Pianificazione e gestione dei servizi sanitari

IMPLEMENTAZIONE DELL'EURISTICA (continuazione seminario)

Prof. Assistente di Domenico Conforti – 16/11/2023 – Autori/ Revisionatori: Gervasi e Raponi

In questa lezione, le nozioni trattate nelle lezioni precedenti riguardo al software verranno implementate con il linguaggio OPL.

Il linguaggio OPL si distingue dagli altri linguaggi di programmazione poiché non è procedurale, ovvero, tutte le istruzioni sono eseguite in contemporanea. Il modello viene, quindi, valutato complessivamente e viene sviluppata una soluzione.

IBM ILOG SCIPT: IMPLEMENTARE UNA EURISTICA

Per problemi di grandi dimensioni o problemi da risolvere in brevissimo tempo, bisogna accontentarsi della soluzione euristica (poiché la soluzione ottima richiede troppo tempo).
L'euristica è "l'implementazione del buon senso", ovvero, anziché esplorare tutte le soluzioni ammissibili e individuare quella con il costo più basso, si cerca di individuare una soluzione basata su criteri di equilibrio fra l'ottimizzazione matematica e il buon senso.

*Nella lezione, la logica euristica sarà applicata su un problema non troppo difficile da risolvere per aiutarne la comprensione e per capire come gestire il "**controllo del flusso**", ovvero, l'insieme delle istruzioni che portano all'individuazione della soluzione.*

PROBLEMA:
RideEasy ha deciso di lanciare una bicicletta in edizione speciale di alta qualità destinata al mercato di New York. Tali biciclette, denominate CycleMe, sono realizzate su ordinazione in una officina dedicata alla periferia della città.
La responsabile dell'officina ha appena ricevuto gli ordinativi da consegnare nei prossimi 20 giorni (t); indagini di mercato hanno mostrato che il grado di soddisfazione del cliente risente, oltre che dalla qualità costruttiva delle biciclette, dalla puntualità nelle consegne.
La responsabile della produzione deve definire il piano di produzione tenendo conto:

- delle **capacità giornaliera di produzione (c_t),** variabile nei prossimi 20 giorni

e cercando di minimizzare i costi complessivi, suddivisibili nelle seguenti voci (tutte variabili nei prossimi 20 giorni):

- **costi di setup (s_t):** costi fissi, sostenuti giornalmente, se in quel giorno vengono prodotte biciclette;
- **costi di produzione (p_t):** costo di produzione di ogni bicicletta prodotta (costo unitario);
- **costi di stoccaggio (h_t):** costo di stoccaggio per unità di tempo e unità di prodotto.

LotSizing_Model.mod | LotSizing_Data.dat

```
/*********************************************
 * OPL 22.1.0.0 Model
 * Author: Win10
 * Creation Date: 30 ago 2023 at 11:11:15
 Dal libro di Nickel et al. pag. 257
 *********************************************/

int numPeriodi = ...;
range Periodi = 1..numPeriodi;

int C_SetUp [Periodi] = ...;
int C_Stoccaggio [Periodi] = ...;
int C_Produzione [Periodi] = ...;
int Domanda [Periodi] = ...;
int Capacita [Periodi] = ...;

dvar boolean z[Periodi];
dvar float+ i[0..numPeriodi];
dvar float+ q[Periodi];

minimize sum (t in Periodi) (C_SetUp[t]*z[t] + C_Stoccaggio[t]*i[t] + C_Produzione[t]*q[t]);

subject to{

   i[0] == 0;

   forall (t in Periodi)
          i[t] == i[t-1] + q[t] - Domanda[t];

   forall (t in Periodi)
          q[t] <= Capacita[t]*z[t];
}
```

Il modello è sviluppato sull'orizzonte di pianificazione (=20 giorni).

Inizialmente si definiscono le variabili:

- *numPeriodi*: numero totale di periodi (in questo caso 20).
- *Periodi*: un range variabile da 1 a *numPeriodi.*
- Diverse array *(C_SetUp, C_Stoccaggio, C_Produzione, Domanda, Capacita*) che rappresentano costi, domande e capacità nei vari periodi.
- z: è un array booleano che rappresenta una variabile decisionale binaria che indica se la produzione è attiva o meno in un periodo.

Inoltre, assumiamo che la quantità di bici prodotta (q[t]) sia una variabile continua.

Funzione obiettivo:

$$minimize\ sum\ (t\ in\ Periodi)(C\_SetUp[t]\ *z[t] + C\_Stoccaggio[t] * i[t] + C\_Produzione\ [t] * q[t])$$

L'obiettivo del modello è minimizzare la somma dei costi associati alle variabili decisionali.

Vincoli:

- *[i] == 0* inizializza la variabile *i* a 0 (non ci sono bici inizialmente).
- *i[t] == i[t-1] + q[t] - Domanda[t]:* indica che il valore di *i* in ogni periodo è uguale al valore di *i* nel periodo precedente, più la quantità prodotta *q[t`*, meno la domanda del periodo corrente.

- *q[t] <= Capacita[t]*z[t]:* indica che la quantità prodotta *q[t]* in ogni periodo non può superare la capacità massima di produzione *Capacita[t]*.

Table 16.1 Data for lot sizing [with d_t for demand on day t, c_t for available capacities for production on day t, s_t for setup costs on day t in USD, h_t for holding costs on day t in USD, p_t for production costs on day t in USD]

t	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
d_t	12	6	20	5	7	25	0	5	5	21	9	5	27	0	5	20	5	10	5	25
c_t	20	15	5	30	20	20	21	10	10	5	0	12	20	5	20	5	40	20	10	20
s_t	25	30	40	20	80	80	150	30	45	60	30	90	80	150	25	20	30	40	20	80
h_t	6	5	4	5	6	9	9	6	5	4	5	6	9	9	6	5	4	5	6	9
p_t	5	3	7	2	1	10	15	2	1	5	1	10	15	2	1	10	1	5	1	10

Viene tradotta, quindi, in OPL la struttura di questa tabella che deve essere implementata con le informazioni sottostanti:

$$\text{Minimize} \sum_{t=1}^{T} (s_t \cdot z_t + h_t \cdot i_t + p_t \cdot q_t) \tag{16.1}$$

subject to the constraints

$$i_0 = 0 \tag{16.2}$$

$$i_t = i_{t-1} + q_t - d_t \qquad \forall\, t = 1, \ldots, T \tag{16.3}$$

$$q_t \leq c_t \cdot z_t \qquad \forall\, t = 1, \ldots, T \tag{16.4}$$

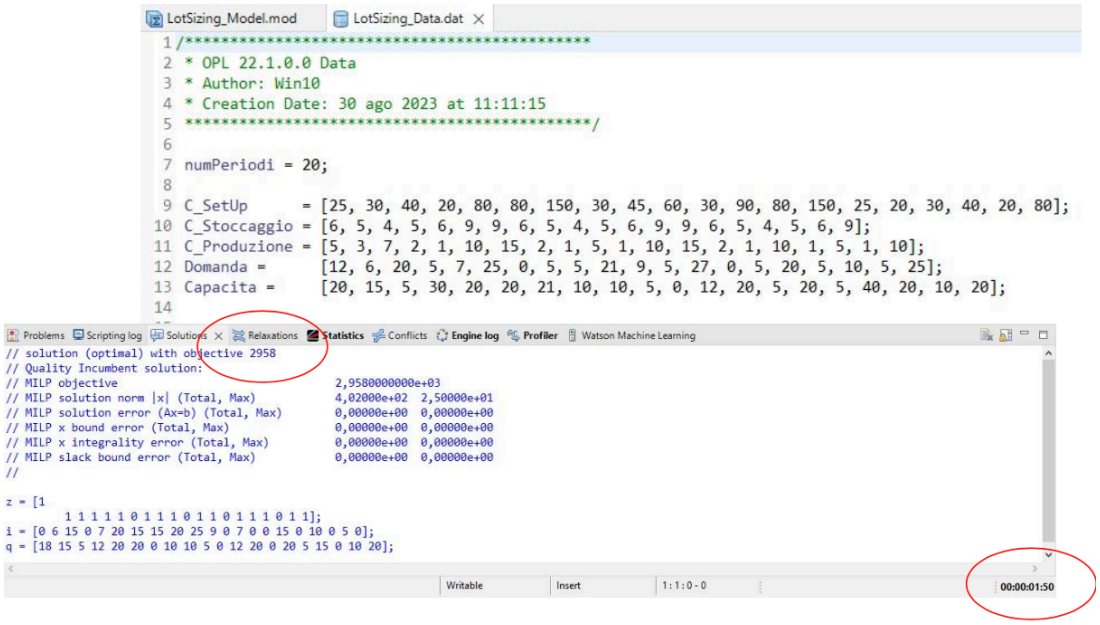
$$q_t, i_t \geq 0 \qquad \forall\, t = 1, \ldots, T \tag{16.5}$$

$$z_t \in \{0; 1\} \qquad \forall\, t = 1, \ldots, T \tag{16.6}$$

Con *i* è stata fatta una indicizzazione particolare (nella 16.2; 16.3). Invece di definire un nuovo set di periodi che va da 0 al numero di periodi complessivo (*T*), si può direttamente modificare l'indicizzazione con $\forall t$. In questo caso, si definisce che il vettore di variabile decisionale i va da 0 a *T* direttamente.

Il 16.3 è un vincolo di bilancio che afferma che alla fine dell'istante t, il valore di *i* è uguale al valore di *i* nell'istante precedente, più la quantità prodotta in t *q[t],* meno la domanda (=quello che ho venduto) nell'istante t.
Il 16.4 afferma che *q[t]* non deve superare *c[t]*. *z[t]* è presente perché posso produrre biciclette solo se l'impianto è stato attivato. Se togliamo *z[t]* stiamo traducendo un vincolo tecnologico (non si possono produrre in un giorno più di *c[t]* biciclette; se aggiungiamo *z[t]* diventa una condizione logica perché il vincolo si presenta solo se la produzione è attiva.
Il 16.5 afferma che il numero di biciclette prodotte è sempre positivo.
Il 16.6 afferma che z_t è una variabile booleana.

Creiamo un file con i data della tabella ed otteniamo la soluzione:



Le variabili difficili nei modelli di ottimizzazione sono quelle non continue. Nel nostro modello l'unica variabile non continua è z_t. Quindi, implementiamo l'euristica "LP-and-Fix heuristic (Pochet and Wolsey, 2006)" e dobbiamo:

1) Risolvere il rilassato lineare del problema MIP di partenza ottenuto sostituendo le variabili $z_t \in 0, 1$ con le variabili $z_t \in [0, 1]$.
   Il rilassato lineare è il problema che si ottiene prendendo il problema iniziale e trasformando le variabili intere in variabili continue. Il numero di soluzioni ammissibili cresce e la complessità del problema diminuisce. L'insieme comprenderà tutti i numeri reali tra 0 e 1.

2) Fissare tutte le variabili che assumono spontaneamente valore 0 o 1 nella soluzione ottima del rilassato lineare ottenuto al passo precedente. Le variabili del rilassato lineare che assumono i valori 0 e 1 vengono fissate come dati del problema e risolveremo nuovamente il problema.

Come si fa?

```
 * OPL 22.1.0.0 Model
 * Author: Win10
 * Creation Date: 30 ago 2023 at 11:33:35
 ********************************************/
main {
  // Creazione del modello
  var source = new IloOplModelSource("LotSizing_Model.mod");
  var def    = new IloOplModelDefinition(source);
  var opl    = new IloOplModel(def, cplex);

  // Aggiunta dei dati sorgente
  var data   = new IloOplDataSource("LotSizing_Data.dat");
  opl.addDataSource(data)

  // Rilassamento dei vincoli di integrità sulle variabili intere
  opl.convertAllIntVars();

  // Generazione del modello
  opl.generate();

   if (cplex.solve()) {
   // Creazione e generazione di una seconda istanza del modello
   var cplex2 = new IloCplex();
   var opl2   = new IloOplModel(def, cplex2);
   opl2.addDataSource(data);
   opl2.generate();
   }

   for (var t=1; t <= opl.dataElements.numPeriodi; t++){
    if (opl.z[t].solutionValue == 0 || opl.z[t].solutionValue == 1){
          opl2.z[t].LB = opl.z[t].solutionValue;
          opl2.z[t].UB = opl.z[t].solutionValue;
    }
   }

   if (cplex2.solve()) {
     writeln ("LP-and-Fix-Heuristic");
     writeln ();
     writeln ("Days of production: " + opl2.z.solutionValue);
     writeln ("Inventories: "        + opl2.i.solutionValue);
     writeln ("Lot sizes: "          + opl2.q.solutionValue);
     writeln ("Total cost: "         + cplex2.getObjValue());
   }

}
```

Creiamo un nuovo file che deve contenere un modello. CPLEX supporta, oltre al linguaggio OPL, un altro linguaggio che prende il nome di IBM ILOG Script (procedurale) derivato da Java Script. Infatti, alcune operazioni necessarie per la costruzione e la risoluzione di un modello di ottimizzazione non possono essere rappresentate attraverso il paradigma dichiarativo dal linguaggio OPL standard.
Il linguaggio IBM ILOG script, che estende OPL, permette:

- **Preprocessamento dei dati di input:** invece di inserire i dati nel punto DAT, li assegniamo attraverso cicli for e cambiano nel corso delle soluzioni del programma;
- **Postprocessamento dei risultati:** possiamo stampare i dati, trasferirli all'esterno e darli in ingresso ad un altro programma;
- **Settaggio dei parametri di funzionamento di CPLEX;**
- **Gestire il flusso:** condizionare l'insieme delle operazioni per trovare le soluzioni.

Le righe dello script vengono eseguite l'una dopo l'altra.

BLOCCO MAIN

Il blocco *main* è il blocco principale (=viene eseguito per primo) e ci può essere un unico blocco *main* per ogni programma.
Le istruzioni che andiamo ad analizzare sono pressoché standard:

- var *source* = new *IloOplModelSource* ("LotSizing_Model.mod")*: crea una variabile oggetto che prende in ingresso il file "*LotSizing_Model.mod* che contiene il modello matematico del problema di ottimizzazione.

- var *def* = new *IloOplModelDefinition* (source): questa variabile prende in ingresso la variabile *source* precedentemente creata. Crea, quindi, un oggetto che può puntare ad un secondo oggetto che pounta ad un modello.
- var *opl* = new *IloOplModel* (def, cplex): prende in ingresso la variabile *def* che deve essere risolta attraverso Cplex. Così si crea una struttura del modello che può ricevere in ingresso il foglio dati. Da questo comando in poi, la variabile *opl* è legata a Cplex.

**IloOplModelSource viene anche definito dal prof "costruttore": è un modello di oggetti che hanno la stessa struttura. Va immaginato come una casella di memoria che punta (=puntatore) al file indicato.*

I comandi *IloOplModelSource, IloOplModelDefinition e new IloOplModel* rappresentano le classi.

Abbiamo così creato la struttura del modello che non è stata ancora istanziata. Ora dobbiamo aggiungere i dati sorgente con la medesima procedura:

- var *data* = new *IloOplDataSource* ("LotSizing_Data.dat")
- *opl.addDataSource* (data): agganciamo i data al modello.

A questo punto, per questo modello, rilassiamo il vincolo di interezza sulle variabili "z", facciamo *opl* quindi chiamiamo l'oggetto di tipo *ModelDefinition* che contiene l'istanza di cplex che risolverà il modello e l'aggancio alla definitiva al modello iniziale (modello di ottimizzazione) . utilizziamo un metodo che si chiama *opl.convertAllIntVars()*, che permette di convertire tutte le variabili intere di questo modello. All'interno delle parentesi non vi sono argomenti perché il metodo non ha, sostanzialmente, argomenti. Praticamente, quando noi chiamiamo il modello di ottimizzazione e applichiamo questo metodo, tutte le volte che quest'ultimo vede "Int" da qualche parte, lo rilassa e lo mette come di continuo.

A questo punto possiamo costruire il modello (per ora gli abbiamo detto come saranno fatte le istanze di questo modello) quindi generarlo:

- //Generazione del modello
  opl.generate( )

Tra le parentesi va messo qualcosa?
No, perché questo è un metodo della classe *IloOplModel,* che si può chiamare solo se è stato definito un oggetto di tipo *IloOplModel* e lui sa che, su questo tipo di oggetti può essere definita questa funzione, ovvero possono essere rilassate le variabili interne.
Se ci fosse una classe persone e in questa vi fosse un metodo saluta che stampa "ciao" allora si può scrivere: *massimiliano.saluta()* ovvero la Chiamata al metodo "saluta" dell'istanza dell'oggetto "Persona" che si chiama Massimiliano.

A questo punto abbiamo un modello in cui i vincoli non sono più vincoli di interezza "0,1" bensì un nuovo modello in cui le Z hanno questa struttura, ovvero possono assumere qualsiasi valore intero tra 0 e 1. È evidente che Z=0,5 dal punto di vista fisico non ha senso, non vi si può lavorare; è solo una costruzione per la risoluzione del problema.

- if (cplex.solve()) {
  //Creazione e generazione di una seconda istanza del modello

cplex= è scritto così poiché ci sarà un *cplex2* e agganciato *opl* che è un oggetto che contiene al suo interno il modello che, per essere risolto, deve aprire la chiamata a cplex cioè all' istanza di Cplex (scritto con la C grande) che prende il nome di cplex (con c piccolo) ovvero questa è l'istanza

dell'oggetto. Come c'è la classe "persone" (esempio precedente) vi è la classe "Cplex", in questo modo è stato creato un oggetto, di questa classe, che si chiama "cplex" (con la "c" piccola). Si fa solitamente così, infatti nel linguaggio della programmazione, le maiuscole e le minuscole hanno un significato particolare, solitamente:

- con la lettera maiuscola si dà il nome all'intera classe
- con la lettera minuscola si dà il nome alle istanze

ciò può creare un po' di confusione ma è importante comprendere che, ciò che abbiamo preso in considerazione, non è la classe "Cplex" bensì un'istanza di questa classe che è agganciata ad opl che è oggetto di *IloOplModel*.
"cplex.solve" scritto così restituisce 1 o 0, true o false, quindi entreremo nell'if esclusivamente se il modello iniziale, risolto con le variabili rilassate, ammette soluzioni. Se non è ammissibile non ha senso applicare l'euristica perché il problema con una regione ammissibile allargata non ammette soluzione, e, restringendola non potrà mai esistere una soluzione.
Qualora questa esista cosa bisogna fare?
Concettualmente bisogna costruire un secondo modello in cui le variabili non siano rilassate, all'interno del quale tutte le variabili che, in questa soluzione, hanno assunto valore 0,1 devono essere fissate.

Spiegazione a livello pratico (del prof): io ho rilassato le "z", risolvo il modello e scopro che, su 5 giorni, la prima vale "0,3" , "0,4", "1", "0" e "0,2"; allora devo creare un nuovo modello, uguale al precedente, in cui la prima, la seconda e l'ultima giornata non importino, terza e quarta però le voglio fissare, rispettivamente, su 1 e su 0. Ciò vuol dire che la prima fase ci restituisce due giornate in cui, spontaneamente, la soluzione è binaria, allora io fisso un nuovo modello in cui non vi sono rilassamenti in cui dico che queste variabili non le voglio considerare (so che in quei giorni devo lavorare) e lancio il modello in modo tale che lui mi mostri soltanto i tre rimanenti. **Questo è, concettualmente, ciò che bisogna fare.**

```
22    if (cplex.solve()) {
23    // Creazione e generazione di una seconda istanza del modello
24    var cplex2 = new IloCplex();
25    var opl2   = new IloOplModel(def, cplex2);
26    opl2.addDataSource(data);
27    opl2.generate();
28    }
```

- var *cplex2*=istanza della classe "Cplex" uguale alla precedente ma una copia (speculare) questo perché non si può modificare direttamente cplex (le soluzioni al suo interno ci servono). cplex2 permette poi di agganciare la stessa cosa, infatti equivale al dire "questa casella aggancia Massimiliano" o "questa casella aggancia Christian", la seconda non aggancia Massimiliano, quindi l'informazione Christian viene cancellata e non può più essere recuperata e si hanno due istanze che si associano ad una persona, che in questo caso è il modello, quindi non puoi andare a prendere le variabili intere perché poi ne hai bisogno, e ti serve averne due per passare le soluzioni dall'una all'altra.

Questa è la prima volta in cui vediamo la classe delle istanze di cplex , con entrambi gli elementi come oggetti della classe (es Christian e Massimiliano).
cplex2 per ora è vuoto perché non sa quale modello risolvere, sa che c'è, da qualche parte, un'istanza che può essere collegata ad un modello di ottimizzazione.

- var opl2 = new IloOplModel(def, cplex2): il modello è lo stesso perché *def* aggancia la definizione del modello che a sua volta aggancia 16, 11, 23 etc però questa volta non è agganciato alla stessa istanza cplex di prima bensì ad una seconda. Quindi io ho il primo modello con le variabili rilassate attaccate a cplex e un secondo modello con le variabili non rilassate agganciate a cplex2; quest'ultimo non sa quali sono i dati ma noi li abbiamo già collegati ad una variabile, ovvero il foglio.dati agganciato a "Data", quindi lo possiamo utilizzare.

- Opl2.addDataSource(data)
  Quindi dopo opl2, su quest'oggetto di tipo iloOplModel chiamiamo il metodo di aggiunta del foglio dati che è *addDataSource* e agganciamo anche qui Data.

Quello che ci manca ora è generare, per l'appunto, il modello:
- Opl2.generate();
  }

Ora che abbiamo entrambi i modelli. Del primo, avendo utilizzato questo metodo, abbiamo già la soluzione, se esiste. Quando arriva a questo livello (alla fine del primo modello) lui creerà questo oggetto solo se conosce la soluzione.

A questo punto bisogna capire quali siano le variabili z_t che nel modello rilassato assumono valore intero e dobbiamo fissarle nel modello non rilassato; ovvero dobbiamo scorrere la soluzione che abbiamo già generato nello specifico il vettore z_t nella soluzione che è stata generata in *cplex.solve* agganciato alla configurazione del modello che ha le variabili rilassate, tutte le volte che troviamo delle variabili che assumono spontaneamente valore 0 o 1 dobbiamo fissarle nel secondo modello che non ha le variabili rilassate perché bisogna avere alla fine una soluzione ammissibile.
- *for (var t=1; t<= opl.dataElements.numPeriodi; t++) {*
  t =1 quindi "t" è una variabile locale che ci serve per scorrere;
  t<= di **opl** (primo modello)**.dataElements**(bisogna immaginare come una freccia che, addentrandosi, dall'oggetto punta al data Elements riconfigurato) e da questo ci serve prendere il "numero periodi di tempo" che abbiamo chiamato **.numPeriodi**. Quindi qui, la specifica istanza, sarà un numero pari a 20, perché, ricordiamo `7 numPeriodi = 20;`

digressione: t++ corrisponde a t=t+1 → incremento +1. Essendo il for un ciclo, ad ogni iterazione "t", che è la variabile di ciclo, cresce di 1

A questo punto cosa deve controllare?
Se sta scorrendo le "t"; se è vero che opl.z[t] (del primo modello) sia uguale precisamente a 1
Quindi ci colleghiamo ad opl, che è la prima versione del modello, e prendiamo z_t (che è un vettore alla t-esima componente) e utilizziamo un metodo "solution"
Opl contiene al suo interno un vettore "z" che è indicizzato in "t" e cresce ogni volta di 1 unità.
Il vettore Entra in Opl (primo modello), entra nel vettore z alla posizione "t" e di questo deve prendere il valore di soluzione e *deve chiedersi* se sia uguale ad 1; oppure chiedersi se lo stesso valore sia uguale a 0:
- *if (opl.z[t].solutionValue = = 0) {*

se una di queste due condizione è vera (essendo *or* è verificato se almeno una delle due è pari a 1; o è 1 o è 0, non saranno mai verificate entrambe) allora **opl2** che è il secondo modello e che ha le variabili intere, **.z[t]** si scrive:

```
2    opl2.z[t].LB = opl.z[t].solutionValue;
3    opl2.z[t].UB = opl.z[t].solutionValue;
4   }
5   else if (opl.z[t].solutionValue > 0.5){
6      opl2.z[t].LB = 1;
7      opl2.z[t].UB = 1;
```

abbiamo scoperto che, **.z[t]**, in un dato giorno, vale 1 in maniera spontanea, ovvero risolvendo il problema senza vincolarlo ad assumere questo valore, assume valore 1, *per ipotesi*. Se questo è vero, entra nel secondo modello, quello in cui le variabili non sono rilassate, e poni il suo lower bound pari ad 1 (restituisce 1 e 0) e restituisce i valori di soluzione.

Lower bound significato : quando si scrive un vincolo A>= B; B= Lower Bound, ovvero il valore al di sopra del quale quell'espressione deve stare. (tipico delle disequazioni)
Es: A<=5 e A>=3 A deve stare tra 3 e 5 e il suo estremo inferiore (3) è il Lower Bound (l'estremo superiore è indicato come Upper Bound)
Il comando che implementa il lower bound si chiama "Lower Bound".
Ciò che è scritto nel codice esprime l'indicazione: entra nel modello opl2, prendi z[t], ovvero prendi la variabile decisionale z in posizione t dove t è variabile da 1 a 20, per la quale hai scoperto che il valore vale 0 o 1 e poni:

- il suo Lower Bound (supponiamo sia 0) a 0, e anche l'Upper Bound a 0→ la soluzione, dovendo stare tra 0 e 0, corrisponderà a 0
- il suo Lower Bound (supponiamo sia 1) a 1, e anche l'Upper Bound a 1→ la soluzione, dovendo stare tra 1 e 1, corrisponderà a 1

in questo modo abbiamo detto al programma:

- esplora la soluzione del problema con i valori rilassati
- identifica tutti i valori di z che assumono spontaneamente il valore 0,1
- per queste variabili, nel problema non rilassato, poni il LB e l'UB pari a 1, in modo tale che la soluzione sia necessariamente 1
- se ciò non è vero, ovvero il valore non corrisponde ne a 0 ne a 1, bensì vale 0,5, allora non entra e non cambia nulla

A questo punto dobbiamo farci stampare delle cose. *Come si fa?*
Se un main o un blocco che si chiama "execute" introducono o il pre processamento o il post processamento dei dati, questi possono essere stampati:

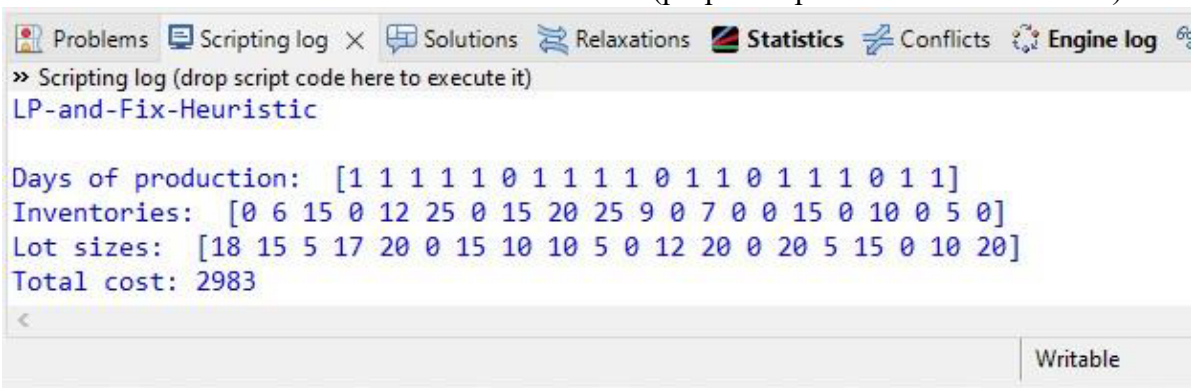
```
39      }
40
41      if (cplex2.solve()) {
42        writeln ("LP-and-Fix-Heuristic");
43        writeln ();
44        writeln ("Days of production: " + opl2.z.solutionValue);
45        writeln ("Inventories: "        + opl2.i.solutionValue);
46        writeln ("Lot sizes: "          + opl2.q.solutionValue);
47        writeln ("Total cost: "         + cplex2.getObjValue());
48      }
49  }
```

Attributi:
z: variabile decisionale
i: quantità che viene stoccata all'inventario
q: quantità che viene prodotta

ci serve il valore della funzione obiettivo:
cplex2.getObjValue())

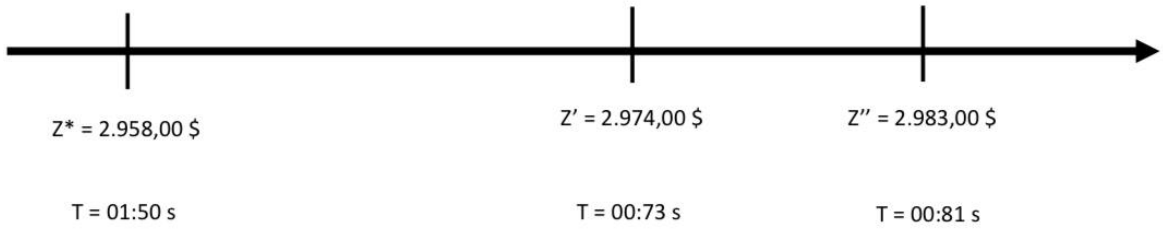
Notiamo che non abbiamo dovuto fare il *for*, infatti lo stampa direttamente come vettore, altrimenti dovremmo fare il *for* su ogni elemento del vettore e stamparli uno ad uno, questo semplifica ulteriormente la risoluzione.
Non è presente la doppia parentesi perché, gli oggetti di una classe, sono dotati di attributi e di metodi; gli attributi sono caratteristiche statiche (es nome, cognome etc) e l'accesso ad essi avviene con il dot (punto) e a seguire vi è proprio il nome dell'attributo.
La doppia parentesi invece indica che è definita la funzione all'interno della classe, che ha un certo valore restituito.

L'euristica va attaccata sotto al file con il modello (proprio copia e incolla sul file mod).



L'ultima cosa che vediamo è che la soluzione ottima si presenta così ed è generata in 1secondo e 50 e corrisponde a Z*= 2958 (costo)
Se la risolviamo come euristica, come quella che abbiamo implementato, il tempo diminuisce ma il costo di funzione obiettivo solitamente cresce; la soluzione ovviamente non è ottimale.

IBM ILOG Script: implementare una euristica



Qual è la configurazione migliore?
Provando tutte le configurazioni, ci si metterebbe un tempo illimitato. Un'euristica permette di impiegare meno tempo.

Seconda versione di esecuzione (il prof l'ha corretta direttamente sulle slides): rilassiamo il modello, dopodiché, anziché fissare tutte le variabili a 0 e ad 1 se hanno spontaneamente assunto valore 0,1, fissiamo a 0 tutte le variabili che assumono spontaneamente valore 0 e ad 1 tutte quelle che, nella soluzione rilassata, assumono valore strettamente maggiore di 0,5; ciò si traduce in tutto ciò visto in precedenza seguito dal *for* e scoviamo le variabili come fatto in precedenza: ci chiediamo se il rilassato in z_t ha valore 0, e se è così assume il valore di 0 e viceversa ci chiediamo se la soluzione ottima del problema rilassato in z_t assuma valore strettamente maggiore di 0,5 , se è così lo fissiamo a 1. Quindi aggiungiamo un blocco *else if* successivamente al blocco *if*

```
35      else if (opl.z[t].solutionValue > 0.5){
36         opl2.z[t].LB = 1;
37         opl2.z[t].UB = 1;
38      }
39     }
```

Se risolviamo anche questo (seconda istanza del problema) migliora l'euristica poiché, anziché fissare a 0 o 1 le variabili che spontaneamente hanno assunto valore 0 o 1, **fissa a 0 quelle che spontaneamente hanno assunto valore 0 e a 1 tutte quelle che hanno assunto valore continuo >= 0,5**
Prima era presente ***l'or*** (o 0 o 1)
In un'euristica importa quanto tempo ci metta e quanto sia "buona" la soluzione.

Come facciamo a capire quando una soluzione è buona (accett.)?
Ovviamente non si ha l'ottimo come riferimento perché, per generarlo, ci vorrebbe troppo tempo.
Per capire se la soluzione è buona:

- si prende il problema
- si tolgono dei vincoli
- si ottiene un problema semplice

Quest'ultimo sarà migliore (dal punto di vista di tempistiche) per poter calcolare la soluzione.
Infine si valuta quanto sia la differenza tra una soluzione e l'altra. Se lo scarto è piccolo (in percentuale) la soluzione è buona, altrimenti no.
Se lo scarto è al più lontano il 5% dalla soluzione ottima, ad esempio, è accettabile.