Problema del Lot-Sizing: specifica

Il problema che è stato analizzato viene chiamato Lot-Sizing. Si tratta di un problema di *production planning* a una macchina.
I seguenti dati vengono forniti per ogni istanza:

- Un insieme $I = \{0, \ldots, m-1\}$ che rappresenta i tipi di oggetto che possono essere prodotti.
- Un insieme $P = \{0, \ldots, p-1\}$ che rappresenta i periodi temporali (discreti) durante i quali produciamo un oggetto.
- Una matrice $I \times P$ (la chiamo D) tale che $d_t^i = 1$ se un oggetto di tipo i è richiesto al tempo p, $d_t^i = 0$ altrimenti.
- Una matrice $I \times I$ (la chiamo C) tale che c^{ij} rappresenta il costo che va pagato quando la macchina passa dalla produzione di un oggetto di tipo i ad uno di tipo j ($\forall i \; c^{ii} = 0$).
- Un array H di lunghezza m tale che h^i rappresenta il costo dato dal tenere in magazzino un oggetto di tipo i per un periodo (una unità temporale).

Assumendo di dover produrre esattamente un oggetto per ogni periodo, il problema consiste nel determinare quale oggetto vada prodotto dalla macchina in ciascun periodo di tempo. La soluzione quindi può essere rappresentata tramite un array V di lunghezza p, dove ogni V^t rappresenterà il tipo di oggetto da produrre al tempo t.

Si ha un unico constraint di tipo hard che possiamo chiamare **NoBacklog**: ogni oggetto deve essere prodotto non più tardi rispetto a quando è richiesto. In altre parole, per ogni oggetto i e ogni periodo t, il numero di oggetti i prodotti fino al tempo t deve essere maggiore o uguale al numero totale di richieste di i fino al tempo t.

La **funzione obiettivo** da minimizzare è costituita da due elementi:

- **StockingCost**: è la somma dei costi di stoccaggio di tutti gli oggetti. Questi costi vengono generati quando la macchina produce un oggetto al tempo t ma l'oggetto è richiesto solamente "dopo" (ad un tempo $t' > t$).

- **SetupCost**: è la somma dei costi di setup della macchina: vengono generati ogni volta che questa passa dal produrre un tipo di oggetto ad uno diverso. Il primo oggetto prodotto genera un SetupCost nullo.

Riporto qui di seguito una piccola istanza esemplificativa in formato MiniZinc:

```
Periods = 8;
Items = 3;
Demands = [|0, 0, 0, 0, 0, 1, 0, 1
           |0, 0, 1, 1, 0, 0, 1, 1
           |0, 0, 0, 0, 0, 1, 1, 0|];
StockingCosts = [10, 15, 12];
SetupCosts = [|0, 131, 109
              |193, 0, 175
              |101, 136, 0|];
```

con la corrispondente soluzione ottima:

```
[1, 1, 2, 2, 0, 0, 1, 1]
```

di costo 569.

Il modello MiniZinc

Riporto qui il codice diviso in parti, al termine di ogni frammento è presente un commento al codice.

```
include "globals.mzn";

int: Periods;
int: Items;
array[1..Items,1..Periods] of 0..1: Demands;
array[1..Items] of int: StockingCosts;
array[1..Items,1..Items] of int: SetupCosts;
```

Qui vengono inizializzate le strutture che servono a modellare l'input del problema. Inoltre è presente l'istruzione include che permetterà poi di utilizzare il vincolo globale *exactly*.

```
array[1..Items,1..Periods] of var 0..1: SolMatrix;        % Solution matrix
array[1..Periods] of var 0..Items-1: Sol;                 % Output array
```

$SolMatrix$ è una matrice $item \times period$ di "0" e "1". s_p^i è uguale a 1 se e solo se nel periodo p produco un item di tipo i. Sol è un array di dimensione uguale al numero di periodi ed esprime, per ogni periodo, quale tipo di item viene prodotto.

```
% Schedule the production of exactly one item for each period
constraint
  forall (t in 1..Periods)
    (exactly(1,[SolMatrix[i,t] | i in 1..Items],1));
```

Questo primo insieme di vincoli costringe ad assegnare per ogni periodo di tempo esattamente un tipo di item da produrre. Ho fatto alcune prove esprimendo il vincolo in altri modi (ad esempio, vincolare la somma degli elementi di ogni colonna ad essere uguale a 1) ma i tempi erano sempre peggiori rispetto alla versione con l'*exactly*.

```
% Every item must be produced before the time for which it's requested
constraint
  forall (i in 1..Items, j in 1..Periods)
    (sum (k in 1..j) (Demands[i,k]) <= sum (k in 1..j) (SolMatrix[i,k]));
```

Questi vincoli esprimono **NoBacklog**, cioè la non possibilità di produrre un item in un periodo successivo a quello per il quale è richiesto.

```
% This constraint "links" SolMatrix and Sol
constraint
  forall(t in 1..Periods)
    (sum (i in 1..Items)(i*SolMatrix[i,t]) == Sol[t]+1);
```

Qui $SolMatrix$ e Sol vengono "collegati" per far corrispondere alla matrice l'array che sarà poi l'effettiva soluzione data in output.
Sono state mantenute entrambe le strutture (anche se ridondanti) perchè lavorare solo utilizzando la matrice rallentava di molto il calcolo del valore della funzione obiettivo, mentre utilizzare solo l'array rendeva più lento il lavoro fatto dai primi due insiemi di vincoli presentati precedentemente. Utilizzando entrambe le strutture ho ottenuto i risultati migliori.

```
% Minimize the stocking costs (keeping an already produced item stored)
%  and
%  the setup costs (switching from producing a certain type of item
%    to another)
solve:: int_search([SolMatrix[i,j] | i in 1..Items, j in 1..Periods],
                   occurrence, indomain_min, complete)
minimize (
  sum (i in 1..Items)
    (StockingCosts[i] * sum(t in 1..Periods, j in 1..t)
                          (SolMatrix[i,j]-Demands[i,j]))
  +
  sum (i in 1..Periods-1) (SetupCosts[Sol[i]+1,Sol[i+1]+1])
);
```

Qui sono presenti le annotazioni per la ricerca e la funzione obiettivo:
La ricerca parte assegnando valori alla matrice $SolMatrix$ (altre possibilità sono state testate e sono risultate sensibilmente peggiori). L'euristica per la variable selection è *occurrence*, viene scelta la variabile più vincolata.

Anche qui è stato visto sperimentalmente che è la scelta migliore. Per quanto riguarda value choice, *indomain_min* è chiaramente la migliore visto che la mia soluzione corrisponderà ad una matrice nella quale la maggioranza dei valori è 0.
La funzione da minimizzare è la somma degli *stockingcosts* e dei *setupcosts*. Qui si nota anche l'utilità di avere sia $SolMatrix$ che Sol.

```
output ["Cost: "++show (
  sum (i in 1..Items)
    (StockingCosts[i] * sum(t in 1..Periods, j in 1..t)
                          (SolMatrix[i,j]-Demands[i,j]))
  +
  sum (i in 1..Periods-1) (SetupCosts[Sol[i]+1,Sol[i+1]+1])
)++" "]
++
[show(Sol)]
```

L'output è composto dal costo della soluzione trovata e dall'array che esprime la soluzione.

La codifica ASP

Riporto qui di seguito la codifica dell'istanza riportata a pagina 2:

```
periods (1..8).
items (1..3).
demand (2,3).
demand (2,4).
demand (1,6).
demand (3,6).
demand (2,7).
demand (3,7).
demand (1,8).
demand (2,8).

stocking_cost (1,10).
stocking_cost (2,15).
stocking_cost (3,12).

setup_cost (X,X,0) :- items(X).
setup_cost (1,2,131).
setup_cost (1,3,109).
setup_cost (2,1,193).
setup_cost (2,3,175).
setup_cost (3,1,101).
setup_cost (3,2,136).
```

La prima dichiarazione di *setup_cost* fa in modo di non avere costi di setup quando la macchina produce un altro *item* dello stesso tipo che aveva prodotto in precedenza (si potrebbe fare a meno di questa dichiarazione).

```
1 { assign(P,I) : items(I) } 1 :- periods(P).
```

Qui vincolo *assign* ad essere una funzione che assegna ad ogni periodo esattamente un *item*. Corrisponde al vettore *Sol* del modello MiniZinc.

```
sum_prod(I,P,N) :- N = #count{ P1 : assign(P1,I), periods(P1), P1<=P },
                   items(I), periods(P).

sum_dem(I,P,N) :- N = #count{ P1 : demand(I,P1), periods(P1), P1<=P },
                  items(I), periods(P).

:- sum_dem(I,P,N1), sum_prod(I,P,N2), N1>N2, items(I), periods(P).
```

Questo frammento esprime i vincoli hard del problema.
sum_prod mi dice per ogni coppia *I-P* quanti item di tipo I sono stati prodotti fino al periodo P. *sum_dem* fa la stessa cosa ma invece di parlare degli item prodotti parla degli item richiesti fino al periodo P.
L'ultima riga è il vincolo vero è proprio: assicura che per ogni coppia *item-period* il numero di item di un certo tipo richiesti non superi il numero di quelli prodotti.

```
stock (I,P,N) :- N = #sum{ M : sum_dem(I,P,N1), sum_prod(I,P,N2),
                           stocking_cost(I,C), M = C*(N2-N1) },
                 items(I), periods(P).

sum_stock (N) :- N = #sum{ M,I,P : stock(I,P,M) }.
```

stock conta per ogni periodo il numero di item di un certo tipo che sono stati prodotti ma per i quali non c'è stata ancora la corrispondente richiesta (non sono stati "consumati"). A questo punto moltiplica questi valori per i costi di stoccaggio dei singoli item.
sum_stock fa semplicemente la somma di tutti i costi calcolati precedentemente.

```
setup (P,N) :- N = #sum{ M : assign(P,I1), assign(P+1,I2),
                         setup_cost(I1,I2,C), M = C },
                   periods(P).

sum_setup (N) :- N = #sum{ M,P : setup(P,M) }.
```

setup serve a calcolare i costi di setup per ognuno dei periodi, mentre *sum_setup* fa la somma di questi valori.

```
sum_total (N) :- N = N1+N2, sum_stock(N1), sum_setup(N2).

#minimize { N : sum_total(N) }.
```

Qui sommo i costi di stocking con quelli di setup, per poi minimizzare tale somma.

```
#show assign/2.
#show sum_total/1.
```

Stampo in output l'oggetto da produrre ad ogni periodo (*assign*) e il costo di tale soluzione (*sum_total*).

Test

Nella seguente tabella ho riportato nella prima colonna il nome dell'istanza, nelle altre due il tempo richiesto da MiniZinc (Gecode) e da Clingo:

Istanza	MiniZinc	Clingo
8-1	13ms	8.147s
8-2	15ms	2.000s
8-3	15ms	7.166s
8-4	16ms	38.829s
8-5	16ms	76.192s
8-6	16ms	16.147s
8-7	16ms	20.703s
8-8	18ms	4.743s
10-1	15ms	4.202s
10-2	16ms	113.604s
10-3	16ms	153.553s
10-4	15ms	46.374s
10-5	16ms	15.204s
10-6	17ms	115.447s
10-7	16ms	169.772s
10-8	18ms	49.704s
20	938ms	-
25	47.270s	-
30	79.000s	-

I nomi delle istanze corrispondono al numero di periodi presenti nell'istanza, ho testato 8 istanze contenenti 8 periodi, 8 da 10 e 3 istanze più grandi che ho fatto girare solo su MiniZinc in quanto Clingo richiedeva tempi superiori ai 5 minuti.
Il numero di test effettuati è ridotto perchè mi sono reso conto che questo tipo di modelli non viene risolti velocemente da Clingo. Questo problema è dato fondamentalmente dagli stocking costs e dall'espressione che si usa nel modello per minimizzarli (l'espressione è la seguente: $M = C * (N2 - N1)$ dove $(N2 - N1)$ è la quantità di item in stock e C lo stocking cost di un singolo item).
Una cosa che si nota è che quando gli stocking cost sono uguali (come nel caso dell'istanza 10-1) i tempi sono più bassi. Facendo qualche test più

approfondito (usando il comando `--text`) si nota che il *grounding* ha bisogno di molto tempo perchè deve istanziare un numero molto alto di "stock", di conseguenza si ha un numero elevatissimo di "sum_total" (per l'istanza 8-1, più di 500.000). I costi di setup invece non sembrano essere un problema al pari degli stocking costs.

Una possibile soluzione è quella di usare *Clingcon*, un answer set solver per programmi a vincoli. Si tratta di un'estensione che aggiunge ai tradizionali programmi logici la capacità di lavorare su vincoli in modo "effettivo".
Questo software è stato sviluppato a partire da *Clingo* e *Gecode*