Introduzione ai problemi di Scheduling

A. Agnetis*

December 12, 2000

1 Introduzione

Con il termine *scheduling* si intende una vasta classe di problemi, molto diversi tra loro per complessità e struttura. Molti autori hanno tentato una sistematizzazione e una sintesi dell'ampio corpo metodologico che si è andato formando negli ultimi 30 anni relativamente a tali problemi. Questo sforzo ha prodotto molti risultati interessanti, che consentono oggi di classificare molti modelli in modo organico e di unificare alcuni approcci algoritmici. Tuttavia, a differenza di quanto accade per altre aree dell'ottimizzazione combinatoria, a tutt'oggi per i problemi di scheduling più difficili non è possibile indicare un unico approccio nettamente preferibile per la loro soluzione, ma di volta in volta può essere più appropriato utilizzare euristiche, algoritmi di enumerazione, algoritmi approssimati. In questi appunti vedremo una rassegna di alcuni risultati fondamentali della teoria dello scheduling.

Un possibile modo di definire i problemi di scheduling è quello di fare riferimento all'ambito applicativo in cui questi problemi sono nati, e possiamo allora dire che problemi di scheduling sono tutti quei problemi decisionali in cui riveste importanza il fattore tempo, visto come risorsa (scarsa) da allocare in modo ottimo a determinate attività.

1.1 Esempi di problemi di scheduling

Questi sono solo alcuni dei possibili esempi di problemi che ricadono in questa categoria.

*Dipartimento di Ingegneria dell'Informazione - Università di Siena

- Un'azienda produce sacchetti di carta per cemento, cibo per cani, carbone etc. La materia prima è costituita da rotoli di carta. Il processo produttivo consiste di tre stadi: nel primo, viene stampato il logo sui sacchetti, nel secondo i due lati del sacchetto vengono incollati, e infine nel terzo una delle due estremità viene cucita. A ogni stadio, l'operazione è effettuata da un operaio. Ogni ordine di produzione consiste di un certo numero di sacchetti di un certo tipo che devono essere consegnati al cliente entro una certa data. Il costo legato a un eventuale ritardo rispetto alla data di consegna dipende dall'entità del ritardo e dall'importanza dell'ordine (ovvero del cliente). Conoscendo i tempi richiesti dalle varie operazioni per ciascun ordine, un obiettivo della pianificazione è quello di organizzare le lavorazioni in modo tale da minimizzare queste penali.

- Nell'industria meccanica, i centri di lavorazione devono effettuare lavorazioni (taglio, fresatura, tornitura) su vari pezzi che vengono montati sui centri stessi. Diverse operazioni richiedono tempi diversi, e/o diversi tipi di utensile, il che può comportare un certo tempo di riconfigurazione (set-up) da parte delle macchine stesse. Il problema può consistere nel determinare l'ordinamento dei pezzi in modo da terminare tutte le lavorazioni prima possibile.

- Uno dei compiti di un sistema operativo è quello di disciplinare l'accesso alla CPU dei diversi programmi di calcolo. Ciascun programma può avere una certa priorità. L'obiettivo tipico del sistema operativo è allora quello di gestire l'insieme dei programmi in modo tale da minimizzare il tempo complessivo di attesa dei programmi, tenendo conto della loro importanza relativa (converrà privilegiare i programmi a priorità più elevata). In questa particolare applicazione, il sistem aoperativo potrà eventualmente decidere di interrompere certi programmi per consentire il completamento di altri. Questa modalità operativa prende il nome di *preemption*.

- In un'officina di carrozzeria, vi sono quattro stazioni, dedicate rispettivamente a messa in forma, ribattitura, verniciatura, essiccatura a forno. In ciascuna stazione è attivo un operaio, che può lavorare su una sola autovettura alla volta. In una data giornata di lavoro, devono essere riparate un certo numero di autovetture sinistrate, ciascuna delle quali richiede il servizio da parte di alcune stazioni, in un dato ordine (ad esempio non si può riverniciare la carrozzeria prima di avere aggiustato le parti danneggiate). Il problema consiste nel gestire le varie operazioni in modo da terminare tutte le lavorazioni nel minor tempo possibile.

1.2 Elementi di un problema di scheduling

Negli esempi visti, il problema di scheduling ruota attorno alle modalità di assegnamento di una risorsa (macchina, centro di lavorazione, CPU, stazione) ad un'attività che deve essere effettuata (stampa di un sacchetto di carta, taglio di un pezzo meccanico, esecuzione di un programma di calcolo, riparazione di un'autovettura). Nei problemi di scheduling, risorse e attività vengono indicate con i termini *macchina* e *task*, mentre il termine *job* si riferisce in genere a insiemi di task tecnologicamente legati tra loro (ad esempio, i tre task necessari a produrre uno stesso sacchetto di carta formano un job). Nel seguito indicheremo con m e n il numero di macchine e di job rispettivamente.

Varie informazioni possono essere associate a un job:

- *Tempo di processamento* o *durata* p_{ij}. Questo è il tempo (deterministico) che il job j richiede alla macchina i per essere eseguito. In genere esso corrisponde ad uno dei task (l'i-esimo) che compongono il job j.

- *Tempo di rilascio (release date)* r_j. Indica l'istante di tempo (rispetto a un tempo iniziale 0) prima del quale non è possibile iniziare l'esecuzione del job j. Ad esempio, se le materie prime per effettuare il job j arrivano il giorno 3 (a partire da oggi), prima di allora il job j non può iniziare.

- *Tempo di consegna (due date)* d_j. Indica l'istante di tempo (rispetto a un tempo iniziale 0) entro il quale l'esecuzione del job j dovrebbe essere terminata. In genere, la violazione di una due date comporta dei costi (penale, perdita di fiducia da parte del cliente, etc).

- *Peso* w_j. Rappresenta l'importanza relativa del job j rispetto agli altri. Ad esempio, questo peso rappresenta il costo di tenere nel sistema il job (ad es., un costo di immagazzinamento).

Oltre alle caratteristiche relative ai job, vi sono quelle relative al sistema. Vi è una grande varietà di architetture del sistema produttivo o di servizio; noi ci limiteremo ai casi più semplici.

- *Macchina singola.* Questo è evidentemente il caso più semplice, in cui i job richiedono tutti la stessa risorsa per essere eseguiti. In questo caso in genere ciascun job consiste di un singolo task (e dunque si può parlare indifferentemente di job o task).

- *Flow shop.* In questo caso il sistema consiste di m macchine disposte in serie, e ciascun job deve essere eseguito da ciascuna delle m macchine successivamente,

ossia prima deve visitare la macchina 1, poi la macchina 2, ..., infine la macchina m. Spesso si assume che ciascuna macchina abbia un buffer di tipo FIFO, ossia l'ordine in cui i job visitano ciascuna macchina è lo stesso per tutte le macchine. In altre parole, i job non possono sorpassarsi. In questo caso si parla di *permutation flow shop*.

- *Job shop*. Anche qui vi sono m macchine, ma ciascun job ha un proprio ordine con cui visitarle. Si noti che il job shop è una generalizzazione del flow shop, ovvero il flow shop può essere visto come un job shop in cui ciascun job deve visitare le macchine nell'ordine 1, 2, ..., m.

Oltre a job e macchine, vi sono particolari ulteriori specifiche che contribuiscono a definire esattamente un problema di scheduling. Tali specifiche possono comprendere:

- *Tempi di set-up*. Questa caratteristica è presente soprattutto nei problemi singola macchina, e indica che se si vuole fare seguire il job k al job j, allora tra il completamento del job j e l'inizio di k è necessario riconfigurare la macchina, e questo richiede un tempo s_{jk}. In questo caso si assume che il tempo di set-up tra j e k sia comunque indipendente dai job precedenti j e seguenti k.

- *Preemption*. In certi casi è consentito interrompere un job per permettere l'esecuzione di uno più urgente. Il problema in questo caso si dice *preemptive*.

- *Vincoli di precedenza*. In molti casi esistono vincoli di precedenza tra task di un job (come accade nei casi del flow shop o del job shop), o tra diversi job. I vincoli di precedenza sono espressi per mezzo di un grafo orientato (ovviamente aciclico), in cui i nodi corrispondono ai job e gli archi esprimono vincoli di precedenza, ossia un arco da j a k sta a indicare che il job k non può iniziare prima che sia terminato j.

- *blocking e no-wait*. In un flow shop, i job in attesa di essere eseguiti da una macchina sono in genere ospitati in un buffer. Se a un dato istante il buffer della macchina i è pieno, un job terminato sulla macchina $i-1$ non può trovare posto nel buffer della macchina i, ed è quindi costretto a tenere bloccata la macchina $i-1$, che non può quindi iniziare un nuovo task fino a che non verrà liberata dal job. Questo fenomeno si chiama *blocking*. Se in particolare *non c'è buffer* in ingresso alla macchina i, il pezzo terrà bloccata la macchina $i-1$ fino a che la macchina i è occupata. Ancora più restrittiva è la situazione *no-wait*, in cui ai job non è permesso nemmeno di attendere su una macchina, e invece occorre garantire che all'istante di completamento di un task su una macchina, la macchina successiva sia già libera per processare il job.

1.3 Definizione di schedule, notazioni

Una soluzione di un problema di scheduling prende il nome di *schedule*. In termini generali, uno *schedule* è una descrizione completa dell'utilizzo temporale delle macchine da parte dei job che devono essere eseguiti. Nel caso in cui i task che compongono i job non possano essere interrotti, uno schedule è completamente specificato da un assegnamento di istanti di inizio a tutti i task che devono essere eseguiti, altrimenti occorre specificare l'istante di inizio di ciascuna delle parti in cui viene suddivisa l'esecuzione un task. Spesso si distingue il concetto di *sequenza* da quello di schedule. La sequenza specifica solo l'ordine in cui i job devono essere eseguiti da ciascuna macchina, lo schedule ne specifica anche gli istanti d'inizio. Molto spesso però, data la sequenza è immediato risalire allo schedule. Nel seguito, dato uno schedule S, indicheremo con $S(j)$ l'istante di inizio di un job j.

Ovviamente, saremo interessati a schedule *ammissibili*, cioè che rispettino tutti i vincoli impliciti in qualsiasi problema di scheduling, quali ad esempio che una stessa macchina non può eseguire due job contemporaneamente, che uno stesso job non può essere eseguito da due macchine contemporaneamente, che un job non può essere interrotto (tranne che nei problemi preemptive), che eventuali precedenze devono essere rispettate. Ad esempio, si consideri una singola macchina, e tre job, con tempi di processamento $p_1 = 10$, $p_2 = 4$, $p_3 = 5$. Una possibile sequenza è $\sigma = (3, 1, 2)$, ovvero, supponendo che ciascun job inizi non appena il precedente è concluso, si ha lo schedule S definito da $S(1) = 5$, $S(2) = 15$, $S(3) = 0$. Si noti che se $S(3) = 0$, non può aversi ad esempio $S(1) = 4$, perche' all'istante 4 la macchina è ancora impegnata a processare il job 3.

Come visto nei precedenti esempi, gli obiettivi di un problema di scheduling possono essere diversi. Per esprimerli formalmente, occorre introdurre alcune funzioni associate ai vari job in uno schedule ammissibile.

- *Tempo di completamento* C_j. È il tempo a cui l'ultimo task del job j (e quindi l'intero job j) termina. Se non sono ammesse interruzioni, C_j è dato dalla somma dell'istante di inizio dell'ultimo task di quel job e del tempo di processamento dell'ultimo task.

- *Lateness* L_j. È la differenza tra il tempo di completamento e la data di consegna del job j. Si noti che se è positiva, la lateness indica un ritardo, se negativa un anticipo rispetto alla due date. Si ha dunque: $L_j = C_j - d_j$.

- *Tardiness* T_j. Coincide con la lateness quando questa è positiva, ed è zero altrimenti, ossia $T_j = \max\{0, C_j - d_j\}$.

Diverse funzioni obiettivo possono costruirsi con queste grandezze. Quelle di cui ci occuperemo sono le seguenti:

- *Massimo tempo di completamento* o *makespan* C_{max}. Dato uno schedule S, è definito come $\max\{C_1, \ldots, C_n\}$, ossia è il tempo di completamento del job che termina per ultimo. Esso rappresenta la misura (rispetto al tempo 0) del tempo necessario a completare tutte le attività.

- *Massima lateness* L_{max}. Dato uno schedule S, è definita come $\max\{L_1, \ldots, L_n\}$, ossia è il ritardo del job che termina in maggior ritardo rispetto alla propria data di consegna. (Si noti che potrebbe anche essere negativo, e in tal caso rappresenta l'anticipo del job che termina con minore anticipo rispetto alla propria data di consegna).

- *Massima tardiness* T_{max}. E' definita come $\max\{0, L_{max}\}$.

- *Somma pesata dei tempi di completamento.* Dato uno schedule S, è definita come $\sum_{j=1}^{n} w_j C_j$. Nel caso in cui i pesi siano tutti uguali tra loro, è il tempo complessivamente trascorso nel sistema dai job (dall'istante 0 al loro completamento), ovvero una misura del livello di servizio offerto dal sistema. I pesi introducono inoltre un elemento di priorità.

Nel seguito utilizzeremo la classica notazione di problemi di scheduling a tre campi $\alpha|\beta|\gamma$ in cui α identifica il sistema e il numero di macchine, β le (eventuali) caratteristiche particolari e γ la funzione obiettivo. Per quanto concerne il sistema, indicheremo le tre possibili strutture (singola macchina, flow shop, job shop) con 1, P e J rispettivamente. Ad esempio, il problema di minimizzare il massimo ritardo di un job su una macchina, con release dates, verrà indicato con $1|r_j|L_{max}$. Minimizzare il makespan in un flow shop con due macchine, senza ulteriori specifiche, verrà indicato con $F2||C_{max}$. Con riferimento agli esempi della sezione 1.1, i problemi descritti possono essere indicati brevemente come

- $F3||\sum_{j=1}^{n} w_j T_j$
- $1|s_{jk}|C_{max}$
- $1|preemption|\sum_{j=1}^{n} w_j C_j$
- $J4||C_{max}$

2 Macchina singola

Iniziamo l'analisi dai problemi più semplici, vale a dire quelli in cui n job devono essere eseguiti da una singola macchina. Ogni job j è costituito da un solo task, di durata p_j.

Dunque in questo caso scompare la distinzione tra task e job. Anzitutto osserviamo che, nei problemi in cui non vi siano release dates, per qualunque funzione obiettivo tra quelle introdotte, possiamo limitarci a considerare schedule in cui la macchina è sempre attiva, dall'istante 0 all'istante finale $\sum_{j=1}^{n} p_j$. Per questo motivo, data una sequenza σ dei job, uno schedule S si ottiene semplicemente eseguendo i job nell'ordine della sequenza, senza inserire tempi di attesa tra la fine di un job e l'inizio del successivo. Questo ci consente, nei problemi singola macchina, di identificare sequenze e schedule.

2.1 Somma pesata dei tempi di completamento

Il primo problema che consideriamo è $1||\sum_{j=1}^{n} w_j C_j$. A ogni job sono associate due quantità, il tempo di processamento p_j e il peso w_j. Ordiniamo i job secondo valori non decrescenti del rapporto p_j/w_j. Questa regola di ordinamento prende il nome di WSPT (Weighted Shortest Processing Time).

TEOREMA 1 *Lo schedule che si ottiene processando i job secondo l'ordine WSPT è ottimo per il problema* $1||\sum_{j=1}^{n} w_j C_j$.

Dim.- Si consideri uno schedule S che non segue la regola WSPT, e supponiamo sia ottimo. Ci saranno allora almeno due job, j e k tale che k segue immediatamente j in S, ma $p_j/w_j > p_k/w_k$. Se poniamo $S(j) = t$, avremo dunque che nello schedule S, $C_j = t+p_j$ e $C_k = t + p_j + p_k$. Pertanto il contributo alla funzione obiettivo di questi due job nello schedule S è dato complessivamente da

$$w_j(t + p_j) + w_k(t + p_j + p_k) \tag{1}$$

Se ora scambiamo di posto j e k, otteniamo un nuovo schedule S'. Per tutti gli altri job, non è cambiato niente, e dunque il loro contributo alla funzione obiettivo è uguale al precedente. Se invece indichiamo con C'_j e C'_k i nuovi tempi di completamento dei job j e k, avremo $C'_j = t + p_k + p_j$ e $C'_k = t + p_k$. In S' il contributo alla funzione obiettivo di questi due job è quindi dato da

$$w_j(t + p_k + p_j) + w_k(t + p_k) \tag{2}$$

Confrontando allora i contributi di j e k alla funzione obiettivo nei due casi, si ha che (1) è strettamente maggiore di (2), dal momento che

$$w_k p_j > w_j p_k$$

Dunque, scambiando j e k la funzione obiettivo diminuisce e questo contraddice il fatto che S fosse uno schedule ottimo. $\square$

Si noti che nel caso in cui $w_j = 1$ per tutti i job j (caso non pesato), la regola consiste nel processare i job in ordine non decrescente di durata (regola SPT).

In definitiva, il problema $1||\sum_{j=1}^{n} w_j C_j$ può essere risolto in tempo $O(n \log n)$ in quanto è sufficiente ordinare i job e poi scorrere la lista una sola volta.

2.2 Algoritmo di Lawler

In questo paragrafo analizziamo il problema di minimizzare una funzione obiettivo abbastanza generale:

$$\max_j \{\gamma_j(C_j)\} \tag{3}$$

ove $\gamma_j(C_j)$ è una arbitraria funzione non decrescente di C_j (funzioni di questo tipo sono chiamate funzioni *regolari*). Dunque, a ogni job è associata una diversa funzione, e l'obiettivo è la minimizzazione del valore *più alto* tra quelli assunti dalle varie funzioni per i rispettivi job. Si noti che la (3) racchiude molte funzioni obiettivo. Ad esempio, la funzione L_{max} è un caso particolare di (3), che si ha ponendo, per ogni job j:

$$\gamma_j(C_j) = C_j - d_j$$

che è chiaramente una funzione regolare. Analogo discorso per T_{max}. Si noti invece che una funzione obiettivo come $\sum_{j=1}^{n} w_j C_j$ non può essere ottenuta come caso particolare di (3), in quanto vi compare la somma, e non l'operazione di max.

Inoltre, supponiamo che tra i job sussista una generica relazione di precedenza, espressa da un grafo aciclico G_P.

Vogliamo dunque analizzare il problema $1|prec|\max_j\{\gamma_j(C_j)\}$. Tale problema può essere risolto in modo efficiente, qualunque forma abbiano i vincoli di precedenza. Poiche' non vi è alcun interesse a tenere la macchina ferma, qualunque sia lo schedule S, l'ultimo job verrà completato all'istante $\sum_{j=1}^{n} p_j$, che indichiamo con P. Il problema può essere risolto costruendo lo schedule a partire dal fondo; ossia iniziamo chiedendoci chi sarà l'ultimo job a essere eseguito. Si considerino allora i nodi (job) del grafo G_P privi di archi uscenti (Poiche' G_P è aciclico, è facile mostrare che ne esiste sicuramente almeno uno). Evidentemente l'ultimo job andrà scelto tra questi. Dal momento che l'istante di completamento, P, è noto, possiamo calcolare il valore che in P assume ciascuna delle funzioni associate a questi job. Il job j^* per cui si ha il minimo dei valori $\gamma_j(P)$ verrà posto in ultima posizione. Cancellando il job j, si avrà un nuovo insieme di job privi di archi

```
Algoritmo di Lawler
{
        G := G_P; Ĉ := P;
        for  c = n to 1
        {
            J̄ := {j|δ⁺(j) = ∅ in G};
            sia j* tale che γ_{j*}(Ĉ) = min_{j∈J̄}{γ_j(Ĉ)}
            σ(c) := j*;
            G := G − {j*};
            Ĉ := Ĉ − p_{j*}
        }
        return the schedule σ;
}
```

Figura 1: Algoritmo di Lawler.

uscenti, e dunque candidati a occupare la penultima posizione in S. A questo punto, il penultimo job, chiunque esso sia, terminerà all'istante $P - p_{j^*}$, e di nuovo potremo selezionarlo prendendo quello la cui funzione, calcolata per $P - p_{j^*}$, ha il valore più basso degli altri, e cosi' via. Riassumendo, abbiamo l'algoritmo in Figura 1, in cui a ogni passo $\bar{J}$ indica l'insieme dei job privi di successori nel grafo delle precedenze.

TEOREMA 2 *Il problema* $1|prec|\max_j\{\gamma_j(C_j)\}$ *è risolto all'ottimo dall'algoritmo di Lawler in tempo* $O(n^2)$.

Dim.- Al primo passo dell'algoritmo, indichiamo con j^* il job prescelto per l'ultima posizione nello schedule, ossia $\gamma_{j^*}(P) = \min_j\{\gamma_j(P)\}$. Vogliamo mostrare che anche nella sequenza ottima j^* è in ultima posizione. Consideriamo una sequenza arbitraria S, in cui il job in ultima posizione è il job $k \neq j^*$. La struttura di S è quindi

$$S = (A, j^*, B, k)$$

dove A e B stanno a indicare due sottosequenze, contenenti gli altri $n-2$ job. Sia poi S' la sequenza ottenuta da S spostando j^* in ultima posizione, ossia:

$$S' = (A, B, k, j^*)$$

se S rispettava tutti i vincoli di precedenza, di certo anche S' li rispetta, dal momento che, come supponiamo, j^* può essere posto in ultima posizione. Ora, osserviamo che passando da S a S', per i job in A non è cambiato nulla. Il tempo di completamento dei job in B

e del job k è diminuito di p_{j^*}, e dunque il valore della loro funzione associata non può essere peggiorato. Rimane il solo job j^*, il cui tempo di completamento è aumentato. Tuttavia, per ipotesi $\gamma_{j^*}(P) \leq \gamma_k(P)$, e dunque anche se il costo per j^* aumenta passando da S a S', comunque non supera il costo per k in S, ossia in definitiva $\max_j\{\gamma_j(C_j)\}$ non aumenta passando da S a S'. Dunque, esiste senz'altro una sequenza ottima S' con j^* in ultima posizione. Tutta la discussione può ripetersi sull'insieme dei job che si ottiene cancellando j^*, e cosi' via.

Per quanto riguarda la complessità, osserviamo che l'algoritmo consiste di n passi, a ognuno dei quali devono essere confrontati tra loro $O(n)$ valori. Supponendo che il calcolo dei valori delle funzioni $\gamma_j(\cdot)$ richieda un tempo costante, segue la tesi. □

È interessante osservare che, applicato al problema $1||L_{max}$, l'algoritmo di Lawler può essere espresso in modo più semplice. Infatti, in quel caso il job per cui si ha il minimo valore della funzione $\gamma_j(\cdot)$ sarà quello avente la due date più grande, tra tutti i job non ancora sequenziati. In altre parole, l'algoritmo prescrive semplicemente di sequenziare i job in ordine *non decrescente* di data di consegna. Questa regola prende il nome di EDD (Earliest Due Date), e può evidentemente essere implementata in $O(n \log n)$.

2.3 Minimizzazione del massimo ritardo, con release dates

La regola EDD non è più valida se i job hanno una release date non nulla. In questo caso, il problema diviene significativamente più difficile, anche nel caso in cui - come supporremo in questo capitolo - non vi siano vincoli di precedenza.

2.3.1 Il problema $1|r_j, preempt|L_{max}$

Analizziamo dapprima il caso più semplice in cui sia consentita l'interruzione dei job. Nel seguito supporremo che tutti i tempi di processamento e le release date siano interi non negativi. È allora facile verificare che si ha interesse a interrompere un job solamente a istanti di tempo interi.

In questo caso, una semplice (e piuttosto intuitiva) modifica della regola EDD consente di calcolare la soluzione ottima. Dato un generico istante t, possiamo considerare l'insieme dei job *rilasciati* $R(t)$ definito come:

$$R(t) = \{j \mid r_j \leq t\}$$

è evidente che il job in esecuzione a un generico istante t apparterrà necessariamente all'insieme $R(t)$.

Poiche' sono consentite le interruzioni, un job può essere iniziato, poi interrotto, poi ripreso e cosi' via molte volte fino al suo completamento. La regola operativa PEDD (Preemptive Earliest Due Date) prescrive di avere in esecuzione, a ogni istante t, il job avente due date più bassa tra i job non ancora terminati dell'insieme $R(t)$. Nel seguito, data una soluzione S, indicheremo con $L_{max}(S)$ il corrispondente valore di L_{max}.

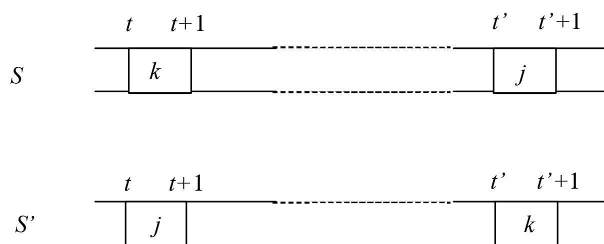

Figura 2: Schedule S e S' nella dimostrazione del Teorema 3.

TEOREMA 3 *Il problema* $1|r_j, preempt|L_{max}$ *è risolto all'ottimo dalla regola PEDD.*

Dim.- Consideriamo una soluzione S che non rispetta la regola PEDD, ossia supponiamo che nella finestra temporale che va da t a $t+1$ (t intero), sia in esecuzione un job $k \in R(t)$, mentre esiste un altro job $j \in R(t)$, non ancora terminato, tale che $d_j < d_k$. Poiche' j non è ancora terminato, esisterà almeno un intervallo temporale $[t', t'+1]$, con $t' > t$, in cui è in esecuzione j. Chiamiamo S' la soluzione ottenuta scambiando i job attivi nei due intervalli (vedi Figura 2). Chiaramente, gli unici due job per i quali cambia qualcosa sono j e k, mentre per tutti gli altri la situazione rimane identica. Indicando con C_j e C_k i tempi di completamento nello schedule S, vogliamo mostrare che $L_{max}(S') \leq L_{max}(S)$.

(*i*) $C_j > t' + 1$ e $C_k > t' + 1$. In tal caso, Poiche' ambedue i job finiscono dopo $t' + 1$, il fatto di scambiare le due porzioni di job non ha alcun effetto sui tempi di completamento dei due job, e dunque $L_{max}(S') = L_{max}(S)$.

(*ii*) $C_j = t' + 1$ e $C_k > t' + 1$. Poiche' il job k termina oltre $t' + 1$, il suo tempo di completamento rimane inalterato. Invece il tempo di completamento di j diminuisce, e dunque $L_{max}(S') \leq L_{max}(S)$.

(*iii*) $C_j \geq t' + 1$ e $C_k < t' + 1$. In questo caso, passando da S a S', il tempo di completamento di k aumenta, divenendo pari a $t' + 1$. La lateness di k in S' è dunque $t' + 1 - d_k$. Invece la lateness di j di certo non aumenta. Considerando che in S $L_j = C_j - d_j \geq t' + 1 - d_j$, e che $d_j < d_k$, si ha che la lateness di k in S' è comunque non superiore a quella di j in S, e dunque anche in questo caso $L_{max}(S') \leq L_{max}(S)$.

□

TEOREMA 4 *Il problema* $1|r_j, preempt|L_{max}$ *può essere risolto all'ottimo in tempo* $O(n \log n)$.

Dim. - La regola PEDD può essere implementata come segue. Consideriamo una lista contenente tutti i job rilasciati e non ancora terminati, ordinata secondo due date non decrescenti (ovvero, in ordine EDD). Con lo scorrere del tempo, questa lista verrà aggiornata. Gli eventi che causano una variazione della lista sono due: il rilascio di un nuovo job oppure il completamento del job correntemente in esecuzione. Al momento del rilascio, un job viene inserito nella lista, e questa operazione ha costo $O(\log n)$. Il primo job della lista è quello correntemente in esecuzione. Quando esso finisce, viene rimosso dalla lista e il nuovo primo job va in esecuzione. Tra un evento e l'altro, la lista non cambia e dunque nemmeno il job in esecuzione. Peraltro, se all'istante t un job k inizia o viene ripreso, indicando con TR_k il suo tempo residuo di esecuzione, sappiamo già che il prossimo evento sarà all'istante $\min\{t + TR_k, r_{min}\}$, ove $r_{min} = \min\{r_j | r_j \geq t\}$. Dunque, gli eventi sono in tutto $O(n)$, ciascuno ha costo non superiore a $O(n)$ e il calcolo dell'istante del prossimo evento può essere effettuato in tempo costante. □

2.3.2 Il problema $1|r_j|L_{max}$

Consideriamo ora il problema di minimizzare la massima lateness, con release date, senza possibilità di interruzioni. Come vedremo, l'importanza di tale problema è anche legata al fatto che compare come sottoproblema di procedure risolutive di problemi molto più complessi.

Il problema $1|r_j|L_{max}$ è significativamente più difficile di tutti quelli incontrati finora[1], ma sfruttando opportunamente alcune sue proprietà, è ancora possibile risolverlo in tempi accettabili.

[1]Questo problema appartiene alla classe dei problemi NP-completi, per i quali *non possono* esistere algoritmi esatti polinomiali per la loro soluzione, sempre che valga una certa relazione tra classi di complessità ($P \neq NP$), non ancora dimostrata ma supposta vera da moltissimi ricercatori.

L'algoritmo utilizzato più comunemente per risolvere questo problema è un algoritmo di enumerazione implicita, ovvero di branch-and-bound. Si tratta cioè di enumerare tutte le possibili sequenze di job, senza però farlo esplicitamente - il che sarebbe proibitivo dal punto di vista computazionale - ma sfruttando opportunamente le informazioni specifiche che abbiamo su questo problema.

Per quanto riguarda il *branching*, l'algoritmo considera tutti i possibili schedule sequenziando i job da sinistra verso destra, ossia secondo il loro ordinamento naturale nello schedule. Cosi', costruiremo l'*albero di enumerazione* ponendo nella radice un nodo che rappresenta la sequenza vuota, e che avrà tanti figli quanti sono i modi di iniziare lo schedule, ovvero in generale n. Il generico nodo dell'albero di enumerazione conterrà una *sequenza parziale* $\mathcal{S}_h$ che specifica una possibile scelta e un ordinamento dei primi h job della soluzione.

Osservazione 1 *Sia $\mathcal{S}_h$ una sequenza parziale dei primi h job, e sia t il tempo di completamento dell'ultimo job di $\mathcal{S}_h$. Supponiamo esistano due job, $j \notin \mathcal{S}_h$ e $k \notin \mathcal{S}_h$, tali che*

$$r_j \geq \max\{t, r_k\} + p_k$$

Questo fatto implica che, se volessimo continuare la sequenza con j, dovremmo aspettare l'istante r_j, mentre prima di allora la macchina avrebbe tutto il tempo di iniziare e finire il job k. Dai figli del nodo $\mathcal{S}_h$, si può allora eliminare il nodo corrispondente alla sequenza parziale ottenuta aggiungendo j in coda alla sequenza $\mathcal{S}_h$, ossia $(\mathcal{S}_h, j)$.

L'osservazione 1 consente di risparmiare la generazione esplicita di molte sequenze. Tuttavia, per rendere efficace il procedimento, conviene utilizzare un'ulteriore informazione. Come in tutti gli algoritmi di branch-and-bound, l'efficienza dell'algoritmo dipenderà in modo essenziale dalla possibilità di calcolare in ciascun nodo un lower bound significativo sul valore della migliore soluzione ottenibile da quel nodo in giù. Nella risoluzione di problemi di programmazione intera, per ottenere un lower bound si risolve un *rilassamento* del problema in esame, una tipica scelta essendo quella di rilassare i vincoli di interezza (rilassamento lineare). La scelta è motivata da due fatti: (*i*) la soluzione ottima del PL fornisce un bound sulla soluzione ottima del problema di PLI, e (*ii*) la soluzione ottima del PL può essere calcolata in modo efficiente. Concettualmente possiamo allora ragionare allo stesso modo, ossia considerare un problema che sia risolvibile in modo efficiente, e la cui soluzione ottima fornisca effettivamente un bound sul problema in esame. Nel nostro caso, avendo fissato una sequenza parziale $\mathcal{S}_h$, possiamo ottenere un lower bound sul valore ottimo di L_{max} sulla parte rimanente dello schedule risolvendo un'istanza del

j	p_j	r_j	d_j
1	4	0	11
2	2	1	12
3	6	3	11
4	5	5	10

Tabella 1: Dati dell'esempio numerico per il problema $1|r_j|L_{max}$.

problema $1|r_j, preempt|L_{max}$, in cui cioè rilassiamo il vincolo che i job non siano interrompibili, limitatamente ai job che non fanno parte di $\mathcal{S}_h$. Se, risolvendo tale istanza, la soluzione ottima del problema rilassato risultasse essere uno schedule senza interruzioni, avremmo chiaramente ottenuto anche una soluzione ammissibile per il problema originario e dunque potremmo eliminare il nodo dell'albero di enumerazione. Altrimenti, indicando con z il valore della soluzione ottima del problema rilassato al nodo $\mathcal{S}_h$, se esiste una soluzione ammissibile S con $L_{max}(S) < z$, possiamo evidentemente eliminare il nodo $\mathcal{S}_h$ in quanto siamo certi che, proseguendo in profondità, non potremmo trovare nessuna soluzione migliore di S.

Esempio 1 *Si consideri l'esempio con i dati in Tabella 1. Risolvendo la versione preemptive del problema, si ottiene uno schedule in cui viene eseguito dapprima per intero il job 1, quindi una unità di job 3, quindi il job 4 per intero, poi il job 3 fino al suo completamento e infine il job 2. Il valore di L_{max} è pari a 5 (determinato dal job 2). Al primo livello dell'albero di enumerazione avremo quattro possibili figli del nodo radice, corrispondenti ai quattro modi diversi di fissare il primo job della sequenza. Utilizzando l'osservazione 1, possiamo però vedere che le sequenze parziali $\{3\}$ e $\{4\}$ possono essere tralasciate. Infatti, essendo le release date di questi due job rispettivamente 3 e 5, vediamo che in ambedue i casi la macchina avrebbe il tempo di eseguire per intero il job 2. Dunque, possiamo limitarci a considerare solo le sequenze parziali $\{1\}$ e $\{2\}$.*

Fissando il job 1 in prima posizione, esso termina all'istante 4 ($L_1 = -7$). Applicando la regola PEDD agli altri tre job, abbiamo che nell'intervallo $[4, 5]$ viene processato 3, il quale viene interrotto dal rilascio di 4. Il job 4 viene processato fino al suo completamento, all'istante 10 ($L_4 = 0$). A questo punto il job 3 può riprendere fino al termine, all'istante 15 ($L_3 = 4$). Infine, segue il job 2, che termina all'istante 17 ($L_2 = 5$). La massima lateness è quella del job 2, pari a 5. Dunque, il lower bound associato al nodo $\{1\}$ è 5.

Passando all'esame del nodo $\{2\}$, si può analogamente costruire uno schedule fissando in prima posizione il job 2. Si noti che poiche' $r_2 = 1$, all'inizio la macchina sarà inattiva

fino a $t = 1$. Il job 2 termina all'istante 3 ($L_2 = -9$). Applicando PEDD, al tempo $t = 3$ possiamo iniziare il job 1 oppure il job 3, ambedue disponibili; supponiamo di scegliere 1. All'istante 5, il job 1 dovrà essere interrotto per il sopraggiungere del job 4, che verrà eseguito fino al suo completamento, all'istante 10 ($L_4 = 0$). Quindi si può riprendere 1 (oppure si poteva iniziare 3), che terminerà all'istante 12 ($L_1 = 1$), e infine viene processato 3, che sarà completato all'istante 18 ($L_3 = 7$). Essendo $L_{max} = L_3 = 7$, questo è il valore del lower bound al nodo $\{2\}$.

Fissando la sequenza parziale $\{1, 2\}$, la regola PEDD dà luogo alla sequenza senza interruzioni $\{1, 2, 4, 3\}$, con $L_{max} = L_3 = 6$. Possiamo elimiare dunque il nodo $\{2\}$, dal momento che disponiamo di una soluzione ammissibile di valore migliore del lower bound a quel nodo.

Fissando invece la sequenza $\{1, 3\}$, otteniamo la sequenza, pure senza interruzioni, $\{1, 3, 4, 2\}$ che è ancora migliore della precedente, essendo $L_{max} = L_4 = 5$. Unico nodo ancora da esplorare rimane $\{1, 4\}$, ma considerando che al nodo radice il lower bound era già 5, concludiamo che $\{1, 3, 4, 2\}$ è la soluzione ottima. □

2.4 Problemi multi-obiettivo

Vogliamo ora analizzare un problema in cui vi sono due distinti obiettivi, anzichè uno solo. Infatti, in realtà vi possono essere diverse esigenze, talora contrastanti, tra le quali si vuole trovare il miglior compromesso.

Nel momento in cui si considera più di un obiettivo, è necessario riconsiderare cosa sia da intendersi con *soluzione ottima*. Dato uno schedule S, e dette $f_1(S)$ e $f_2(S)$ le due funzioni obiettivo (da minimizzare), S si dice *nondominato* se non esiste nessun altro schedule S' tale che $f_1(S') \leq f_1(S)$, $f_2(S') \leq f_2(S)$, con almeno una delle due disuguaglianze valida in senso stretto. In altre parole, se S è nondominato, è possibile trovare uno schedule migliore per la prima funzione obiettivo solo a patto di peggiorare rispetto alla seconda (e viceversa).

Nei problemi multiobiettivo possiamo allora adottare diversi approcci.

(i) Ci si riconduce a un problema con singolo obiettivo, ottimizzando rispetto a $f_1(\cdot)$ ponendo un vincolo sul massimo valore che può assumere $f_2(\cdot)$ (o viceversa).

(ii) Si determinano tutte le soluzioni *nondominate*.

Peraltro, per determinare l'insieme di tutte le soluzioni nondominate, può essere necessario risolvere varie istanze del problema con singolo obiettivo.

2.4.1 Massima lateness e somma dei tempi di completamento

Nel seguito vedremo come trovare le soluzioni nondominate rispetto alle due funzioni obiettivo L_{max} e $\sum_{j=1}^{n} C_j$. In questo capitolo, per brevità indicheremo la seconda con $\bar{C}$. Vogliamo determinare l'insieme delle soluzioni nondominate rispetto a questi due obiettivi.[2] Per fare questo, risolveremo iterativamente il problema singolo obiettivo che possiamo indicare con $1|L_{max} \leq \Delta|\bar{C}$, ossia vogliamo trovare uno schedule che minimizzi la somma dei tempi di completamento, tra tutti quegli schedule tali che la massima lateness non superi un certo valore Δ (che può essere anche negativo).

Anzitutto, associamo a ogni job una *due date ausiliaria* d'_j ottenuta aumentando di Δ la propria, cioè $d'_j = d_j + \Delta$. Considerando le d'_j come le due date dei vari job, chiaramente uno schedule in cui nessuna di queste due date viene violata, definisce uno schedule in cui la massima lateness (rispetto alle due date originali) è non superiore a Δ. Dunque, possiamo esprimere il nostro problema dicendo che cerchiamo uno schedule che minimizza $\bar{C}$ tra tutti quelli in cui tutti i job arrivano in tempo (rispetto alle due date ausiliarie).

Il problema può essere risolto costruendo lo schedule a partire dal fondo (in modo simile all'algoritmo di Lawler). Poiche' comunque abbiamo a che fare con funzioni obiettivo regolari, e non ci sono release date, al solito non vi è alcuna ragione di tenere la macchina ferma, e dunque sappiamo che l'ultimo job terminerà all'istante $P = \sum_{j=1}^{n} p_j$. Ora, se per tutti i job si ha $d'_j < P$, possiamo concludere che non esistono schedule in cui tutti i job rispettano la propria due date, dal momento che uno dei job dovrà terminare in P. Consideriamo allora l'insieme $J(P)$ di job tali che $d'_j \geq P$. Questi sono i job candidati a essere messi in ultima posizione nello schedule. È facile a questo punto dimostrare che vale il seguente risultato:

TEOREMA 5 *Esiste una soluzione ottima al problema* $1|L_{max} \leq \Delta|\bar{C}$ *in cui in ultima posizione vi è il job* k *tale che*

$$p_k = \max\{p_j | j \in J(P)\}.$$

Dim.- Se in ultima posizione non vi fosse k ma un altro job j, pure appartenente a $J(P)$, e tale che $p_j < p_k$, potremmo scambiare i due job e cosi' facendo otterremmo uno schedule ancora ammissibile dal punto di vista della lateness, e strettamente migliore dal punto di vista della somma dei tempi di completamento, dal momento che tutti i job che nello

[2] Per maggiore precisione dovremmo dire che cerchiamo tutte le coppie (L_{max}, $\bar{C}$) tali che esista uno schedule nondominato che abbia quel valore di L_{max} e quel valore di $\bar{C}$. Non siamo interessati a trovare *tutti* gli schedule che danno luogo alla stessa coppia di valori, cosi' come in un problema singolo obiettivo non si è in genere interessati a determinare *tutte* le soluzioni ottime.

```
Algoritmo multiobiettivo
{
for  c = 1 to n let d'_j := d_j + Δ;
T := Σ_{j=1}^n p_j
     for  c = n down to 1
     {
          J̄ := {j | j non sequenziato, d'_j ≥ T};
          sia p* = max{p_j | j ∈ J̄}
          sia j* tale che d'_{j*} = max{d'_j | p_j = p*, j ∈ J̄}
          σ(c) := j*;
          T := T − p_{j*}
     }
}
```

Figura 3: Algoritmo per il problema $1|L_{max} \leq \Delta|\bar{C}$.

schedule di partenza sono compresi fra k e j anticipano il loro completamento di $p_k - p_j$. □

Appurato chi è l'ultimo job nella sequenza, possiamo allora sottrarre p_k da P, e ripetere il discorso rispetto al nuovo istante di completamento. Avremo adesso un nuovo insieme di job candidato, ossia $J(P - p_k)$. Di nuovo, dovremo scegliere quello più lungo e cosi' via.

Si noti che se, al generico passo dell'algoritmo, vi è più di un job avente massimo tempo di processamento (indicato con p^*) nell'insieme di job candidati, dal punto di vista della correttezza dell'algoritmo possiamo scegliere uno qualsiasi di questi. Tuttavia, per i motivi che vedremo dopo, facciamo un'ulteriore scelta, ossia, tra quelli di durata p^*, scegliamo di schedulare quello avente due date più alta. Si ha in definitiva l'algoritmo indicato in Figura 3.

L'algoritmo in Figura 3 consente dunque di trovare la soluzione che minimizza la somma dei tempi di completamento, con il vincolo che la massima lateness non superi un certo valore Δ. A questo punto siamo in grado di utilizzare più volte questo algoritmo per determinare tutte le soluzioni nondominate rispetto ai due obiettivi L_{max} e $\bar{C}$.

Anzitutto, è chiaro che esiste un valore massimo di Δ oltre il quale la soluzione ottima rispetto a $\bar{C}$ rimane la stessa: infatti, supponiamo di applicare l'algoritmo in Figura 3 per un valore di Δ molto elevato (ad esempio $\Delta > \sum_{j=1}^{n} p_j$). I job risulteranno sequenziati in ordine SPT, ottenendo cosi' lo schedule σ_{SPT}, e indichiamo la sua somma dei tempi di

j	p_j	d_j
A	2	17
B	3	4
C	4	14
D	3	10
E	5	12

Tabella 2: Dati dell'esempio numerico per il problema $1||L_{max}, \bar{C}$.

completamento con $\bar{C}(\sigma_{SPT})$. (Chiaramente, σ_{SPT} può anche essere ottenuto direttamente applicando la regola SPT, con l'avvertenza di sequenziare prima, tra due job di uguale durata, quello avente due date più bassa.) Se allora indichiamo con $L(\sigma_{SPT})$ il valore della massima lateness di questo schedule, σ_{SPT} è ottimo per $1|L_{max} \leq \Delta|\bar{C}$, per qualsiasi $\Delta \geq L(\sigma_{SPT})$.

TEOREMA 6 *σ_{SPT} è uno schedule nondominato.*

Dim.- Ovviamente, non esiste alcuno schedule avente somma dei tempi di completamento inferiore a $\bar{C}(\sigma_{SPT})$. Dunque, dobbiamo mostrare che non è possibile trovare uno schedule avente massima lateness inferiore a $L(\sigma_{SPT})$ senza peggiorare $\bar{C}$. Osserviamo anzitutto che, nell'algoritmo in Figura 3, quando due job di $\bar{J}$ hanno durata massima p^*, viene sequenziato prima (e dunque verrà eseguito dopo, nello schedule finale) quello avente due date più lontana. Dunque scambiarli non può migliorare L_{max}, e non ha alcun effetto su $\bar{C}$, in quanto hanno stessa durata. Quindi, per migliorare la massima lateness rispetto a $L(\sigma_{SPT})$ è necessario cambiare di posto job aventi durate diverse. Ma qualunque scambio di job aventi durate diverse porta sicuramente a violare l'ordine SPT, e quindi implica un aumento della somma dei tempi di completamento. □

Il teorema 6 vale anche se si considera, al posto di σ_{SPT}, lo schedule ottenuto applicando l'algoritmo in Figura 3 per qualsiasi valore di Δ. Dunque, se σ è uno schedule nondominato, e vogliamo trovare una soluzione nondominata con un valore di massima lateness inferiore a $L_{max}(\sigma)$, basterà risolvere il problema ausiliario con $\Delta = L_{max}(\sigma) - 1$. La soluzione ottima σ' darà luogo a un valore di lateness $L_{max}(\sigma') \leq L_{max}(\sigma)$ e cosi' via. In definitiva si ottiene l'algoritmo in Figura 4, in cui Δ viene inizializzato al valore P in modo da garantire che al primo passo viene trovato lo schedule σ_{SPT}.

ESEMPIO 2 *Si consideri l'esempio in Tabella 2. Lo schedule SPT nondominato è*

$$\sigma_1 = (A,\ B,\ D,\ C,\ E)$$

```
Algoritmo nondominate
{
let ND:=∅; Δ := P;
          repeat
          {
          risolvi 1|L_max ≤ Δ|C̄
          if esiste soluzione ammissibile
             {
             sia σ la sol.  ottima
             ND := ND  ∪ σ
             Δ := L_max(σ) − 1
             }
          else σ = ∅
          }
          until σ = ∅
}
```

Figura 4: Algoritmo che trova le soluzioni nondominate per il problema $1||L_{max}, \bar{C}$.

per il quale si ha $L_{max}(\sigma_1) = 5$ *(determinato dal job E) e* $\bar{C}(\sigma_1) = 44$*. Cerchiamo allora un altro schedule nondominato ponendo* $\Delta = 4$*, e risolviamo il problema* $1|L_{max} \leq \Delta|\bar{C}$*. Essendo* $P = 17$*, abbiamo che* $J(P) = \{A, C\}$*. Essendo C il più lungo, poniamo C in ultima posizione. A questo punto consideriamo i job dell'insieme* $J(P - p_C) = \{A, D, E\}$*. La scelta ricade su E. Al passo successivo,* $J(P - p_C - p_E) = \{A, B, D\}$ *e viene scelto D in quanto ha una due date più alta di B. In definitiva si ottiene*

$$\sigma_2 = (A,\ B,\ D,\ E,\ C)$$

per cui si ha $L_{max}(\sigma_2) = 3$ *(determinato dal job C) e* $\bar{C}(\sigma_2) = 45$*. Si noti che se, tra B e D, fosse stato scelto prima B (ossia, se avessimo scambiato B e D), la soluzione ottenuta sarebbe stata ancora ottima dal punto di vista del problema* $1|L_{max} \leq \Delta|\bar{C}$*, ma avremmo avuto un valore di* L_{max} *pari a 4, e dunque non sarebbe stata una soluzione nondominata. Proseguendo nell'analisi porremo allora* $\Delta = 2$*, e iterando nuovamente si ottiene lo schedule*

$$\sigma_3 = (B,\ D,\ E,\ C,\ A)$$

per cui si ha $L_{max}(\sigma_3) = 0$ *(determinato dal job A) e* $\bar{C}(\sigma_3) = 52$*. Si potrebbe a questo punto continuare e porre* $\Delta = -1$*, ma osservando che* $d_j \leq P = 17$ *per tutti i* j*, eviden-*

temente non è possibile trovare una soluzione in cui tutti i job arrivano con lateness non superiore a -1, e quindi in definitiva l'insieme degli schedule nondominati è $\{\sigma_1, \sigma_2, \sigma_3\}$. □

Infine, alcune brevi osservazioni riguardo la complessità. È lasciato come esercizio verificare che ciascuna istanza del problema $1|L_{max} \leq \Delta|\bar{C}$ può essere risolta in $O(n \log n)$. Ci si può chiedere inoltre *quante* possono essere, come ordine di grandezza, le soluzioni nondominate. Anche se l'esempio 2 potrebbe far pensare il contrario, si può vedere che esistono casi in cui il numero di soluzioni nondominate è esponenziale rispetto al numero di job. Dunque, in generale l'enumerazione di tutte le soluzioni può richiedere un tempo anche molto lungo, anche se comunque questi casi sono più di interesse teorico che pratico.

3 Flow shop

Consideriamo ora il modello del flow shop, in cui n job devono essere lavorati su m macchine disposte in serie. Stavolta un job consiste di m task (uno per ogni macchina), e p_{ij} indica il tempo di processamento del job j sulla macchina i. L'ordine in cui le macchine sono visitate da ciascun job è lo stesso per tutti i job, e dunque numeriamo le macchine corrispondentemente. Questo modello è rappresentativo di una gamma molto ampia di applicazioni, dai sistemi di produzione seriali alle supply chain.

Nel seguito supporremo che tra una macchina e la successiva sia presente un buffer, di capacità sufficientemente alta da poter ospitare i job in attesa di essere processati. In generale, il buffer può consentire di risequenziare i job tra una macchina e l'altra, ossia di fare in modo che l'ordine con cui i job visitano la macchina $i+1$ sia diverso da quello con cui visitano la macchina i. In questo caso generale le macchine potranno essere sequenziate in modo diverso l'una dall'altra, e le possibili soluzioni saranno allora $(n!)^m$. Se invece non sono consentiti sorpassi (permutation flow shop), il sequenziamento è unico per tutte le macchine, e quindi le possibili soluzioni sono $n!$.

I problemi di flow shop sono considerevolmente più complessi di quelli con macchina singola. Nel seguito, limiteremo la nostra analisi al problema in cui si vuole minimizzare il massimo tempo di completamento o makespan, ossia il problema $Fm||C_{max}$.

3.1 Il grafo disgiuntivo

Introduciamo ora uno strumento di rappresentazione delle soluzioni ammissibili che risulta molto utile in molti problemi di scheduling. Data un'istanza di $Fm||C_{max}$, possiamo

definire un grafo $\mathcal{G}$ con $mn+2$ nodi, definito come segue. Per ognuno degli mn task vi è un nodo. Il nodo $[ij]$ corrisponde al task i-esimo del job j, e dunque vi associamo, come peso, la quantità p_{ij}. Tra i nodi dei task di uno stesso job esistono dei vincoli di precedenza. Formalmente, l'i-esimo task di un job non può iniziare prima che lo stesso job sia stato completato sulla macchina $i-1$, e questo è espresso per mezzo di un arco orientato da $[i-1,j]$ a $[ij]$, per ogni $i=2,\ldots,m$, $j=1,\ldots,n$ (archi *orizzontali*). Consideriamo ora invece i task relativi alla stessa macchina i. Chiaramente, un qualunque sequenziamento su i induce altre relazioni di precedenza tra i task. Indichiamo allora questo fatto introducendo in $\mathcal{G}$, per ogni coppia di task relativi alla stessa macchina, un arco *non orientato*. Formalmente, avremo cioè un arco tra $[ij]$ e $[ik]$ per ogni $i=2,\ldots,m$, $j,k=1,\ldots,n$. Questi si chiamano archi *disgiuntivi*, in quanto, in qualunque soluzione ammissibile, dovranno essere orientati in un verso oppure nell'altro. Infine, per completare il grafo $\mathcal{G}$, aggiungiamo un nodo-sorgente s, che non ha predecessori e ha come successori tutti i nodi del tipo $[1,j]$, e un nodo-pozzo t, che non ha successori e ha come predecessori tutti i nodi del tipo $[m,j]$.

Dato, per ciascuna macchina i, un sequenziamento σ^i, indichiamo con σ l'insieme degli m sequenziamenti, e con $\mathcal{G}(\sigma)$ il grafo orientato ottenuto da $\mathcal{G}$ orientando gli archi disgiuntivi secondo σ. Si noti che, scelto in qualsiasi modo σ, il grafo $\mathcal{G}(\sigma)$ è aciclico. Infatti, qualunque ciclo dovrebbe necessariamente fare uso di archi orizzontali, e questi sono tutti orientati nello stesso verso, ossia non esiste alcun arco $[pj] \rightarrow [qj]$ con $q<p$.

Il grafo $\mathcal{G}(\sigma)$ è concettualmente identico alle reti di attività utilizzate nella gestione progetti (PERT, CPM...), con l'unica differenza che in questo caso i task corrispondono ai nodi e le precedenze agli archi. Se allora consideriamo qualsiasi cammino dal nodo s al nodo t, avremo che il makespan corrispondente all'insieme di sequenziamenti σ non può essere inferiore alla somma dei pesi dei nodi del cammino. Dal fatto che la regolarità della funzione obiettivo non rende mai conveniente far attendere un job, discende la conclusione:

Teorema 7 *Dato un insieme di sequenziamenti σ, il makespan è dato dal peso del cammino di peso massimo dal nodo s al nodo t su $\mathcal{G}(\sigma)$.*

Dunque, specificato un insieme di sequenziamenti σ, il makespan corrispondente può essere ottenuto calcolando un percorso massimo su grafo aciclico. Nel seguito useremo quindi il termine *schedule* un insieme di sequenziamenti per le varie macchine. Il problema è dunque quello di trovare uno schedule che minimizzi il makespan.

3.2 Schedule di tipo permutation

Vogliamo dunque risolvere il problema $Fm||C_{max}$, ossia trovare uno schedule che minimizzi il makespan. Come abbiamo detto, l'insieme degli schedule nel caso generale è molto più ampio che non nel caso permutation. Tuttavia, è facile dimostrare che, in un flow shop con m macchine, non si perde in generalità se si adotta lo stesso sequenziamento per le prime due macchine e analogamente per le ultime due macchine.

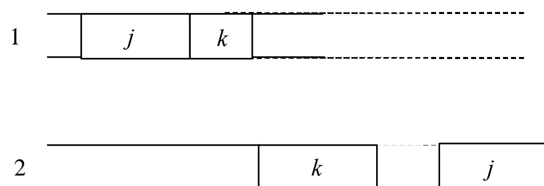

Figura 5: Le prime due macchine nella dimostrazione del Teorema 8.

TEOREMA 8 *Nel problema* $Fm||C_{max}$*, esiste sempre una soluzione ottima in cui* $\sigma^1 = \sigma^2$ *e* $\sigma^{m-1} = \sigma^m$.

Dim.- Si consideri uno schedule S in cui esistono due job, j e k, tali che j precede immediatamente k sulla macchina 1, e viceversa k precede j (eventualmente con altri job tra loro) sulla macchina 2 (Figura 5). Poichè $S_1(j) < S_1(k) < S_2(k) < S_2(j)$, possiamo evidentemente scambiare fra loro j e k sulla macchina 1, senza che questo ritardi alcun job sulla macchina 2. Questo procedimento si può ripetere fino a rendere la sequenza sulla macchina 1 uguale a quella sulla macchina 2. Il discorso per le ultime due macchine è identico, potendo modificare il sequenziamento sulla macchina m fino a renderlo uguale a quello sulla macchina $m - 1$. □

Il Teorema 8 implica che per $m = 2$ e $m = 3$ possiamo limitarci a considerare i soli schedule di tipo *permutation*, ossia in cui σ^i è uguale per tutte le macchine, mentre per m generico le soluzioni che è necessario considerare saranno $(n!)^{m-2}$.

Mentre per $m = 2$, come vedremo, questa osservazione è sufficiente a derivare un algoritmo molto efficiente di soluzione, per $m \geq 3$ non esistono algoritmi risolutivi polinomiali. In quei casi è allora necessario adottare un approccio di enumerazione implicita.

3.3 Il problema $F2||C_{max}$

Il caso di due macchine è risolto per mezzo di un celebre algoritmo dovuto a Selmer Johnson (1954). L'idea alla base dell'algoritmo è relativamente semplice. Poichè supponiamo che non vi siano release dates (con le quali il problema diverrebbe molto più difficile), la macchina 1 è sempre attiva, dall'istante 0 fino a completamento di tutti i job (ossia, all'istante $\sum_{j=1}^{n} p_{1j}$, si noti che questo valore non dipende dal sequenziamento). Dal momento che l'obiettivo è terminare le lavorazioni il più presto possibile, dovremo cercare di fare in modo che la macchina 2 sia sempre attiva. Per fare questo, converrà tenere il buffer intermedio il più possibile pieno di job, in modo che la macchina 2 possa essere sempre "alimentata". Dunque, sembra ragionevole sequenziare, come primo job, uno che abbia un tempo di processamento breve sulla macchina 1 (in modo da dare subito qualcosa da fare alla macchina 2) e magari lungo sulla macchina 2 (in modo che il buffer abbia tempo di riempirsi). Viceversa, come ultimo job dello schedule sembra ragionevole porre un job che abbia un tempo di processamento breve sulla macchina 2 e lungo sulla macchina 1.

L'algoritmo di Johnson costruisce lo schedule ottimo dagli estremi verso il centro. All'inizio, si considera, tra tutti i tempi di processamento dei task, il più piccolo. Se tale tempo minimo corrisponde a un task sulla macchina 1, allora il job cui quel task appartiene viene posto in prima posizione. Se invece il tempo minimo corrisponde a un task sulla macchina 2, il job cui quel task appartiene viene posto in ultima posizione. La stessa regola si applica al generico passo. Nei due casi, rispettivamente, il job viene accodato ai primi job della sequenza, o posto in testa agli ultimi. Si ottiene in definitiva l'algoritmo in Figura 6.

Per dimostrare che l'algoritmo di Johnson trova la soluzione ottima, ci limitiamo per brevità a dimostrare che il primo job viene collocato in modo ottimo. Ripetendo la discussione sull'insieme dei job rimanenti, si arriva a concludere che l'algoritmo di Johnson trova lo schedule ottimo.

TEOREMA 9 *Sia* $p_{min} = \min\{p_{ij} | i = 1, 2; j = 1, \ldots, n\}$. *Se* $p_{min} = p_{1k}$, *allora esiste una soluzione ottima in cui* k *è in prima posizione. Se* $p_{min} = p_{2h}$, *allora esiste una soluzione ottima in cui* h *è in ultima posizione.*

Dim.- Dimostriamo la prima parte del teorema, ossia che se il minimo tempo di processamento è sulla macchina 1, allora il job k corrispondente va posto in testa allo schedule. Per fare questo, consideriamo uno schedule S in cui k è preceduto da un job j, e chiamiamo S' lo schedule ottenuto scambiando di posto j e k. Vogliamo mostrare che il valore di C_{max} non può peggiorare per effetto di questo scambio. Siano Q e R gli istanti in cui la macchina 1 e, rispettivamente, la macchina 2, terminano di lavorare il job che precede

```
Algoritmo di Johnson
{
        r := 1; s := n; U := {1, 2, ..., n}
        while  U ≠ ∅
        {
              p_{i*,j*} := min{p_ij | j ∈ U};
              if  i* = 1  then  σ(r) := j*;  r := r + 1
              if  i* = 2  then  σ(s) := j*;  s := s − 1
              U := U − {j*};
        }
        return the schedule σ;
}
```

Figura 6: Algoritmo di Johnson.

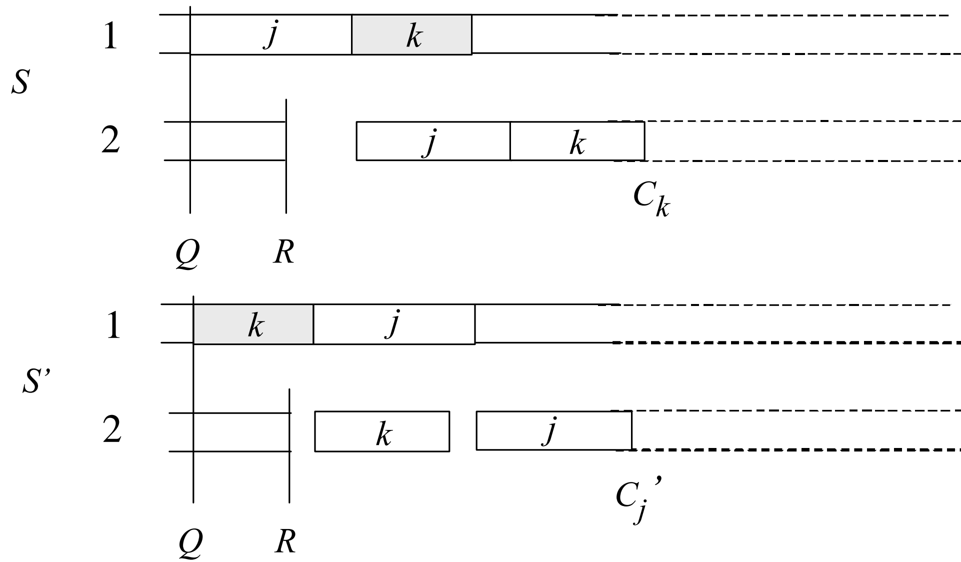

Figura 7: Le prime due macchine nella dimostrazione del Teorema 9.

j	p_{1j}	p_{2j}
1	3	3
2	4	5
3	2	3
4	5	2
5	3	4

Tabella 3: Dati dell'esempio numerico per il problema $F2||C_{max}$.

j in S. Indichiamo allora con C_k il tempo di completamento di k in S e con C'_j quello di j in S' (Figura 7). Osserviamo prima lo schedule S. Essendo $p_{1k} = p_{min}$, di certo j sulla macchina 2 termina dopo k sulla macchina 1. Di conseguenza:

$$C_k = \max\{Q + p_{1j}, R\} + p_{2j} + p_{2k}$$

Nello schedule S', il job j può anche terminare sulla macchina 1 dopo la fine di k sulla macchina 2 (come illustrato in Figura 7). Dunque avremo:

$$C'_j = \max\{\max\{Q + p_{1k}, R\} + p_{2k} + p_{2j}, Q + p_{1k} + p_{1j} + p_{2j}\} \tag{4}$$

Consideriamo il primo dei due termini della (4). Poichè $p_{1k} \leq p_{1j}$, senz'altro

$$\max\{Q + p_{1k}, R\} + p_{2k} + p_{2j} \leq C_k \tag{5}$$

e inoltre, essendo $p_{1k} \leq p_{2k}$ e ovviamente $p_{1j} \leq \max\{Q + p_{1j}, R\}$,

$$Q + p_{1k} + p_{1j} + p_{2j} \leq C_k \tag{6}$$

dalle (5) e (6) segue che $C'_j \leq C_k$. Reiterando il ragionamento, conviene quindi anticipare k fino a portarlo in prima posizione. La seconda parte del teorema è del tutto simmetrica e viene lasciata per esercizio al lettore. □

Esempio 3 *Si consideri l'esempio in Tabella 3. Il tempo di processamento più piccolo è* $p_{13} = p_{24} = 2$. *L'algoritmo di Johnson porrà dunque il job 3 in prima posizione, e il job 4 in ultima. A questo punto, il più piccolo tempo di processamento è* $p_{11} = p_{21} = p_{15} = 2$. *Possiamo allora a scelta mettere il job 1 in coda al job 3, oppure in testa al job 4 o ancora possiamo porre il job 5 in coda al job 3. Effettuiamo, arbitrariamente, la prima scelta. A questo punto necessariamente porremo il job 5 in coda al job 1, e rimane poi solo il job 2. In definitiva la sequenza ottima prodotta è* $\{3, 1, 5, 2, 4\}$, *per un makespan di* $C_{max} = 19$.
□

4 Job shop

Si è già avuto modo di osservare che il flow shop è un caso particolare di job shop in cui le macchine sono visitate nello stesso ordine da tutti i job. Nel job shop generale, invece, si ha che tale ordine è diverso da job a job, e anzi uno stesso job può anche visitare più volte (o nessuna) la stessa macchina. La sequenza di macchine per un certo job j viene chiamata *instradamento* del job j. Data un'istanza di job shop, indichiamo allora con p_{ij} la durata dell'i-esimo task del job j, e sia μ_{ij} la macchina sulla quale esso deve essere svolto.

4.1 Il grafo disgiuntivo per il job shop

Anche per il problema del job shop è possibile utilizzare la stessa rappresentazione delle soluzioni ammissibili vista per il flow shop. Precisamente, possiamo ancora definire un grafo disgiuntivo $\mathcal{G}$ in cui ogni nodo corrisponde a un task e gli archi esprimono vincoli di precedenza. Oltre agli archi orizzontali, fissi, ci sono gli archi disgiuntivi, non orientati, uno per ogni coppia di task che richiedono la stessa macchina, tranne che per i nodi dello stesso job. Infatti se vi sono due task, diciamo $[ij]$ e $[kj]$, con $k > i$, anche se $\mu_{ij} = \mu kj$ è inutile introdurre un arco disgiuntivo tra questi due task, perchè $[ij]$ verrà senz'altro svolto prima di $[kj]$.

Una importante differenza tra job shop e flow shop è espressa nel seguente teorema, che costituisce il corrispettivo del Teorema 7. Come prima, indichiamo con σ l'insieme dei sequenziamenti σ^i sulle singole macchine, $i = 1, \dots, m$.

TEOREMA 10 *Dato un sequenziamento σ:*

- *Se il grafo $\mathcal{G}(\sigma)$ presenta cicli, il sequenziamento è inammissibile*
- *Se il grafo $\mathcal{G}(\sigma)$ è aciclico, il makespan è dato dal peso del cammino di peso massimo dal nodo s al nodo t.*

Dunque, scelto un sequenziamento σ^i per ogni macchina i, non è detto che lo schedule risultante σ sia ammissibile.

ESEMPIO 4 *Si consideri l'esempio in Tabella 4, con 3 job e 4 macchine. La Figura 8 mostra una soluzione ammissibile, in cui è mostrato il grafo disgiuntivo $\mathcal{G}(\sigma)$ e il corrispondente diagramma di Gantt delle attività su ciascuna delle 4 macchine. I tre job sono identificati per mezzo dei colori (1 bianco, 2 grigio chiaro, 3 grigio scuro). La*

j	p_{1j}	μ_{1j}	p_{2j}	μ_{2j}	p_{3j}	μ_{3j}	p_{4j}	μ_{4j}
1	3	1	4	2	2	3	8	4
2	2	2	5	1	6	4		
3	3	3	6	1	4	2		

Tabella 4: Dati dell'esempio numerico per l'esempio di problema $J4||C_{max}$.

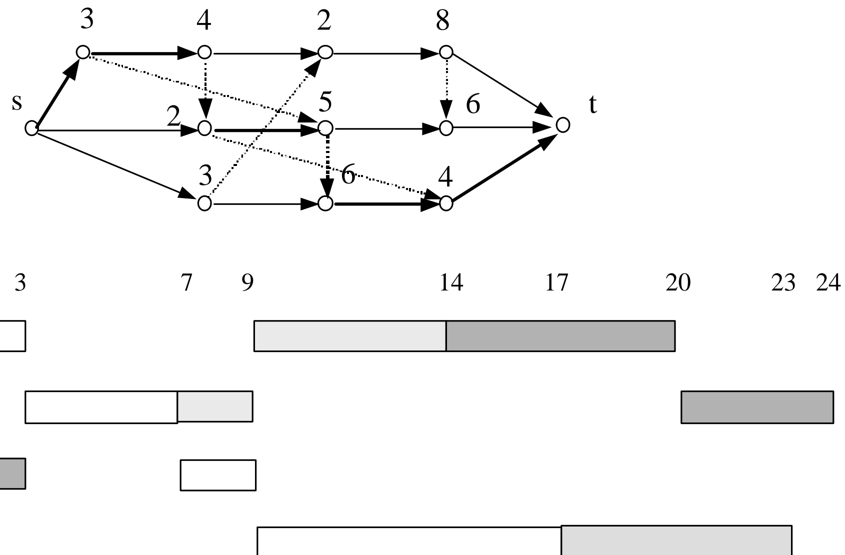

Figura 8: Soluzione ammissibile per il problema in Tabella 4. Le tre ombreggiature corrispondono ai job 1 (bianco), 2 (grigio chiaro), 3 (grigio scuro).

soluzione è $\sigma^1 = \{1, 2, 3\}$, $\sigma^2 = \{1, 2, 3\}$, $\sigma^3 = \{3, 1\}$, $\sigma^4 = \{1, 2\}$. *Il makespan è determinato dal job 3, che finisce dopo tutti gli altri, ed è pari a* $C_{max} = C_3 = 24$. *Sul grafo disgiuntivo è inoltre indicato il cammino di peso massimo. Osserviamo che se si cambiasse l'ordinamento sulla macchina 2 in* $\tilde{\sigma}^2 = \{3, 2, 1\}$, *si otterrebbe il grafo disgiuntivo in Figura 9. Poichè presenta un ciclo, lo schedule definito dai sequenziamenti* $\{\sigma^1, \tilde{\sigma}^2, \sigma^3, \sigma^4\}$ *non è ammissibile.* □

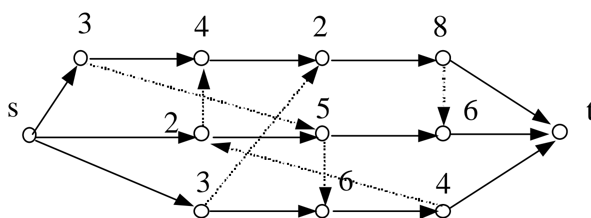

Figura 9: Esempio di schedule non ammissibile.

4.2 Euristica shifting bottleneck per il job shop

Il problema $J||C_{max}$ è considerato uno dei più difficili problemi di ottimizzazione combinatoria. E' in questo senso indicativo che una particolare istanza del problema, con $n = 10$ e $m = 10$, proposta da Muth e Thompson nel 1963, sia rimasta insoluta per più di 20 anni, quando è stata infine risolta (da Carlier e Pinson) grazie a un ingegnoso algoritmo di branch-and-bound (dopo molti giorni di tempo di calcolo). Nel seguito esamineremo invece l'algoritmo forse più utilizzato per risolvere problemi di job shop. Va sottolineato che tale algoritmo è di tipo euristico, ossia non vi è alcuna garanzia che la soluzione trovata dall'algoritmo sia ottima. Il motivo però della sua popolarità sta nel fatto che, da un lato, questo algoritmo richiede un tempo di calcolo relativamente basso, ma mediamente la qualità della soluzione prodotta è molto elevata.

L'idea alla base dell'algoritmo è abbastanza semplice. In considerazione del Teorema 10, il problema consiste nel trovare uno schedule σ dei vari job su ciascuna delle m macchine, in modo tale, ovviamente, che il grafo $\mathcal{G}(\sigma)$ sia aciclico, e che la lunghezza del percorso critico su tale grafo sia la più bassa possibile. Ora, tra le varie macchine ve ne sarà una in qualche modo più "critica" delle altre, nel senso che il modo in cui i job sono

sequenziati su questa macchina influenza il valore del makespan in modo più consistente rispetto ad altre macchine. In genere la risorsa critica viene indicata col termine ”collo di bottiglia” o *bottleneck*. L'algoritmo, a ogni passo, identifica la macchina che costituisce il bottleneck corrente, e fissa il sequenziamento su questa macchina. Al passo successivo, va in cerca della macchina che, tra quelle ancora non sequenziate, risulta maggiormente critica, sequenzia quest'ultima e cosi' via. Poichè il ruolo di macchina bottleneck viene rivestito a ogni passo da una macchina diversa, l'algoritmo si chiama *shifting bottleneck*.

Nel seguito, indichiamo con M l'insieme delle macchine, e con M_0 quelle che, al passo corrente, sono state già sequenziate, ossia per le quali è stato già scelto un sequenziamento. Un passo dell'algoritmo consiste nell'individuare una nuova macchina da inserire nell'insieme M_0.

Al generico passo, consideriamo allora il grafo ottenuto, a partire da $\mathcal{G}$, fissando gli archi che esprimono il sequenziamento sulle macchine di M_0, mentre tutti gli archi disgiuntivi delle macchine in $M - M_0$ sono cancellati. Indichiamo questo grafo con $\mathcal{G}(M_0)$. Il fatto di non inserire gli archi disgiuntivi associati a una particolare macchina vuol dire trascurare il fatto che quei task andranno effettuati in realtà in sequenza. Calcoliamo la lunghezza del percorso critico su $\mathcal{G}(M_0)$, e indichiamola con $C_{max}(M_0)$. Non siamo ancora in possesso di una soluzione ammissibile, perchè abbiamo sequenziato solo alcune macchine, e dunque, procedendo a fissare il sequenziamento anche sulle altre, aggiungeremo archi a $\mathcal{G}(M_0)$.

Sul grafo $\mathcal{G}(M_0)$, sia $[ij]$ un task relativo a una macchina non ancora sequenziata e consideriamo la lunghezza x del percorso critico da s fino al nodo $[ij]$. Questo implica che, con il sequenziamento fissato fino a questo momento, il task $[ij]$ non può iniziare prima dell'istante $x - p_{ij}$. Sia invece y la lunghezza del percorso critico da $[ij]$ a t. Se vogliamo che il task $[ij]$ non determini un allungamento del makespan rispetto al valore corrente $C_{max}(M_0)$, esso non potrà iniziare più tardi dell'istante $C_{max}(M_0)-y$. Quindi, sul grafo corrente è possibile individuare, per ogni task non ancora sequenziato, una finestra temporale tale che, se l'esecuzione del task avviene in questa finestra, il mantenimento del makespan corrente è garantito. Se invece l'esecuzione di $[ij]$ terminasse oltre questa finestra, la soluzione subirebbe un aumento del valore del makespan pari all'entità dello sforamento stesso. In definitiva allora, data una macchina k non ancora sequenziata, possiamo definire un'istanza del problema $1|r_j|L_{max}$ in cui i job corrispondono ai task che devono essere eseguiti da quella macchina, mentre release e due dates individuano le finestre temporali di cui sopra. Il valore ottimo di L_{max} indica di quanto, il fatto di sequenziare i task su quella macchina, fa peggiorare il makespan rispetto al valore corrente. Indichiamo allora con $L_{max}(k)$ tale valore ottimo. Il discorso può ripetersi,

```
Algoritmo Shifting Bottleneck
{
        M_0 := ∅; cont := 0
        while cont < m
        {
            for k ∉ M_0
            {
                calcola su G(M_0) release date e due date
                          per i task della macchina k;
                risolvi l'istanza di 1|r_j|L_max associata alla macchina k;
                sia L_max(k) il valore della soluzione ottima,
                          σ^k il sequenziamento;
            }
            L_max(k*) = max_k{L_max(k)};
            M_0 := M_0 ∪ {k*};
            cont:=cont+1;
        }
        return the schedule σ = {σ^k, k = 1,...,m};
}
```

Figura 10: Algoritmo Shifting Bottleneck

separatamente, per tutte le macchine. Alla fine, quella macchina k^* per la quale il valore $L_{max}(k)$ è più alto delle altre, è quella che maggiormente determina il valore del makespan, ovvero è il bottleneck. Poichè è possibile dimostrare che gli archi che definiscono la soluzione ottima del problema su macchina singola non possono creare cicli nel grafo $\mathcal{G}(M_0)$, i task di k^* vengono allora sequenziati secondo la soluzione ottima dell'istanza di $1|r_j|L_{max}$. La macchina k^* viene allora aggiunta all'insieme M_0, e $\mathcal{G}(M_0)$ viene di conseguenza aggiornato, aggiungendo gli archi che esprimono il sequenziamento su k^* e si va al passo successivo.

Il procedimento consiste dunque di m passi, in quanto a ogni passo esattamente una macchina viene sequenziata e non verrà più considerata nel seguito. Alla fine tutte le macchine sono sequenziate e quella ottenuta è una soluzione ammissibile, come detto di qualità in genere molto buona. Si noti che il problema $1|r_j|L_{max}$ può essere risolto per mezzo dell'algoritmo di branch and bound visto nel capitolo 2.3.2. L'algoritmo è schematizzato in Figura 10 e illustrato nell'esempio seguente.

job	p_{1j}, μ_{1j}	p_{2j}, μ_{2j}	p_{3j}, μ_{3j}	p_{4j}, μ_{4j}
1	10,1	8,2	4,3	
2	8,2	3,1	5,4	6,3
3	4,1	7,2	3,4	

Tabella 5: Dati dell'esempio numerico per l'euristica shifting bottleneck.

job j	p_j	r_j	d_j
1	10	0	10
2	3	8	11
3	4	0	12

Tabella 6: Istanza di $1|r_j|L_{max}$ per la macchina 1.

job j	p_j	r_j	d_j
1	8	10	18
2	8	0	8
3	7	4	19

Tabella 7: Istanza di $1|r_j|L_{max}$ per la macchina 2, primo passo dell'algoritmo.

job j	p_j	r_j	d_j
1	8	10	23
2	8	0	10
3	7	17	24

Tabella 8: Istanza di $1|r_j|L_{max}$ per la macchina 2, secondo passo dell'algoritmo.

job j	p_j	r_j	d_j
1	4	18	27
2	6	18	27

Tabella 9: Istanza di $1|r_j|L_{max}$ per la macchina 3, secondo passo dell'algoritmo.

macchina i	σ^i
1	1,2,3
2	2,1,3
3	2,1
4	2,3

Tabella 10: La soluzione prodotta dall'algoritmo shifting bottleneck per l'istanza in Tabella 5.

Esempio 5 [3] *Si consideri l'esempio in Tabella 5, con 3 job e 4 macchine. Inizialmente, $M_0 = \emptyset$. Il grafo $\mathcal{G}(\emptyset)$ (Figura 11) non contiene archi disgiuntivi, e il makespan $C_{max}(\emptyset)$ è semplicemente pari al massimo tempo di processamento tra i job, ossia 22 (ottenuto in corrispondenza del job 1 e del job 2).*

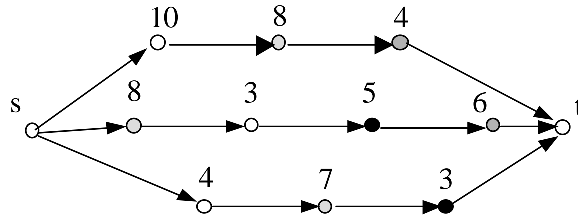

Figura 11: Il grafo $\mathcal{G}(\emptyset)$. Le macchine sono indicate con diverse ombreggiature (1: bianco, 2: grigio chiaro, 3: grigio scuro, 4: nero)

Passo 1. *Consideriamo le quattro macchine singolarmente. Per la macchina 1, si ha l'istanza del problema $1|r_j|L_{max}$ in Tabella 6. Si noti ad esempio che il task 2 non può iniziare prima dell'istante 8 (poichè è preceduto da un task di durata 8), e se si vuole mantenere il makespan al valore 22, non può terminare oltre l'istante 22-11=11. Risolvendo questa istanza di $1|r_j|L_{max}$ con tre job, si ha che la sequenza ottima è $1, 2, 3$ che dà $L_{max}(1) = 5$. Analogamente possiamo associare istanze alle altre tre macchine. Quella associata alla macchina 2 è riportata in Tabella 7, e ha come soluzione ottima la sequenza $2, 3, 1$ di valore $L_{max}(2) = 5$. In modo analogo possono derivarsi le istanze associate alle macchine 3 e 4. Queste danno come valore della soluzione ottima $L_{max}(3) = 4$ e $L_{max}(4) = 0$. Dunque, vi sono due macchine per le quali $L_{max}(k)$ ha il massimo valore,*

[3]Questo esempio è tratto dal libro di M.Pinedo, *Scheduling*, Wiley and Sons, 1995.

vale a dire la macchina 1 e la macchina 2. Una qualsiasi di queste può essere scelta come macchina bottleneck, ad esempio poniamo $k^* = 1$. *Poniamo allora* $M_0 := \{1\}$ *e otteniamo il nuovo grafo* $\mathcal{G}(M_0)$ *fissando il sequenziamento ottimo di* $1|r_j|L_{max}$ *(Figura 12). Si noti che poichè* $L_{max}(1) = 5$, *la lunghezza del percorso critico è aumentata di 5, per cui* $C_{max}(\{1\}) = C_{max}(\emptyset) + L_{max}(1) = 27$. *Il primo passo dell'algoritmo è completato.*

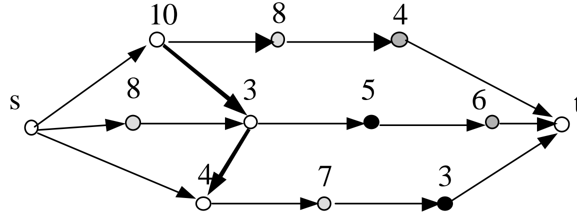

Figura 12: Il grafo $\mathcal{G}(\{1\})$ ottenuto dopo aver sequenziato la macchina 1.

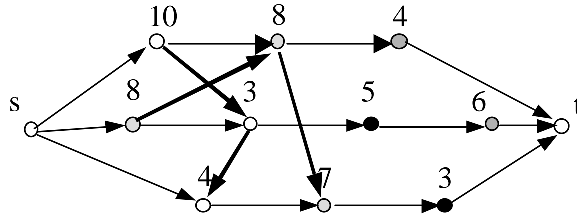

Figura 13: Il grafo $\mathcal{G}(\{1, 2\})$ ottenuto dopo aver sequenziato anche la macchina 2.

Passo 2. *Partendo da* $\mathcal{G}(\{1\})$, *consideriamo nuovamente un'istanza di* $1|r_j|L_{max}$ *per ciascuna delle macchine non sequenziate, vale a dire le macchine 2, 3 e 4. Per la macchina 2 otteniamo l'istanza in Tabella 8. Si noti in particolare come la presenza in* $\mathcal{G}(\{1\})$ *degli archi* $[11] \rightarrow [12]$ *e* $[12] \rightarrow [13]$ *faccia aumentare la release date del task 3 (da 4 a 17) mentre le due date sono aumentate (a causa dell'aumento del makespan). Lo schedule ottimo è* $2, 1, 3$ *con valore* $L_{max}(2) = 1$. *Formulando l'istanza corrispondente alla macchina 3, si ottiene la Tabella 9. Le due sequenze* $1, 2$ *e* $2, 1$ *sono ambedue ottime e hanno valore* $L_{max}(3) = 1$. *Si può invece verificare che* $L_{max}(4) = 0$. *Di nuovo dunque abbiamo una situazione di parità tra due macchine, e arbitrariamente scegliamo la macchina 2 come*

macchina bottleneck. Dunque, aggiungiamo la macchina 2 a M_0*, e sul grafo andiamo ad aggiungere i due archi* $[22] \rightarrow [21]$ *e* $[21] \rightarrow [23]$ *(Figura 13). Il nuovo valore del makespan è* $C_{max}(\{1,2\}) + L_{max}(2) = 28$.

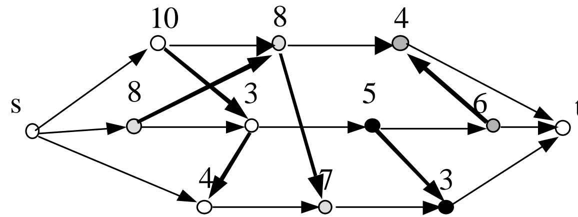

Figura 14: Il grafo $\mathcal{G}(\{1,2,3,4\})$ al termine dell'algoritmo.

Passi successivi. *Al passo 3, rimangono da sequenziare le macchine 3 e 4. Le istanze a esse associate danno ambedue valore* $L_{max} = 0$*, dunque una qualunque di esse può essere sequenziata di conseguenza. Comunque venga fatta questa scelta, si può facilmente verificare che all'ultimo passo, l'altra macchina può essere sequenziata ancora in modo da avere* $L_{max} = 0$ *e dunque in definitiva la soluzione trovata dall'algoritmo shifting bottleneck ha valore* $C_{max}(\{1,2,3,4\}) = 28$*. La soluzione è raffigurata in Figura 14 e in Tabella 10.*
□