Politecnico di Torino

Politecnico di Torino

Ingegneria Gestionale classe L-8
A.a. 2020/2021
Sessione di Laurea novembre/dicembre 2021

Schedulazione su macchine parallele con conflitti

Una risoluzione mediante mat-euristica

Relatore:
Prof. Federico Della Croce di Dojola

Candidato:
Giovanni Genna

Indice:

1

Introduzione

1.1 Cos'è la Ricerca Operativa[7]

Per poter meglio analizzare e comprendere il problema preso in considerazione, è utile capire cosa sia la Ricerca Operativa, quando e perché questa disciplina sia nata.
La Ricerca Operativa è nata durante la II guerra mondiale, in quanto era necessario per le forze alleate risolvere complessi problemi tattici e strategici; furono organizzati, perciò, gruppi di scienziati, ingegneri, matematici, economisti, etc. che affrontarono problemi quali l'individuazione dei sottomarini nemici o il controllo del fuoco della contraerea.
La attività di questi scienziati venne chiamata *Operational Research* in Gran Bretagna e, successivamente, *Operations Research* in USA (quindi, ricerca delle operazioni più opportune per arrivare ad un certo obbiettivo).
Terminato il conflitto mondiale, si osservò che l'approccio messo a punto fino ad allora per scopi puramente militari potesse essere utilizzato per risolvere problemi di natura civile. Alcuni scienziati, tornati alle loro università, svilupparono un fondamento teorico delle tecniche precedentemente utilizzate in maniera empirica; altri si dedicarono alla creazione di nuove tecniche di Ricerca Operativa.
Le prime applicazioni civili di Ricerca Operativa furono nell'ambito della pianificazione della produzione dell'industria del petrolio.
Oggigiorno, banche, industrie, ospedali, società di trasporto, supermercati, etc. riconoscono la necessità di utilizzare la Ricerca Operativa per aumentare l'esigenza e l'efficacia nella fornitura dei propri servizi.

Lo sviluppo e l'affermarsi della Ricerca Operativa sono stati e sono tuttora aiutati fortemente dalla disponibilità di computer sempre più potenti e veloci. Dal suo canto, la Ricerca Operativa ha contribuito e continua tutt'oggi a contribuire allo sviluppo delle discipline informatiche (si pensi, ad esempio, alla progettazione di reti di telecomunicazione).

Non esiste una definizione univoca di Ricerca Operativa. L'Associazione nazionale inglese di Ricerca Operativa (*OR Society*) dà la seguente definizione:
"Ricerca Operativa è l'applicazione di metodi scientifici per la soluzione di problemi complessi che nascono nella direzione e gestione di grandi sistemi di uomini, macchine, materiali e denaro nell'industria, affari, amministrazione e difesa. L'approccio caratteristico della Ricerca Operativa è lo sviluppo di un modello matematico del sistema allo studio, che incorpora la misurazione di fattori di casualità e di rischio, mediante il quale predire e confrontare i risultati di decisioni, strategie e controlli alternativi. Lo scopo consiste nello aiutare i decisori a determinare scientificamente la loro politica e le loro azioni".

1.2 Assetto organizzativo della Tesi

Il problema considerato in questa Tesi di Laurea Triennale è un problema di schedulazione classica su macchine parallele identiche, con conflitti tra i job (*parallel machine scheduling with conflicts*).
La tesi è organizzata nel seguente modo: il Capitolo 2 introduce il problema e le sue peculiarità, analizzando il Modello di Programmazione Lineare e, alla fine, disaminando il Modello di Programmazione Lineare sviluppato sul solver *FICO@Xpress* . Il Capitolo 3 verte sull'approccio risolutivo adottato e ne spiega dettagliatamente i vari passi. Il Capitolo 4 analizza i risultati computazionali ottenuti dai testing, riportando tabelle che ne dettagliano gli esiti. Il Capitolo 5 è un dedicato ad alcune osservazioni finali. Il Capitolo 6 è una Appendice, in cui è presente tutto il codice per implementare il Modello di Programmazione Lineare, le euristiche e la Mat-euristica. Il Capitolo 7 cita la bibliografia di riferimento.

2

Il Problema: *schedulazione su macchine parallele con conflitti*

2.1 Descrizione del problema

2.1.1 Notazione preliminare

La *schedulazione*[1] è l'allocazione di risorse nel tempo per l'esecuzione di compiti.
I problemi di schedulazione sono classificati con la *notazione di Graham: α/β/γ*, dove:
-*α* indica il sistema e il numero di macchine;
-*β* indica eventuali caratteristiche particolari;
-*γ* indica la funzione obbiettivo.
Ipotesi generali: la stessa macchina non può effettuare più di un'operazione per volta; ogni job deve essere lavorato da una macchina per volta; le precedenze devono essere rispettate;
Ipotesi semplificative: tutti i job sono disponibili all' inizio del problema; i tempi sono deterministici; i tempi di processamento non dipendono dalla sequenza; le macchine lavorano un unico job per volta; ogni job visita una risorsa al più una volta; non è possibile interrompere la lavorazione di un job; non vi è cancellazione di job; i lotti sono indivisibili; è possibile avere Work-In-Process (WIP).

Un *grafo*[10] *G=(N,E)* è una coppia ordinata di un insieme finito non-vuoto *N* di nodi e un insieme di spigoli (Edges) *E*. Ad ogni elemento *e* di *E* sono associati due elementi di *N*, detti "estremi" di *e*.

Un *problema*[9] può essere visto come una questione generale da risolvere, caratterizzata da diversi parametri i cui valori non sono specificati.
Si ha un'*istanza* di un problema quando vengono specificati dei valori per i parametri del problema.
Un *algoritmo* è una procedura passo-passo che porta alla soluzione del problema.
La *funzione di complessità temporale* (tempo di esecuzione) di un algoritmo esprime i suoi requisiti di tempo ponendo, per ogni possibile dimensione dell'istanza, la quantità di tempo necessaria all'algoritmo per risolvere tale istanza.
Sia *f* la funzione di complessità temporale associata ad un algoritmo, *n* la dimensione dell'istanza di un problema e $g: N \rightarrow N$. Un algoritmo caratterizzato da una funzione di complessità temporale che risulta essere $O(g(n))$, con $g(n)$ funzione polinomiale di *n* (come ad esempio n, n^3, n^{10}, etc.), si dice *algoritmo polinomiale*. Non sono polinomiali, invece, quegli algoritmi dove $g(n)= 2^n$ oppure $n!$.
La *classe P* è l'insieme di tutti i problemi risolubili nel caso peggiore da un algoritmo polinomiale.
Si definisce la *classe NP* l'insieme di tutti i problemi di decisione risolubili nel caso peggiore mediante un algoritmo non-deterministico polinomiale, ossia problemi le cui soluzioni possono essere verificate in un tempo polinomiale.

Un problema *w'* è *polinomialmente riconducibile* ad un altro problema *w* se per ogni istanza di *w'* si può costruire in tempo polinomiale un'istanza di *w* tale che dalla soluzione dell'istanza considerata di *w* si determina immediatamente la soluzione ottima per l'istanza di *w'*.
Si consideri un problema appartenente alla classe *NP*, il *Problema della Soddisfacibilità* (*SAT*), che può essere così enunciato: "Data un'espressione booleana di *n* variabili binarie $x_1, x_2, \ldots, x_n$, esiste una soluzione $x_1^*, x_2^*, \ldots, x_n^*$ che rende l'espressione booleana vera?"
La classe dei problemi *NP-completi* è l'insieme di tutti i problemi che verificano le seguenti due condizioni:
1. *w* appartiene alla classe *NP*;
2. *SAT* è *polinomialmente riconducibile* a *w*.
Si consideri un problema *w* di cui non è possibile verificare l'appartenenza alla classe *NP*, ma che soddisfi la seconda condizione sopracitata. Il problema *w* è quindi almeno difficile quanto un problema *NP-completo*, ma non è possibile garantire la sua appartenenza a *NP*. Problemi di questo tipo prendono il nome di problemi *NP-hard*; sono detti anche *problemi "difficili"*. Esempi di problemi *NP-hard* sono i problemi di ottimizzazione le cui versioni decisionali sono *NP-complete*.

2.1.2 Il problema considerato[7]

Come già preannunciato, il problema analizzato è un problema di schedulazione.
Nello specifico, esso è una estensione del problema di schedulazione su macchine parallele identiche; in più, anche un grafo è parte dell'input: ogni nodo del grafo rappresenta un job e un arco implica che i due jobs siano in conflitto tra loro, quindi, che essi non possano essere schedulati sulla stessa macchina.
L' obbiettivo è trovare l'assegnazione dei jobs nelle macchine tale che il massimo tempo di completamento (*makespan,* C_{max}) sia minimizzato.

I jobs sono raggruppati in un insieme $J= \{1, \ldots, n\}$ di jobs e devono essere schedulati in *m* macchine identiche, senza preferenze, in modo da minimizzare il *makespan*. Ad ogni job è associato un tempo di lavorazione o processamento (*processing time*) $p_j \in No$ e ogni job deve essere assegnato a una sola macchina.
Le macchine sono riunite in un insieme $M= \{1, \ldots, m\}$ e ogni macchina può processare al massimo un job per volta.
Un grafo $G= (J, E)$, detto *grafo dei conflitti* (*conflicts graph*) costituisce un dato di input fondamentale del problema: se $\{j, j'\} \in E$, allora *j* e *j'* sono *conflicting jobs* e devono necessariamente essere schedulati su macchine diverse.
Il problema risultante è il *problema di schedulazione su macchine parallele con conflitti (parallel machine scheduling with conflicts - PMC*).

Questo problema è *NP-hard* perché è la fusione di due problemi *NP-hard*: contiene sia il problema $P_m//C_{max}$, quindi il classico problema di schedulazione su macchine parallele (P_m) (senza conflitti), sia il *vertex coloring problem* (una schedulazione ammissibile esiste se e solo se il *grafo dei conflitti* può essere *colorato* con al massimo *m colori*).

Si assumerà sempre $m<n$ per evitare soluzioni banali.

2.2 Il modello di Programmazione Lineare

2.2.1 Cos'è un modello di Programmazione Lineare[8]

Un modello è una rappresentazione semplificata di un sistema reale, solitamente complesso. Quando tale rappresentazione è effettuata mediante equazioni, disequazioni e funzioni matematiche, prende il nome di *modello matematico*.

I modelli matematici di problemi di ottimizzazione di tipo lineare (continuo e intero) vengono chiamati *modelli di Programmazione Lineare (PL)*.
Un *modello PL* consiste in:

- una funzione obiettivo, rappresentata da un'espressione lineare delle incognite del problema da massimizzare o minimizzare;
- un sistema di vincoli, che rappresenta geometricamente la regione di ammissibilità del problema ed è costituito da un insieme di disequazioni lineari, che le incognite devono rispettare.

Un generico modello di programmazione matematica è composto da tre elementi fondamentali:

- Variabili decisionali ($x_1, x_2, \ldots, x_n$): sono le entità di cui si deve conoscere il valore per poter verificare ammissibilità e costo di una qualunque soluzione del problema in esame.
- Vincoli: sono relazioni matematiche che descrivono l'insieme dove le variabili decisionali possono assumere dei valori ammissibili del problema.
- Funzione obiettivo: è una funzione f delle variabili decisionali $x_1, x_2, \ldots, x_n$.

Nel caso di modelli di *PL* si assume che i vincoli e la funzione obiettivo siano lineari.

Una soluzione ammissibile[11] è ogni n-upla che verifica le soluzioni del problema e quindi appartiene al cosiddetto *campo di scelta*. In *PL,* esse fanno parte di una regione convessa che, se è finita, ha un contorno poligonale. Si distingue perciò tra:

- *soluzioni di base*: ogni soluzione che deriva dall'intersecarsi a due a due delle equazioni che delimitano l'area dei vincoli;
- *soluzioni ammissibili*: ogni soluzione che appartiene al campo di scelta;
- *soluzioni ammissibili di base*: ogni soluzione che si trova ai vertici della regione, aperta o chiusa, che costituisce il campo di scelta. Tra queste, si trova la *soluzione ottima*;

Un problema di programmazione matematica può in generale essere formulato nel seguente modo:

$$\min_{\bar{x} \in X} f(\bar{x})$$

dove $X \subseteq R^n$ è l'insieme delle soluzioni ammissibili del problema e $f: X \rightarrow R$ è la funzione obiettivo. Quando sia $f(\bar{x})$ che $\bar{x} \in X$ sono lineari si ha un problema di Programmazione Lineare (*PL*), che diventa un problema di Programmazione Lineare Intera (*PLI*) se tutte le variabili sono intere e Misto-Intera (*PLMI*) se alcune variabili sono intere e altre continue.

Un problema di ottimizzazione combinatoria può essere definito ne seguente modo:

- sia X un set finito di punti;
- sia $\Omega \subseteq X$ lo spazio delle soluzioni ammissibili definito dai vincoli del problema
- sia F una funzione obiettivo che assegna un valore reale ad ogni $s \in X$, cioè $F: X \rightarrow R$.

Si consideri un problema di minimizzazione. Si vuole determinare una soluzione ammissibile $s^* \in \Omega$ tale che $F(s^*) \leq F(s), \forall s^* \in \Omega$. In altre parole, si vuole minimizzare $F(s)$ con il vincolo che $s \in \Omega$. Alla gran parte dei problemi di Ottimizzazione Combinatoria è associabile un modello di *PLI* o *PLMI*.

2.2.2 Il modello di Programmazione Lineare di *PMC*[6]

Per il *parallel machine scheduling with conflicts* è possibile formulare un modello di Programmazione Lineare Misto-Intera (*PLMI)*.
Per ogni job $j \in J$ e macchina $i \in M$, si introduce una variabile binaria x_{ij} che è uguale a *1,* se il job j è assegnato alla macchina i, a *0* altrimenti. Si introduce anche una variabile y che sarà uguale al *makespan* risultante dalla schedulazione.

Un possibile modello per il *PMC* è il seguente:

$min(y)$ *(1a)*

s.t.

$\sum_{i \in M} xij \leq 1 \quad , \quad \forall j \in J$ *(1b)*

$x_{ij} + x_{ij'} \leq 1 \quad , \quad \forall \{j, j'\} \in E, \forall i \in M$ *(1c)*

$\sum_{j \in J} pj \; xij \leq y \quad , \quad \forall i \in M$ *(1d)*

$x_{ij} \in \{0,1\} \quad , \quad \forall j \in J, \; \forall i \in M$ *(1e)*

$y \geq 0$ *(1f)*

(1a) rappresenta la funzione obiettivo del modello: la minimizzazione del *makespan*.
s.t. indica l'inizio dell'elencazione dei vincoli (dall'inglese *subject to*).

(1b) è un set di vincoli che garantisce l'assegnazione di ogni job ad esattamente una macchina.
La disuguaglianza *(1c)* forza i job in conflitto (estremi di un arco del *grafo dei conflitti*) ad essere schedulati su macchine diverse.
I vincoli *(1d)* impongono che per ogni macchina il *makespan y* sia almeno uguale al tempo di processamento totale su quella macchina.
Il vincolo *(1e)* rappresenta le variabili come binarie.
Infine, il vincolo *(1f)* garantisce, essendo il *makespan* un tempo di completamento, che esso sia maggiore o uguale a zero.

2.2.3 Introduzione a *FICO@Xpress* e al *Branch & Bound*

Il modello di Programmazione per il problema *PMC* descritto nel Sottoparagrafo 2.2.2 è stato implementato utilizzando *FICO@Xpress,* un software commerciale che funge da solver per problemi di Programmazione Lineare. Tale software utilizza come metodo risolutivo l'algoritmo del *Branch & Bound*.

Il metodo del *Branch & Bound*[8,4] fa parte dei *metodi di enumerazione implicita,* caratterizzabili generalmente da uno schema ad albero, che conserva la traccia della enumerazione, e da una procedura di valutazione del nodo che mira ad evidenziare quei rami che non potranno mai condurre alla soluzione ottima cercata. In pratica, questi metodi, pur di tipo enumerativo, riescono ad eliminare interi sottoinsiemi di soluzioni ammissibili, evitando così di dover enumerare esplicitamente tutte le soluzioni.
Il metodo del *Branch & Bound (B&B)* decompone l'insieme delle soluzioni ammissibili del problema di partenza in sottoinsiemi via via più ristretti. Tale processo è detto *processo di branch*: esso, applicato ricorsivamente ad ogni sottoinsieme, costruisce una struttura ad albero, detta *albero delle decisioni* o *albero di branch*, in cui i nodi sono i sottoinsiemi e gli archi sono i *branch* effettuati. Il nodo radice rappresenta il problema di cui si vuole trovare la soluzione; dato un problema, i sottoinsiemi derivanti da un suo *branch* sono detti *nodi figli.*
Dato un nodo ed il problema ad esso associato, può essere calcolata una stima (inferiore, o *lower bound*, nel caso di un *problema di minimo*, quindi come nel problema *PMC,* o superiore o *upper bound*, nel caso di un problema di massimo) del valore ottimo della sua funzione obiettivo. Il processo di calcolo di tale stima è detto processo di rilassamento o *bounding*.
Due dei metodi più utilizzati per il calcolo del *bound* sono:

- il *rilassamento continuo*: si ignorano i vincoli di interezza delle variabili intere. Il problema che si ottiene è un normale problema di Programmazione Lineare (PL), risolvibile all'ottimo efficientemente con un metodo di risoluzione adeguato (ad esempio, il *Metodo del Simplesso*);
- il *rilassamento lagrangiano*: uno o più vincoli vengono tolti dalla formulazione e la loro eventuale violazione viene presa in considerazione come penalizzazione della funzione obiettivo del problema rilassato.

La dimensione dell'albero delle decisioni scresce esponenzialmente con l'aumentare del numero di variabili, per cui ben presto diventa troppo elevato per poter generare tali soluzioni in tempi computazionali ragionevoli. Per tale motivo, l'algoritmo del *B&B* affianca alla fase di *branch* una serie di regole che permettono di chiudere un nodo senza generare il

rispettivo sottoalbero. Tali regole sfruttano proprio il calcolo del *bound* e sono dette regole di chiusura del nodo (*fathoming*):

- chiusura per *inammissibilità;*
- chiusura per *ottimalità;*
- chiusura per *non migliorabilità;*

Oltre a questa struttura principale, numerose altre componenti sono presenti in un algoritmo di *B&B*:

- regola di esplorazione dell'albero: regola con cui, dati i nodi ancora non valutati, viene scelto il prossimo nodo a cui applicare le regole di fathoming e branch. Tra le principali: *Depth First* (visita in profondità); *Best First* (si valuta il nodo con *lower bound* minore per problemi di minimo, come nel caso del *PMC*, e *upper bound* maggiore per problemi di massimo).
- regola di *branch*: regola che genera il partizionamento in sottoinsiemi del problema. La più classica è la cosiddetta regola di *branch binario* associata al rilassamento continuo.

2.2.4 Il Modello di Programmazione Lineare del *PMC* su *FICO@Xpress*

La prima fase del lavoro di Tesi è stata lo studio del *MPL* del *PMC* e la conseguente scrittura di esso sul solver *FICO@Xpress* per poter testare per quali istanze il solver non fosse in grado di trovare l'ottimo entro un tempo prestabilito, che è stato fissato a 60s (per maggiori dettagli, si veda il Capitolo 3).
Seguiranno alcuni estratti di codice, per esplicare il *MPL* su *FICO@Xpress*.

In primis, sono stati forniti al solver i dati di input:

```
y: mpvar
macchine=1..20
jobs=1..200
x: array(macchine, jobs) of mpvar
p: array(jobs) of integer
coppiaConflitto: array(jobs, jobs) of boolean
alea: real
```

Una variabile dichiarata *mpvar* è una variabile decisionale (come y, quindi il *makespan* (*f.o.*), e x, quindi l'array di tutte le variabili binarie x_{ij}).
Nell'estratto di codice, a mero titolo esplicativo, sono stati dichiarati 20 macchine e 200 jobs.
p è l'array contenente tutti i process times dei rispettivi jobs.
coppiaConflitto è un array di booleani: se *coppiaConflitto(j, j')* è uguale a *true, j* e *j'* sono *conflicting jobs*.
alea è una variabile reale che rappresenta una densità di probabilità, nello specifico la densità di probabilità di conflitto tra i jobs considerati.

L'estratto sottostate indica che ogni job deve essere assegnato ad esattamente una macchina (equivalente al vincolo (1b)):

```
forall(j in jobs)do
sum (i in macchine) x(i,j)=1
end-do
```

L'estratto sottostante impone che, per ogni macchina, il *makespan y* sia almeno uguale al tempo di processamento totale di quella macchina (equivalente al vincolo (1d)):

```
forall(i in macchine)do
sum(j in jobs) p(j)*x(i,j)<=y
end-do
```

L'estratto che segue dichiara le variabili decisionali x_{ij} variabili binarie, come nel vincolo (1e):

```
forall(i in macchine, j in jobs) do
x(i,j) is_binary
end-do
```

Il codice sottostante crea per ogni coppia di jobs un numero casuale: se questo è minore o uguale di *alea* (che, in questo esempio, è stata dichiarata uguale a 0,10), i jobs della coppia corrente vengono considerati *conflicting jobs*. In questo modo, si ottiene una densità di probabilità di conflitto del 10%.
Nella seconda parte, si verifica che i conflicting jobs non siano posizionati sulla medesima macchina (in maniera equivalente al vincolo (1c)).

```
forall(j in jobs)do
forall(k in jobs)do
if(k>j)then
alea:=random
if(alea<=0.10)then
coppiaConflitto(j,k):=true
end-if
if(alea>0.10)then
coppiaConflitto(j,k):=false
end-if
end-if
end-do
end-do

forall(i in macchine)do
forall(j in jobs)do
```

```
forall(k in jobs)do
if(k>j) then
if(coppiaConflitto(j,k)=true)then
x(i,j)+x(i,k)<=1
end-if
end-if
end-do
end-do
end-do
```

Si impone, infine, che il *makespan* sia maggiore o uguale a zero (come nel vincolo (1f)) e si chiede di minimizzare tale variabile (funzione obiettivo):

```
y>=0

minimize(y)
```

3
L’ approccio risolutivo

Dopo l’analisi del *MPL* del *PMC* e la scrittura di esso su *FICO@Xpress,* come accennato precedentemente, il lavoro è proseguito con il testing del MPL sul solver, per verificare per quali istanze questo non trovasse l’ottimo entro un tempo massimo prestabilito, che è stato fissato a 60s.
Successivamente, sono state coniate, per mezzo del linguaggio di programmazione *Java*, sull’ambiente di sviluppo *Eclipse IDE,* quattro euristiche di ordinamento dei jobs sulle macchine, nella fattispecie delle regole di priorità statiche, e sono state testate per quelle istanze in cui il solver non fosse in grado di raggiungere l’ottimo entro il tempo massimo.
Infine, viene implementata una procedura mateuristica di ricerca locale applicata in cascata ad una prima soluzione, ottenuta mediante la “migliore” delle euristiche sviluppate (“migliore” in base a cento istanze considerate, sulle quali sono state testate tutte e quattro le euristiche). L'obiettivo di questa seconda fase è quello di trovare una soluzione che possa migliorare quella ottenuta con l’euristica e che sia, sperabilmente, migliore di quella ottenuta con il solver.

3.1 Istanze per le quali il solver non trovi l’ottimo entro il tempo massimo

Il tempo massimo è stato fissato a 60s, dopo una serie di prove di carattere sperimentale, avendo notato che fosse una soglia ragionevole per le istanze prese in considerazione.
Il solver è stato testato su istanze con numero di macchine, numero di job e densità di probabilità di conflitto via via crescenti, fino a riscontrare delle difficoltà a raggiungere l’ottimo entro il tempo prestabilito.
Nello specifico, per tutte le prove effettuate, sono stati scelti casualmente dei tempi di lavorazione p_j dei job coinvolti nel range [50, 100].
Il solver è stato lanciato per *m=5, 10, 15, 20, 25, 30; n= 20, 40, 60, 80, 100; densità di probabilità= 0.1, 0.2, 0.3, 0.4, 0.5*. Sono state considerate solo le istanze con $m \leq 3n$, per evitare soluzioni banali o troppo semplici. Per istanze con stesso *m*, stesso *n* e stessa *densità di probabilità*, sono state effettuate cinque esecuzioni.
Si può notare come la *densità di probabilità* non vada oltre lo *0.5*, in quanto, come è naturale intuire, maggiore è la probabilità di conflitto tra jobs, maggiore è la possibilità che non ci siano soluzioni ammissibili. Nei risultati computazionali riportati al Paragrafo 4.1, si potrà notare come già per alcune delle istanze considerate non esista nessuna soluzione ammissibile (sarà riportata la dicitura *y=0*).

3.2 Le procedure euristiche sviluppate

Una procedura euristica[12] è una procedura in grado di individuare soluzioni subottime senza garanzia di ottimalità in tempo limitato.

Conseguentemente all'individuazione delle istanze che mandassero "in crisi" il solver, sono state cercate delle euristiche, in particolare quattro, che potessero fornire delle soluzioni ammissibili, in un tempo inferiore al secondo, per quelle istanze per le quali il solver non trovasse l'ottimo entro il tempo massimo prestabilito.
Le euristiche sviluppate sono delle regole di priorità statiche: si crea un ordinamento, dando priorità ad un job piuttosto che ad un altro; in base a tale ordinamento, con un processo iterativo, ogni job viene posizionato sulla macchina ammissibile (dove non siano già stati schedulati job con il quale quello considerato è in conflitto), in cui la somma dei tempi di processamento dei job fino a quel momento processati su tale macchina sia minima
Si può intuire come queste regole di priorità, essendo statiche, forniscano un solo ordinamento ad ogni istanza; quindi, è possibile riscontrare che queste non trovino soluzione ammissibile perché, in base all'ordinamento e al conseguente posizionamento dei job precedenti, ad un certo punto, non ci sia più una macchina ammissibile per il job corrente.
L'implementazione delle quattro diverse euristiche è stata svolta mediante il linguaggio di programmazione *Java*, un linguaggio di programmazione ad alto livello, orientato agli oggetti e a tipizzazione statica, sull'ambiente di sviluppo *Eclipse IDE*.
Dopo lo sviluppo di tali, le euristiche sono state testate su cento istanze, per diversi valori di *m*, *n* e *densità di probabilità di conflitto*, al fine di scegliere la "migliore", in base ai risultati ottenuti.

3.2.1 La prima euristica

La prima euristica sistema i job in ordine non crescente di numero di conflitti del job stesso. A parità di numero di conflitti, viene considerato il process time p_j dei job: si sequenziano in ordine non crescente di p_j (come nella regola di priorità statica *LPT, Longest Process Time*). Segue un piccolo estratto di codice *Java*, nel quale attraverso l'utilizzo di un *comparatore*, si definisce l'ordinamento desiderato. Un altro metodo chiederà come input la lista di job ancora non ordinati e il comparatore, così effettuerà l'ordinamento desiderato.

```
import java.util.Comparator;
public class ComparatoreDiJobs implements Comparator<Job>{
	@Override
	public int compare(Job j1, Job j2) {

		if(j1.getConflitti().size()!=j2.getConflitti().size()) {
			return -Integer.compare(j1.getConflitti().size(),
					j2.getConflitti().size());
		}else {
			return -Integer.compare(j1.getP(), j2.getP());
		}
	}
}
```

3.2.2 La seconda euristica

La seconda euristica sistema i jobs in ordine non crescente di p_j (quindi, come nella regola di priorità statica *LPT, Longest Process Time*); a parità di p_j, i job vengono ordinati in base al numero di conflitti non crescente.
Segue un piccolo estratto di codice *Java*, nel quale attraverso l'utilizzo di un *comparatore*, si definisce l'ordinamento desiderato. Un altro metodo chiederà come input la lista di job ancora non ordinati e il comparatore, così effettuerà l'ordinamento desiderato.

```
import java.util.Comparator;
public class ComparatoreDiJobs2 implements Comparator<Job>{
	@Override
	public int compare(Job j1, Job j2) {
		if(j1.getP()!= j2.getP()) {
			return -Integer.compare(j1.getP(), j2.getP());
		}else {
			return -Integer.compare(j1.getConflitti().size(),
					j2.getConflitti().size());
		}
	}
}
```

3.2.3 La terza euristica

La terza euristica è una fusione delle due precedenti.
Il primo passo è calcolare la percentuale esatta di conflitti: numero totale di coppie di jobs in conflitto diviso numero totale di coppie di jobs ottenibili (il divisore si calcola come *coefficiente binomiale n su 2,* $\binom{n}{2}$).
Avendo la percentuale, che per comodità verrà indicata con *perc*, si può procedere con l'ordinamento: i primi *perc·n* job verranno ordinati mediante la *prima euristica*, i restanti, quindi *(1-perc)·n* job, verranno sequenziati per mezzo della *seconda euristica*.

3.2.4 La quarta euristica[3]

La quarta, ed ultima, euristica sviluppata effettua un ordinamento che può essere meglio chiarito elencando i diversi passaggi:

1. Si ordinano i job mediante la *seconda euristica*;
2. Si considerano $\left\lceil \frac{m}{n} \right\rceil$ tuple di job: *1, …, m; m+1, …, 2m,* etc.. Se *n* non è multiplo di *m*, si aggiungono dei job *fittizi* (con $p_j=0$ e numero di conflitti pari a zero) necessari a far diventare *n* un multiplo di *m*, e si allocano nell' ultima tupla;

3. Per ogni tupla, si calcola lo *SLACK* associato, cioè*:* $p_1 - p_m$, $p_{(m+1)} - p_{2m}$,..., $p_{(n-m+1)} - p_n$.

4. Si ordinano le tuple per *SLACK* associato non crescente; a parità di *SLACK* si sequenziano le tuple per numero di conflitti totale (la somma dei conflitti di tutti i job che costituiscono la tupla) non crescente.

3.2.5 Confronti tra le euristiche sviluppate

I fattori determinanti per la scelta dell'euristica "migliore" sono stati l'ottenimento di una soluzione ammissibile per l'istanza fornita come input e il *makespan* risultante dalla schedulazione ottenuta.
Come detto, è stato effettuato un testing consistente in cento istanze diverse fornite come input a tutte e quattro le euristiche. Sono stati prese in considerazione quelle istanze per le quali il solver non riuscisse ad ottenere l'ottimo entro il tempo massimo, spingendosi talvolta anche a input ancora più grandi (in numero di macchine m e numero di job n) per poter capire e riscontrare quando le regole di priorità statiche andassero in difficoltà, non riuscendo ad ottenere soluzioni ammissibili. Nello specifico, per tutte le prove effettuate, sono stati scelti dei p_j in modo casuale nel range [50, 100). Le euristiche sono state lanciate per m= 10, 20, 30, 50; n= 60, 80, 100, 150, 200, 300, 500; densità di *probabilità di conflitto*= 0.1, 0.2, 0.3, 0.4, 0.5. Sono state considerate istanze con $m \leq 3n$, per evitare soluzioni banali o troppo semplici. Per istanze con stesso m, stesso n e stessa *densità di probabilità*, sono state effettuate cinque esecuzioni.

Per maggiori informazioni sui risultati computazionali, si riporta al Paragrafo 4.2.

Di seguito, si riporta una tabella che riassume ed evidenzia l'esito delle prove:

Risultati ottenuti	Numero di volte in cui l'euristica ha ottenuto il *makespan* minore (migliore).	Numero di volte in cui l'euristica NON ha fornito una soluzione ammissibile.
Prima euristica	36	6
Seconda euristica	23	42
Terza euristica	24	45
Quarta euristica	14	34

L'euristica che è stata dichiarata "migliore" dai risultati computazionali effettuati è la **prima**, in quanto, come si può notare dalla tabella sovrastante, i posizionamenti da questa eseguiti hanno portato 94 volte su 100 a delle soluzioni ammissibili, mentre le altre

euristiche hanno tutte risultati ben peggiori (riescono a fornire soluzioni ammissibili solo tra il 55% e il 66% delle volte); inoltre, è stata anche la regola di priorità statica a fornire il maggior numero di soluzioni migliori rispetto alle soluzioni fornite dalle altre tre.

3.3 La mat-euristica

L'idea di fondo della *Mat-euristica*[2] è l'ibridizzazione, quindi la combinazione, di approcci per la risoluzione di problemi di ottimizzazione combinatoria; nella fattispecie, l'obiettivo è mettere insieme algoritmi esatti, che quindi permettono di ottenere la soluzione ottima di un problema, e algoritmi euristici.
Moltissimi problemi di produzione, distribuzione, etc., e tra questi anche *PMC*, sono problemi *NP-hard*. Di fronte a questi problemi, ci sono due alternative: cercare l'ottimo o approssimare, accontentandosi di una soluzione ammissibile, non troppo lontana dall'ottimo. Per la prima ipotesi, si ricorre metodi esatti, quali, ad esempio il *Branch & Bound*; ma per evitare che, nel peggiore dei casi, si debba enumerare l'intero spazio delle soluzioni, che causerebbe un tempo di esecuzione troppo lungo, si può ricorrere a metodi euristici, i quali hanno tempi di esecuzione polinomiali. La *Mat-euristica*, come già introdotto, cerca di unire le peculiarità di queste tecniche diverse, per prendere il meglio di entrambe.

Sono molteplici i metodi mat-euristici che, mediante la commistione di approcci esatti ed euristici, risolvono problemi *NP-hard*.

L'approccio mat-euristico scelto per la risoluzione del problema *PMC*, parte da una soluzione ammissibile generata dalla "migliore" delle euristiche sviluppate, che, come dichiarato in precedenza, risulta essere la prima, e applica iterativamente per un certo tempo il seguente approccio:

- "rilassa" alcune (poche) variabili decisionali della soluzione risultante dalla precedentemente esecuzione;
- applica l'algoritmo esatto, quindi lancia il solver di PL, che dovrà fissare soltanto le variabili "rilassate" al passo precedente;
-

L'obiettivo è, quindi, ottenere un sottoinsieme piccolo del problema, al quale applicare un algoritmo esatto e fare ciò un numero di volte tale che il tempo di esecuzione totale non superi il tempo massimo scelto, consentito per l'esecuzione della mat-euristica.

In particolare, è stato scritto un programma su *FICO@Xpress* per studiare le prestazioni della mat-euristica. Tale programma, ogni volta che viene eseguito, legge un file contenente i valori delle variabili decisionali x_{ij} (alla prima esecuzione leggerà il risultato fornito dalla prima euristica), e "fissa" una percentuale molto alta (scelta valutando sperimentalmente la quantità più adatta alle istanze considerate) di variabili; la restante parte di variabili rimane "rilassata". Il solver, quindi, risolve un problema di dimensioni esigue (rispetto al problema di partenza), cioè applica il *Branch & Bound* soltanto alle variabili che sono state liberate, e sovrascrive sul file datogli in input risultati ottenuti. In questo modo, il programma può

essere eseguito il numero di volte desiderato (come detto, in base al tempo massimo si esecuzione scelto): ad ogni iterazione, il file su cui esso lavora sarà il file risultante dall'elaborazione precedente.

Sono stati effettuati dei confronti tra le soluzioni fornite dalla *mat-euristica* e le *Best Solution* fornite dal solver, considerando un tempo massimo di 60s per quasi la totalità delle istanze (per istanze molto grandi, sono stati considerati tempi più lunghi).
La scelta della percentuale di variabili decisionali x_{ij} da mantenere "fissa" è stata valutata sperimentalmente, sulla base del numero di macchine, del numero di job, della densità di probabilità di conflitto e della differenza tra la soluzione fornita dalla prima euristica e la *Best Solution ,* per cercare di ottenere un compromesso tra la grandezza del sottoproblema fornito al solver, proporzionale al tempo di esecuzione della singola iterazione, e la possibilità di miglioramento (se si rilassa una quantità troppo esigua di variabili decisionali, è difficile che il solver abbia la possibilità di trovare una soluzione migliore). In generale, si è notato che all'aumentare del numero di job e del numero di macchine era necessario "fissare" una percentuale maggiore di variabili decisionali, per evitare che anche i sottoproblemi diventassero troppo grandi per poter consentire al solver di fare più iterazioni nel tempo prestabilito. Per le istanze considerate nel testing, le percentuali di variabili "fissate" oscillano tra 0.73 e 0.99.
È stato valutato il risultato di cinquanta istanze diverse per il confronto tra il solver e la mat-euristica.

Per maggiori informazioni sui risultati computazionali, si riporta al Paragrafo 4.3.

Di seguito, si riporta una tabella che riassume ed evidenzia l'esito delle prove:

	Numero di "vittorie"	Numero di "pareggi"
Best Solution (60s)	*14*	*4*
Mat-euristica (60s)	*32*	*4*

Si può notare come per il 64% delle prove la mat-euristica abbia fornito una soluzione migliore della *Best Solution* fornita dal solver; per l'8% delle volte abbiano fornito la stessa soluzione; e per il 28% delle prove, la *Best Solution* abbia avuto la meglio sulla mat-euristica.

Per un'analisi complessiva finale sui risultati ottenuti, si riporta al Capitolo 5.

4
I risultati computazionali

Il modello di Programmazione Lineare, le euristiche sviluppate e la mat-euristica sono stati fatti girare, come già noto, su *FICO@Xpress* e su *Eclipse IDE*, nel computer personale con processore *Intel(R) Core(TM) i7-8550U CPU @ 1.80GHz 1.99 GHz* e RAM da 8,00 GB.

Sono stati effettuati dei test e, quindi, analizzati i risultati computazionali ottenuti per:
- capire per quali istanze il solver di *PL* andasse "in crisi", cioè non fosse in grado di trovare la soluzione ottima entro il tempo massimo prestabilito;
- confrontare le euristiche sviluppate per scegliere quale fosse la migliore, cioè quale ottenesse il maggior numero di soluzioni ammissibili, e quante volte le soluzioni ottenute fossero migliori rispetto alle soluzioni risultanti dagli altri algoritmi;
- misurare le prestazioni della mat-euristica sviluppata, mettendo le sue soluzioni a paragone con ottenute dal solver.

4.1 Testing del Modello di Programmazione Lineare sul solver

Il tempo massimo è stato fissato a 60s, dopo una serie di prove di carattere sperimentale, avendo notato che fosse una soglia ragionevole per le istanze prese in considerazione.
Per istanze con stesso *m*, stesso *n* e stessa *densità di probabilità*, sono state effettuate cinque esecuzioni.

Nella tabella riportata sotto, l'elemento di studio principale è il tempo; per questo non si riporta il valore del *makespan* ottenuto, ma soltanto il tempo di esecuzione per trovare la soluzione ottima. Qualora questa non fosse stata trovata nel tempo utile, si evidenzierà l'evento con *MAXTIME*. Se il problema analizzato non ammetteva nessuna soluzione ammissibile, l'evento si marcherà con *y=0*.

Numero di macchine *m*	**Numero di jobs *n***	**Densità di probabilità di conflitto**	**Tempo prima Istanza(s)**	**Tempo seconda Istanza(s)**	**Tempo terza Istanza(s)**	**Tempo quarta Istanza(s)**	**Tempo quinta Istanza(s)**
5	*20*	*0.1*	*0.801*	*0.309*	*0.09*	*0.117*	*0.042*
5	*40*	*0.1*	*4.138*	*0.489*	*1.373*	*1.36*	*0.461*
5	*60*	*0.1*	*21.205*	*2.454*	*0.249*	*2.045*	*2.632*
5	*80*	*0.1*	*1.692*	*7.35*	*6.582*	*5.65*	*6.34*

5	100	0.1	9.153	MAXTIME	13.622	0.532	13.118
5	20	0.2	0.584	1.573	0.222	0.371	0,333
5	40	0.2	MAXTIME	19.467	6.022	MAXTIME	15.797
5	60	0.2	MAXTIME	MAXTIME	MAXTIME	34.533	MAXTIME
5	80	0.2	24.695	MAXTIME	MAXTIME	MAXTIME	MAXTIME
5	100	0.2	14.76	MAXTIME	MAXTIME	MAXTIME	MAXTIME
5	20	0.3	5.836	2.103	0.16	1.362	0.585
5	40	0.3	6.645	5.979	15.095	15.216	3.303
5	60	0.3	12.907	0.45	1.341	y=0	10.253
5	80	0.3	y=0	0.978	y=0	y=0	2.312
5	100	0.3	y=0	y=0	y=0	y=0	y=0
5	20	0.4	0.552	4.901	3.064	0.958	1.977
5	40	0.4	y=0	y=0	0,299	y=0	y=0
5	60	0.4	y=0	y=0	y=0	y=0	y=0
5	80	0.4	y=0	y=0	y=0	y=0	y=0
5	100	0.4	y=0	y=0	y=0	y=0	y=0
5	20	0.5	y=0	y=0	y=0	y=0	y=0
5	40	0.5	y=0	y=0	y=0	y=0	y=0
5	60	0.5	y=0	y=0	y=0	y=0	y=0
5	80	0.5	y=0	y=0	y=0	y=0	y=0
5	100	0.5	y=0	y=0	y=0	y=0	y=0
10	40	0.1	17.185	2.709	3.487	2.354	3.147
10	60	0.1	4.332	2.379	9.176	1.078	17.92
10	80	0.1	8.268	24.203	5.56	17.107	17.117
10	100	0.1	5.644	4.263	3.674	8.525	16.692
10	40	0.2	8.418	14.34	21.866	MAXTIME	MAXTIME
10	60	0.2	MAXTIME	MAXTIME	24.658	MAXTIME	MAXTIME
10	80	0.2	MAXTIME	MAXTIME	MAXTIME	MAXTIME	MAXTIME
10	100	0.2	MAXTIME	MAXTIME	MAXTIME	MAXTIME	MAXTIME
10	40	0.3	MAXTIME	MAXTIME	MAXTIME	MAXTIME	MAXTIME
10	60	0.3	MAXTIME	MAXTIME	MAXTIME	MAXTIME	MAXTIME
10	80	0.3	MAXTIME	MAXTIME	MAXTIME	MAXTIME	MAXTIME
10	100	0.3	MAXTIME	MAXTIME	MAXTIME	MAXTIME	MAXTIME
10	40	0.4	MAXTIME	MAXTIME	MAXTIME	MAXTIME	MAXTIME
10	60	0.4	MAXTIME	MAXTIME	MAXTIME	MAXTIME	MAXTIME
10	80	0.4	MAXTIME	MAXTIME	MAXTIME	MAXTIME	MAXTIME
10	100	0.4	MAXTIME	MAXTIME	MAXTIME	MAXTIME	MAXTIME
10	40	0.5	MAXTIME	MAXTIME	MAXTIME	MAXTIME	MAXTIME
10	60	0.5	MAXTIME	MAXTIME	MAXTIME	MAXTIME	MAXTIME
10	80	0.5	MAXTIME	MAXTIME	MAXTIME	MAXTIME	MAXTIME
10	100	0.5	MAXTIME	MAXTIME	MAXTIME	MAXTIME	MAXTIME
15	60	0.1	MAXTIME	20.483	MAXTIME	4.1	14.883
15	80	0.1	6.244	0.702	22.909	54.874	37.795
15	100	0.1	37.041	MAXTIME	MAXTIME	36.447	MAXTIME

15	60	0.2	MAXTIME	31.788	MAXTIME	MAXTIME	11.598
15	80	0.2	MAXTIME	MAXTIME	MAXTIME	MAXTIME	MAXTIME
15	100	0.2	MAXTIME	MAXTIME	MAXTIME	MAXTIME	MAXTIME
15	60	0.3	MAXTIME	MAXTIME	MAXTIME	MAXTIME	MAXTIME
15	80	0.3	MAXTIME	MAXTIME	MAXTIME	MAXTIME	MAXTIME
15	100	0.3	MAXTIME	MAXTIME	MAXTIME	MAXTIME	MAXTIME
15	60	0.4	MAXTIME	MAXTIME	MAXTIME	MAXTIME	MAXTIME
15	80	0.4	MAXTIME	MAXTIME	MAXTIME	MAXTIME	MAXTIME
15	100	0.4	MAXTIME	MAXTIME	MAXTIME	MAXTIME	MAXTIME
15	60	0.5	MAXTIME	MAXTIME	MAXTIME	MAXTIME	MAXTIME
15	80	0.5	MAXTIME	MAXTIME	MAXTIME	MAXTIME	MAXTIME
15	100	0.5	MAXTIME	MAXTIME	MAXTIME	MAXTIME	MAXTIME
20	60	0.1	11.582	MAXTIME	MAXTIME	MAXTIME	MAXTIME
20	80	0.1	12.806	8.892	24.841	MAXTIME	MAXTIME
20	100	0.1	MAXTIME	MAXTIME	MAXTIME	MAXTIME	MAXTIME
20	60	0.2	MAXTIME	MAXTIME	MAXTIME	MAXTIME	MAXTIME
20	80	0.2	MAXTIME	MAXTIME	MAXTIME	MAXTIME	MAXTIME
20	100	0.2	MAXTIME	MAXTIME	MAXTIME	MAXTIME	MAXTIME
20	60	0.3	MAXTIME	MAXTIME	MAXTIME	MAXTIME	MAXTIME
20	80	0.3	MAXTIME	MAXTIME	MAXTIME	MAXTIME	MAXTIME
20	100	0.3	MAXTIME	MAXTIME	MAXTIME	MAXTIME	MAXTIME
20	60	0.4	MAXTIME	MAXTIME	MAXTIME	MAXTIME	MAXTIME
20	80	0.4	MAXTIME	MAXTIME	MAXTIME	MAXTIME	MAXTIME
20	100	0.4	MAXTIME	MAXTIME	MAXTIME	MAXTIME	MAXTIME
20	60	0.5	MAXTIME	MAXTIME	MAXTIME	MAXTIME	MAXTIME
20	80	0.5	MAXTIME	MAXTIME	MAXTIME	MAXTIME	MAXTIME
20	100	0.5	MAXTIME	MAXTIME	MAXTIME	MAXTIME	MAXTIME
25	80	0.1	MAXTIME	MAXTIME	MAXTIME	MAXTIME	MAXTIME
25	100	0.1	MAXTIME	MAXTIME	MAXTIME	MAXTIME	MAXTIME
25	80	0.2	MAXTIME	MAXTIME	MAXTIME	MAXTIME	MAXTIME
25	100	0.2	MAXTIME	MAXTIME	MAXTIME	MAXTIME	MAXTIME
25	80	0.3	MAXTIME	MAXTIME	MAXTIME	MAXTIME	MAXTIME
25	100	0.3	MAXTIME	MAXTIME	MAXTIME	MAXTIME	MAXTIME
25	80	0.4	MAXTIME	MAXTIME	MAXTIME	MAXTIME	MAXTIME
25	100	0.4	MAXTIME	MAXTIME	MAXTIME	MAXTIME	MAXTIME
25	80	0.5	MAXTIME	MAXTIME	MAXTIME	MAXTIME	MAXTIME
25	100	0.5	MAXTIME	MAXTIME	MAXTIME	MAXTIME	MAXTIME
30	100	0.1	MAXTIME	MAXTIME	MAXTIME	MAXTIME	MAXTIME
30	100	0.2	MAXTIME	MAXTIME	MAXTIME	MAXTIME	MAXTIME
30	100	0.3	MAXTIME	MAXTIME	MAXTIME	MAXTIME	MAXTIME
30	100	0.4	MAXTIME	MAXTIME	MAXTIME	MAXTIME	MAXTIME
30	100	0.5	MAXTIME	MAXTIME	MAXTIME	MAXTIME	MAXTIME

Si può notare che, per la maggior parte delle istanze considerate, il solver non sia riuscito ad ottenere una soluzione ottima entro il tempo massimo fissato.
Per istanze contenenti 5 macchine, da 60 a 100 jobs e densità di probabilità da 0.3 a 0.5, nella quasi totalità dei casi, il solver ha constatato che non esistesse alcuna soluzione ammissibile ($y=0$). Infatti, come già accennato in precedenza, maggiore è la probabilità di conflitto tra jobs, maggiore è la possibilità che non ci siano soluzioni ammissibili. Per questo motivo, non sono state analizzate istanze, né qui né per i testing su euristiche e mat-euristica, con densità di probabilità maggiore di 0.5.

4.2 Testing per il confronto tra le euristiche implementate

Avendo a disposizione le istanze per le quali il solver non sia riuscito ad ottenere soluzione ottima entro il tempo massimo stabilito, come detto, si è pensato di trovare e implementare quattro euristiche, per ordinare e schedulare in poco tempo i job.
È stato effettuato un testing consistente in cento istanze diverse fornite come input a tutte e quattro le euristiche. Sono stati prese in considerazione quelle istanze per le quali il solver non riuscisse ad ottenere l'ottimo entro il tempo massimo, spingendosi talvolta anche a input ancora più grandi (in numero di macchine m e numero di job n) per poter capire e riscontrare quando le regole di priorità statiche andassero in difficoltà, non riuscendo ad ottenere soluzioni ammissibili.
Nelle tabelle riportate sotto, l'oggetto di studio è la capacità degli algoritmi di ritornare una soluzione ammissibile o meno. Quindi, si marca l'attenzione sui *makespan* ottenuti (qualora gli algoritmi siano in grado di trovare soluzione ammissibile; in caso contrario comparirà la dicitura *NA*). Per avere un'idea più concreta di quanto i risultati ottenuti si discostino dall'ottimo, oltre ai valori ottenuti dalle euristiche, si elencano la *best solution* e il *best bound* ottenuti dal solver in 10s.
Non sono presenti i tempi di esecuzione delle istanze, in quanto basti sapere che, per tutte le istanze prese in considerazione, l'ordine di grandezza dei tali è nel range $[10^{-3}, 10^{-1}]$ secondi.
La prima, la seconda, la terza, la quarta euristica vengono, nelle tabelle, identificate, rispettivamente, come *Euristica1, Euristica2, Euristica3, Euristica4.* La densità di probabilità come *prob*.

m=10					
n=80					
prob=0.2	*Istanza 1*	*Istanza 2*	*Istanza 3*	*Istanza 4*	*Istanza 5*
Euristica 1	*652*	*653*	*629*	*661*	*620*
Euristica 2	*NA*	*NA*	*637*	*NA*	*NA*
Euristica 3	*NA*	*NA*	*651*	*NA*	*NA*
Euristica 4	*NA*	*NA*	*NA*	*NA*	*639*
Best Sol (10s)	*610*	*606*	*597*	*616*	*585*
Best Bound (10s)	*603*	*601*	*590*	*609*	*580*

m=10					
n=60					
prob=0.3	*Istanza 1*	*Istanza 2*	*Istanza 3*	*Istanza 4*	*Istanza 5*
Euristica 1	*518*	*498*	*NA*	*NA*	*490*
Euristica 2	*NA*	*NA*	*NA*	*507*	*NA*
Euristica 3	*NA*	*NA*	*NA*	*NA*	*NA*
Euristica 4	*NA*	*NA*	*NA*	*NA*	*NA*
Best Sol (10s)	*454*	*468*	*463*	*467*	*455*
Best Bound (10s)	*450*	*458*	*456*	*455*	*444*

m=20					
n=80					
prob=0.2	*Istanza 1*	*Istanza 2*	*Istanza 3*	*Istanza 4*	*Istanza 5*
Euristica 1	*352*	*343*	*356*	*340*	*347*
Euristica 2	*306*	*353*	*310*	*350*	*350*
Euristica 3	*338*	*350*	*351*	*340*	*308*
Euristica 4	*369*	*395*	*394*	*304*	*392*
Best Sol (10s)	*308*	*313*	*311*	*307*	*309*
Best Bound (10s)	*303*	*305*	*305*	*302*	*304*

m=20					
n=150					
Prob=0.2	*Istanza 1*	*Istanza 2*	*Istanza 3*	*Istanza 4*	*Istanza 5*
Euristica 1	*618*	*602*	*607*	*620*	*624*
Euristica 2	*596*	*NA*	*584*	*592*	*NA*
Euristica 3	*629*	*603*	*NA*	*NA*	*NA*
Euristica 4	*636*	*587*	*573*	*593*	*635*
Best Sol (10s)	*577*	*575*	*567*	*584*	*574*
Best Bound (10s)	*568*	*558*	*554*	*561*	*567*

m=20					
n=300					
Prob=0,1	*Istanza 1*	*Istanza 2*	*Istanza 3*	*Istanza 4*	*Istanza 5*
Euristica 1	*1171*	*1181*	*1165*	*1156*	*1186*
Euristica 2	*NA*	*NA*	*1175*	*1177*	*1170*
Euristica 3	*1166*	*1193*	*NA*	*1174*	*1146*
Euristica 4	*1170*	*1195*	*1134*	*1211*	*NA*
Best Sol (10s)	*1136*	*1152*	*1148*	*1149*	*1135*
Best Bound (10s)	*1117*	*1129*	*1125*	*1124*	*1121*

m=20					
n=100					
prob=0.3	*Istanza 1*	*Istanza 2*	*Istanza 3*	*Istanza 4*	*Istanza 5*
Euristica 1	*411*	*415*	*406*	*428*	*419*
Euristica 2	*427*	*NA*	*NA*	*423*	*426*
Euristica 3	*NA*	*386*	*NA*	*421*	*413*
Euristica 4	*NA*	*429*	*433*	*446*	*424*
Best Sol (10s)	*387*	*382*	*378*	*385*	*389*
Best Bound (10s)	*378*	*371*	*370*	*374*	*374*

m=20					
n=80					
Prob=0.5	*Istanza 1*	*Istanza 2*	*Istanza 3*	*Istanza 4*	*Istanza 5*
Euristica 1	*NA*	*NA*	*402*	*417*	*NA*
Euristica 2	*NA*	*NA*	*NA*	*NA*	*NA*
Euristica 3	*NA*	*NA*	*NA*	*NA*	*NA*
Euristica 4	*NA*	*NA*	*NA*	*NA*	*NA*
Best Sol (10s)	*327*	*322*	*345*	*310*	*328*
Best Bound (10s)	*312*	*298*	*333*	*297*	*301*

m=20					
n=400					
prob=0.1	*Istanza 1*	*Istanza 2*	*Istanza 3*	*Istanza 4*	*Istanza 5*
Euristica 1	*1590*	*1573*	*1593*	*1598*	*1558*
Euristica 2	*NA*	*NA*	*NA*	*NA*	*NA*
Euristica 3	*NA*	*NA*	*NA*	*NA*	*NA*
Euristica 4	*NA*	*NA*	*NA*	*NA*	*NA*
Best Sol (10s)	*2077*	*2709*	*2972*	*2896*	*2883*
Best Bound (10s)	*1507*	*0*	*0*	*0*	*0*

m=20					
n=200					
prob=0.2	*Istanza 1*	*Istanza 2*	*Istanza 3*	*Istanza 4*	*Istanza 5*
Euristica 1	*839*	*822*	*848*	*815*	*828*
Euristica 2	*NA*	*NA*	*NA*	*NA*	*NA*
Euristica 3	*NA*	*NA*	*NA*	*NA*	*NA*
Euristica 4	*NA*	*NA*	*NA*	*NA*	*NA*
Best Sol (10s)	*806*	*801*	*809*	*806*	*820*
Best Bound (10s)	*765*	*761*	*762*	*768*	*759*

m=30					
n=100					
prob=0.5	*Istanza 1*	*Istanza 2*	*Istanza 3*	*Istanza 4*	*Istanza 5*
Euristica 1	*309*	*311*	*299*	*313*	*322*
Euristica 2	*299*	*293*	*288*	*NA*	*300*
Euristica 3	*298*	*282*	*291*	*NA*	*307*
Euristica 4	*325*	*318*	*302*	*324*	*325*
Best Sol (10s)	*287*	*280*	*290*	*289*	*286*
Best Bound (10s)	*254*	*251*	*256*	*255*	*255*

m=30					
n=100					
prob=0.4	*Istanza 1*	*Istanza 2*	*Istanza 3*	*Istanza 4*	*Istanza 5*
Euristica 1	*288*	*308*	*288*	*296*	*300*
Euristica 2	*277*	*280*	*280*	*279*	*280*
Euristica 3	*279*	*290*	*290*	*291*	*288*
Euristica 4	*279*	*311*	*274*	*275*	*278*
Best Sol (10s)	*264*	*269*	*266*	*267*	*265*
Best Bound (10s)	*241*	*245*	*243*	*243*	*243*

m=30					
n=150					
prob=0.4	*Istanza 1*	*Istanza 2*	*Istanza 3*	*Istanza 4*	*Istanza 5*
Euristica 1	*NA*	*496*	*452*	*504*	*443*
Euristica 2	*NA*	*NA*	*NA*	*NA*	*435*
Euristica 3	*422*	*NA*	*NA*	*NA*	*NA*
Euristica 4	*NA*	*422*	*NA*	*NA*	*NA*
Best Sol (10s)	*405*	*413*	*441*	*422*	*434*
Best Bound (10s)	*367*	*368*	*383*	*374*	*382*

m=30					
n=150					
prob=0.3	*Istanza 1*	*Istanza 2*	*Istanza 3*	*Istanza 4*	*Istanza 5*
Euristica 1	*417*	*436*	*428*	*428*	*430*
Euristica 2	*NA*	*429*	*434*	*428*	*426*
Euristica 3	*NA*	*426*	*414*	*406*	*423*
Euristica 4	*452*	*459*	*443*	*452*	*465*
Best Sol (10s)	*403*	*419*	*410*	*419*	*424*
Best Bound (10s)	*378*	*383*	*377*	*379*	*383*

m=30					
n=200					
prob=0.3	*Istanza 1*	*Istanza 2*	*Istanza 3*	*Istanza 4*	*Istanza 5*
Euristica 1	*565*	*564*	*586*	*562*	*547*
Euristica 2	*576*	*574*	*NA*	*NA*	*NA*
Euristica 3	*NA*	*NA*	*NA*	*NA*	*NA*
Euristica 4	*NA*	*NA*	*520*	*654*	*528*
Best Sol (10s)	*1062*	*556*	*1119*	*1038*	*552*
Best Bound (10s)	*0*	*500*	*0*	*0*	*491*

m=30					
n=200					
prob=0.2	*Istanza 1*	*Istanza 2*	*Istanza 3*	*Istanza 4*	*Istanza 5*
Euristica 1	*539*	*568*	*547*	*552*	*551*
Euristica 2	*519*	*522*	*528*	*569*	*521*
Euristica 3	*530*	*530*	*535*	*525*	*528*
Euristica 4	*526*	*526*	*567*	*575*	*559*
Best Sol (10s)	*524*	*576*	*575*	*560*	*533*
Best Bound (10s)	*499*	*504*	*507*	*504*	*502*

m=30					
n=300					
prob=0.2	*Istanza 1*	*Istanza 2*	*Istanza 3*	*Istanza 4*	*Istanza 5*
Euristica 1	*812*	*784*	*812*	*798*	*769*
Euristica 2	*NA*	*NA*	*774*	*NA*	*NA*
Euristica 3	*NA*	*783*	*816*	*NA*	*NA*
Euristica 4	*848*	*NA*	*823*	*774*	*NA*
Best Sol (10s)	*1477*	*1573*	*1441*	*1688*	*1429*
Best Bound (10s)	*0*	*0*	*0*	*0*	*0*

m=30					
n=300					
prob=0.1	*Istanza 1*	*Istanza 2*	*Istanza 3*	*Istanza 4*	*Istanza 5*
Euristica 1	*772*	*784*	*778*	*775*	*797*
Euristica 2	*787*	*771*	*775*	*777*	*743*
Euristica 3	*732*	*728*	*772*	*776*	*790*
Euristica 4	*757*	*737*	*776*	*759*	*789*
Best Sol (10s)	*2228*	*805*	*770*	*2255*	*831*
Best Bound (10s)	*0*	*722*	*726*	*727*	*742*

m=30					
n=500					
prob=0.1	*Istanza 1*	*Istanza 2*	*Istanza 3*	*Istanza 4*	*Istanza 5*
Euristica 1	*1291*	*1295*	*1292*	*1272*	*1290*
Euristica 2	*1248*	*1246*	*1297*	*1249*	*1247*
Euristica 3	*NA*	*1278*	*1290*	*1271*	*1267*
Euristica 4	*1314*	*1293*	*1281*	*1285*	*1305*
Best Sol (10s)	*3026*	*2934*	*2940*	*3353*	*3075*
Best Bound (10s)	*0*	*0*	*0*	*0*	*0*

m=50					
n=150					
prob=0.5	*Istanza 1*	*Istanza 2*	*Istanza 3*	*Istanza 4*	*Istanza 5*
Euristica 1	*269*	*270*	*277*	*272*	*279*
Euristica 2	*271*	*275*	*271*	*271*	*270*
Euristica 3	*251*	*276*	*259*	*289*	*257*
Euristica 4	*315*	*273*	*276*	*274*	*272*
Best Sol (10s)	*677*	*654*	*634*	*730*	*686*
Best Bound (10s)	*0*	*0*	*0*	*0*	*0*

m=50					
n=300					
prob=0.3	*Istanza 1*	*Istanza 2*	*Istanza 3*	*Istanza 4*	*Istanza 5*
Euristica 1	*504*	*494*	*501*	*488*	*489*
Euristica 2	*495*	*489*	*491*	*493*	*491*
Euristica 3	*488*	*509*	*487*	*485*	*485*
Euristica 4	*498*	*489*	*491*	*493*	*498*
Best Sol (10s)	*930*	*1109*	*1126*	*1145*	*1108*
Best Bound (10s)	*0*	*0*	*0*	*0*	*0*

Come era stato preannunciato al Capitolo 3 (che riporta il sunto finale dei risultati elencati nelle tabelle sovrastanti), l'euristica che risulta migliore sulla base dei risultati ottenuti dalle esecuzioni effettuate con le istanze prese in considerazione è la **prima**.
Come è logico intuire, più grande è il problema in termini di *m* e *n* e maggiore è il numero di volte in cui, i risultati delle euristiche (se ottengono un risultato) sono migliori di quelli del solver (*best solution*).

4.3 Testing per valutare le prestazioni della *mateuristica*

Avendo attinto dal confronto tra le euristiche che la prima sia quella col il "comportamento" migliore, è stata scelta questa per fare da input per la mat-euristica.
Sotto vengono riportati i risultati dei cinquanta problemi presi in considerazione per effettuare il confronto tra le soluzioni fornita dal solver e quelle fornite dalla mat-euristica in 60s: l'oggetto di studio è quindi il *makespan*.
Per la mat-euristica, sono state effettuate tante più iterazioni quante il tempo massimo ne concedesse.
Si indica con *prob* la densità di probabilità di conflitto e con *perc* la percentuale di variabili decisionali x_{ij} "fissate" (come spiegato nel Capitolo 3, scelta in base a risultati sperimentali).
In pochi casi, per istanze molto grandi, per avere un confronto maggiormente significativo, è stato utilizzato un tempo massimo superiore a 60s (specificato in tabella).

m=10	
n=60	
prob=0.2	
BestSol(60s)	*450*
BestBound(60s)	*447*
Euristica 1	*508*
Mateuristica(60s)	*448*

perc=0.78

m=10	
n=60	
prob=0.3	
BestSol(60s)	*433*
BestBound(60s)	*431*
Euristica 1	*520*
Mateuristica(60s)	*436*

perc=0.73

m=20	
n=60	
prob=0.5	
BestSol(60s)	*227*
BestBound(60s)	*219*
Euristica 1	*278*
Mateuristica(60s)	*230*

perc=0.82

m=20	
n=60	
prob=0.4	
BestSol(60s)	*245*
BestBound(60s)	*240*
Euristica 1	*284*
Mateuristica(60s)	*244*

perc=0.87

m=20	
n=80	
prob=0.2	
BestSol(60s)	*305*
BestBound(60s)	*300*
Euristica 1	*334*
Mateuristica(60s)	*307*

perc=0.9

m=20	
n=80	
prob=0.3	
BestSol(60s)	*299*
BestBound(60s)	*292*
Euristica 1	*336*
Mateuristica(60s)	*297*

perc=0.88

m=20	
n=100	
prob=0.2	
BestSol(60s)	*385*
BestBound(60s)	*381*
Euristica 1	*428*
Mateuristica(60s)	*385*

perc=0.92

m=20	
n=100	
prob=0.3	
BestSol(60s)	*390*
BestBound(60s)	*379*
Euristica 1	*413*
Mateuristica(60s)	*388*

perc=0.89

m=20	
n=150	
prob=0.2	
BestSol(60s)	*573*
BestBound(60s)	*562*
Euristica 1	*613*
Mateuristica(60s)	*570*

perc=0.91

m=20	
n=200	
prob=0.2	
BestSol(60s)	*749*
BestBound(60s)	*743*
Euristica 1	*828*
Mateuristica(60s)	*751*

perc=0.89

m=20	
n=250	
prob=0.1	
BestSol(60s)	*944*
BestBound(60s)	*937*
Euristica 1	*979*
Mateuristica(60s)	*944*

perc=0.95

m=20	
n=300	
prob=0.1	
BestSol(60s)	*1123*
BestBound(60s)	*1112*
Euristica 1	*1165*
Mateuristica(60s)	*1120*

perc=0.95

m=30	
n=100	
prob=0.4	
BestSol(60s)	*253*
BestBound(60s)	*246*
Euristica 1	*284*
Mateuristica(60s)	*260*

perc=0.92

m=30	
n=100	
prob=0.5	
BestSol(60s)	*261*
BestBound(60s)	*243*
Euristica 1	*291*
Mateuristica(60s)	*257*

perc=0.92

m=30	
n=150	
prob=0.3	
BestSol(60s)	*394*
BestBound(60s)	*383*
Euristica 1	*441*
Mateuristica(60s)	*395*

perc=0.94

m=30	
n=150	
prob=0.4	
BestSol(60s)	*390*
BestBound(60s)	*375*
Euristica 1	*429*
Mateuristica(60s)	*393*

perc=0.91

m=30	
n=200	
prob=0.2	
BestSol(60s)	*509*
BestBound(60s)	*495*
Euristica 1	*556*
Mateuristica(60s)	*509*

perc=0.95

m=30	
n=200	
prob=0.3	
BestSol(60s)	*511*
BestBound(60s)	*488*
Euristica 1	*567*
Mateuristica(60s)	*507*

perc=0.93

m=30	
n=250	
prob=0.1	
BestSol(60s)	*626*
BestBound(60s)	*615*
Euristica 1	*659*
Mateuristica(60s)	*623*

perc=0.97

m=30	
n=250	
prob=0.2	
BestSol(60s)	*650*
BestBound(60s)	*624*
Euristica 1	*679*
Mateuristica(60s)	*636*

perc=0.95

m=30	
n=300	
prob=0.1	
BestSol(60s)	*766*
BestBound(60s)	*745*
Euristica 1	*792*
Mateuristica(60s)	*761*

perc=0.94

m=30	
n=300	
prob=0.2	
BestSol(60s)	*786*
BestBound(60s)	*744*
Euristica 1	*819*
Mateuristica(60s)	*765*

perc=0.9

m=30	
n=400	
prob=0.1	
BestSol(60s)	*1010*
BestBound(60s)	*1002*
Euristica 1	*1065*
Mateuristica(60s)	*1011*

perc=0.97

m=30	
n=500	
prob=0.1	
BestSol(60s)	*1272*
BestBound(60s)	*1248*
Euristica 1	*1302*
Mateuristica(60s)	*1259*

perc=0.97

m=50	
n=150	
prob=0.4	
BestSol(60s)	*232*
BestBound(60s)	*223*
Euristica 1	*276*
Mateuristica(60s)	*234*

perc=0.96

m=50	
n=150	
prob=0.5	
BestSol(60s)	*230*
BestBound(60s)	*220*
Euristica 1	*257*
Mateuristica(60s)	*236*

perc=0.96

m=50	
n=300	
prob=0.2	
BestSol(60s)	*477*
BestBound(60s)	*442*
Euristica 1	*483*
Mateuristica(60s)	*456*

perc=0.975

m=50	
n=300	
prob=0.3	
BestSol(60s)	*485*
BestBound(60s)	*444*
Euristica 1	*487*
Mateuristica(60s)	*471*

perc=0.97

m=50	
n=500	
prob=0.1	
BestSol(90s)	*781*
BestBound(90s)	*756*
Euristica 1	*797*
Mateuristica(90s)	*766*

perc=0.985

m=50	
n=500	
prob=0.2	
BestSol(180s)	*804*
BestBound(180s)	*756*
Euristica 1	*820*
Mateuristica(180s)	*773*

perc=0.97

m=10	
n=150	
prob=0.1	
BestSol(60s)	*1133*
BestBound(60s)	*1131*
Euristica 1	*1182*
Mateuristica(60s)	*1131*

perc=0.85

m=10	
n=80	
prob=0.2	
BestSol(60s)	*608*
BestBound(60s)	*605*
Euristica 1	*668*
Mateuristica(60s)	*609*

perc=0.75

m=10	
n=100	
prob=0.1	
BestSol(60s)	*741*
BestBound(60s)	*738*
Euristica 1	*779*
Mateuristica(60s)	*739*

perc=0.85

m=20	
n=150	
prob=0.1	
BestSol(60s)	*554*
BestBound(60s)	*547*
Euristica 1	*612*
Mateuristica(60s)	*552*

perc=0.96

m=20	
n=200	
prob=0.1	
BestSol(60s)	*742*
BestBound(60s)	*737*
Euristica 1	*772*
Mateuristica(60s)	*742*

perc=0.955

m=20	
n=350	
prob=0.1	
BestSol(60s)	*1323*
BestBound(60s)	*1293*
Euristica 1	*1349*
Mateuristica(60s)	*1301*

perc=0.96

m=20	
n=80	
prob=0.4	
BestSol(60s)	*310*
BestBound(60s)	*301*
Euristica 1	*347*
Mateuristica(60s)	*309*

perc=0.875

m=20	
n=80	
prob=0.5	
BestSol(60s)	*307*
BestBound(60s)	*290*
Euristica 1	*349*
Mateuristica(60s)	*305*

perc=0.85

m=20	
n=100	
prob=0.4	
BestSol(60s)	*376*
BestBound(60s)	*364*
Euristica 1	*429*
Mateuristica(60s)	*380*

perc=0.865

m=20	
n=400	
prob=0.1	
BestSol(60s)	*1510*
BestBound(60s)	*1491*
Euristica 1	*1538*
Mateuristica(60s)	*1497*

perc=0.94

m=30	
n=150	
prob=0.2	
BestSol(60s)	*389*
BestBound(60s)	*381*
Euristica 1	*416*
Mateuristica(60s)	*388*

perc=0.96

m=30	
n=200	
prob=0.1	
BestSol(60s)	*498*
BestBound(60s)	*492*
Euristica 1	*550*
Mateuristica(60s)	*503*

perc=0.98

m=30	
n=100	
prob=0.3	
BestSol(60s)	*258*
BestBound(60s)	*249*
Euristica 1	*303*
Mateuristica(60s)	*259*

perc=0.93

m=30	
n=250	
prob=0.2	
BestSol(60s)	*646*
BestBound(60s)	*621*
Euristica 1	*676*
Mateuristica(60s)	*635*

perc=0.95

m=30	
n=350	
prob=0.1	
BestSol(60s)	*893*
BestBound(60s)	*876*
Euristica 1	*920*
Mateuristica(60s)	*887*

perc=0.97

m=30	
n=600	
prob=0.1	
BestSol(90s)	*1564*
BestBound(90s)	*1489*
Euristica 1	*1547*
Mateuristica(90s)	*1500*

perc=0.97

m=50	
n=200	
prob=0.3	
BestSol(60s)	*324*
BestBound(60s)	*299*
Euristica 1	*349*
Mateuristica(60s)	*316*

perc=0.97

m=50	
n=200	
prob=0.4	
BestSol(60s)	*330*
BestBound(60s)	*300*
Euristica 1	*348*
Mateuristica(60s)	*321*

perc=0.96

m=50	
n=400	
prob=0.1	
BestSol(90s)	*625*
BestBound(90s)	*598*
Euristica 1	*646*
Mateuristica(90s)	*618*

perc=0.99

m=50	
n=400	
prob=0.2	
BestSol(120s)	*639*
BestBound(120s)	*602*
Euristica 1	*649*
Mateuristica(120s)	*628*

perc=0.98

Come era stato preannunciato dalla tabella presente nel Paragrafo 3, la mat-euristica ha ottenuto risultati migliori del solver nel 64% dei casi.
Quando le soluzioni del solver sono migliori, queste discostano in media da quelle della mat-euristica di 2.929, con varianza di 3.764.
Al contrario, quando le soluzioni della mat-euristica sono migliori, queste discostano in media da quelle del solver di 9.938, con varianza 155.3.
È possibile affermare, quindi, che, per le prove effettuate, la mat-euristica, quando ottiene delle soluzioni migliori, tali sono ben più buone, in media, di quelle ottenute dal solver; mentre, quando il solver ottiene delle soluzioni migliori, queste non sono molto distanti da quelle ottenute mediante mat-euristica.

5
Conclusioni ed osservazioni finali

Come risulta evidente dai risultati computazionali ottenuti, la mat-euristica lavora bene: per la maggior parte delle volte (72%) ottiene dei risultati migliori o uguali rispetto a quelli del solver.

Assumendo di voler ottenere una soluzione ottima o ammissibile, ma vicina all'ottimo, del problema *PMC,* si consiglia l'utilizzo di un solver di Programmazione Lineare come *FICO@Xpress,* come risulta scontato, per tutte quelle istanze in cui il solver trovi l'ottimo entro il tempo prestabilito: istanze relativamente piccole, come quelle riportate in tabella al Paragrafo 4.1 (quando non vi è scritto *MAXTIME* o *y=0*).
Per istanze più grandi, o con maggior numero di conflitti, come quelle riportate in tabella al Paragrafo 4.1 (quando vi è scritto *MAXTIME),* o istanze ancora più grandi, come quelle utilizzate per il confronto tra le euristiche o per il testing della mat-euristica, è ragionevole fare delle osservazioni sulla base dei risultati ottenuti:

- per istanze grandi, ma comunque con $m \leq 30$ *e* $n \leq 250$, risulta quasi indifferente scegliere se usare la mat-euristica o il solver: si nota che talvolta si ottengono addirittura gli stessi risultati e talvolta prevale l'una sull'altra o viceversa, ma osservando i valori di *Best Bound,* ci si rende conto che le soluzioni ottenute sono molto vicine all'ottimo;
- per istanze molto grandi, quindi con $m \geq 30$ *e* $n \geq 250$, risulta più performante l'utilizzo della mat-euristica, perché, anche nei casi il confronto è stato svolto con un tempo massimo maggiore di un minuto, questa otteneva già dalle primissime iterazioni, quindi prima di 60s, un risultato migliore della *Best Solution*. Naturalmente è possibile applicare la mat-euristica solo se l'euristica sia in grado di fornire una soluzione ammissibile da cui partire; ciò risulta quasi sempre vero per le istanze considerate durante le analisi, ma per istanze ancora più grandi o per densità di probabilità ancora maggiori, ciò non è scontato.

6

Appendice

Codice completo per Modello di Programmazione Lineare su *FICO@Xpress*:

```
model ParallelMachineScheduling2
uses "mmxprs", "mmive", "mmsystem";

parameters

   PROJECTDIR="
end-parameters

declarations

 y: mpvar
 macchine=1..10
 jobs=1..30
 x: array(macchine, jobs) of mpvar
 p: array(jobs) of integer
 coppiaConflitto: array(jobs, jobs) of boolean
 alea: real

end-declarations

 forall(j in jobs) do
 p(j):=1+integer(99*random)
 end-do

setparam("XPRS_MAXTIME",-60)
starttime:= gettime


forall(j in jobs)do
sum (i in macchine) x(i,j)=1
end-do


forall(i in macchine)do
sum(j in jobs) p(j)*x(i,j)<=y
end-do


forall(i in macchine, j in jobs) do
x(i,j) is_binary
end-do
```

```
forall(j in jobs)do
forall(k in jobs)do
if(k>j)then
alea:=random
if(alea<=0.50)then
coppiaConflitto(j,k):=true
end-if
if(alea>0.50)then
coppiaConflitto(j,k):=false
end-if
end-if
end-do
end-do

forall(i in macchine)do
forall(j in jobs)do
forall(k in jobs)do
if(k>j) then
if(coppiaConflitto(j,k)=true)then
x(i,j)+x(i,k)<=1
end-if
end-if
end-do
end-do
end-do

y>=0

minimize(y)

writeln("Il makespan calcolato è: " , getobjval)
writeln("Time = ",gettime-starttime);

end-model
```

Codice *Java* completo per le euristiche, implementato sull'ambiente di *sviluppo Eclipse IDE* (sono presenti anche alcuni metodi per creare e scrivere su file da fornire alla mateuristica):

```
package euristicaMain;
import euristicaClassi.Problem;

public class MainEuristica {
	public static void main(String[] args) {

		long start =System.currentTimeMillis();
		Problem scheduling= new Problem();
		int macchine= 50;
		int jobs= 400;
```

```
            for(int i=1; i<=macchine; i++) {
                  scheduling.inserisciMacchina(i);
            }

            scheduling.leggiProcessTime("inputProcessTime.txt");
            int nConfl=scheduling.leggiConflitti("inputConflitti.txt");
            double probConfl=(double)(2*nConfl)/((jobs*(jobs-1)));
            scheduling.setProbConfl(probConfl);

            if(scheduling.schedula()) {
                  System.out.println("Il makespan calcolato con
l'euristica consederata è: "+scheduling.Makespan()+"\nIl tempo di
esecuzione è: "+ (System.currentTimeMillis()-start) +" ms");

                  scheduling.fileXIJ();
                  scheduling.fileConflicts();

            }else {
                  System.out.print("L'euristica considerata non riesce a
trovare una soluzione ammissibile per gli input inseriti!");
            }
      }
}
```

```
package euristicaClassi;
import java.util.List;

public class Macchina {

      private int machineId;
      private int c;
      private List<Job> jobSchedulati;

      public Macchina(int machineId, int c, List<Job> jobSchedulati) {
            super();
            this.machineId = machineId;
            this.c = c;
            this.jobSchedulati = jobSchedulati;
     }

      public int getMachineId() {
            return machineId;
      }

      public void setMachineId(int machineId) {
            this.machineId = machineId;
      }

      public int getC() {
```

```
		return c;
	}

	public void setC(int c) {
		this.c = c;
	}

	public List<Job> getJobSchedulati() {
		return jobSchedulati;
	}

	public void setJobSchedulati(List<Job> jobSchedulati) {
		this.jobSchedulati = jobSchedulati;
	}

	@Override
	public int hashCode() {
		final int prime = 31;
		int result = 1;
		result = prime * result + machineId;
		return result;
	}

	@Override
	public boolean equals(Object obj) {
		if (this == obj)
			return true;
		if (obj == null)
			return false;
		if (getClass() != obj.getClass())
			return false;
		Macchina other = (Macchina) obj;
		if (machineId != other.machineId)
			return false;
		return true;
	}

	@Override
	public String toString() {
		return "Macchina [machineId=" + machineId + ", c=" + c + "]";
	}
}
```

```
package euristicaClassi;
import java.util.List;

public class Job {
	private int id;
	private int p;
```

```
	private Macchina macchinaACuiVieneAssegnato;
	private List<Integer> conflitti;

	public Job(int id, int p, Macchina macchinaACuiVieneAssegnato,
List<Integer> conflitti) {

		this.id = id;
		this.p = p;
		this.macchinaACuiVieneAssegnato = macchinaACuiVieneAssegnato;
		this.conflitti = conflitti;
	}

	public int getId() {
		return id;
	}

	public void setId(int id) {
		this.id = id;
	}

	public int getP() {
		return p;
	}

	public void setP(int p) {
		this.p = p;
	}

	public Macchina getMacchinaACuiVieneAssegnato() {
		return macchinaACuiVieneAssegnato;
	}

	public void setMacchinaACuiVieneAssegnato(Macchina
macchinaACuiVieneAssegnato) {
		this.macchinaACuiVieneAssegnato = macchinaACuiVieneAssegnato;
	}

	public List<Integer> getConflitti() {
		return conflitti;
	}

	public void setConflitti(List<Integer> conflitti) {
		this.conflitti = conflitti;
	}

	@Override
	public String toString() {
		return "Job [id=" + id + ", p=" + p + "]";
	}
```

```
	@Override
	public int hashCode() {
		final int prime = 31;
		int result = 1;
		result = prime * result + id;
		return result;
	}

	@Override
	public boolean equals(Object obj) {
		if (this == obj)
			return true;
		if (obj == null)
			return false;
		if (getClass() != obj.getClass())
			return false;
		Job other = (Job) obj;
		if (id != other.id)
			return false;
		return true;
	}
}
```

```
package euristicaClassi;

public class Conflitto {

   private int idJob1;
   private int idJob2;

public Conflitto(int idJob1, int idJob2) {
	super();
	this.idJob1 = idJob1;
	this.idJob2 = idJob2;
}

public int getIdJob1() {
	return idJob1;
}
public void setIdJob1(int idJob1) {
	this.idJob1 = idJob1;
}

public int getIdJob2() {
	return idJob2;
}

public void setIdJob2(int idJob2) {
	this.idJob2 = idJob2;
}
```

```
}
```

```
package euristicaClassi;

import java.util.Comparator;

public class ComparatoreDiJobs implements Comparator<Job>{

	@Override
	public int compare(Job j1, Job j2) {

		if(j1.getConflitti().size()!=j2.getConflitti().size()) {
			return -Integer.compare(j1.getConflitti().size(),
					j2.getConflitti().size());
		}else {
			return -Integer.compare(j1.getP(), j2.getP());
		}
	}
}
```

```
package euristicaClassi;
import java.util.Comparator;

public class ComparatoreDiJobs2 implements Comparator<Job>{

	@Override
	public int compare(Job j1, Job j2) {

		if(j1.getP()!= j2.getP()) {
			return -Integer.compare(j1.getP(), j2.getP());
		}else {
			return -Integer.compare(j1.getConflitti().size(),
					j2.getConflitti().size());
		}

	}
}
```

```
package euristicaClassi;
import java.io.BufferedReader;
import java.io.File;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
import java.util.Collections;
import java.util.LinkedList;
import java.util.List;
import java.util.TreeMap;
```

```java
public class Problem {

	List<Macchina> macchine= new LinkedList<>();
	List<Job> jobs=  new LinkedList<>();
	List<Conflitto> conflitti =  new LinkedList<>();
	double probConfl;

	public Macchina inserisciMacchina(int id) {
		Macchina temp= new Macchina(id, 0, null);
		this.macchine.add(temp);
		return temp;

	}

	public Macchina macchina(int id) {
		for(Macchina m: this.macchine) {
			if(m.getMachineId()==id) {
				return m;
			}
		}
		return null;
	}

	public Job inserisciJob(int id, int p) {
		Job temp= new Job(id, p, null, new LinkedList<Integer>());
		this.jobs.add(temp);
		return temp;
	}

	public Job job(int id) {
		for(Job j :this.jobs) {
			if(j.getId()==id) {
				return j;
			}
		}
		return null;
	}

	public void setProbConfl(double probConfl) {
		this.probConfl=probConfl;
	}

	public Conflitto inserisciConflitto(int id1, int id2) {
		Conflitto temp= new Conflitto(id1, id2);

		if(this.job(id1).getConflitti()==null) {
			this.job(id1).setConflitti(new LinkedList<>());

		}
```

```
        if(this.job(id2)!=null && this.job(id1)!= null)
           this.job(id1).getConflitti().add(id2);

        if(this.job(id2).getConflitti()==null)
             this.job(id2).setConflitti(new LinkedList<>());


        if(this.job(id2)!=null && this.job(id1)!= null) {
           this.job(id2).getConflitti().add(id1);

        }
        return temp;
    }

    public List<Job> jobOrdinati(){
        LinkedList<Job> jobsOrdinati= new LinkedList<>(this.jobs);
        ComparatoreDiJobs comparatore= new ComparatoreDiJobs();

        Collections.sort(jobsOrdinati, comparatore);

        return jobsOrdinati;
    }

    public  List<Job> jobOrdinati2(){
        LinkedList<Job> jobsOrdinati2= new LinkedList<>(this.jobs);
        ComparatoreDiJobs2 comparatore2= new ComparatoreDiJobs2();

        Collections.sort(jobsOrdinati2, comparatore2);

        return jobsOrdinati2;
    }

    public  List<Job> jobOrdinati3(){

        ComparatoreDiJobs comparatoreParz1= new ComparatoreDiJobs();
        ComparatoreDiJobs2 comparatoreParz2= new ComparatoreDiJobs2();
        LinkedList<Job> primaMeta= new LinkedList<>();
        LinkedList<Job> secondaMeta= new LinkedList<>();
        LinkedList<Job> listaOrdinata= new LinkedList<>();
        int size=this.jobs.size();
        for(int i=0; i<size; i++) {
             Job temp= this.jobs.get(i);
             if(i<this.probConfl*size) {
                  primaMeta.add(temp);
             }else {
                  secondaMeta.add(temp);
             }
        }
        Collections.sort(primaMeta, comparatoreParz1);
        Collections.sort(secondaMeta, comparatoreParz2);
```

```
            listaOrdinata.addAll(primaMeta);
            listaOrdinata.addAll(secondaMeta);

            return listaOrdinata;
      }

      public boolean schedula() {
            boolean ammissibile= true;

            for(int i=0; i<this.macchine.size(); i++) {
                  Macchina tempM= macchine.get(i);
                  tempM.setJobSchedulati(new LinkedList<>());
                  Job tempJ=this.jobOrdinati().get(i);
                  tempM.getJobSchedulati().add(tempJ);
                  tempM.setC(tempJ.getP());
                  tempJ.setMacchinaACuiVieneAssegnato(tempM);

            }

            int nJobs=this.jobOrdinati().size();

            for(int i=this.macchine.size(); i<nJobs && ammissibile==true;
i++) {
                  Job temp= this.jobOrdinati().get(i);
                  if(this.macchineAmmissibili(temp)==null) {
                        ammissibile= false;
                        break;
                  }
                 Macchina scelta=
this.macchinaConMinimoC(this.macchineAmmissibili(temp));

this.job(temp.getId()).setMacchinaACuiVieneAssegnato(scelta);

this.macchina(scelta.getMachineId()).getJobSchedulati().add(temp);

this.macchina(scelta.getMachineId()).setC(scelta.getC()+temp.getP());
            }

            return ammissibile;
      }

      public Macchina macchinaConMinimoC(List<Macchina> listaMacchine) {
            int minimo = Integer.MAX_VALUE;
            Macchina macchinaMin=null;

            for(Macchina m: listaMacchine) {
                  if(m.getC()<minimo) {
                        minimo= m.getC();
                        macchinaMin=m;
                  }
```

```
            }

            return macchinaMin;
      }

      public List<Macchina> macchineAmmissibili(Job j){
            LinkedList<Macchina> macchineAmmissibili= new
LinkedList<>(this.macchine);

            for(Macchina m: this.macchine) {
                for(Job job : m.getJobSchedulati()) {
                 for(int i=0; i<j.getConflitti().size(); i++) {
                           if(job.getId()==j.getConflitti().get(i)) {
                            macchineAmmissibili.remove(m);
                           }
                 }
                 }
            }
            if(macchineAmmissibili.size()==0) {
                  return null;
            }

            return macchineAmmissibili;
      }

      public int Makespan() {
            int max = Integer.MIN_VALUE;

            for(Macchina m: this.macchine) {
                  if(m.getC()>max) {

                        max= m.getC();
                  }
            }

            return max;
      }

      public void leggiProcessTime(String nomeFile) {
            try {
                  FileReader fr = new FileReader(nomeFile);
                  BufferedReader br = new BufferedReader(fr);

                  String riga;
                  int id=1;
                  while( (riga = br.readLine()) != null ) {

                        String pString= riga;
                        int p= Integer.valueOf(pString);
                        this.inserisciJob(id, p);
```

```
                    id++;

               }
               br.close();
               fr.close();
          }
          catch(Exception e) {
               e.printStackTrace();
          }

     }

     public int leggiConflitti(String nomeFile) {
          try {
               FileReader fr = new FileReader(nomeFile);
               BufferedReader br = new BufferedReader(fr);

               String riga;
              int nConfl=0;
               while( (riga = br.readLine()) != null ) {
                    if(riga.contains(",")) {
                      nConfl++;
                      String campi[] = riga.split(",");
                      int id1= Integer.parseInt(campi[0]);
                      int id2= Integer.parseInt(campi[1]);
                      this.inserisciConflitto(id1, id2);
                    }
               }
               br.close();
               fr.close();
               return nConfl;
          }
          catch(Exception e) {
               e.printStackTrace();
               return 0;
          }



     }

     public List<Job> jobOrdinati4(){
          LinkedList<Job> jobsOrdinati2= new LinkedList<>(this.jobs);
          ComparatoreDiJobs2 comparatore2= new ComparatoreDiJobs2();

          Collections.sort(jobsOrdinati2, comparatore2);

          int nJobs=this.jobs.size();
          int nMac=this.macchine.size();
          double nDivisoM= (double)nJobs/nMac;
```

```
         int nDivisoMInt= (int)nJobs/nMac;

         int jobsTotali=nJobs;
          if(nDivisoM!=nDivisoMInt) {
             int interoSup= nDivisoMInt+1;
             int jobsDaAgg= (interoSup*nMac)-nJobs;
             jobsTotali+=jobsDaAgg;
             for(int i=nJobs+1; i<=jobsTotali; i++) {
                  Job temp= new Job(i, 0,  null, new
LinkedList<Integer>());
                  jobsOrdinati2.add(temp);
             }
          }

          List<Job> listaOutput= new LinkedList<>();

          int nTuple= jobsTotali/nMac;
          int inizioTupla[]= new int[nTuple];
          int fineTupla[]= new int[nTuple];

          TreeMap<Double, LinkedList<Job>> mappeSlackJobs= new
TreeMap<>(Collections.reverseOrder());
          LinkedList<Job> temp= new LinkedList<>();
          inizioTupla[0]=0;
          fineTupla[0]=nMac;

          for(int i=0; i<nTuple; i++) {
               if(i==0) {
                    int slack=jobsOrdinati2.get(inizioTupla[i]).getP()-
jobsOrdinati2.get(fineTupla[i]-1).getP();
                    int nConflittiTupla=0;
                    for(int j=inizioTupla[i]; j<fineTupla[i]; j++) {
                         temp.add(jobsOrdinati2.get(j));

     nConflittiTupla+=jobsOrdinati2.get(j).getConflitti().size();
                    }
               double indiceD=(double) (nTuple-i)/10000000;

                    Double
nConflittiTuplaD=(double)(nConflittiTupla)/10000;
                    double slackD=(double)slack*10;
                    double key=slackD+nConflittiTuplaD+indiceD;

                    mappeSlackJobs.put(key, new LinkedList<>(temp));
                   temp.clear();
               }else {
                    inizioTupla[i]=fineTupla[i-1];
                    fineTupla[i]=inizioTupla[i]+nMac;
```

```
                    int slack=jobsOrdinati2.get(inizioTupla[i]).getP()-
jobsOrdinati2.get(fineTupla[i]-1).getP();
                    int nConflittiTupla=0;
                    for(int j=inizioTupla[i]; j<fineTupla[i]; j++) {
                         temp.add(jobsOrdinati2.get(j));

     nConflittiTupla+=jobsOrdinati2.get(j).getConflitti().size();
                    }
                    double indiceD=(double) (nTuple-i)/10000000;

                    Double
nConflittiTuplaD=(double)(nConflittiTupla)/10000;
                    double slackD=(double)slack*10;
                    double key=slackD+nConflittiTuplaD+indiceD;

                    mappeSlackJobs.put(key, new LinkedList<>(temp));
                   temp.clear();
               }
          }

          for(double i: mappeSlackJobs.keySet()) {
               listaOutput.addAll(mappeSlackJobs.get(i));

          }

          return listaOutput;
     }


     public void stampaProcessTime() {
          int nJobsTot= this.jobOrdinati().size();
          System.out.println("\n\n");
          for(int i=0; i<nJobsTot; i++) {
               Job temp =this.jobOrdinati().get(i);
               System.out.println(temp.getP());
          }
     }

     public void fileXIJ(){
          try {
                File fileXIJ = new File("inputXIJ.txt");
               fileXIJ.createNewFile();
               FileWriter myWriter = new FileWriter("inputXIJ.txt");

               String s="";


               for(Macchina m: this.macchine) {
                for(Job j: this.jobs) {
                     if(m.getJobSchedulati().contains(j)) {
```

```
                              s+="1 \n";

                        }else {
                              s+="0 \n";

                        }
                    }
                  }

                  myWriter.write(s);
                  myWriter.close();
                } catch (IOException e) {
                  System.out.println("An error occurred.");
                  e.printStackTrace();
                }
      }

      public void fileConflicts() {
            try {
            File fileCon = new File("inputConflicts.txt");
            fileCon.createNewFile();
            FileWriter myWriter = new FileWriter("inputConflicts.txt");

            String s="";

            for(Job j: this.jobs) {
              for(Job k: this.jobs) {
                  if(k.getId()>j.getId()) {

                        if(j.getConflitti().contains(k.getId())) {
                              s+="true \n";

                        }else {
                              s+="false \n";

                        }
                  }
              }
            }

            myWriter.write(s);
            myWriter.close();
          } catch (IOException e) {
            System.out.println("An error occurred.");
            e.printStackTrace();
          }
      }
}
```

Codice per la mateuristica:

```
model ParallelMachineScheduling2
uses "mmxprs", "mmive", "mmsystem";

parameters

   PROJECTDIR=' '
end-parameters

declarations

 y: mpvar
 macchine=1..50
 jobs=1..400
 x: array(macchine, jobs) of mpvar
 p: array(jobs) of integer
 coppiaConflitto: array(jobs, jobs) of boolean
 alea: real
 xIJ: array(macchine, jobs) of integer

end-declarations

 fopen("inputP.txt", F_INPUT)
 forall(j in jobs) read(p(j))
 fclose(F_INPUT)


 fopen("inputConflicts.txt", F_INPUT)
 forall(j in jobs)
 forall(k in jobs)
 if(k>j)then
 read(coppiaConflitto(j,k))
 end-if
 fclose(F_INPUT)

 starttime:= gettime

 fopen("inputXIJ.txt", F_INPUT)
 forall(i in macchine)
 forall(j in jobs)
 read(xIJ(i,j))
 fclose(F_INPUT)

 forall(i in macchine)do
 forall(j in jobs)do
 alea:=random
 if(xIJ(i,j)=0 and alea<=0.98)then
 settype(x(i,j), CT_EQ)
 end-if
```

```
 end-do
 end-do

forall(j in jobs)do
sum (i in macchine) x(i,j)=1
end-do


forall(i in macchine)do
sum(j in jobs) p(j)*x(i,j)<=y
end-do


forall(i in macchine, j in jobs) do
x(i,j) is_binary
end-do

forall(i in macchine)do
forall(j in jobs)do
forall(k in jobs)do
if(k>j) then
if(coppiaConflitto(j,k)=true)then
x(i,j)+x(i,k)<=1
end-if
end-if
end-do
end-do
end-do

y>=0

minimize(y)

 fopen("inputXIJ.txt", F_OUTPUT)
 forall(i in macchine)
 forall(j in jobs)
 writeln(getsol(x(i,j)))
 fclose(F_OUTPUT)

 stoptime:=gettime;


writeln("Il makespan ottenuto è: " , getobjval);
writeln("Time = ",stoptime-starttime);

end-model
```

7
Bibliografia

[1] P.Chiabert, Appunti del Corso di Programmazione e Gestione della Produzione

[2] F. Della Croce, A. Grosso and F. Salassa, Heuristics: Theory and Applications, Chapter3: Matheuristics: embedding MILP solvers into heuristic algorithms for combinatorial optimization problems. Nova Science Publishers, Inc, 2013. Pagg. 53, 54.

[3] F. Della Croce, R. Scatamacchia, Journal of Scheduling, 23:163–176: The Longest Processing Time rule for identical parallel machines revisited. Springer Science, Business Media, LLC, part of Springer Nature, 2018. Pag. 172.

[4] M. Ghirardi, A. Grosso, G. Perboli, Esercizi svolti di ricerca operativa. Società Editrice Esculapio, 2019. Cap. 5

[5] D. Kowalczyk, R. Leus, An exact algorithm for parallel machine scheduling with conflicts. Springer Science, Business Media New York, 2016. Pagg. 1, 2.

[6] D. Kowalczyk, R. Leus, An exact algorithm for parallel machine scheduling with conflicts. Springer Science, Business Media New York, 2016. Pagg. 2, 3.

[7] R. Tadei, F. Della Croce, Elementi di ricerca operativa. Società Editrice Esculapio, 2019. Cap. 1.

[8] R. Tadei, F. Della Croce, Elementi di ricerca operativa. Società Editrice Esculapio, 2019. Capp. 2, 7.

[9] R. Tadei, F. Della Croce, Elementi di ricerca operativa. Società Editrice Esculapio, 2019. Cap. 5.

[10] R. Tadei, F. Della Croce, Elementi di ricerca operativa. Società Editrice Esculapio, 2019. Cap. 6

[11] www.treccani.it, Enciclopedia della Matematica, soluzione ammissibile, 2013.

[12] www.treccani.it, Enciclopedia on line, euristica