Steel mill slab design

© Copyright IBM Corp. 2014. All Rights Reserved. U.S. Government Users Restricted Rights - Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.
IBM®, ILOG®, and CPLEX® are trademarks or registered trademarks of International Business Machines in the US and/or other countries. Other company, product, or service names may be trademarks or service marks of others.

Steel mill slab design

How can a set of orders be produced while respecting production rules and minimizing the waste material?

A rolling mill has to produce a set of steel coils, each of a particular predefined weight, from a number of slabs of raw material. The goal is to produce the coils from the slabs while minimizing the quantity of material wasted.

A single coil cannot be produced from more than one slab, but in theory any number of coils can be produced from a single slab. Of course, the weight of a slab must be no less than the sum of the weights of the coils produced from it. Any additional steel left over from a slab after the production of the coils is considered "lost". The total loss has to be minimized.

Each coil is subject to one of a number of specific finishing processes, which is referred to as the "color" of the coil. Due to restrictions on the factory floor, any one slab can only be used to produce coils of at most two colors.

Slabs come in a number of standard (discrete) weights, and in this problem it is assumed there is no issue with stock: that is, an indefinite number of slabs of each weight is available to produce the coils.

This problem is solved using the constraint programming engine CP Optimizer. Constraint programming problems generally have discrete decision variables, and they may include nonlinear and logical constraints.

The data (`steelmill.dat`)

For DropSolve, all data must be defined in the form of tables (which in OPL implies using sets of tuples to define these tables).

In the steel mill problem, the data includes the number of orders together with the weight and the color of each order. In addition, the data includes the set of available slab weights. All weights are integer. The maximum number of slabs to be used is obviously no more than the number of orders.

```
Params =
<
  12, // max nb slabs
   2   // max colors per slab
>;

Orders =
{
    <1, 22, 5>
  , <2,  9, 3>
  , <3,  9, 4>
  , <4,  8, 5>
  , <5,  8, 7>
  , <6,  6, 3>
  , <7,  5, 6>
  , <8,  3, 0>
  , <9,  3, 2>
  , <10, 3, 3>
  , <11, 2, 1>
  , <12, 2, 5>
};

// A slab of size 0 represents no slab
slabWeights = { 0, 11, 13, 16, 17, 19, 20, 23, 24, 25,
               26, 27, 28, 29, 30, 33, 34, 40, 43, 45 };
```

Note that zero is a possible slab weight. It is not known in advance how many slabs will need to be used - using less than all of the possible slabs – 12 in this case – can be represented by the unused slabs having zero weight. This notion should become clearer in the description of the decision variables.

The model (`steelmill.mod`)

Certain useful derived structures are precomputed from the input data: this allows the model to be written in a more compact form, avoiding repetition. First, it is useful to represent the indexation of orders and slabs by ranges:

```
int nbOrders = card(Orders);
range orderIndexRange = 0..nbOrders-1;

int nbSlabs = Params.maxNbSlabs;
range Slabs = 1..nbSlabs;
```

Second, the set of all possible colors is produced from the color of each order as follows:

```
 {int} allcolors = union(o in Orders) { o.color};
```

Finally, the maximum weight of a slab is calculated:

```
   int maxSlabWeight = max (sw in slabWeights) sw;
```

Modeling the variables and expressions

There is a decision variable associated with each order. This decision variable indicates from which slab the order will be manufactured. The domain of each of these variables is the set of possible slabs, that is, from 1 to `nbSlabs`. This formulation makes the assumption that the selected slabs have a placement: in other words, there is the notion of a first slab, a second slab, and so on. There is also a decision variable associated with each slab. This decision variable indicates the amount of material used from the slab. That is, the sum of the weights of the coils manufactured from the slab.

```
  dvar int productionSlab[Orders] in Slabs;
  dvar int slabUse[Slabs] in 0..maxSlabWeight;
```

Modeling the constraints

One set of constraints is that no slab can produce orders that together represent more than two colors. This is achieved by using the `or` construct over each color to determine whether a particular slab is used to produce a nonzero number of orders of that color, and then limiting the sum of these `or` constructs to 2. There is an implicit interpretation of a Boolean expression as a 0-1 integer, so this sum limits the number of `true` Boolean expressions in the `or` to 2.

```
  forall (s in Slabs) {
    colorCt:
      // At most 2 colors per slab
      sum (c in allcolors) (
        or (o in Orders : o.color == c) (productionSlab[o] == s)
      ) <= Params.maxColorsPerSlab;
  }
```

For each slab, its use must be equal to the sum of the weights of orders that are manufactured from it. The specialized `pack` constraint links the `productionSlab` and `slabUse` variables. In general terms, the `pack` constraint maintains the load of a set of containers, given a set of weighted items and an assignment of items to containers.

```
  packCt:
    // The orders are allocated to the slabs with capacity
    pack(slabUse, productionSlab, orderWeights);
```

Modeling the objective

The objective is to minimize loss, where the loss is the material in a slab not used for producing orders. Loss can be nonzero because slabs only come in discrete weights, so to produce a certain total order weight from one slab, the lightest slab that is heavy enough to accommodate all the orders is chosen. This slab may be strictly heavier than that the total order weight itself.

To simplify the objective function, the minimum loss is calculated for all slab usage values in the range `[0, maxSlabWeight]`. The calculation of minimum loss is based on the weights of the available slabs. For example, with the data presented here, the loss for a slab use of 31 is 2, since the lightest slab with weight at least 31 has weight 33.

```
  int loss[use in 0..maxSlabWeight] =
          min (sw in slabWeights : sw >= use) (sw - use);
```

The objective is then to minimize the total loss.

```
  dexpr int totalLoss = sum (s in Slabs) loss[slabUse[s]];
```

The solution

To solve this problem, a search phase is used. Here, a useful strategy is to first decide on the production slab for each order, rather than concentrating on choosing the slab use. Once all the decision variables in the `productionSlab` array have been given values, the values for all of the `slabUse` variables will have been determined through constraint propagation.

To ask the search to concentrate on the `productionSlab` variables, a `searchPhase` object is created. This object takes the array of `productionSlab` variables as its argument.

```
  var f = cp.factory;
  cp.setSearchPhases(f.searchPhase(productionSlab));
```

The model file contains declarations of sets to create output data which will be provided after the solution has been obtained. These sets provide the set of orders manufactured from each slab and the set of colors associated with each slab.

```
{int} fromSlab[s in Slabs] =
{ o.index | o in Orders : productionSlab[o] == s };

{int} slabColors[s in Slabs] = { o.color | o in Orders : o.index in fromSlab[s] };
```

A postprocessing script displays the result

```
execute {
  for (s in Slabs) {
    if (Opl.card(fromSlab[s]) > 0) {
      write("Slab " + s + ", Loss = " + loss[slabUse[s]]
          + ", colors =" + slabColors[s] + ", Orders =");
      for (o in fromSlab[s]) {
        write(" " + o);
      }
      writeln();
    }
  }
}
```

In this problem, five or more nonzero weight slabs are used, and the orders are distributed such that there is no loss. For example, the following is a possible optimal solution, as found in the Results file:

```
// solution with objective 0
productionSlab = [4 2 8 12 5 7 7 12 4 5 2 8];
slabUse = [0 11 0 25 11 0 11 11 0 0 0 11];

fromSlab = [{} {2 11} {} {1 9} {5 10} {} {6 7} {3 12} {} {} {} {4 8}];
slabColors = [{} {3 1} {} {5 2} {7 3} {} {3 6} {4 5} {} {} {} {5 0}];
```

The Log file from Dropsolve provides the results of the post-processing script:

```
Slab 2, Loss = 0, colors = {3 1}, Orders = 2 11
Slab 4, Loss = 0, colors = {5 2}, Orders = 1 9
Slab 5, Loss = 0, colors = {7 3}, Orders = 5 10
Slab 7, Loss = 0, colors = {3 6}, Orders = 6 7
Slab 8, Loss = 0, colors = {4 5}, Orders = 3 12
Slab 12, Loss = 0, colors = {5 0}, Orders = 4 8
```

Possible modifications

One possible modification is to increase the number of orders. Another more challenging modification is to add a fixed cost to each slab used. Finally, all slabs indices can be considered equivalent: any symmetries can be broken by making sure that if k slabs are used, then only slabs 1 to k have nonzero weight.