IJCAI-05 Modelling Challenge

Paul Shaw **Philippe Laborie**
ILOG S.A., France

1 Introduction

This paper tackles the IJCAI-05 modelling challenge using the constraint programming libraries Solver [ILOG, 2005b] and Scheduler [ILOG, 2005a] from ILOG. The resulting "min-stack" solver uses a combination of modelling, propagation algorithms and search strategies, all of which contribute to increased performance on the challenge instances.

2 Base model

Here, a foundation model is introduced which is sufficient to model the problem. Later, redundant modelling methods are described which do not change the optimal solution of the problem, but can help to solve it more quickly.

In the problem, there are n products and m customers. The set of products demanded by customer j (an order) is denoted by P_j. The set of orders demanding product k is denoted by O_k.

The problem is modelled using a constrained variable for each manufacturing time slot. Variable $p_i \in \{1 \dots n\}$ indicates which product is manufactured at slot i, where slots range from 1 to n. A dual model is also created which represents the time slot of the manufacture of each product. Variable $s_k \in \{1 \dots n\}$ indicates in which slot product k is produced. These two sets of variables are maintained in consistency via an *inverse* constraint which specifies that $s_{p_i} = i$. An *all-different* constraint [Régin, 1994] is also imposed on the variables p which increases domain filtering.

Each order has a *span of activity* between the first and last time slots involving a product demanded by the order. Variables for these first and last slots are defined for order j as $f_j = \min\{s_k | k \in P_j\}$ and $l_j = \max\{s_k | k \in P_j\}$. The constraint $l_j \geq f_j + |P_j| - 1$ is also imposed.

A Boolean variable (which will also be considered to be $0-1$) α_{ij} indicates if order j is *active* at time slot i via the constraint $\alpha_{ij} = (f_j \leq i \wedge l_j \geq i)$. Using this, the number of active orders (number of open stacks) α_i at time slot i is given by $\alpha_i = \sum_{1 \leq j \leq m} \alpha_{ij}$. Finally, the objective is to minimize the maximum number of open stacks (active orders) $\alpha = \max_{1 \leq i \leq n} \alpha_i$.

3 Problem simplifications

Before even attempting to resolve a problem, certain simplifications can be made to it which reduce its difficulty. First of all, any customer orders demanding no products can be removed. Any products which are not ordered by any customer can likewise be removed.

A further observation is that if two products are ordered by exactly the same customers, then one of the products can be removed without affecting the value of the optimal solution. To see this, note that removing a product can never increase the optimum. Also, for any solution not involving the removed product, the removed product can be inserted directly before or after its twin without increasing the number of open stacks.

The final observation is the most powerful, and in practice can aid search considerably. If for two products k and l, product l is ordered by a subset of the customers of k ($O_l \subset O_k$), then product l can be removed from the problem. This can be seen in two ways. First, as above, for any solution without l, l can be inserted next to k without increasing the number of open stacks. Second, O_l could be modified by adding orders (which can never decrease the optimum) to make it equal to O_k, then product l removed using the equality rule.

The product removal rule is effective on many instances. For example, on Simonis's problem_10_20 instances, on average 10 of the initial 20 products are removed.

4 Lower bounds

Computing a lower bound on the minimum number of open stacks can be useful in proving the optimality of a solution; if a solution is found with a number of stacks equal to the lower bound, search can be stopped.

Perhaps the simplest lower bound L_1 is the maximum number of customers demanding a product: $L_1 = \max_{1 \leq k \leq n} |O_k|$. A better lower bound can be found by observing that if any two customer orders involve the same product, the two orders must be active together in at least one slot: that is, the orders overlap in time. An 'order' graph can be constructed with a node per order and an edge between two nodes when their corresponding orders share at least one product. The chromatic number L_3 of this graph forms a lower bound on the minimum number of open stacks.

Instead of colouring the order graph optimally to find bound L_3, a bound L_2 that is cheaper to compute is based on finding a (large) clique in the order graph. The size of this clique is a lower bound on L_3. A greedy clique finding al-

gorithm is used in the solver, and the constraint $\alpha \geq L_2$ is added after finding it.

The bound L_2 is indeed useful for proving optimality. All large instances in gp100by100 were closed because a solution was found with a number of open stacks equal to the size of the maxiumum clique found in the order graph.

5 Redundant modelling

Redundant modelling is the addition of supplementary constraints and variables to the problem, which, although do not reduce the solution space, help the search for solutions by making deductions (filtering domains) more effectively. Here, two redundant models are added to the original: one based on graph colouring, and another based on resource-constrained scheduling.

5.1 Colouring model

The previous section discussed how colouring the order graph could be used to pre-compute a lower bound. A colouring model which bounds the number of open stacks can also increase domain filtering during search. A *colour* variable $c_j \in \{1 \dots m\}$ is introduced for each customer order j. Each pair of orders i, j that overlap must be coloured differently. This is ensured by constraints of the form $f_i \leq l_j \wedge f_j \leq l_i \Rightarrow c_i \neq c_j$. (Note that when orders i and j share a product, both sides of the implication are added directly, as the orders must overlap in this case.) Finally, the constraint that the number of open stacks is at least the number of colours used is imposed: $\alpha \geq \max_{1 \leq j \leq m} c_j$.

The colour variables are symmetric under any value permutation. To break this symmetry, and to colour the maximum number of variables and thus aid domain filtering, a large clique is found (the same one as was used in the bounding function L_2) and coloured with colours from 1 to L_2. This pre-assignment of the colour variables is done at the start of search.

The redundant colouring model is extremely useful. For example, when disabled, one instance in the ShawInstances suite had to be stopped at over three million choice points, when it is normally solved in a few thousand.

5.2 Scheduling model

From a scheduling perspective, each order i can be seen as an activity over the time interval $[f_j, l_j]$ that requires one unit of a discrete capacity resource of total capacity α. The problem is then to find a solution that minimizes the maximal usage of the resource. Classical constraint-based scheduling algorithms available in ILOG Scheduler [ILOG, 2005a] are used to strengthen the propagation.

Let a_j denote the activity representing order j. The following constraints are imposed:

- The activity a_j starts at the first slot of j ($start(a_j) = f_j$) and ends at the last slot ($end(a_j) = l_j + 1$).[1] These constraints allow the base model and the scheduling model to communicate.
- Whenever two orders i and j share at least one product, it means that the two activities a_i and a_j have to overlap for a duration that is at least equal to $o_{ij} = |P_i \cap P_j|$. Thus, the two following temporal constraints can be stated: $end(a_i) \geq start(a_j) + o_{ij}$ and $end(a_j) \geq start(a_i) + o_{ij}$.
- Whenever two orders i and j are such that $P_i \subset P_j$ it means activity a_j *covers* activity a_i. Thus, the two following temporal constraints can be stated: $start(a_i) \geq start(a_j)$ and $end(a_i) \leq end(a_j)$.

On this scheduling model, resource propagation is enforced using the *balance constraint*. This constraint maintains the transitive closure of a precedence graph whose nodes are the start and end time-points of activities and arcs represent precedence relations. The basic idea of the algorithm is to compute, for each activity a_j on the resource, a lower bound on the resource usage at the start time of a_j (symmetrical reasoning can be applied to perform propagation based on a lower bound on the resource usage at the completion time of a_j). Using the precedence graph a lower bound on the resource utilization at date $start(a_j) + \epsilon$ just after the start time of a_j can be computed assuming that all the resource requirements that do not necessarily overlap $start(a_j)$ will not overlap it.

Given this bound, the balance constraint is able to find dead ends, to derive new bounds on activity start and end times, and to find new precedence relations that are added into the precedence graph.

Details of the balance constraint in the more general case of reservoir resources are available in [Laborie, 2003]. What is important is that the balance constraint reasons not only on the time-bounds of activities but also on the precedence relations between activities. It usually allows for a stronger pruning when precedence constraints between activity time-points are fairly dense as is often the case for the challenge problems.

6 Symmetry breaking

Any solution can be mapped into another solution simply by reversing the production sequence $\langle p_1, \dots, p_n \rangle$. In order to break this evident symmetry, a product p_k among the most demanded ones (that is, such that $|O_k|$ is maximal) is selected and constrained to be produced in the first half of the production sequence: $s_k \leq \lfloor (n+1)/2 \rfloor$.

7 Search strategies

The master search used is essentially constructive in nature, although some local search is included (see section 7.3). The standard constraint programming method of finding and proving an optimal solution is used, where the upper bound on the cost function is continually maintained at one less than the cost of the last solution found. Optimality is proven when the complete search tree has been searched. Two search strategies are described here, one quite classic, but with some iteresting optimizations, and one more esoteric.

[1] The additional $+1$ here is because in resource constrained scheduling, activity execution intervals are normally open to the right. That is, an activity of duration 3 starting at time 1 is said to end at time 4. The activity executes in the interval $[1, 4)$.

7.1 A search strategy

A natural way to search for solutions is to build up the schedule chronologically; that is, instantiate the variables $\{p_1, \ldots, p_n\}$ by increasing index. This can be reasonably effective, but a number of peculiarities of the problem allow its efficiency to be significantly increased. These can be demonstrated most easily by a transformation of the problem during search. (This is a descriptive tool; in in reality, no actual transformation takes place).

Consider that during a search, the schedule has been completed from slot 1 to slot h. Let $S_h = \cup_{1 \leq i \leq h} p_i$ be the products already scheduled up to and including slot h. Let $A_h = \{j | f_j \leq h \wedge l_j > h\}$ be the set of active orders *just after* slot h. The remaining sub-problem (to schedule the remainder of the products from slot $h+1$ onwards) can be transformed into an equivalent one by creating a new problem with the remaining products $R_h = \{1, \ldots, n\} - S_h$ and a single new product, say product z_h, with $O_{z_h} = A_h$. In the new transformed sub-problem, product z_h must be scheduled first. This transformation essentially melds all the products already scheduled into one single product representing the active orders just after slot h.

There are two points to note here. First, the transformation makes the sub-problem look like a form of the original problem; the partial assignment of products has been replaced by a single product. Second, the solubility or otherwise of the sub-problem does not depend on the order of product instantiations made up to slot h.

The first of the above points can be exploited by recalling the problem simplifications of section 3. If, for any product k in the new problem, $O_k \subseteq O_{z_h}$, then product k can be inserted next to product z_h. What does this mean for the original search? It means that if after scheduling up to and including slot h, an unscheduled product k exists with $O_k \subseteq A_h$, then product k can be placed in slot $h + 1$ without creating a choice point.[2]

The second point can be exploited by cutting off search when an identical sub-problem has already been encountered [Focacci and Shaw, 2002; Smith, 2005]. For instance, suppose that for a given α, search has proved that there is no feasible extension of the product assignment $\langle 1, 2, 3 \rangle$. Then, by the ordering rule, none exists for any permutation of $\langle 1, 2, 3 \rangle$. Each time the search backtracks, proving a sub-problem insoluble, the set of products scheduled up to that point is recorded in a set of no-goods. Then, at each point in search, the set of currently scheduled products is looked up in the set of no-goods; if it is found then the search is pruned.

These two rules vastly increase the speed of the simple generation scheme. However, another search strategy seems to be more robust in practice.

7.2 A more robust search

In the current implementation, a different strategy is used which was found to be more robust than the more standard left-to-right scheduling already described. The method is based on subdivision of the products to the left and right of the schedule, then solving these two sub-problems recursively. More precisely, to divide the products from slots l to r, a midpoint $m = \lfloor (l + r)/2 \rfloor$ is chosen. Then, for each product k for which $\min(s_k) \leq m \wedge \max(s_k) > m$, a branch is made with the choices $s_k \leq m$ and $s_k > m$. Once all products have been divided, the two sub-problems from slots l to m and slots $m + 1$ to r are solved recursively.

The method works in practice as subdividing the products creates two *independent* sub-problems; no rearragement of the products on the left hand side can affect the solubility or otherwise of the problem on the right. The reasons are similar to those already described for the left-to-right strategy; the two sub-problems can be independently transformed to smaller problems involving the products in their half plus one other product z, with $O_z = A_m$. (Again, these new products need to be placed at one extremity of the schedule of the sub-problem.) Sub-problem independence makes search much more efficient as search can backtrack if any of the two sub-problems is independently insoluble.

Good decisions about how to partition the products can find good solutions more quickly. The approach taken is to place a product in the side which has least products already assigned. To choose *which* product to assign, a calculation is made of the increase to $|A_m|$ that would result in placing each product at the chosen side. The product minimizing this value is placed. On backtracking, the product is placed on the other side.

7.3 Local improvement

Each time a new (better) solution is found, a local search is launched using the new solution as a starting point. The local search technique used is Large Neighborhood Search [Shaw, 1998], which is particularly adapted to constraint programming.

One iteration of LNS undoes the assignments of all variables p_k where $i \leq k \leq j$ and i and j are chosen randomly in $[1 \ldots n]$. LNS then attempts to reassign these variables using a reduced number of open stacks. This reassignment instantiates the product variables by ascending slot and uses a *random* value (*i.e.* product) choice in each slot. The size of this search was limited to 100 backtracks. The choice points explored during LNS are counted just as choice points in complete search, and the sum of all the choice points (from complete search and from LNS) is reported in the final results.

LNS continues accepting improvements until $m + n$ iterations have passed without improving the current solution, at which point search reverts to complete (constructive) search. If a better solution was found during LNS, the new upper bound found is used to constrain the search by reducing the maximum allowed value of α.

LNS proved invaluable for solving larger instance classes, such as gp100by100, and in general helps achieve a better upper bound earlier in the search. However, for most smaller instances, run time increases. LNS is disabled if the problem is thought to be "easy". In the current implementation, this is when $n < 20$.

[2] If more than one such product can be committed, the lowest indexed one should be placed at slot $h + 1$. The remainder of these products will be placed in subsequent slots by re-application of the rule at the next slot.

8 Results

The solver developed closes all problems in the challenge suite except the three largest problems in class sp4. Search effort was measured using run time and number of choice points. For the results in table 2, a time limit of one hour was set. Experiments were run on a Dell D610 Laptop with a 2GHz Pentium-M processor.

9 Comments

The time for this challenge was quite limited, and most of the ideas presented in this paper have not been properly explored. We believe that the results produced here can be greatly improved and there is still much to investigate. For example: deriving finer dominance rules, finding lower bounds *during search* either by colouring or other methods, deriving a good lower bound on A_m while dividing products, applying subproblem simplification in the division strategy, and so on.

References

[Focacci and Shaw, 2002] F. Focacci and P. Shaw. Pruning sub-optimal search branches using local search. In *Proceedings of CP-AI-OR '02*, pages 181–189. Springer, 2002.

[ILOG, 2005a] ILOG. ILOG SCHEDULER 6.1 Reference Manual, 2005. *http://www.ilog.com/*.

[ILOG, 2005b] ILOG. ILOG SOLVER 6.1 Reference Manual, 2005. *http://www.ilog.com/*.

[Laborie, 2003] P. Laborie. Algorithms for propagation resource constraints in AI planning and scheduling: Existing approaches and new results. *Artificial Intelligence*, 143:151–188, 2003.

[Régin, 1994] J.-C. Régin. A filtering algorithm for constraints of difference in CSPs. In *Proceedings of the 12th AAAI*, pages 362–367. AAAI Press / The MIT Press, 1994.

[Shaw, 1998] P. Shaw. Using constraint programming and local search methods to solve vehicle routing problems. In M. Maher and J.-F. Puget, editors, *Fourth International Conference on Principles and Practice of Constraint Programming (CP '98)*, pages 417–431. Springer-Verlag, 1998.

[Smith, 2005] B. M. Smith. Caching search states in permutation problems. To appear in the Proceedings of CP 2005, 2005.

Table 1: Aggregated Result Summary

File	% opt.	Mean Stacks	Total run time (s)			Choice points to optimum			Total choice points		
			Mean	Median	Max	Mean	Median	Max	Mean	Median	Max
ShawInstances	100	13.68	1.33	0.94	4.46	410.24	192.5	3577	1561.08	1187.5	5543
wbo_10_10	100	5.92	0.01	0.01	0.04	45.80	44.5	76	48.92	46	99
wbo_10_20	100	7.35	0.03	0.01	0.22	114.38	101.5	264	156.90	117	644
wbo_10_30	100	8.20	0.20	0.01	1.68	186.95	120	1147	424.50	120.5	2593
wbo_15_15	100	9.35	0.12	0.08	0.44	133.92	90	416	257.60	207.5	842
wbo_15_30	100	11.58	4.76	1.51	22.83	498.73	164	3067	4855.02	1835	18601
wbo_20_10	100	12.90	0.08	0.08	0.22	64.50	67	143	94.07	91.5	189
wbo_20_20	100	13.69	1.77	1.17	10.16	594.71	223.5	7326	1896.44	1377	11046
wbo_30_10	100	20.05	0.33	0.32	0.74	75.42	72	198	125.53	124	271
wbo_30_15	100	20.96	1.74	1.61	5.42	234.73	166.5	904	703.23	669.5	1870
wbo_30_30	100	22.56	97.52	75.82	390.54	5823.58	594	78248	42541.08	36650	208323
wbop_10_10	100	6.75	0.01	0.01	0.02	40.23	25	106	41.67	36.5	107
wbop_10_20	100	8.07	0.07	0.01	0.65	125.42	91	705	250.97	101	1505
wbop_10_30	100	8.55	0.40	0.01	7.53	199.47	119	1158	631.17	119	7629
wbop_15_15	100	10.37	0.12	0.07	0.56	140.25	107.5	631	292.30	225	1088
wbop_15_30	100	12.15	5.46	0.04	42.29	811.68	154	8471	6007.27	155.5	35292
wbop_20_10	100	14.28	0.07	0.04	0.18	58.48	48.5	113	86.62	69	216
wbop_20_20	100	14.87	1.82	0.62	8.58	474.56	203	3778	2174.44	922.5	10161
wbop_30_10	100	22.48	0.23	0.16	0.75	63.90	50.5	203	109.50	93	301
wbop_30_15	100	22.38	2.38	1.70	9.71	311.53	133.5	2886	1020.63	877.5	3942
wbop_30_30	100	23.84	130.69	14.34	969.29	9914.45	303.5	244192	56881.51	8463	475608
wbp_10_10	100	7.28	0.01	0.01	0.01	20.57	17	49	21.98	17	57
wbp_10_20	100	8.71	0.01	0.01	0.09	45.07	37	118	53.41	37	285
wbp_10_30	100	9.31	0.01	0	0.20	54.50	45	391	63.24	49	551
wbp_15_15	100	11.05	0.06	0.02	0.70	78.17	69	340	145.38	80.5	1211
wbp_15_30	100	13.09	0.76	0.02	18.57	209.22	103.5	2258	1281.26	109	25112
wbp_20_10	100	15.12	0.06	0.04	0.19	46.45	44	103	69.60	62	152
wbp_20_20	100	15.41	1.03	0.22	10.61	253.34	121.5	5106	1249.40	230	12572
wbp_30_10	100	23.18	0.28	0.17	0.80	68.60	49.5	188	116.55	99	299
wbp_30_15	100	22.98	2.62	1.42	16.03	283.57	133.5	3938	1086.27	629.5	6273
wbp_30_30	100	24.46	469.57	1.78	6776.43	18021.93	230	822249	180410.06	908.5	3911218
problem_10_10	100	8.03	0.00	0	0.02	18.77	16.5	70	19.26	17	73
problem_10_20	100	8.92	0.00	0	0.05	34.29	29	119	36.63	30	279
problem_15_15	100	12.87	0.02	0.01	0.29	59.93	44.5	297	80.55	49.5	540
problem_15_30	100	14.02	0.29	0.01	13.58	130.53	89	1799	515.85	89	21020
problem_20_10	100	15.88	0.04	0.02	0.23	49.93	46	166	64.41	56	215
problem_20_20	100	17.97	0.58	0.02	16.31	146.85	89	1684	748.96	91	21486
problem_30_10	100	23.95	0.18	0.12	0.91	67.47	57	231	99.99	90.5	301
problem_30_15	100	25.97	1.18	0.10	8.05	194.98	92	2659	606.39	150	3736
problem_30_30	100	28.32	221.65	0.02	4105.60	4079.43	119	175712	89314.88	119	1694494
problem_40_20	100	36.38	21.73	0.67	158.89	855.65	91	13085	5982.75	315	41003

Table 2: Individual Result Summary

Instance	Best obj.	Proven?	Run time	Choice points to optimum	Total choice points
Miller	13	yes	295.45	992	144013
GP1	45	yes	0.61	328	328
GP2	40	yes	1.35	446	446
GP3	40	yes	1.97	475	475
GP4	30	yes	4.07	1153	1153
GP5	95	yes	12.57	1341	1341
GP6	75	yes	96.39	3788	3788
GP7	75	yes	47	2673	2673
GP8	60	yes	90.44	4146	4146
NWRS1	3	yes	0.01	66	66
NWRS2	4	yes	0	46	46
NWRS3	7	yes	0.01	68	68
NWRS4	7	yes	0.02	117	117
NWRS5	12	yes	0.09	168	169
NWRS6	12	yes	0.27	565	565
NWRS7	10	yes	1.1	712	712
NWRS8	16	yes	478.78	1485	302272
SP1	9	yes	4.66	414	1901
SP2	21	no	3600	N/A	956633
SP3	38	no	3600	N/A	157322
SP4	61	no	3600	N/A	19070