An Efficient Model and Strategy for the Steel Mill Slab Design Problem

Antoine Gargani[1] and Philippe Refalo[2]

[1] ILOG, 9, rue de Verdun, BP85,
94253 Gentilly Cedex, France
agargani@ilog.fr
[2] ILOG, Les Taissounières, 1681, route des Dolines,
06560 Sophia-Antipolis, France
refalo@ilog.fr

Abstract. The steel mill slab design problem from the CSPLIB is real-life problem from the steel industry. Finding optimal solutions to this problem is difficult. Existing constraint programming approaches can solve problems up to 30 orders. We propose a strong constraint programming model based on logical and global constraints. By designing a specific strategy for variable and value selection, we are able to solve instances having more than 70 orders to optimality using depth-first search. Injecting this strategy into a large neighborhood search, we are able to solve the real-life instance of the CSPLIB having 111 orders in just 3 seconds.

1 Introduction

The steel mill slab design problem (referenced in the CSP library[1] as problem 38) is difficult to solve to optimality. This problem arises from operations planning in the process industry. The problem consists of packing a set of orders onto slabs so as to minimize the total capacity of the slabs needed to fulfill the order book. In practice, two constraints must be satisfied. First, the total weight of the orders assigned to a slab cannot exceed the slab weight. Second, there is a route specification associated with each order represented by the *color* of the order. Packing different colors on a slab involves cutting the slab in different pieces. The cutting machine being the bottleneck of production line, the number of allowed cuttings must not exceed one, and thus the number of different colors on a slab must not exceed two.

The slab design problem is the second step of a more general problem that optimizes the process of orders in a steel mill (see [6]). Before designing and producing slabs, the orders are matched with a slab surplus inventory. Some orders may be assigned to existing slabs, and thus only the orders that were not matched are used in the slab design problem. This inventory matching problem has some similarities with the slab design problem, but the number of available

[1] CSPLIB problems are available at `http://www.csplib.org`.

slabs is limited, and a cost for non-packed orders must be handled. This inventory matching problem was addressed with integer programming techniques and in particular by developing approximation algorithms [1] More recently, better results were found using a column generation approach [2].

In the slab design problem, there is no limit on the number of available slabs, and all orders must be packed. This problem, described in the CSP library, has been addressed with constraint programming techniques. In the CSPLIB, an instance with 111 orders is available that so far could not be solved to optimality by constraint programming approaches. A study of different models has been presented in [3], and the role of symmetries has been discussed and experienced in [4]. A hybrid approach combining constraint programming and linear relaxations described in [5] gave the best results and could solve an instance with 30 orders (a subinstance of the 111 orders instance) in about 1000s. Local search techniques were also used in [7]. In this report, the local search solver for pseudo booleans WSAT(OIP) [12] is applied to the decision problem where the cost function is forced to the lower bound of the problem (the sum of order weights). A solution to the 111 instance could be found in about 2000s.

The models used for constraint programming approaches to this problem were basically linear models over binary variables. While such models are suited for integer programming solvers that can tighten the formulation by cutting-plane generation, these models are notoriously not well suited to a constraint programming approach because of the limited domain reductions they produce.

We introduce in this article a strong constraint programming model based on logical and global constraints that achieves more domain reduction. This new model is simple and elegant; it does not contain binary variables but exploits the structure of the problem. By designing a specific strategy for variable and value selection, we are able to use depth-first search to solve instances having more than 70 orders to optimality in less than 200s. We have used this strategy in a large neighborhood search, and we are able to solve the largest instance with 111 orders in just 3s.

2 Problem Description

The problem consists in producing n orders from a set of slabs. Several orders can be made from the same slab but there is no limitation on the number of slabs that can be requested. Each order o has a color c_o and requires an amount of capacity (weight) w_o of the slab to which it is assigned. Each slab has a weight that must be chosen from the increasing set of weights $\{u_1, u_2, \ldots, u_k\}$. The constraints of the problem are

1. an order must be produced from a single slab, and
2. the sum of order weights on a slab must not exceed the slab weight, and
3. a slab can be used to produce orders of at most two different colors.

The first two constraint describe a bin-packing problem. The third one is called the color constraint. Therefore this problem is also called a *variable sized* *bin-packing problem with color constraints* in the literature [1]. In the steel mill slab design problem, the objective is to produce a few slabs as possible to satisfy the demande. More precisely, the objective is to minimize the cumulative sum of the weights of the slabs used. An obvious lower bound to this problem is the sum of order weights.

Example 1. *Here is a small, illustrative example of an instance and of a solution. Assume that we have 10 orders whose weights and colors are*

Order	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*
Weight	*1*	*3*	*2*	*9*	*9*	*11*	*3*	*3*	*5*	*2*
Color	*Red*	*Black*	*Black*	*Red*	*Red*	*White*	*Red*	*White*	*Black*	*Red*

and let the set of possible slab weight be $\{5, 7, 9, 11, 15, 18\}$*. A solution to this problem is to use 4 slabs and assign the orders in the following way:*

Slab	*Orders*	*Weight sum*	*Slab weight*
1	*1, 2, 3*	*6*	*7*
2	*4, 5*	*9*	*9*
3	*6, 7, 8*	*17*	*18*
4	*9, 10*	*7*	*7*

Note that in this solution

– *there are no more than two different colors on the same slab;*
– *the maximum slab weight is not exceeded;*
– *the slab weight is just large enough for producing the orders;*
– *the cost of this solution (the sum of the slab weights) is 41;*
– *a lower bound is the sum of order weights that is 39.*

3 A Basic Model

The basic model described here is very similar to the model used for a constraint programming solver in [4]. The model uses primarily binary variables.

Assume that O is the set of orders and S is the set of slabs. Since there is no limitation on slab weight, we artificialy create as many slabs as orders and $|S| = |O|$. Let C bet the set of colors of orders in O and let Q be the set of slab weights $Q = \{u_0 = 0, u_1, u_2, \dots, u_k\}$. The value 0 introduced in the set is the weight of unused slabs.

The variables of the problem are binary variables. The first matrix of variables determines the positions of the orders:

$$x_{os} \in \{0, 1\} \text{ for } o \in O, s \in M$$

We have $x_{os} = 1$ when order o is packed onto the slab s. The second matrix of variables determines the weight of a slab:

$$y_{sq} \in \{0, 1\} \text{ for } s \in O, q \in Q$$

We have $y_{sq} = 1$ when the slab s has weight q. The third matrix of variables is related to colors:

$$z_{cs} \in \{0, 1\} \text{ for } c \in C, s \in S$$

We have $z_{cs} = 1$ when at least one order of color c is assigned to slab s.

The first constraint of the problem states that an order must be on a single slab:

$$\sum_{o \in O} x_{os} = 1 \text{ for every slab } s \in S$$

The second one states that a slab must have a single weight:

$$\sum_{q \in Q} y_{sq} = 1 \text{ for every slab } s \in S$$

The third constraint states that the orders must fit within the slab weight:

$$\sum_{o \in O} w_o x_{os} \leq \sum_{q \in Q} q \times y_{qm} \text{ for every slab } s \in S$$

When an order is on a slab, then its color is on a slab

$$x_{os} \leq z_{c_o s} \text{ for } o \in O, s \in M$$

There are orders of at most two different colors on the same slab:

$$\sum_{c \in C} z_{cs} \leq 2 \text{ for every slab } s \in S$$

Finally, the objective function is to minimize the sum of the slab weights:

$$\min \sum_{s \in S, q \in S} q \times y_{sq} \text{ for } s \in S$$

In addition, the authors introduced in [5] a variable for each order whose value is the slab it uses ($Order[o] \in S$). These variables are linked to the binary variables via channeling constraints like $(Order[o] = s) \leftrightarrow (x_{os} = 1)$. The purpose of these variables is to state symmetry-breaking constraints and to define the search strategy. The strategy used is to choose first the order variable $Order[o]$ with minimum domain size and assign it to the slab with the smallest index. Pure constraint programming could solve problems having up to about 20 orders. By hybridizing constraint programming and integer programming, instances with 30 orders could be solved in about 1000s [5].

The basic model defined above is basically a linear model over binary variables. Such a model is well-suited to an integer programming solver. Integer programming solvers can tighten the formulation by adding cutting planes and use the relaxed optimal solution to guide the search. However for constraint programming solvers, this model involves few domain reductions and can be seen as a typical worst case. Moreover, this model involves different groups of binary variables, and it is not easy to determine how to branch on them. Should we start branching with order variables, or with slab weight variables ? Should we merge the groups ? A comparison has been made in [5] but there is no clear winner.

4 A Stronger Constraint Programming Model

The constraint programming model we propose is rather different from the basic one. For clarity, we will consider the minimization of the unused capacity of the selected slab, also called the *loss*. This is equivalent to the objective function of the basic model up to a constant. An obvious lower bound is 0. It is reached when the orders assigned to slabs fit exactly the slab weights. Consequently, it gives a clearer view of solution cost and of the distance to optimality that no longer depends on the order weights.

Example 2. *The solution in Example 1 above induces an total loss of* 2 *because slabs 1 and 3 are not fully filled*

Slab	*Orders*	*Weight sum*	*Slab weight*	*Loss*
1	*1, 2, 3*	*6*	*7*	*1*
2	*4, 5*	*9*	*9*	*0*
3	*6, 7, 8*	*17*	*18*	*1*
4	*9, 10*	*7*	*7*	*0*

To design a stronger constraint programming model we had to *unlinearize* the basic model in order to replace linear constraints by global and logical constraints which achieve more propagation more efficiently.

The first model change is in regards to the variables. Since an order uses a single slab, we can avoid creating a binary variable for each couple (order, slab) and instead introduce a single variable for each order that specifies the slab it uses. That is for each order $o \in O$ a variable x_o whose domain is S is created. The constraint stating that an order uses a single slab becomes implicit.

The variables x_o are the decision variables of the problem; the instantiation of these variables suffices to define completely a solution to the problem and the value of the objective function. Therefore, a good approach is to state all constraints only on those variables or, at most, to introduce auxiliary variables that are all fixed to a value when the decision variables are fixed. This permits the search strategy to be applied to the decision variables only and avoids the need to determine priorities between groups of variables. These priorities are often quite difficult to determine, and this is one of the drawbacks of the basic model.

Packing orders onto slabs can be expressed directly over the variables x_o. A straightforward approach is to constrain, for each slab s, the sum of weights of orders using the slab:

$$\sum_{o \in O} (x_o = s) \times w_o \leq u_k$$

where u_k is the maximum capacity of a slab. However, to compute the loss we need to know the load of each slab. An auxiliary load variable $l_s \in \{0, \ldots, u_k\}$ can be introduced for each slab s and an equivalent constraint is stated instead:

$$l_s = \sum_{o \in O} (x_o = s) \times w_o$$

The upper bound on the l_s variables enforces constraint on the maximum slab weight. Note that variables l_s are all fixed when the variables x_o are all fixed. This formulation could be strengthened by replacing each linear constraint by a knapsack constraint. An arc-consistency algorithm has been given in [10]. Knapsack global constraints are also used in the `Comet` system [11]. An alternative stronger formulation is to replace the whole set of inequalities by the global packing constraint introduced in [9]. It constrains a set of items, given with their sizes, to be packed into a set of bins. The load of each bin is given as a variable. Upper bounds on load variables can be used to model the bin capacity. The packing constraint can be used to strengthen formulations not ony for bin-packing problem, but more generaly on assignment problems where capacity is involved such as warehouse location or resource allocation. The packing constraint achieves more domain reductions than the set of linear constraints above. We have used it to pack orders on slabs. It is stated over variables x_o and l_s and uses order weights w_o as item sizes:

$$\text{pack}([l_1,\ldots,l_n],[x_1,\ldots,x_n],[w_1,\ldots,w_n])$$

A straightforward way to express the loss on each slab is to introduce a variable $y_i \in \{0, u_1, u_2, \ldots, u_k\}$ representing the weight of the slab i. We can then state that the loss on slab s is $\text{loss}_s = (y_s - l_s)$. To achieve more propagation we can observe that the loss on a slab is simply a function of its load. For instance in the Example 1 above for a load of 12, the loss is 5, that is the difference between the load and the smallest slab weight that can contain the orders on that slab. The loss of a slab can thus be defined by

$$\text{loss}(\text{load}) = \min\{u_j \mid j \in \{1,\ldots,k\} \wedge u_j \geq \text{load}\} - \text{load}$$

Therefore the loss of a slab s can be expressed by a constraint in extension (that lists the set of solutions) where the number of solutions is equal to the largest slab weight. Most of the constraint solvers achieve arc-consistency on this constraint. In the Example 1 above the set of solutions is:

load	0	1	2	3	4	**5**	6	**7**	8	**9**	10	**11**	12	13	14	**15**	16	17	**18**
loss	0	4	3	2	1	0	1	0	1	0	1	0	3	2	1	0	2	1	0

A simpler way to express this constraint with the same perfect domain reduction is to use the good old element constraint that indexes the array of precomputed loss with the load variable l_s. This expression is $\text{element}(l_s, \text{loss})$. The objective function is thus

$$\min \sum_{s\in S} \text{element}(l_s,\ \text{loss})$$

Note that when the objective function is to minimize the sum of the slab weights, the loss array is replaced by the array of slab weights indexed by the load.

Now we can express the color constraints without introducing extra variables. First we need an expression that is equal to one when a color k is used on a

```
1.  using CP;

2.  int nbSlabs  = 10;
3.  int nbOrders = 10;
4.  int nbColors = 3;
5.  int nbCap    = 7;
6.  int capacities[1..nbCap] = [0, 5, 7, 9, 11, 15, 18];
7.  int weight[1..nbOrders] = [1, 3, 2, 9, 9, 11, 3, 3, 5, 2];
8.  // White = 0, Black = 1, Red = 2
9.  int colors[1..nbOrders] = [2, 1, 1, 2, 2, 0, 2, 0, 1, 2];

10. int maxCap  = max(i in 1..nbCap) capacities[i];
11. int loss[c in 0..maxCap]
12.       = min(i in 1..nbCap : capacities[i] >= c) capacities[i] - c;

13. dvar int x[1..nbOrders] in 1..nbSlabs;
14. dvar int l[1..nbSlabs] in 0..maxCap;

15. minimize sum(s in 1..nbSlabs) loss[l[s]];
16. subject to {
17.   pack(l, x, weight);
18.   forall(s in 1..nbSlabs)
19.     sum (c in 1..nbColors)
20.         (or(o in 1..nbOrders : colors[o] == c) (x[o] == s)) <= 2;
21. }
```

Fig. 1. An OPL 5.2 model for the steel mill slab design problem

slab s. Such an expression is simply modeled by a disjunction over order position variables x_o whose color c_o is equal to k:

$$\bigvee_{\{o\in O,\ c_o=\ k\}} (x_o = s)$$

The color constraint on a slab is then

$$\sum_{k\in C}\left(\bigvee_{\{o\in O,\ c_o=\ k\}} (x_o = s)\right) \leq 2$$

Figure 1 shows a complete OPL 5.2 model whose objective is to minimize the total loss. ILOG OPL 5.2 includes ILOG CP Optimizer 1.0, which is the ILOG constraint programming C++ library. From lines 2 to 9, the constants and arrays of the instance used in Example 1 are initialized. The `maxCap` constant is set to the maximum capacity of a slab at line 10. The array `loss` initialized at lines 11 and 12 contains the loss induced by each slab load. This is to be used in the objective function. At lines 13 and 14, the variables `x` (that represent order positions) and `l` (the slab load) are created. The cost function is defined at line 15. An element constraint in OPL is created by indexing a constant array with a variable. The pack constraint is stated at line 17. And for each slab, a color constraint is stated at lines 18 to 20.

5 A Search Strategy

The steel mill slab design can be seen as a bin-packing problem with color constraints. The orders must be packed onto slabs. Therefore we have implemented a typical search strategy for solving bin-packing problems. The strategy consists of

- choosing the order with the largest weight first and
- placing that order in the first available slab.

The choice of the variable follows the first-fail principle strategy: largest orders are the most difficult to assign and placing them first reduces the search space. The choice of a value involves grouping orders on a small set of slabs. Avoiding the orders being spread on too many slabs reduces the chances of creating a loss on several slabs and thus avoids producing solutions with a large loss. This strategy has been implemented within a depth-first search.

5.1 Test Instances

The instance provided with the problem definition in the CSPLIB is one with 111 orders. Previous studies of this problem have considered subinstances of this instance by keeping only the k first orders with k varying from 12 to 30. In order to have a precise evaluation of the effectiveness of our choices, we have considered all subinstances from $k = 12$ to $k = 110$ and, of course, the original instance. Every instance has a solution where the total loss is zero.

Our comparisons measure the computation time needed to reach an optimal solution. The experiments were made with ILOG CP Optimizer 1.0, a constraint programming library in C++ and we have run the test with a time limit of 1000 seconds on a PC with a Pentium-4 processor at 2.6 Mhz. The results given by our strategy are presented in Figure 2. The instances index the x-axis while the computation time (in seconds) to reach an optimal solution is on the y-axis. We were able to solve all the instances ranging from 12 to 74 orders in less than 200 seconds. This is much better than all previous constraint programming approaches to this problem.

5.2 Symmetry Breaking

Previous studies on the solution of the steel mill slab design problem insist on the importance of breaking symmetries by adding extra constraints. In [4], two main classes of symmetries are identified:

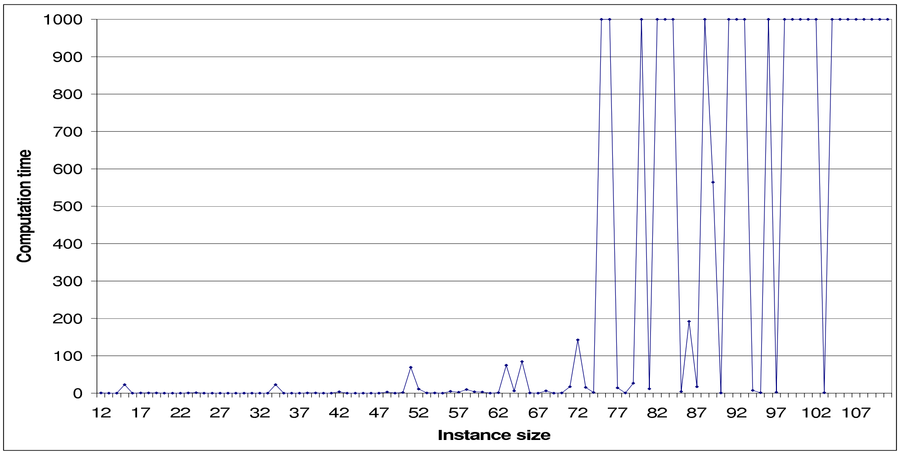

Fig. 2. Depth-first search

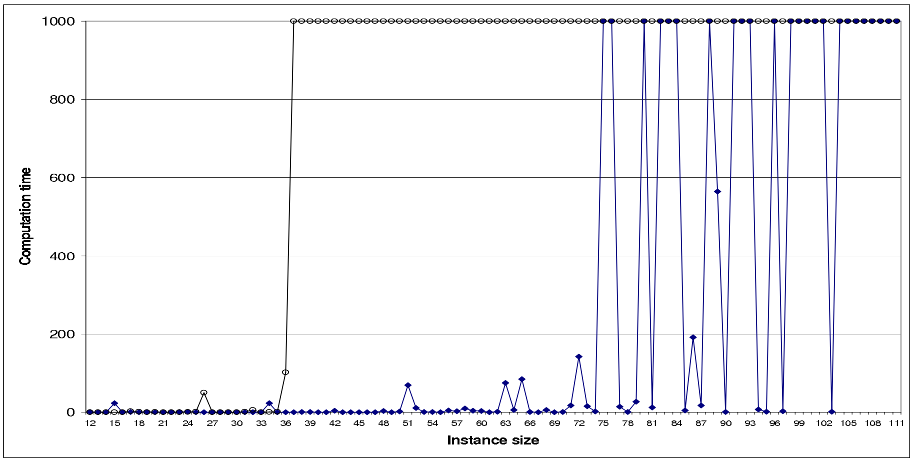

Fig. 3. Symmetry-breaking constraints (circles) versus no symmetry-breaking constraints (diamond-shaped)

1. Slab weight symmetries: slabs weights can be permuted without changing the objective value of the solution.
2. Identical order symmetries: two identical orders (w.r.t. weight and color) on different slabs can be swapped.

To avoid the search strategy producing symmetrical solutions, some additional constraints can be added. The first symmetry can be broken by forcing the slab s to be of greater or equal weight than $s+1$ ($l_s \geq l_{s+1}$). The second symmetry can be broken by adding a constraint for each pair or identical orders i and j.

If $i > j$ the constraint forces the order i to be on a slab whose index is greater or equal than the slab index of order j ($x_i \geq x_j$ for $i, j \in O$ such that $w_i = w_j$ and $c_i = c_j$) .

We have tested adding these symmetry breaking constraints to our constraint programming model. Experiments show that these constraints can be useful for small instances but negatively impact performance on larger instances. The results are presented in Figure 3. For the instance with 37 orders and higher, no optimal solution can be found within the time limit.

The symmetry-breaking constraints prevent our strategy from finding good solutions causing the loss in performance. As the biggest orders are placed first, the first slabs are filled with big orders and are more likely to have a loss (small orders help fill the slab completely). As a consequence, the first slabs are not fully filled, and, since symmetry breaking constraints force the slab loads to decrease, this increases the number of slabs used. The scattering of orders is reinforced by the color constraints. All of this increases the chance of getting a solution with a high loss. Depth-first search makes this even worse as a bad decision made at the beginning of the search to satisfy the symmetry-breaking constraints will be reconsidered only when search has exhausted the whole tree below that bad decision.

We have also observed that in some cases, it even becomes very difficult to find a first solution to the problem. This is because symmetry breaking constraints can create unsatisfiable configurations of slabs. By imposing that the first slabs must have a load greater than or equal to the following ones, it happens that the first slabs become impossible to fill to the required load. Depth-first search may need a considerable enumeration to discover this and to reconsider the bad choices.

For these reasons, we have not used symmetry breaking constraints in our constraint programming solution, and we have dramatically improved the convergence of the search by using a local search approach.

5.3 Large Neighborhood Search

Large neighborhood search (LNS) is a local search technique that improves solutions by solving small decision problems [8].

Assume we want to solve the optimization model

$$\min f(x) \text{ s.t. } M$$

where x are the decision variables, f is the objective function and M is the set of constraints. The LNS method starts from a solution x^* of M whose objective value is f^*. It first chooses a fragment F that is a set of equations $x_i = x_i^*$ and injects it in the model M. That is, it fixes the variables of the fragment to their value in the current solution but keeps the other variables unfixed. Additionally, a constraint is stated on the objective function to force it to be improved. This new model is called a *submodel* since its solution set is included in the solution set of M. A search method is then applied to the submodel

$$R = M \cup F \cup \{f(x) \leq f^* - \epsilon\}$$

Where ϵ is the (positive) optimality tolerance. Consequently when a solution is found in the submodel, it has a necessarily better objective function than the previous one, and it shares with the current solution the values of variables that are in the fragment F. The new solution found becomes the current solution, and the process is repeated.

This method is well suited to optimization in constraint programming because constraint programming is good at finding solutions in small constrained problems.

Example 3. *If we reconsider the data and solution in Example 1, the current solution is* $x_1^* = 1, x_2^* = 1, x_3^* = 1, x_4^* = 2, x_5^* = 2, x_6^* = 3, x_7^* = 3, x_8^* = 3, x_9^* = 4, x_{10}^* = 4$ *and the objective value of this solution is* 2. *Let* M *be the strong constraint programming model of Section 4. A possible fragment for LNS is* $F = \{x_2 = 1, x_3 = 1, x_4 = 2, x_5 = 2, x_6 = 3, x_9 = 4, x_{10} = 4\}$ *where only* x_1 *and* x_8 *are unfixed. The subproblem solved by LNS in this case is*

$$R = M \cup F \cup \{f(x) \leq 1\}$$

The subproblem is solved with a standard constraint programming search that is comprised of depth-first search and constraint propagation. If a good strategy is known for the solving the problem with depth-first search, it is in general also good for solving the subproblem as both have basically the same structure.

Example 4. *The submodel of the previous example contains an optimal solution. Instantiating the variable* x_1 *to* 3 *permits the reduction of the weight of slab* 1 *from* 7 *to* 5*, and its loss becomes 0. The slab 3 now has an order of weight one more, and it fills slab completely. The loss is thus 0. The new solution is*

Slab	*Orders*	*Weight sum*	*Slab weight*	*Loss*
1	*2, 3*	*5*	*5*	*0*
2	*4, 5*	*9*	*9*	*0*
3	*6, 7, 8, 1*	*18*	*18*	*0*
4	*9, 10*	*7*	*7*	*0*

This solution would have been harder to find than the previous one with the full search space. Having a small search space to explore (and of course, the right fragment) makes it easier to find.

In order for the LNS method to explore several neighborhoods and thus several fragments, the submodel is not solved completely. It is crucial to set a limit on the solution method. It can be a limit in time or on the size of the search tree. For instance, when the strategy has encountered a certain number of failures without finding a solution, one can consider that the fragment is not likely to lead to a solution and the search can be stopped. This limit on failures is often used in constraint programming based LNS methods, and the failure limit used is often quite small.

Large neighborhood search has improved dramatically the convergence of our search strategy. For the steel mill slab design problem, we have used the following configuration:

- the fragment size is chosen randomly (between 50% and 95% of the variables are fixed in the fragment);
- variables appearing in the fragment are chosen randomly until the required size is reached;
- load variables l_s are never included in the fragment, the load must not be fixed in order to allow the uninstantiated orders to be assigned to any slab to improve the solution;
- the search strategy for solving the subproblems is the one used for depth-first search (largest orders first, smallest index slab first);
- the failure limit is set to 60.

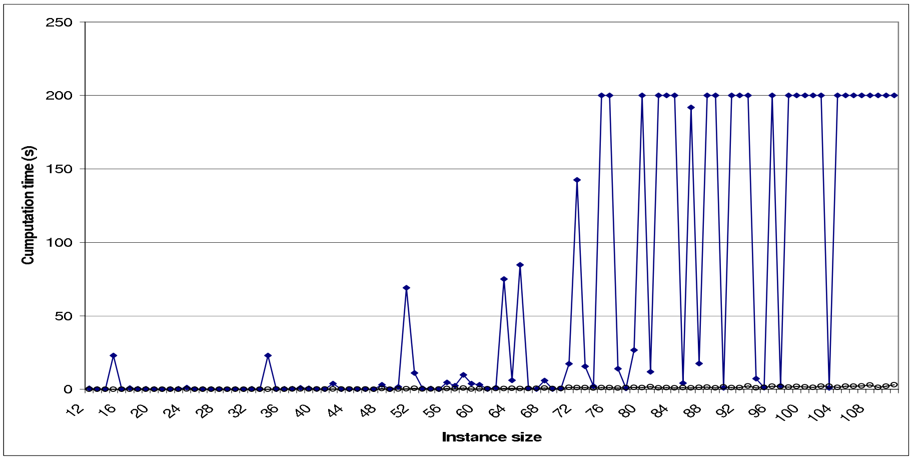

Fig. 4. Large neighborhood search (circles) versus depth-first search (diamond-shaped)

The results are shown in Figure 4. The largest instance with 111 orders is solved in 2.98 seconds and only 27 fragments are explored. All other smaller instances are solved in less than 3 seconds. Interestingly, even with different random seeds, the computation time does not vary much. This demonstrates the robustness of this approach.

6 Conclusion

The constraint programming model we have presented is based on logical and global constraints and is more effective than a linear model over binary variables. A dedicated search strategy used in conjunction with large neighborhood search was able to solve the CSPLIB instance in a few seconds. This demonstrates that a combination of a strong model, a dedicated strategy and randomization can make large neighborhood search effective.

The solutions developed for solving the steel mill slab design can be applied to solving the first-step problem (inventory matching) where orders are not necessarily all packed. Minimizing the weight of unpacked orders is also part of the objective function. The pack global constraint cannot be used any longer, and the strategy would need to be adapted. This is the topic of our current research.

References

1. Dawande, M., Kalagnanam, J., Sethuraman, J.: Variable-sized bin packing with color constraints. Electronic Notes in Discrete Mathematics 7 (2001)
2. Forrest, J., Ladanyi, L., Kalagnanam, J.: A column-generation approach to the multiple knapsack problem with color constraints. INFORMS Journal on Computing 18(1), 129–134 (2006)
3. Frisch, A., Miguel, I., Walsh, T.: Modelling a steel mill slab design problem. In: Proceedings of the IJCAI-01 Workshop on Modelling and Solving Problems with Constraints (2001)
4. Frisch, A., Miguel, I., Walsh, T.: Symmetry and implied constraints in the steel mill slab design problem. In: Walsh, T. (ed.) CP 2001. LNCS, vol. 2239. Springer, Heidelberg (2001)
5. Hnich, B., Kiziltan, Z., Miguel, I., Walsh, T.: Hybrid modelling for robust solving. Annals of Operations Research 130(1-4), 19–39 (2004)
6. Kalagnanam, J., Dawande, M., Trumbo, M., Lee, H.S.: Inventory matching problems in the steel industry. Technical Report RC 21171, IBM Research Report, T.J. Watson Research Center (1988)
7. Prestwich, S.: Search and modelling issues in steel mill slab design. Technical Report 4/148, Cork Constraint Computation Center (2002)
8. Shaw, P.: Using constraint programming and local search methods to solve vehicle routing problems. In: Maher, M.J., Puget, J.-F. (eds.) Principles and Practice of Constraint Programming - CP98. LNCS, vol. 1520, pp. 417–431. Springer, Heidelberg (1998)
9. Shaw, P.: A constraint for bin-packing. In: Wallace, M. (ed.) CP 2004. LNCS, vol. 3258, pp. 648–662. Springer, Heidelberg (2004)
10. Trick, M.: A dynamic programming approach for consistency and propagation for knapsack constraints. In: Proceedings of the Third International Workshop on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR-01), Lille, France, pp. 113–124 (2001)
11. van Hentenryck, P., Michel, L.: Constraint-Based Local Search. MIT Press, Cambridge, Mass. (2005)
12. Walser, J.P.: Solving linear pseudo-boolean constraints with local search. In: Proceedings of the Eleventh Conference on Artificial Intelligence, pp. 269–274 (1997)