esiste un algoritmo risolutivo polinomiale con complessità $O(n)$.

Per comodità di notazione si pone

$$a_j = p_{1j}, b_j = p_{2j},\ j = 1,2,...n,$$

$$\bar{a} = \sum_{j=1}^{n} a_j,\ \bar{b} = \sum_{j=1}^{n} b_j.$$

Un ovvio *lower bound* per il valore della funzione obiettivo, e quindi per la lunghezza di uno schedule ammissibile, è

$$Lw = \max\{\bar{a}, \bar{b}, \max_{j=1,2,\ldots,n} (a_j + b_j)\};$$

infatti il valore ottimo della funzione obiettivo, e quindi del minimo del tempo massimo di completamento, è maggiore o uguale del tempo totale di esecuzione dei job sulla prima macchina, del tempo totale di esecuzione dei job sulla seconda macchina e del tempo totale di esecuzione del job che richiede lavorazione più lunga. Si vuole proporre un algoritmo che individua uno schedule ammissibile in corrispondenza del quale la funzione obiettivo assume valore *Lw*. È evidente che un tale schedule è anche ottimo, dal momento che non è possibile individuare un altro schedule con valore della funzione obiettivo inferiore a *Lw*.

Sia A l'insieme dei job aventi tempo di esecuzione sulla prima macchina non inferiore al tempo di esecuzione sulla seconda macchina, $A = \{J_j \,|\, a_j \geq b_j\}$, e sia B il suo complementare, cioè l'insieme dei job aventi tempo di esecuzione sulla prima macchina inferiore al tempo di esecuzione sulla seconda macchina $B = \{J_j \,|\, a_j < b_j\}$.

Si scelgano due job, J_r e J_l, appartenenti entrambi ad A oppure entrambi a B oppure uno ad A e uno a B tali che

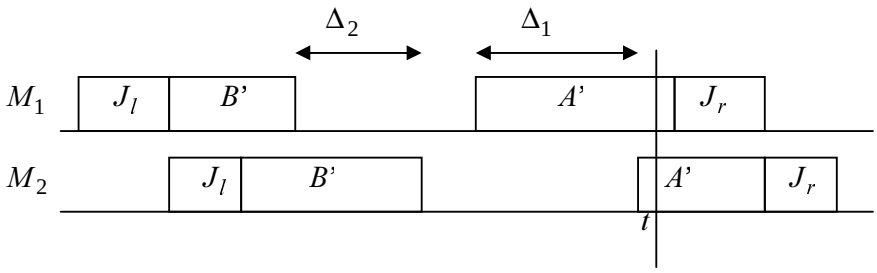
$$a_r \geq \max_{J_j \in A} b_j \quad \text{e} \quad b_l \geq \max_{J_j \in B} a_j$$

(tali job esistono per costruzione a meno che uno dei due insiemi A e B non sia vuoto; in tal caso è possibile applicare lo stesso procedimento che però risulta semplificato). Questo significa che il job J_r deve avere tempo di esecuzione sulla prima macchina non inferiore ai tempi di esecuzione sulla seconda macchina di tutti i job che stanno in A e il job J_l deve avere tempo di esecuzione sulla seconda macchina non inferiore ai tempi di esecuzione sulla prima macchina di tutti i job che stanno in B.

Siano poi

$$A' = A - \{J_r, J_l\} \quad \text{e} \quad B' = B - \{J_r, J_l\}.$$

E' possibile costruire schedule ammissibili per $B \cup \{J_l\}$ e $A \cup \{J_r\}$ separatamente come indicato nella seguente figura:



Si osserva che nei due schedule non ci sono periodi di inattività per le macchine.

Si dimostra che i due schedule parziali sono ammissibili, cioè che non c'è sovrapposizione di job tra le due macchine. Siano n_1 e n_2 la cardinalità di A' e B' rispettivamente e siano $A'=\{J_1^A, J_2^A, ..., J_{n_1}^A\}$ e $B'=\{J_1^B, J_2^B, ..., J_{n_2}^B\}$. Siano a_k^A, $k=1,\ldots,n_1$, e a_k^B, $k=1,\ldots,n_2$, i tempi di esecuzione sulla prima macchina dei job che stanno in A' e B' rispettivamente. Siano b_k^A, $k=1,\ldots,n_1$, e b_k^B, $k=1,\ldots,n_2$, i tempi di esecuzione sulla seconda macchina dei job che stanno in A' e B' rispettivamente. Risulta

$$\Delta_1 = \sum_{k=1}^{n_1} a_k^A + a_r - \sum_{k=1}^{n_1} b_k^A = \sum_{k=1}^{n_1} (a_k^A - b_k^A) + a_r$$

e

$$\Delta_2 = \sum_{k=1}^{n_2} b_k^B + b_l - \sum_{k=1}^{n_2} a_k^B = \sum_{k=1}^{n_2} (b_k^B - a_k^B) + b_l .$$

Sia J_i il job che è in lavorazione all'istante t sulla macchina M_1 e sia Δa_i^A il tempo trascorso dall'inizio della sua lavorazione su M_1. Si supponga per assurdo che ci sia sovrapposizione, cioè che all'istante t il job J_i sia in lavorazione anche sulla macchina M_2 e sia Δb_i^A il tempo trascorso dall'inizio della sua lavorazione su M_2. Allora

$$a_1^A + a_2^A + .. + a_{i-1}^A + \Delta a_i^A = \Delta_1 + b_1^A + b_2^A + ... + b_{i-1}^A + \Delta b_i^A$$

da cui

$$\sum_{k=1}^{i-1} (a_k^A - b_k^A) + (\Delta a_i^A - \Delta b_i^A) = \sum_{k=1}^{n_1} (a_k^A - b_k^A) + a_r$$

e quindi

$$\sum_{k=i}^{n_1}(a_k^A - b_k^A) + a_r = (\Delta a_i^A - \Delta b_i^A)\,. \qquad (*)$$

Ora sicuramente

$$\Delta a_i^A - \Delta b_i^A < a_i^A\text{, poiché } \Delta a_i^A \le a_i^A \text{ e } \Delta b_i^A > 0\,;$$

inoltre

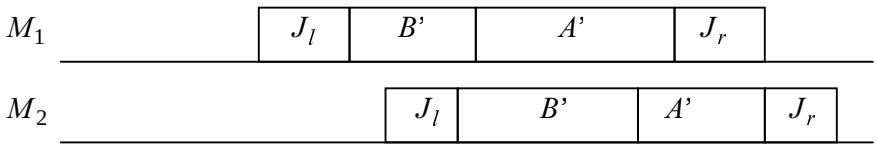
$$a_i^A - b_i^A + a_r \ge a_i^A\text{, poichè } a_r \ge b_i^A \text{ per definizione,}$$

e

$$\sum_{k=i+1}^{n_1}(a_k^A - b_k^A) \ge 0\text{, poiché } a_k^A \ge b_k^A \text{ per } k=i+1,\ldots,n_1\,.$$

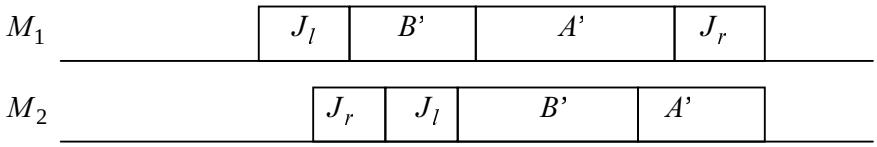
Quindi il secondo membro di (*) è minore di a_i^A, mentre il primo membro di (*) non inferiore a a_i^A. Si ottiene di conseguenza l'assurdo cercato.

Si tratta ora di mettere insieme i due schedule parziali che sono stati costruiti. Se $\bar{a} - a_l \ge \bar{b} - b_r$ è possibile combinare i due schedule spostando verso destra i job in $B' \cup \{J_l\}$ sulle macchine, come mostrato nella seguente figura:



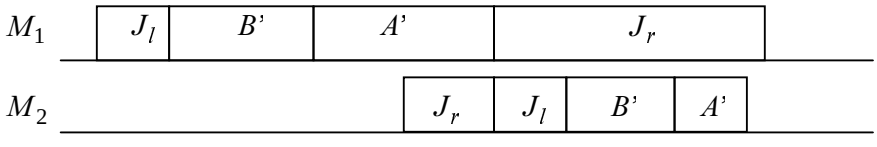
E' possibile a questo punto anticipare J_r su M_2 e metterlo in prima posizione.

Se $a_r \le \bar{b} - b_r$ si ottiene lo schedule



avente lunghezza pari a $\max\{\bar{a}, \bar{b}\}$.

Se invece $a_r > \bar{b} - b_r$ lo schedule risultante è



Esso ha lunghezza pari a $\max\{\bar{a}, a_r + b_r\}$.

In entrambe i casi si raggiunge il *lower bound,* costruendo così schedule ottimi.

Si osservi che nel caso $\bar{a} - a_l < \bar{b} - b_r$ è possibile combinare i due schedule spostando verso sinistra i job in $A \cup \{J_r\}$ sulle macchine e poi procedere nello stesso modo.

Algoritmo

Passo 1: Costruire gli insiemi *A* e *B*.

Passo 2: Scegliere i job J_r e J_l tali che $a_r \geq \max_{J_j \in A} b_j$ e $b_l \geq \max_{J_j \in B} a_j$. Considerare

$$A' = A - \{J_r, J_l\} \text{ e } B' = B - \{J_r, J_l\}.$$

Passo 3: Costruire i due schedule parziali J_l, B' e A' e J_r sulle due macchine, facendo attenzione a non sovrapporre la lavorazione di J_r e J_l sulle macchine.

Passo 4: Se $\bar{a} - a_l \geq \bar{b} - b_r$ spostare verso destra i job in $B \cup \{J_l\}$ sulle macchine e riposizionare J_r sulla seconda macchina, anticipandolo e facendo attenzione alle sovrapposizioni. Se $\bar{a} - a_l < \bar{b} - b_r$ spostare verso sinistra i job in $A \cup \{J_r\}$ sulle macchine e riposizionare J_r sulla seconda macchina, anticipandolo e facendo attenzione alle sovrapposizioni.

Esempio

Sia *n*=10 e siano i seguenti i tempi di esecuzione dei job sulle due macchine:

	a_j	b_j
J_1	5	7
J_2	3	4
J_3	9	3
J_4	2	5
J_5	7	3
J_6	8	1
J_7	2	1
J_8	3	3
J_9	4	2
J_{10}	2	4

L'insieme A contiene tutti i job che hanno tempo di esecuzione sulla prima macchina non inferiore al tempo di esecuzione sulla seconda macchina:

$$A = \{J_3, J_5, J_6, J_7, J_8, J_9\}.$$

L'insieme B contiene tutti i job che hanno tempo di esecuzione sulla prima macchina inferiore al tempo di esecuzione sulla seconda macchina:

$$B = \{J_1, J_2, J_4, J_{10}\}.$$

Si devono ora scegliere un job J_r tale che $a_r \geq \max_{J_j \in A} b_j$ e un job J_l tale che $b_l \geq \max_{J_j \in B} a_j$:

$$a_r \geq \max\{3,1,2\}$$

e

$$b_l \geq \max\{5,3,2\}.$$

Siano allora, per esempio, $J_r = J_1$ e $J_l = J_4$.

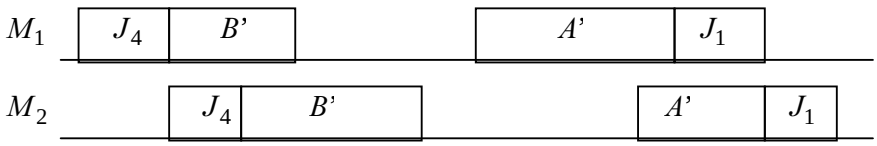
Si devono costruire gli insiemi $A' = A - \{J_r, J_l\}$ e $B' = B - \{J_r, J_l\}$:

$$A' = A - \{J_1, J_4\} = A$$

e

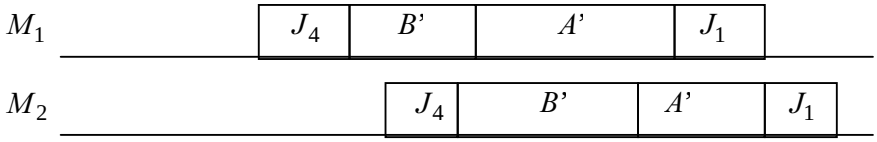
$$B' = B - \{J_1, J_4\} = \{J_2, J_{10}\}.$$

I due schedule parziali sono i seguenti:

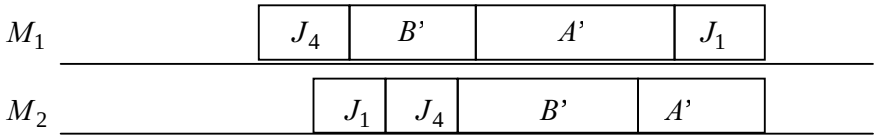


Si osservi che $\sum_{J_j \in B'} a_j < \sum_{J_j \in B'} b_j$ e $\sum_{J_j \in A'} a_j > \sum_{J_j \in A'} b_j$.

Poiché $\bar{a} - a_4 \geq \bar{b} - b_1$, infatti $\bar{a} - a_4 = 43$ e $\bar{b} - b_1 = 26$, si devono spostare verso destra i job in $B' \cup \{J_4\}$ sulle macchine e riposizionare J_1 sulla seconda macchina, anticipandolo e facendo attenzione alle sovrapposizioni.



Poiché $a_r \leq \bar{b} - b_r$, lo schedule ottimo è



avente lunghezza uguale a $\max\{\bar{a}, \bar{b}\} = \max\{45,33\} = 45$.

Esercizio

Sia *n*=10 e siano i seguenti i tempi di esecuzione dei job sulle due macchine:

	a_j	b_j
J_1	5	6
J_2	3	4
J_3	3	3
J_4	2	5
J_5	3	3
J_6	1	1
J_7	2	1
J_8	3	3
J_9	4	2
J_{10}	2	4

Si risolva il problema $O,2/o/C_{\max}$ utilizzando l'algoritmo descritto in precedenza.

11. Metodi euristici

Come è stato detto, nel caso in cui il problema considerato sia *NP-hard*, ci si accontenta di individuare una soluzione vicina alla soluzione ottima (soluzione

approssimata) ma in tempi ragionevoli. Algoritmi che individuano una soluzione approssimata si dicono algoritmi *approssimati* o *euristici*.

Si descrive in seguito l'algoritmo di Giffler e Thompson che, dato un problema di *job shop*, individua uno schedule sub-ottimo pianificando un'operazione alla volta. È un'euristica della quale si riesce a dire molto poco, nel senso che non si riesce a limitare l'errore. Il procedimento consiste nello schedulare un'operazione alla volta. Essendo *n* i lavori ed *m* le macchine, l'euristiche prevede al più di $m \cdot n$ iterazioni.

All'iterazione *t* siano

P_t lo schedule parziale relativo alle *t-1* operazioni pianificate,

S_t l'insieme delle operazioni che possono essere pianificate all'iterazione *t*,

$S_t = \{o_k,\ k = 1,..., K\}$.

σ_k il primo istante in cui l'operazione o_k in S_t può essere eseguita,

ϕ_k il primo istante in cui l'operazione o_k in S_t può essere conclusa, cioè

$\phi_k = \sigma_k + p_k$ se p_k è il tempodi esecuzione dell'operazione o_k.

Algoritmo di Giffer e Thompson

Passo 1: Siano *t=1* e $P_1 = \varnothing$. S_1 è l'insieme delle operazioni che non hanno predecessori, cioè l'insieme delle operazioni che sono prime nel loro job.

Passo 2: Siano $\phi^* = \min_{o_k \in S_k} \{\phi_k\}$ e M^* la macchina su cui viene individuato ϕ^*. A parità di condizioni si sceglie arbitrariamente.

Passo 3: Scegliere un'operazione o_j in S_t tale che

(1) essa venga lavorata su M^*,

(2) $\sigma_j < \phi^*$.

Passo 4 Passare alla fase successiva

(1)aggiungendo o_j a P_t, creando così P_{t+1};

(2)cancellando o_j da S_t, creando così S_{t+1}, costituita da tutte le operazioni di S_t a cui si toglie o_j e si aggiunge l'operazione che segue o_j nel suo job;

(3) incrementando *t* di 1.

Passo 5: Se $t<mn$, cioè se ci sono ancora operazioni da pianificare, tornare al passo 2, altrimenti STOP.

Talvolta si utilizzano al Passo 3 criteri di scelta più selettivi. I più studiati sono i seguenti:

SPT (Shortest Processing Time) Si tratta di scegliere un'operazione con il tempo di esecuzione più breve.

FCFS (First Come-First Served) Si sceglie un'operazione che è rimasta in S_i per il maggior numero di fasi.

MWKR (Most Work Remaining) Si sceglie un'operazione che appartiene al job con tempo complessivo d'esecuzione delle operazioni rimanenti più grande.

LWKR (Least Work Remaining) Si sceglie un'operazione che appartiene al job con tempo complessivo d'esecuzione delle operazioni rimanenti più piccolo.

MOPNR (Most Operations Remaining) Si sceglie un'operazione che appartiene al job che contiene il maggior numero di operazioni che devono ancora essere processate.

RANDOM (Random) Si sceglie casualmente un'operazione.