UNIVERSITÀ DEGLI STUDI DI PISA

Facoltà di Scienze Matematiche Fisiche e Naturali

Corso di laurea in Informatica

Algoritmi di ricerca locale basati su grafi di miglioramento per il problema di assegnamento di lavori a macchine

Relatrice:
Prof. Maria Grazia Scutellà

Controrelatore:
Dott. Paolo Ferragina

Candidato
Emiliano Necciari

Anno Accademico 1998/99

Indice

Introduzione

In molte applicazioni in cui si devono risolvere problemi di Ottimizzazione Combinatoria, un ruolo rilevante viene svolto dagli algoritmi di Ricerca Locale, che consentono spesso di risolvere in tempi rapidi problemi "difficili" di grandi dimensioni. Data una soluzione ammissibile per il problema in esame, tali algoritmi utilizzano un'opportuna funzione *intorno* che definisce un insieme di soluzioni ammissibili "vicine" a quella corrente, per cercare di migliorare tale soluzione (fase di esplorazione dell'intorno), in base alla funzione obiettivo del problema considerato.

In questo lavoro di tesi, è stato affrontato il problema di individuare algoritmi di ricerca locale particolari, basati su tecniche "multiscambio", per un problema di ottimizzazione combinatoria classico, il problema di assegnamento di lavori a macchine. Tale problema consiste nell'assegnare n lavori, di cui sono noti i tempi di lavorazione $d_i, i = 1, \ldots, n$, ad m macchine identiche, in modo da minimizzare il tempo di completamento dei lavori ("makespan"). Gli algoritmi di ricerca locale basati su tecniche "multiscambio" utilizzano una definizione di intorno più sofisticata di quelle solitamente adottate in letteratura, sulla base della seguente proprietà: per molti problemi di Ottimizzazione Combinatoria, tra cui il problema di assegnamento di lavori a macchine, le soluzioni ammissibili possono essere rappresentate mediante una partizione di un insieme di oggetti in sottoinsiemi, aventi spesso particolari proprietà strutturali.

Le tecniche "multiscambio" definiscono intorni che tendono a coinvolgere un numero rilevante di tali sottoinsiemi, mentre le tecniche più "classiche"

tendono a coinvolgere pochi sottoinsiemi, effettuando quindi spesso una ricerca locale non approfondita. Per evitare l'enumerazione esplicita delle soluzioni intorno, inoltre, gli algoritmi basati su tecniche "multiscambio" riescono spesso a rappresentare, in modo compatto, l'insieme degli scambi multipli ammissibili mediante un opportuno grafo, detto di miglioramento. Spesso, la ricerca di una buona soluzione nell'intorno, vale a dire una soluzione migliore di quella corrente, corrisponde a cercare un ciclo particolare, di costo negativo, in tale grafo di miglioramento.

In letteratura, algoritmi di ricerca locale del tipo descritto, vale a dire basati su grafi di miglioramento per individuare scambi multipli, che migliorano la soluzione corrente, sono stati proposti solo per il problema di "Vehicle Routing" (VRP) e per alcune sue varianti (Thompson e Psaraftis, 1993), e per il problema dell'albero di copertura capacitato di costo minimo (CMST) (Ahuja et al., 1998).

Tali algoritmi descritti nei Capitoli 1 e 2, rispettivamente; tali capitoli contengono anche le formulazioni dei problemi e una rassegna delle principali tecniche di ricerca locale proposte in letteratura per VRP e CMST.

Le tecniche di ricerca locale di tipo "multiscambio" sono state estese al problema di assegnamento di lavori a macchine. È stata infatti proposta una funzione intorno basata su un opportuno grafo di miglioramento. È stato dimostrato che, per il problema in esame, la ricerca di una soluzione che migliori quella corrente nell'intorno proposto corrisponde a cercare un opportuno ciclo o cammino nel grafo di miglioramento, detto K-ciclo (cammino) orientato disgiunto. Trovare tali strutture è tuttavia un problema "difficile". Sono stati proposti allora due algoritmi euristici per la loro individuazione: il primo è una variante di un algoritmo classico per l'individuazione di un albero di cammini minimi, mentre il secondo è stato ottenuto a partire da un algoritmo per l'individuazione di un albero di cammini "bottleneck".

Questo ha portato alla definizione di due algoritmi di ricerca locale, MS_SPT e MS_BPT, descritti nel Capitolo 3. Per ognuno di tali algoritmi sono state inoltre proposte due versioni, che differiscono per la politica

con cui viene gestita la lista dei nodi candidati, e per come è definito il costo degli archi del grafo di miglioramento.

È stata anche condotta un'ampia sperimentazione, descritta nel Capitolo 4, in cui gli algoritmi proposti sono stati implementati e testati su diverse tipologie di istanze del problema al fine di valutare la bontà degli approcci proposti da un punto di vista computazionale. La bontà della soluzione ottenuta è stata valutata stimando l'errore relativo commesso rispetto alla soluzione "ideale", data da $\sum_i \frac{d_i}{m}$.

Poiché la bontà di una soluzione ottenuta mediante algoritmi di ricerca locale dipende fortemente dalla soluzione ammissibile iniziale, gli algoritmi proposti sono stati testati a partire da soluzioni iniziali ottenute mediante diverse euristiche ("List Scheduling", "Longest Processing Time", "Multifit").

Alcune istanze del problema, di tipo uniforme, sono state generate in base ai suggerimenti raccolti dalla letteratura (vedi Hübscher e Glover, 1994). Sulle istanze di questo tipo gli algoritmi proposti ottengono un errore relativo medio inferiore a 10^{-5}. Tali istanze sono tuttavia da considerarsi "facili", in quanto l'euristica "Longest Processing Time" riesce da sola ad ottenere soluzioni con errore relativo medio dell'ordine di $10^{-3}/10^{-4}$.

Sono state perciò proposte due tipologie di istanze del problema più "difficili". Le due tipologie **alm-1** e **alm-2**, differiscono per l'ampiezza dell'intervallo all'interno del quale vengono generate le durate dei lavori, e sono state ottenute dopo un lungo processo di raffinamento di un generatore sviluppato in questo lavoro di tesi. Il processo di raffinamento è stato condotto utilizzando come algoritmo "benchmark" proprio l'euristica "Longest Processing Time". La sperimentazione ha mostrato che in tempi rapidi, l'errore relativo medio commesso dagli algoritmi sulle istanze di tipo **alm-1** è sempre inferiore a 10^{-3}, mentre sulle istanze di tipo **alm-2** è inferiore a 10^{-5}. La soluzione ammissibile iniziale è, inoltre, notevolmente migliorata dalla maggior parte degli algoritmi testati.

Gli algoritmi di ricerca locale proposti per il problema di assegnamento di lavori a macchine sembrano quindi essere uno strumento efficace per l'individuazione di buone soluzioni ammissibili per il problema. In tempi rapidi, infatti, tali algoritmi sono risultati in grado di migliorare, anche notevolmente, soluzioni ottenute mediante classici algoritmi di costruzione, che, da risultati apparsi in letteratura, solitamente mostrano un buon comportamento, soprattutto su alcune famiglie di istanze del problema.

Ci proponiamo di ampliare ulteriormente la sperimentazione, utilizzando anche altre euristiche per costruire la soluzione ammissibile iniziale.

1

Il problema di "Vehicle Routing"

1.1 Introduzione

Il problema consiste nel trovare un insieme di percorsi, per altrettanti veicoli che partono tutti da uno stesso deposito, in modo da servire un insieme di clienti minimizzando la lunghezza totale della strada percorsa. Più formalmente, siano dati un insieme di clienti $C = \{1, \ldots, n\}$, un insieme di veicoli $V = \{1, \ldots, m\}$, ognuno con una capacità di carico Q_k per $k \in V$, ed un deposito (si assume che il deposito coincida con il cliente 1); siano inoltre q_i la richiesta di bene del cliente i e c_{ij} il costo per andare dal cliente i al cliente j. Obiettivo del problema del "vehicle routing"(VRP) è individuare, per ogni veicolo, un percorso o ciclo che parta dal deposito e ritorni al deposito stesso, in modo che tutti i clienti siano visitati, la capacità di ogni veicolo sia rispettata, e sia minimizzato il costo totale dei percorsi. Consideriamo la seguente formulazione, di Fisher e Jaikumar (1978):

$$x_{ijk} = \begin{cases} 1 & \text{se il veicolo } k \text{ visita } j \text{ dopo } i \\ 0 & \text{altrimenti} \end{cases}$$

$$y_{ik} = \begin{cases} 1 & \text{se il cliente } i \text{ è visitato dal veicolo } k \\ 0 & \text{altrimenti} \end{cases}$$

$$\min \sum_{(i,j)\in C\times C} c_{ij} \sum_{k\in V} x_{ijk} \tag{1.1}$$

$$\sum_{k\in V} y_{ik} = \begin{cases} 1 & \text{se } i = 2, \ldots, n \\ m & \text{se } i = 1 \end{cases} \tag{1.2}$$

$$\sum_{i\in C} q_i y_{ik} \leq Q_k \qquad \forall k \in V \tag{1.3}$$

$$\sum_{j\in C} x_{ijk} = \sum_{j\in C} x_{jik} = y_{ik} \qquad \forall i \in C,\ \forall k \in V \tag{1.4}$$

$$\sum_{i,j\in S} x_{ijk} \leq |S| - 1 \qquad \forall S \subseteq \{2, \ldots, n\},\ \forall k \in V \tag{1.5}$$

$$y_{ik} \in \{0, 1\} \qquad \forall i \in C,\ \forall k \in V \tag{1.6}$$

$$x_{ijk} \in \{0, 1\} \qquad \forall i, j \in C,\ \forall k \in V \tag{1.7}$$

I vincoli (1.3) assicurano che la richiesta totale di bene dei clienti assegnati al veicolo k-esimo non superi la capacità del veicolo, Q_k. I vincoli (1.4) assicurano che ogni cliente, diverso dal deposito, sia visitato da un solo veicolo. Il numero di veicoli che visitano un nodo è quindi 1 nel caso di un cliente, e m nel caso del deposito. Questa proprietà è garantita dai vincoli (1.2). I vincoli (1.5) assicurano che nella soluzione non siano presenti sottocicli.

Nel 1980 Christofides, Mingozzi e Toth hanno dato una diversa formulazione per il problema VRP. In questa formulazione si suppone che i veicoli siano ordinati in modo non crescente relativamente alle loro capacità. Vale a dire, $Q_1 \geq Q_2 \geq \ldots \geq Q_m$. Inoltre:

- $\{1, \ldots, \hat{r}\}$ denota l'insieme di tutti i percorsi ammissibili per il veicolo 1;
- M_r denota l'insieme dei clienti che si trovano lungo il percorso r, e c_r indica il costo di tale percorso, $r = 1, \ldots, \hat{r}$;
- $N_i = \{r \mid i \in M_r\}$ rappresenta l'insieme dei percorsi a cui appartiene il cliente i, $i = 1, \ldots, n$;
- r_k indica il minimo r tale che $K_r \leq Q_k$, dove $K_r = \sum_{i \in M_r} q_i$ (si assume che i percorsi siano ordinati in ordine non crescente rispetto a K_r); r_k indica cioè il primo percorso, in base all'ordinamento sopra indicato, che risulta essere ammissibile per il veicolo k, $k = 1, \ldots, m$;
- infine $r_{m+1} = \hat{r} + 1$.

Introducendo le variabili binarie:

$$y_r = \begin{cases} 1 & \text{se il percorso } r \text{ appartiene alla soluzione} \\ 0 & \text{altrimenti} \end{cases}$$

si ottiene la seguente formulazione:

$$\min \sum_{r=1}^{\hat{r}} c_r y_r \tag{1.8}$$

$$\sum_{r \in N_i} y_r = 1 \qquad i = 2, \ldots, n \tag{1.9}$$

$$\sum_{r=1}^{r_{k+1}-1} y_r \leq k \qquad k = 1, \ldots, m,\ r_k \neq r_{k+1} \tag{1.10}$$

$$\sum_{r=1}^{\hat{r}} y_r = m \tag{1.11}$$

$$y_r \in \{0, 1\} \qquad r = 1, \ldots, \hat{r} \tag{1.12}$$

I vincoli (1.9) assicurano che ogni cliente sia visitato una sola volta. I vincoli (1.11) assicurano che si utilizzino tanti percorsi per quanti sono i veicoli. I vincoli (1.10) sono i vincoli di capacità.

Esempio 1.1. *Siano* $\{1, 2\}$ *due veicoli con capacità* $Q_1 = 4$, $Q_2 = 2$. *Siano* $\{1, 2, 3, 4, 5\}$ *i percorsi ammissibili per il veicolo* 1 *e siano* $K_1 = 4$, $K_2 = 3$, $K_3 = 2$, $K_4 = 2$ *e* $K_5 = 1$. *Il minimo* r *per cui vale* $K_r \leq Q_2$ *è* 3*; quindi* $r_2 = 3$.

$$\overbrace{\underbrace{1, 2}_{\text{possiamo scegliere al più 1 percorso}}, 3, 4, 5}^{\text{tra questi percorsi possiamo sceglierne al più 2}}$$

I vincoli (1.10) *indicano che possiamo selezionare al più un percorso in* $\{1, 2\}$*, da assegnare al veicolo 1, ed al più 2 percorsi in* $\{1, 2, 3, 4, 5\}$.

Se non consideriamo i vincoli di capacità associati ai veicoli, il problema VRP può essere interpretato come un problema del commesso viaggiatore multiplo (MTSP). MTSP è una generalizzazione del problema del commesso viaggiatore (TSP), in cui è presente più di un commesso viaggiatore. In questo problema sono presenti m viaggiatori che devono visitare gli n nodi di un grafo, partendo da un deposito e tornando al deposito stesso, in modo che la distanza totale percorsa sia minimizzata. Ogni nodo, ad eccezione

del deposito, deve essere visitato da un unico commesso. Il deposito deve essere attraversato da tutti i commessi. MTSP può essere trasformato in un problema TSP equivalente (vedi Bodin et al., 1983). Dato un problema MTSP, il problema TSP equivalente si ottiene replicando il deposito m volte. Ogni copia del deposito dovrà essere connessa a tutti i clienti, come nel problema originario, cioè mantenendo i costi degli archi originali. Le copie del deposito non sono connesse tra di loro, oppure sono connesse con archi a costo $+\infty$.

Una volta trasformato, il problema può essere risolto mediante un algoritmo per il problema TSP. La soluzione del problema originario è ottenuta effettuando la trasformazione inversa, cioè eliminando le copie del deposito. In Figura 1.1 è illustrata la trasformazione per un'istanza di MTSP, caratterizzato da due commessi e tre clienti.

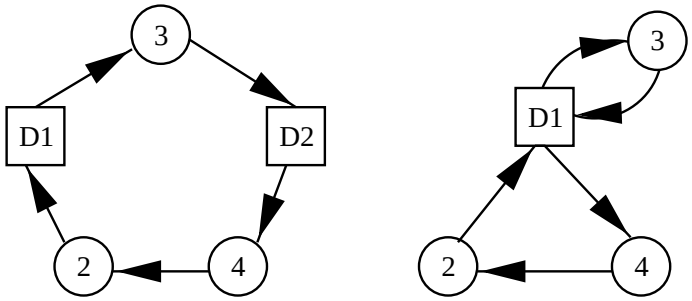

Figura 1.1: Soluzione per un'istanza del problema TSP(a sinistra), e soluzione per l'istanza MTSP corrispondente(a destra)

1.2 Algoritmi di ricerca locale per VRP: l'algoritmo di Thompson e Psaraftis

Sia $G = (N, A)$ il grafo orientato completo che rappresenta un'istanza del problema VRP, avente un nodo per ogni cliente, incluso il deposito (l'arco (i, j) rappresenta la connessione tra il cliente i ed il cliente j, e ad esso viene associato il costo c_{ij}, $\forall i, j \in N$).

Data una soluzione ammissibile, vale a dire un insieme di m cicli in G, aventi il deposito come unico nodo in comune, che copre N, le principali funzioni intorno usate in letteratura per il problema VRP, sono:

- *Relocation.* Selezionati due nodi i e j, appartenenti a due percorsi distinti, gli archi $(i-1,i)$, $(i,i+1)$ e $(j-1,j)$ sono rimpiazzati da $(i-1,i+1)$, $(j-1,i)$ e (i,j), vale a dire il cliente i è spostato al percorso cui appartiene il cliente j. Bisogna osservare che tale scambio può essere effettuato solo se non si violano i vincoli di capacità associati ai veicoli. In Figura 1.2 è riportato un esempio. Un'estensione di questo meccanismo si ha in Gendreau et al. (1992).

- *Exchange.* Gli archi $(i-1,i)$, $(i,i+1)$, $(j-1,j)$ e $(j,j+1)$ sono rimpiazzati da $(i-1,j)$, $(j,i+1)$, $(j-1,i)$ e $(i,j+1)$, vale a dire si prendono due clienti appartenenti a due percorsi diversi, i e j, e li si scambia. In Figura 1.3 è riportato un esempio.

- *Crossover.* Gli archi $(i,i+1)$ e $(j,j+1)$ sono rimpiazzati da $(i,j+1)$ e $(j,i+1)$, vengono cioè distrutti eventuali incroci presenti tra i percorsi. In Figura 1.4 è riportato un esempio. Un *crossover* particolare si ha quando l'arco $(i,i+1)$ è l'ultimo di un percorso e l'arco $(j,j+1)$ è il primo di un altro percorso o viceversa. In tal caso, una mossa nell'intorno *crossover* comporta l'unione dei due percorsi in esame a patto che non si violino i vincoli di capacità associati ai veicoli.

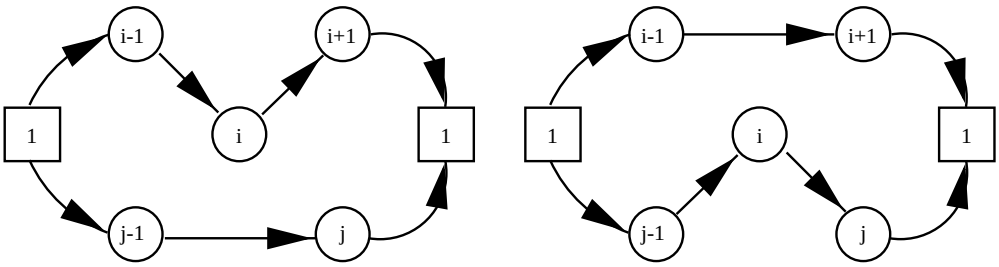

Figura 1.2: Esempio di mossa nell'intorno *Relocation*

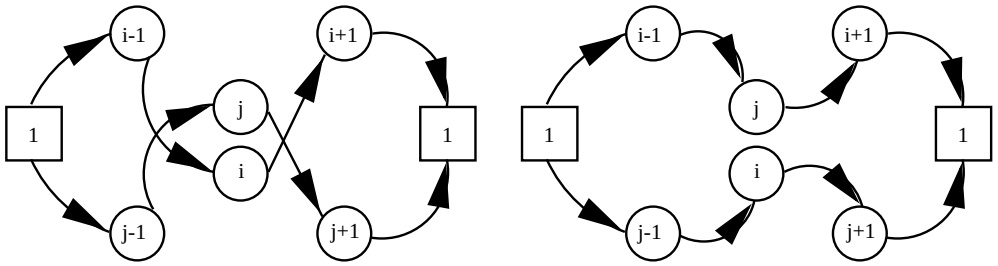

Figura 1.3: Esempio di mossa nell'intorno *Exchange*

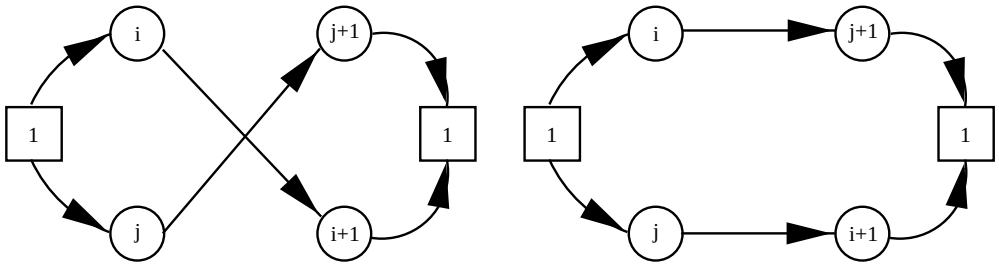

Figura 1.4: Esempio di mossa nell'intorno *Crossover*

Le funzioni intorno sopra citate possono essere interpretate come casi particolari della funzione intorno introdotta da Thompson e Psaraftis (1993), descritta nel seguito.

1.2.1 L'algoritmo per scambi ciclici di Thompson e Psaraftis

Descriviamo la funzione intorno di Thompson e Psaraftis, che si basa sul concetto di *scambio ciclico*. Sia $I = \{I_1, \ldots, I_m\}$ un insieme di percorsi che rappresentano la soluzione ammissibile corrente per il problema VRP. Sia ρ una permutazione di un sottoinsieme $\bar{V}$ di $\{1, \ldots, m\}$.

Uno *scambio ciclico* è il trasferimento simultaneo di un certo numero di nodi clienti dal percorso I_j al percorso $I_{\rho(j)}$ per ogni $j \in \bar{V}$, purché tale scambio sia ammissibile, rispetti cioè le capacità dei veicoli in gioco. Nel

lavoro si prendono in considerazione *scambi ciclici* in cui si ha il trasferimento di k nodi clienti tra il percorso I_j e il percorso $I_{\rho(j)}$ per ogni $j \in \bar{V}$, dove k è fissato. Chiameremo questo tipo di scambi *k-scambi ciclici*. Inoltre, si considerano i casi in cui la permutazione ρ ha cardinalità fissata b, cioè $|\bar{V}| = b$.

Il costo di uno *scambio ciclico* è la variazione della funzione obiettivo dovuta allo *scambio ciclico*. Sia $J = \{J_1, \ldots, J_m\}$ l'insieme dei percorsi dopo aver effettuato uno scambio ciclico a partire dalla soluzione corrente I. Sia $c(I)$ il costo del percorso I. Allora il costo dello *scambio ciclico* è:

$$\sum_{j=1}^{m} [c(J_j) - c(I_j)]$$

L'*intorno* della soluzione ammissibile I è l'insieme delle soluzioni ottenute da I con uno *scambio ciclico*. Diciamo che la soluzione I è ottima(CT-opt) se nell'*intorno* di I non esiste una soluzione J migliore di I(in termini della funzione obiettivo), vale a dire tutti gli *scambi ciclici* relativi a I hanno un costo non negativo. Analoghe definizioni si hanno nel caso di *k-scambi ciclici*.

Thompson e Orlin (1989) hanno sviluppato una metodologia generale di ricerca di *scambi ciclici* di costo negativo. Questa metodologia consiste nell'utilizzare un grafo di miglioramento all'interno del quale si cercano cicli *disgiunti* di costo negativo, dove un ciclo è detto *disgiunto* se i suoi nodi corrispondono a sottoinsiemi di clienti assegnati a veicoli (vale a dire percorsi) distinti nella soluzione corrente: Thompson e Orlin hanno dimostrato infatti che cercare un ciclo disgiunto di costo negativo nel grafo di miglioramento è equivalente a cercare uno *scambio ciclico* di costo negativo nell'intorno della soluzione corrente.

Più formalmente, data la soluzione I e fissato k, il *grafo di miglioramento* utilizzato per individuare scambi ciclici di costo negativo è $G^k(I) =$

$(N^k(I), A^k(I))$, dove:

$$N^k(I) = \{\text{insiemi di } k \text{ clienti serviti da uno stesso veicolo }\}$$
$$A^k(I) = \{(v,w) \in N^k(I) \times N^k(I) \mid I(v) \neq I(w), \text{ il percorso } \{I(w) \cup v \setminus w\}$$
$$\text{è ammissibile per il veicolo cui era assegnato}$$
$$I(w) \text{ nella soluzione corrente } I\}$$

($I(v)$ rappresenta il percorso a cui è stato assegnato l'insieme di clienti v nella soluzione corrente I).

Ad ogni arco $(v,w) \in A^k(I)$ è associato un costo α_{vw}, che indica la variazione del costo del percorso $I(w)$ dovuta all'inserimento dell'insieme di nodi v e alla rimozione dell'insieme di nodi w. Indicando con $c(I(w) \cup v \setminus w)$ il costo minimo di un ciclo che include i nodi in $\{I(w) \cup v \setminus w\}$, allora

$$\alpha_{vw} = c(I(w) \cup v \setminus w) - c(I(w)).$$

In Figura 1.5 è riportato un esempio relativo a un'istanza di problema VRP. In questo problema, ogni cliente richiede una unità di bene, il numero dei veicoli è 4 e ognuno di essi ha capacità pari a 5. La soluzione corrente è $I = \{I_1, I_2, I_3, I_4\}$ con $I_1 = \{a1, a2, a3, a4, a5\}$, $I_2 = \{b1, b2, b3, b4\}$, $I_3 = \{c1, c2, c3\}$ e $I_4 = \{d1, d2, d3, d4\}$. In Figura 1.6 è riportata la soluzione ottenuta da quella riportata in Figura 1.5 in seguito ad un scambio ciclico con $k = 2$ e $\rho = (1, 2, 3)$. Lo scambio consiste nel muovere i clienti $\{a1, a3\}$ dal percorso I_1 al percorso I_2, i clienti $\{b1, b5\}$ dal percorso I_2 al percorso I_3 e i clienti $\{c2, c3\}$ dal percorso I_3 al percorso I_1.

Dalla definizione del grafo di miglioramento $G^k(I)$ è chiaro che, per costruzione, un ciclo disgiunto di costo negativo in $G^k(I)$ corrisponde ad uno scambio ciclico di costo negativo relativo alla soluzione corrente I.

La ricerca di cicli disgiunti di costo negativo in $G^k(I)$ è comunque complessa, per tre motivi fondamentali.

Il primo motivo è che il calcolo dei costi degli archi in $G^k(I)$ è un problema NP-hard (vedi Thompson e Orlin, 1989). Infatti, il costo di un arco (v,w) in

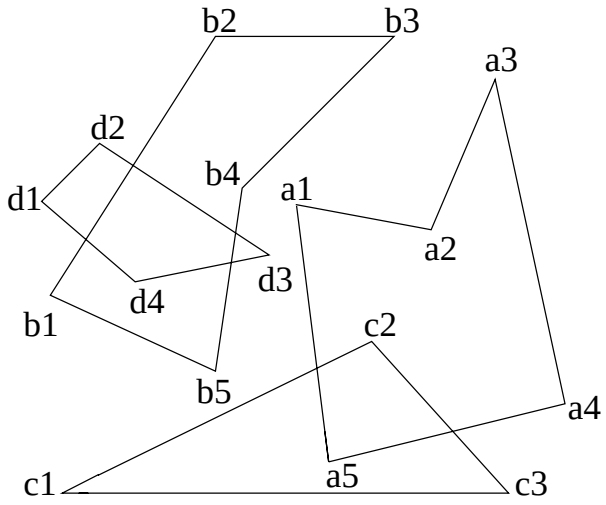

Figura 1.5: Istanza di VRP

$G^k(I)$ rappresenta la variazione nella funzione obiettivo dovuta alla rimozione da $I(w)$ dell'insieme di nodi w e al simultaneo inserimento dell'insieme di nodi v, vale a dire equivale a risolvere un problema di TSP nel sottografo di G indotto dai nodi in $\{I(w) \cup v \setminus w\}$.

Il secondo motivo è legato al numero di archi del grafo $G^k(I)$, che può essere esponenziale rispetto alle dimensioni dell'input.

Il terzo motivo, che complica il tutto, è il fatto che cercare cicli disgiunti di costo negativo nel grafo di miglioramento è un problema NP-hard (vedi Thompson e Orlin, 1989).

Thompson e Psaraftis, allora, hanno sviluppato un'euristica per il problema dell'individuazione di un ciclo disgiunto di costo negativo in $G^k(I)$. Innanzitutto, i costi degli archi del grafo di miglioramento vengono calcolati in modo approssimato mediante una funzione polinomiale in tempo; inoltre, la ricerca di un ciclo disgiunto di costo negativo avviene a sua volta mediante un algoritmo di ricerca locale, in un intorno di dimensioni ristrette.

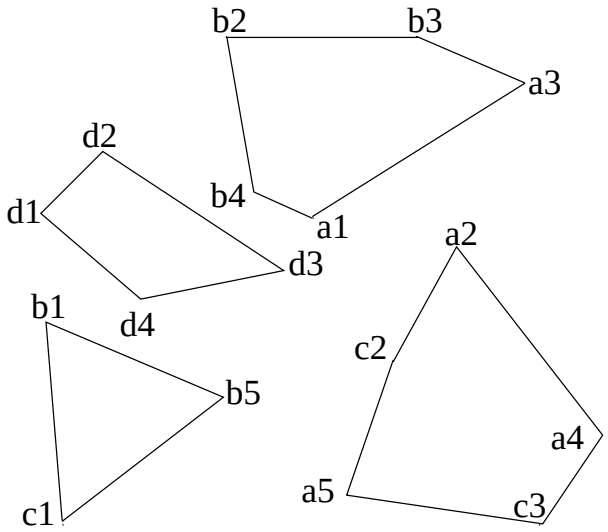

Figura 1.6: Soluzione ottenuta da quella della Figura 1.5 dopo aver effettuato uno scambio ciclico

Calcolo dei costi

La funzione che calcola i costi degli archi è la seguente. Per ogni arco (v, w), il costo α_{vw} si ottiene rimuovendo l'insieme di nodi w dal percorso a cui era stato assegnato; poi si cerca un punto del percorso in cui inserire l'insieme di nodi v a costo minimo. Gli *scambi ciclici* che usano questa funzione costo vengono detti *scambi ciclici LCI*. Una soluzione ammissibile è ottima (LCI CT-opt) se il suo valore non può essere migliorato da nessuno *scambio ciclico LCI*. In riferimento a quanto detto, valgono le seguenti proprietà:

1. Ogni soluzione CT-opt è anche LCI CT-opt, ma non vale il viceversa.

2. Se $k = 1$ e la permutazione ρ ha cardinalità 2, un *k-scambio ciclico* corrisponde ad una mossa nell'intorno *Exchange*; se uno dei due insiemi di nodi corrisponde ad un sottoinsieme di nodi fittizio, un *k-scambio ciclico* corrisponde ad una mossa nell'intorno *Relocation*.

3. La funzione costo semplifica il calcolo dei costi degli archi del grafo di miglioramento. Nel caso in cui il grafo di miglioramento utilizzato è

$G^k(I)$ con $k = 1$, l'algoritmo utilizzato per il calcolo dei costi α_{vw}, per $(v, w) \in A^k(I)$, ha complessità computazionale in tempo $O(n^2)$. Per la dimostrazione si rimanda a (Thompson e Psaraftis, 1993).

Ricerca di scambi ciclici di costo negativo

La ricerca di uno *scambio ciclico* nell'intorno può avvenire in due modi, con l'ausilio di $G^k(I)$.

Il primo modo consiste nel limitare la ricerca di cicli disgiunti di costo negativo in $G^k(I)$, limitando ad esempio la cardinalità della permutazione ρ a 2 o a 3, e cercando poi di migliorare i cicli ottenuti. Un ciclo può essere migliorato cercando di aumentare la sua lunghezza oppure cercando di migliorare il suo costo.

Il secondo modo consiste nel generare solo una parte del grafo di miglioramento $G^k(I)$. Ciò riduce la complessità computazionale in tempo sia per il calcolo dei costi degli archi che per la ricerca dei cicli disgiunti di costo negativo. In particolare, invece di considerare ogni insieme di k clienti serviti da uno stesso veicolo, si possono considerare solo insiemi di k clienti serviti consecutivamente da uno stesso veicolo. Ad esempio, se un veicolo serve i clienti $\{5, 7, 12, 3, 8\}$ in questo ordine, allora $G^3(I)$ avrà come nodi, relativi a quel veicolo, solo gli insiemi $\{5, 7, 12\}$, $\{7, 12, 3\}$ e $\{12, 3, 8\}$. Ciò riduce il numero dei nodi in $G^k(I)$ da $O(mn^k)$ a $O(mn)$; quindi, il numero degli archi diventa $O(m^2n^2)$, che è indipendente da k.

Possiamo ora presentare formalmente l'algoritmo di Thompson e Psaraftis.

Algoritmo di Thompson e Psaraftis

procedure $CTVRP(I)$
  begin
    $I := SoluzioneIniziale();$
    $CalcolaPercorso\lambda\text{-}opt(I_j), \forall I_j \in I;$
    $Crea(G^k(I));$
    while $EsisteCicloDisgiuntoNegativoIn(G^k(I))$ **do**
        $C := TrovaCicloDisgiuntoIn(G^k(I));$
        $Aggiorna(I, C);$
        $CalcolaPercorso\lambda\text{-}opt(I_j), \forall I_j \in I;$
        $Aggiorna(G^k(I));$
    end
  end.

Nell'algoritmo $CTVRP(I)$ si utilizza la procedura $CalcolaPercorso\lambda\text{-}opt(I_j)$ per ottenere una buona soluzione iniziale e per migliorare i percorsi generati dall'algoritmo di ricerca locale. Questa procedura implementa un'euristica sviluppata per il problema TSP. Data la soluzione ammissibile I, la procedura considera un opportuno intorno di I e lo esplora nel tentativo di migliorare la soluzione corrente. L'intorno di I è costituito da tutte le soluzioni I' che differiscono da I per λ archi, con λ fissato. L'euristica termina quando non riesce a migliorare la soluzione corrente. Il percorso ottenuto è detto λ-*opt*. Rimandiamo a Lin (1965) e Lin e Kernighan (1973) per una descrizione più dettagliata di questa famiglia di algoritmi.

Analisi di complessità nel caso pessimo

In Thompson e Psaraftis (1993) viene fornita un'analisi di complessità computazionale dell'algoritmo proposto nel caso pessimo. Sia R_{wc} il rapporto tra il costo della soluzione trovata dall'algoritmo e il costo della soluzione ottima. Usando la metrica Euclidea, Thompson e Psaraftis dimostrano che l'algoritmo, nel caso più sfavorevole, restituisce una soluzione m (dove m è il numero dei veicoli) volte peggiore della soluzione ottima, per ogni k fissato. Inoltre, quando l'algoritmo è applicato a problemi VRP con più depositi, R_{wc} non è

limitato superiormente. In Figura 1.7 è riportato un esempio in cui si hanno $m = 4$ veicoli con capacità $m(k+1)$ e $n = 7$ clienti. I clienti $\{a, b, c, d\}$, che richiedono $k+1$ unità di bene, sono a distanza 1 dal deposito (coincidente con il nodo 1) e a distanza δ(con $\delta \ll \frac{2\pi}{m}$) tra loro, mentre i clienti $\{e, f, g\}$, che richiedono $(m-1)(k+1)$ unità di bene, sono a distanza δ dal deposito (negli esempi menzionati, il costo coincide con la distanza). La soluzione trovata dall'algoritmo è quella in cui ogni veicolo visita un cliente che sta a distanza 1 e il cliente, a distanza δ, che sta sulla linea che congiunge il primo cliente servito al deposito. Quindi il costo di questa soluzione è $2m$. La soluzione ottima è invece quella in cui un veicolo visita i clienti $\{a, b, c, d\}$, mentre gli altri tre veicoli visitano ognuno un cliente a distanza δ. Il costo di questa soluzione è $2 + 3(m-1)\delta$. Da ciò si deduce che R_{wc} tende ad m quando δ tende a 0. In Figura 1.8 è riportato un esempio di istanza di problema VRP con 2 depositi, in corrispondenza del nodo 1 e del nodo 5. In ogni deposito è presente un veicolo con capacità $3(k+1)$. Ogni nodo diverso dal deposito richiede $k+1$ unità di bene. L'algoritmo di ricerca locale individua i percorsi $\{1, 2, 3, 4, 1\}$ e $\{5, 6, 7, 8, 5\}$ che costano $4(L+\delta)$. Se $2\delta \leq L$ allora i percorsi $\{1, 2, 7, 8, 1\}$ e $\{5, 6, 3, 4, 5\}$ sono ottimi, con costo 12δ. Da ciò si deduce che, quando δ tende a 0, R_{wc} tende a $+\infty$.

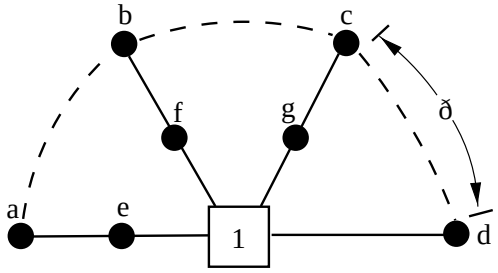

Figura 1.7: Esempio di istanza di problema VRP in cui $R_{wc} = m$.

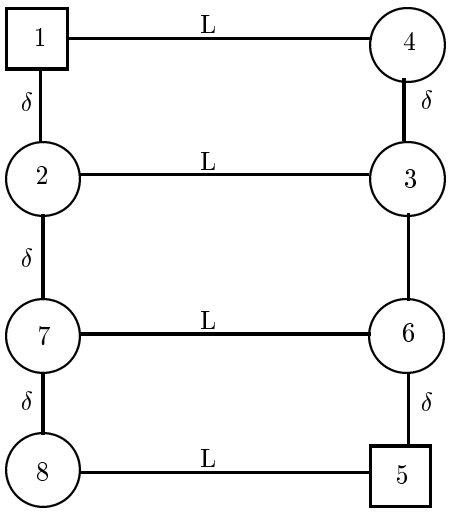

Figura 1.8: Esempio di istanza di problema VRP con 2 depositi e R_{wc} non superiormente limitato.

1.2.2 Risultati Sperimentali

Thompson e Psaraftis hanno implementato tre versioni dell'algoritmo proposto, per risolvere tre varianti del problema VRP. Le tre varianti del problema per le quali è stata condotta la sperimentazione sono:

- il problema VRP classico,
- il problema di "vehicle routing and scheduling", con vincoli di precedenza;
- il problema di "vehicle routing and scheduling", con "time windows".

La sperimentazione effettuata sulle istanze di tipo VRP classico è stata condotta confrontando l'algoritmo proposto dagli autori con altri algoritmi esistenti in letteratura. Dai risultati riportati si nota che l'algoritmo di Thompson e Psaraftis si comporta come l'euristica di assegnamento generalizzato di Fisher e Jaikumar (1982), e sempre meglio delle euristiche proposte da Clark e Wright (1964), da Gillett e Miller (1974), da Lin (1965) e da Christofides et al. (1981).

Per quanto riguarda il problema di "vehicle routing and scheduling" con vincoli di precedenza, la sua peculiarità è che ogni cliente richiede un servizio che consiste di una fase di prelievo e una fase di consegna. La fase di prelievo deve avvenire prima della fase di consegna (per maggiori dettagli si rimanda a Bodin et al. (1983)).

La sperimentazione è stata condotta valutando il miglioramento, in termini della funzione obiettivo, della soluzione ottenuta con l'algoritmo proposto rispetto alla soluzione iniziale. Dai risultati prodotti si nota che questo miglioramento è maggiore del 60% e in molti casi è vicino al 100%. Secondo gli autori un miglioramento così elevato è dovuto, in larga parte, alla struttura particolare di questo problema.

La terza variante del problema VRP considerata dagli autori in fase di sperimentazione è quella con "time windows". La caratteristica principale è che i veicoli possono soddisfare le richieste di bene dei clienti solo durante determinati intevalli di tempo, detti "time windows". Questi intervalli di tempo devono essere rispettati, vale a dire un veicolo deve visitare un cliente sempre all'interno dell'intervallo di tempo prestabilito. Quindi, per calcolare il tempo speso da un veicolo nell'effettuare un percorso, dobbiamo considerare il tempo in cui il veicolo deve sostare da un cliente (ad esempio per lasciare la quantità di bene richiesta), il tempo impiegato nel viaggio per andare da un cliente ad un altro e un'eventuale tempo di attesa prima che il cliente possa essere servito. Gli obiettivi di questo problema sono molteplici. Il primo è trovare un numero di percorsi ammissibili minimizzando il numero di veicoli utilizzati. Il secondo è minimizzare il tempo impiegato nel servire tutti i clienti assegnati ad uno stesso veicolo. Il terzo obiettivo è minimizzare la totalità della strada percorsa. Per maggiori informazioni riguardo al problema si rimanda a Solomon (1987).

Durante la sperimentazione, relativa a tale variante del problema VRP, l'algoritmo proposto è stato confrontato con le euristiche di Solomon (1987),

di Solomon et al. (1988) e Backer e Schaffer (1986). Nel 50.9% dei casi l'algoritmo di Thompson e Psaraftis ha ottenuto la soluzione migliore. Dai dati riportati si nota che l'algoritmo proposto è sempre in grado di migliorare la soluzione iniziale.

2

Il problema dell'Albero di copertura capacitato di costo minimo

2.1 Introduzione

Sia $G = (N, A)$ un grafo non orientato e completo. Sia $N = \{0, 1, \ldots, n\}$ l'insieme dei nodi di G, dove 0 è un nodo *sorgente*. Ad ogni nodo i, diverso dal nodo sorgente, è associata una domanda di bene $d_i > 0$ intera, che deve essere inviata dal nodo sorgente. Sia A l'insieme degli archi. Ad ogni arco (i, j) è associato un costo c_{ij} e una capacità K che limita la quantità di bene che lo può attraversare.

Una partizione dell'insieme dei nodi $N \setminus \{0\}$ nei sottoinsiemi $R_1, \ldots, R_L$ è detta ammissibile se ogni sottoinsieme soddisfa la condizione $\sum_{i \in R_p} d_i \leq K$, $1 \leq p \leq L$. Sia $T[p]$ un albero di copertura di costo minimo nel sottografo indotto da $R_p \cup \{0\}$, $1 \leq p \leq L$, e sia T^* l'unione di questi alberi. T^* è cioè formato dai sottoalberi $T[1], \ldots, T[L]$ e dagli archi che connettono il nodo 0 alle radici di $T[1], \ldots, T[L]$. Il costo dell'albero T^*, che è di copertura per G, è $\sum_{(i,j) \in T^*} c_{ij}$. T^* è detto albero di copertura capacitato associato alla partizione $R_1, \ldots, R_L$. Esso rappresenta una connessione ad albero tra il nodo sorgente 0 e i nodi in $N \setminus \{0\}$ che garantisce l'invio di d_i unità di bene dalla sorgente ad ogni nodo $i \neq 0$, soddisfacendo i vincoli di capacità associati agli archi del grafo.

Il problema di determinare un albero di copertura di costo minimo capacitato (CMST) consiste nell'individuare una partizione dell'insieme di nodi $N \setminus \{0\}$ in modo che il costo dell'albero di copertura associato alla partizione sia minimo. Nel caso particolare in cui la domanda di bene associata ad ogni nodo sia 1, il problema si riduce a trovare un albero di copertura di costo minimo tale che ogni sottoalbero radicato in 0 contenga al massimo K nodi.

2.2 Algoritmi di ricerca locale per CMST: gli algoritmi di Ahuja, Orlin e Sharma

Tra le diverse funzioni *intorno* proposte in letteratura per il problema CMST, citiamo quella di Amberg et al. (1996) e di Sharaia et al. (1995).

Nel lavoro di Amberg et al., l'intorno di una soluzione ammissibile è costituito da tutte le soluzioni ammissibili ottenute da quella corrente effettuando un'operazione di "*node shift*" oppure di "*node exchange*".

Un'operazione di "node shift" consiste nello scegliere un nodo e spostarlo dal sottoalbero cui appartiene ad un altro. Sia i il nodo da spostare dall'insieme di nodi R_h all'insieme R_k; l'operazione avviene calcolando un albero di copertura di costo minimo nel sotto grafo di G indotto dall'insieme di nodi $R_k \cup \{i\} \cup \{0\}$. Inoltre, se il nodo i è un nodo non foglia, allora bisogna anche calcolare un albero di copertura di costo minimo nel sotto grafo di G indotto dall'insieme di nodi $R_h \setminus \{i\} \cup \{0\}$.

Un'operazione di "node exchange" consiste nello scegliere due nodi, appartenenti a due sottoalberi diversi, e scambiarli. Siano $i \in R_h$ e $j \in R_k$ i nodi prescelti per lo scambio; dopo aver effettuato lo scambio bisogna calcolare alberi di copertura di costo minimo nei sottografi di G indotti rispettivamente da $R_h \setminus \{i\} \cup \{0, j\}$ e da $R_k \setminus \{j\} \cup \{0, i\}$.

Nel lavoro di Sharaia et al., l'intorno di una soluzione ammissibile è costituito da tutte le soluzioni ammissibili ottenute da quella corrente effettuando un'operazione di "*cut&paste*". Un'operazione di "cut&paste" consiste nel selezionare un sottoalbero (non necessariamente radicato in 0) e connetterlo ad un altro sottoalbero oppure direttamente alla radice.

La cardinalità degli intorni definiti da Amberg et al. e da Sharaia et al. è $O(n^2)$.

Recentemente, Ahuja et al. hanno suggerito due funzioni intorno basate sulla tecnica degli scambi ciclici proposta da Thompson e Psaraftis per il problema VRP, descritta nel Capitolo 1. La prima permette scambi di singoli nodi tra sottoalberi diversi, mentre la seconda permette scambi di insiemi di nodi tra sottoalberi diversi.

Poiché la cardinalità di un intorno così definito può essere esponenziale rispetto alle dimensioni dell'input, e la sua esplicitazione sarebbe quindi improponibile, analogamente a quanto proposto da Thompson e Psaraftis, Ahu-

ja et al.. utilizzano un grafo, detto *grafo di miglioramento*, per rappresentare gli intorni.

Nel seguito descriveremo le due funzioni intorno e i relativi grafi di miglioramento. Descriveremo quindi gli algoritmi basati su tali funzioni intorno.

Dato un albero di copertura T avente 0 come nodo radice, nel seguito i sottoalberi di T aventi come radice un nodo figlio di 0 verranno chiamati *alberi radicati*. Per ogni nodo i, con $T[i]$ indicheremo l'albero radicato contenente il nodo i, mentre con $S[i]$ indicheremo l'insieme dei nodi contenuti nell'albero radicato $T[i]$.

Dato un insieme di nodi S, con $d(S) = \sum_{i \in S} d_i$ indicheremo la domanda di bene totale in S. Diremo che S è ammissibile se e solo se $d(S) \leq K$. Analogamente, diremo che l'albero radicato $T[i]$ è ammissibile se $\sum_{j \in T[i]} d_j \leq K$, per $i \in N \setminus \{0\}$. Dato un insieme ammissibile S, indicheremo con $c(S)$ il costo di un albero di copertura di costo minimo nel sotto grafo di G indotto da $S \cup \{0\}$.

2.2.1 Una funzione intorno basata sullo scambio di singoli nodi

La prima funzione intorno sviluppata da Ahuja et al. è una generalizzazione di quella proposta da Amberg et al.. Vengono infatti permessi scambi di singoli nodi tra più di un albero radicato. Questi scambi vengono chiamati dagli autori *multi-scambi*. Nel lavoro vengono definiti due tipi di multi-scambio: scambi ciclici e scambi basati su cammino.

Scambi ciclici

Sia T un albero di copertura ammissibile. Uno *scambio ciclico* è una sequenza $i_1 - \ldots - i_r - i_1$, dove i nodi $i_1, \ldots, i_r$ appartengono ad alberi radicati diversi, cioè, $T[i_r] \neq T[i_s]$ per $r \neq s$. Lo scambio ciclico ha il seguente significato: il nodo i_1 viene spostato dall'albero $T[i_1]$ all'albero $T[i_2]$, il nodo i_2 viene spostato dall'albero $T[i_2]$ all'albero $T[i_3]$, così via, fino al nodo i_r che passa

dall'albero $T[i_r]$ all'albero $T[i_1]$. Uno scambio ciclico $i_1 - \ldots - i_r - i_1$ è ammissibile se l'albero ottenuto dopo aver effettuato lo scambio soddisfa i vincoli di capacità. Sia T' l'albero ottenuto dopo aver effettuato lo scambio ciclico. Il costo dello scambio è:

$$\begin{aligned} c(T') - c(T) &= c(\{i_r\} \cup S[i_1] \setminus \{i_1\}) - c(S[i_1]) + \\ &\quad \sum_{p=2}^{r} (c(\{i_{p-1}\} \cup S[i_p] \setminus \{i_p\}) - c(S[i_p])). \end{aligned}$$

In altre parole, per calcolare il costo dello scambio ciclico bisogna calcolare alberi di copertura di costo minimo relativi agli insiemi della nuova partizione. Uno scambio ciclico è uno *scambio di miglioramento* se $c(T') < c(T)$.

Osserviamo che il numero di alberi radicati può aumentare a causa di uno scambio ciclico; questo si verifica quando, calcolando un albero di copertura di costo minimo nel sottografo indotto da $\{0, i_{p-1}\} \cup S[i_p] \setminus \{i_p\}$, $1 \leq p \leq r$ ($p-1 = r$ nel caso $p = 1$), la stella del nodo sorgente ha cardinalità maggiore di 1. Il numero di alberi radicati non può invece mai diminuire.

Scambi basati su cammino

Uno *scambio basato su cammino* è una sequenza $i_1 - \ldots - i_r$, dove i nodi $i_1, \ldots, i_r$, appartengono ad alberi radicati diversi. Lo scambio basato su cammino va interpretato nel modo seguente: il nodo i_1 viene spostato dall'albero $T[i_1]$ all'albero $T[i_2]$, il nodo i_2 viene spostato dall'albero $T[i_2]$ all'albero $T[i_3]$, così via, fino al nodo i_{r-1} che passa dall'albero $T[i_{r-1}]$ all'albero $T[i_r]$. Dopo uno scambio basato su cammino, l'albero $T[i_1]$ perde un nodo mentre l'albero $T[i_r]$ ne guadagna uno. Uno scambio basato su cammino $i_1 - \ldots - i_r$ è ammissibile se l'albero di copertura ottenuto dopo aver effettuato lo scambio, T', è ammissibile. Il costo dello scambio è:

$$\begin{aligned} c(T') - c(T) &= c(S[i_1] \setminus \{i_1\}) + \sum_{p=2}^{r-1} c(\{i_{p-1}\} \cup S[i_p] \setminus \{i_p\}) + \\ &\quad c(S[i_r] \cup \{i_{r-1}\}) - \sum_{p=1}^{r} c(S[i_p]). \end{aligned}$$

Uno scambio basato su cammino è uno *scambio di miglioramento* se $c(T') < c(T)$. Osserviamo che un tale tipo di scambio può fare sia aumentare che diminuire il numero di alberi radicati.

Dato un albero ammissibile T, un albero T' è *vicino* di T se T' è ammissibile per CMST ed è ottenuto da T mediante uno scambio ciclico o uno scambio basato su cammino.

Intorno di una soluzione T è l'insieme contenente tutti i *vicini* di T.

Il Grafo di miglioramento

Data una soluzione ammissibile T, il grafo di miglioramento nel caso di scambio di singoli nodi è un grafo orientato $G^1(T) = (N, A^1)$, dove N è l'insieme dei nodi di G.

L'insieme A^1 viene definito considerando tutte le coppie (i, j) di nodi in N, e inserendo l'arco (i, j) se e solo se: $T[i] \neq T[j]$, e $\{i\} \cup S[j] \setminus \{j\}$ è un insieme di nodi ammissibile. Il costo α_{ij} dell'arco (i, j) è definito nel seguente modo:

$$\alpha_{ij} = c(\{i\} \cup S[j] \setminus \{j\}) - c(S[j]). \tag{2.1}$$

Nel grafo di miglioramento, un ciclo orientato $i_1 - \ldots - i_r - i_1$ è detto *disgiunto* se i sottoalberi $T[i_1], T[i_2], \ldots, T[i_r]$ sono tutti diversi tra loro.

Ahuja et al. dimostrano che esiste una corrispondenza biunivoca tra gli scambi ciclici rispetto alla soluzione ammissibile corrente T e i cicli disgiunti in $G^1(T)$, e che strutture corrispondenti hanno lo stesso costo.

Per calcolare i costi degli archi di $G^1(T)$ si usa (2.1). Assumiamo che il costo $c(S[j])$, per ogni albero radicato $T[j]$, sia conosciuto. Per determinare α_{ij} dobbiamo quindi calcolare $c(\{i\} \cup S[j] \setminus \{j\})$. Il calcolo di $c(\{i\} \cup S[j] \setminus \{j\})$ consiste nel togliere da $T[j]$ il nodo j, aggiungere il nodo i, e calcolare il costo di un albero di copertura di costo minimo nel sottografo indotto dall'insieme di nodi $\{i\} \cup S[j] \setminus \{j\}$. Poiché l'insieme $\{i\} \cup S[j] \setminus \{j\}$ ha $O(K)$ nodi,

possiamo calcolare α_{ij} in tempo $O(K^2)$ per ogni arco (i, j),usando ad esempio l'algoritmo di Prim (vedi Ahuja et al., 1998). Il tempo computazionale impiegato per calcolare tutti i costi degli archi (i, j) è quindi $O(n^2K^2)$.

Il grafo di miglioramento $G^1(T)$ può essere modificato in modo da stabilire una corrispondenza tra scambi basati su cammino e cicli disgiunti di ugual costo. La trasformazione consiste nell'aumentare $G^1(T)$ aggiungendo alcuni nodi e alcuni archi. Precisamente, aggiungiamo un nodo fittizio v e, per ogni albero radicato in T, aggiungiamo uno pseudonodo h. Colleghiamo il nodo v ad ogni nodo $i \in N$ attraverso un arco (v, i) di costo $c(S[i] \setminus \{i\}) - c(S[i])$. Colleghiamo ogni nodo h al nodo v con un arco (h, v) di costo nullo. Colleghiamo, infine, ogni nodo $i \in N$ ad ogni pseudonodo h, se il nodo i non appartiene all'albero radicato $T[h]$ e $\sum_{k \in T[h]} d_k + d_i \leq K$. Un arco (v, i), con v nodo fittizio e i nodo originario, rappresenta l'eliminazione del nodo i dall'albero radicato $T[i]$. Un arco (i, h), con h pseudonodo e i nodo originario, rappresenta l'aggiunta del nodo i all'albero radicato $T[h]$. Di conseguenza, il costo dell'arco (i, h) è:

$$\alpha_{ih} = c(S[h] \cup \{i\}) - c(S[h]).$$

Si può dimostrare che esiste una corrispondenza biunivoca tra l'insieme dei cicli disgiunti in $G^1(T)$ così modificato e l'insieme degli scambi basati su cammino e degli scambi ciclici. Quindi, cercare scambi di miglioramento nell'intorno della soluzione T è equivalente a cercare cicli disgiunti di costo negativo in $G^1(T)$, una volta effettuata la modifica.

Ricerca di cicli disgiunti di costo negativo

Poiché, come dimostrato da Thompson e Orlin (1989), trovare un *ciclo orientato disgiunto* di costo negativo è un problema NP-hard, Ahuja et al. utilizzano un metodo euristico, definito a partire da un algoritmo per cammini

minimi di tipo "label correcting" (per una descrizione di tale classe di algoritmi si rimanda a (Ahuja et al., 1993)). Dato un grafo $G' = (N', A')$, ai cui archi è associato un costo reale $c'(i,j)$, $\forall (i,j) \in A'$, questo tipo di algoritmi risolve il problema di determinare un albero di cammini minimi, avente radice prefissata s, senza ipotesi sulle caratteristiche di G' e del vettore dei costi c'. Questi algoritmi mantengono un'etichetta $l(j)$ associata ad ogni nodo j. Ad ogni iterazione, l'etichetta del nodo j indica che non è stato trovato nessun cammino dal nodo origine s a j se $l(j) = +\infty$, o che esiste un cammino da s a j di lunghezza $l(j)$ quando $l(j) < +\infty$. Inoltre, in $pred(j)$ viene mantenuto il predecessore di j nel cammino, da s a j, di costo $l(j)$. Infine, viene mantenuto un insieme di nodi, Q, contenente i nodi i per cui esiste un arco (i,j) che viola le "condizioni di ottimalità"(Ahuja et al., 1993), cioè $l(j) > l(i) + c'_{ij}$.

Durante una generica iterazione si seleziona e si rimuove un nodo i da Q , e si esaminano gli archi della stella uscente di i, $FS(i)$, che violano le condizioni di ottimalità. Se un arco (i,j) viola le condizioni di ottimalità, allora $l(j) := l(i) + c'_{ij}$ e il predecessore di $j(pred(j))$ diventa i. Nel caso in cui G' non contenga cicli orientati di costo negativo, l'algoritmo termina non appena Q è vuoto, cioè sono verificate le condizioni di ottimalità, ovvero $l(j) \leq l(i) + c'_{ij}$ per ogni arco (i,j) del grafo. In tal caso, $l(j)$ rappresenta il costo di un cammino minimo da s a j. Se invece sono presenti cicli di costo negativo, alcuni algoritmi "label correcting" sono in grado di rilevare la presenza di tali cicli, con complessità in tempo polinomiale.
Più formalmente:

procedure *label-correcting*$(G', s, pred)$
  begin
    $l(s) := 0$;
    $pred(s) := 0$;
    $l(j) := +\infty \ \ \forall j \in N' \setminus \{s\}$;
    $Q := \{s\}$;
    while $Q \neq \emptyset$ **do**

```
        select i from Q;
        Q := Q \ {i};
        foreach arc (i,j) ∈ FS(i) do
          if l(j) > l(i) + c'_ij then
            l(j) := l(i) + c'_ij;
            pred(j) := i;
            if j ∉ Q then Q := Q ∪ {j}; fi
          fi
        end
    end
end.
```

Un algoritmo di tipo "label correcting", in cui gli archi che violano le condizioni di ottimalità sono esaminati senza nessun ordine predefinito, ha una complessità computazionale in tempo che può essere esponenziale rispetto alle dimensioni dell'input. Nel caso in cui l'insieme dei nodi candidati Q venga implementato mediante una struttura dati "coda", la complessità computazionale in tempo diventa $O(|N'||A'|)$. Se invece Q è implementato mediante una struttura dati *deque*, si ottiene un algoritmo molto efficiente in pratica, anche se con una complessità computazionale teorica esponenziale. Nella versione con deque, un nodo viene inserito in testa alla struttura dati se l'algoritmo ha già migliorato la sua etichetta, altrimenti viene aggiunto in coda. L'estrazione di un nodo avviene sempre dalla testa.

Dovendo individuare un ciclo di costo negativo in $G^1(T)$ che sia anche disgiunto, la variante utilizzata da Ahuja, Orlin e Sharma, dopo aver selezionato un nodo i da Q, verifica se il cammino corrente da s a i (inizialmente il nodo fittizio v viene scelto come nodo radice, cioè v svolge il ruolo di s) sia un cammino disgiunto, cioè verifica se tutti i nodi in esso appartengano a alberi radicati diversi di T. Solo se questa condizione è verificata, si passa ad esaminare la stella uscente del nodo i. Se esiste un arco (i, j) tale che $l(j) > l(i) + \alpha_{ij}$, vanno considerati i seguenti tre casi:

Caso 1. Il nodo j è uno pseudonodo. Se $l(i)+\alpha_{ij} < 0$, allora è stato trovato un ciclo orientato disgiunto di costo negativo, corrispondente ad uno scambio di miglioramento basato su cammino.

Caso 2. Il nodo j è un nodo originario ($j \in N$), e il cammino da s a j è disgiunto; in tal caso si aggiorna $l(j) := l(i) + c_{ij}$, $pred(j) := i$ e si inserisce j in Q.

Caso 3. Il nodo j è un nodo originario ($j \in N$), ma il cammino da s a j non è disgiunto; in tal caso, se j appartiene al cammino da s a i, abbiamo trovato un ciclo disgiunto di miglioramento.

L'algoritmo viene eseguito finché non trova un ciclo disgiunto di costo negativo. Inoltre, l'algoritmo viene eseguito diverse volte, scegliendo ogni volta un nodo radice s diverso.

La complessità computazionale in tempo di questo algoritmo è n volte la complessità in tempo dell'algoritmo di tipo "label correcting" modificato, per via della verifica, ad ogni iterazione, che il cammino corrente sia disgiunto.

2.2.2 Una funzione intorno basata sullo scambio di sottoalberi

La seconda funzione intorno proposta da Ahuja et al. è una generalizzazione di quella proposta da Sharaia et al.. Obiettivo è permettere scambi di sottoalberi di alberi radicati.

Sia T l'albero ammissibile corrente. Uno scambio ciclico è una sequenza $i_1 - \ldots - i_r - i_1$, con $i_1, \ldots, i_r$ appartenenti a alberi radicati diversi; lo scambio ciclico ha ora il seguente significato: i nodi appartenenti al sottoalbero di radice i_1, T_{i_1}, vengono spostati da $T[i_1]$ a $T[i_2]$, i nodi del sottoalbero di radice i_2, T_{i_2} vengono spostati da $T[i_2]$ a $T[i_3]$ e così via, fino ai nodi del sottoalbero T_{i_r} che vengono spostati da $T[i_r]$ a $T[i_1]$. Uno scambio ciclico $i_1 - \ldots - i_r - i_1$ è *ammissibile* se l'albero ottenuto dopo aver effettuato

lo scambio soddisfa i vincoli di capacità. Vale a dire, uno scambio ciclico $i_1 - \ldots - i_r - i_1$ è ammissibile se e solo se:

$$\sum_{k \in T[i_p]} d_k + \sum_{k \in T_{i_{p-1}}} d_k - \sum_{k \in T_{i_p}} d_k \leq K,\ p = 1, 2, \ldots, r$$

dove definiamo, con un certo abuso di notazione, $p - 1 = r$ nel caso in cui $p = 1$.

Sia T' l' albero ottenuto dopo aver effettuato lo scambio ciclico. Il costo dello scambio è:

$$\begin{aligned} c(T') - c(T) &= c(S_{i_r} \cup S[i_1] \setminus S_{i_1}) - c(S[i_1]) + \\ &\quad \sum_{p=2}^{r} (c(S_{i_{p-1}} \cup S[i_p] \setminus S_{i_p}) - c(S[i_p])). \end{aligned}$$

Uno scambio ciclico è uno *scambio di miglioramento* se $c(T') < c(T)$. E' possibile definire anche scambi basati su cammino analogamente a quanto fatto in precedenza; uno scambio basato su cammino è uno *scambio di miglioramento* se $c(T') < c(T)$.

Dato un albero ammissibile T, l'albero T' è un *vicino* di T se T' è un albero ammissibile per CMST ed è ottenuto da T con uno scambio ciclico o uno scambio basato su cammino. L'intorno di una soluzione T è l'insieme contenente tutti i suoi *vicini*.

Il Grafo di miglioramento

Analogamente al grafo $G^1(T)$, definiamo $G^2(T) = (N, A^2)$ in modo che ogni scambio ciclico o basato su cammino corrisponda ad un ciclo disgiunto.

N è l'insieme dei nodi di G. L'insieme A^2 viene definito considerando tutte le coppie (i, j) di nodi in N, e inserendo l'arco (i, j) se e solo se: $T[i] \neq T[j]$, e se $\{S_i \cup S[j] \setminus S_j\}$ è un insieme di nodi ammissibile, dove S_i indica l'insieme dei nodi del sottoalbero T_i, $\forall i \in N$; vale a dire:

$$\sum_{k \in S[j]} d_k + \sum_{k \in S_i} d_k - \sum_{k \in S_j} d_k \leq K.$$

Un arco (i, j) in $G^2(T)$, con $T[i] \neq T[j]$, rappresenta lo spostamento dell'insieme di nodi S_i da $T[i]$ a $T[j]$, e la rimozione dell'insieme di nodi S_j da $T[j]$.

Il costo dell'arco (i, j) è quindi:

$$\beta_{ij} = c(S_i \cup S[j] \setminus S_j) - c(S[j]). \tag{2.2}$$

β_{ij} rappresenta cioè la variazione di costo dovuta alla cancellazione dell'insieme di nodi S_j da $T[j]$ e all'aggiunta dell'insieme di nodi S_i. Per calcolare i costi β_{ij} degli archi (i, j) di $G^2(T)$, assumiamo che il costo $c(S[j])$, per ogni albero radicato $T[j]$, sia conosciuto. Per determinare β_{ij} dobbiamo, quindi, calcolare $c(S_i \cup S[j] \setminus S_j)$. Il calcolo di $c(S_i \cup S[j] \setminus S_j)$ consiste nel togliere da $T[j]$ il sottoalbero T_j, radicato nel nodo j, aggiungere l'albero T_i, radicato nel nodo i, e calcolare il costo dell'albero di copertura di costo minimo sull'insieme $S_i \cup S[j] \setminus S_j$. Poiché l'insieme $S_i \cup S[j] \setminus S_j$ ha $O(K)$ nodi, è possibile calcolare ogni β_{ij} in tempo $O(K^2)$, usando ad esempio l'algoritmo di Prim (vedi Ahuja et al., 1998). Il calcolo di tutti i valori β_{ij} può essere quindi eseguito in tempo $O(n^2K^2)$.

Ahuja et al. hanno dimostrato che, anche in questo caso, esiste una corrispondenza biunivoca tra scambi ciclici nell'intorno di T e cicli disgiunti in $G^2(T)$. Inoltre, scambi ciclici nell'intorno di T e cicli disgiunti in $G^2(T)$ hanno lo stesso costo.

Analogamente a quanto eseguito nel caso di scambi di singoli nodi, è possibile modificare $G^2(T)$ in modo da includere nella corrispondenza anche gli scambi basati su cammino. L'unica differenza riguarda la definizione del costo degli archi (i, h)[1] e quello degli archi (v, i)[2]. In questo caso definiamo:

$$\beta_{ih} = c(S[h] \cup S_i) - c(S[h]),$$
$$\beta_{vi} = c(S[i] \setminus S_i);$$

[1] h è uno pseudonodo che rappresenta un albero radicato

[2] v è il nodo fittizio

β_{ih} rappresenta cioè il costo di aggiungere l'insieme di nodi S_i all'albero $T[h]$, mentre β_{vi} rappresenta il costo di togliere l'albero T_i da $T[i]$. Cercare scambi di miglioramento nell'intorno della soluzione T è equivalente a cercare cicli disgiunti di costo negativo in $G^2(T)$. E' possibile quindi utilizzare la stessa euristica descritta per il caso di scambi di singoli nodi.

2.2.3 Gli algoritmi di Ahuja, Orlin e Sharma

Ahuja et al. hanno implementato due algoritmi di ricerca locale, uno classico e uno basato su tecniche di tipo *tabu search*.

Un algoritmo di ricerca locale di tipo "tabu search" è caratterizzato da una funzione *intorno* e da una lista di mosse tabù (mosse non permesse). Come negli algoritmi di ricerca locale di tipo classico, anche in questo tipo di algoritmi la ricerca di una nuova soluzione, data la soluzione corrente S, avviene all'interno dell'intorno di S. Le soluzioni ammissibili vengono confrontate in base alla funzione obiettivo del problema in esame o ad una funzione obiettivo modificata. Ciò, da un lato, permette di non restare intrappolati in un ottimo locale, dall'altro però può far esaminare più volte una stessa soluzione. La lista delle mosse tabù viene introdotta proprio al fine di evitare di esaminare più volte una stessa soluzione. L'algoritmo termina quando non riesce a migliorare la soluzione corrente. Rimandiamo a (Glover, 1989a) e (Glover, 1989b) per una descrizione più dettagliata di questa famiglia di algoritmi di ricerca locale.

Entrambi gli algoritmi hanno bisogno di una soluzione ammissibile iniziale, e la loro efficacia dipende fortemente dalla soluzione di partenza selezionata. Soluzioni ammissibili sono state generate usando una variante dell'algoritmo di Esau e Williams (1966). L'algoritmo di Esau e Williams inizia con una soluzione in cui ogni albero radicato al nodo sorgente è costituito da un unico nodo. Durante la generica iterazione dell'algoritmo, due alberi radicati vengono uniti in un unico albero radicato in modo che il nuovo albero non violi i vincoli di capacità e sia massima la riduzione di costo avuta in seguito

all'operazione di unione. In Figura 2.1 è riportato un esempio di operazione di unione effettuata dall'algoritmo di Esau e Williams per un'istanza in cui $K = 3$ ed ogni nodo diverso da 0 ha domanda unitaria (i numeri associati agli archi rappresentano i costi).

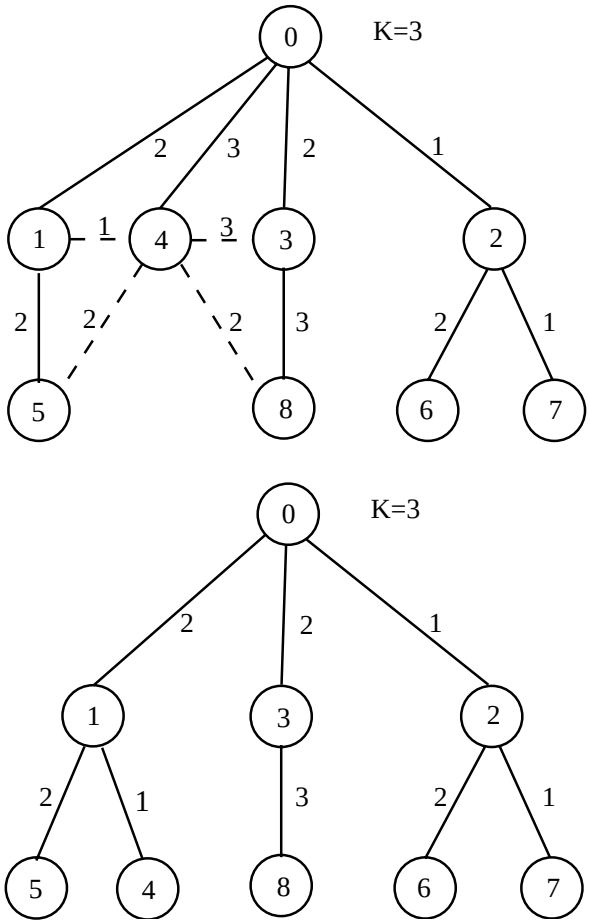

Figura 2.1: In alto è riportata una soluzione per il problema CMST. In basso è riportato il risultato di un'operazione di unione dei due sottoalberi aventi $\{4\}$ e $\{1, 5\}$ come insiemi di nodi

Nella versione utilizzata da Ahuja et al., ad ogni iterazione si determinano le p operazioni di unioni più vantaggiose. Poi si genera un intero k, uniformemente distribuito tra 1 e p, e si effettua la $k-$esima operazione di unione.

L'algoritmo di ricerca locale classico

Questo algoritmo parte da una soluzione iniziale ammissibile T, ottenuta mediante l'algoritmo di Esau e Williams modificato. L'algoritmo cerca allora uno scambio di miglioramento rispetto alla soluzione T. Se questo esiste, allora lo effettua e quindi aggiorna T. Questo processo si interrompe quando non si trovano più scambi di miglioramento rispetto alla soluzione corrente. La soluzione ottenuta è un ottimo locale. Ciò completa un'esecuzione dell'algoritmo, che è solitamente eseguito più volte, partendo da soluzioni ammissibili iniziali diverse, sempre ottenute mediante l'algoritmo di Esau e Williams modificato. L'esecuzione dell'algoritmo viene ripetuta finché non si raggiunge un numero prestabilito di esecuzioni, oppure il tempo totale di esecuzione raggiunge un prefissato limite superiore.

L'algoritmo di ricerca locale di tipo "tabu search"

L'algoritmo di tipo "tabu search" parte da una soluzione iniziale ammissibile T, ottenuta con l'algoritmo di Esau e Williams modificato. L'algoritmo cerca uno scambio di miglioramento rispetto alla soluzione T. Se questo esiste, lo effettua e, quindi, aggiorna T; altrimenti identifica lo scambio di costo minimo che non è tabù, e aggiorna la soluzione corrente T, effettuando questo scambio. Se lo scambio effettuato non migliora la soluzione corrente, allora la sua trasformazione inversa diventa tabù. Il criterio usato è questo: se lo scambio coinvolge i nodi $i_1, i_2, \ldots, i_r$, allora tutti gli scambi che coinvolgono questi nodi diventano tabù per un certo numero di iterazioni.

Anche in questo caso l'algoritmo viene eseguito più volte, partendo da soluzioni ammissibili iniziali diverse, ottenute con l'algoritmo di Esau e Williams modificato. L'esecuzione dell'algoritmo viene ripetuta finché non si raggiunge un numero prestabilito di esecuzioni, oppure il tempo totale di esecuzione raggiunge un prefissato limite superiore.

2.2.4 Risultati Sperimentali

È stata condotta un'ampia sperimentazione su quattro classi di istanze del problemi CMST, tre delle quali, denominate tc,te e cm, sono reperibili al sito web http://www.ms.ic.ac.uk/info.html, mentre la quarta è stata proposta dagli autori. Gli autori hanno implementato quattro algoritmi, IMP1, IMP2, TABU1 e TABU2. IMP1 è l'implementazione dell'algoritmo di ricerca locale classico che utilizza una funzione intorno basata su scambio di singoli nodi. IMP2 è l'implementazione dell'algoritmo di ricerca locale classico che utilizza una funzione intorno basata su scambio di sottoalberi. TABU1 è l'implementazione dell'algoritmo di ricerca locale di tipo "tabu search" che utilizza una funzione intorno basata su scambio di singoli nodi. TABU2 è l'implementazione dell'algoritmo di ricerca locale di tipo "tabu search" che utilizza una funzione intorno basata su scambio di sottoalberi.

I risultati riportati dagli autori dimostrano che l'algoritmo TABU1 ottiene soluzioni che eguagliano la miglior soluzione generata per tutti i problemi di tipo tc e te; inoltre, per tre di essi tale soluzione viene migliorata. L'algoritmo TABU2, invece, riesce a migliorare gran parte delle miglior soluzioni conosciute per i problemi di tipo cm. Il miglioramento medio è del 3%, mentre il massimo miglioramento ottenuto è del 18%.

3

Il problema di Assegnamento di lavori a macchine

3.1 Introduzione

Sia $M = \{1, \ldots, m\}$ un insieme di m macchine identiche che devono eseguire un insieme di lavori, $L = \{1, \ldots, n\}$, tra loro indipendenti. Ognuna di queste macchine può eseguire un lavoro alla volta. Il tempo di lavorazione del lavoro i è d_i (per $i \in L$), indipendentemente dalla macchina che lo esegue. Si assume che, per ogni $i \in L$, il tempo di lavorazione d_i sia intero. Si vogliono assegnare tutti i lavori alle macchine in modo da minimizzare il tempo di completamento della macchina più carica("makespan").

Una possibile formulazione del problema in termini di Programmazione Lineare Intera è la seguente, dove:

$$x_{ij} = \begin{cases} 1 & \text{se il lavoro } i \text{ è assegnato alla macchina } j \\ 0 & \text{altrimenti} \end{cases}$$

$$\min \quad z \tag{3.1}$$

$$\sum_{j \in M} x_{ij} \quad = 1 \qquad \forall i \in L \tag{3.2}$$

$$\sum_{i \in L} d_i \cdot x_{ij} \quad \leq z \qquad \forall j \in M \tag{3.3}$$

$$x_{ij} \in \{0, 1\} \qquad \forall (i, j) \in L \times M \tag{3.4}$$

I vincoli (3.2) e (3.4) assicurano che ogni lavoro sia assegnato ad una sola macchina, mentre i vincoli (3.3) assicurano che tutte le macchine lavorino al massimo per z unità di tempo.

Poiché ogni lavoro deve essere associato ad una macchina soltanto, ogni soluzione ammissibile per il problema, S, rappresenta una partizione di L in m sottoinsiemi. In Figura 3.1 è riportato un esempio di assegnamento di lavori a macchine in cui il numero delle macchine è 2 ed il numero dei lavori è 5; i

tempi di lavorazione sono $[2, 1, 3, 3, 4]$. Nel seguito, dato un sottoinsieme di lavori $L' \subseteq L$, con $c(L')$ indicheremo la somma delle durate dei lavori in L':

$$c(L') = \sum_{j \in L'} d_j$$

Data una soluzione ammissibile $S = \{L_1, \ldots, L_m\}$, il costo di S ("makespan") è il massimo tempo di lavorazione delle macchine:

$$C_{max}(S) = \max_{j \in M} c(L_j)$$

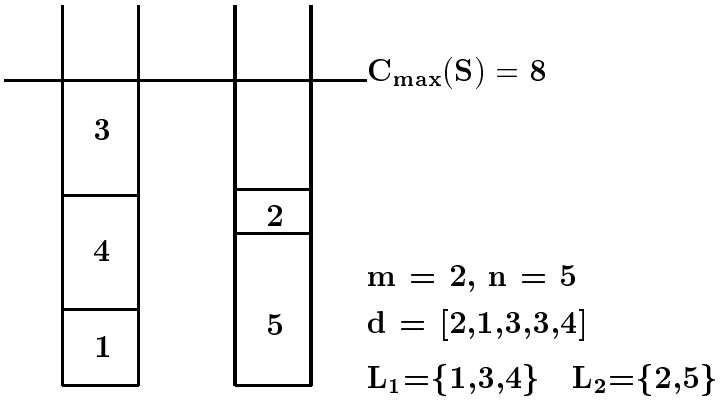

Figura 3.1: Esempio di assegnamento di lavori a macchine

3.2 Algoritmi euristici basati su tecniche di tipo "greedy"

Gli algoritmi più semplici per il problema di Assegnamento di lavori a macchine si basano sulla regola "*list scheduling*" (LS). Data una sequenza dei lavori da eseguire, la regola LS consiste nel selezionare il primo lavoro disponibile in base all'ordinamento prefissato, e assegnarlo alla macchina "più scarica", ovvero avente il minimo tempo di completamento corrente. Vale il seguente teorema.

Teorema 3.1. *Graham,1966:* Il rapporto tra il valore della soluzione ottenuta mediante un'euristica di tipo $LS(C_{max}(LS))$ e il valore della soluzione ottima (C^*_{max}) è al più $2 - \frac{1}{m}$.

Dimostrazione. Sia l un lavoro completato per ultimo nella soluzione restituita dall'euristica. Osserviamo che tutte le macchine lavorano almeno fino all'istante di tempo $t = C_{max}(LS) - d_l$, che è il momento in cui il lavoro l viene assegnato ad una macchina. Si ha che $\sum_{i \neq l} d_i \geq mt$ e quindi

$$C_{max}(LS) = t + d_l \leq \frac{1}{m} \sum_{i \neq l} d_i + d_l = \frac{1}{m} \sum_i d_i + \frac{m-1}{m} d_l.$$

Poiché $C^*_{max} \geq \frac{1}{m} \sum_i d_i$ e $C^*_{max} \geq d_l$ si ha

$$\frac{C_{max}(LS)}{C^*_{max}} \leq 2 - \frac{1}{m}$$

♢

Achugbue e Chin (1981) hanno dimostrato che, applicando la regola LS ad istanze particolari, si possono ottenere "bounds" migliori. In particolare, se $\pi = \frac{\max_i d_i}{\min_i d_i} \leq 3$ allora

$$\frac{C_{max}(LS)}{C^*_{max}} \leq \begin{cases} \frac{5}{3} & \text{se } m = 3, 4 \\ \frac{17}{10} & \text{se } m = 5 \\ 2 - \frac{1}{3\lfloor \frac{m}{3} \rfloor} & \text{se } m \geq 6 \end{cases} .$$

Se invece $\pi \leq 2$ si ha

$$\frac{C_{max}(LS)}{C^*_{max}} \leq \begin{cases} \frac{3}{2} & \text{se } m = 2, 3 \\ \frac{5}{3} - \frac{1}{3\lfloor \frac{m}{2} \rfloor} & \text{se } m \geq 4 \end{cases}$$

Se si considera la particolare regola LS basata sull'ordinamento dei lavori in ordine di durate non crescente, otteniamo la regola "*longest processing*

time" (LPT) che garantisce, come dimostrato da Graham (1969), il seguente "bound":

$$\frac{C_{max}(LS)}{C^*_{max}} \leq \frac{4}{3} - \frac{1}{3m}.$$

Un algoritmo considerato migliore è l'algoritmo "*multifit*" (MF) di Coffman Jr et al. (1978). I lavori sono considerati in ordine non crescente rispetto al loro tempo di lavorazione. L'algoritmo MF fissa un valore per il "makespan", e cerca di assegnare tutti i lavori alle macchine in modo che il tempo di lavorazione di ogni macchina non ecceda il "makespan" prefissato.

La ricerca del "makespan" avviene, mediante ricerca binaria, all'interno di un intervallo $[low, high]$. In (Coffman Jr et al., 1978) vengono suggeriti i seguenti valori per low e $high$:

$$low = \max\left\{\frac{1}{m}\sum_i d_i\,,\ \max_i d_i\right\}, high = \max\left\{\frac{2}{m}\sum_i d_i\,,\ \max_i d_i\right\}.$$

La generica iterazione dell'algoritmo consiste nel selezionare il punto mediano, len, dell'intervallo $[low, high]$ e applicare la procedura FFD(len). La procedura FFD(len) cerca di assegnare i lavori alle macchine in modo che il loro tempo di lavorazione non ecceda len. Durante la generica iterazione, FFD(len) seleziona il primo lavoro i non ancora considerato in base all'ordinamento prefissato, e cerca di assegnarlo alla prima macchina j ($j = 1, \ldots, m$) per la quale vale $c(L_j)+d_i < len$ (L_j rappresenta l'insieme dei lavori assegnati a j). Se FFD(len) termina con successo, vale a dire tutti i lavori sono assegnati alle macchine, allora il "makespan" corrente è minore o uguale a len, e quindi l'algoritmo MF continua la ricerca all'interno dell'intervallo $[low, len]$; altrimenti la ricerca continua in $[len, high]$. La ricerca binaria viene iterata per un numero prefissato k di iterazioni. Più formalmente:

procedure *MF(L,d,k)*
  begin
    init (low , $high$);

```
    sort(L, d);
    for i := 1 to k do
        begin
          len := (high + low) div 2;
          if FFD(len) then {tutti i lavori sono stati assegnati}
            high := len;
          else
            low := len;
          fi
        end
end.
```

Coffman Jr et al. hanno dimostrato che la complessità computazionale in tempo dell'algoritmo MF è $O(n \log n + kn \log m)$, e vale:

$$\frac{C_{max}(MF)}{C^*_{max}} \leq 1.22 + 2^{-k}$$

Se i lavori non vengono ordinati, ma vengono considerati arbitrariamente, si può dimostrare solo che:

$$\frac{C_{max}(MF)}{C^*_{max}} \leq 2 - \frac{2}{m+1}$$

Friesen e Langston (1986) hanno migliorato l'algoritmo MF ottenendo un algoritmo, MF', che ha la stessa complessità computazionale in tempo ma garantisce, nel caso pessimo, un rapporto tra "makespan" calcolato e "makespan" ottimo limitato da:

$$\frac{C_{max}(MF')}{C^*_{max}} \leq \frac{72}{61} + 2^{-k}$$

Un altro algoritmo euristico è l'algoritmo Z_k, di Graham (1969), che individua un assegnamento ottimo dei primi k lavori[1], per un prefissato valore di k, e poi applica la regola LS ai restanti lavori. Graham (1969) ha dimostrato che

$$\frac{C_{max}(Z_k)}{C^*_{max}} \leq 1 + \frac{1 - \frac{1}{m}}{1 + \lfloor \frac{k}{m} \rfloor}.$$

[1] I lavori sono ordinati in ordine non crescente rispetto alle loro durate

Scegliendo $k = \frac{m}{\epsilon}$, per un opportuno $\epsilon > 0$, si ottiene

$$\frac{C_{max}(Z_k)}{C^*_{max}} \leq 1 + \epsilon$$

L'algoritmo proposto è quindi uno schema di approssimazione. Sfortunatamente, la complessità computazionale in tempo di questo algoritmo è $O(n^{km})$. L'algoritmo è quindi polinomiale solo per m fissato.

La proposta di Sahni (1976) è un algoritmo di tipo euristico derivato da un algoritmo di programmazione dinamica.
L'algoritmo di tipo programmazione dinamica è il seguente:

```
procedure MinimumFinishTime(d, S)
  begin
    S^(0) := {(0)};
    for i := 1 to n do
        begin
          V := ∅;
          foreach l̄ ∈ S^(i-1) do
            V := V ∪ {(l̄ + d_i)}
          end
          S^(i) := merge(V, S^(i-1));
        end
    S := CalcolaSoluzione(S^(n));
  end.
```

Nel caso in cui il numero delle macchine sia $m = 2$, gli elementi di $S^{(i)}$ rappresentano i possibili tempi di lavorazione di una delle due macchine dopo l'assegnamento dei primi i lavori, $i = 1, \ldots, n$. L'algoritmo può essere generalizzato al caso $m > 2$ scegliendo come elementi di $S^{(i)}$ tuple di $m - 1$ elementi.

$S^{(i)}$ rappresenta quindi un insieme di soluzioni ammissibili parziali (i lavori $\{1, \ldots, i\}$ non appartenenti ad un elemento di $S^{(i)}$ sono implicitamente assegnati alla seconda macchina). L'algoritmo di programmazione dinamica

genera iterativamente gli insiemi $S^{(1)}, S^{(2)}, \ldots, S^{(n)}$, utilizzando la procedura $merge(V, S)$, che unisce i due insiemi V e S eliminando tutti gli elementi che eccedono $\frac{\sum_i d_i}{2}$. Al termine dell'esecuzione dell'algoritmo, gli elementi dell'insieme $S^{(n)}$ rappresentano le soluzioni ammissibili del problema; indicano infatti il tempo di completamento di una delle due macchina, dato dalla somma dei tempi di lavorazione dei lavori assegnati a tale macchina. I lavori non assegnati a questa macchina sono implicitamente assegnati all'altra. La complessità computazionale in tempo dell'algoritmo è $O(\min\{2^n, n\sum_i d_i\})$.

L'algoritmo di tipo euristico derivante dallo schema di programmazione dinamica appena descritto è stato proposto in due versioni. La prima versione si basa sulla tecnica "*dynamic interval partition*", mentre la seconda si basa sulla tecnica "*static interval partition*". Consideriamo per semplicità il caso $m = 2$, e sia $M = \sum_i d_i$.

Una volta ottenuto $S^{(i)}$, la tecnica "dynamic interval partition" consiste nel trovare $l = \max\{k \mid k \in S^{(i)}\}$ e dividere l'intervallo $[1, l]$ in $\lceil \frac{n}{\epsilon} \rceil$ parti uguali di dimensioni al più $\lceil \frac{\epsilon l}{n} \rceil$, per un opportuno valore di $\epsilon > 0$. Durante la costruzione di $S^{(i+1)}$, per ognuno degli $\lceil \frac{n}{\epsilon} \rceil$ sottointervalli di $[1, l]$ si considera un solo elemento, più precisamente si mantiene l'elemento che ha valore minimo all'interno del sottointervallo, eliminando gli altri elementi. Si ottiene così un'approssimazione di $S^{(i+1)}$.

La tecnica "static interval partition" consiste nel dividere l'intervallo $[1, M]$ in $\lceil \frac{2n}{\epsilon} \rceil$ parti uguali, ognuna di dimensioni $\lceil \frac{M\epsilon}{2n} \rceil$. L'ultimo sottointervallo potrebbe avere dimensioni minori. Durante la generica iterazione, analogamente alla tecnica "dynamic interval partition", viene considerato un solo elemento tra tutti quelli appartenenti ad uno stesso sottointervallo. Così ogni insieme $S^{(i)}$ ha al più $\lceil \frac{2n}{\epsilon} \rceil$ elementi e quindi la costruzione di $S^{(i+1)}$ costa $O(\frac{n}{\epsilon})$ tempo. L'errore commesso ad ogni iterazione è al più $\lfloor \frac{M\epsilon}{2n} \rfloor$; quindi, l'errore dopo n iterazioni è al più $n\lfloor \frac{M\epsilon}{2n} \rfloor \leq \epsilon C^*_{max}$. La complessità computazionale in tempo dell'algoritmo dipende da m; se $m = 2$, entrambe le versioni hanno complessità $O(\frac{n^2}{\epsilon})$.

Citiamo infine il lavoro di Hochbaum e Shmoys (1987). Questi autori hanno proposto una variante dell'algoritmo "multifit" per risolvere il problema di Assegnamento di lavori a macchine. Dato un tempo di lavorazione r, dato un insieme di lavori da assegnare e dato ρ ($\rho > 1$), l'algoritmo produce una soluzione in cui i lavori sono assegnati in modo che ogni macchina lavori al più per $r\rho$ unità di tempo. Fissati ρ e k, cioè il numero di iterazioni(di ricerca binaria) svolte dall'algoritmo, si ottiene un algoritmo ϵ-approssimato con $\epsilon = (\rho + 2^{-k})$.

3.3 Algoritmi di ricerca locale

Le principali funzioni *intorno* utilizzate in letteratura per il problema di Assegnamento di lavori a macchine sono (cfr. Anderson et al., 1997):

- *Ri-Assegnamento*: consiste nel rimuovere un lavoro da una macchina e assegnarlo ad un'altra.
- *Swap*: consiste nel prendere due lavori i_1 e i_2, assegnati a macchine diverse, γ_1 e γ_2 rispettivamente, e invertirli (vale a dire i_1 è assegnato a γ_2 e i_2 è assegnato a γ_1).
- *k-Ri-Assegnamento*: consiste nel prendere h ($h \leq k$) lavori di una macchina e assegnarli a macchine diverse.

La cardinalità dell'intorno Ri-Assegnamento è $n(m-1)$, mentre quella del k-Ri-Assegnamento è al più $\sum_{i=1}^{k} \binom{n}{i}(m-1)^i$; anche nel caso dell'intorno di tipo Swap è possibile fornire solo una limitazione superiore, $\frac{m(m-1)}{2}\lceil\frac{n}{m}\rceil^2$, che si ottiene assumendo che i lavori siano ripartiti in ugual numero tra le macchine. Nei lavori di Finn e Horowitz (1979), di França et al. (1994) e di Hübscher e Glover (1994) vengono utilizzate le suddette funzioni intorno.

Finn e Horowitz (1979) hanno proposto un algoritmo di complessità $O(nlogm)$ in cui alcuni lavori assegnati alle macchine più cariche sono *ri-assegnati* a quelle meno cariche.

França et al. (1994) hanno proposto un algoritmo più sofisticato in cui si usa una funzione intorno risultante da una combinazione del k-Ri-assegnamento e dello Swap. Per scegliere le mosse da fare, vale a dire per esplorare l'intorno, si partizionano i lavori in base ai rispettivi tempi di lavorazione.

Hübscher e Glover (1994) hanno proposto un algoritmo di ricerca locale di tipo "tabu search" in cui la funzione obiettivo viene sostituita con $\sum_{j=1}^{m}(c(L_j) - \bar{C})^2$, dove $c(L_j)$ è il tempo totale di lavorazione della macchina j nella soluzione corrente S, mentre $\bar{C} = \sum_{i=1}^{n} \frac{d_i}{m}$ rappresenta il carico ideale. In questa euristica si usa sia una funzione intorno basata sul *k-Ri-assegnamento "critico"* che una funzione intorno basata sullo *Swap "critico"*. Quando un lavoro viene mosso da una macchina, per evitare di reinserire nella macchina un lavoro con lo stesso tempo di lavorazione, tale lavoro viene inserito in una lista di mosse "tabù". Inoltre, alcuni lavori che sono nella lista delle mosse tabù sono periodicamente attivati o disattivati, con lo scopo di diversificare ed intensificare la ricerca.

3.4 Algoritmi di ricerca locale basati su grafi di miglioramento

3.4.1 Grafi di miglioramento e funzioni intorno

Nel lavoro di Thompson e Psaraftis (1993) per il problema di *"Vehicle routing"* e nel lavoro di Ahuja et al. (1998) per il problema dell'*Albero di copertura di costo minimo capacitato*, per i quali si rimanda rispettivamente ai Capitoli 1 e 2, sono stati presentati algoritmi di ricerca locale particolari, in cui la funzione intorno è definita in base ad un opportuno *grafo di miglioramento*. Come dimostrato nei due lavori, particolari cicli di costo negativo in tale grafo permettono di migliorare la soluzione ammissibile corrente; il problema originario è quindi ridotto al problema di determinare tali cicli negativi nel grafo di miglioramento.

Obiettivo di questo paragrafo è estendere la definizione di intorno, e con-

seguentemente quella di grafo di miglioramento, al problema di assegnamento di lavori a macchine.

Indichiamo con $L_l(S)$ l'insieme dei lavori assegnati alla macchina cui è assegnato l nella soluzione ammissibile corrente S.

Definiamo uno *scambio ciclico* come una sequenza di lavori $W = l_1 - l_2 \ldots l_r - l_1$, dove $L_{l_i}(S) \neq L_{l_j}(S)$ per $i \neq j$, $i, j \in \{1, 2, \ldots, r\}$: lo scambio consiste nell'assegnare simultaneamente ogni lavoro l_i $(1 \leq i < r \leq m)$ alla macchina cui è assegnato l_{i+1}, mentre l_r sarà assegnato alla macchina che esegue l_1. Analogamente a quanto definito nei Capitoli 1 e 2, lo scambio ciclico è cioè un meccanismo per modificare la soluzione ammissibile corrente. Infatti, se $m(l_i)$ è la macchina alla quale è assegnato il lavoro l_i nella soluzione corrente S, effettuare lo scambio ciclico W significa che, nella nuova soluzione S' ottenuta da S, il lavoro l_1 non sarà svolto dalla macchina $m(l_1)$ ma sarà svolto dalla macchina $m(l_2)$, il lavoro l_2 non sarà svolto dalla macchina $m(l_2)$ ma dalla macchina $m(l_3)$, così via fino al lavoro l_r che non sarà svolto dalla macchina $m(l_r)$ ma dalla macchina $m(l_1)$.

Definiamo inoltre uno *scambio basato su cammino* come una sequenza $P = l_1 - l_2 \ldots l_{r-1} - m_r$, dove $L_{l_i}(S) \neq L_{l_j}(S)$ per $i \neq j$, $i, j \in \{1, 2, \ldots, r-1\}$ e $m(l_i) \neq m_r$ per $1 \leq i \leq r - 1 \leq m$. Il significato di P è che ogni lavoro $l_i (1 \leq i \leq r - 2)$ sarà simultaneamente assegnato alla macchina cui è assegnato l_{i+1}, mentre l_{r-1} verrà assegnato alla macchina m_r. Analogamente allo scambio ciclico, uno scambio basato su cammino P indica quindi come ottenere una nuova soluzione S' a partire da quella corrente S. Infatti, se $m(l_i)$ è la macchina alla quale è assegnato il lavoro l_i nella soluzione S, effettuare lo scambio P significa che, nella nuova soluzione S' ottenuta da S, il lavoro l_1 non sarà svolto dalla macchina $m(l_1)$ ma sarà svolto dalla macchina $m(l_2)$, il lavoro l_2 non sarà svolto dalla macchina $m(l_2)$ ma dalla macchina $m(l_3)$, così via fino al lavoro l_{r-1} che non sarà svolto dalla macchina $m(l_{r-1})$ ma dalla macchina m_r.

Definiamo allora *intorno* di una soluzione ammissibile S l'insieme delle soluzioni ammissibili ottenute da S mediante uno scambio ciclico oppure

mediante uno scambio basato su cammino. Data la soluzione S' ottenuta da S effettuando lo scambio, si definisce *costo dello scambio*:

$$C_{max}(S') - C_{max}(S) = \max_{j \in M} \sum_{i \in L'_j} d_i - \max_{j \in M} \sum_{i \in L_j} d_i$$

dove $S' = \{L'_1, \ldots, L'_m\}$ e $S = \{L_1, \ldots, L_m\}$.

Lo scambio ciclico o lo scambio basato su cammino è uno *scambio di miglioramento* se $C_{max}(S') - C_{max}(S) < 0$.

Il grafo di miglioramento

Data una soluzione ammissibile $S = \{L_1, \ldots, L_m\}$, il grafo di miglioramento relativo a S, $G(S) = (N(S), A(S))$, è un grafo orientato con:

$$\begin{aligned} N(S) =& L \cup \{\, j \in M \mid c(L_j) < C_{max}(S)\} \\ A(S) =& \{(i,j) \in L \times L \mid L_i(S) \neq L_j(S), c(L_j(S) \setminus \{j\} \cup \{i\}) < C_{max}(S)\} \cup \\ & \{(i,j) \in L \times M \mid i \notin L_j,\, c(L_j) + d_i < C_{max}(S)\}. \end{aligned}$$

$N(S)$ contiene un nodo per ogni lavoro i nell'istanza in esame, e per ogni macchina j avente un tempo di lavorazione minore del "makespan" associato alla soluzione S, cioè $C_{max}(S)$. Un arco $(i,j) \in A(S)$ ha un significato diverso a seconda che j rappresenti in $G(S)$ un lavoro o una macchina. Infatti, se j rappresenta un lavoro, allora l'arco (i,j) indica che il lavoro i può essere assegnato alla macchina che nella soluzione corrente S esegue j, la quale viene privata del lavoro j, ottenendo un tempo di lavorazione inferiore al "makespan" corrente. Se invece j rappresenta una macchina, allora l'arco (i,j) indica che il lavoro i può essere assegnato alla macchina j, ottenendo un tempo di lavorazione inferiore al "makespan" corrente. Ad ogni arco $(i,j) \in A(S)$ è associato il seguente costo:

$$c_{ij} = \begin{cases} c(L_j(S) \setminus \{j\} \cup \{i\}) - c(L_j(S)) & \text{se } j \in L,\, c(L_j(S)) = C_{max}(S) \\ 0 & \text{altrimenti} \end{cases} \tag{3.5}$$

In base alla definizione, se il costo dell'arco (i, j) è negativo (in tal caso il nodo j rappresenta un lavoro), allora la macchina cui è assegnato j nella soluzione corrente è tra quelle più cariche, e l'eventuale scambio del lavoro i con il lavoro j può ridurre il tempo di lavorazione di tale macchina , e quindi il "makespan" corrente, di c_{ij} unità. Osserviamo che c_{ij} non tiene in considerazione la variazione del tempo di lavorazione della macchina che esegue il lavoro i.

Nel caso in cui $c_{ij} = 0$, invece, il nodo j può rappresentare un lavoro o una macchina. Se il nodo j rappresenta un lavoro, significa che la macchina a cui è assegnato j nella soluzione corrente non è tra quelle più cariche, e l'eventuale scambio del lavoro i con il lavoro j può peggiorare il tempo di lavorazione di tale macchina ma non il valore della funzione obiettivo, cioè il "makespan". Se invece il nodo j rappresenta una macchina, significa che questa, nella soluzione corrente, non è tra quelle più cariche, e l'eventuale assegnamento del lavoro i alla macchina j può peggiorare il tempo di lavorazione di j, ma non il "makespan".

In Figura 3.2 è riportato il grafo di miglioramento corrispondente alla soluzione ammissibile S rappresentata in Figura 3.1, con i costi definiti usando l'espressione (3.5).

Possiamo, alternativamente, definire il costo c_{ij}, $\forall (i, j) \in A(S)$, nel modo seguente:

$$c_{ij} = \begin{cases} c(L_j(S) \setminus \{j\} \cup \{i\}) - C_{max}(S) & \text{se } j \in L \\ c(L_j) + d_i - C_{max}(S) & \text{altrimenti} \end{cases} \tag{3.6}$$

Se $j \in L$, il costo c_{ij} rappresenta la variazione di tempo di lavorazione della macchina che eseguiva j, dovuta allo scambio del lavoro i con il lavoro j, rispetto al tempo di lavorazione della macchina più carica nella soluzione corrente S.

Se $j \in M$, l'arco (i, j) rappresenta l'assegnamento del lavoro i alla macchina j. Il costo c_{ij} misura in tal caso la variazione di tempo di lavorazione

della macchina j rispetto alla macchina più carica nella soluzione corrente S, in seguito all'assegnamento del lavoro i.

Per come è stato definito $G(S)$, in questa definizione alternativa dei costi si ha $c_{ij} < 0, \forall (i,j) \in A(S)$. In Figura 3.3 è riportato il grafo di miglioramento corrispondente alla soluzione ammissibile S rappresentata in Figura 3.1, con i costi definiti usando l'espressione (3.6).

Osserviamo che $N(S)$ ha cardinalità al più $n+m-1$; infatti i nodi del grafo rappresentano tutti i lavori più le macchine il cui tempo di lavorazione è minore di $C_{max}(S)$, che al massimo sono $m-1$. $A(S)$ ha cardinalità al più $n(n+m-2)$, perché ogni nodo che rappresenta un lavoro potrebbe avere un arco uscente verso uno qualunque dei restanti $n-1+m-1$ nodi.

In $G(S)$ definiamo *ciclo orientato disgiunto* un ciclo $W = l_1 - l_2 \ldots l_r - l_1$ in cui $l_1, l_2, \ldots, l_r$ rappresentano lavori, e tale che $L_{l_i}(S) \neq L_{l_j}(S)$ per $i \neq j$. Si definisce inoltre *cammino orientato disgiunto* un cammino $P = l_1 - l_2 \ldots l_{r-1} - m_r$ tale che $L_{l_i}(S) \neq L_{l_j}(S)$ per $i \neq j$, $i, j \in \{1, \ldots, r-1\}$, e $m(l_i) \neq m_r$ per $i = 1, \ldots, r-1$ ($l_1, l_2, \ldots, l_{r-1}$ rappresentano lavori, mentre il nodo m_r rappresenta una macchina).
Vale la seguente proprietà:

Lemma 3.1. *Ogni ciclo orientato disgiunto in $G(S)$ corrisponde ad uno scambio ciclico in S.*

Dimostrazione. Consideriamo un ciclo orientato disgiunto $W = l_1 - l_2 \ldots l_r - l_1$ in $G(S)$; per ogni coppia (l_i, l_j) in W, con $i \neq j$, vale $L_{l_i}(S) \neq L_{l_j}(S)$, che è proprio la definizione di scambio ciclico. $\diamondsuit$

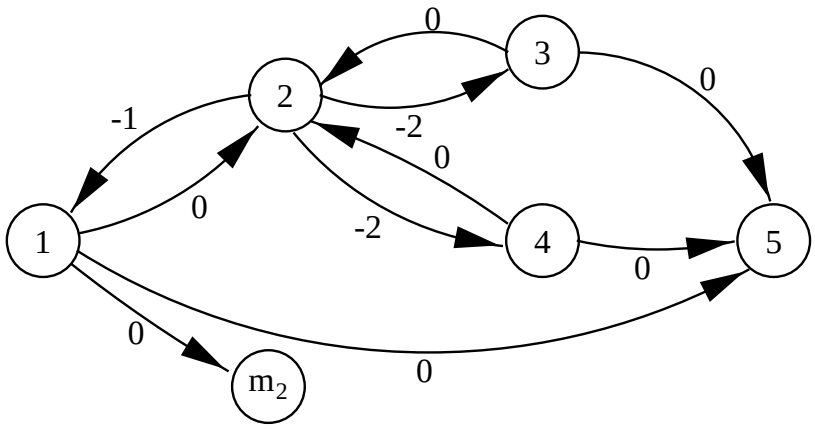

Figura 3.2: Grafo di miglioramento relativo all'assegnamento riportato in Figura 3.1 con costi (3.5)

Dato un ciclo orientato disgiunto W, il suo costo è:

$$c(W) = \sum_{i=1,\ldots,r} c_{l_{i-1}l_i}$$

dove, con un leggero abuso di notazione, si è posto $i-1=r$ se $i=1$. Osserviamo che $c(W) \leq 0$ sia che si adotti la definizione di costo (3.5), sia che si utilizzi la definizione alternativa (3.6). Se indichiamo con $K = \{j \in M \mid c(L_j) = C_{max}(S)\}$, vale la seguente proprietà:

Proprietà 3.1. *Un ciclo orientato disgiunto migliora la soluzione corrente S solo se nel ciclo sono coinvolte tutte le macchine che lavorano per $C_{max}(S)$ unità di tempo, vale a dire tutte le macchine appartenenti all'insieme K. Un tale ciclo ha sempre costo negativo, sia che si utilizzi la definizione di costo* (3.5) *che* (3.6).

Una analoga corrispondenza può essere stabilita tra i cammini orientati disgiunti in $G(S)$ e gli scambi basati su cammino. L'individuazione di uno scambio di miglioramento nell'intorno della soluzione corrente S è stata quindi ridotta al problema di individuare un ciclo/cammino orientato disgiunto

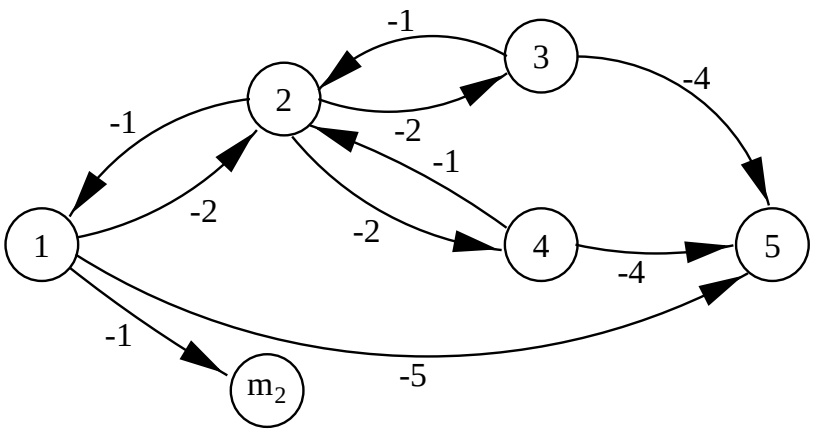

Figura 3.3: Grafo di miglioramento relativo all'assegnamento riportato in Figura 3.1 con costi (3.6)

in $G(S)$ che coinvolga tutte le macchine in K. Tali cicli(cammini) verranno chiamati *K-cicli(cammini) disgiunti.*

La definizione di grafo di miglioramento può essere generalizzata, considerando come nodi del grafo insiemi di lavori; fissato un intero k, si può infatti definire $G^k(S) = (N^k(S), A^k(S))$, dove:

$$
\begin{aligned}
N^k(S) =& L^k(S) \cup \{\, j \in M \mid c(L_j) < C_{max}(S)\} \\
A^k(S) =& \{(I,J) \in L^k(S) \times L^k(S) \mid L_I(S) \neq L_J(S), \\
& \quad c(L_J(S) \setminus J \cup I) < C_{max}(S)\} \\
& \quad \cup \{(I,j) \in L^k(S) \times M \mid i \in I \Rightarrow i \notin L_j, c(L_j) + c(I) < C_{max}(S)\}
\end{aligned}
$$

$L^k(S)$ denota la famiglia di insiemi costituiti da k lavori assegnati ad una stessa macchina nella soluzione S, mentre $L_I(S)$ denota l'insieme dei lavori assegnati alla macchina che esegue i lavori in I nella soluzione corrente S.

Ad ogni arco $(I, J) \in A^k(S)$ può essere associato un costo:

$$c_{IJ} = \begin{cases} c(L_J(S) \setminus J \cup I) - c(L_J(S)) & \text{se } J \in L^k(S),\ c(L_J(S)) = C_{max}(S) \\ 0 & \text{altrimenti} \end{cases}$$

oppure:

$$c_{IJ} = \begin{cases} c(L_J(S) \setminus J \cup I) - C_{max}(S) & \text{se } J \in L^k(S) \\ c(L_j) + c(I) - C_{max}(S) & \text{altrimenti.} \end{cases}$$

Se i lavori sono assegnati in ugual numero alle macchine nella soluzione corrente S, si ha che $N^k(S)$ ha al più $m\binom{\lceil \frac{n}{m} \rceil}{k}$ nodi, corrispondenti agli insiemi di lavori, e al massimo $m-1$ nodi corrispondenti alle macchine il cui tempo di lavorazione è minore di $C_{max}(S)$; quindi $|N^k(S)| \leq m\binom{\lceil \frac{n}{m} \rceil}{k} + m - 1$, mentre $|A^k(S)| \leq m\binom{\lceil \frac{n}{m} \rceil}{k}[(m-1)\binom{\lceil \frac{n}{m} \rceil}{k} + m - 1]$.

3.4.2 Due algoritmi di Ricerca Locale basati su grafi di miglioramento

Usando la funzione intorno introdotta nel paragrafo 3.4.1, proponiamo il seguente schema di algoritmo di ricerca locale per il problema di assegnamento di lavori a macchine:

procedure *RicercaLocaleALM*(S)
  begin
    $S :=$ *SoluzioneIniziale*(L, M);
    $G(S) :=$ *CreaGrafo*(S);
    while *EsisteK-CicloIn(G(S))* **do**
      $C :=$ *TrovaK-Ciclo(G(S))*;
      $S :=$ *Aggiorna*(S, C);
      $G(S) :=$ *CreaGrafo*(S);
  end
end.

La funzione *SoluzioneIniziale*(L, M) determina una soluzione ammissibile per il problema di assegnamento dei lavori in L alle macchine in M.

La funzione $CreaGrafo(S)$, data una soluzione ammissibile S, crea il grafo di miglioramento $G(S)$.

La funzione *EsisteK-CicloIn*$(G(S))$ verifica se nel grafo di miglioramento $G(S)$ esiste un ciclo (cammino) disgiunto di miglioramento, vale a dire un K-ciclo(cammino) disgiunto.

La funzione *Aggiorna*(S, C) riceve in input la soluzione S e un ciclo (cammino) disgiunto di miglioramento C e restituisce la nuova soluzione ottenuta mediante gli scambi indotti dal ciclo (cammino).

La funzione *TrovaK-Ciclo*$(G(S))$ cerca in $G(S)$ un K-ciclo (cammino) orientato disgiunto. Poiché trovare tale tipo di struttura in $G(S)$ è un problema NP-hard l'algoritmo cerca un ciclo (cammino) orientato disgiunto di tale tipo attraverso un metodo euristico.

A partire dallo schema algoritmico *RicercaLocaleALM*, sono stati proposti due diversi algoritmi, che si differenziano per la funzione costo associata agli archi del grafo di miglioramento ((3.5) nel primo algoritmo, (3.6) nel secondo), e per l'euristica mediante la quale viene cercato un K-ciclo (cammino) orientato disgiunto ad ogni iterazione, vale a dire per la realizzazione di *TrovaK-Ciclo*$(G(S))$.

La prima variante, indicata in seguito MS_SPT, ad ogni iterazione cerca un K-cammino(ciclo) orientato disgiunto nel grafo di miglioramento $G(S)$ utilizzando una variante dell'algoritmo per cammini minimi di tipo "label correcting" descritto nel paragrafo 2.2.1 (analogamente a quanto proposto da Ahuja et al. (1998) per il problema CMST). Dato un nodo sorgente s, la ricerca di un K-cammino(ciclo) orientato disgiunto in $G(S)$ necessita di un'ulteriore etichetta $nm(j)$ per ogni nodo j, in aggiunta a quelle utilizzate nell'algoritmo "label correcting". L'etichetta $nm(j)$ indica il numero di macchine coinvolte nel cammino dal nodo radice s a j che, nella soluzione corrente S, hanno tempo di lavorazione pari a $C_{max}(S)$. Come nella versione propostada Ahuja, Orlin e Sharma per il problema CMST (descritta a

pag.27), ogni volta che si rimuove un nodo i dall'insieme dei nodi candidati Q l'algoritmo controlla se il cammino da s a i è disgiunto. Se tale proprietà vale, viene esaminata la stella uscente di $i(FS(i))$. Sia $(i,j) \in FS(i)$ un arco che viola le condizioni di ottimalità, cioè tale che $l(j) > l(i) + c_{ij}$: vengono considerati i seguenti tre casi:

Caso 1. Il nodo j rappresenta una macchina in $G(S)$; se aggiungendo j al cammino corrente questo resta disgiunto e $nm(j) = |K|$, allora è stato individuato un K-cammino orientato disgiunto (l'algoritmo termina).

Caso 2. Il nodo j rappresenta un lavoro in $G(S)$ e il cammino da s a j è disgiunto; in tal caso si aggiorna $l(j) := l(i) + c_{ij}$, $pred(j) := i$, si aggiorna $nm(j)$[2] e si inserisce j in Q.

Caso 3. Il nodo j rappresenta un lavoro in $G(S)$, ma il cammino da s a j non è disgiunto; in tal caso si verifica se j appartiene già a questo cammino. Se j appartiene già al cammino e $nm(i) - nm(pred(j)) = |K|$, allora è stato individuato un K-ciclo orientato disgiunto (l'algoritmo termina).

La complessità computazionale in tempo dell'algoritmo di ricerca di K-cicli (cammini) orientati disgiunti è $O(m)$ volte la complessità in tempo dell'algoritmo di tipo "label correcting" modificato, per via della verifica, ad ogni iterazione, che il cammino corrente sia disgiunto e per il fatto che un cammino disgiunto può coinvolgere al più m nodi.

Il secondo algoritmo euristico proposto, indicato in seguito MS_BPT, si differenzia per la funzione costo associata agli archi del grafo di miglioramento ((3.6) anziché (3.5)), e per la strategia utilizzata per individuare un K-ciclo (cammino) orientato disgiunto ad ogni iterazione L'algoritmo utilizza

[2] $nm(j) := \begin{cases} nm(i)+1 & \text{se } c(L_j(S)) = C_{max}(S) \\ nm(i) & \text{altrimenti} \end{cases}$

infatti per la ricerca una variante dell'algoritmo che individua un albero dei cammini "bottleneck" di radice prefissata s su un grafo $G' = (N', A')$. Un cammino da s a i è detto "bottleneck" se è minimo l'arco di costo massimo. L'algoritmo per l'individuazione di un albero di cammini "bottleneck" può essere ottenuto dall'algoritmo di tipo "label correcting" descritto nel paragrafo 2.2 sostituendo la condizione di ottimalità $l(j) \leq l(i) + c'_{ij}$ con $l(j) \leq \max\{l(i), c'_{ij}\}$.

Più precisamente, durante una generica iterazione l'algoritmo seleziona e rimuove un nodo i da Q , e esamina gli archi della stella uscente di i, $FS(i)$, che violano le condizioni di ottimalità. Se un arco (i, j) viola le condizioni di ottimalità, allora $l(j) := \max\{l(i), c'_{ij}\}$ e il predecessore di $j(pred(j))$ diventa i. L'algoritmo termina non appena Q è vuoto, cioè sono verificate le condizioni di ottimalità, ovvero $l(j) \leq \max\{l(i), c_{ij}\}$ per ogni arco (i, j) del grafo. In tal caso, $l(j)$ rappresenta il costo di un cammino "bottleneck" da s a j.
Più formalmente:

procedure *BottleneckPathTree*$(G', s, pred)$
  begin
    $l(s) := -\infty$;
    $pred(s) := 0$;
    $l(j) := +\infty \ \ \forall j \in N' \setminus \{s\}$;
    $Q := \{s\}$;
    while $Q \neq \emptyset$ **do**
          select i *from* Q;
          $Q := Q \setminus \{i\}$;
          foreach *arc (i,j)* $\in$ *FS(i)* **do**
            if $l(j) > \max\{l(i), c'_{ij}\}$ **then**
              $l(j) := \max\{l(i), c'_{ij}\}$;
              $pred(j) := i$;
              if $j \notin Q$ **then** $Q := Q \cup \{j\}$; **fi**
            fi
          end
    end
  end.

La variante di *BottleneckPathTree* proposta per cercare un K-ciclo (cammino) disgiunto utilizza, in aggiunta, l'etichetta $nm(j)$ precedentemente introdotta. Come nella versione di Ahuja, Orlin e Sharma, descritta a pag.27, ogni volta che si rimuove un nodo $i \in G(S)$ dall'insieme dei nodi candidati Q si controlla se il cammino da s a i è disgiunto. Se tale proprietà vale, si esamina la stella uscente di $i(FS(i))$. Sia $(i,j) \in FS(i)$ un arco che viola le condizioni di ottimalità in $G(S)$, cioè tale che $l(j) > \max\{l(i), c_{ij}\}$; vanno considerati i seguenti casi:

Caso 1. Il nodo j rappresenta una macchina in $G(S)$; se aggiungendo j al cammino corrente questo resta disgiunto e $nm(j) = |K|$; allora abbiamo trovato un K-cammino orientato (l'algoritmo termina).

Caso 2. Il nodo j rappresenta un lavoro in $G(S)$ e il cammino da s a j è disgiunto; si aggiorna $l(j) := \max\{l(i), c_{ij}\}$, $pred(j) := i$, si aggiorna $nm(j)$[3] e si inserisce j in Q.

Caso 3. Il nodo j rappresenta un lavoro in $G(S)$, ma il cammino da s a j non è disgiunto; in tal caso si verifica se j appartiene già a questo cammino. Se j appartiene già al cammino e $nm(i) - nm(pred(j)) = |K|$, allora abbiamo trovato un K-ciclo orientato disgiunto (l'algoritmo termina).

Osserviamo che l'algoritmo MS_BPT mira ad individuare, ad ogni iterazione, un K-ciclo (cammino) orientato disgiunto, nell'intorno della soluzione corrente S, che consenta una massima riduzione del "makespan". Infatti, l'algoritmo cerca cicli (cammini) che minimizzano il massimo c_{ij} ovvero, essendo $c_{ij} < 0 \ \forall (i,j) \in A(S)$, e rappresentando c_{ij} la riduzione del "makespan" dovuta alla mossa associata all'arco (i,j), l'algoritmo cerca di massimizzare la riduzione del "makespan".

L'algoritmo MS_SPT, invece, nonostante il suo obiettivo primario sia trovare un ciclo (cammino) disgiunto che coinvolga tutti i nodi in K, indipen-

[3] $nm(j) := \begin{cases} nm(i)+1 & \text{se } c(L_j(S)) = C_{max}(S) \\ nm(i) & \text{altrimenti} \end{cases}$

dentemente dal suo costo, ritiene ragionevole nella ricerca del ciclo essere guidato dalla funzione costo definita mediante l'espressione (3.5), al fine di ottenere, euristicamente, una buona riduzione del "makespan" corrente.

3.5 Una variante del problema di Assegnamento di lavori a macchine

Sia $M = \{1, \ldots, m\}$ un insieme di m macchine identiche che devono eseguire un insieme di lavori, $L = \{1, \ldots, n\}$, tra loro indipendenti. Ognuna di queste macchine può eseguire un lavoro alla volta. Il tempo di lavorazione del lavoro i è d_i (per $i \in L$), indipendentemente dalla macchina che lo esegue. Inoltre, ogni lavoro i diventa disponibile ad una certa *data di rilascio* r_i. La data di rilascio r_i è l'istante di tempo a partire dal quale il lavoro i può essere eseguito su una qualsiasi delle macchine libere. Si assume che, per ogni $i \in L$, sia il tempo di lavorazione d_i che la data di rilascio r_i siano interi positivi.

Obiettivo del problema è assegnare tutti i lavori alle macchine in modo da minimizzare la somma:

$$\sum_{i \in L} (D_i - r_i),$$

dove D_i indica l'istante di tempo in cui termina l'esecuzione del lavoro i. Nel seguito, l'espressione $(D_i - r_i)$ verrà indicato come *tempo di lavorazione* del lavoro i, per $i \in L$.

3.5.1 Algoritmi di ricerca locale basati su grafi di miglioramento

È possibile estendere sia la definizione di intorno che quella di grafo di miglioramento proposte per il problema di Assegnamento di lavori a macchine. Come in quel caso, indichiamo con $L_l(S)$ l'insieme dei lavori assegnati alla macchina cui è assegnato il lavoro l nella soluzione ammissibile corrente S. Indichiamo inoltre con $c(L_l(S))$ la somma dei tempi di completamento dei

lavori assegnati alla macchina che , nella soluzione corrente S, esegue il lavoro l.

Definiamo uno *scambio ciclico* come una sequenza di lavori $W = l_1 - l_2 \ldots l_r - l_1$, dove $L_{l_i}(S) \neq L_{l_j}(S)$ per $i \neq j$, $i, j \in \{1, 2, \ldots, r\}$.

Definiamo inoltre uno *scambio basato su cammino* come una sequenza $P = l_1 - l_2 \ldots l_{r-1} - l_r$, dove $L_{l_i}(S) \neq L_{l_j}(S)$ per $i \neq j$, $i, j \in \{1, 2, \ldots, r\}$.

Definiamo allora *intorno* di una soluzione ammissibile S l'insieme delle soluzioni ammissibili ottenute da S mediante uno scambio ciclico oppure mediante uno scambio basato su cammino.

Data la soluzione S' ottenuta da S effettuando uno scambio W, si definisce *costo dello scambio ciclico*:

$$\begin{aligned} c(S') - c(S) &= c(L_{l_1}(S) \setminus \{l_1\} \cup \{l_r\}) + \\ &\quad \sum_{p=2}^{r} c(L_{l_p}(S) \setminus \{l_p\} \cup \{l_{p-1}\}) - \sum_{p=1}^{r} c(L_{l_p}(S)). \end{aligned} \tag{3.7}$$

Definiamo inoltre *costo di uno scambio per cammino*:

$$\begin{aligned} c(S') - c(S) &= c(L_{l_1}(S) \setminus \{l_1\}) + \sum_{p=2}^{r-1} c(L_{l_p}(S) \setminus \{l_p\} \cup \{l_{p-1}\}) + \\ &\quad c(L_{l_r}(S) \cup \{l_{r-1}\}) - \sum_{p=1}^{r} c(L_{l_p}(S)). \end{aligned} \tag{3.8}$$

Il grafo di miglioramento

Data una soluzione ammissibile $S = \{L_1, \ldots, L_m\}$, il grafo di miglioramento relativo a S, $G(S) = (N(S), A(S))$, è un grafo orientato con:

$$\begin{aligned} N(S) &= L \\ A(S) &= \{(i, j) \in L \times L \mid L_i(S) \neq L_j(S)\} \end{aligned}$$

Un arco $(i, j) \in A(S)$ indica che il lavoro i può essere assegnato alla macchina che nella soluzione corrente S esegue j ad un costo c_{ij}.
Il costo c_{ij} associato ad ogni arco $(i, j) \in A(S)$ è

$$c_{ij} = c(L_j(S) \setminus \{j\} \cup \{i\}) - c(L_j(S)) \tag{3.9}$$

Per calcolare i costi degli archi di $G(S)$ utilizziamo l'espressione (3.9). Per determinare c_{ij} dobbiamo calcolare $c(L_j(S) \setminus \{j\} \cup \{i\})$, vale a dire assegnare i lavori in $\{L_j(S) \setminus \{j\} \cup \{i\}\}$ ad una macchina, tenendo conto delle date di rilascio, in modo da minimizzare la somma $\sum_{k \in \{L_j(S) \setminus \{j\} \cup \{i\}\}} (D_k - r_k)$. Poiché, come dimostrato da Lenstra et al. (1977), questo problema è NP-hard è opportuno far ricorso ad un'euristica per approssimare i costi degli archi del grafo di miglioramento. L'euristica proposta consiste nel considerare i lavori dell'insieme $\{L_j(S) \setminus \{j\} \cup \{i\}\}$ in ordine non decrescente rispetto alle date di rilascio, e nell'assegnarli in questo ordine alla macchina.

Lemma 3.2. *Esiste una corrispondenza biunivoca tra cicli orientati disgiunti in $G(S)$ e scambi ciclici in S. Inoltre, ogni ciclo orientato disgiunto ha un costo che approssima per eccesso il costo dello scambio ciclico corrispondente.*

Dimostrazione. Consideriamo uno scambio ciclico $W = l_1 - l_2 \ldots l_r - l_1$. Per ogni coppia (l_i, l_j) in W, con $i \neq j$, vale $L_{l_i}(S) \neq L_{l_j}(S)$, che è proprio la definizione di ciclo disgiunto in $G(S)$. L'approssimazione del costo di W è dovuta all'utilizzo di un'euristica per il calcolo dei costi (3.9). $\diamondsuit$

Se trasformiamo opportunamente il grafo $G(S)$, anche gli scambi basati su cammino possono essere messi in corrispondenza con i cicli disgiunti. La trasformazione consiste nell'aggiungere a $G(S)$ alcuni nodi e archi. Aggiungiamo uno pseudonodo h per ogni macchina presente nella soluzione, e un nodo fittizio v. Aggiungiamo l'arco (v, i), di costo $c(L_i(S) \setminus \{i\}) - c(L_i(S))$, per ogni nodo i originario del grafo. Aggiungiamo l'arco (h, v), di costo 0, per ogni pseudonodo h. Aggiungiamo inoltre l'arco (i, h) se il lavoro i non è assegnato alla macchina h nella soluzione corrente. Quest'ultimo arco indica l'assegnamento del lavoro i alla macchina h, perciò ha un costo $c_{ih} = c(L_h \cup \{i\}) - c(L_h)$, se $S = \{L_1, \ldots, L_h, \ldots, L_m\}$.

Effettuata la trasformazione del grafo $G(S)$, si può dimostrare, con argomentazioni analoghe a quelle usate per gli scambi ciclici, che scambi basati su cammino inducono cicli orientati disgiunti in $G(S)$.

L'algoritmo che si può utilizzare per la ricerca di scambi ciclici che siamo di miglioramento per S, vale a dire cicli orientati disgiunti di costo negativo nel grafo di miglioramento $G(S)$, è analogo a quello descritto nel paragrafo 2.2.3.

4

Risultati Sperimentali

4.1 Introduzione

Gli algoritmi di ricerca locale descritti nel paragrafo 3.4.2 sono stati implementati al fine di valutare la loro efficienza da un punto di vista computazionale.

Sia per l'algoritmo MS_SPT che per MS_BPT sono state implementate due diverse versioni, che differiscono nella struttura dati utilizzata per realizzare l'insieme Q dei nodi candidati all'interno delle procedure di ricerca di K-cicli (cammini) orientati disgiunti: nella prima versione Q viene implementato mediante una *coda*, cioè le operazioni di inserzione e rimozione sono gestite con politica "FIFO", mentre nella seconda versione si utilizza una struttura dati di tipo *deque*.

Gli algoritmi sono stati implementati utilizzando il linguaggio C++. È stato utilizzato il compilatore GNU CC (egcs-1.0.3 release). La macchina utilizzata per i test è un Pentium a 100Mhz con sistema operativo Debian GNU/Linux 2.0.

4.2 Risultati sperimentali in letteratura

In letteratura non sono presenti molti lavori contenenti risultati sperimentali relativi al problema di Assegnamento di lavori a macchine.

Tra i lavori reperibili citiamo innanzitutto quello di Hübscher e Glover (1994). Tale lavoro contiene la descrizione di un algoritmo di ricerca locale di tipo "tabu search" per l'assegnamento di lavori a macchine; esso presenta inoltre i risultati di una sperimentazione condotta su istanze proposte dagli autori. In Tabella 4.1 sono riportati il numero di lavori (n) e di macchine (m) che caratterizzano tali istanze. Le durate dei lavori di tali istanze non sono intere, e sono generate uniformemente all'interno dell'intervallo $[0, 1]$.

La sperimentazione è stata condotta generando per ogni tipo di problema 10 istanze e stimando l'errore relativo commesso. L'errore relativo è stato valutato mediante l'espressione $\frac{C_{max}-\bar{C}}{\bar{C}}$, dove C_{max} indica il "makespan"

m	n
2	50
2	100
3	100
3	200
5	100
5	200
10	200
10	500
20	500
20	1000

Tabella 4.1: Tipologie di istanze utilizzate per la sperimentazione di Hübscher e Glover

calcolato dall'algoritmo mentre $\bar{C}$, pari a $\frac{1}{m}\sum_{i=1}^{n} d_i$, indica il carico "ideale" delle macchine. I risultati della sperimentazione condotta da Hübscher e Glover sono riportati in Tabella 4.2. In tale tabella è riportato l'errore relativo medio sulle 10 istanze, l'errore relativo massimo e quello minimo; inoltre è riportato anche il numero medio di iterazioni effettuate per risolvere ciascun tipo di problema.

Un altro lavoro contenente risultati sperimentali è quello di França et al. (1994). In tale lavoro, gli autori presentano un'euristica di costruzione per il problema in esame, e confrontano il suo comportamento con quello dell'euristica "Longest Processing Time" e con quello di due versioni dell'algoritmo di Finn e Horowitz (1979). Gli autori non hanno riportato i risultati ottenuti mediante l'euristica "Multifit", in quanto simili, a loro dire, a quelli ottenuti mediante l'euristica "Longest Processing Time". Le tipologie di istanze proposte dagli autori sono tre; esse si differenziano per l'ampiezza dell'intervallo all'interno del quale sono scelte le durate dei lavori ($[1, 100]$, $[1, 1000]$ e $[1, 10000]$). Tali durate sono intere e generate uniformemente.

L'algoritmo proposto da França et al. domina, in talune istanze, gli algoritmi con cui è messo a confronto; in molti casi le euristiche considerate

m	n	Media iter.	$\frac{C_{max}-\bar{C}}{\bar{C}}$ media	min	max
2	50	17607	1.08e-08	9.48e-10	1.71e-08
2	100	19157	7.08e-10	2.06e-11	2.13e-09
3	100	16384	7.52e-09	1.17e-09	3.34e-08
3	200	39293	6.77e-10	4.98e-11	1.69e-09
5	100	36392	4.38e-08	9.72e-09	1.57e-07
5	200	51982	1.38e-08	3.97e-09	2.90e-08
10	200	101801	4.98e-08	2.68e-08	8.99e-08
10	500	84816	1.11e-08	4.19e-09	1.78e-08
20	500	153526	3.19e-08	1.59e-08	5.89e-08
20	1000	124540	1.76e-08	8.59e-09	3.46e-08

Tabella 4.2: Tabella dei risultati ottenuti da Hübscher e Glover

riescono ad individuare una soluzione ottima (per una descrizione dettagliata dei risultati sperimentali si rimanda a França et al. (1994)).

4.3 Il piano della sperimentazione

È stata condotta un'ampia sperimentazione, su tre diverse tipologie di istanze del problema, indicate in seguito **alm-u**, **alm-1** e **alm-2**. Il numero di lavori (n) e di macchine (m) che caratterizzano i problemi sono gli stessi riportati in Tabella 4.1, secondo quanto suggerito in Hübscher e Glover (1994). Sfortunatamente non è stato possibile ottenere esattamente le istanze utilizzate da Hübscher e Glover, al fine di effettuare un confronto tra gli approcci algoritmici basati su grafi di miglioramento e l'algoritmo di ricerca locale di Hübscher e Glover. Queste tipologie di istanze vanno quindi intese come punto di partenza, e di raccordo con la letteratura, per l'impostazione di una sperimentazione relativa agli algoritmi proposti in questo lavoro di tesi.

I tre tipi di istanze, **alm-u**, **alm-1** e **alm-2**, differiscono per il modo in cui sono state generate le durate dei lavori.

Le durate dei lavori delle istanze **alm-u** sono uniformemente distribuite nell'intervallo $[0, 1]$, come suggerito in Hübscher e Glover (1994). Le istanze **alm-u** sono quindi dello stesso tipo utilizzato da Hübscher e Glover per testare l'algoritmo di ricerca locale da loro proposto.

Le istanze di tipo **alm-1** e **alm-2** sono state invece generate attraverso il generatore **mktest** riportato in Appendice, e sviluppato in questo lavoro di tesi. Tali istanze sono state generate con l'obiettivo di superare quello che sembra essere il limite principale della tipologia di istanze proposta da Hübscher e Glover, vale a dire la distribuzione uniforme delle durate dei lavori in $[0, 1]$. Tale tipo di istanze sembra essere infatti "facile" di natura, come testimoniato dai risultati sperimentali riportati nel seguito; inoltre, nel problema di Assegnamento di lavori a macchine le durate dei lavori non sono solitamente distribuite in modo uniforme.

Abbiamo voluto allora generare tipologie di istanze ragionevolmente più "difficili", caratterizzate da durate intere non uniformemente distribuite, al fine di valutare più accuratamente la bontà degli algoritmi proposti.

Più specificatamente, le durate dei lavori delle istanze **alm-1** sono intere e sono distribuite all'interno dell'intervallo $[1, 100]$. Le durate dei lavori delle istanze **alm-2** sono intere e sono distribuite all'interno dell'intervallo $[1, 1000]$. Nel prossimo paragrafo verranno illustrate le modalità di generazione di tali tipi di istanze, e le motivazioni che hanno supportato le scelte effettuate.

La sperimentazione è stata condotta generando, per ogni tipologia **alm-u**, **alm-1** e **alm-2**, 10 istanze diverse variando il seme del rispettivo generatore, ed eseguendo su ognuna di queste gli algoritmi implementati (il numero massimo di iterazioni eseguibili è stato fissato a 1000). Poiché l'efficacia degli algoritmi di ricerca locale dipende fortemente dalla soluzione iniziale utilizzata, ogni algoritmo è stato testato a partire da soluzioni iniziali diverse. Tali soluzioni iniziali sono state generate mediante le euristiche "List Sche-

duling", "Longest Processing Time" e "Multifit", descritte nel paragrafo 3.2, (anche Hübscher e Glover utilizzano un'euristica di tipo "List Scheduling" per generare la soluzione di partenza nel loro algoritmo di ricerca locale).

La bontà degli approcci algoritmici è stata valutata stimando l'errore relativo commesso mediante l'espressione $\frac{C_{max}-\bar{C}}{\bar{C}}$. Inoltre, è stata valutata anche l'espressione $\frac{C_{SI}-\bar{C}}{\bar{C}}$ al fine di valutare il miglioramento apportato nei confronti della soluzione ammissibile iniziale, dove C_{SI} è il "makespan" corrispondente alla soluzione iniziale. Nelle tabelle che seguono, contenenti risultati sperimentali, vengono anche riportati il tempo di esecuzione medio, in secondi, impiegato dagli algoritmi nella fase di miglioramento della soluzione iniziale, ed il numero medio di iterazioni eseguito. Il tempo necessario alla costruzione della soluzione iniziale non è stato riportato in quanto trascurabile.

4.3.1 Algoritmi per la generazione delle istanze

Le istanze di tipo **alm-1** e **alm-2** sono state ottenute attraverso un progressivo raffinamento del generatore, utilizzando l'euristica "Longest Processing Time" come "benchmark" per ottenere istanze "difficili".

Più precisamente, per ogni tipologia (m, n) indicata in Tabella 4.1, è stato valutato l'errore relativo medio commesso dall'euristica "Longest Processing Time" su 10 istanze. È stato effettuato un confronto tra:

1. istanze aventi durate dei lavori intere e uniformemente distribuite nell'intervallo $[1, 1000]$ (istanze **unif.**) e

2. istanze aventi durate dei lavori intere, generate mediante due diverse versioni del generatore (**mktest1** e **mktest2**), ideate allo scopo di rendere le istanze "difficili". I risultati sono riportati in Tabella 4.3, dove **mt1** indica le istanze ottenute mediante la versione **mktest1** del generatore, mentre **mt2** indica le istanze ottenute mediante **mktest2**.

L'euristica "Longest Processing Time" è stata scelta come "benchmark" per-

ché, come riportato in França et al. (1994), si ritiene che, per istanze caratterizzate da durate uniformi in $[1, 1000]$, essa riesca ad ottenere "buone soluzioni", spesso ottime.

		$\frac{C_{max}-\bar{C}}{\bar{C}}$		
m	**n**	**unif.**	**mt1**	**mt2**
2	50	3.85e-04	6.54e-04	1.13e-02
2	100	1.16e-04	2.41e-04	5.29e-04
3	100	2.00e-04	1.82e-03	3.75e-03
3	200	6.17e-05	6.98e-04	1.32e-03
5	100	4.41e-04	4.47e-03	1.26e-02
5	200	1.06e-04	1.66e-03	7.52e-03
10	200	3.47e-04	3.09e-03	1.03e-02
10	500	6.45e-05	2.27e-03	5.41e-03
20	500	1.67e-04	2.79e-03	6.62e-03
20	1000	4.01e-05	1.97e-03	5.93e-03

Tabella 4.3: Risultati dei test effettuati per la generazione di istanze "difficili"

Più in dettaglio, dato un intervallo di durate dei lavori $[1, Max]$, il generatore divide $[1, Max]$ in k sottointervalli. Dopo diversi tentativi, si è notato sperimentalmente che i migliori risultati si ottenevano con $k = 3$. L'intervallo $[1, Max]$ è quindi diviso in 3 sottointervalli I_1, I_2 e I_3. Le versioni **mktest1** e **mktest2** differiscono per l'ampiezza dei 3 intervalli e per il numero di durate generate all'interno di uno stesso intervallo. In entrambi i casi, la generazione delle durate dei lavori, in un dato sottointervallo I_h, $h = 1, 2, 3$, avviene scegliendo un valore base b, generato casualmente all'interno di I_h, e sommandovi un valore Δ variabile, generato in modo da garantire che il valore $b + \Delta$ appartenga all'intervallo I_h considerato. Più precisamente, Δ è uniformemente distribuito nell'intervallo $[0, \lceil rI_h * 0.1 \rceil]$, dove rI_h denota l'ampiezza dell'intervallo I_h.

Per quanto riguarda la differenza tra le due versioni del generatore, la

versione **mktest1** divide l'intervallo $[1, Max]$ in tre sottointervalli così definiti: $I_1 = [1 + \lceil (Max - 1) \cdot 0.66 \rceil, Max]$, $I_2 = [1 + \lceil (Max - 1) \cdot 0.33 \rceil, 1 + \lceil (Max - 1) \cdot 0.66 \rceil]$ e $I_3 = [1, 1 + \lceil (Max - 1) \cdot 0.33 \rceil]$. Inoltre, questa versione genera lo stesso numero di durate dei lavori in ognuno dei tre intervalli.

La versione **mktest2**, invece, divide l'intervallo $[1, Max]$ in tre sottointervalli così definiti: $I_1 = [1 + \lceil (Max - 1) \cdot 0.8 \rceil, Max]$, $I_2 = [1 + \lceil (Max - 1) \cdot 0.4 \rceil, 1 + \lceil (Max - 1) \cdot 0.7 \rceil]$ e $I_3 = [1, 1 + \lceil (Max - 1) \cdot 0.2 \rceil]$. Inoltre, il 33% delle durate dei lavori è generato nell'intervallo I_1, il 2% nell'intervallo I_3 e il resto nell'intervallo I_2.

In breve, la versione **mktest1** partiziona l'intervallo $[1, Max]$ in tre sottointervalli di ampiezza uguale e sceglie da ogni sottointervallo lo stesso numero di durate dei lavori. La versione **mktest2**, invece, divide l'intervallo $[1, Max]$ in tre sottointervalli di ampiezza diversa e genera in ognuno di questi un numero diverso di durate di lavori. La maggior parte delle durate vengono generate infatti nell'intervallo intermedio I_2, il 33% nell'intervallo I_1, mentre poche durate vengono generate nell'intervallo I_3, vale a dire nell'intervallo che caratterizza le durate minori.

I risultati riportati nella colonna **unif.** della Tabella 4.3 dimostrano che queste istanze sono relativamente "facili"; infatti l'errore relativo medio ottenuto mediante l'euristica "Longest Processing Time" è dell'ordine di 10^{-4}.

Dai risultati relativi alle istanze **mt1** e **mt2**, vale a dire le istanze generate mediante le versioni **mktest1** e **mktest2**, invece, si può osservare che l'euristica "Longest Processing Time" ottiene soluzioni sempre peggiori. In particolare le istanze **mt2** sono risultate mediamente più "difficili"; la versioni **mktest2** è stata quindi scelta per la nostra sperimentazione. Il generatore **mktest** indicato nel paragrafo 4.3 coincide quindi con **mktest2**.

4.4 Risultati Sperimentali

Al fine di fornire un quadro riassuntivo dell'andamento della sperimentazione condotta sugli algoritmi di ricerca locale basati su grafi di miglioramento, riportiamo in questo paragrafo i risultati ottenuti mediante l'algoritmo MS_SPT con politica "FIFO" applicato alle tre tipologie di istanze **alm-u**, **alm-1** e **alm-2**. L'algoritmo MS_SPT con politica "FIFO" è infatti l'algoritmo che, nella maggior parte dei casi, ha ottenuto i risultati migliori. Per informazioni più dettagliate sui risultati dell'intera sperimentazione si rimanda ai paragrafi 4.5, 4.6 e 4.7.

In Tabella 4.4 sono riportati i risultati ottenuti dall'algoritmo MS_SPT su istanze di tipo **alm-u**, a partire da soluzioni iniziali ottenute sia mediante "List Scheduling" che mediante "Longest Processing Time". Non sono stati riportati i risultati ottenuti dall'algoritmo MS_SPT a partire da soluzioni iniziali generate mediante l'euristica "Multifit" perché , nel corso della sperimentazione, è emerso che MS_SPT non è in grado di apportare miglioramenti significativi alla soluzione iniziale; utilizzando l'euristica "Multifit", MS_SPT ha inoltre generato soluzioni sempre peggiori di quelle ottenute a partire da soluzioni costruite con le altre euristiche.

Dalla tabella risulta che l'errore relativo medio commesso dagli algoritmi MS_SPT su istanze di tipo **alm-u** è inferiore a 10^{-5}.

La soluzione iniziale è inoltre sempre notevolmente migliorata.

In Tabella 4.5 sono riportati i risultati ottenuti dall'algoritmo MS_SPT su istanze di tipo **alm-1**, a partire da soluzioni iniziali ottenute sia mediante "List Scheduling" che mediante "Longest Processing Time".

Complessivamente, su queste istanze, più "difficili" in natura, gli algoritmi proposti riescono ad individuare un errore relativo medio sempre inferiore a 10^{-3}.

In Tabella 4.6 sono riportati infine i risultati ottenuti dall'algoritmo MS_SPT su istanze di tipo **alm-2**, a partire da soluzioni iniziali ottenute sia mediante "List Scheduling" che mediante "Longest Processing Time". Su

		"List Scheduling"				"Longest Processing Time"			
m	**n**	$\frac{C_{max}-\bar{C}}{\bar{C}}$	$\frac{C_{SL}-\bar{C}}{\bar{C}}$	**sec.**	**n.iter.**	$\frac{C_{max}-\bar{C}}{\bar{C}}$	$\frac{C_{SL}-\bar{C}}{\bar{C}}$	**sec.**	**n.iter.**
2	50	6.12e-05	1.27e-02	0.01	74	2.94e-05	6.19e-04	0.01	93
2	100	7.39e-06	9.83e-03	0.06	173	5.57e-06	9.90e-05	0.03	141
3	100	2.52e-05	1.31e-02	0.12	140	9.34e-06	3.95e-04	0.11	262
3	200	3.43e-06	5.56e-03	0.65	231	1.23e-06	7.57e-05	0.78	557
5	100	6.53e-05	3.76e-02	0.22	152	6.04e-05	1.27e-03	0.18	273
5	200	8.55e-06	1.27e-02	1.17	252	1.12e-05	3.21e-04	1.45	694
10	200	7.41e-05	3.87e-02	2.16	310	6.41e-05	1.20e-03	1.76	891
10	500	3.52e-06	1.91e-02	24.04	629	8.69e-06	2.21e-04	4.67	1000
20	500	1.54e-05	4.51e-02	34.88	702	9.24e-05	8.05e-04	6.58	1000
20	1000	2.16e-06	1.83e-02	191.60	1000	2.19e-05	1.83e-04	20.85	1000

Tabella 4.4: Risultati ottenuti mediante l'algoritmo MS_SPT eseguito su istanze di tipo **alm-u**

		"List Scheduling"				"Longest Processing Time"			
m	**n**	$\frac{C_{max}-\bar{C}}{\bar{C}}$	$\frac{C_{SL}-\bar{C}}{\bar{C}}$	**sec.**	**n.iter.**	$\frac{C_{max}-\bar{C}}{\bar{C}}$	$\frac{C_{SL}-\bar{C}}{\bar{C}}$	**sec.**	**n.iter.**
2	50	2.42e-04	1.41e-02	0	27	3.60e-04	1.01e-02	0.01	61
2	100	1.82e-04	7.31e-03	0.03	71	1.82e-04	9.89e-04	0.02	72
3	100	2.25e-04	1.48e-02	0.04	61	2.25e-04	3.71e-03	0.03	60
3	200	1.60e-04	6.78e-03	0.29	149	1.60e-04	1.59e-03	0.26	149
5	100	9.06e-04	2.67e-02	0.1	106	6.78e-04	1.05e-02	0.09	108
5	200	3.72e-04	1.25e-02	0.79	210	2.98e-04	6.80e-03	0.56	174
10	200	1.07e-03	2.83e-02	1.38	236	1.30e-03	1.03e-02	1.4	241
10	500	3.90e-04	1.35e-02	14.18	528	5.11e-04	5.61e-03	17.09	550
20	500	9.89e-04	2.69e-02	49.82	648	1.75e-03	6.76e-03	61.37	574
20	1000	4.75e-04	1.09e-02	262.87	1000	7.77e-04	5.43e-03	391.88	1000

Tabella 4.5: Risultati ottenuti mediante l'algoritmo MS_SPT eseguito su istanze di tipo **alm-1**

		"List Scheduling"				"Longest Processing Time"			
m	**n**	$\frac{C_{max}-\bar{C}}{\bar{C}}$	$\frac{C_{SL}-\bar{C}}{\bar{C}}$	**sec.**	**n.iter.**	$\frac{C_{max}-\bar{C}}{\bar{C}}$	$\frac{C_{SL}-\bar{C}}{\bar{C}}$	**sec.**	**n.iter.**
2	50	2.35e-05	1.41e-02	0	40	2.35e-05	1.13e-02	0	77
2	100	1.77e-05	7.29e-03	0.03	82	1.77e-05	5.29e-04	0.03	92
3	100	3.62e-05	1.43e-02	0.09	115	3.62e-05	3.75e-03	0.08	122
3	200	1.81e-05	6.23e-03	0.46	184	1.81e-05	1.32e-03	0.42	197
5	100	5.78e-04	2.48e-02	0.15	137	4.51e-05	1.26e-02	0.09	177
5	200	3.08e-05	1.24e-02	0.78	213	3.46e-05	7.52e-03	0.72	233
10	200	1.66e-04	2.59e-02	2.32	317	1.50e-04	1.03e-02	1.66	407
10	500	3.57e-05	1.30e-02	20.02	661	4.75e-05	5.41e-03	16.33	675
20	500	1.19e-04	2.46e-02	41.32	709	2.16e-04	6.62e-03	44.09	815
20	1000	6.47e-05	1.07e-02	228.26	1000	1.14e-04	5.93e-03	285.36	1000

Tabella 4.6: Risultati ottenuti mediante l'algoritmo MS_SPT eseguito su istanze di tipo **alm-2**

queste istanze l'algoritmo MS_SPT mostra un comportamento molto buono, riuscendo ad ottenere, nella maggioranza delle istanze testate, un errore relativo medio inferiore a 10^{-5}. Anche per queste istanze, le soluzioni iniziali sono notevolmente migliorate.

4.5 Risultati relativi alle istanze di tipo alm-u

In questo paragrafo vengono presentati in dettaglio i risultati ottenuti mediante le varianti degli algoritmi MS_SPT e MS_BPT applicati alle istanze di tipo **alm-u**. I risultati sono presentati nelle Tabelle numerate da 4.7 a 4.18, e illustrati mediante i grafici nelle Figure da 4.1 fino a 4.7.

Dai grafici si può osservare che i risultati migliori sono ottenuti quando la soluzione iniziale viene costruita mediante le euristiche "List Scheduling" e "Longest Processing Time". In particolare, nel caso delle istanze aventi un numero di lavori e macchine pari a $(2, 50)$, $(3, 100)$, $(3, 200)$, $(5, 100)$ e $(5, 200)$, si ottengono i risultati migliori partendo da soluzioni iniziali ricavate con l'euristica "Longest Processing Time", mentre nel caso in cui il numero dei lavori e delle macchine è pari a $(10, 200)$, $(10, 500)$, $(20, 500)$ e $(20, 1000)$,

la soluzione migliore si ottiene partendo da soluzioni iniziali ricavate mediante l'euristica "List Scheduling".

Quando la soluzione iniziale è costruita mediante l'euristica "List Scheduling", l'algoritmo che riesce ad individuare, nella maggior parte dei casi, la soluzione migliore è MS_SPT con politica "FIFO". Tale algoritmo risulta anche essere più veloce dell'algoritmo MS_SPT con politica di gestione dei nodi basata su "deque". Le versioni dell'algoritmo MS_BPT hanno tempi di esecuzione paragonabili.

Nel caso in cui la soluzione iniziale sia costruita mediante l'euristica "Longest Processing Time", tutti gli algoritmi proposti mostrano un comportamento analogo, per quanto riguarda la bontà della soluzione costruita. Comunque, la miglior soluzione è ottenuta dall'algoritmo MS_SPT con politica "FIFO". I tempi di esecuzione delle quattro varianti sono simili.

Infine, quando la soluzione iniziale è costruita mediante l'euristica "Multifit", le soluzioni ottenute dalle varie versioni degli algoritmi sono identiche. Anche in questo caso, i tempi di esecuzione delle quattro varianti sono paragonabili.

Complessivamente, l'errore relativo medio commesso dagli algoritmi MS_SPT su istanze di tipo **alm-u** è inferiore a 10^{-5} se si parte da soluzioni iniziali costruite mediante euristiche di tipo "List Scheduling" e "Longest Processing Time", ed è comunque inferiore a 10^{-4} utilizzando l'euristica iniziale "Multifit". Le varianti di MS_BPT riescono, in alcuni casi, a migliorare leggermente l'errore relativo medio, mentre per altre istanze mostrano un comportamento peggiore.

La soluzione iniziale è sempre notevolmente migliorata nel caso in cui essa sia costruita mediante euristiche di tipo "List Scheduling" e "Longest Processing Time" (tranne, per alcune istanze, nel caso si utilizzino i codici derivanti da MS_BPT). Un comportamento diverso si ha invece nel caso in cui si costruisca la soluzione iniziale mediante l'euristica "Multifit": in tali

casi, gli algoritmi di ricerca locale basati su grafi di miglioramento sembrano non essere in grado di apportare miglioramenti significativi.

Per quanto riguarda il confronto con la letteratura, è possibile osservare che, sulle istanze di tipo **alm-u** (vedi Hübscher e Glover, 1994), l'algoritmo da loro proposto, assai sofisticato, individua soluzioni "quasi ottime", come testimoniato dai risultati in Tabella 4.2.

Sulle istanze di tipo uniforme considerate in França et al. (1994), invece, in taluni casi si ottengono soluzioni con errore relativo maggiore. Il confronto è comunque solo indicativo, trattandosi di istanze differenti.

Osserviamo infine che, dalla nostra sperimentazione, emerge che le euristiche "Longest Processing Time" e "Multifit" non sono sempre computazionalmente paragonabili, come asserito invece nel lavoro di França et al. (1994).

		$\frac{C_{max}-C}{C}$			$\frac{C_{SL}-C}{C}$			Media	
m	n	max	min	media	max	min	media	sec	iter.
2	50	1.28e-04	2.55e-06	6.12e-05	2.70e-02	2.75e-03	1.27e-02	0.01	74
2	100	3.85e-05	8.21e-08	7.39e-06	1.87e-02	8.05e-04	9.83e-03	0.06	173
3	100	7.86e-05	3.02e-06	2.52e-05	3.63e-02	2.65e-04	1.31e-02	0.12	140
3	200	6.83e-06	4.12e-07	3.43e-06	9.69e-03	9.31e-04	5.56e-03	0.65	231
5	100	2.19e-04	1.27e-05	6.53e-05	6.56e-02	1.27e-02	3.76e-02	0.22	152
5	200	2.10e-05	2.58e-06	8.55e-06	2.22e-02	5.97e-03	1.27e-02	1.17	252
10	200	1.33e-04	3.16e-05	7.41e-05	5.40e-02	2.29e-02	3.87e-02	2.16	310
10	500	6.66e-06	2.20e-06	3.52e-06	2.33e-02	1.26e-02	1.91e-02	24.04	629
20	500	2.25e-05	9.47e-06	1.54e-05	5.74e-02	3.38e-02	4.51e-02	34.88	702
20	1000	2.75e-06	1.61e-06	2.16e-06	2.37e-02	1.17e-02	1.83e-02	191.60	1000

Tabella 4.7: Risultati dell'algoritmo MS_SPT con politica "FIFO" eseguito su istanze di tipo **alm-u**. Soluzione iniziale generata mediante l'euristica "List Scheduling".

		$\frac{C_{max}-C}{C}$			$\frac{C_{SL}-C}{C}$			Media	
m	n	max	min	media	max	min	media	sec	iter.
2	50	1.28e-04	2.55e-06	6.12e-05	2.70e-02	2.75e-03	1.27e-02	0.01	74
2	100	3.85e-05	8.21e-08	7.39e-06	1.87e-02	8.05e-04	9.83e-03	0.07	173
3	100	4.53e-05	8.75e-06	2.69e-05	3.63e-02	2.65e-04	1.31e-02	0.12	129
3	200	8.06e-06	2.61e-07	2.61e-06	9.69e-03	9.31e-04	5.56e-03	0.70	240
5	100	1.78e-04	1.49e-05	8.29e-05	6.56e-02	1.27e-02	3.76e-02	0.23	152
5	200	1.43e-05	9.12e-07	7.95e-06	2.22e-02	5.97e-03	1.27e-02	1.34	263
10	200	8.55e-05	2.68e-05	5.19e-05	5.40e-02	2.29e-02	3.87e-02	2.24	297
10	500	1.19e-05	1.64e-06	3.87e-06	2.33e-02	1.26e-02	1.91e-02	27.98	660
20	500	3.14e-05	1.45e-05	2.15e-05	5.74e-02	3.38e-02	4.51e-02	37.41	697
20	1000	4.85e-06	1.87e-06	2.83e-06	2.37e-02	1.17e-02	1.83e-02	213.33	1000

Tabella 4.8: Risultati dell'algoritmo MS_SPT con politica di gestione dei nodi basata su "deque" eseguito su istanze di tipo **alm-u**. Soluzione iniziale generata mediante l'euristica "List Scheduling".

		$\frac{C_{max}-\bar{C}}{\bar{C}}$			$\frac{C_{SI}-\bar{C}}{\bar{C}}$			Media	
m	n	max	min	media	max	min	media	sec	iter.
2	50	1.80e-04	8.21e-07	2.94e-05	2.16e-03	2.56e-05	6.19e-04	0.01	93
2	100	1.45e-05	9.90e-07	5.57e-06	2.86e-04	1.44e-06	9.90e-05	0.03	141
3	100	2.16e-05	1.34e-06	9.34e-06	9.67e-04	9.34e-06	3.95e-04	0.11	262
3	200	4.21e-06	2.26e-07	1.23e-06	1.67e-04	1.02e-05	7.57e-05	0.78	557
5	100	1.48e-04	1.71e-05	6.04e-05	2.33e-03	3.03e-04	1.27e-03	0.18	273
5	200	3.88e-05	3.17e-06	1.12e-05	7.53e-04	2.99e-05	3.21e-04	1.45	694
10	200	8.66e-05	4.82e-05	6.41e-05	2.31e-03	4.47e-04	1.20e-03	1.76	891
10	500	1.32e-05	4.93e-06	8.69e-06	5.29e-04	1.09e-04	2.21e-04	4.67	1000
20	500	1.45e-04	6.33e-05	9.24e-05	1.21e-03	4.21e-04	8.05e-04	6.58	1000
20	1000	4.07e-05	1.29e-05	2.19e-05	3.56e-04	5.87e-05	1.83e-04	20.85	1000

Tabella 4.9: Risultati dell'algoritmo MS_SPT con politica "FIFO" eseguito su istanze di tipo **alm-u**. Soluzione iniziale generata mediante l'euristica "Longest Processing Time".

		$\frac{C_{max}-C}{\bar{C}}$			$\frac{C_{SI}-C}{\bar{C}}$			Media	
m	n	max	min	media	max	min	media	sec	iter.
2	50	1.80e-04	8.21e-07	2.94e-05	2.16e-03	2.56e-05	6.19e-04	0.01	93
2	100	1.45e-05	9.90e-07	5.57e-06	2.86e-04	1.44e-06	9.90e-05	0.03	141
3	100	2.16e-05	1.34e-06	9.34e-06	9.67e-04	9.34e-06	3.95e-04	0.11	262
3	200	4.21e-06	2.26e-07	1.23e-06	1.67e-04	1.02e-05	7.57e-05	0.79	557
5	100	1.48e-04	1.71e-05	6.73e-05	2.33e-03	3.03e-04	1.27e-03	0.17	268
5	200	3.88e-05	3.17e-06	1.12e-05	7.53e-04	2.99e-05	3.21e-04	1.42	694
10	200	8.43e-05	4.82e-05	6.43e-05	2.31e-03	4.47e-04	1.20e-03	1.71	880
10	500	1.32e-05	4.93e-06	8.94e-06	5.29e-04	1.09e-04	2.21e-04	4.60	1000
20	500	1.11e-04	4.66e-05	8.79e-05	1.21e-03	4.21e-04	8.05e-04	6.12	1000
20	1000	4.07e-05	1.29e-05	2.16e-05	3.56e-04	5.87e-05	1.83e-04	20.05	1000

Tabella 4.10: Risultati dell'algoritmo MS_SPT con politica di gestione dei nodi basata su "deque" eseguito su istanze di tipo **alm-u**. Soluzione iniziale generata mediante l'euristica "Longest Processing Time".

		$\frac{C_{max}-C}{C}$			$\frac{C_{SI}-C}{C}$			Media	
m	n	max	min	media	max	min	media	sec	iter.
2	50	6.24e-04	4.57e-06	2.05e-04	6.24e-04	4.57e-06	2.39e-04	0.00	55
2	100	2.04e-04	2.24e-06	6.09e-05	2.04e-04	4.42e-06	6.81e-05	0.01	109
3	100	2.54e-04	1.48e-05	1.16e-04	2.55e-04	1.48e-05	1.27e-04	0.03	104
3	200	9.75e-05	4.38e-06	3.42e-05	1.06e-04	4.38e-06	3.83e-05	0.17	217
5	100	8.24e-04	1.08e-04	3.97e-04	8.24e-04	1.08e-04	4.31e-04	0.05	113
5	200	1.42e-04	3.16e-05	8.96e-05	1.42e-04	3.16e-05	9.72e-05	0.39	233
10	200	5.47e-04	1.03e-04	3.85e-04	5.47e-04	1.03e-04	3.86e-04	0.53	201
10	500	1.03e-04	2.88e-05	5.67e-05	1.03e-04	2.88e-05	6.04e-05	9.17	621
20	500	1.84e-04	5.97e-05	1.16e-04	2.28e-04	7.39e-05	1.27e-04	10.08	768
20	1000	3.68e-05	1.34e-05	2.78e-05	4.41e-05	1.34e-05	2.92e-05	61.56	1000

Tabella 4.11: Risultati dell'algoritmo MS_SPT con politica "FIFO" eseguito su istanze di tipo **alm-u**. Soluzione iniziale generata mediante l'euristica "Multifit".

		$\frac{C_{max}-C}{C}$			$\frac{C_{SI}-C}{C}$			Media	
m	n	max	min	media	max	min	media	sec	iter.
2	50	6.24e-04	4.57e-06	2.05e-04	6.24e-04	4.57e-06	2.39e-04	0.01	55
2	100	2.04e-04	2.24e-06	6.09e-05	2.04e-04	4.42e-06	6.81e-05	0.01	109
3	100	2.54e-04	1.48e-05	1.16e-04	2.55e-04	1.48e-05	1.27e-04	0.03	104
3	200	9.75e-05	4.38e-06	3.42e-05	1.06e-04	4.38e-06	3.83e-05	0.17	217
5	100	8.24e-04	1.08e-04	3.97e-04	8.24e-04	1.08e-04	4.31e-04	0.05	113
5	200	1.42e-04	3.16e-05	8.96e-05	1.42e-04	3.16e-05	9.72e-05	0.39	233
10	200	5.47e-04	1.03e-04	3.85e-04	5.47e-04	1.03e-04	3.86e-04	0.54	201
10	500	1.03e-04	2.88e-05	5.67e-05	1.03e-04	2.88e-05	6.04e-05	9.18	621
20	500	1.84e-04	5.97e-05	1.16e-04	2.28e-04	7.39e-05	1.27e-04	9.93	768
20	1000	3.68e-05	1.34e-05	2.78e-05	4.41e-05	1.34e-05	2.92e-05	59.41	1000

Tabella 4.12: Risultati dell'algoritmo MS_SPT con politica di gestione dei nodi basata su "deque" eseguito su istanze di tipo **alm-u**. Soluzione iniziale generata mediante l'euristica "Multifit".

		$\frac{C_{max}-\tilde{C}}{\tilde{C}}$			$\frac{C_{SI}-\tilde{C}}{\tilde{C}}$			Media	
m	n	max	min	media	max	min	media	sec	iter.
2	50	1.28e-04	2.55e-06	6.12e-05	2.70e-02	2.75e-03	1.27e-02	0.01	74
2	100	3.85e-05	8.21e-08	7.39e-06	1.87e-02	8.05e-04	9.83e-03	0.07	173
3	100	3.34e-05	6.17e-06	1.81e-05	3.63e-02	2.65e-04	1.31e-02	0.13	198
3	200	9.89e-06	3.03e-07	3.41e-06	9.69e-03	9.31e-04	5.56e-03	0.81	368
5	100	1.51e-04	4.22e-05	7.20e-05	6.56e-02	1.27e-02	3.76e-02	0.28	270
5	200	2.51e-05	2.76e-06	9.74e-06	2.22e-02	5.97e-03	1.27e-02	1.97	612
10	200	1.13e-04	2.39e-05	5.82e-05	5.40e-02	2.29e-02	3.87e-02	3.53	784
10	500	9.06e-06	3.53e-06	6.55e-06	2.33e-02	1.26e-02	1.91e-02	28.50	988
20	500	5.09e-05	2.94e-05	3.81e-05	5.74e-02	3.38e-02	4.51e-02	39.29	1000
20	1000	1.23e-05	5.67e-06	8.58e-06	2.37e-02	1.17e-02	1.83e-02	170.28	1000

Tabella 4.13: Risultati dell'algoritmo MS_BPT con politica "FIFO" eseguito su istanze di tipo **alm-u**. Soluzione iniziale generata mediante l'euristica "List Scheduling".

		$\frac{C_{max}-C}{C}$			$\frac{C_{SI}-C}{C}$			Media	
m	n	max	min	media	max	min	media	sec	iter.
2	50	1.28e-04	2.55e-06	6.12e-05	2.70e-02	2.75e-03	1.27e-02	0.01	74
2	100	3.85e-05	8.21e-08	7.39e-06	1.87e-02	8.05e-04	9.83e-03	0.07	173
3	100	3.34e-05	6.17e-06	1.81e-05	3.63e-02	2.65e-04	1.31e-02	0.13	198
3	200	9.89e-06	3.03e-07	3.41e-06	9.69e-03	9.31e-04	5.56e-03	0.82	368
5	100	1.51e-04	4.22e-05	7.20e-05	6.56e-02	1.27e-02	3.76e-02	0.28	270
5	200	2.51e-05	2.76e-06	9.74e-06	2.22e-02	5.97e-03	1.27e-02	1.97	612
10	200	1.13e-04	2.39e-05	5.82e-05	5.40e-02	2.29e-02	3.87e-02	3.55	784
10	500	9.06e-06	3.53e-06	6.55e-06	2.33e-02	1.26e-02	1.91e-02	28.54	988
20	500	5.09e-05	2.94e-05	3.81e-05	5.74e-02	3.38e-02	4.51e-02	39.33	1000
20	1000	1.23e-05	5.67e-06	8.58e-06	2.37e-02	1.17e-02	1.83e-02	170.36	1000

Tabella 4.14: Risultati dell'algoritmo MS_BPT con politica di gestione dei nodi basata su "deque" eseguito su istanze di tipo **alm-u**. Soluzione iniziale generata mediante l'euristica "List Scheduling".

		$\frac{C_{max}-\bar{C}}{\bar{C}}$			$\frac{C_{SI}-\bar{C}}{\bar{C}}$			Media	
m	n	max	min	media	max	min	media	sec	iter.
2	50	1.80e-04	8.21e-07	2.94e-05	2.16e-03	2.56e-05	6.19e-04	0.01	93
2	100	1.45e-05	9.90e-07	5.57e-06	2.86e-04	1.44e-06	9.90e-05	0.03	141
3	100	2.16e-05	1.42e-06	9.90e-06	9.67e-04	9.34e-06	3.95e-04	0.09	270
3	200	4.21e-06	2.26e-07	1.30e-06	1.67e-04	1.02e-05	7.57e-05	0.60	534
5	100	1.48e-04	1.71e-05	7.06e-05	2.33e-03	3.03e-04	1.27e-03	0.13	253
5	200	3.88e-05	3.23e-06	1.14e-05	7.53e-04	2.99e-05	3.21e-04	1.04	693
10	200	8.88e-05	2.30e-05	6.12e-05	2.31e-03	4.47e-04	1.20e-03	1.32	894
10	500	2.36e-05	3.46e-06	1.10e-05	5.29e-04	1.09e-04	2.21e-04	4.21	1000
20	500	1.40e-04	6.56e-05	1.00e-04	1.21e-03	4.21e-04	8.05e-04	6.10	1000
20	1000	4.07e-05	1.34e-05	2.05e-05	3.56e-04	5.87e-05	1.83e-04	21.34	1001

Tabella 4.15: Risultati dell'algoritmo MS_BPT con politica "FIFO" eseguito su istanze di tipo **alm-u**. Soluzione iniziale generata mediante l'euristica "Longest Processing Time".

		$\frac{C_{max}-C}{C}$			$\frac{C_{SI}-C}{C}$			Media	
m	n	max	min	media	max	min	media	sec	iter.
2	50	1.80e-04	8.21e-07	2.94e-05	2.16e-03	2.56e-05	6.19e-04	0.01	93
2	100	1.45e-05	9.90e-07	5.57e-06	2.86e-04	1.44e-06	9.90e-05	0.03	141
3	100	2.16e-05	1.42e-06	9.90e-06	9.67e-04	9.34e-06	3.95e-04	0.10	270
3	200	4.21e-06	2.26e-07	1.30e-06	1.67e-04	1.02e-05	7.57e-05	0.61	534
5	100	1.48e-04	1.71e-05	7.06e-05	2.33e-03	3.03e-04	1.27e-03	0.12	253
5	200	3.88e-05	3.23e-06	1.14e-05	7.53e-04	2.99e-05	3.21e-04	1.04	693
10	200	8.88e-05	2.30e-05	6.12e-05	2.31e-03	4.47e-04	1.20e-03	1.33	894
10	500	2.36e-05	3.46e-06	1.10e-05	5.29e-04	1.09e-04	2.21e-04	4.22	1000
20	500	1.40e-04	6.56e-05	1.00e-04	1.21e-03	4.21e-04	8.05e-04	6.09	1000
20	1000	4.07e-05	1.34e-05	2.05e-05	3.56e-04	5.87e-05	1.83e-04	21.19	1001

Tabella 4.16: Risultati dell'algoritmo MS_BPT con politica di gestione dei nodi basata su "deque" eseguito su istanze di tipo **alm-u**. Soluzione iniziale generata mediante l'euristica "Longest Processing Time".

		$\frac{C_{max}-C}{C}$			$\frac{C_{SI}-C}{C}$			Media	
m	n	max	min	media	max	min	media	sec	iter.
2	50	6.24e-04	4.57e-06	2.05e-04	6.24e-04	4.57e-06	2.39e-04	0.00	55
2	100	2.04e-04	2.24e-06	6.09e-05	2.04e-04	4.42e-06	6.81e-05	0.01	109
3	100	2.54e-04	1.48e-05	1.16e-04	2.55e-04	1.48e-05	1.27e-04	0.03	104
3	200	9.75e-05	4.38e-06	3.42e-05	1.06e-04	4.38e-06	3.83e-05	0.16	217
5	100	8.24e-04	1.08e-04	3.97e-04	8.24e-04	1.08e-04	4.31e-04	0.05	113
5	200	1.42e-04	3.16e-05	8.96e-05	1.42e-04	3.16e-05	9.72e-05	0.34	233
10	200	5.47e-04	1.03e-04	3.85e-04	5.47e-04	1.03e-04	3.86e-04	0.49	201
10	500	1.03e-04	2.88e-05	5.67e-05	1.03e-04	2.88e-05	6.04e-05	7.63	621
20	500	1.84e-04	5.97e-05	1.16e-04	2.28e-04	7.39e-05	1.27e-04	9.28	768
20	1000	3.68e-05	1.34e-05	2.78e-05	4.41e-05	1.34e-05	2.92e-05	53.46	1000

Tabella 4.17: Risultati dell'algoritmo MS_BPT con politica "FIFO" eseguito su istanze di tipo **alm-u**. Soluzione iniziale generata mediante l'euristica "Multifit".

		$\frac{C_{max}-C}{C}$			$\frac{C_{SI}-C}{C}$			Media	
m	n	max	min	media	max	min	media	sec	iter.
2	50	6.24e-04	4.57e-06	2.05e-04	6.24e-04	4.57e-06	2.39e-04	0.00	55
2	100	2.04e-04	2.24e-06	6.09e-05	2.04e-04	4.42e-06	6.81e-05	0.01	109
3	100	2.54e-04	1.48e-05	1.16e-04	2.55e-04	1.48e-05	1.27e-04	0.03	104
3	200	9.75e-05	4.38e-06	3.42e-05	1.06e-04	4.38e-06	3.83e-05	0.16	217
5	100	8.24e-04	1.08e-04	3.97e-04	8.24e-04	1.08e-04	4.31e-04	0.05	113
5	200	1.42e-04	3.16e-05	8.96e-05	1.42e-04	3.16e-05	9.72e-05	0.34	233
10	200	5.47e-04	1.03e-04	3.85e-04	5.47e-04	1.03e-04	3.86e-04	0.49	201
10	500	1.03e-04	2.88e-05	5.67e-05	1.03e-04	2.88e-05	6.04e-05	7.70	621
20	500	1.84e-04	5.97e-05	1.16e-04	2.28e-04	7.39e-05	1.27e-04	9.37	768
20	1000	3.68e-05	1.34e-05	2.78e-05	4.41e-05	1.34e-05	2.92e-05	53.34	1000

Tabella 4.18: Risultati dell'algoritmo MS_BPT con politica di gestione dei nodi basata su "deque" eseguito su istanze di tipo **alm-u**. Soluzione iniziale generata mediante l'euristica "Multifit".

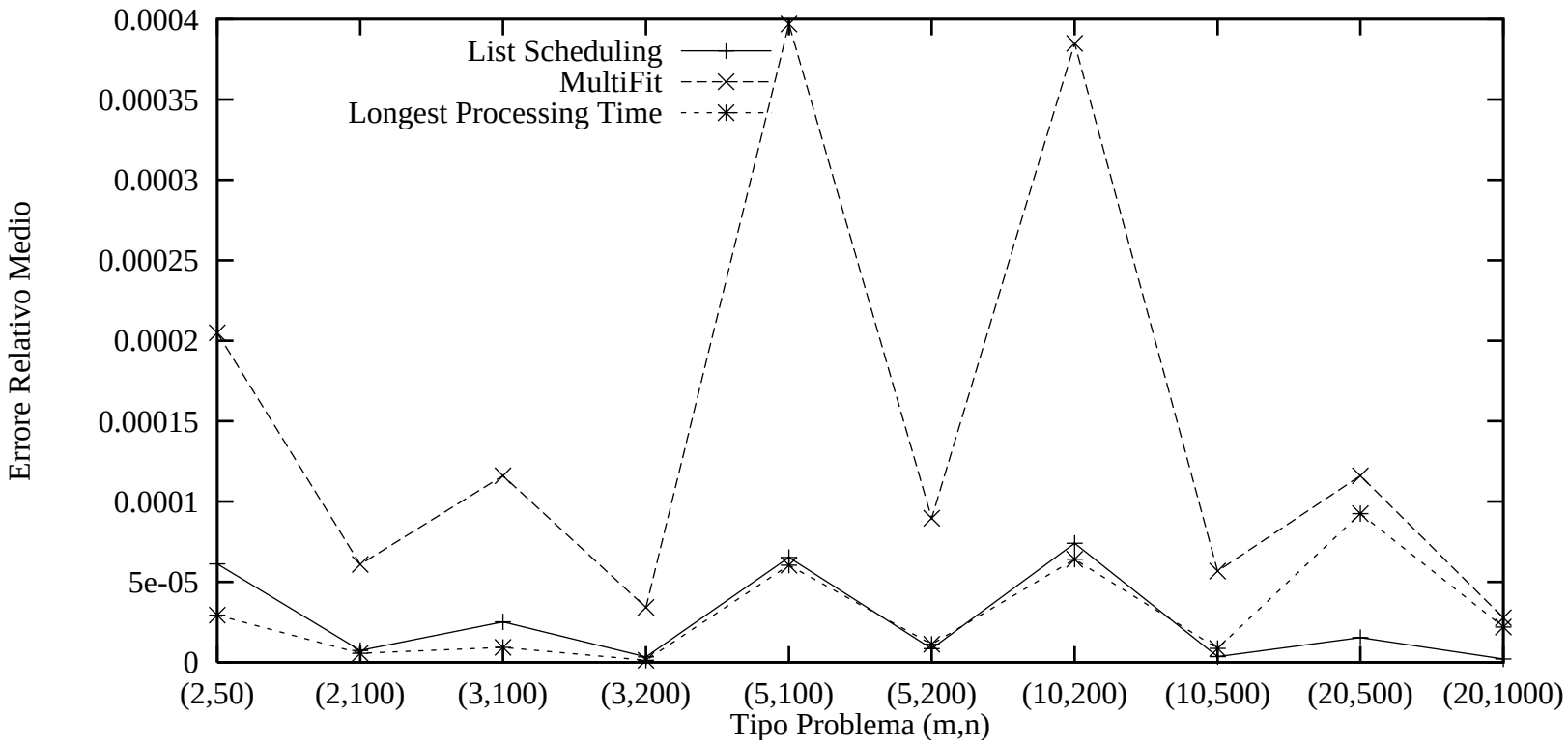

Figura 4.1: Confronto dei risultati ottenuti mediante l'algoritmo MS_SPT con politica "FIFO" eseguito a partire da soluzioni diverse su istanze di tipo **alm-u**

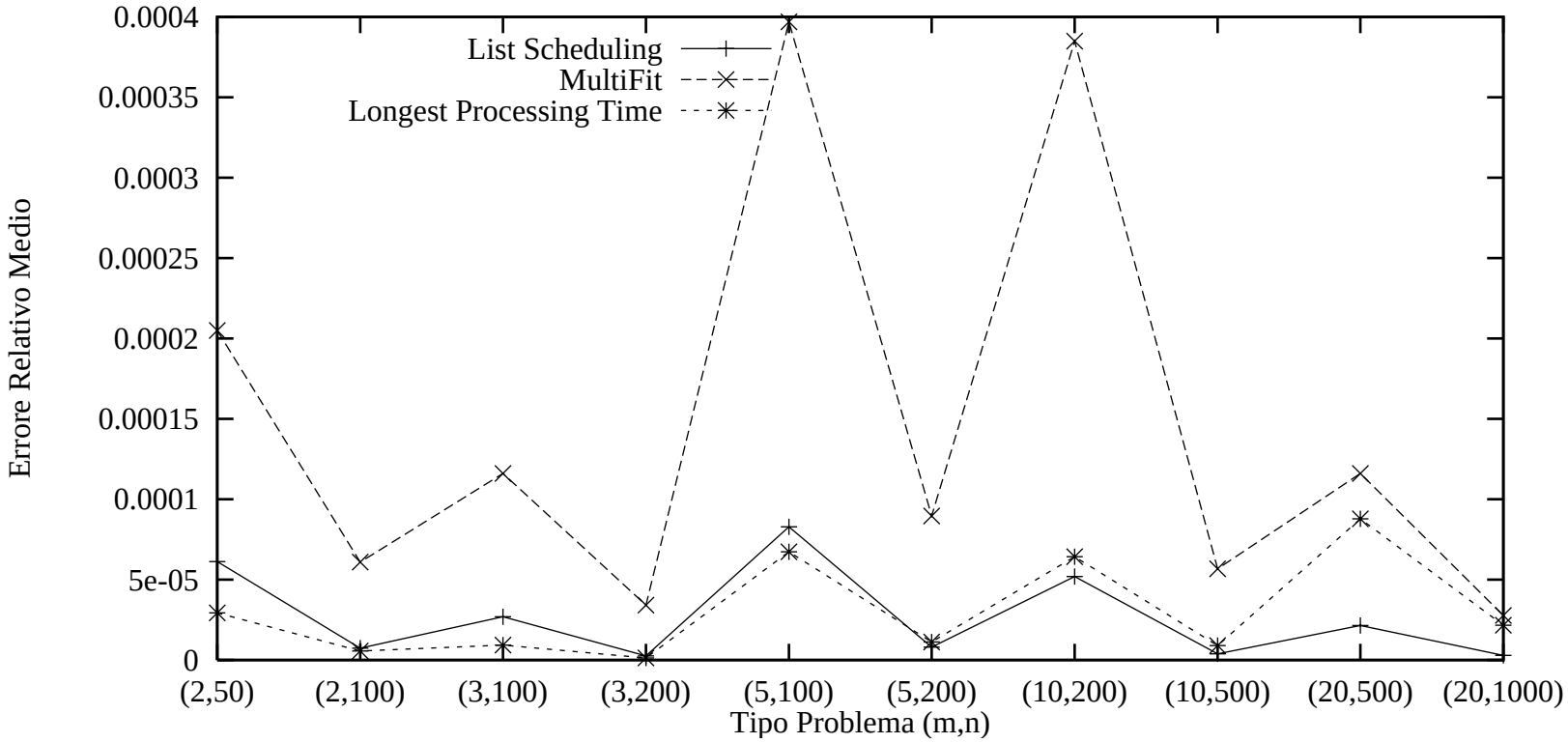

Figura 4.2: Confronto dei risultati ottenuti mediante l'algoritmo MS_SPT con politica "DEQUE" eseguito a partire da soluzioni diverse su istanze di tipo **alm-u**

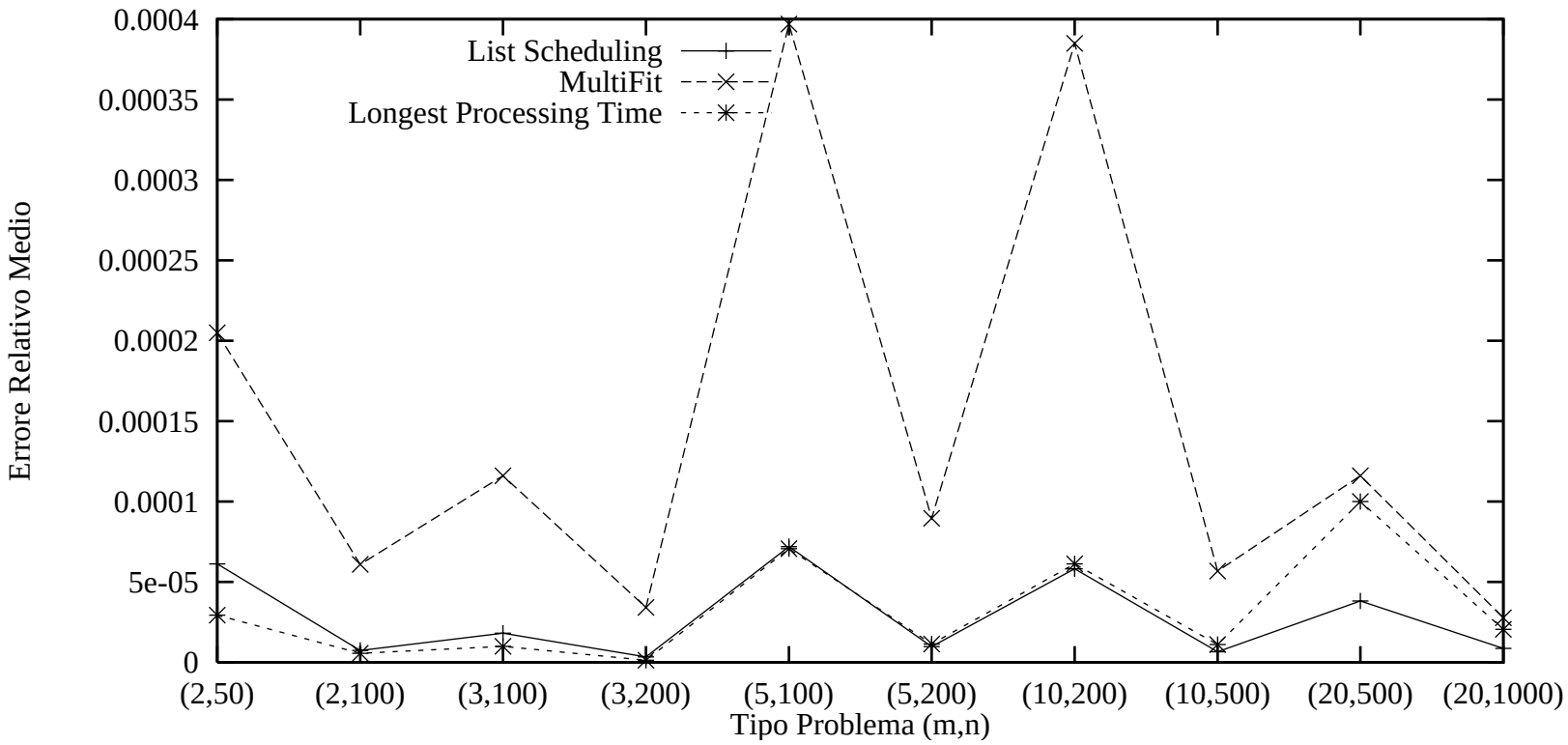

Figura 4.3: Confronto dei risultati ottenuti mediante l'algoritmo MS_BPT con politica "FIFO" eseguito a partire da soluzioni diverse su istanze di tipo **alm-u**

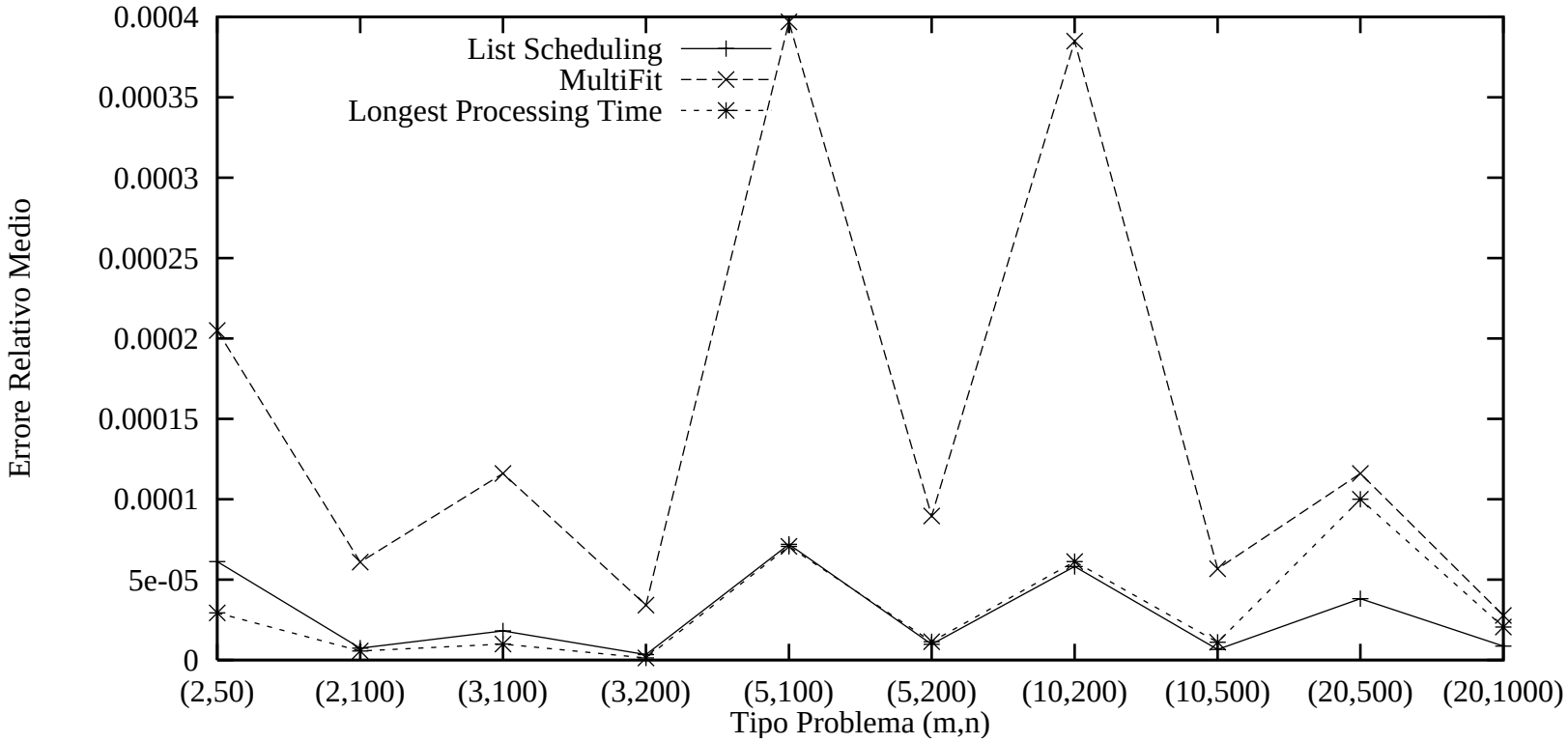

Figura 4.4: Confronto dei risultati ottenuti mediante l'algoritmo MS_BPT con politica "DEQUE" eseguito a partire da soluzioni diverse su istanze di tipo **alm-u**

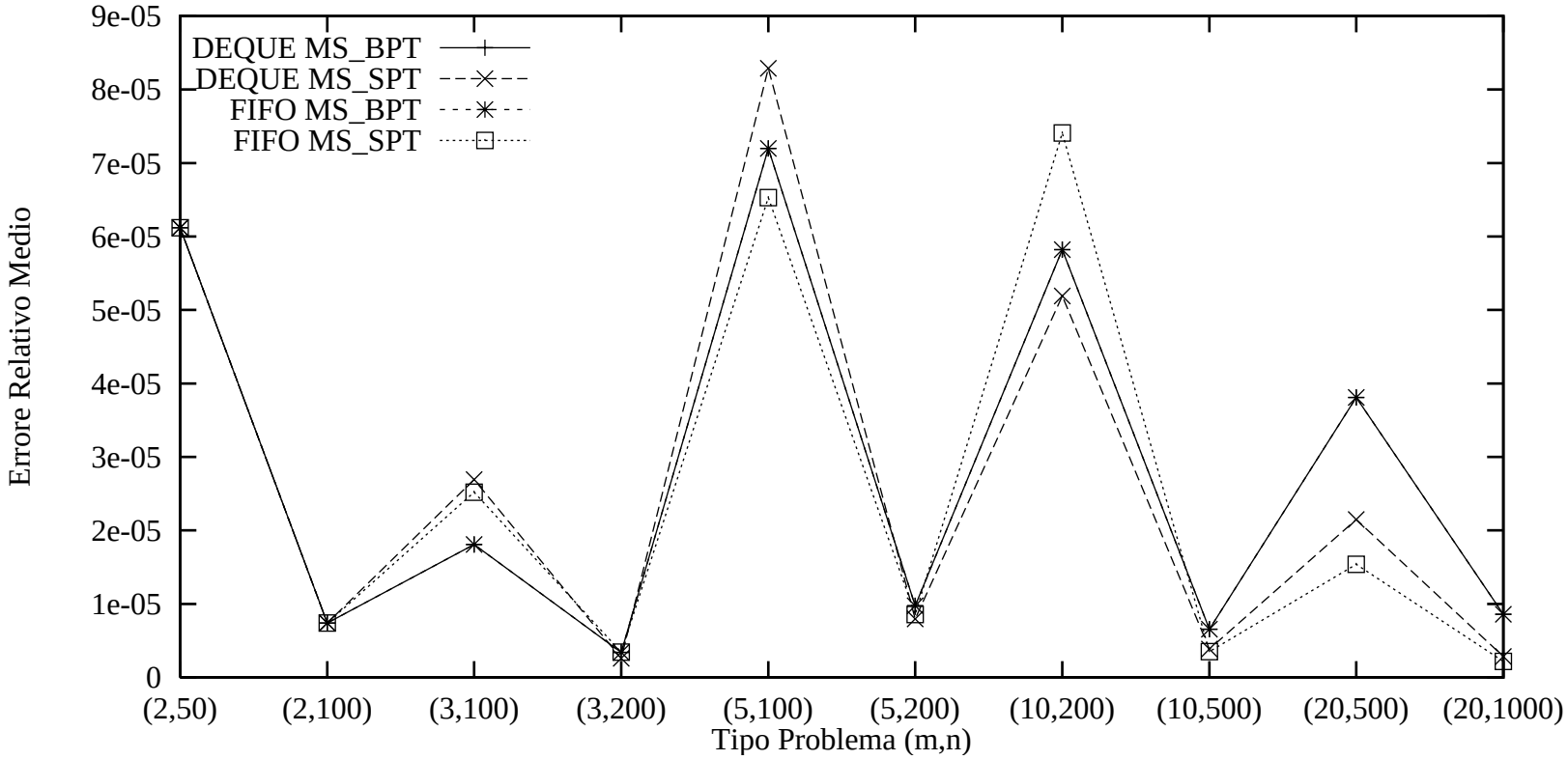

Figura 4.5: Confronto dei risultati ottenuti mediante algoritmi diversi eseguiti su istanze di tipo **alm-u** a partire da soluzioni iniziali ottenute con l'euristica "List Scheduling"

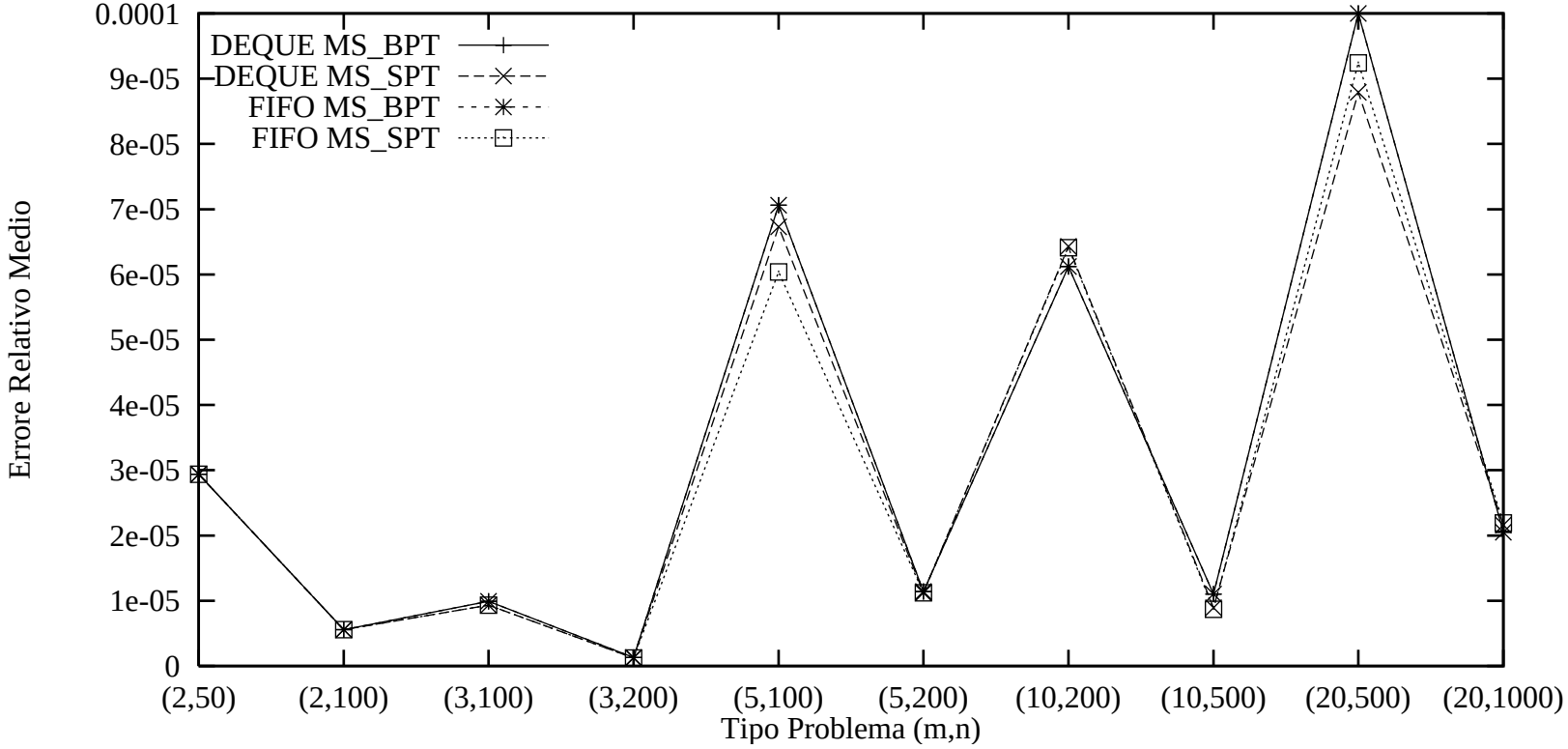

Figura 4.6: Confronto dei risultati ottenuti mediante algoritmi diversi eseguiti su istanze di tipo **alm-u** a partire da soluzioni iniziali ottenute con l'euristica "Longest Processing Time"

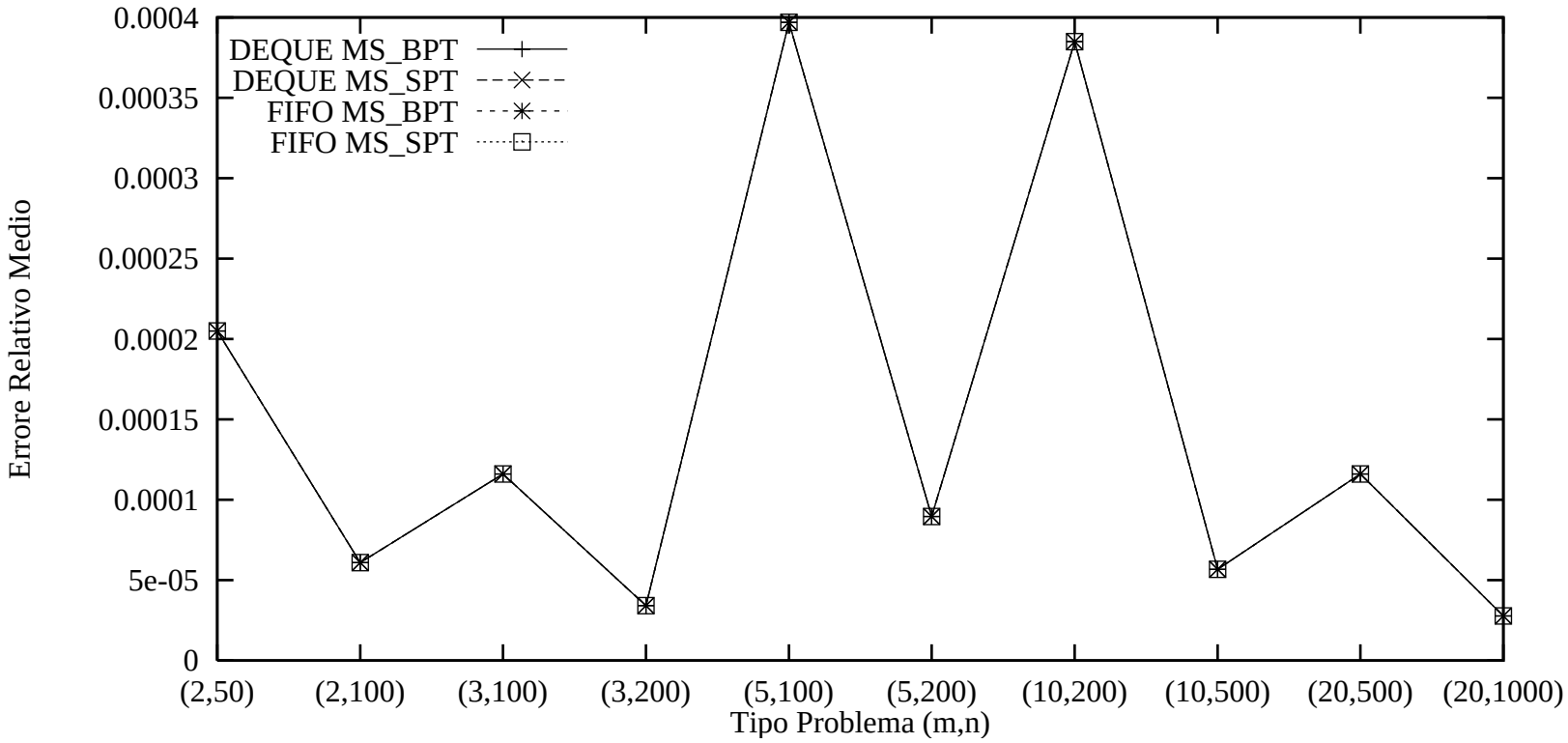

Figura 4.7: Confronto dei risultati ottenuti mediante algoritmi diversi eseguiti su istanze di tipo **alm-u** a partire da soluzioni iniziali ottenute con l'euristica "Multifit"

4.6 Risultati relativi alle istanze di tipo alm-1

In questo paragrafo vengono presentati i risultati ottenuti mediante le varianti degli algoritmi MS_SPT e MS_BPT applicati alle istanze di tipo **alm-1**. I risultati sono riportati nelle Tabelle da 4.19 a 4.30, e visualizzati mediante i grafici nelle Figure da 4.8 a 4.14.

È possibile osservare che i risultati migliori sono ottenuti dalle varianti di MS_SPT nel caso in cui la soluzione iniziale venga ricavata con le euristiche "List Scheduling" e "Longest Processing Time". In particolare, nel caso delle istanze aventi un numero di lavori e macchine pari a $(2, 50)$, $(3, 100)$, $(3, 200)$, $(5, 100)$ e $(5, 200)$, si ottengono risultati migliori partendo da soluzioni iniziali ricavate con l'euristica "Longest Processing Time", mentre nel caso in cui il numero dei lavori e delle macchine sia pari a $(10, 200)$, $(10, 500)$, $(20, 500)$ e $(20, 1000)$, la soluzione migliore si ottiene partendo da soluzioni iniziali ricavate mediante l'euristica "List Scheduling". Nel caso delle due versioni dell'euristica MS_BPT, invece, le soluzioni ottenute a partire da una soluzione iniziale ricavata con l'euristica "List Scheduling" (o "Longest Processing Time") e quelle ottenute a partire da una soluzione iniziale ricavata con l'euristica "Multifit" differiscono di poco.

Ponendosi nell'ottica delle capacità di miglioramento degli algoritmi proposti nei confronti della soluzione iniziale, quando la soluzione iniziale è ricavata mediante l'euristica "List Scheduling" la versione dell'algoritmo che ottiene, in tutti i casi, la soluzione migliore è MS_SPT con politica "FIFO". Il tempo impiegato da tale euristica è, in tutti i casi, notevolmente minore di quello impiegato dall'euristica MS_SPT con politica di gestione dei nodi basata su "deque". Invece le versioni dell'euristica MS_BPT hanno tempi di esecuzione paragonabili.

Nel caso in cui la soluzione iniziale sia ricavata mediante l'euristica "Longest Processing Time", le soluzioni ottenute dalle varie versioni degli algoritmi sono molto vicine nella maggioranza delle istanze testate. Per quanto riguarda le istanze con numero di macchine e numero di lavori pari a $(2, 50)$,

$(2, 100)$, $(3, 100)$, $(5, 100)$ e $(5, 200)$, le soluzioni ottenute dai diversi algoritmi sono paragonabili. Tra tutte le soluzioni comunque la migliore è quella ottenuta dall'algoritmo MS_SPT con politica "FIFO". Per le istanze con numero di macchine e numero di lavori pari a $(10, 200)$, $(10, 500)$, $(20, 500)$ e $(20, 1000)$ invece, le due varianti dell'algoritmo MS_BPT ottengono risultati peggiori di quelli ottenuti con gli algoritmi MS_SPT; l'algoritmo MS_SPT con politica di gestione dei nodi basata su "deque" ottiene la soluzione migliore.

Il tempo impiegato dall'euristica MS_SPT con politica "FIFO" è, in tutti i casi, maggiore di quello dell'euristica MS_SPT con politica di gestione dei nodi basata su "deque". Invece le versioni dell'euristica MS_BPT hanno tempi di esecuzione paragonabili.

Nel caso in cui la soluzione iniziale sia ricavata mediante l'euristica "Multifit", le soluzioni ottenute dalle varie versioni degli algoritmi sono paragonabili. Si osserva una notevole differenza tra la soluzione ottenuta mediante gli algoritmi MS_BPT e la soluzione ricavata con gli algoritmi MS_SPT nel caso in cui l'istanza abbia un numero di macchine e un numero di lavori pari a $(20, 500)$. Il tempo impiegato dall'euristica MS_SPT con politica "FIFO" è, nella maggior parte dei casi, minore di quello dell'euristica MS_SPT con politica di gestione dei nodi basata su "deque". Invece nel caso dell'euristica MS_BPT la versione con politica di gestione dei nodi basata su "deque" è sempre più lenta.

Complessivamente, su queste istanze, più "difficili" in natura, gli algoritmi proposti riescono ad individuare un errore relativo medio sempre inferiore a 10^{-3}.

La soluzione iniziale è migliorata di almeno un ordine di grandezza (salvo alcune eccezioni riguardanti MS_BPT) nel caso in cui la soluzione iniziale sia costruita mediante le euristiche "List Scheduling" e "Longest Processing Time". Analogamente a quanto accade nel caso delle istanze **alm-u**, invece, gli algoritmi non riescono solitamente ad apportare miglioramenti significativi se si parte da soluzioni generate mediante "Multifit".

		$\frac{C_{max}-C}{C}$			$\frac{C_{SL}-C}{C}$			Media	
m	n	max	min	media	max	min	media	sec	iter.
2	50	6.34e-04	0	2.42e-04	2.11e-02	4.52e-03	1.41e-02	0.00	27
2	100	3.29e-04	0	1.82e-04	1.36e-02	2.24e-03	7.31e-03	0.03	71
3	100	4.88e-04	0	2.25e-04	2.30e-02	9.40e-03	1.48e-02	0.04	61
3	200	2.38e-04	0	1.60e-04	1.12e-02	2.51e-03	6.78e-03	0.29	149
5	100	1.57e-03	0	9.06e-04	4.72e-02	2.07e-02	2.67e-02	0.10	106
5	200	7.67e-04	0	3.72e-04	2.42e-02	6.52e-03	1.25e-02	0.79	210
10	200	1.61e-03	7.25e-04	1.07e-03	4.64e-02	1.46e-02	2.83e-02	1.38	236
10	500	8.94e-04	2.79e-04	3.90e-04	2.04e-02	7.58e-03	1.35e-02	14.18	528
20	500	1.33e-03	5.69e-04	9.89e-04	4.33e-02	1.53e-02	2.69e-02	49.82	648
20	1000	6.34e-04	2.76e-04	4.75e-04	1.72e-02	6.11e-03	1.09e-02	262.87	1000

Tabella 4.19: Risultati dell'algoritmo MS_SPT con politica "FIFO" eseguito su istanze di tipo **alm-1**. Soluzione iniziale generata mediante l'euristica "List Scheduling".

		$\frac{C_{max}-C}{C}$			$\frac{C_{SL}-C}{C}$			Media	
m	n	max	min	media	max	min	media	sec	iter.
2	50	6.34e-04	0	2.42e-04	2.11e-02	4.52e-03	1.41e-02	0.01	27
2	100	3.29e-04	0	1.82e-04	1.36e-02	2.24e-03	7.31e-03	0.03	71
3	100	4.88e-04	0	2.25e-04	2.30e-02	9.40e-03	1.48e-02	0.04	61
3	200	2.38e-04	0	1.60e-04	1.12e-02	2.51e-03	6.78e-03	0.30	149
5	100	1.56e-03	0	8.27e-04	4.72e-02	2.07e-02	2.67e-02	0.12	118
5	200	8.07e-04	0	3.78e-04	2.42e-02	6.52e-03	1.25e-02	0.66	213
10	200	1.61e-03	7.25e-04	1.15e-03	4.64e-02	1.46e-02	2.83e-02	1.28	224
10	500	1.19e-03	2.79e-04	4.51e-04	2.04e-02	7.58e-03	1.35e-02	15.58	544
20	500	2.48e-03	5.94e-04	1.29e-03	4.33e-02	1.53e-02	2.69e-02	53.20	560
20	1000	9.16e-04	2.76e-04	5.37e-04	1.72e-02	6.11e-03	1.09e-02	332.85	1000

Tabella 4.20: Risultati dell'algoritmo MS_SPT con politica di gestione dei nodi basata su "deque" eseguito su istanze di tipo **alm-1**. Soluzione iniziale generata mediante l'euristica "List Scheduling".

		$\frac{C_{max}-\bar{C}}{\bar{C}}$			$\frac{C_{SI}-\bar{C}}{\bar{C}}$			Media	
m	n	max	min	media	max	min	media	sec	iter.
2	50	6.34e-04	0	3.60e-04	1.30e-02	6.87e-03	1.01e-02	0.01	61
2	100	3.29e-04	0	1.82e-04	2.56e-03	2.74e-04	9.89e-04	0.02	72
3	100	4.88e-04	0	2.25e-04	4.98e-03	1.79e-03	3.71e-03	0.03	60
3	200	2.38e-04	0	1.60e-04	2.45e-03	4.48e-04	1.59e-03	0.26	149
5	100	7.85e-04	0	6.78e-04	1.45e-02	6.28e-03	1.05e-02	0.09	108
5	200	3.90e-04	0	2.98e-04	8.35e-03	5.17e-03	6.80e-03	0.56	174
10	200	2.33e-03	7.25e-04	1.30e-03	1.45e-02	8.12e-03	1.03e-02	1.40	241
10	500	8.94e-04	2.86e-04	5.11e-04	7.52e-03	3.24e-03	5.61e-03	17.09	550
20	500	3.32e-03	5.94e-04	1.75e-03	8.81e-03	3.32e-03	6.76e-03	61.37	574
20	1000	1.47e-03	2.89e-04	7.77e-04	6.98e-03	3.34e-03	5.43e-03	391.88	1000

Tabella 4.21: Risultati dell'algoritmo MS_SPT con politica "FIFO" eseguito su istanze di tipo **alm-1**. Soluzione iniziale generata mediante l'euristica "Longest Processing Time".

		$\frac{C_{max}-C}{C}$			$\frac{C_{SI}-C}{C}$			Media	
m	n	max	min	media	max	min	media	sec	iter.
2	50	6.34e-04	0	3.60e-04	1.30e-02	6.87e-03	1.01e-02	0.01	61
2	100	3.29e-04	0	1.82e-04	2.56e-03	2.74e-04	9.89e-04	0.02	72
3	100	4.88e-04	0	2.25e-04	4.98e-03	1.79e-03	3.71e-03	0.04	60
3	200	2.38e-04	0	1.60e-04	2.45e-03	4.48e-04	1.59e-03	0.27	149
5	100	2.16e-03	7.18e-04	8.93e-04	1.45e-02	6.28e-03	1.05e-02	0.10	118
5	200	3.90e-04	0	2.98e-04	8.35e-03	5.17e-03	6.80e-03	0.57	174
10	200	1.57e-03	7.25e-04	9.91e-04	1.45e-02	8.12e-03	1.03e-02	1.29	226
10	500	6.22e-04	2.79e-04	3.32e-04	7.52e-03	3.24e-03	5.61e-03	13.44	526
20	500	3.32e-03	5.69e-04	9.46e-04	8.81e-03	3.32e-03	6.76e-03	33.04	522
20	1000	6.34e-04	2.76e-04	4.16e-04	6.98e-03	3.34e-03	5.43e-03	260.54	1000

Tabella 4.22: Risultati dell'algoritmo MS_SPT con politica di gestione dei nodi basata su "deque" eseguito su istanze di tipo **alm-1**. Soluzione iniziale generata mediante l'euristica "Longest Processing Time".

		$\frac{C_{max}-C}{C}$			$\frac{C_{SI}-C}{C}$			Media	
m	n	max	min	media	max	min	media	sec	iter.
2	50	5.07e-03	5.65e-04	1.79e-03	1.30e-02	5.79e-04	5.53e-03	0.01	57
2	100	4.91e-03	0	1.33e-03	4.91e-03	0	1.80e-03	0.01	100
3	100	4.17e-03	0	1.39e-03	4.71e-03	0	1.87e-03	0.03	94
3	200	1.38e-03	2.21e-04	5.25e-04	1.99e-03	2.25e-04	1.04e-03	0.17	241
5	100	5.46e-03	1.44e-03	2.94e-03	1.67e-02	2.16e-03	8.45e-03	0.08	131
5	200	7.11e-03	1.15e-03	2.21e-03	7.99e-03	1.38e-03	3.52e-03	0.52	252
10	200	8.88e-03	2.91e-03	4.29e-03	1.74e-02	2.97e-03	9.23e-03	1.36	279
10	500	3.03e-03	2.86e-04	1.56e-03	6.01e-03	1.53e-03	2.72e-03	15.04	629
20	500	1.16e-02	1.86e-03	5.12e-03	1.18e-02	5.98e-03	9.32e-03	66.58	646
20	1000	3.20e-03	1.10e-03	2.22e-03	4.97e-03	2.43e-03	3.39e-03	288.33	1000

Tabella 4.23: Risultati dell'algoritmo MS_SPT con politica "FIFO" eseguito su istanze di tipo **alm-1**. Soluzione iniziale generata mediante l'euristica "Multifit".

		$\frac{C_{max}-C}{C}$			$\frac{C_{SI}-C}{C}$			Media	
m	n	max	min	media	max	min	media	sec	iter.
2	50	5.07e-03	5.65e-04	1.79e-03	1.30e-02	5.79e-04	5.53e-03	0.00	57
2	100	4.91e-03	0	1.33e-03	4.91e-03	0	1.80e-03	0.01	100
3	100	4.17e-03	0	1.39e-03	4.71e-03	0	1.87e-03	0.02	94
3	200	1.38e-03	2.21e-04	5.25e-04	1.99e-03	2.25e-04	1.04e-03	0.17	241
5	100	5.46e-03	1.44e-03	2.94e-03	1.67e-02	2.16e-03	8.45e-03	0.08	131
5	200	7.11e-03	1.15e-03	2.21e-03	7.99e-03	1.38e-03	3.52e-03	0.58	252
10	200	6.22e-03	2.95e-03	4.41e-03	1.74e-02	2.97e-03	9.23e-03	1.56	267
10	500	3.03e-03	8.59e-04	1.67e-03	6.01e-03	1.53e-03	2.72e-03	21.32	621
20	500	1.16e-02	2.28e-03	5.17e-03	1.18e-02	5.98e-03	9.32e-03	72.11	559
20	1000	3.20e-03	1.10e-03	1.89e-03	4.97e-03	2.43e-03	3.39e-03	199.47	1000

Tabella 4.24: Risultati dell'algoritmo MS_SPT con politica di gestione dei nodi basata su "deque" eseguito su istanze di tipo **alm-1**. Soluzione iniziale generata mediante l'euristica "Multifit".

		$\frac{C_{max}-\bar{C}}{\bar{C}}$			$\frac{C_{SI}-\bar{C}}{\bar{C}}$			Media	
m	n	max	min	media	max	min	media	sec	iter.
2	50	6.34e-04	0	2.42e-04	2.11e-02	4.52e-03	1.41e-02	0.00	27
2	100	3.29e-04	0	1.82e-04	1.36e-02	2.24e-03	7.31e-03	0.03	71
3	100	4.88e-04	0	3.14e-04	2.30e-02	9.40e-03	1.48e-02	0.05	83
3	200	2.38e-04	0	1.60e-04	1.12e-02	2.51e-03	6.78e-03	0.29	148
5	100	1.57e-03	7.18e-04	9.72e-04	4.72e-02	2.07e-02	2.67e-02	0.11	119
5	200	7.67e-04	0	4.15e-04	2.42e-02	6.52e-03	1.25e-02	0.59	200
10	200	5.44e-03	1.48e-03	2.74e-03	4.64e-02	1.46e-02	2.83e-02	1.31	233
10	500	1.79e-03	2.96e-04	8.50e-04	2.04e-02	7.58e-03	1.35e-02	14.47	521
20	500	1.19e-02	1.71e-03	6.49e-03	4.33e-02	1.53e-02	2.69e-02	33.77	523
20	1000	6.87e-03	5.78e-04	3.08e-03	1.72e-02	6.11e-03	1.09e-02	261.00	1000

Tabella 4.25: Risultati dell'algoritmo MS_BPT con politica "FIFO" eseguito su istanze di tipo **alm-1**. Soluzione iniziale generata mediante l'euristica "List Scheduling".

		$\frac{C_{max}-C}{C}$			$\frac{C_{SI}-C}{C}$			Media	
m	n	max	min	media	max	min	media	sec	iter.
2	50	6.34e-04	0	2.42e-04	2.11e-02	4.52e-03	1.41e-02	0.00	27
2	100	3.29e-04	0	1.82e-04	1.36e-02	2.24e-03	7.31e-03	0.03	71
3	100	4.88e-04	0	3.14e-04	2.30e-02	9.40e-03	1.48e-02	0.06	87
3	200	2.38e-04	0	1.60e-04	1.12e-02	2.51e-03	6.78e-03	0.28	148
5	100	1.57e-03	7.18e-04	9.72e-04	4.72e-02	2.07e-02	2.67e-02	0.10	113
5	200	7.67e-04	0	4.15e-04	2.42e-02	6.52e-03	1.25e-02	0.59	200
10	200	7.46e-03	1.48e-03	3.55e-03	4.64e-02	1.46e-02	2.83e-02	1.56	225
10	500	1.79e-03	5.72e-04	1.09e-03	2.04e-02	7.58e-03	1.35e-02	16.25	521
20	500	1.19e-02	1.71e-03	5.57e-03	4.33e-02	1.53e-02	2.69e-02	34.04	540
20	1000	6.87e-03	5.78e-04	2.83e-03	1.72e-02	6.11e-03	1.09e-02	260.48	1000

Tabella 4.26: Risultati dell'algoritmo MS_BPT con politica di gestione dei nodi basata su "deque" eseguito su istanze di tipo **alm-1**. Soluzione iniziale generata mediante l'euristica "List Scheduling".

		$\frac{C_{max}-\tilde{C}}{\tilde{C}}$			$\frac{C_{SI}-\tilde{C}}{\tilde{C}}$			Media	
m	n	max	min	media	max	min	media	sec	iter.
2	50	6.34e-04	0	3.60e-04	1.30e-02	6.87e-03	1.01e-02	0.01	61
2	100	3.29e-04	0	1.82e-04	2.56e-03	2.74e-04	9.89e-04	0.02	72
3	100	4.88e-04	0	2.25e-04	4.98e-03	1.79e-03	3.71e-03	0.03	64
3	200	2.38e-04	0	1.60e-04	2.45e-03	4.48e-04	1.59e-03	0.26	149
5	100	5.49e-03	7.18e-04	1.60e-03	1.45e-02	6.28e-03	1.05e-02	0.10	115
5	200	1.15e-03	0	4.84e-04	8.35e-03	5.17e-03	6.80e-03	0.56	189
10	200	1.31e-02	1.61e-03	7.34e-03	1.45e-02	8.12e-03	1.03e-02	1.89	204
10	500	7.52e-03	2.98e-04	3.79e-03	7.52e-03	3.24e-03	5.61e-03	29.41	505
20	500	8.32e-03	3.32e-03	6.32e-03	8.81e-03	3.32e-03	6.76e-03	38.54	502
20	1000	6.98e-03	3.34e-03	5.25e-03	6.98e-03	3.34e-03	5.43e-03	356.91	1000

Tabella 4.27: Risultati dell'algoritmo MS_BPT con politica "FIFO" eseguito su istanze di tipo **alm-1**. Soluzione iniziale generata mediante l'euristica "Longest Processing Time".

		$\frac{C_{max}-C}{C}$			$\frac{C_{SI}-C}{C}$			Media	
m	n	max	min	media	max	min	media	sec	iter.
2	50	6.34e-04	0	3.60e-04	1.30e-02	6.87e-03	1.01e-02	0.01	61
2	100	3.29e-04	0	1.82e-04	2.56e-03	2.74e-04	9.89e-04	0.02	72
3	100	4.88e-04	0	2.25e-04	4.98e-03	1.79e-03	3.71e-03	0.03	64
3	200	2.38e-04	0	1.60e-04	2.45e-03	4.48e-04	1.59e-03	0.26	149
5	100	5.49e-03	7.18e-04	1.60e-03	1.45e-02	6.28e-03	1.05e-02	0.09	115
5	200	1.15e-03	0	4.84e-04	8.35e-03	5.17e-03	6.80e-03	0.57	189
10	200	1.31e-02	1.61e-03	7.34e-03	1.45e-02	8.12e-03	1.03e-02	2.28	204
10	500	7.52e-03	2.98e-04	3.79e-03	7.52e-03	3.24e-03	5.61e-03	34.85	505
20	500	8.32e-03	3.32e-03	6.32e-03	8.81e-03	3.32e-03	6.76e-03	48.90	502
20	1000	6.98e-03	3.34e-03	5.25e-03	6.98e-03	3.34e-03	5.43e-03	453.92	1000

Tabella 4.28: Risultati dell'algoritmo MS_BPT con politica di gestione dei nodi basata su "deque" eseguito su istanze di tipo **alm-1**. Soluzione iniziale generata mediante l'euristica "Longest Processing Time".

		$\frac{C_{max}-C}{C}$			$\frac{C_{SI}-C}{C}$			Media	
m	n	max	min	media	max	min	media	sec	iter.
2	50	5.07e-03	5.65e-04	1.79e-03	1.30e-02	5.79e-04	5.53e-03	0.01	57
2	100	4.91e-03	0	1.33e-03	4.91e-03	0	1.80e-03	0.01	100
3	100	4.17e-03	0	1.39e-03	4.71e-03	0	1.87e-03	0.03	94
3	200	1.38e-03	2.21e-04	5.03e-04	1.99e-03	2.25e-04	1.04e-03	0.18	241
5	100	6.25e-03	1.44e-03	3.02e-03	1.67e-02	2.16e-03	8.45e-03	0.08	138
5	200	7.11e-03	1.15e-03	2.17e-03	7.99e-03	1.38e-03	3.52e-03	0.54	251
10	200	8.44e-03	2.97e-03	5.19e-03	1.74e-02	2.97e-03	9.23e-03	1.11	241
10	500	3.03e-03	8.59e-04	1.72e-03	6.01e-03	1.53e-03	2.72e-03	16.94	625
20	500	1.18e-02	5.98e-03	8.59e-03	1.18e-02	5.98e-03	9.32e-03	40.81	529
20	1000	3.52e-03	1.52e-03	2.64e-03	4.97e-03	2.43e-03	3.39e-03	193.64	1000

Tabella 4.29: Risultati dell'algoritmo MS_BPT con politica "FIFO" eseguito su istanze di tipo **alm-1**. Soluzione iniziale generata mediante l'euristica "Multifit".

		$\frac{C_{max}-C}{C}$			$\frac{C_{SI}-C}{C}$			Media	
m	n	max	min	media	max	min	media	sec	iter.
2	50	5.07e-03	5.65e-04	1.79e-03	1.30e-02	5.79e-04	5.53e-03	0.01	57
2	100	4.91e-03	0	1.33e-03	4.91e-03	0	1.80e-03	0.01	100
3	100	4.17e-03	0	1.39e-03	4.71e-03	0	1.87e-03	0.03	94
3	200	1.38e-03	2.21e-04	5.03e-04	1.99e-03	2.25e-04	1.04e-03	0.19	241
5	100	6.25e-03	1.44e-03	3.02e-03	1.67e-02	2.16e-03	8.45e-03	0.08	138
5	200	7.11e-03	1.15e-03	2.17e-03	7.99e-03	1.38e-03	3.52e-03	0.54	251
10	200	8.44e-03	2.97e-03	5.19e-03	1.74e-02	2.97e-03	9.23e-03	1.16	241
10	500	3.03e-03	8.59e-04	1.72e-03	6.01e-03	1.53e-03	2.72e-03	17.66	625
20	500	1.18e-02	5.98e-03	8.59e-03	1.18e-02	5.98e-03	9.32e-03	52.84	529
20	1000	3.52e-03	1.52e-03	2.64e-03	4.97e-03	2.43e-03	3.39e-03	213.46	1000

Tabella 4.30: Risultati dell'algoritmo MS_BPT con politica di gestione dei nodi basata su "deque" eseguito su istanze di tipo **alm-1**. Soluzione iniziale generata mediante l'euristica "Multifit".

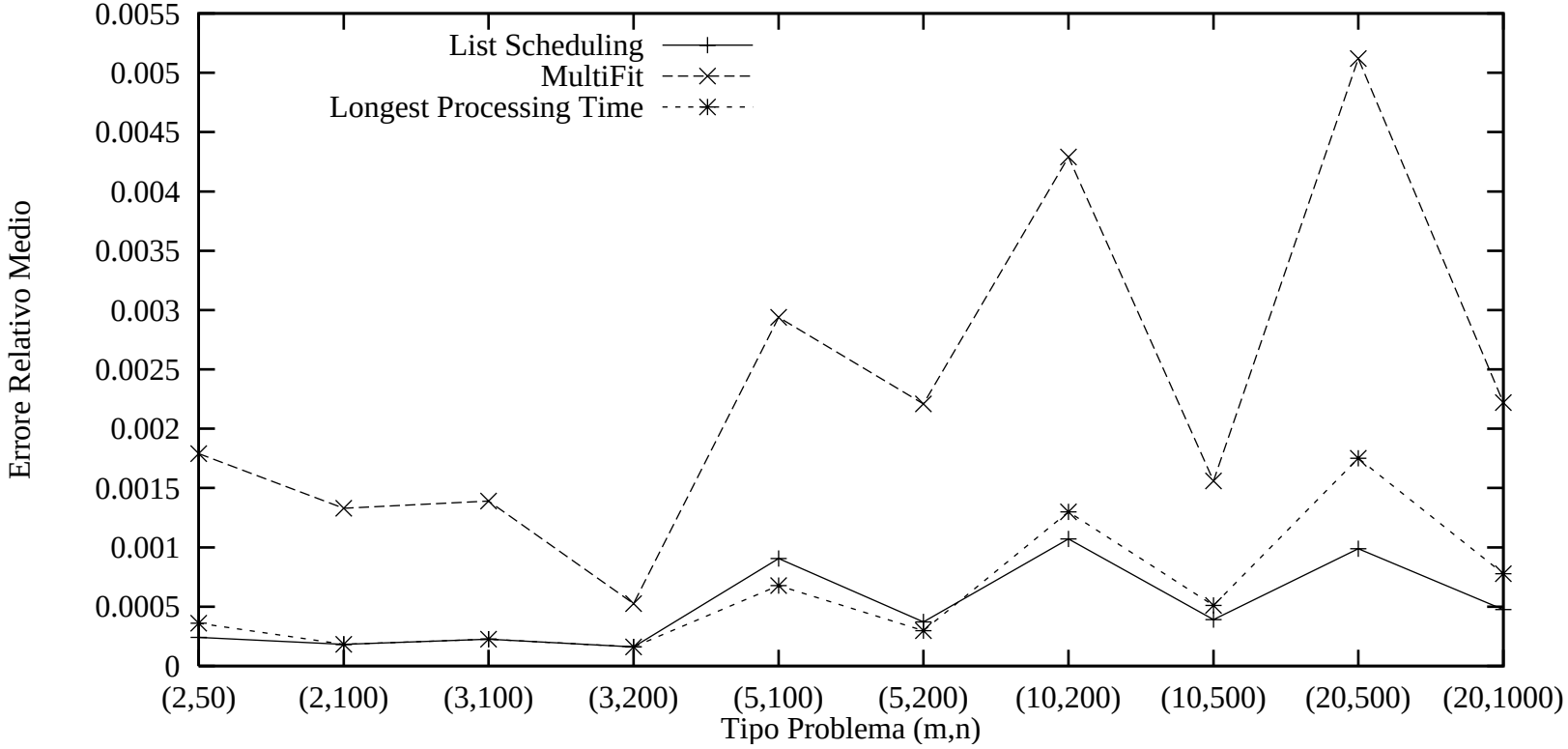

Figura 4.8: Confronto dei risultati ottenuti mediante l'algoritmo MS_SPT con politica "FIFO" eseguito a partire da soluzioni diverse su istanze di tipo **alm-1**

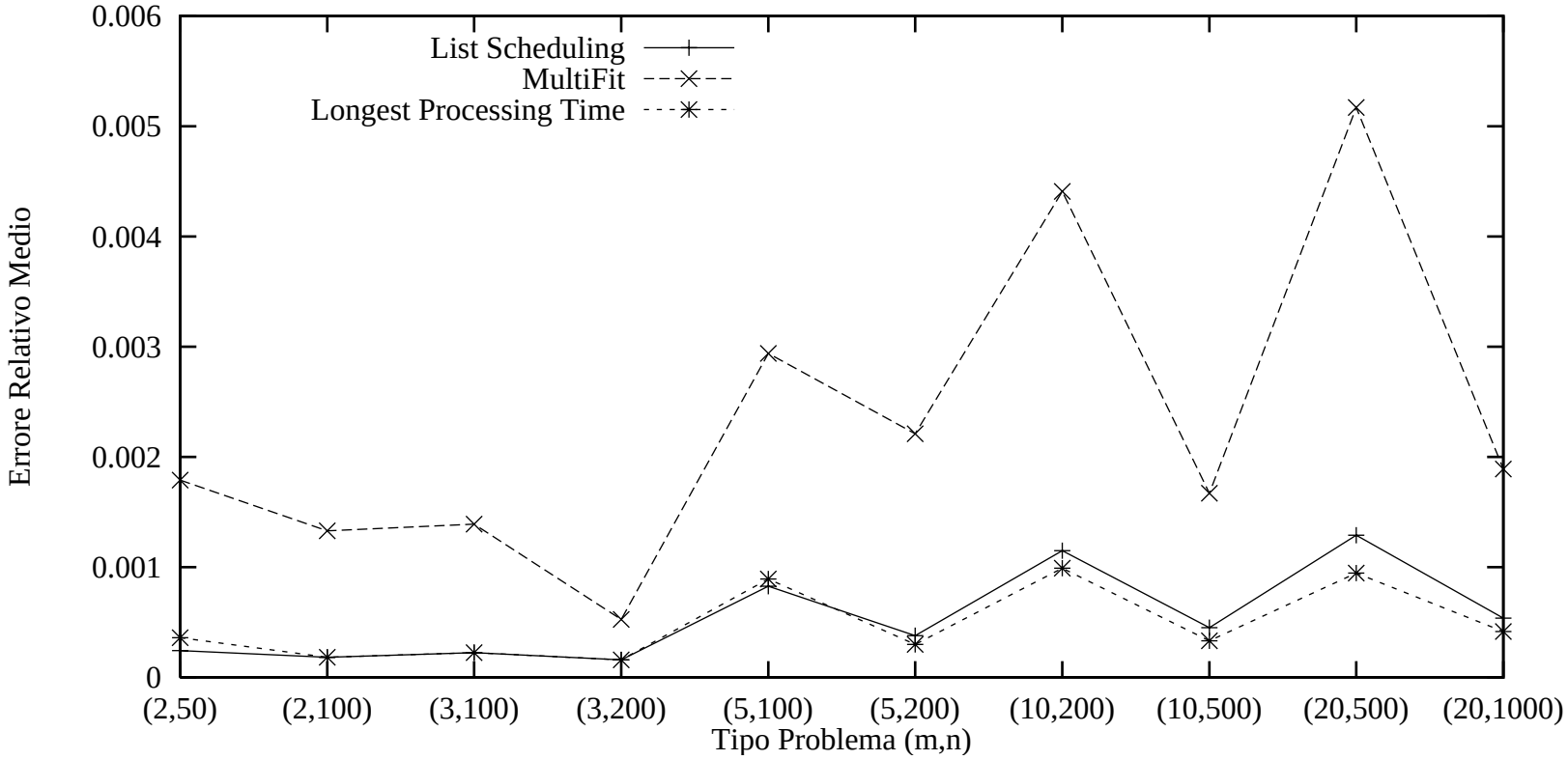

Figura 4.9: Confronto dei risultati ottenuti mediante l'algoritmo MS_SPT con politica "DEQUE" eseguito a partire da soluzioni diverse su istanze di tipo **alm-1**

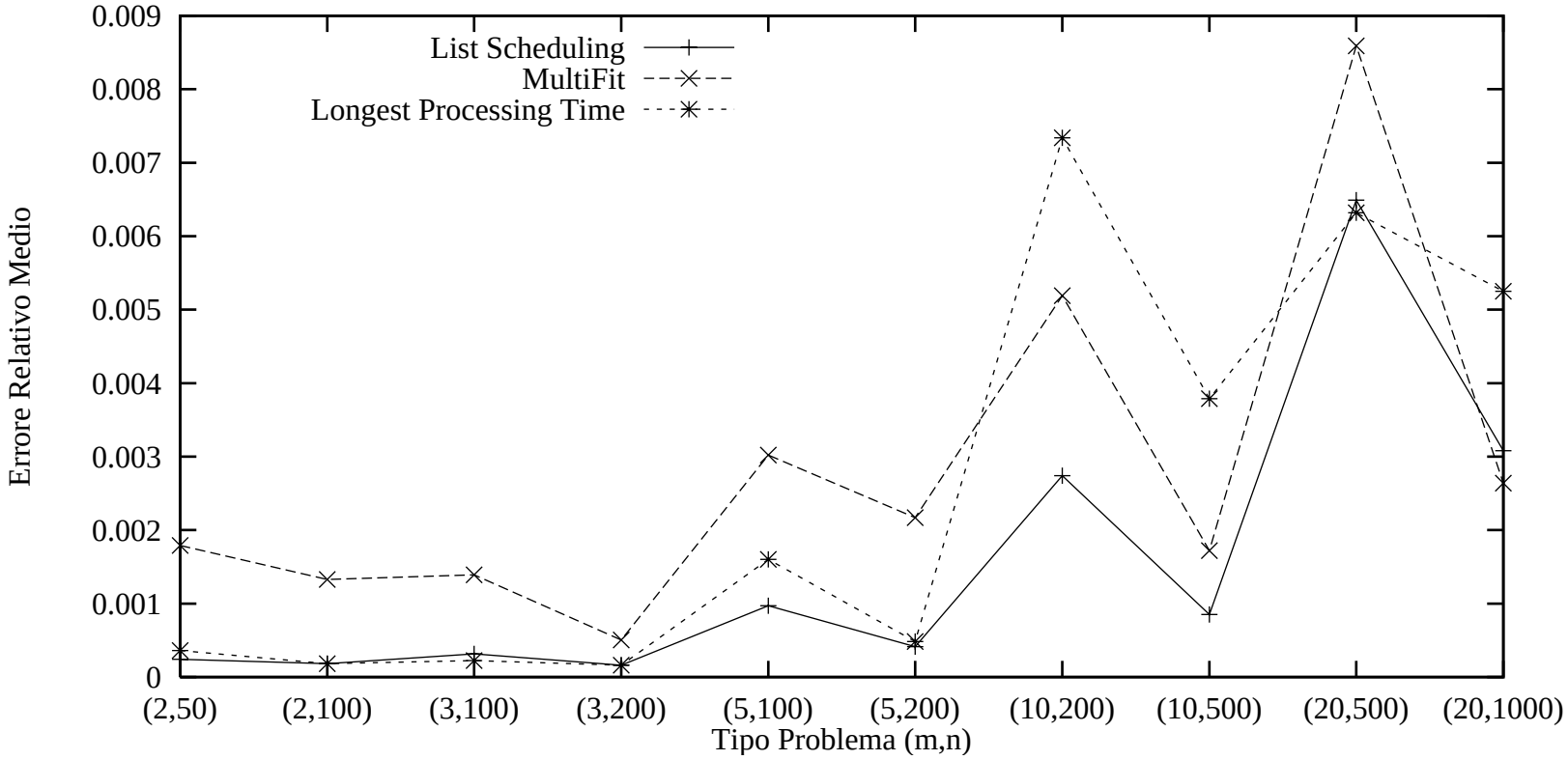

Figura 4.10: Confronto dei risultati ottenuti mediante l'algoritmo MS_BPT con politica "FIFO" eseguito a partire da soluzioni diverse su istanze di tipo **alm-1**

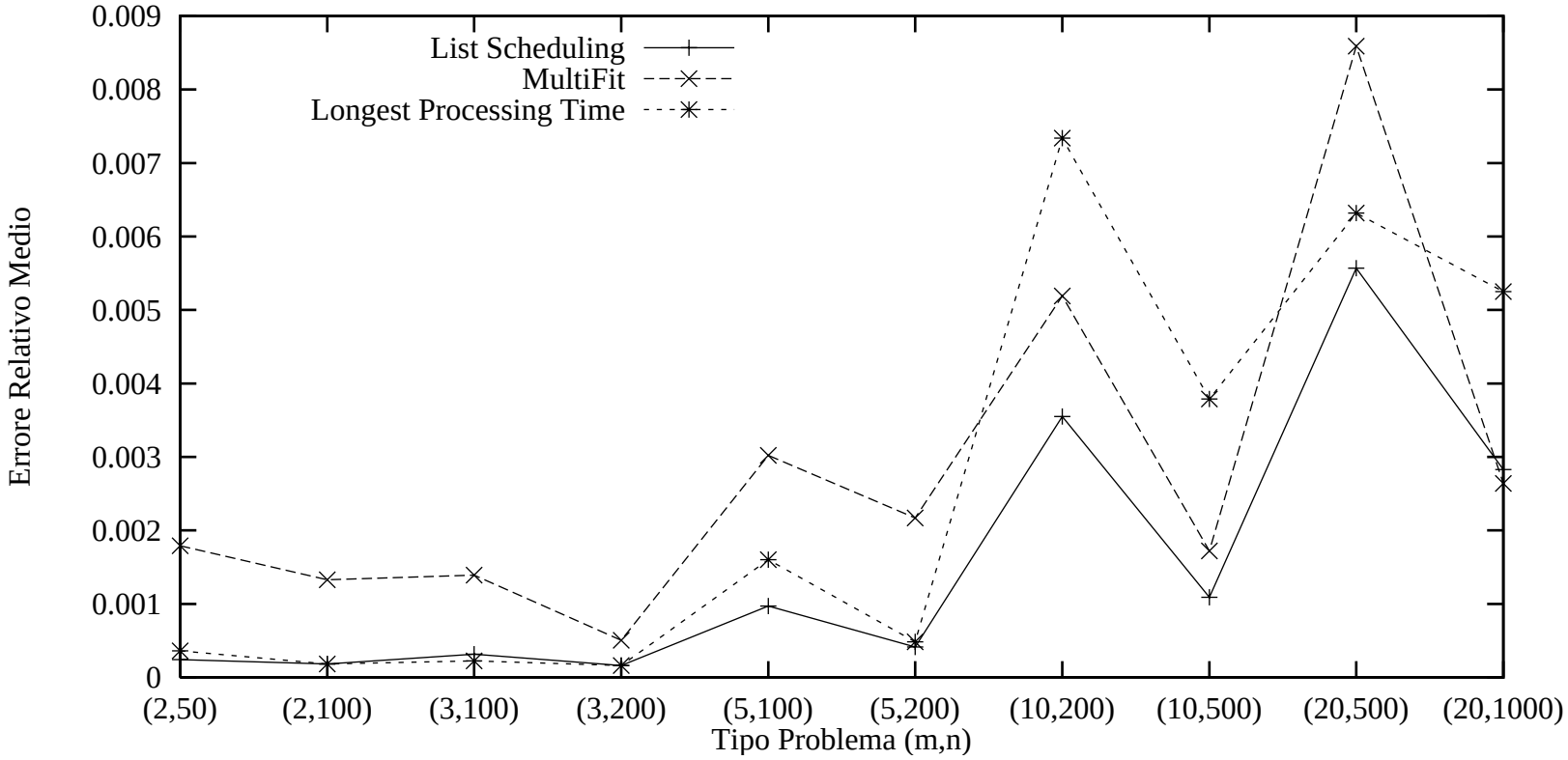

Figura 4.11: Confronto dei risultati ottenuti mediante l'algoritmo MS_BPT con politica "DEQUE" eseguito a partire da soluzioni diverse su istanze di tipo **alm-1**

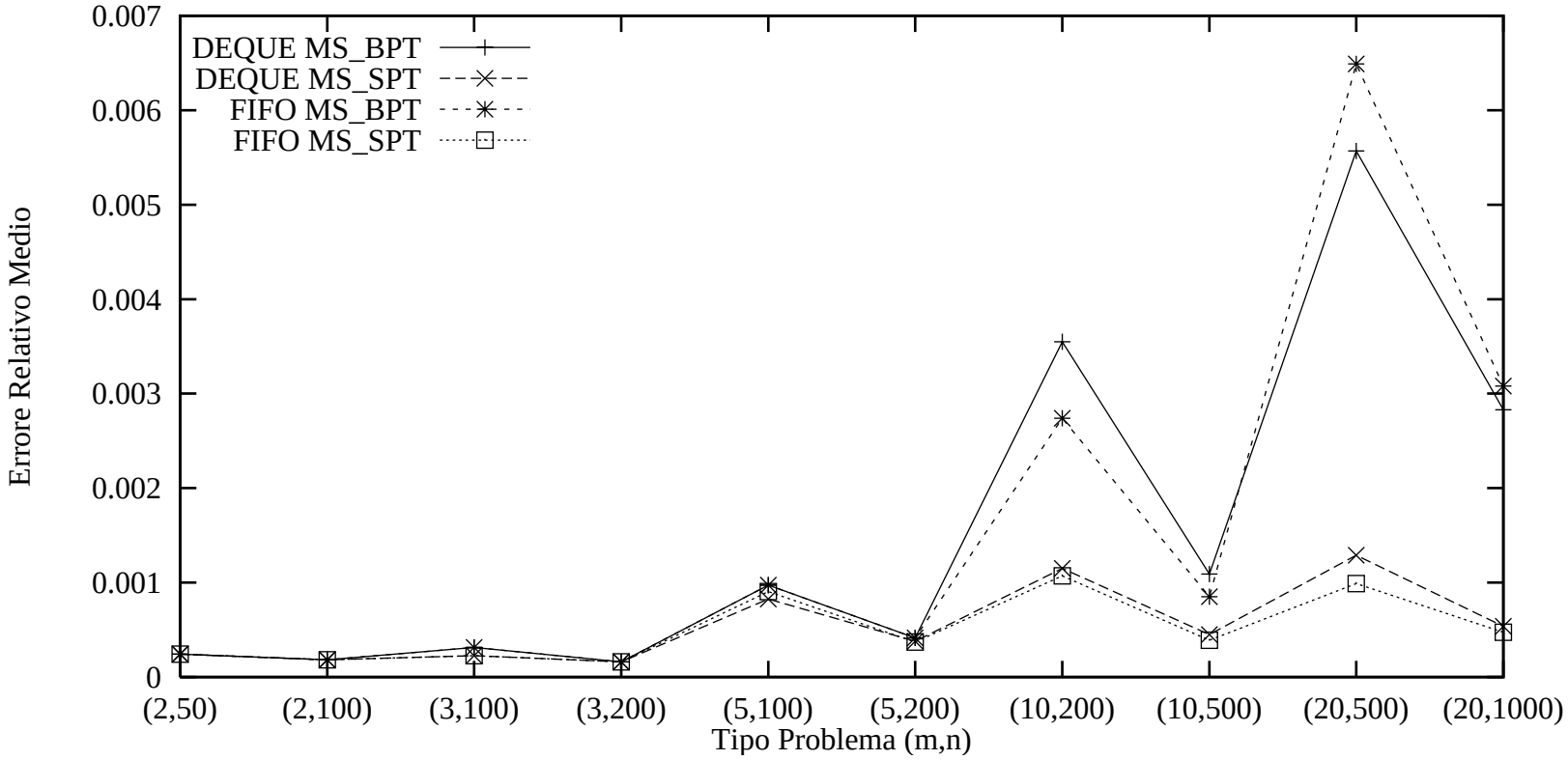

Figura 4.12: Confronto dei risultati ottenuti mediante algoritmi diversi eseguiti su istanze di tipo **alm-1** a partire da soluzioni iniziali ottenute con l'euristica "List Scheduling"

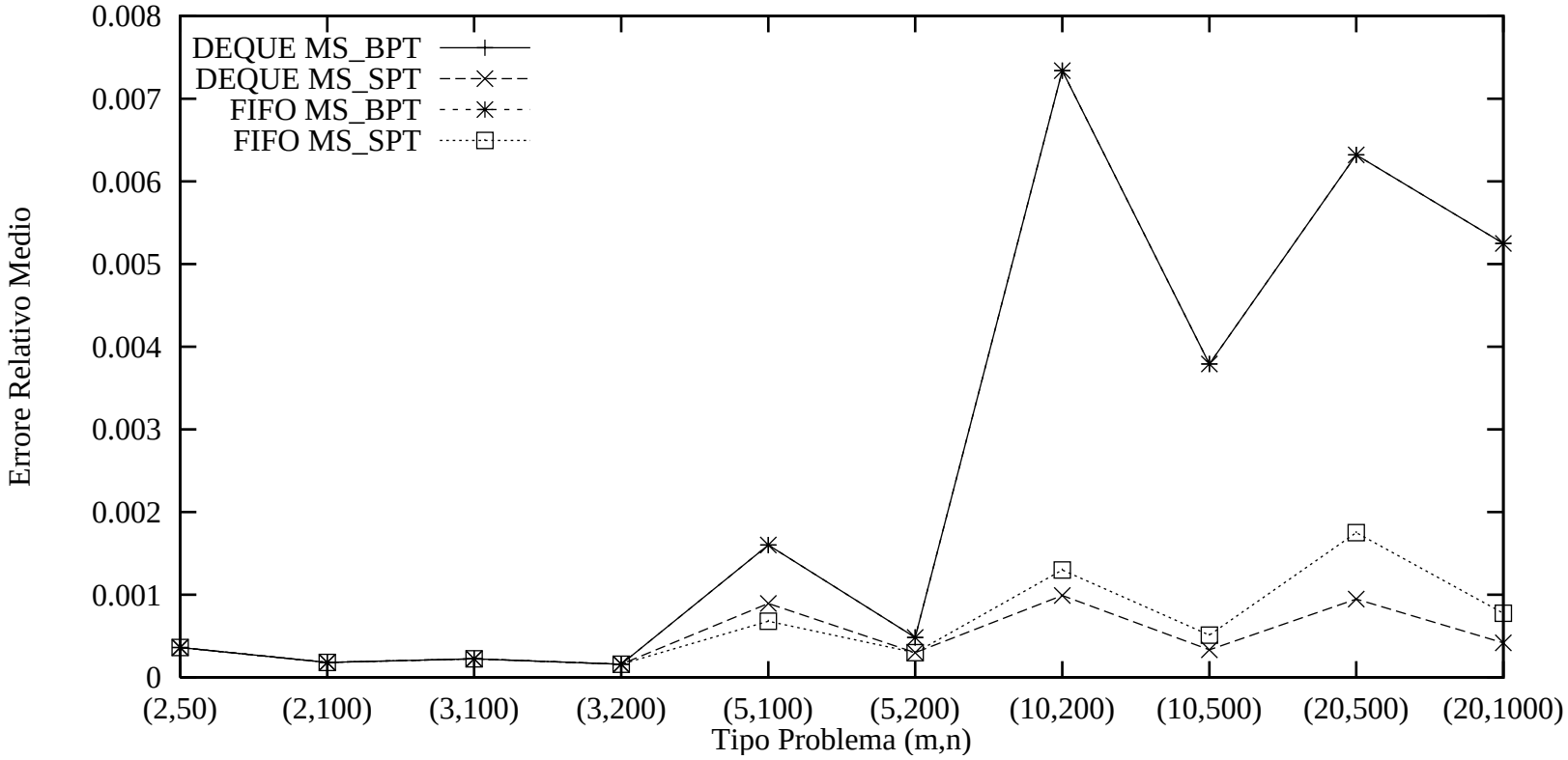

Figura 4.13: Confronto dei risultati ottenuti mediante algoritmi diversi eseguiti su istanze di tipo **alm-1** a partire da soluzioni iniziali ottenute con l'euristica "Longest Processing Time"

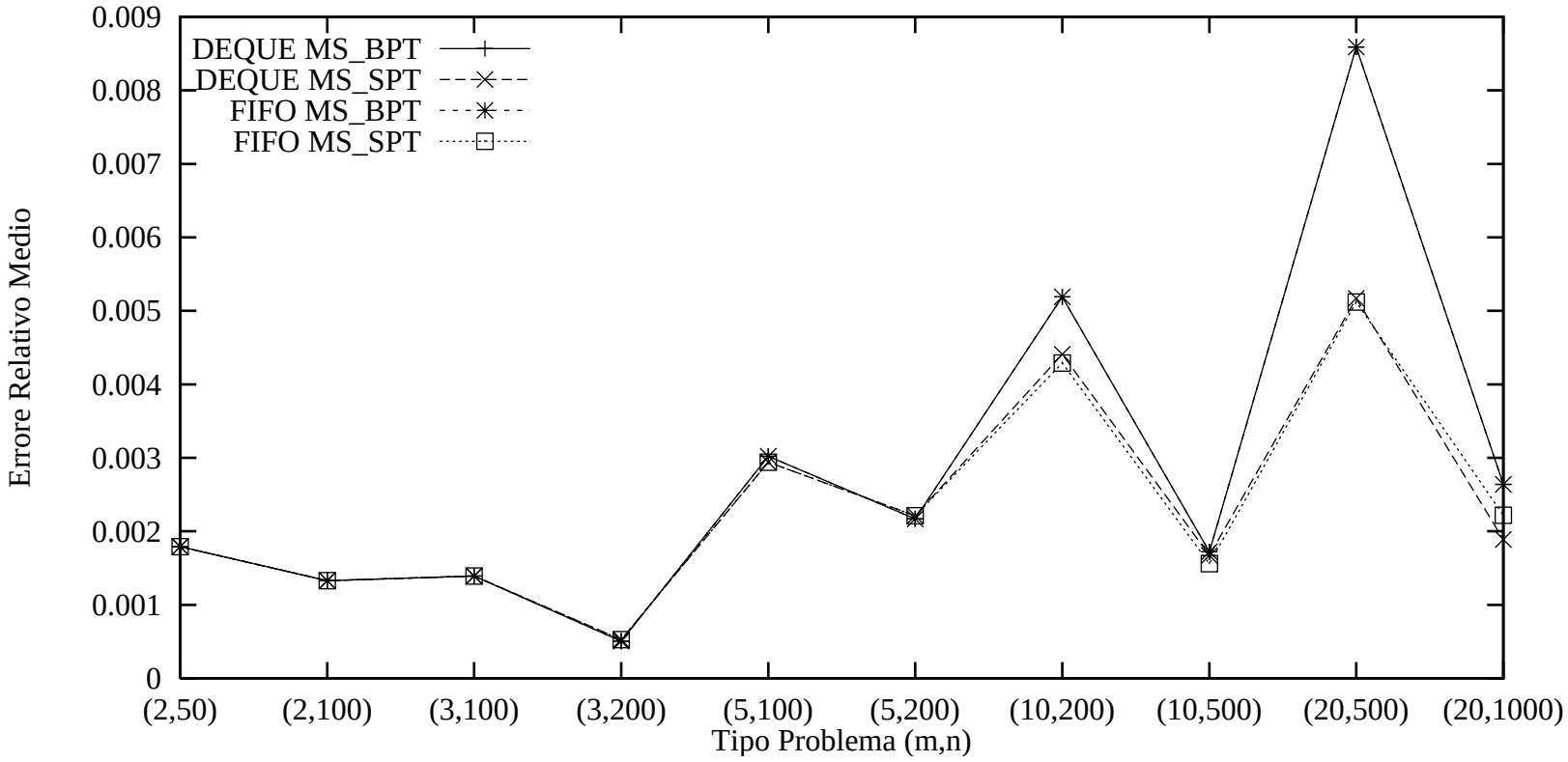

Figura 4.14: Confronto dei risultati ottenuti mediante algoritmi diversi eseguiti su istanze di tipo **alm-1** a partire da soluzioni iniziali ottenute con l'euristica "Multifit"

4.7 Risultati relativi alle istanze di tipo alm-2

In questo paragrafo vengono presentati i risultati ottenuti mediante le varianti degli algoritmi MS_SPT e MS_BPT applicati alle istanze di tipo **alm-2**. I risultati sono riportati nelle Tabelle da 4.31 a 4.42, e visualizzati mediante i grafici nelle Figure da 4.15 a 4.21.

Anche per queste istanze i risultati migliori sono stati ottenuti partendo da soluzioni iniziali costruite con le euristiche "List Scheduling" e "Longest Processing Time". Nel caso delle due versioni dell'euristica MS_BPT, si nota una differenza notevole tra le soluzioni ottenute a partire da una soluzione iniziale ricavata con l'euristica "List Scheduling" e quelle ottenute a partire da una soluzione iniziale ricavata con l'euristica "Multifit"o con l'euristica "Longest Processing Time", soprattutto nel caso delle istanze di dimensioni maggiori.

Quando la soluzione iniziale è ricavata mediante l'euristica "List Scheduling", la versione dell'algoritmo che ottiene, in tutti i casi, la soluzione migliore è MS_SPT con politica "FIFO". Il tempo impiegato dalle due euristiche con politica "FIFO" è, in tutti i casi, notevolmente inferiore rispetto a quello richiesto dalle euristiche con politica di gestione dei nodi basata su "deque".

Nel caso in cui la soluzione iniziale sia ricavata mediante l'euristica "Longest Processing Time", le soluzioni ottenute dalle due versioni degli algoritmi MS_SPT sono paragonabili. Le due versioni dell'algoritmo MS_BPT ottengono invece proprio le stesse soluzioni. Comunque, analogamente a quanto avviene per le istanze **alm-1**, sulle istanze con numero di macchine e numero di lavori pari a $(2, 50)$, $(2, 100)$, $(3, 100)$, $(5, 100)$ e $(5, 200)$, le soluzioni ottenute dai diversi algoritmi sono paragonabili, mentre sulle istanze con numero di macchine e numero di lavori pari a $(10, 200)$, $(10, 500)$, $(20, 500)$ e $(20, 1000)$, le due varianti dell'algoritmo MS_BPT ottengono risultati peggiori di quelli ottenuti con gli algoritmi MS_SPT. Sulle istanze con numero di macchine e di lavori pari a $(10, 200)$, $(10, 500)$, $(20, 500)$ e $(20, 1000)$ la soluzione migliore

è ottenuta con l'algoritmo MS_SPT con politica di gestione dei nodi basata su "deque". Sulle istanze con numero di macchine e di lavori pari a $(2, 50)$, $(2, 100)$, $(3, 100)$, $(3, 200)$, $(5, 100)$ e $(5, 200)$ la soluzione migliore è quella ottenuta dall'algoritmo MS_SPT con politica "FIFO".

Nel caso in cui la soluzione iniziale sia ricavata mediante l'euristica "Multifit", le soluzioni ottenute dalle varie versioni degli algoritmi sono molto vicine in quasi tutti i casi. Soltanto le due versioni dell'algoritmo MS_BPT si comportano diversamente sulle istanze con numero di macchine maggiore o uguale a 10 e numero di lavori maggiore o uguale a 200. Il tempo impiegato dalle euristiche con politica "FIFO" è, in tutti casi, minore di quello delle euristiche con politica di gestione dei nodi basata su "deque".

Per queste istanze, da considerare più "difficili" rispetto alle precedenti, gli algoritmi derivati da MS_SPT mostrano un comportamento molto buono, riuscendo ad ottenere, nella maggioranza delle istanze testate, un errore relativo medio inferiore a 10^{-5}, partendo da soluzioni costruite mediante euristiche "List Scheduling" e "Longest Processing Time". Anche per queste istanze le soluzioni iniziali sono sempre notevolmente migliorate. L'euristica di costruzione "Multifit" mostra invece lo stesso comportamento "patologico" manifestato per le istanze **alm-u** e **alm-1**.

		$\frac{C_{max}-C}{C}$			$\frac{C_{SI}-C}{C}$			Media	
m	n	max	min	media	max	min	media	sec	iter.
2	50	6.14e-05	0	2.35e-05	2.39e-02	5.52e-04	1.41e-02	0.00	40
2	100	3.04e-05	0	1.77e-05	1.32e-02	7.39e-04	7.29e-03	0.03	82
3	100	4.83e-05	0	3.62e-05	2.76e-02	1.23e-03	1.43e-02	0.09	115
3	200	2.49e-05	0	1.81e-05	1.07e-02	4.03e-03	6.23e-03	0.46	184
5	100	5.33e-03	0	5.78e-04	3.95e-02	4.48e-03	2.48e-02	0.15	137
5	200	4.16e-05	0	3.08e-05	2.52e-02	5.43e-03	1.24e-02	0.78	213
10	200	5.32e-04	7.44e-05	1.66e-04	4.61e-02	5.01e-03	2.59e-02	2.32	317
10	500	6.05e-05	2.69e-05	3.57e-05	1.88e-02	5.25e-03	1.30e-02	20.02	661
20	500	1.78e-04	5.86e-05	1.19e-04	3.96e-02	1.49e-02	2.46e-02	41.32	709
20	1000	1.20e-04	2.99e-05	6.47e-05	1.46e-02	6.04e-03	1.07e-02	228.26	1000

Tabella 4.31: Risultati dell'algoritmo MS_SPT con politica "FIFO" eseguito su istanze di tipo **alm-2**. Soluzione iniziale generata mediante l'euristica "List Scheduling".

		$\frac{C_{max}-C}{C}$			$\frac{C_{SI}-C}{C}$			Media	
m	n	max	min	media	max	min	media	sec	iter.
2	50	6.14e-05	0	2.35e-05	2.39e-02	5.52e-04	1.41e-02	0.01	40
2	100	3.04e-05	0	1.77e-05	1.32e-02	7.39e-04	7.29e-03	0.03	82
3	100	4.83e-05	0	3.62e-05	2.76e-02	1.23e-03	1.43e-02	0.09	112
3	200	2.49e-05	0	1.81e-05	1.07e-02	4.03e-03	6.23e-03	0.50	188
5	100	2.88e-04	0	8.12e-05	3.95e-02	4.48e-03	2.48e-02	0.20	157
5	200	4.16e-05	0	3.08e-05	2.52e-02	5.43e-03	1.24e-02	0.81	222
10	200	3.68e-04	7.47e-05	1.65e-04	4.61e-02	5.01e-03	2.59e-02	3.12	327
10	500	1.52e-04	2.86e-05	6.59e-05	1.88e-02	5.25e-03	1.30e-02	27.77	632
20	500	2.98e-04	5.95e-05	1.66e-04	3.96e-02	1.49e-02	2.46e-02	90.68	727
20	1000	1.21e-04	5.66e-05	7.96e-05	1.46e-02	6.04e-03	1.07e-02	445.60	1000

Tabella 4.32: Risultati dell'algoritmo MS_SPT con politica di gestione dei nodi basata su "deque" eseguito su istanze di tipo **alm-2**. Soluzione iniziale generata mediante l'euristica "List Scheduling".

		$\frac{C_{max}-\tilde{C}}{\tilde{C}}$			$\frac{C_{SI}-\tilde{C}}{\tilde{C}}$			Media	
m	n	max	min	media	max	min	media	sec	iter.
2	50	6.14e-05	0	2.35e-05	1.30e-02	6.84e-03	1.13e-02	0.00	77
2	100	3.04e-05	0	1.77e-05	1.13e-03	9.43e-05	5.29e-04	0.03	92
3	100	4.83e-05	0	3.62e-05	5.62e-03	2.25e-03	3.75e-03	0.08	122
3	200	2.49e-05	0	1.81e-05	2.14e-03	3.18e-04	1.32e-03	0.42	197
5	100	7.94e-05	0	4.51e-05	1.64e-02	8.07e-03	1.26e-02	0.09	177
5	200	4.16e-05	0	3.46e-05	8.44e-03	5.93e-03	7.52e-03	0.72	233
10	200	2.99e-04	7.54e-05	1.50e-04	1.19e-02	8.13e-03	1.03e-02	1.66	407
10	500	8.57e-05	2.78e-05	4.75e-05	7.57e-03	4.08e-03	5.41e-03	16.33	675
20	500	5.41e-04	5.97e-05	2.16e-04	8.31e-03	5.13e-03	6.62e-03	44.09	815
20	1000	1.98e-04	6.05e-05	1.14e-04	6.96e-03	5.10e-03	5.93e-03	285.36	1000

Tabella 4.33: Risultati dell'algoritmo MS_SPT con politica "FIFO" eseguito su istanze di tipo **alm-2**. Soluzione iniziale generata mediante l'euristica "Longest Processing Time".

		$\frac{C_{max}-C}{C}$			$\frac{C_{SI}-C}{C}$			Media	
m	n	max	min	media	max	min	media	sec	iter.
2	50	6.14e-05	0	2.35e-05	1.30e-02	6.84e-03	1.13e-02	0.01	77
2	100	3.04e-05	0	1.77e-05	1.13e-03	9.43e-05	5.29e-04	0.02	92
3	100	4.83e-05	0	3.62e-05	5.62e-03	2.25e-03	3.75e-03	0.08	122
3	200	2.49e-05	0	1.81e-05	2.14e-03	3.18e-04	1.32e-03	0.41	197
5	100	7.94e-05	0	4.49e-05	1.64e-02	8.07e-03	1.26e-02	0.11	158
5	200	4.16e-05	0	3.08e-05	8.44e-03	5.93e-03	7.52e-03	0.75	239
10	200	1.52e-04	7.35e-05	8.28e-05	1.19e-02	8.13e-03	1.03e-02	1.93	438
10	500	8.08e-05	2.86e-05	5.33e-05	7.57e-03	4.08e-03	5.41e-03	41.78	570
20	500	1.30e-04	5.69e-05	8.41e-05	8.31e-03	5.13e-03	6.62e-03	66.24	812
20	1000	6.13e-05	2.78e-05	4.75e-05	6.96e-03	5.10e-03	5.93e-03	496.35	1000

Tabella 4.34: Risultati dell'algoritmo MS_SPT con politica di gestione dei nodi basata su "deque" eseguito su istanze di tipo **alm-2**. Soluzione iniziale generata mediante l'euristica "Longest Processing Time".

		$\frac{C_{max}-C}{C}$			$\frac{C_{SI}-C}{C}$			Media	
m	n	max	min	media	max	min	media	sec	iter.
2	50	1.10e-02	0	2.29e-03	1.10e-02	1.24e-04	3.16e-03	0.00	64
2	100	5.08e-03	0	7.82e-04	5.08e-03	0	8.94e-04	0.01	115
3	100	2.02e-03	4.43e-05	6.23e-04	9.02e-03	1.12e-03	3.94e-03	0.06	259
3	200	5.95e-04	4.29e-05	2.68e-04	1.69e-03	2.87e-04	7.47e-04	0.25	548
5	100	7.18e-03	5.05e-04	2.73e-03	8.14e-03	2.46e-03	5.02e-03	0.11	200
5	200	3.81e-03	1.90e-04	1.27e-03	7.12e-03	3.80e-04	2.40e-03	0.71	490
10	200	1.06e-02	7.60e-04	2.96e-03	1.56e-02	3.57e-03	8.86e-03	1.76	637
10	500	2.68e-03	6.85e-04	1.53e-03	3.98e-03	1.51e-03	2.73e-03	13.19	847
20	500	5.23e-03	1.19e-03	2.86e-03	1.33e-02	4.33e-03	8.15e-03	29.90	825
20	1000	2.30e-03	4.90e-04	1.26e-03	4.07e-03	1.79e-03	3.22e-03	119.62	1000

Tabella 4.35: Risultati dell'algoritmo MS_SPT con politica "FIFO" eseguito su istanze di tipo **alm-2**. Soluzione iniziale generata mediante l'euristica "Multifit".

		$\frac{C_{max}-C}{C}$			$\frac{C_{SI}-C}{C}$			Media	
m	n	max	min	media	max	min	media	sec	iter.
2	50	1.10e-02	0	2.29e-03	1.10e-02	1.24e-04	3.16e-03	0.00	64
2	100	5.08e-03	0	7.82e-04	5.08e-03	0	8.94e-04	0.01	115
3	100	2.02e-03	4.43e-05	6.23e-04	9.02e-03	1.12e-03	3.94e-03	0.06	259
3	200	5.95e-04	4.29e-05	2.68e-04	1.69e-03	2.87e-04	7.47e-04	0.26	548
5	100	7.18e-03	5.05e-04	2.73e-03	8.14e-03	2.46e-03	5.02e-03	0.13	200
5	200	3.81e-03	1.90e-04	1.27e-03	7.12e-03	3.80e-04	2.40e-03	1.08	490
10	200	1.06e-02	2.23e-04	2.93e-03	1.56e-02	3.57e-03	8.86e-03	2.72	566
10	500	2.68e-03	4.72e-04	1.50e-03	3.98e-03	1.51e-03	2.73e-03	36.43	847
20	500	6.42e-03	1.72e-03	3.03e-03	1.33e-02	4.33e-03	8.15e-03	53.81	856
20	1000	2.30e-03	3.93e-04	1.28e-03	4.07e-03	1.79e-03	3.22e-03	424.71	1000

Tabella 4.36: Risultati dell'algoritmo MS_SPT con politica di gestione dei nodi basata su "deque" eseguito su istanze di tipo **alm-2**. Soluzione iniziale generata mediante l'euristica "Multifit".

		$\frac{C_{max}-\tilde{C}}{\tilde{C}}$			$\frac{C_{SI}-\tilde{C}}{\tilde{C}}$			Media	
m	n	max	min	media	max	min	media	sec	iter.
2	50	6.14e-05	0	2.35e-05	2.39e-02	5.52e-04	1.41e-02	0.01	40
2	100	3.04e-05	0	1.77e-05	1.32e-02	7.39e-04	7.29e-03	0.03	82
3	100	4.83e-05	0	3.62e-05	2.76e-02	1.23e-03	1.43e-02	0.08	126
3	200	2.49e-05	0	1.81e-05	1.07e-02	4.03e-03	6.23e-03	0.41	184
5	100	3.03e-04	0	1.33e-04	3.95e-02	4.48e-03	2.48e-02	0.15	148
5	200	3.42e-04	0	8.73e-05	2.52e-02	5.43e-03	1.24e-02	0.60	205
10	200	2.45e-03	1.48e-04	5.92e-04	4.61e-02	5.01e-03	2.59e-02	1.78	348
10	500	1.15e-03	5.39e-05	2.81e-04	1.88e-02	5.25e-03	1.30e-02	15.75	533
20	500	1.23e-02	2.41e-04	2.22e-03	3.96e-02	1.49e-02	2.46e-02	26.40	564
20	1000	1.49e-03	1.50e-04	4.93e-04	1.46e-02	6.04e-03	1.07e-02	160.28	1000

Tabella 4.37: Risultati dell'algoritmo MS_BPT con politica "FIFO" eseguito su istanze di tipo **alm-2**. Soluzione iniziale generata mediante l'euristica "List Scheduling".

		$\frac{C_{max}-C}{C}$			$\frac{C_{SI}-C}{C}$			Media	
m	n	max	min	media	max	min	media	sec	iter.
2	50	6.14e-05	0	2.35e-05	2.39e-02	5.52e-04	1.41e-02	0.01	40
2	100	3.04e-05	0	1.77e-05	1.32e-02	7.39e-04	7.29e-03	0.03	82
3	100	4.83e-05	0	3.62e-05	2.76e-02	1.23e-03	1.43e-02	0.08	126
3	200	2.49e-05	0	1.81e-05	1.07e-02	4.03e-03	6.23e-03	0.41	183
5	100	5.92e-04	7.40e-05	1.94e-04	3.95e-02	4.48e-03	2.48e-02	0.17	159
5	200	3.04e-04	0	8.37e-05	2.52e-02	5.43e-03	1.24e-02	0.70	228
10	200	2.13e-03	1.51e-04	8.84e-04	4.61e-02	5.01e-03	2.59e-02	1.83	305
10	500	7.79e-04	3.02e-05	2.09e-04	1.88e-02	5.25e-03	1.30e-02	15.75	541
20	500	1.10e-02	1.80e-04	2.35e-03	3.96e-02	1.49e-02	2.46e-02	33.77	586
20	1000	2.73e-03	1.51e-04	8.20e-04	1.46e-02	6.04e-03	1.07e-02	216.37	1000

Tabella 4.38: Risultati dell'algoritmo MS_BPT con politica di gestione dei nodi basata su "deque" eseguito su istanze di tipo **alm-2**. Soluzione iniziale generata mediante l'euristica "List Scheduling".

		$\frac{C_{max}-\bar{C}}{\bar{C}}$			$\frac{C_{SI}-\bar{C}}{\bar{C}}$			Media	
m	n	max	min	media	max	min	media	sec	iter.
2	50	6.14e-05	0	2.35e-05	1.30e-02	6.84e-03	1.13e-02	0.01	77
2	100	3.04e-05	0	1.77e-05	1.13e-03	9.43e-05	5.29e-04	0.03	92
3	100	4.83e-05	0	3.62e-05	5.62e-03	2.25e-03	3.75e-03	0.07	128
3	200	2.49e-05	0	1.81e-05	2.14e-03	3.18e-04	1.32e-03	0.33	187
5	100	8.07e-03	0	9.89e-04	1.64e-02	8.07e-03	1.26e-02	0.11	134
5	200	1.99e-04	3.58e-05	7.73e-05	8.44e-03	5.93e-03	7.52e-03	0.69	250
10	200	1.16e-02	2.21e-04	2.61e-03	1.19e-02	8.13e-03	1.03e-02	1.61	245
10	500	5.91e-03	2.78e-05	1.74e-03	7.57e-03	4.08e-03	5.41e-03	23.29	526
20	500	7.70e-03	3.80e-03	5.65e-03	8.31e-03	5.13e-03	6.62e-03	48.63	511
20	1000	6.39e-03	1.51e-04	4.97e-03	6.96e-03	5.10e-03	5.93e-03	431.57	1000

Tabella 4.39: Risultati dell'algoritmo MS_BPT con politica "FIFO" eseguito su istanze di tipo **alm-2**. Soluzione iniziale generata mediante l'euristica "Longest Processing Time".

		$\frac{C_{max}-C}{C}$			$\frac{C_{SI}-C}{C}$			Media	
m	n	max	min	media	max	min	media	sec	iter.
2	50	6.14e-05	0	2.35e-05	1.30e-02	6.84e-03	1.13e-02	0.00	77
2	100	3.04e-05	0	1.77e-05	1.13e-03	9.43e-05	5.29e-04	0.02	92
3	100	4.83e-05	0	3.62e-05	5.62e-03	2.25e-03	3.75e-03	0.07	128
3	200	2.49e-05	0	1.81e-05	2.14e-03	3.18e-04	1.32e-03	0.33	187
5	100	8.07e-03	0	9.89e-04	1.64e-02	8.07e-03	1.26e-02	0.12	134
5	200	1.99e-04	3.58e-05	7.73e-05	8.44e-03	5.93e-03	7.52e-03	0.71	250
10	200	1.16e-02	2.21e-04	2.61e-03	1.19e-02	8.13e-03	1.03e-02	2.68	245
10	500	5.91e-03	2.78e-05	1.74e-03	7.57e-03	4.08e-03	5.41e-03	50.48	526
20	500	7.70e-03	3.80e-03	5.65e-03	8.31e-03	5.13e-03	6.62e-03	131.83	511
20	1000	6.39e-03	1.51e-04	4.97e-03	6.96e-03	5.10e-03	5.93e-03	1682.82	1000

Tabella 4.40: Risultati dell'algoritmo MS_BPT con politica di gestione dei nodi basata su "deque" eseguito su istanze di tipo **alm-2**. Soluzione iniziale generata mediante l'euristica "Longest Processing Time".

		$\frac{C_{max}-C}{C}$			$\frac{C_{SI}-C}{C}$			Media	
m	n	max	min	media	max	min	media	sec	iter.
2	50	1.10e-02	0	2.29e-03	1.10e-02	1.24e-04	3.16e-03	0.00	64
2	100	5.08e-03	0	7.82e-04	5.08e-03	0	8.94e-04	0.01	115
3	100	2.02e-03	4.43e-05	6.23e-04	9.02e-03	1.12e-03	3.94e-03	0.06	230
3	200	5.95e-04	4.29e-05	2.68e-04	1.69e-03	2.87e-04	7.47e-04	0.27	548
5	100	7.18e-03	5.77e-04	2.85e-03	8.14e-03	2.46e-03	5.02e-03	0.10	175
5	200	3.81e-03	1.90e-04	1.27e-03	7.12e-03	3.80e-04	2.40e-03	0.68	459
10	200	1.06e-02	5.23e-04	2.63e-03	1.56e-02	3.57e-03	8.86e-03	1.85	735
10	500	2.68e-03	5.13e-04	1.58e-03	3.98e-03	1.51e-03	2.73e-03	14.34	898
20	500	9.69e-03	1.54e-03	5.86e-03	1.33e-02	4.33e-03	8.15e-03	40.58	690
20	1000	4.07e-03	1.56e-03	2.59e-03	4.07e-03	1.79e-03	3.22e-03	238.50	1000

Tabella 4.41: Risultati dell'algoritmo MS_BPT con politica "FIFO" eseguito su istanze di tipo **alm-2**. Soluzione iniziale generata mediante l'euristica "Multifit".

		$\frac{C_{max}-C}{C}$			$\frac{C_{SI}-C}{C}$			Media	
m	n	max	min	media	max	min	media	sec	iter.
2	50	1.10e-02	0	2.29e-03	1.10e-02	1.24e-04	3.16e-03	0.00	64
2	100	5.08e-03	0	7.82e-04	5.08e-03	0	8.94e-04	0.01	115
3	100	2.02e-03	4.43e-05	6.23e-04	9.02e-03	1.12e-03	3.94e-03	0.06	230
3	200	5.95e-04	4.29e-05	2.68e-04	1.69e-03	2.87e-04	7.47e-04	0.27	548
5	100	7.18e-03	5.77e-04	2.85e-03	8.14e-03	2.46e-03	5.02e-03	0.10	175
5	200	3.81e-03	1.90e-04	1.27e-03	7.12e-03	3.80e-04	2.40e-03	0.80	459
10	200	1.06e-02	5.23e-04	2.63e-03	1.56e-02	3.57e-03	8.86e-03	2.20	735
10	500	2.68e-03	5.13e-04	1.58e-03	3.98e-03	1.51e-03	2.73e-03	22.28	898
20	500	9.69e-03	1.54e-03	5.86e-03	1.33e-02	4.33e-03	8.15e-03	89.24	690
20	1000	4.07e-03	1.56e-03	2.59e-03	4.07e-03	1.79e-03	3.22e-03	540.40	1000

Tabella 4.42: Risultati dell'algoritmo MS_BPT con politica di gestione dei nodi basata su "deque" eseguito su istanze di tipo **alm-2**. Soluzione iniziale generata mediante l'euristica "Multifit".

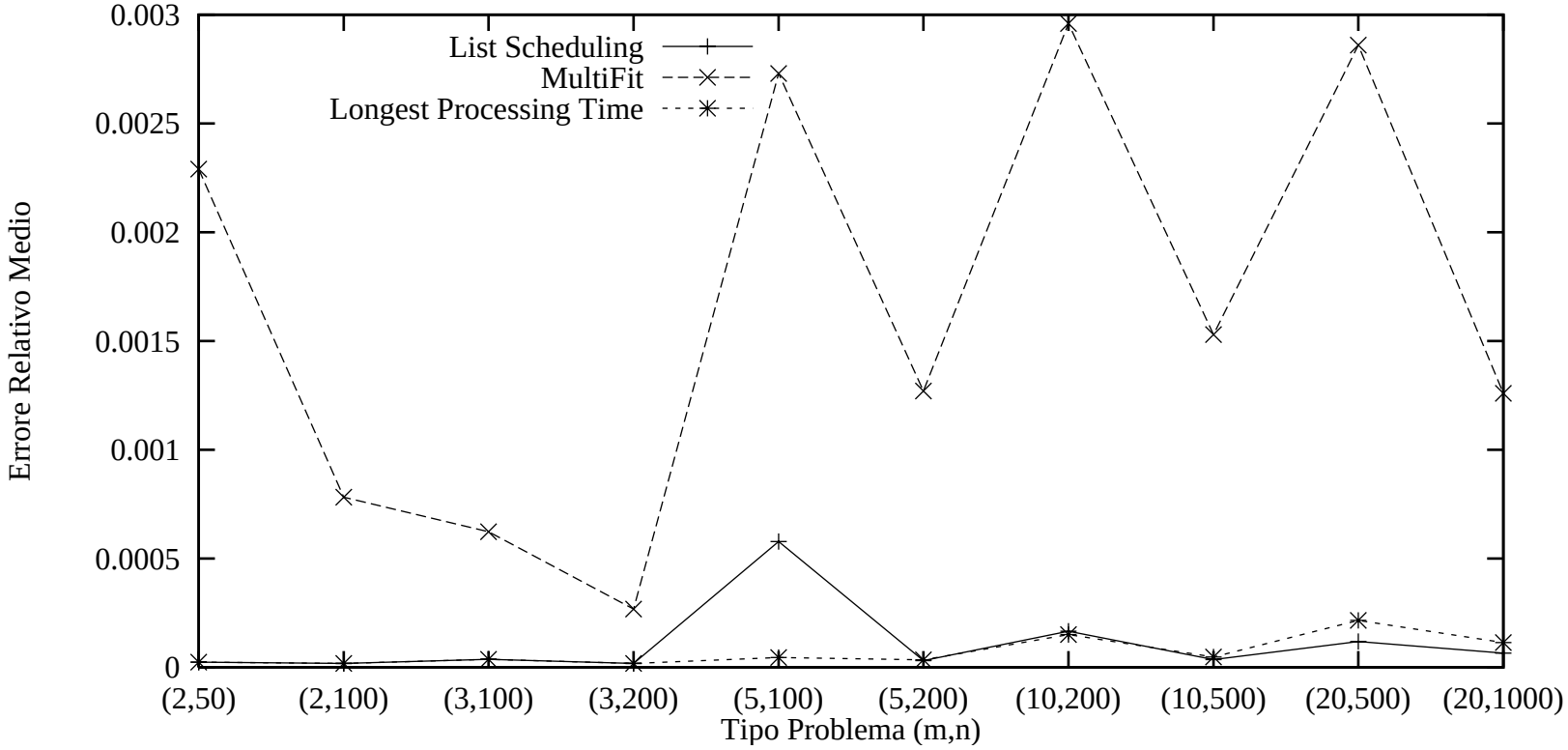

Figura 4.15: Confronto dei risultati ottenuti mediante l'algoritmo MS_SPT con politica "FIFO" eseguito a partire da soluzioni diverse su istanze di tipo **alm-2**

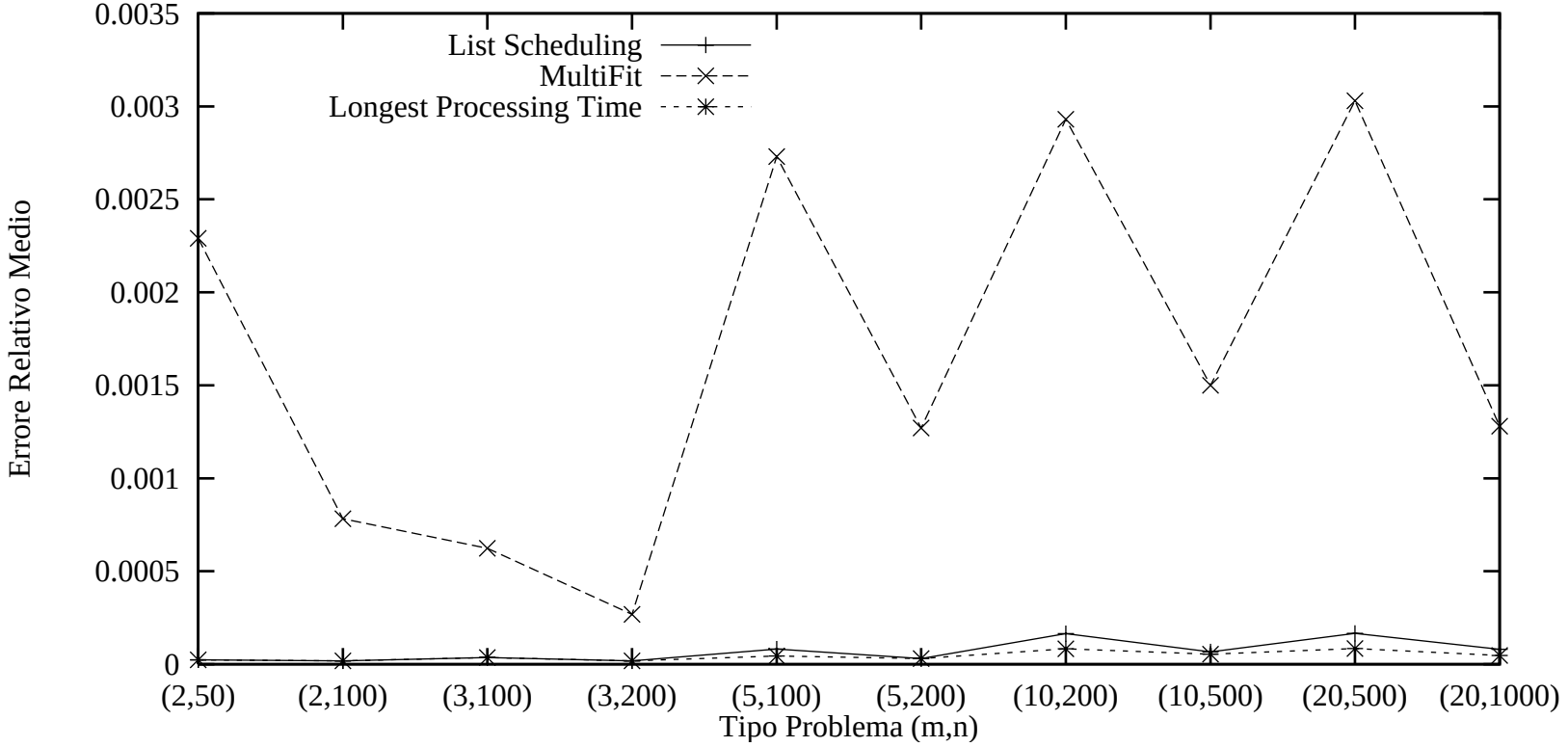

Figura 4.16: Confronto dei risultati ottenuti mediante l'algoritmo MS_SPT con politica "DEQUE" eseguito a partire da soluzioni diverse su istanze di tipo **alm-2**

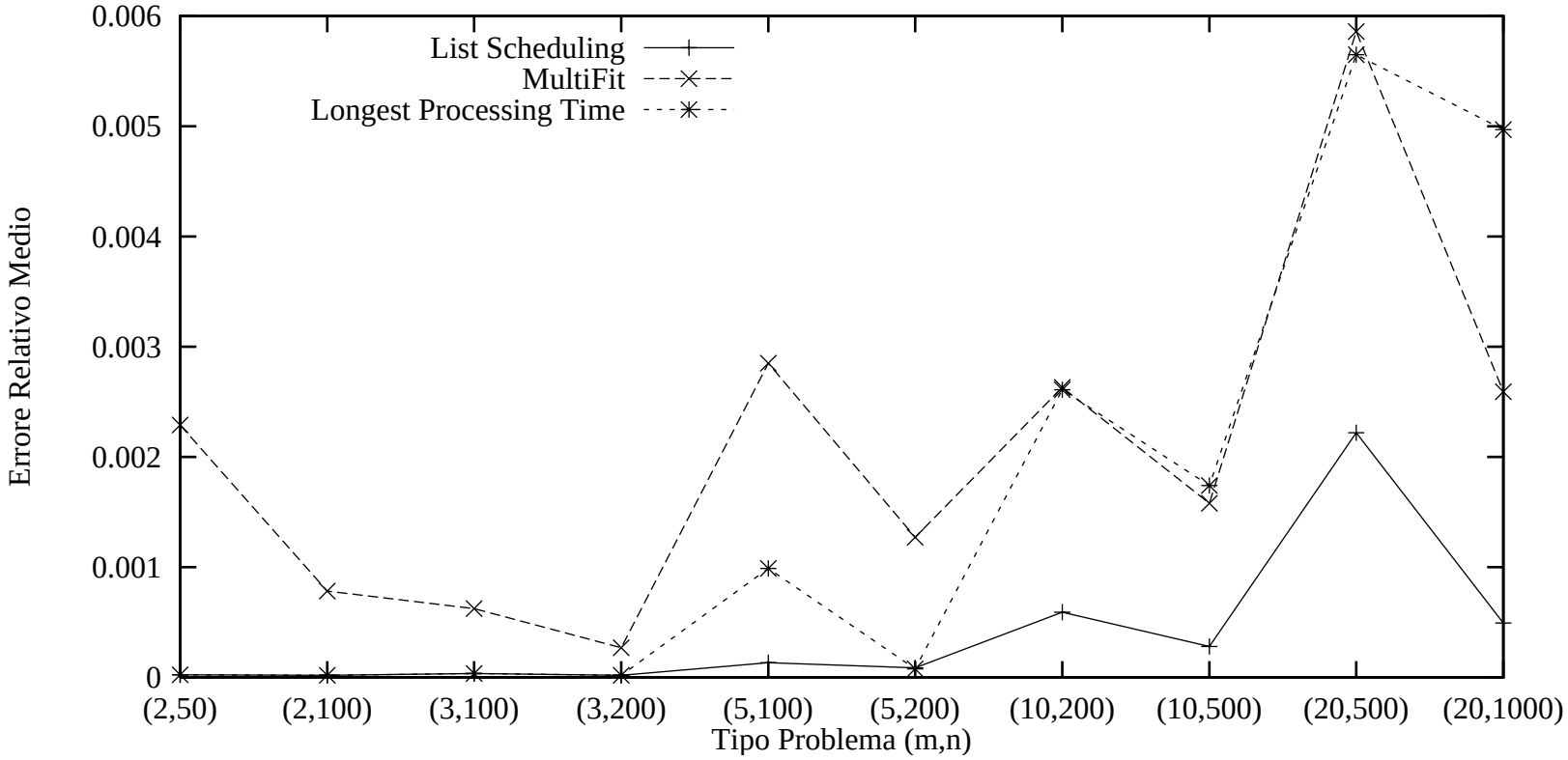

Figura 4.17: Confronto dei risultati ottenuti mediante l'algoritmo MS_BPT con politica "FIFO" eseguito a partire da soluzioni diverse su istanze di tipo **alm-2**

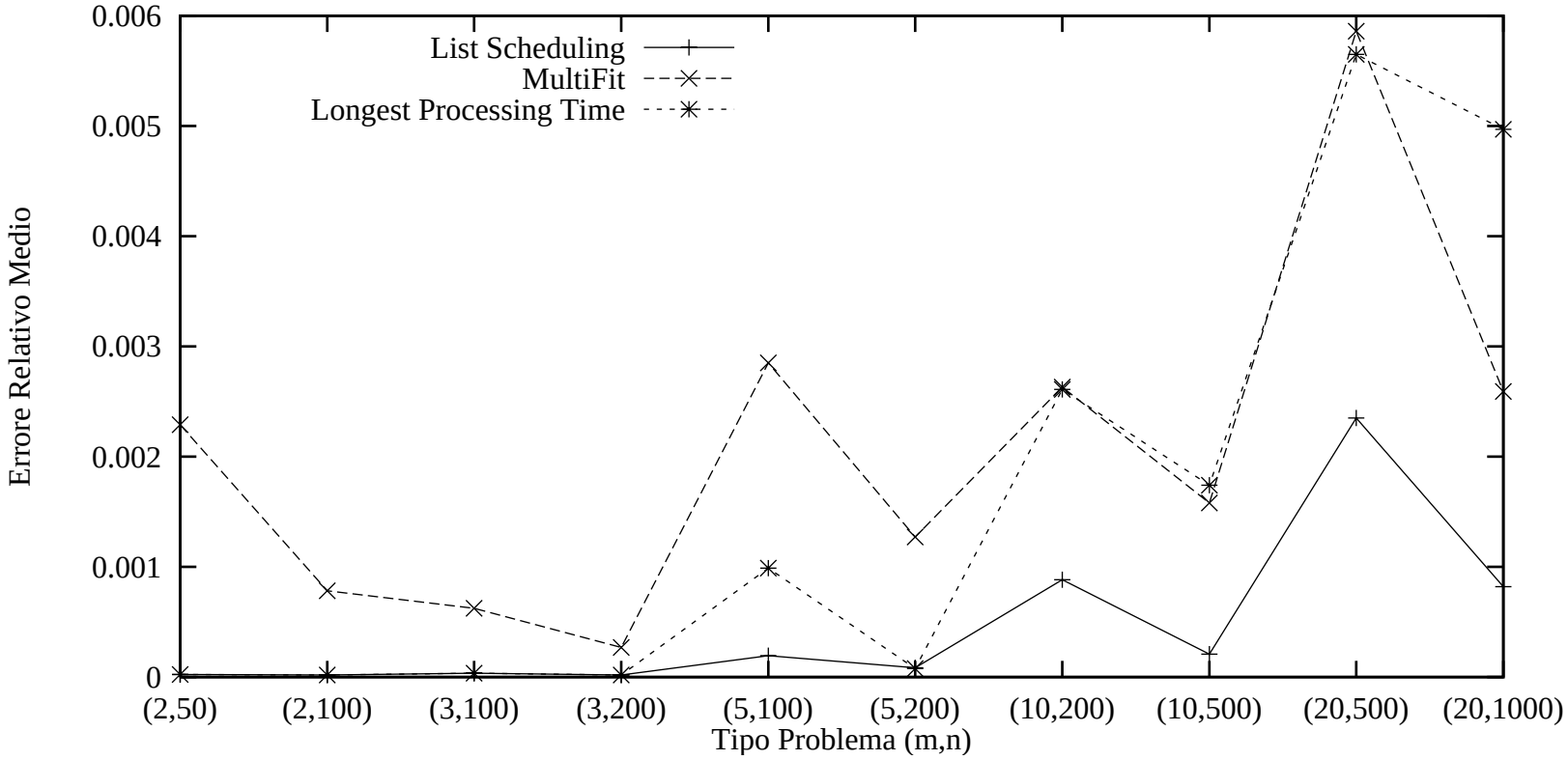

Figura 4.18: Confronto dei risultati ottenuti mediante l'algoritmo MS_BPT con politica "DEQUE" eseguito a partire da soluzioni diverse su istanze di tipo **alm-2**

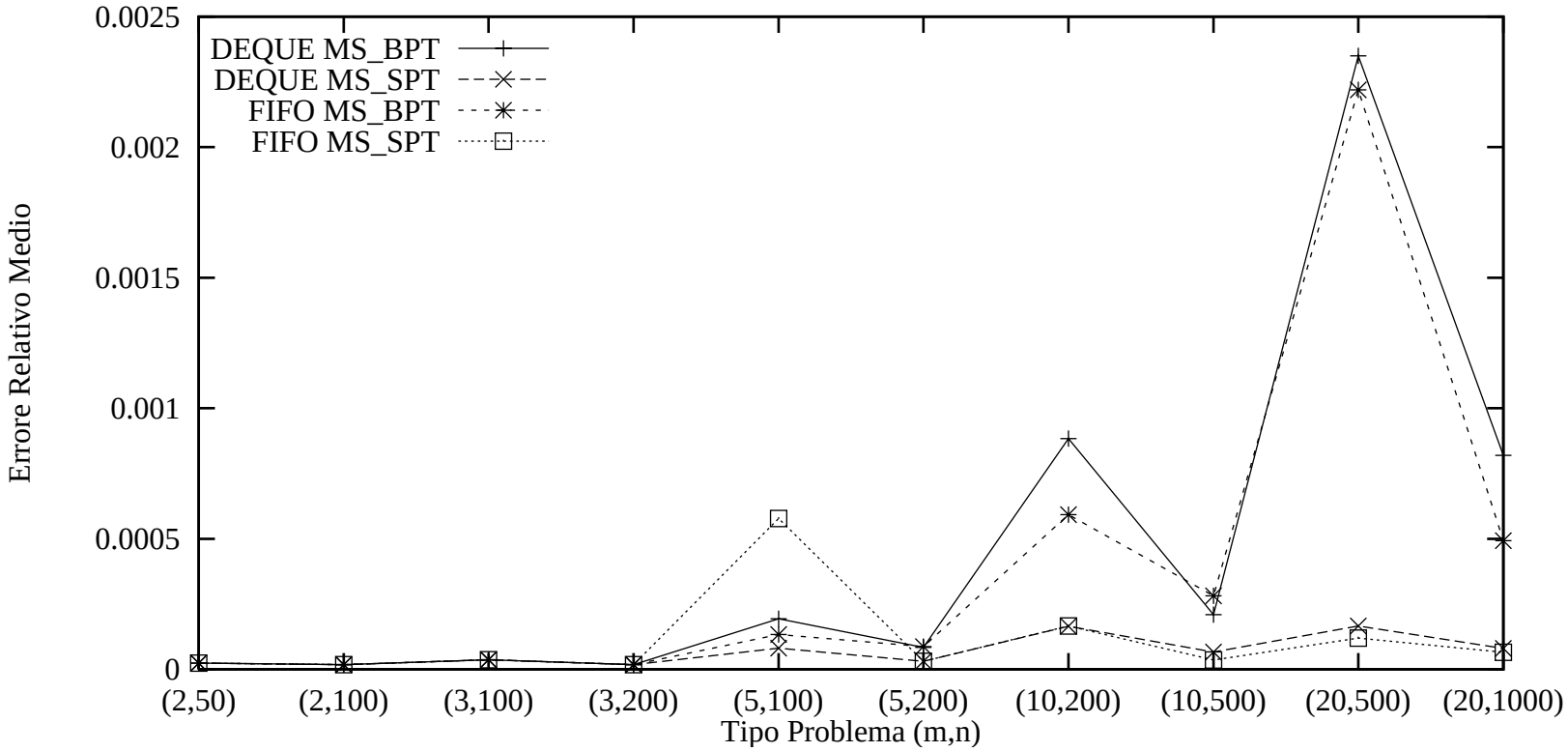

Figura 4.19: Confronto dei risultati ottenuti mediante algoritmi diversi eseguiti su istanze di tipo **alm-2** a partire da soluzioni iniziali ottenute con l'euristica "List Scheduling"

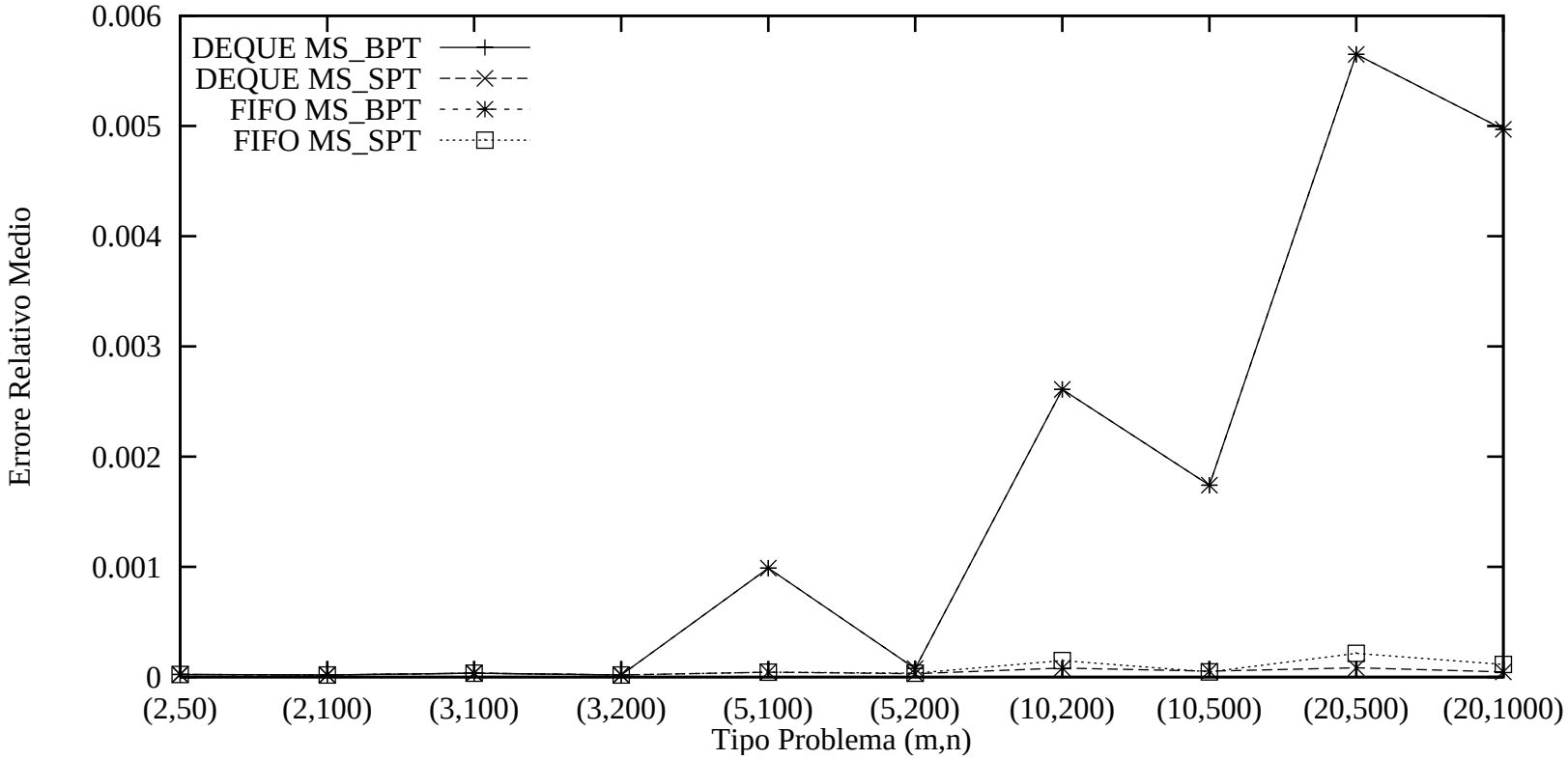

Figura 4.20: Confronto dei risultati ottenuti mediante algoritmi diversi eseguiti su istanze di tipo **alm-2** a partire da soluzioni iniziali ottenute con l'euristica "Longest Processing Time"

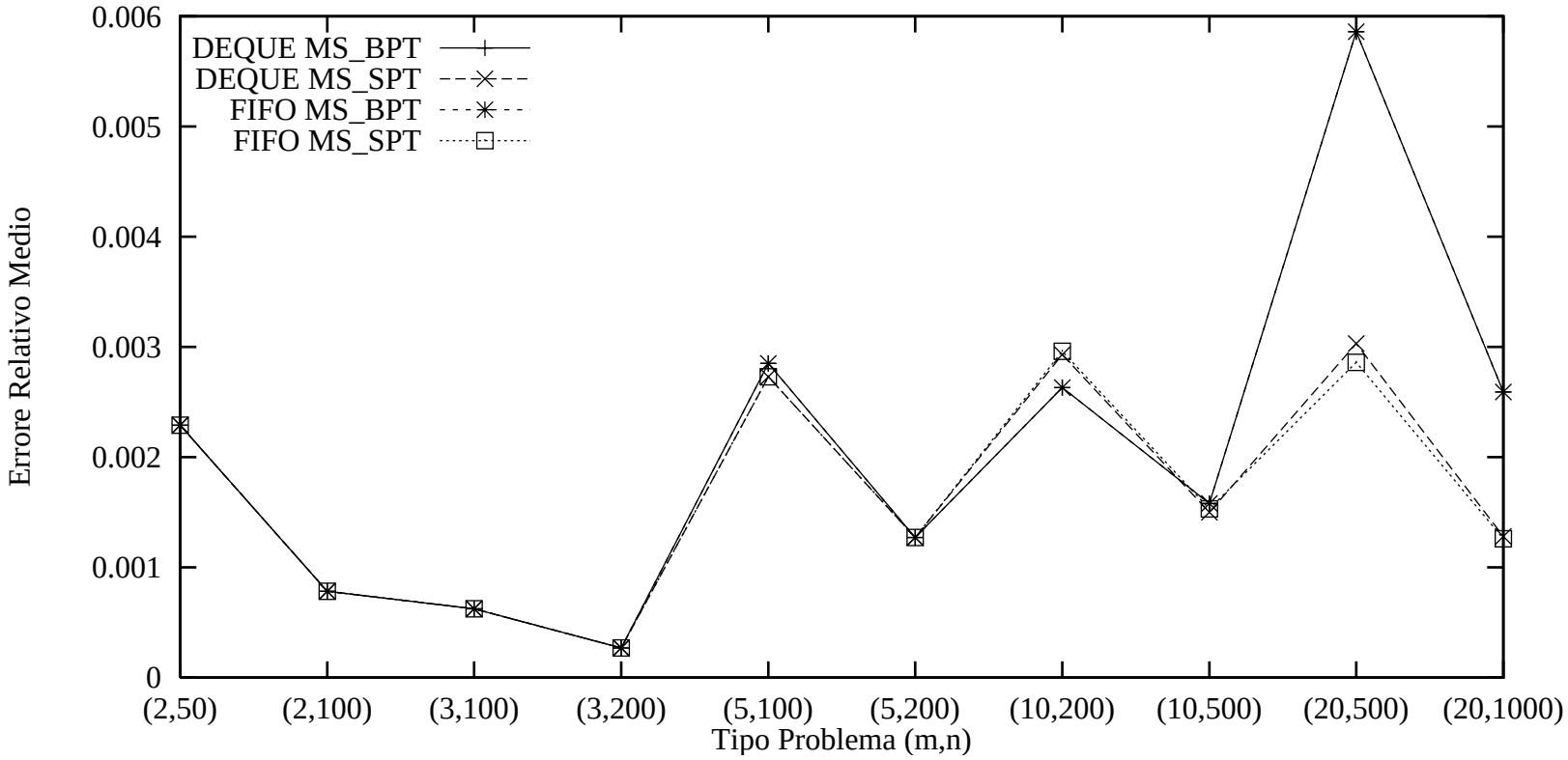

Figura 4.21: Confronto dei risultati ottenuti mediante algoritmi diversi eseguiti su istanze di tipo **alm-2** a partire da soluzioni iniziali ottenute con l'euristica "Multifit"

Appendice A

Il codice

A.1 Codice relativo al generatore mktest

A.1.1 File: mktest.cc

```
#include <iostream.h>
#include <getopt.h>
#include <time.h>
#include <math.h>
#include <stdlib.h>
#include <iomanip.h>

#include "ACG.h"
#include "RndInt.h"

int main( int argc, char** argv ){

  int numLavori = 50;
  int numMacchine = 2;
  int rangeMax = 30;
  int rangeMin = 1;
```

```
int seed = time(NULL);

int opt;

while( (opt = getopt(argc,argv,"-l:m:a:b:s:") ) != -1 )
  switch( opt ){
  case 'l': numLavori   = atoi(optarg);break;
  case 'm': numMacchine = atoi(optarg);break;
  case 'a': rangeMin    = atoi(optarg);break;
  case 'b': rangeMax    = atoi(optarg);break;
  case 's': seed        = atoi(optarg);break;
  case '?':
    cout<<endl<<"usage: "<<argv[0]<<" [-lnumLavori
                -mnumMacchine -arangeMin -brangeMax]"<<endl;
    cout<<"\t defalut rangeMin 1"<<endl;
    cout<<"\t defalut rangeMax 30"<<endl;
    return 1;
  }

const int minCI   = int(ceil((rangeMax-rangeMin)*0.8));
const int maxCI   = rangeMax;

const int minCII  = int(ceil((rangeMax-rangeMin)*0.4));
const int maxCII  = int(ceil((rangeMax-rangeMin)*0.7));

const int minCIII = rangeMin;
const int maxCIII = int(ceil((rangeMax-rangeMin)*0.2));

const int deltaMaxCI   = int(floor((maxCI-minCI)*0.1));
const int deltaMaxCII  = int(floor((maxCII-minCII)*0.1));
const int deltaMaxCIII = int(floor((maxCIII-minCIII)*0.1));
```

```
const int deltaMinCI   = 0;
const int deltaMinCII  = deltaMaxCI+1;
const int deltaMinCIII = 0;

ACG* gen = new ACG(seed,220);

RandomInteger classeI(minCI,maxCI-deltaMaxCI,gen);
RandomInteger classeII(minCII,maxCII-deltaMaxCII,gen);
RandomInteger classeIII(minCIII,maxCIII-deltaMaxCIII,gen);

RandomInteger deltaCI(deltaMinCI,deltaMaxCI,gen);
RandomInteger deltaCII(deltaMinCII,deltaMaxCII,gen);
RandomInteger deltaCIII(deltaMinCIII,deltaMaxCIII,gen);

RandomInteger numD(0,numLavori/3,gen);

int i = 0;

int MaxCI = int(floor(numLavori*0.33));
int MaxCIII =int(ceil(numLavori*0.01));
int MaxCII = numLavori-MaxCI-MaxCIII;

cout<<numLavori<<" "<<numMacchine<<endl;

while( true ){

  int tempMax = numD();

  int kMaxCI = int(floor(tempMax*0.3));
```

```
int kMaxCIII =int(ceil(tempMax*0.01));
int kMaxCII = tempMax-kMaxCI-kMaxCIII;

int temp = classeI();
for( int k = 0; (k < kMaxCI)&&(MaxCI) ; k++){
  cout<<temp+deltaCI()<<endl,i++;
  MaxCI--;
  if( i == numLavori ){
    delete gen;
    return 0;
  }
}

temp = classeII();
for( int k = 0; (k < kMaxCII)&&(MaxCII) ; k++){
  cout<<temp+deltaCII()<<endl,i++;
  MaxCII--;
  if( i == numLavori ){
    delete gen;
    return 0;
  }
}

temp = classeIII();
for( int k = 0; (k < kMaxCIII)&&(MaxCIII) ; k++){
  cout<<temp+deltaCIII()<<endl,i++;
  MaxCIII--;
  if( i == numLavori ){
    delete gen;
    return 0;
  }
```

```
    }
  }

  delete gen;
  return 0;
}
```

A.2 Codice relativo agli algoritmi proposti

A.2.1 File: ctms.h

```
// -*- C++ -*-
// File:   ctms.h
// Autore: Emiliano Necciari
//

#ifndef __CTMS_H
#define __CTMS_H

#include <math.h>
#include <iomanip.h>
#include <fstream.h>

#include "ctmsopt.h"
#include "cputime.h"
#include "opt.h"
#include "printopt.h"

// L'oggetto CTMS e' un'implementazione di un'euristica di
// ricerca locale per il problema  di Assegnamento di Lavori
// a Macchine.
```

```
// Questa euristica si basa su una funzione intorno
// particolare.
// La funzione intorno è rappresentata mediante un grafo di
// miglioramento  costruito in base alla soluzione corrente.
// Nel grafo cerchiamo cammini(o cicli) disgiunti di costo
// negativo con la funzione _trovaCDisgiuntoNegativo().
//
// Cambiando le opzioni del file ctmsopt.h si possono ottenere
// diverse varianti
//

class CTMS{

  Index _n;  // numero lavori
  Index _m;  // numero macchine

  // Grafo di migliramento --
  //   Il grafo di miglioramento e' memorizzato per stelle
  //   uscenti mediante i vettori: nodi, archi, c.
  //   La stella uscente del nodo i inizia all'indirizzo
  //   archi[nodi[i]] e termina all'indirizzo
  //   archi[nodi[i+1]-1].
  //

  // nodi       archi                    c
  //|    |     |     |                |     |
  //| i  |---->|  j  | inizio FS(i)   | cij | costo arco ij
  //| i+1|-    |     |                |     |
  //|    | |   |     |                |     |
  //|    | |   |  k  | fine FS(i)     |     |
```

```
  //|    |  -->|  h  | inizio FS(i+1) |     |

  PtrTpElemNodi  nodi;
  PtrTpNodo      archi;
  PtrTpCosti     c;
  // ------------------------

  PtrTpCosti l;
  PtrTpEtichetta pred;
  PtrTpEtichetta nm;

  PtrTpCosti  d;      // vettore dei tempi di lavorazione
  PtrTpSol    S;      // soluzione corrente
  TpCosti     cS;     // massimo tempo di lavorazione
  PtrTpCosti  cM;     // vettore dei tempi di lavorazione di
                      // ogni macchina
  TpEtichetta K;      // numero di macchine critiche

  BOOL* macchina;

  Index _i;
  Index _j;

  // LISTA ------------------
  TpEtichetta _last;   // puntatore all'ultimo elemento della
                       // lista Q
  PtrTpEtichetta Q;    //
  PtrTpEtichetta Q_n;  // primo elemento della lista
#ifdef DEQUE
  BOOL* giaInQ;
#endif
```

```
#ifndef GESTIONE_Q_INLINE
  inline void _insert( const TpEtichetta elem );
  inline TpEtichetta _get( );
  inline void _creaQ( );
#endif
  // ------------------------

  TpCosti LB;         // Lower Bound
  TpCosti cSolIn;     // soluzione iniziale
  Index   nIter;

  CpuTime cputime;

  // crea il grafo di miglioramento in base alle informazioni
  // sui tempi di lavorazione dei lavori, delle macchine
  void _creaGrafo( void );

  // dopo aver trovato un cammino( o ciclo) di miglioramento
  // _aggiornaSoluzione( ) effettua lo scambio corrispondente
  // al cammino( o ciclo).
  void _aggiornaSoluzione( void );

  BOOL _trovaKCiclo( const int );

  BOOL _isDisjoint( void );

public:

// Costruttore della classe CTMS.
// Parametri
//        numLavori  : indica il numero di lavori
```

```
//        numMacchine: indica il numero delle macchine
//        d          : vettore delle durate dei lavori
  CTMS(const int numLavori,
       const int numMacchine,
       cPtrTpCosti& d);

  CTMS(ifstream& in);
  ~CTMS();

  int solve( void );

  void soluzioneIniziale(cPtrTpSol& Sol);

  void printProblema ( const int stampa =
       STAMPA_COINCISO ) const;
  void printSoluzione( const int stampa =
       STAMPA_COINCISO ) const;

  PtrTpCosti& getD( void ){ return d; };
  Index getNumLavori( void ){ return _n; };
  Index getNumMacchine( void ){ return _m; };

};
#endif
```

A.2.2 File: ctms.C

```
// -*- C++ -*-
// File:   ctms.C
// Autore: Emiliano Necciari
```

```
#include "ctms.h"

CTMS::CTMS(const int numLavori,const int numMacchine,
   cPtrTpCosti& d)
  : _n(numLavori),_m(numMacchine) {

  l    = new TpCosti[(_n+_m)];

  pred = TpEtichettaAlloc(_n+_m);
  nm   = TpEtichettaAlloc(_n+_m);

  this->d = new TpCosti[_n];   // vettore dei tempi di
                               // lavorazione;
  for( _i = 0; _i < _n; _i++ )
    this->d[_i] = d[_i];
  S = new TpSol[_n];   // soluzione corrente

  cM = new TpCosti[_m];  // tempo di lavorazione di ogni
                         // macchina

  // Grafo di migliramento
  nodi    = new TpElemNodi[_n+1];
  nodi[0] = 0;           // il primo elemento dell'array
                         // nodi è sempre 0

  archi= TpNodoAlloc(_n*(_n+_m-2));
  c    = new TpCosti[_n*(_n+_m-2)];
  // ----------------------

  // LISTA Q
  Q = TpEtichettaAlloc(_n+1);
```

```
  Q_n =Q +_n;
#ifdef DEQUE
  giaInQ = new BOOL[_n];
#endif
  // ----------------------
  macchina =  new BOOL[_m];


  TpCosti totale = TpCosti_0;
  TpCosti max = TpCosti_MIN;

  PtrTpCosti last = this->d+_n;
  for( ; --last >= this->d ; ){
    totale += *last;

    if( max < *last )
      max = *last;
  }

#if( TIPO_COSTI == REALI )
  LB =  totale / (TpCosti)_m;
#else
  LB = (TpCosti)ceil( totale / _m );
#endif

  if( LB < max)
    LB = max;
};

CTMS::CTMS(ifstream& in){
  // input numero lavori, numero macchine
```

```
  in>>_n>>_m;
  // fine input numero lavori, numero macchine
  l    = new TpCosti[(_n+_m)];

  pred = TpEtichettaAlloc(_n+_m);
  nm   = TpEtichettaAlloc(_n+_m);

  d = new TpCosti[_n];   // vettore dei tempi di lavorazione;
  for( _i = 0; _i < _n; _i++ )
    in>>d[_i];
  S = new TpSol[_n];   // soluzione corrente

  cM = new TpCosti[_m];  // tempo di lavorazione di ogni
                         // macchina

  // Grafo di migliramento
  nodi    = new TpElemNodi[_n+1];
  nodi[0] = 0;           // il primo elemento dell'array nodi
                         // è sempre 0

  archi= TpNodoAlloc(_n*(_n+_m-2));
  c    = new TpCosti[_n*(_n+_m-2)];
  // ----------------------

  // LISTA Q
  Q = TpEtichettaAlloc(_n+1);
  Q_n =Q +_n;
#ifdef DEQUE
  giaInQ = new BOOL[_n];
#endif
  // ----------------------
```

```
  macchina =  new BOOL[_m];


  TpCosti totale = TpCosti_0;
  TpCosti max = TpCosti_MIN;

  PtrTpCosti last = this->d+_n;
  for( ; --last >= this->d ; ){
    totale += *last;

    if( max < *last )
      max = *last;
  }

#if( TIPO_COSTI == REALI )
  LB =  totale / (TpCosti)_m;
#else
  LB = (TpCosti)ceil( totale / _m );
#endif

  if( LB < max)
    LB = max;
};


CTMS::~CTMS(){

  delete l;
  delete pred;
  delete nm;
```

```
  delete d;
  delete S;

  delete cM;

  delete nodi;
  delete archi;
  delete c;

  delete Q;
#ifdef DEQUE
  delete giaInQ;
#endif

  delete macchina;
};


//Crea il grafo di miglioramento
//
void CTMS::_creaGrafo( void ){

  register Index numNodo = 0;
  register Index numArco = 0;
  register Index j;

  for(  ; numNodo < _n ;  ){

    j = 0;

    for(  ; j < _n ; j++){
```

```
      if( ( S[j] != S[numNodo] )&&
          ( cM[S[j]]-d[j]+d[numNodo] < cS) ){
        archi[numArco] = j;

#ifdef COSTI_ARCHI_NEG
        c[numArco] = cM[S[j]]+d[numNodo]-d[j]-cS;
#else
        c[numArco] = TpCosti_0;

        if( cM[S[j]] == cS )
          c[numArco] = d[numNodo]-d[j];
#endif
        numArco++;
      }
    }

    for( j = 0 ; j < _m ; j++ ){

      if( ( S[numNodo] != j  )&&
          ( cM[j]+d[numNodo] < cS) ){
#ifdef COSTI_ARCHI_NEG
        c[numArco] = cM[j]+d[numNodo]-cS;
#else
        c[numArco] = TpCosti_0;
#endif
        archi[numArco] = j+_n;
        numArco++;
      }
    }
    numNodo++;
```

```
    nodi[numNodo] = numArco;
  }
};

// Aggiorna la soluzione in base ad un cammino
// o ad un ciclo di miglioramento
//
void CTMS::_aggiornaSoluzione(){
  register Index u;
  register Index last;
  register Index oldSu;

  if( _j < _n ){ // abbiamo trovato un ciclo di miglioramento
    // il ciclo di miglioramento e' _i->_j->...->pred[_i]->_i
    u    = _i;
    last = _i;
    oldSu = S[u];

    // aggiorniamo la soluzione corrente
    while( u != _j ){
      TpSol temp = S[pred[u]];
      cM[oldSu] += d[pred[u]]-d[u];
      S[pred[u]] = oldSu;
      oldSu = temp;
      u = pred[u];
    }

    cM[oldSu] += d[last]-d[_j];
    S[last] = oldSu;
  }
  else{ // abbiamo trovato un cammino di miglioramento
```

```
    // il cammino di miglioramento e' _s->...->pred[_i]->_i->_j
    u    = pred[_j];
    oldSu = S[u];
    S[u] = _j-_n;
    cM[_j-_n] += d[u];

    // aggiorniamo la soluzione corrente
    while( pred[u] != -1 ){
      TpSol temp = S[pred[u]];
      cM[oldSu] += d[pred[u]]-d[u];
      S[pred[u]] = oldSu;
      oldSu = temp;
      u = pred[u];
    }
    cM[oldSu] -= d[u];
  }

  cS = TpCosti_0;
  K  = 0;
  register PtrTpCosti iter = cM+_m;
  for( ; --iter >= cM ; )
    if( *iter == cS )
      K++;
    else
      if( *iter > cS ){
        cS = *iter;
        K  = 1;
      }

  // aggiornata la soluzione corrente dobbiamo creare
  // il grafo di miglioramento corrispondente
```

```
  // inizio _creaGrafo();
  register Index numNodo = 0;
  register Index numArco = 0;
  register Index j;

  for(  ; numNodo < _n ;  ){

    j = 0;

    for(  ; j < _n ; j++){

      if( ( S[j] != S[numNodo] )&&
          ( cM[S[j]]-d[j]+d[numNodo] < cS) ){
        archi[numArco] = j;

#ifdef COSTI_ARCHI_NEG
        c[numArco] = cM[S[j]]+d[numNodo]-d[j]-cS;
#else
        c[numArco] = TpCosti_0;

        if( cM[S[j]] == cS )
          c[numArco] = d[numNodo]-d[j];
#endif
        numArco++;
      }
    }

    for( j = 0 ; j < _m ; j++ ){

      if( ( S[numNodo] != j  )&&
```

```
          ( cM[j]+d[numNodo] < cS) ){
#ifdef COSTI_ARCHI_NEG
        c[numArco] = cM[j]+d[numNodo]-cS;
#else
        c[numArco] = TpCosti_0;
#endif
        archi[numArco] = j+_n;
        numArco++;
      }
    }
    numNodo++;
    nodi[numNodo] = numArco;
  }
  //// fine _creaGrafo();
};

// cerca un cammino o ciclo disgiunto di costo negativo // BOOL
BOOL CTMS::_trovaKCiclo( const int s ){

  register Index u;
  register Index k;

  // Inizializzazione dei vettori l,pred,nm
  // InitToMax(l , _n+_m);
  // Init(pred,-1,_n+_m);
  // Init(nm  , 0,_n+_m);

  PtrTpCosti ptr2l = l;
  PtrTpEtichetta ptr2pred = pred , ptr2nm = nm;
  for( u = _n+_m; u--  ; )
    *(ptr2l++)=TpCosti_MAX,*(ptr2pred++)=-1,*(ptr2nm++)=0;
```

```
#ifdef BOTTLENECK_PATH
  l[s] = TpCosti_MIN;
#else
  l[s] = TpCosti_0;
#endif

  if( cM[S[s]] == cS )
    nm[s] = 1;

#ifndef GESTIONE_Q_INLINE
  _creaQ();
#else
  // svuoto la coda
  //Init(Q,-1,_n);
  int tmp;
  while((tmp = *Q_n) != _n ){
    *Q_n = Q[tmp];
    Q[tmp] = -1;
  }
  _last = _n;
#ifdef DEQUE
  Init(giaInQ,FALSE,_n);
#endif
  // fine svuota coda Q;
#endif // GESTIONE_Q_INLINE

#ifndef GESTIONE_Q_INLINE
  _insert(s);
#else
```

```
  // inizio _insert(s);
  // in questo caso la lista e' vuota
#ifdef DEQUE
  giaInQ[s] = TRUE;
#endif
  Q[_last] = s;
  Q[s] = _n;
  _last = s;
  // fine _insert(s);
#endif // GESTIONE_Q_INLINE

  while( *Q_n != _n ){ // la lista Q non e' vuota

#ifndef GESTIONE_Q_INLINE
    _i = _get();
#else
    // inizio _i = get();
    _i = *Q_n;
    *Q_n = Q[_i];
    Q[_i] = -1;
    if( *Q_n == _n )
      _last = _n;
    //fine get();
#endif
    // verifico se il cammino da s ad i e' disgiunto

    // Init(macchina,FALSE,_m);
    BOOL* ptr2macchina = macchina;
    for( k = _m ; k-- ; )
      *(ptr2macchina++)=FALSE;
    // fine Init
```

```
    u = _i;
    BOOL isDisjoint = TRUE;
    while( u != -1 ){
      if( macchina[S[u]] ){
isDisjoint = FALSE;
break;
      }
      macchina[S[u]] = TRUE;
      u = pred[u];
    }
    ////// fine verifica

    if( isDisjoint ){ // il cammino da s a _i e' disgiunto
      Index h = nodi[_i];
      Index fineFS = nodi[_i+1];
      for( ; h <  fineFS ; h++ ){
        _j = archi[h];

        Index tmpMacchina = ( _j < _n ? S[_j] : _j-_n );
        if( !macchina[tmpMacchina] ){
          // il cammino da s a _j e' disgiunto
#ifdef BOTTLENECK_PATH
          TpCosti tmpMax = max(l[_i],c[h]);
#else
          TpCosti tmplc = l[_i] + c[h];
#endif
          if(
#ifdef BOTTLENECK_PATH
     l[_j] > tmpMax
#else
```

```
     l[_j] > tmplc
#endif
     ){
#ifdef BOTTLENECK_PATH
    l[_j] = tmpMax;
#else
    l[_j] = tmplc;
#endif

    // aggiorniamo nm[j]
    nm[_j] = nm[_i];

    if( cM[tmpMacchina] == cS )
      nm[_j]++;

    // aggiorniamo pred[_j]
    pred[_j] = _i;

    if( _j >= _n ){
      if( nm[_j] == K )
// abbiamo trovato un cammino di miglioramento
return TRUE;
    }
    else{
#ifndef GESTIONE_Q_INLINE
      _insert(_j);
#else
      // inizio _insert(_j);
#ifdef DEQUE
      if( Q[_j] == -1 ){
                if( giaInQ[_j] ){
```

```
                    Q[_j] = *Q_n;
                    *Q_n = _j;
                  }
                  else{
                    giaInQ[_j] = TRUE;
                    Q[_last] = _j;
                    Q[_j] = _n;
                    _last = _j;
                  }
                }
#else // non e' definito DEQUE
      if( Q[_j] == -1 ){ // _j non e' in lista
                Q[_last] = _j;   // _j viene inserito in coda
                Q[_j] = _n;
                _last = _j;
      }
#endif
      // fine _insert(_j);
#endif // GESTIONE_Q_INLINE
    }
  }
}
else{
  // il cammino da s a _j non e' disgiunto
  if( _j < _n ){
    // controlliamo se _j appartiene al cammino da s a _i
    // se _j appartiene al cammino s->..->_i allora
    // abbiamo trovato un ciclo di miglioramento
    for( u = pred[_i]; u != -1 ; u = pred[u])
      if( u == _j ){
                k = nm[_i];
```

```
                if( pred[_j] != -1 )
                  k -= nm[pred[_j]];
                  if( k ==  K )
                  // abbiamo trovato un ciclo di miglioramento
                  return TRUE;
                break;
              }
          }
        }
      }
    }
  }
  return FALSE;
};


//Verifca che il cammino da s a i sia disgiunto
//
BOOL CTMS::_isDisjoint( void ) {

  TpEtichetta u = _i;
  Init(macchina,FALSE,_m);

  while( u != -1 ){
    if( macchina[S[u]] ) return FALSE;
    macchina[S[u]] = TRUE;
    u = pred[u];
  }

  return TRUE;
};
```

```
// Gestione Q
//
#ifndef GESTIONE_Q_INLINE
void CTMS::_creaQ( ){
  Init(Q,-1,_n);
  *Q_n = _last = _n;
#ifdef DEQUE
  Init(giaInQ,FALSE,_n);
#endif
};

void CTMS::_insert( const TpEtichetta elem ){
#ifdef DEQUE
  if( Q[elem] == -1 ){
    if( giaInQ[elem] ){
      Q[elem] = *Q_n;
      *Q_n = elem;
    }
    else{
      giaInQ[elem] = TRUE;
      Q[_last] = elem;
      Q[elem] = _n;
      _last = elem;
    }
  }
#else // non e' definito DEQUE
  if( Q[elem] == -1 ){
    Q[_last] = elem;
    Q[elem] = _n;
    _last = elem;
```

```
  }
#endif
}

TpEtichetta CTMS::_get( ){
  temp = *Q_n;
  *Q_n = Q[temp];
  Q[temp] = -1;
  if( *Q_n == _n )
    _last = _n;
  return temp;
}
#endif // GESTIONE_Q_INLINE
//----------------------------------------

// risolve il problema di assegnamento di
// Lavori a Macchine
//
int CTMS::solve( ){
  cputime.start();

  _creaGrafo();

  nIter = 0;

#ifdef POSTOTT
  BOOL cambiata = TRUE;
#endif

  // inizio _creaQ();
  PtrTpEtichetta ptr2Q = Q;
```

```
  for( int u = _n ; u-- ; )
    *(ptr2Q++) = -1;
  *Q_n = _last = _n;
  // fine  _creaQ();

  for( int s = 0 ; (s < _n)&&(nIter<1000) ; s++ ){

    nIter++;
    while( _trovaKCiclo( s ) ){
      _aggiornaSoluzione( );
      nIter++;
#ifdef POSTOTT
      cambiata = TRUE;
#endif
      s = 0;
    }


    if( cS == LB ) // il valore della soluzione correnente
      // e' uguale al LowerBound
      return 0;

#ifdef POSTOTT
    if( cambiata ){ //la soluzione e' cambiata
      BOOL postott = FALSE;
      cambiata = FALSE;
      for( register Index i = 0 ; i < _n ; i++){
        Index   tempmacc;
        TpCosti maxsaving = TpCosti_MAX;
        register Index j = 0;
        for(  ; j < _m ; j++){
```

```
      if( ( S[i] != j ) &&
          ( cM[j]+d[i] - cS < maxsaving ) ){
        maxsaving =  cM[j]+d[i] - cS;
        tempmacc = j;
      }
    }

    if( maxsaving <= 0 ){
      Index old = S[i];
      S[i] = tempmacc;
      cM[tempmacc] += d[i];
      cM[old] -= d[i];

      postott =TRUE;
      cS = TpCosti_0;
      K  = 0;
      register PtrTpCosti iter = cM+_m;
      for( ; --iter >= cM ; )
        if( *iter == cS )
          K++;
        else
          if( *iter > cS ){
            cS = *iter;
            K  = 1;
          }
    }
  }

  if( postott ){
    //  _creaGrafo();
    // inizio _creaGrafo();
```

```
register Index numNodo = 0;
register Index numArco = 0;
register Index j;

for(  ; numNodo < _n ;  ){

  j = 0;

  for(  ; j < _n ; j++){
    if( ( S[j] != S[numNodo] )&&
                ( cM[S[j]]-d[j]+d[numNodo] < cS ) ){
              archi[numArco] = j;

#ifdef COSTI_ARCHI_NEG
              c[numArco] = cM[S[j]]+d[numNodo]-d[j]-cS;
#else
              c[numArco] = TpCosti_0;
              if( cM[S[j]] == cS )
                c[numArco] = d[numNodo]-d[j];
#endif
              numArco++;
            }
          }

  for( j = 0 ; j < _m ; j++ ){
    if( ( S[numNodo] != j  )&&
                ( cM[j]+d[numNodo] < cS) ){
#ifdef COSTI_ARCHI_NEG
              c[numArco] = cM[j]+d[numNodo]-cS;
#else
              c[numArco] = TpCosti_0;
```

```
#endif
              archi[numArco] = j+_n;
              numArco++;
            }
          }
          numNodo++;
          nodi[numNodo] = numArco;
        }
        //// fine _creaGrafo();
      }// fine post ottimizzazione
    }// if( cambiata )
#endif //POSTOTT
  }// fine for( int s= 0...
  cputime.stop();
  return 0;
};

void CTMS::soluzioneIniziale(cPtrTpSol& Sol){
  if( Sol ){
    Init(cM,0,_m);
    for( int i = 0; i <_n ; i++ ){
      S[i] = Sol[i];
      cM[S[i]] += d[i];
    }
  }
  else{
    cerr<<endl<<"Errore: la soluzione iniziale non è valida";
    exit(1);
  }

  cS = TpCosti_0;
```

```
  PtrTpCosti last = cM+_m;
  for( ; --last >= cM ; )

    if( *last > cS )
      cS = *last;

  K = 0;
  last = cM+_m;
  for( ; --last >= cM ; )

    if( *last == cS )
      K++;

  cSolIn = cS;
};


void CTMS::printProblema(const int stampa) const {
  switch( stampa ){
  case STAMPA_PROBLEMA:{
    cout<<endl<<"Assegnamento di Lavori a Macchine"<<endl;
    cout<<endl<<"Numero Lavori:   "<<_n;
    cout<<endl<<"Numero Macchine: "<<_m;
    cout<<endl<<"Lower Bound del tempo di lavorazione: "<<LB;

    cout<<endl<<"d=[";
    for( int i = 0 ; i < _n ; i++ )
      cout<<" "<<d[i];
    cout<<" ]";
    break;
```

```
  }
  case  STAMPA_COINCISO:{
#if( TIPO_COSTI == REALI )
    cout.form("%3d %5d %5.2f %5.2f ",_m,_n,LB,cSolIn);
#else
    cout.form("%3d %5d %9d %9d ",_m,_n,LB,cSolIn);
#endif
    break;
  }
  }
};

void CTMS::printSoluzione( const int stampa ) const {

  switch( stampa ){
  case STAMPA_PROBLEMA:{
    cout<<endl<<"Soluzione";

    cout<<endl<<"S=[";
    for( int i = 0 ; i < _n ; i++ )
      cout<<" "<<S[i];
    cout<<" ]";
    cout<<endl<<"Makespan: "<<cS;
    cout<<endl<<"user time:   "<<cputime.u;
    cout<<endl<<"system time: "<<cputime.s;
    break;
  }
  case  STAMPA_COINCISO:{
#if( TIPO_COSTI == REALI )
    cout.form("%5.2f %9.2e %9.2e %6.2f %5d",
      cS,
```

```
      (cS-LB )/LB,
      (cSolIn-LB )/LB,
      cputime.u,nIter);
#else
    cout.form("%9d %9.2e %9.2e %6.2f %5d",
      cS,
      (double)(cS-LB )/ (double)LB,
      (double)(cSolIn-LB )/ (double)LB,
      cputime.u,nIter);
#endif
    break;
  }
  }
  cout<<endl;
};
```

A.2.3 File: ctmsopt.h

```
// -*- C++ -*-
//   File:   ctmsopt.h
//   Autore: Emiliano Necciari

#ifndef __CTMSOPT_H
#define __CTMSOPT_H

//  Definendo DEQUE, la coda usata in _trovaKCiclo(),
//  viene gestita secondo la politica deque, cioe' un nodo
//  viene inserito in fondo alla coda se non e' stato inserito
//  precedentemente in coda, altrimenti viene inserito in
//  testa.
//  L'estrazione di un nodo dalla coda avviene sempre dalla
```

```
//  testa.

//  Se DEQUE non e' definito la gestione della coda avviene
//  con politica FIFO.

//#define DEQUE


//  Definendo BOTTLENECK_PATH la procedura _trovaKCiclo(),
//  ricerca un cammino(ciclo) di tipo bottleneck.
//  I costi degli archi del grafo di miglioramento
//  sono tutti negativi.

//#define BOTTLENECK_PATH

//  Definenedo COSTI_ARCHI_NEG, i costi degli archi del grafo
//  di miglioramento sono tutti negativi.
#ifdef BOTTLENECK_PATH
#define COSTI_ARCHI_NEG
#else
#undef COSTI_ARCHI_NEG
#endif


//  se e' definito POSTOTT l'algoritmo esegue una fase di post
//  ottimizzazione

#define POSTOTT


// se GESTIONE_Q_INLINE e' definito allora i metodi
```

```
// _insert(s), _get(), _creaQ() della classe  ctms sono
// direttamente sostituiti nel codice
//

#define GESTIONE_Q_INLINE

#endif
```

A.2.4 File: opt.h

```
// -*- C++ -*-
//   File:   opt.h
//   Autore: Emiliano Necciari

#ifndef __OPT_H
#define __OPT_H



#include <values.h>
#include <iostream.h>
#include <stdlib.h>

// se TIPO_COSTI == REALI allora gli elementi del vettore dei
// costi sono di tipo double altrimenti di tipo int
#define INTERI 1
#define REALI  2
#define TIPO_COSTI INTERI
//#define TIPO_COSTI REALI

////////////////////////////////////////////////////////////
#if( TIPO_COSTI == INTERI ) // i costi associati agli archi
```

```
                            // sono interi

typedef int TpCosti;

const TpCosti TpCosti_0   = 0;
const TpCosti TpCosti_MIN = INT_MIN;
const TpCosti TpCosti_MAX = INT_MAX;

#define EQZ(a) ( a == 0 )
#define GTZ(a) ( a >  0 )
#define GEZ(a) ( a >= 0 )
#define LTZ(a) ( a <  0 )
#define LEZ(a) ( a <= 0 )

////////////////////////////////////////////////////////////
#else  // i costi associati agli archi sono di tipo double

typedef double TpCosti;

const TpCosti EPS_PREC = 1e-20;

const TpCosti TpCosti_0 = 0.0;
const TpCosti TpCosti_MIN = -DBL_MIN;
const TpCosti TpCosti_MAX = DBL_MAX;


#define EQZ(a) (( a < EPS_PREC ) && ( a > -EPS_PREC ))
#define GTZ(a) ( a >  -EPS_PREC )
#define GEZ(a) ( a >= EPS_PREC  )
#define LTZ(a) ( a <= -EPS_PREC )
#define LEZ(a) ( a <  EPS_PREC  )
```

```
#endif
////////////////////////////////////////////////////////////

typedef int          TpEtichetta; // tipo delle etichette dei
                                  // nodi
typedef int          TpElemNodi;  // tipo degli elementi
                                  // dell'array "nodi"
typedef TpEtichetta  TpNodo;      // tipo dell'etichetta di un
                                  // nodo
typedef TpEtichetta  TpSol;       // tipo degli elementi
                                  // dell'array S
typedef int          Index;

typedef TpElemNodi*  PtrTpElemNodi;
typedef TpEtichetta* PtrTpEtichetta;
typedef TpSol*       PtrTpSol;
typedef TpCosti*     PtrTpCosti;
typedef TpNodo*      PtrTpNodo;

typedef const TpElemNodi*  cPtrTpElemNodi;
typedef const TpEtichetta* cPtrTpEtichetta;
typedef const TpSol*       cPtrTpSol;
typedef const TpCosti*     cPtrTpCosti;
typedef const TpNodo*      cPtrTpNodo;

typedef bool BOOL;
const BOOL TRUE  = true;
const BOOL FALSE = false;

////////////////////////////////////////////////////////////
```

```
#include <string.h>

template<class tipo>
inline void Init(tipo* mem, int val,int N){
  memset(mem, val,N*sizeof(tipo));
};

template<class tipo>
inline void InitToMax(tipo* mem,int N){
  memset(mem,127,N*sizeof(tipo));
};

template<class tipo>
inline void InitToMin(tipo* mem,int N){
  memset(mem,-128,N*sizeof(tipo));
};

#define TpEtichettaAlloc( N ) new TpEtichetta[N]
#define TpNodoAlloc( N ) new TpNodo[N]

////////////////////////////////////////////////////////////

#define max( i , j ) (i >? j )

#endif
```

A.2.5 File: csi.h

```
// -*- C++ -*-
// File:   csi.h
// Autore: Emiliano Necciari
```

```
#ifndef __CSI_H
#define __CSI_H

#include "opt.h"


class CSI{
protected:
  int numLavori;
  int numMacchine;

  PtrTpSol   sol;
  PtrTpCosti d;

  PtrTpCosti cM;

public:

  CSI(){};

  CSI(int nL,int nM,PtrTpCosti di){

    numLavori   = nL;
    numMacchine = nM;

    d = di;

    cM = new TpCosti[numMacchine];
    sol = NULL;
  };
```

```
  virtual ~CSI( void ){
    delete cM;
  };

  cPtrTpSol& getSol( void ){ return sol; }

  virtual void solve( void )=0;
};

#endif
```

A.2.6 File: clpt.h

```
// -*- C++ -*-
// File:   CLPT.h
// Autore: Emiliano Necciari

#ifndef __CLPT_H
#define __CLPT_H

#include "csi.h"
#include "mqsort.h"

class CLPT: public CSI{
  MQSort<TpCosti>* ord;

public:
  CLPT(int naL,int nM,PtrTpCosti d );

  virtual ~CLPT( void ){
```

```
    delete sol;
    delete ord;
  };

  virtual void solve( );
};

#endif
```

A.2.7 File: clpt.C

```
// -*- C++ -*-
// File:   CLPT.C
// Autore: Emiliano Necciari

#include "clpt.h"

CLPT::CLPT(int nL,int nM, PtrTpCosti di )
  : CSI(nL,nM,di) {
  sol = new TpSol[numLavori];
  ord = new MQSort<TpCosti>(di,numLavori);
  ord->sort("DECR");
};


void CLPT::solve( ){
  Init(cM,0,numMacchine);

  for( int i = 0 ; i < numLavori ; i++ ){
    TpCosti minimo = TpCosti_MAX;
    int macchina = -1;
```

```
    for( int j = 0; j < numMacchine ; j++){
      if( minimo > cM[j] ){
        macchina = j;
        minimo = cM[j];
      }
    }
    cM[macchina] += d[i];
    sol[i] = macchina;
  }
};
```

A.2.8 File: cmf.h

```
// -*- C++ -*-
// File:   cmf.h
// Autore: Emiliano Necciari

#ifndef __CMF_H
#define __CMF_H

#include "csi.h"
#include "mqsort.h"

class CMF
  :public CSI{

  MQSort<TpCosti>* ord;

  int k;

  int index;
```

```
  PtrTpSol Sol[2];
  PtrTpSol solMF;

  TpCosti low;
  TpCosti high;
  TpCosti len;

  int _FFD();

public:

  CMF(int naL,int nM, PtrTpCosti di,int numIter);

  virtual ~CMF( void ){
    delete Sol[0];
    delete Sol[1];
    delete ord;
  };

  virtual void solve( );
};

#endif
```

A.2.9 File: cmf.C

```
// -*- C++ -*-
// File:   cmf.C
// Autore: Emiliano Necciari

#include "cmf.h"
```

```
CMF::CMF(int nL,int nM, PtrTpCosti di,int numIter)
  : CSI(nL,nM,di) {

  k = numIter;

  index = 0;

  Sol[0] = new TpSol[numLavori];
  Sol[1] = new TpSol[numLavori];


  solMF = Sol[index];
  sol   = NULL;

  ord = new MQSort<TpCosti>(di,numLavori);
  ord->sort("DECR");
};

void CMF::solve( ){

  TpCosti totale = 0;
  for( int i = 0 ; i < numLavori ; i++ )
    totale += d[i];

  low  = ((totale / numMacchine)  < d[0] ? d[0] :
          (totale / numMacchine));
  high = ((2*totale / numMacchine) < d[0] ? d[0] :
          (2*totale/ numMacchine));
```

```
  for( int iterazione = 0 ;
       ( iterazione < k ) ; iterazione++ ){
    len = ( low + high ) / (TpCosti)2;

    if( _FFD( ) ){
      high = len;
      sol = solMF;
      index = (index+1) % 2;
      solMF = Sol[index];
    }
    else
      low = len;
  }
};

int CMF::_FFD(){

  //Init(cM,0,numMacchine);
  PtrTpCosti ptr2cM = cM +numMacchine;
  for( ; --ptr2cM>=cM ; )
    (*ptr2cM) = TpCosti_0;

  int lavoriAssegnati = 0;

  for( int i = 0 ; i < numLavori ; i++ ){
    for( int j = 0; j < numMacchine ; j++)
      if( cM[j] + d[i] <= len){
        cM[j] += d[i];
        solMF[i] = j;
        j=numMacchine;
        lavoriAssegnati++;
```

```
      }
  }

  return (lavoriAssegnati == numLavori);
};
```

Bibliografia

J.O. Achugbue and F.Y. Chin. Bounds on schedules for independent tasks with similar execution times. *Journal of the ACM*, 28:81–99, 1981.

R. K. Ahuja, T.L. Magnanti, and J. B. Orlin. *Network Flows: theory, algorithms and applications*. Prentice Hall, New Jersey, 1993.

R. K. Ahuja, J. B. Orlin, and D. Sharma. New neighborhood search structures for the capacitated minimum spanning tree problem. Working Paper, 1998.

A. Amberg, W. Domschke, and Stefan Voβ. Capacitated minimum spanning trees: Algorithms using intelligent search. Working Paper, 1996.

J.E. Anderson, C.A. Glass, and C.N. Potts. Machine scheduling. In E. Aarts and J. K. Lenstra, editors, *Local Search in Combinatorial Optimization*, chapter 11, pages 361–414. Wiley, 1997.

E.K. Backer and J.R. Schaffer. Solution improvement heuristics for the vehicle routing and scheduling problem with time window constraints. *Am. J. Math. Mgmt. Sci.*, 6:261–300, 1986.

L. Bodin, B.L. Golden, A. Assad, and M.O. Ball. Routing and scheduling of vehicles and crews: the state of the art. *Computers & Operations Research*, 10, 1983.

N. Christofides, A. Mingozzi, and P. Toth. Exact algoritms for the vehicle

routing problem based on spanning tree and shortest path relaxations. *Math.Prog.*, 20:255–282, 1981.

G. Clark and J.W. Wright. Scheduling of vehicle from a central depot to a number of delivery point. *Operation Research*, 12, 1964.

E.G. Coffman Jr, M.R. Garey, and D.S. Johnson. An application of bin-packing to multiprocessor scheduling. *SIAM Journal on Computing*, 7: 1–17, 1978.

L.R. Esau and K.C. Williams. On teleprocessing system design. part ii - a method for approximating the optimal network. *IBM Systems Journal*, 5: 142–147, 1966.

G. Finn and E. Horowitz. A linear time approximation algorithm for multiprocessor scheduling. *BIT*, 1979.

M.L. Fisher and R. Jaikumar. A generalized assignment heuristic for vehicle routing. *Networks*, 11:109–124, 1982.

P.M. França, M. Gendreau, G. Laporte, and F.M. Müller. A composite heuristic for the identical parallel machine scheduling problem with minimum makespan objective. *Computers & Operations Research*, 21, 1994.

D.K. Friesen and M.A. Langston. Evaluation if a multifit-based scheduling algorithm. *Journal of Algorithms*, 7:35–39, 1986.

M. Gendreau, A. Hertz, and G. Laporte. New insertion and post optimization procedures for the traveling salesman problem. *Operation Research*, 40: 1086–1094, 1992.

B.E. Gillett and L.R. Miller. A heuristic algorithm for the vehicle-dispatch problem. *Operations Research*, 22, 1974.

F. Glover. Tabu search-part i. *ORSA Journal on Computing*, 1:190–206, 1989a.

F. Glover. Tabu search-part ii. *ORSA Journal on Computing*, 2:4–32, 1989b.

R.L. Graham. Bounds on multiprocessing timing anomalies. *SIAM Journal of Applied Mathematics*, 17:416–429, 1969.

D.S. Hochbaum and D.B. Shmoys. Dual approximation algorithms for scheduling problems. *Journal of the ACM*, 34:144–162, 1987.

R. Hübscher and F. Glover. Applying tabu search with influential diversification to multiprocessor scheduling. *Computers & Operation Research*, 21, 1994.

J. K. Lenstra, A.H.G. Rinnoy Kan, and P. Brucker. Complexity of machine scheduling. In *Studies In Integer Programming*, Annals of Discrete Mathematics 1. P.L. Hammer, E.L. Johnson, B.H. Korte, G.L. Nemhauser(eds.), North-Holland, Amsterdam, 1977.

S. Lin. Computer solution to the traveling salesman problem. *Bell System Tech. J.*, 44:2245–2269, 1965.

S. Lin and B.W. Kernighan. An effective heuristic alghorithm for the traveling salesman problem. *Operations Research*, 21:498–516, 1973.

Sartaj K. Sahni. Algorithms for scheduling independent tasks. *Journal of the ACM*, 23:116–127, 1976.

Y.M. Sharaia, M. Gendreau, G. Laporte, and I.H. Osman. A tabu search algorithm for the capacitated minimum spanning tree problem. Working paper, 1995.

M.M. Solomon. Algorithms for the vehicle routing and scheduling problems with time window constraints. *Operations Research*, 35:254–265, 1987.

M.M. Solomon, E.K. Backer, and J.R. Schaffer. Vehicle routing and scheduling problems with time window constraints: Efficent implementations of solution improvement procedures. In *Vehicle Routing: Methods and*

Studies. B.L.Golden and A.A. Assad(eds.), North-Holland,Amsterdam, 1988.

P. M. Thompson and J. B. Orlin. Theory of ciclic transfers. Working Paper, 1989.

P. M. Thompson and H.N. Psaraftis. Cyclic transfer algorithms for multivehicle routing and scheduling problems. *Operations Research*, 41(5), 1993.